Chapter 21

Adaptive Least Squares Approximation

In this chapter we mention some strategies for solving the least squares problem in an adaptive fashion.

21.1 Adaptive Least Squares using Knot Insertion

A classical technique used to improve the quality of a given initial approximation based on a linear combination of certain basis functions is to adaptively increase the number of basis functions used for the fit. In other words, one refines the space from which the approximation is computed. Since every radial basis function is associated with one particular center (or *knot*), this can be achieved by adding new knots to the existing ones. This idea was explored for multiquadrics on $\mathbb{R}^2$ in [Franke *et al.* (1994); Franke *et al.* (1995)], and for radial basis functions on spheres in [Fasshauer (1995a)].

We will now describe an algorithm that adaptively adds knots to a radial basis function approximant in order to improve the ℓ_2 error.

Let us assume we are given a large number, N, of data and we want to fit them with a radial basis expansion to within a given tolerance. The idea is to start with very few initial knots, and then to repeatedly insert a knot at that data location whose ℓ_2 error component is largest. This is done as long as the least squares error exceeds a given tolerance. The following algorithm may be used.

Algorithm 21.1. Knot insertion

(1) Let data sites $\mathcal{X} = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N\}$, data f_i, $i = 1, \ldots, N$, and a tolerance *tol* be given.
(2) Choose M initial knots $\Xi = \{\boldsymbol{\xi}_1, \ldots, \boldsymbol{\xi}_M\}$.
(3) Calculate the least squares fit

$$\mathcal{Q}_f(\boldsymbol{x}) = \sum_{j=1}^{M} c_j \Phi(\boldsymbol{x}, \boldsymbol{\xi}_j)$$

with its associated error

$$e = \sum_{i=1}^{N} [f_i - \mathcal{Q}_f(\boldsymbol{x}_i)]^2.$$

While $e > tol$ **do**

(4) "Weight" each data point $\boldsymbol{x}_i$, $i = 1, \ldots, N$, according to its error component, *i.e.*, let

$$w_i = |f_i - \mathcal{Q}_f(\boldsymbol{x}_i)|, \quad i = 1, \ldots, N.$$

(5) Find the data point $\boldsymbol{x}_\nu \notin \Xi$ with maximum weight w_ν and insert it as a knot, *i.e.*,

$$\Xi = \Xi \cup \{\boldsymbol{x}_\nu\} \quad \text{and} \quad M = M + 1.$$

(6) Recalculate fit and associated error.

A MATLAB implementation of the knot insertion algorithm is provided in `RBFKnotInsert2D.m` (Program 21.1). This program is a little more technical than previous ones since we need to avoid adding the same point multiple times. This would lead to a singular system. In MATLAB we can easily check this with the command `ismember` (see line 28). We also take advantage of the `sort` command to help us find (possibly several) knots with largest error contribution. The addition of these data sites to the set of centers is accomplished on lines 26–32. Evaluation of the approximant (see lines 34–36) is not required until all of the knots have been inserted.

Program 21.1. `RBFKnotInsert2D.m`

```
% RBFKnotInsert2D
% Script that performs 2D RBF least squares approximation
% via knot insertion
% Calls on: DistanceMatrix
 1  rbf = @(e,r) exp(-(e*r).^2); ep = 5.5;
    % Define Franke's function as testfunction
 2  f1 = @(x,y) 0.75*exp(-((9*x-2).^2+(9*y-2).^2)/4);
 3  f2 = @(x,y) 0.75*exp(-((9*x+1).^2/49+(9*y+1).^2/10));
 4  f3 = @(x,y) 0.5*exp(-((9*x-7).^2+(9*y-3).^2)/4);
 5  f4 = @(x,y) 0.2*exp(-((9*x-4).^2+(9*y-7).^2));
 6  testfunction = @(x,y) f1(x,y)+f2(x,y)+f3(x,y)-f4(x,y);
 7  N = 289; gridtype = 'h';
 8  M = 1;  % Number of initial centers
 9  neval = 40;
10  grid = linspace(0,1,neval); [xe,ye] = meshgrid(grid);
11  epoints = [xe(:) ye(:)];
12  tol = 1e-5;  % Tolerance; stopping criterion
```



```
    % Load data points
13  name = sprintf('Data2D_%d%s',N,gridtype); load(name)
    % Take first M "data sites" as centers
14  ctrs = dsites(1:M,:);
    % Compute exact solution, i.e., evaluate test function
    % on evaluation points
15  exact = testfunction(epoints(:,1),epoints(:,2));
    % Create right-hand side vector, i.e.,
    % evaluate the test function at the data points.
16  rhs = testfunction(dsites(:,1),dsites(:,2));
17  rms_res = 999999;
18  while (rms_res > tol)
       % Compute least squares fit
19     DM_data = DistanceMatrix(dsites,ctrs);
20     CM = rbf(ep,DM_data);
21     coef = CM\rhs;
       % Compute residual
22     residual = abs(CM*coef - rhs);
23     [sresidual,idx] = sort(residual);
24     lres = length(residual);
25     rms_res = norm(residual)/sqrt(lres);
       % Add point(s)
26     if (rms_res > tol)
27        addpoint = idx(lres);   % This is the point we add
          % If already used, try next point
28        while any(ismember(ctrs,dsites(addpoint,:),'rows'))
29           lres = lres-1;  addpoint = idx(lres);
30        end
31        ctrs = [ctrs; dsites(addpoint,:)];
32     end
33  end
    % Compute evaluation matrix
34  DM_eval = DistanceMatrix(epoints,ctrs);
35  EM = rbf(ep,DM_eval);
36  Pf = EM*coef;   % Compute RBF least squares approximation
37  maxerr = max(abs(Pf - exact));  rms_err = norm(Pf-exact)/neval;
38  fprintf('RMS error:     %e\n', rms_err)
39  figure; % Plot data sites and centers
40  plot(dsites(:,1),dsites(:,2),'bo',ctrs(:,1),ctrs(:,2),'r+');
41  PlotSurf(xe,ye,Pf,neval,exact,maxerr,[160,20]);
```

We point out that we have to solve one linear least squares problem in each iteration. We do this using the standard MATLAB backslash (or `mldivide`) QR-

based solver (see line 21). The size of these problems increases at each step which means that addition of new knots becomes increasingly more expensive. This is usually not such a big deal. Both [Franke *et al.* (1994); Franke *et al.* (1995)] and [Fasshauer (1995a)] found that the desired accuracy was usually achieved with fairly few additional knots and thus the algorithm is quite fast.

If the initial knots are chosen to lie at data sites (as we did in our MATLAB implementation), then the collocation matrix A in the knot insertion algorithm will always have full rank. This is guaranteed since we only add data sites as new knots, and we make sure in step (5) of the algorithm that no multiple knots are created (which would obviously lead to a rank deficiency).

Instead of deciding which point to add based on residuals one could also pick the new point by looking at the power function, since the dependence of the approximation error on the data sites is encoded in the power function. This strategy is used to build so-called *greedy* adaptive algorithms that *interpolate* successively more and more data (see [Schaback and Wendland (2000a); Schaback and Wendland (2000b)] or Chapter 33). The power function is also employed in [De Marchi *et al.* (2005)] to compute an optimal set of RBF centers independent of the specific data values.

21.2 Adaptive Least Squares using Knot Removal

The idea of knot removal was primarily motivated by the need for data reduction, but it can also be used for the purpose of adaptive approximation (for a survey of knot removal see, *e.g.*, [Lyche (1992)]). The basic idea is to start with a good fit (*e.g.*, an interpolation to the data), and then successively reduce the number of knots used (and therefore basis functions) until a certain given tolerance is reached.

Specifically, this means we will start with an initial fit and then use some kind of weighting strategy for the knots, so that we can repeatedly remove those contributing least to the accuracy of the fit. The following algorithm was suggested in [Fasshauer (1995a)] for adaptive least squares approximation on spheres and performs this task.

Algorithm 21.2. Knot removal

(1) Let data points $\mathcal{X} = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N\}$, data f_i, $i = 1, \ldots, N$, and a tolerance *tol* be given.
(2) Choose M initial knots $\Xi = \{\boldsymbol{\xi}_1, \ldots, \boldsymbol{\xi}_M\}$.
(3) Calculate an initial fit

$$\mathcal{Q}_f(\boldsymbol{x}) = \sum_{j=1}^{M} c_j \Phi(\boldsymbol{x}, \boldsymbol{\xi}_j)$$

with its associated least squares error

$$e = \sum_{i=1}^{N} [f_i - \mathcal{Q}_f(\boldsymbol{x}_i)]^2.$$

While $e < tol$ **do**

(4) "Weight" each knot $\boldsymbol{\xi}_j$, $j = 1, \ldots, M$, according to its least squares error, *i.e.*, form

$$\Xi^* = \Xi \setminus \{\boldsymbol{\xi}_j\},$$

and calculate the weights

$$w_j = \sum_{i=1}^{N} \left[f_i - \mathcal{Q}_f^*(\boldsymbol{x}_i) \right]^2,$$

where

$$\mathcal{Q}_f^*(\boldsymbol{x}) = \sum_{j=1}^{M-1} c_j \Phi(\boldsymbol{x}, \boldsymbol{\xi}_j^*)$$

is the approximation based on the reduced set of knots Ξ^*.

(5) Find the knot $\boldsymbol{\xi}_\mu$ with lowest weight $w_\mu < tol$ and permanently remove it, *i.e.*,

$$\Xi = \Xi \setminus \{\boldsymbol{\xi}_\mu\} \quad \text{and} \quad M = M - 1.$$

(6) Recalculate fit and associated error.

We present a MATLAB implementation of a knot removal algorithm that is slightly more efficient. Its weighting strategy is based on the leave-one-out cross validation algorithm (see [Rippa (1999)] and Chapter 17). The code is given in `RBFKnotRemoval2D.m` (Program 21.2). This program is similar to the knot insertion program. In fact, it is a little simpler since we do not have to worry about multiple knots.

Program 21.2. `RBFKnotRemove2D.m`

```
% RBFKnotRemove2D
% Script that performs 2D RBF least squares approximation
% via knot removal
% Calls on: DistanceMatrix
 1  rbf = @(e,r) exp(-(e*r).^2); ep = 5.5;
    % Define Franke's function as testfunction
 2  f1 = @(x,y) 0.75*exp(-((9*x-2).^2+(9*y-2).^2)/4);
 3  f2 = @(x,y) 0.75*exp(-((9*x+1).^2/49+(9*y+1).^2/10));
 4  f3 = @(x,y) 0.5*exp(-((9*x-7).^2+(9*y-3).^2)/4);
 5  f4 = @(x,y) 0.2*exp(-((9*x-4).^2+(9*y-7).^2));
 6  testfunction = @(x,y) f1(x,y)+f2(x,y)+f3(x,y)-f4(x,y);
```

```
 7  N = 289; gridtype = 'h';
 8  M = 289;  % Number of initial centers
 9  neval = 40;
10  grid = linspace(0,1,neval); [xe,ye] = meshgrid(grid);
11  epoints = [xe(:) ye(:)];
12  tol = 5e-1;  % Tolerance; stopping criterion
    % Load data points
13  name = sprintf('Data2D_%d%s',N,gridtype); load(name)
    % Take first M "data sites" as centers
14  ctrs = dsites(1:M,:);
    % Compute exact solution, i.e., evaluate test function
    % on evaluation points
15  exact = testfunction(epoints(:,1),epoints(:,2));
    % Create right-hand side vector, i.e.,
    % evaluate the test function at the data points.
16  rhs = testfunction(dsites(:,1),dsites(:,2));
17  minres = 0;
18  while (minres < tol)
       % Compute collocation matrix
19     DM_data = DistanceMatrix(dsites,ctrs);
20     CM = rbf(ep,DM_data);
       % Compute residual
21     invCM = pinv(CM);  EF = (invCM*rhs)./diag(invCM);
22     residual = abs(EF);
23     [sresidual,idx] = sort(residual);  minres = residual(1);
       % Remove point
24     if (minres < tol)
25        ctrs = [ctrs(1:idx(1)-1,:); ctrs(idx(1)+1:M,:)];
26        M = M-1;
27     end
28  end
    % Evaluate final least squares fit
29  DM_data = DistanceMatrix(dsites,ctrs);
30  CM = rbf(ep,DM_data);
31  DM_eval = DistanceMatrix(epoints,ctrs);
32  EM = rbf(ep,DM_eval);
33  Pf = EM*(CM\rhs);
34  maxerr = max(abs(Pf - exact));  rms_err = norm(Pf-exact)/neval;
35  fprintf('RMS error:     %e\n', rms_err)
36  figure;   % Plot data sites and centers
37  plot(dsites(:,1),dsites(:,2),'bo',ctrs(:,1),ctrs(:,2),'r+');
38  caption = sprintf('%d data sites and %d centers', N, M);
```



```
39  title(caption);
40  PlotSurf(xe,ye,Pf,neval,exact,maxerr,[160,20]);
```

Again we would like to comment on the algorithm. As far as computational times are concerned, Algorithm 21.2 as listed above is *much* slower than the MATLAB implementation Program 21.2 based on the LOOCV idea since the weight for every knot is determined by the solution of a least squares problem, *i.e.*, in every iteration one needs to solve M least squares problems. The MATLAB program runs considerably faster, but usually it is still slower than the knot insertion algorithm. This is clear since with the knot removal strategy one starts with large problems that get successively smaller, whereas with knot insertion one begins with small problems that can be solved quickly.

The only way the knot removal approach will be beneficial is when the number of evaluations of the constructed approximant is much larger than its actual computation. This is so since, for comparable tolerances, one would expect knot removal to result in fewer knots than knot insertion. However, our examples show that this is not necessarily true.

If the initial knots are chosen at the data sites then, again, there will be no problems with the collocation matrix becoming rank deficient.

In [Fasshauer (1995a); Fasshauer (1995b)] some other alternatives to this knot removal strategy were considered. One of them is the removal of certain groups of knots at one time in order to speed up the process. Another is based on choosing the weights based on the size of the coefficients c_j in the expansion of $\mathcal{Q}_f$, *i.e.*, to remove that knot whose associated coefficient is smallest.

A further variation of the adaptive algorithms was considered in both [Franke *et al.* (1994)] and in [Fasshauer (1995a)]. Instead of treating only the coefficients of the expansion of $\mathcal{Q}_f$ as parameters in the minimization process, one can also include the knot locations themselves and possibly a (variable) shape parameter. This however, leads to *nonlinear* least squares problems. We will not discuss this topic further here.

Buhmann, Derrien, and Le Méhauté [Buhmann *et al.* (1995); Le Méhauté (1995)] also discuss knot removal. Their approach is based on an *a priori estimate* for the error made when removing a certain knot. These estimates depend on the specific choice of radial basis function, and only cover the inverse multiquadric type, *i.e.*,

$$\varphi(r) = (1 + r^2)^{-\beta}, \quad 0 < \beta \leq s/2.$$

Iske (see [Iske (1999a); Iske (2004)]) suggests an alternative knot removal strategy for least squares approximation. His removal heuristics are based on so-called *thinning algorithms*. In particular, in each iteration a point is removed if it belongs to a pair of points in Ξ with minimal separation distance. The thinning phase of the algorithm is then enhanced by an exchange phase in which points can be "swapped back in" if this process reduces the fill-distance of Ξ. This strategy maintains a relatively stable mesh ratio.

21.3 Some Numerical Examples

For the following examples we consider Franke's test function (2.2). All final fits are evaluated on a grid of 40×40 equally spaced points in the unit square. RMS and maximum errors are also computed on this grid. The programs `RBFKnotInsert2D.m` (Program 21.1) and `RBFKnotRemove2D.m` (Program 21.2) were used to compute the results.

In the top and middle parts of Figure 21.1 we compare the knot insertion and knot removal algorithms. In both cases we use a reference set of 289 Halton data sites and Gaussian basic functions scaled with a shape parameter $\varepsilon = 5.5$. Using the knot insertion algorithm with $tol =$ 1e-004 as in Program 21.1 we select a subset of 154 data sites as centers for the basis functions. These points are displayed on the left side of the top part of Figure 21.1 along with the fit on the right. The knot insertion algorithm is initialized with a single random knot in the unit square.

Table 21.1 Total of knots selected, errors and runtimes for adaptive least squares approximants.

Method	N	M	RMS-error	max-error	time
Knot insert	289	154	1.334611e-003	2.871526e-002	2.66
Knot remove	289	153	1.424598e-003	3.961593e-002	35.09
Knot insert	4225	163	1.198695e-004	1.137886e-003	48.75

In the middle part of Figure 21.1 we show the results for the same configuration, *i.e.*, Gaussians with $\varepsilon = 5.5$ and $N = 289$ initial knots, for the knot removal algorithm. This time we take $tol = 0.5$, and the knot removal algorithm begins with an interpolant to all 289 data values. In Table 21.1 we can compare the errors and runtimes for the two algorithms. It is clear that the knot insertion algorithm is much more efficient for this example.

Another advantage of the knot insertion algorithm is revealed in the bottom part of Figure 21.1 and line 3 of Table 21.1. We still use Gaussians with shape parameter $\varepsilon = 5.5$. However, now we take $N = 4225$ Halton points as our data set. This provides many more candidates as centers for basis functions. The chosen centers are displayed on the left of the bottom part of Figure 21.1, and it is clear that this center selection is closely adapted to the features of the data function, namely the peaks and valley. The rest of the knots are located along the boundary of the domain. Moreover, we see that it is possible to obtain a much more accurate fit with roughly the same number of basis functions. While the runtime for this example is considerably longer than that for the other knot insertion example it is comparable to the time required for the knot removal algorithm with only 289 data sites. Running the knot removal algorithm with 4225 data sites would be prohibitively expensive.

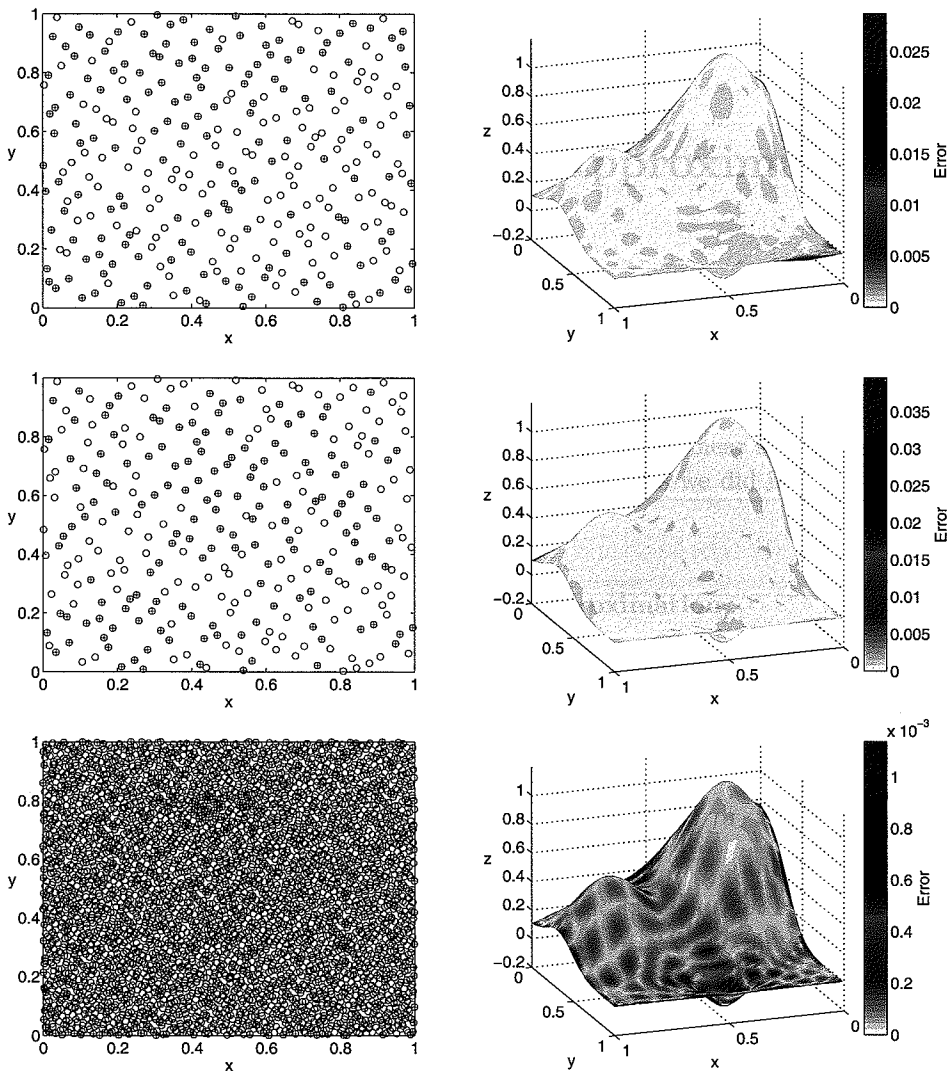

Fig. 21.1 Top: 289 data sites and 154 knots (left) for least squares approximation to Franke's function (right) using knot insertion algorithm with $tol =$ 1e-004. Middle: 289 data sites and 153 knots (left) for least squares approximation to Franke's function (right) using knot removal algorithm with $tol = 0.5$. Bottom: 4225 data sites and 163 knots (left) for least squares approximation to Franke's function (right) using knot insertion algorithm with $tol =$ 1e-004.

Chapter 22

Moving Least Squares Approximation

An alternative to radial basis function interpolation and approximation is the so-called *moving least squares* (MLS) method. As we will see below, in this method the approximation $\mathcal{P}_f$ to f is obtained by solving many (small) linear systems, instead of via solution of a single — but large — linear system as we did in the previous chapters.

22.1 Discrete Weighted Least Squares Approximation

In order to motivate the moving least squares method we begin by discussing discrete weighted least squares approximation from the space of multivariate polynomials. Thus, we consider data $(\boldsymbol{x}_i, f(\boldsymbol{x}_i))$, $i = 1, \ldots, N$, where $\boldsymbol{x}_i \in \Omega \subset \mathbb{R}^s$ and $f(\boldsymbol{x}_i) \in \mathbb{R}$ with arbitrary $s \geq 1$. The approximation space is of the form

$$\mathcal{U} = \text{span}\{p_1, \ldots, p_m\}, \qquad m < N,$$

with multivariate polynomials $p_m \in \Pi_d^s$ of degree at most d.

We intend to find the best discrete weighted least squares approximation from $\mathcal{U}$ to some given function f, *i.e.*, we need to determine the coefficients c_j in

$$u(\boldsymbol{x}) = \sum_{j=1}^{m} c_j p_j(\boldsymbol{x}), \qquad \boldsymbol{x} \in \mathbb{R}^s,$$

such that

$$\|f - u\|_{2,w} \to \min.$$

Here the norm is defined via the discrete (pseudo) inner product

$$\langle f, g \rangle_w = \sum_{i=1}^{N} f(\boldsymbol{x}_i) g(\boldsymbol{x}_i) w(\boldsymbol{x}_i)$$

with scalar weights $w_i = w(\boldsymbol{x}_i)$, $i = 1, \ldots, N$. The induced norm is then of the form

$$\|f\|_{2,w}^2 = \sum_{i=1}^{N} [f(\boldsymbol{x}_i)]^2 \, w(\boldsymbol{x}_i).$$

It is well known that the best approximation u from $\mathcal{U}$ to f is characterized by

$$\begin{aligned} f - u \perp_w \mathcal{U} &\iff \langle f-u, p_k\rangle_w = 0, \quad k = 1,\ldots,m, \\ &\iff \langle f - \sum_{j=1}^{m} c_j p_j, p_k\rangle_w = 0, \quad k = 1,\ldots,m, \\ &\iff \sum_{j=1}^{m} c_j \langle p_j, p_k\rangle_w = \langle f, p_k\rangle_w, \quad k = 1,\ldots,m, \\ &\iff G\boldsymbol{c} = \boldsymbol{f_p}. \end{aligned} \tag{22.1}$$

Here the Gram matrix G has entries $G_{jk} = \langle p_j, p_k\rangle_w$ and the right-hand side vector is $\boldsymbol{f_p} = [\langle f, p_1\rangle_w, \ldots, \langle f, p_m\rangle_w]^T$. We refer to (22.1) as the *normal equations* associated with this problem.

Another way to think of this problem would be as a pure linear algebra problem. To this end, define the $N \times m$ matrix A with entries $A_{ij} = p_j(\boldsymbol{x}_i)$, and the vectors $\boldsymbol{c} = [c_1, \ldots, c_m]^T$ and $\boldsymbol{f} = [f(\boldsymbol{x}_1), \ldots, f(\boldsymbol{x}_N)]^T$. With this notation we seek a solution of the (overdetermined, since $N > m$) linear system $A\boldsymbol{c} = \boldsymbol{f}$. The standard weighted least squares solution is given by the solution of the normal equations $A^T W A\boldsymbol{c} = A^T W \boldsymbol{f}$, where W is the diagonal weighting matrix $W = \mathrm{diag}(w_1, \ldots, w_N)$. This, however, is exactly what is written in (22.1), *i.e.*, the matrix G is of the form $G = A^T W A$, and for the right-hand side vector we have $\boldsymbol{f_p} = A^T W \boldsymbol{f}$.

22.2 Standard Interpretation of MLS Approximation

Several equivalent formulations exist for the moving least squares approximation scheme. In order to make a connection with the discussion of the discrete weighted least squares approximation just presented we start with the standard formulation of MLS approximation. The Backus-Gilbert formulation to be presented in the following section will have a closer connection to previous chapters since it corresponds to a linearly constrained quadratic minimization problem.

The general moving least squares method first appeared in the approximation theory literature in the paper [Lancaster and Šalkauskas (1981)] whose authors also pointed out the connection to the earlier more specialized work [Shepard (1968); McLain (1974)]. We now present a description of MLS approximation that is similar to the discussion in Lancaster and Šalkauskas' original paper and most closely resembles what is found in much of the other literature on MLS approximation.

We consider the following approximation problem. Assume we are given data values $f(\boldsymbol{x}_i)$, $i = 1,\ldots,N$, on some set $\mathcal{X} = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N\} \subset \mathbb{R}^s$ of distinct data sites, where f is some (smooth) function, as well as an approximation space $\mathcal{U} = \mathrm{span}\{u_1, \ldots, u_m\}$ with $m < N$. In addition, we define a weighted ℓ_2 inner product

$$\langle f, g\rangle_{w_{\boldsymbol{y}}} = \sum_{i=1}^{N} f(\boldsymbol{x}_i) g(\boldsymbol{x}_i) w(\boldsymbol{x}_i, \boldsymbol{y}), \qquad \boldsymbol{y} \in \mathbb{R}^s \text{ fixed}, \tag{22.2}$$

where now the weight functions $w_i = w(\boldsymbol{x}_i, \cdot)$, $i = 1,\ldots,N$, vary with the point $\boldsymbol{y}$. Note that the definition of this inner product naturally introduces a second variable, $\boldsymbol{y}$, into the discussion of the problem. This two-variable formulation of MLS approximation will be essential to understanding the connection between the various formulations.

As in the previous sections we wish to find the best approximation u from $\mathcal{U}$ to f. However, we focus our interest on best approximation *at the point* $\boldsymbol{y}$, *i.e.*, with respect to the norm induced by (22.2). In order to keep the discussion as simple as possible we will restrict our discussion to the multivariate polynomial case, *i.e.*, $\mathcal{U} = \Pi_d^s$ with basis $\{p_1, \ldots, p_m\}$. As always, the space Π_d^s of s-variate polynomials of degree d has dimension $m = \binom{s+d}{d}$. We emphasize, however, that everything that is said below also goes through for a more general linear approximation space $\mathcal{U}$.

Since we just introduced the second variable $\boldsymbol{y}$ into our formulation we will now look for the best approximation u in the form

$$u(\boldsymbol{x}, \boldsymbol{y}) = \sum_{j=1}^{m} c_j(\boldsymbol{y}) p_j(\boldsymbol{x} - \boldsymbol{y}), \qquad \boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^s. \tag{22.3}$$

We can think of $\boldsymbol{x}$ as the *global variable* and $\boldsymbol{y}$ as the *local variable*. Thus, expressing the polynomial basis functions in this form is reminiscent of a Taylor expansion. This shift to the local evaluation point $\boldsymbol{y}$ also adds stability to numerical computations.

For the purpose of final evaluation of our approximation we identify the global and the local variable, *i.e.*, we have

$$\mathcal{P}_f(\boldsymbol{x}) = u(\boldsymbol{x}, \boldsymbol{x}) = \sum_{j=1}^{m} c_j(\boldsymbol{x}) p_j(\boldsymbol{0}), \qquad \boldsymbol{x} \in \mathbb{R}^s. \tag{22.4}$$

Since for the polynomial approximation space Π_d^s with standard monomial basis we have $p_1(\boldsymbol{x}) \equiv 1$, and $p_j(\boldsymbol{0}) = 0$ for $j > 1$ we get the standard MLS approximation in the final form

$$\mathcal{P}_f(\boldsymbol{x}) = c_1(\boldsymbol{x}), \qquad \boldsymbol{x} \in \mathbb{R}^s. \tag{22.5}$$

Note, however, that $\boldsymbol{x}$ has been identified with the fixed local point $\boldsymbol{y}$, and therefore in general we still need to recompute the coefficient c_1 every time the evaluation point changes. Examples for some common choices of s and d will be provided in the next chapter.

As in the standard least squares case, the coefficients $c_j(\boldsymbol{y})$ in (22.3) are found by (locally) minimizing the weighted least squares error $\|f - u(\cdot, \boldsymbol{y})\|_{w_{\boldsymbol{y}}}$, *i.e.*,

$$\sum_{i=1}^{N} [f(\boldsymbol{x}_i) - u(\boldsymbol{x}_i, \boldsymbol{y})]^2 \, w(\boldsymbol{x}_i, \boldsymbol{y}) \tag{22.6}$$

is minimized over the coefficients in the expansion (22.3) of $u(\cdot, \boldsymbol{y})$. Note, however, that due to the definition of the inner product (22.2) whose weights "move" with

the local point $\boldsymbol{y}$, the coefficients c_j in (22.3) depend also on $\boldsymbol{y}$. As a consequence one has to solve the *normal equations*

$$\sum_{j=1}^{m} c_j(\boldsymbol{y})\langle p_j(\cdot-\boldsymbol{y}), p_k(\cdot-\boldsymbol{y})\rangle_{w_{\boldsymbol{y}}} = \langle f, p_k(\cdot-\boldsymbol{y})\rangle_{w_{\boldsymbol{y}}}, \qquad k=1,\ldots,m, \tag{22.7}$$

anew each time the point $\boldsymbol{y}$ is changed. In matrix notation (22.7) becomes

$$G(\boldsymbol{y})\boldsymbol{c}(\boldsymbol{y}) = \boldsymbol{f_p}(\boldsymbol{y}). \tag{22.8}$$

Here the positive definite Gram matrix $G(\boldsymbol{y})$ has entries

$$\begin{aligned} G(\boldsymbol{y})_{jk} &= \langle p_j(\cdot-\boldsymbol{y}), p_k(\cdot-\boldsymbol{y})\rangle_{w_{\boldsymbol{y}}} \\ &= \sum_{i=1}^{N} p_j(\boldsymbol{x}_i-\boldsymbol{y})p_k(\boldsymbol{x}_i-\boldsymbol{y})w(\boldsymbol{x}_i,\boldsymbol{y}), \end{aligned} \tag{22.9}$$

and the coefficient vector is of the form $\boldsymbol{c}(\boldsymbol{y}) = [c_1(\boldsymbol{y}),\ldots,c_m(\boldsymbol{y})]^T$. On the right-hand side of (22.8) we have the vector $\boldsymbol{f_p}(\boldsymbol{y}) = [\langle f, p_1(\cdot-\boldsymbol{y})\rangle_{w_{\boldsymbol{y}}},\ldots,\langle f, p_m(\cdot-\boldsymbol{y})\rangle_{w_{\boldsymbol{y}}}]^T$ of projections of the data onto the basis functions.

Several comments are called for. First, to ensure invertibility of the Gram matrix we need to impose a small restriction on the set $\mathcal{X}$ of data sites. Namely, $\mathcal{X}$ needs to be d-unisolvent (*c.f.* Definition 6.1). In this case the Gram matrix is symmetric and positive definite since the polynomial basis is linearly independent and the weights are positive. Second, the fact that the coefficients c_j depend on the point $\boldsymbol{y}$, and thus for every evaluation of $\mathcal{P}_f$ a Gram system (with different matrix $G(\boldsymbol{y})$) needs to be solved, initially scared people away from the moving least squares approach. However, for small values of m, *i.e.*, small polynomial degree d and small space dimensions s, it is possible to solve the Gram system analytically, and thus avoid solving linear systems altogether. We follow this approach and present some examples with explicit formulas in Chapter 23 and use them for our numerical experiments later. Moreover, if one chooses to use compactly supported weight functions, then only a few terms are "active" in the sum defining the entries of $G(\boldsymbol{y})$ (*c.f.* (22.9)).

22.3 The Backus-Gilbert Approach to MLS Approximation

The connection between the standard moving least squares formulation and Backus-Gilbert theory was pointed out in [Bos and Šalkauskas (1989)]. Mathematically, in the Backus-Gilbert approach one considers a *quasi-interpolant* of the form

$$\mathcal{P}_f(\boldsymbol{x}) = \sum_{i=1}^{N} f(\boldsymbol{x}_i)\Psi_i(\boldsymbol{x}), \tag{22.10}$$

where $\boldsymbol{f} = [f(\boldsymbol{x}_1),\ldots,f(\boldsymbol{x}_N)]^T$ represents the given data.

Quasi-interpolation is a generalization of the interpolation idea. If we use a linear function space span$\{\Phi_1,\ldots,\Phi_N\}$ to approximate given data $\{f(\boldsymbol{x}_1),\ldots,f(\boldsymbol{x}_N)\}$, then we saw earlier that we can determine coefficients $c_1,\ldots,c_N$ such that

$$u(\boldsymbol{x}) = \sum_{i=1}^{N} c_i\Phi_i(\boldsymbol{x})$$

interpolates the data, *i.e.*, $u(\boldsymbol{x}_i) = f(\boldsymbol{x}_i)$, $i=1,\ldots,N$. In particular, if the basis functions Φ_i are cardinal functions, *i.e.*, $\Phi_i(\boldsymbol{x}_j) = \delta_{ij}$, $i,j=1,\ldots,N$, then the coefficients are given by the data, *i.e.*, $c_i = f(\boldsymbol{x}_i)$, $i=1,\ldots,N$.

Now, for a general quasi-interpolant we take *generating functions* Ψ_i, $i=1,\ldots,N$ (which can be the same as the basis functions Φ_i) and form the expansion (22.10). This expansion will in general no longer interpolate the data, but it will represent some form of approximation. In order to ensure that a quasi-interpolant achieves a certain approximation power one usually requires that the generating functions reproduce polynomials of a certain degree. The same approach will be followed here, also (cf. (22.13)). The major advantage of quasi-interpolation over interpolation is the fact that we no longer have to solve a (potentially very large) system of linear equations to determine the coefficients c_j. Instead, they are given directly by the data. We will now discuss a scheme that tells us how to choose "good" generating functions Ψ_i.

As before, we consider the more general formulation

$$u(\boldsymbol{x},\boldsymbol{y}) = \sum_{i=1}^{N} f(\boldsymbol{x}_i)\Psi_i(\boldsymbol{x},\boldsymbol{y}),$$

with a global variable $\boldsymbol{x}$ and a local variable $\boldsymbol{y}$. To obtain the Backus-Gilbert approximant we identify $\boldsymbol{x}$ and $\boldsymbol{y}$, *i.e.*,

$$\mathcal{P}_f(\boldsymbol{x}) = u(\boldsymbol{x},\boldsymbol{x}) = \sum_{i=1}^{N} f(\boldsymbol{x}_i)K(\boldsymbol{x}_i,\boldsymbol{x}), \tag{22.11}$$

where we now introduced the notation $K(\boldsymbol{x}_i,\boldsymbol{x}) = \Psi_i(\boldsymbol{x},\boldsymbol{x})$ with a kernel K.

From the discussion above and from Theorem 18.3 we know that the quasi-interpolant that minimizes the point-wise error is given if the generating functions $\Psi_i(\cdot,\boldsymbol{y})$ are cardinal functions (for fixed $\boldsymbol{y}$).

In the Backus-Gilbert formulation of the moving least squares method one does not attempt to minimize the pointwise error, but instead seeks — for a fixed reference point $\boldsymbol{y}$ — to find the *values* of the generating functions $\Psi_i(\boldsymbol{x},\boldsymbol{y})$ at the fixed point $\boldsymbol{x}$ as the minimizers of

$$\frac{1}{2}\sum_{i=1}^{N} \Psi_i^2(\boldsymbol{x},\boldsymbol{y})\frac{1}{w(\boldsymbol{x}_i,\boldsymbol{y})} \tag{22.12}$$

subject to the polynomial reproduction constraints

$$\sum_{i=1}^{N} p(\boldsymbol{x}_i-\boldsymbol{y})\Psi_i(\boldsymbol{x},\boldsymbol{y}) = p(\boldsymbol{x}-\boldsymbol{y}), \quad \text{for all } p\in\Pi_d^s, \tag{22.13}$$

where Π_d^s is the space of s-variate polynomials of total degree at most d with dimension $m = \binom{d+s}{d}$. If we again express the basis polynomials by $p_1, \ldots, p_m$, then we can reformulate (22.13) in matrix-vector form as

$$A(\boldsymbol{y})\Psi(\boldsymbol{x}, \boldsymbol{y}) = \boldsymbol{p}(\boldsymbol{x} - \boldsymbol{y}), \tag{22.14}$$

where $A_{ji}(\boldsymbol{y}) = p_j(\boldsymbol{x}_i - \boldsymbol{y})$, $j = 1, \ldots, m$, $i = 1, \ldots, N$, $\Psi(\boldsymbol{x}, \boldsymbol{y}) = [\Psi_1(\boldsymbol{x}, \boldsymbol{y}), \ldots, \Psi_N(\boldsymbol{x}, \boldsymbol{y})]^T$ is the vector of values of the generating functions, and $\boldsymbol{p} = [p_1, \ldots, p_m]^T$ is the vector of basis polynomials. The corresponding matrix-vector formulation of (22.12) is

$$\frac{1}{2}\Psi^T(\boldsymbol{x}, \boldsymbol{y})Q(\boldsymbol{y})\Psi(\boldsymbol{x}, \boldsymbol{y}), \tag{22.15}$$

where

$$Q(\boldsymbol{y}) = \operatorname{diag}\left(\frac{1}{w(\boldsymbol{x}_1, \boldsymbol{y})}, \ldots, \frac{1}{w(\boldsymbol{x}_N, \boldsymbol{y})}\right), \tag{22.16}$$

and the $w(\boldsymbol{x}_i, \cdot)$ are positive *weight functions* (and thus for any fixed $\boldsymbol{y}$ the matrix $Q(\boldsymbol{y})$ is positive definite).

In the above formulation there is no explicit emphasis on nearness of fit as this is implicitly obtained by the quasi-interpolation "ansatz" and the closeness of the generating functions to the pointwise optimal delta functions. This is achieved by the above problem formulation if the $w(\boldsymbol{x}_i, \cdot)$ are weight functions that decrease with distance from the origin. The strictly positive definite radial functions used earlier are candidates for these weight functions. However, strict positive definiteness is not required at this point, so that, *e.g.*, (radial or tensor product) B-splines can also be used. As mentioned earlier, the polynomial reproduction constraint is added so that the quasi-interpolant will achieve a desired approximation order. This will become clear in Chapter 25.

In pure linear algebra notation the Backus-Gilbert approach corresponds to finding the minimum norm solution of an underdetermined linear system, *i.e.*, we want to solve the polynomial reproduction constraints

$$A(\boldsymbol{y})\Psi(\boldsymbol{x}, \boldsymbol{y}) = \boldsymbol{p}(\boldsymbol{x} - \boldsymbol{y})$$

with $m \times N$ ($m < N$) system matrix. The norm of the solution vector is a weighted norm that varies with the (fixed) reference point $\boldsymbol{y}$ and is measured as in (22.15). In other words, the Backus-Gilbert formulation guarantees that we find the "best" system of generating functions with local polynomial reproduction properties, where "best" is measured in terms of the norm (22.15).

The quadratic form (22.12) (or equivalently (22.15)) can also be interpreted as a smoothness functional. Its use is also motivated by practical applications. In the Backus-Gilbert theory, which was developed in the context of geophysics (see [Backus and Gilbert (1968)]), it is desired that the generating functions Ψ_i are as close as possible to the ideal cardinal functions (*i.e.*, delta functions). Therefore, one needs to minimize their "spread". The polynomial reproduction constraints

are a generalization of an original normalization condition which corresponds to reproduction of constants only.

For any combination of a fixed (evaluation) point $\boldsymbol{x}$ and a fixed (reference) point $\boldsymbol{y}$ the combination of (22.12) and (22.13) (or equivalently (22.15) and (22.14)) present just another constrained quadratic minimization problem of the form discussed in previous chapters.

According to our earlier work we use Lagrange multipliers $\boldsymbol{\lambda}(\boldsymbol{x}, \boldsymbol{y}) = [\lambda_1(\boldsymbol{x}, \boldsymbol{y}), \ldots, \lambda_m(\boldsymbol{x}, \boldsymbol{y})]^T$ (depending on $\boldsymbol{x}$ and $\boldsymbol{y}$), and then know that (*c.f.* (19.4) and (19.5))

$$\boldsymbol{\lambda}(\boldsymbol{x}, \boldsymbol{y}) = \left(A(\boldsymbol{y})Q^{-1}(\boldsymbol{y})A^T(\boldsymbol{y})\right)^{-1}\boldsymbol{p}(\boldsymbol{x} - \boldsymbol{y}) \tag{22.17}$$

$$\Psi(\boldsymbol{x}, \boldsymbol{y}) = Q^{-1}(\boldsymbol{y})A^T(\boldsymbol{y})\boldsymbol{\lambda}(\boldsymbol{x}, \boldsymbol{y}). \tag{22.18}$$

Equation (22.18) tells us how to compute the generating functions for (22.11), *i.e.*, if we write (22.18) componentwise then

$$\Psi_i(\boldsymbol{x}, \boldsymbol{y}) = w(\boldsymbol{x}_i, \boldsymbol{y})\sum_{j=1}^{m}\lambda_j(\boldsymbol{x}, \boldsymbol{y})p_j(\boldsymbol{x}_i - \boldsymbol{y}), \qquad i = 1, \ldots, N. \tag{22.19}$$

Therefore, once the values of the Lagrange multipliers $\lambda_j(\boldsymbol{x}, \boldsymbol{y})$, $j = 1, \ldots, N$, have been determined we have explicit formulas for the values of the generating functions. In particular, we get

$$\begin{aligned}\mathcal{P}_f(\boldsymbol{x}) = u(\boldsymbol{x}, \boldsymbol{x}) &= \sum_{i=1}^{N} f(\boldsymbol{x}_i)\Psi_i(\boldsymbol{x}, \boldsymbol{x}) \\ &= \sum_{i=1}^{N} f(\boldsymbol{x}_i)w(\boldsymbol{x}_i, \boldsymbol{x})\sum_{j=1}^{m}\lambda_j(\boldsymbol{x}, \boldsymbol{x})p_j(\boldsymbol{x}_i - \boldsymbol{x}), \qquad i = 1, \ldots, N, \\ &= \sum_{i=1}^{N} f(\boldsymbol{x}_i)K(\boldsymbol{x}_i, \boldsymbol{x})\end{aligned}$$

with kernels $K(\boldsymbol{x}_i, \boldsymbol{x}) = w(\boldsymbol{x}_i, \boldsymbol{x})\boldsymbol{\lambda}^T(\boldsymbol{x}, \boldsymbol{x})\boldsymbol{p}(\boldsymbol{x}_i - \boldsymbol{x})$.

In general, finding the Lagrange multipliers involves solving a (small) linear system that changes as soon as the reference point $\boldsymbol{y}$ changes (see (22.17)). Using equation (22.17), the Lagrange multipliers are obtained as the solution of a Gram system

$$G(\boldsymbol{y})\boldsymbol{\lambda}(\boldsymbol{x}, \boldsymbol{y}) = \boldsymbol{p}(\boldsymbol{x} - \boldsymbol{y}), \tag{22.20}$$

where the entries of the $m \times m$ matrix $G(\boldsymbol{y})$ are the weighted ℓ_2 inner products

$$G_{jk}(\boldsymbol{y}) = \langle p_j(\cdot - \boldsymbol{y}), p_k(\cdot - \boldsymbol{y})\rangle_{w_{\boldsymbol{y}}} = \sum_{i=1}^{N} p_j(\boldsymbol{x}_i - \boldsymbol{y})p_k(\boldsymbol{x}_i - \boldsymbol{y})w(\boldsymbol{x}_i, \boldsymbol{y}). \tag{22.21}$$

Note that this is identical to the matrix $G(\boldsymbol{y})$ needed for the determination of the coefficients $\boldsymbol{c}(\boldsymbol{y})$ in the standard MLS approach (*c.f.* (22.8) and (22.9)). Equation (22.20) shows us that — for a fixed reference point $\boldsymbol{y}$ — the Lagrange multipliers are polynomials, *i.e.*, $\lambda_j(\cdot, \boldsymbol{y})$, $j = 1, \ldots, m$, are polynomials.

22.4 Equivalence of the Two Formulations of MLS Approximation

We now show that the two formulations of moving least squares approximation described in the previous two sections are equivalent, *i.e.*, we show that $\mathcal{P}_f(\boldsymbol{x})$ computed via (22.4) and (22.11) are the same. The approximant in the standard moving least squares formulation (22.4) establishes $u(\boldsymbol{x},\boldsymbol{y})$ in the form

$$u(\boldsymbol{x},\boldsymbol{y}) = \sum_{j=1}^{m} c_j(\boldsymbol{y}) p_j(\boldsymbol{x}-\boldsymbol{y}) = \boldsymbol{p}^T(\boldsymbol{x}-\boldsymbol{y})\boldsymbol{c}(\boldsymbol{y}),$$

where $\boldsymbol{p} = [p_1,\ldots,p_m]^T$ and $\boldsymbol{c} = [c_1,\ldots,c_m]^T$.

In (22.8) we wrote the normal equations for this approach as

$$G(\boldsymbol{y})\boldsymbol{c}(\boldsymbol{y}) = \boldsymbol{f}_{\boldsymbol{p}}(\boldsymbol{y}).$$

Note that the right-hand side vector $\boldsymbol{f}_{\boldsymbol{p}}(\boldsymbol{y})$ can be written as

$$\begin{aligned}\boldsymbol{f}_{\boldsymbol{p}}(\boldsymbol{y}) &= [\langle f, p_1(\cdot-\boldsymbol{y})\rangle_{w_{\boldsymbol{y}}},\ldots,\langle f, p_m(\cdot-\boldsymbol{y})\rangle_{w_{\boldsymbol{y}}}]^T \\ &= A(\boldsymbol{y})Q^{-1}(\boldsymbol{y})\boldsymbol{f} \end{aligned} \tag{22.22}$$

with the matrix $Q^{-1}(\boldsymbol{y})$ used in the Backus-Gilbert formulation. This implies

$$\boldsymbol{c}(\boldsymbol{y}) = G^{-1}(\boldsymbol{y})A(\boldsymbol{y})Q^{-1}(\boldsymbol{y})\boldsymbol{f}.$$

Thus, using the standard approach, we get

$$u(\boldsymbol{x},\boldsymbol{y}) = \boldsymbol{p}^T(\boldsymbol{x}-\boldsymbol{y})\boldsymbol{c}(\boldsymbol{y}) = \boldsymbol{p}^T(\boldsymbol{x}-\boldsymbol{y})G^{-1}(\boldsymbol{y})A(\boldsymbol{y})Q^{-1}(\boldsymbol{y})\boldsymbol{f}. \tag{22.23}$$

For the Backus-Gilbert approach, on the other hand, the "ansatz" is of the form (22.11)

$$u(\boldsymbol{x},\boldsymbol{y}) = \sum_{i=1}^{N} f(\boldsymbol{x}_i)\Psi_i(\boldsymbol{x},\boldsymbol{y}) = \Psi^T(\boldsymbol{x},\boldsymbol{y})\boldsymbol{f},$$

where as before $\Psi(\boldsymbol{x},\boldsymbol{y}) = [\Psi_1(\boldsymbol{x},\boldsymbol{y}),\ldots,\Psi_N(\boldsymbol{x},\boldsymbol{y})]^T$ and $\boldsymbol{f} = [f(\boldsymbol{x}_1),\ldots,f(\boldsymbol{x}_N)]^T$. For this approach we derived (see (22.17) and (22.18))

$$\begin{aligned}\boldsymbol{\lambda}(\boldsymbol{x},\boldsymbol{y}) &= G^{-1}(\boldsymbol{y})\boldsymbol{p}(\boldsymbol{x}-\boldsymbol{y}) \\ \Psi(\boldsymbol{x},\boldsymbol{y}) &= Q^{-1}(\boldsymbol{y})A^T(\boldsymbol{y})\boldsymbol{\lambda}(\boldsymbol{x},\boldsymbol{y}),\end{aligned}$$

where $G(\boldsymbol{y}) = A(\boldsymbol{y})Q^{-1}(\boldsymbol{y})A^T(\boldsymbol{y})$ (see (22.21) or (22.9)). Therefore, we now obtain for the Backus-Gilbert approximant

$$u(\boldsymbol{x},\boldsymbol{y}) = \Psi^T(\boldsymbol{x},\boldsymbol{y})\boldsymbol{f} = \left[Q^{-1}(\boldsymbol{y})A^T(\boldsymbol{y})G^{-1}(\boldsymbol{y})\boldsymbol{p}(\boldsymbol{x}-\boldsymbol{y})\right]^T\boldsymbol{f}$$

which, by the symmetry of $Q(\boldsymbol{y})$ and $G(\boldsymbol{y})$, is the same as (22.23). Clearly, we also have equivalence when we evaluate either representation at $\boldsymbol{y} = \boldsymbol{x}$, *i.e.*, consider $\mathcal{P}_f(\boldsymbol{x}) = u(\boldsymbol{x},\boldsymbol{x})$.

The equivalence of the two approaches shows that the moving least squares approximant has all of the following properties:

- It reproduces any polynomial of degree at most d in s variables exactly.

- It produces the best locally weighted least squares fit.
- Viewed as a quasi-interpolant, the generating functions Ψ_i are as close as possible to the optimal cardinal basis functions among all functions that produce polynomials of the desired degree in the sense that (22.12) is minimized.
- Since polynomials are infinitely smooth, either of the representations of $\mathcal{P}_f$ shows that its smoothness is determined by the smoothness of the weight function(s) $w_i = w(\boldsymbol{x}_i,\cdot)$.

In particular, the standard moving least squares method will reproduce the polynomial basis functions $p_1,\ldots,p_m$ even though this is not explicitly enforced by the minimization (solution of the normal equations). Moreover, the more general "ansatz" with (non-polynomial) approximation space $\mathcal{U}$ allows us to build moving least squares approximations that also reproduce any other function that is included in $\mathcal{U}$. This can be very beneficial for the solution of partial differential equations with known singularities (see, *e.g.*, the papers [Babuška and Melenk (1997); Belytschko *et al.* (1996)]).

22.5 Duality and Bi-Orthogonal Bases

From the Backus-Gilbert formulation we know that

$$G(\boldsymbol{y})\boldsymbol{\lambda}(\boldsymbol{x},\boldsymbol{y}) = \boldsymbol{p}(\boldsymbol{x}-\boldsymbol{y}) \iff \boldsymbol{\lambda}(\boldsymbol{x},\boldsymbol{y}) = G^{-1}(\boldsymbol{y})\boldsymbol{p}(\boldsymbol{x}-\boldsymbol{y}). \tag{22.24}$$

By taking multiple right-hand sides $\boldsymbol{p}(\boldsymbol{x}-\boldsymbol{y})$ with $\boldsymbol{x}\in\mathcal{X} = \{\boldsymbol{x}_1,\ldots,\boldsymbol{x}_N\}$ we get

$$[\boldsymbol{\lambda}(\boldsymbol{x}_1,\boldsymbol{y}),\ldots,\boldsymbol{\lambda}(\boldsymbol{x}_N,\boldsymbol{y})] = G^{-1}(\boldsymbol{y})[\boldsymbol{p}(\boldsymbol{x}_1-\boldsymbol{y}),\ldots,\boldsymbol{p}(\boldsymbol{x}_N-\boldsymbol{y})]$$

or

$$\Lambda(\boldsymbol{y}) = G^{-1}(\boldsymbol{y})A(\boldsymbol{y}), \tag{22.25}$$

where $A(\boldsymbol{y})$ is the polynomial matrix (22.14) from above and the $m\times N$ matrix $\Lambda(\boldsymbol{y})$ has entries $\Lambda_{ji}(\boldsymbol{y}) = \lambda_j(\boldsymbol{x}_i,\boldsymbol{y})$.

The standard MLS formulation, on the other hand, gives us (see (22.8) and (22.22))

$$G(\boldsymbol{y})\boldsymbol{c}(\boldsymbol{y}) = \boldsymbol{f}_{\boldsymbol{p}}(\boldsymbol{y}) \iff \boldsymbol{c}(\boldsymbol{y}) = G^{-1}(\boldsymbol{y})A(\boldsymbol{y})Q^{-1}(\boldsymbol{y})\boldsymbol{f}. \tag{22.26}$$

By combining (22.25) with (22.26) we get

$$\boldsymbol{c}(\boldsymbol{y}) = G^{-1}(\boldsymbol{y})A(\boldsymbol{y})Q^{-1}(\boldsymbol{y})\boldsymbol{f} = \Lambda(\boldsymbol{y})Q^{-1}(\boldsymbol{y})\boldsymbol{f} = \boldsymbol{f}_{\boldsymbol{\lambda}}(\boldsymbol{y}),$$

where $\boldsymbol{f}_{\boldsymbol{\lambda}}(\boldsymbol{y})$ is defined analogously to $\boldsymbol{f}_{\boldsymbol{p}}(\boldsymbol{y})$ (*c.f.* (22.22)). Componentwise this gives us

$$c_j(\boldsymbol{y}) = \langle f, \lambda_j(\cdot,\boldsymbol{y})\rangle_{w_{\boldsymbol{y}}}, \qquad j = 1,\ldots,m,$$

and therefore,

$$u(\boldsymbol{x},\boldsymbol{y}) = \sum_{j=1}^{m}\langle f, \lambda_j(\cdot,\boldsymbol{y})\rangle_{w_{\boldsymbol{y}}} p_j(\boldsymbol{x}-\boldsymbol{y}). \tag{22.27}$$

It is also possible to formulate the moving least squares method by using the Lagrange multipliers of the Backus-Gilbert approach as basis functions for the approximation space $\mathcal{U}$. Then, using the same argumentation as above, we end up with

$$u(\boldsymbol{x},\boldsymbol{y}) = \sum_{j=1}^{m} d_j(\boldsymbol{y})\lambda_j(\boldsymbol{x},\boldsymbol{y}) \tag{22.28}$$

with

$$d_j(\boldsymbol{y}) = \langle f, p_j(\cdot - \boldsymbol{y})\rangle_{w_{\boldsymbol{y}}}, \qquad j = 1,\ldots,m.$$

We can verify this by applying (22.20) and (22.22) to (22.23), *i.e.*,

$$u(\boldsymbol{x},\boldsymbol{y}) = \boldsymbol{p}^T(\boldsymbol{x}-\boldsymbol{y})G^{-1}(\boldsymbol{y})A(\boldsymbol{y})Q^{-1}(\boldsymbol{y})\boldsymbol{f}.$$

We then obtain

$$u(\boldsymbol{x},\boldsymbol{y}) = \boldsymbol{\lambda}^T(\boldsymbol{x},\boldsymbol{y})\boldsymbol{f}_{\boldsymbol{p}}(\boldsymbol{y}),$$

which corresponds to (22.28).

The calculations just presented show that the Lagrange multipliers form a basis that is *dual* to the polynomial basis. In particular, if we recall the MLS approximant in its standard representation

$$u(\boldsymbol{x},\boldsymbol{y}) = \sum_{j=1}^{m}\langle f, \lambda_j(\cdot,\boldsymbol{y})\rangle_{w_{\boldsymbol{y}}} p_j(\boldsymbol{x}-\boldsymbol{y})$$

and let $f = p_k(\cdot - \boldsymbol{y})$, then the polynomial reproduction property of the method ensures

$$\sum_{j=1}^{m}\langle p_k(\cdot-\boldsymbol{y}), \lambda_j(\cdot,\boldsymbol{y})\rangle_{w_{\boldsymbol{y}}} p_j(\boldsymbol{x}-\boldsymbol{y}) = p_k(\boldsymbol{x}-\boldsymbol{y}).$$

This, however, implies

$$\langle p_k(\cdot-\boldsymbol{y}), \lambda_j(\cdot,\boldsymbol{y})\rangle_{w_{\boldsymbol{y}}} = \delta_{jk}, \qquad j,k = 1,\ldots,m. \tag{22.29}$$

Therefore we have two sets of basis functions that are *bi-orthogonal* with respect to the special weighted inner product $\langle\cdot,\cdot\rangle_{w_{\boldsymbol{y}}}$ on the set $\mathcal{X}$. We will illustrate this duality in Chapter 24.

Earlier we derived the representation (*c.f.* (22.18))

$$\Psi(\boldsymbol{x},\boldsymbol{y}) = Q^{-1}(\boldsymbol{y})A^T(\boldsymbol{y})\boldsymbol{\lambda}(\boldsymbol{x},\boldsymbol{y})$$

for the generating functions in the Backus-Gilbert formulation. Since the Lagrange multipliers are given by $\boldsymbol{\lambda}(\boldsymbol{x},\boldsymbol{y}) = G^{-1}(\boldsymbol{y})\boldsymbol{p}(\boldsymbol{x}-\boldsymbol{y})$ (see (22.20)) we get

$$\Psi(\boldsymbol{x},\boldsymbol{y}) = Q^{-1}(\boldsymbol{y})A^T(\boldsymbol{y})G^{-1}(\boldsymbol{y})\boldsymbol{p}(\boldsymbol{x}-\boldsymbol{y}) = Q^{-1}(\boldsymbol{y})\Lambda^T(\boldsymbol{y})\boldsymbol{p}(\boldsymbol{x}-\boldsymbol{y})$$

due to (22.25). Thus, the Backus-Gilbert representation is given also by the dual representation

$$u(\boldsymbol{x},\boldsymbol{y}) = \sum_{i=1}^{N} f(\boldsymbol{x}_i)w(\boldsymbol{x}_i,\boldsymbol{y})\boldsymbol{\lambda}^T(\boldsymbol{x}_i,\boldsymbol{y})\boldsymbol{p}(\boldsymbol{x}-\boldsymbol{y}). \tag{22.30}$$

By also considering the two dual expansions (22.28) and (22.30) we now have four alternative representations for the moving least squares quasi-interpolant. This is summarized in the following theorem.

Theorem 22.1. *Let $f : \Omega \to \mathbb{R}$ be some function whose values on the d-unisolvent set of points $\mathcal{X} = \{\boldsymbol{x}_i\}_{i=1}^N \subset \mathbb{R}^s$ are given as data. Let $p_1,\ldots,p_m$ be a basis for Π_d^s with $p_1(\boldsymbol{x}) \equiv 1$, let $\{w(\boldsymbol{x}_i,\cdot),\ldots,w(\boldsymbol{x}_N,\cdot)\}$ be a set of positive weight functions, and let λ_j, $j = 1,\ldots,m$, be the Lagrange multipliers defined by (22.17). Furthermore, consider the generating functions*

$$\Psi_i(\boldsymbol{x},\boldsymbol{y}) = w(\boldsymbol{x}_i,\boldsymbol{y})\sum_{j=1}^{m}\lambda_j(\boldsymbol{x},\boldsymbol{y})p_j(\boldsymbol{x}_i-\boldsymbol{y}), \qquad i = 1,\ldots,N,$$

or in dual form

$$\Psi_i(\boldsymbol{x},\boldsymbol{y}) = w(\boldsymbol{x}_i,\boldsymbol{y})\sum_{j=1}^{m}\lambda_j(\boldsymbol{x}_i,\boldsymbol{y})p_j(\boldsymbol{x}-\boldsymbol{y}), \qquad i = 1,\ldots,N.$$

The best local least squares approximation to f on $\mathcal{X}$ in the sense of (22.6) is given by $\mathcal{P}_f(\boldsymbol{x}) = u(\boldsymbol{x},\boldsymbol{x})$, where

$$\begin{aligned} u(\boldsymbol{x},\boldsymbol{y}) &= \sum_{j=1}^{m}\langle f, \lambda_j(\cdot,\boldsymbol{y})\rangle_{w_{\boldsymbol{y}}} p_j(\boldsymbol{x}-\boldsymbol{y}) \\ &= \sum_{j=1}^{m}\langle f, p_j(\cdot-\boldsymbol{y})\rangle_{w_{\boldsymbol{y}}}\lambda_j(\boldsymbol{x},\boldsymbol{y}) \\ &= \sum_{i=1}^{N} f(\boldsymbol{x}_i)\Psi_i(\boldsymbol{x},\boldsymbol{y}). \end{aligned}$$

This results in the four representations

$$\begin{aligned} \mathcal{P}_f(\boldsymbol{x}) &= \sum_{j=1}^{m}\langle f, \lambda_j(\cdot,\boldsymbol{x})\rangle_{w_{\boldsymbol{x}}} p_j(\boldsymbol{0}) = \langle f, \lambda_1(\cdot,\boldsymbol{x})\rangle_{w_{\boldsymbol{x}}} = c_1(\boldsymbol{x}) \\ &= \sum_{j=1}^{m}\langle f, p_j(\cdot-\boldsymbol{x})\rangle_{w_{\boldsymbol{x}}}\lambda_j(\boldsymbol{x},\boldsymbol{x}) \\ &= \sum_{i=1}^{N} f(\boldsymbol{x}_i)w(\boldsymbol{x}_i,\boldsymbol{x})\sum_{j=1}^{m}\lambda_j(\boldsymbol{x},\boldsymbol{x})p_j(\boldsymbol{x}_i-\boldsymbol{x}) \\ &= \sum_{i=1}^{N} f(\boldsymbol{x}_i)w(\boldsymbol{x}_i,\boldsymbol{x})\sum_{j=1}^{m}\lambda_j(\boldsymbol{x}_i,\boldsymbol{x})p_j(\boldsymbol{0}) = \sum_{i=1}^{N} f(\boldsymbol{x}_i)w(\boldsymbol{x}_i,\boldsymbol{x})\lambda_1(\boldsymbol{x}_i,\boldsymbol{x}). \end{aligned}$$

Note that the first two expansions for $\mathcal{P}_f(\boldsymbol{x})$ in Theorem 22.1 can be viewed as generalizations of (finite) eigenfunction or Fourier series expansions.

22.6 Standard MLS Approximation as a Constrained Quadratic Optimization Problem

Finally, it is also possible to formulate the standard MLS approach as a constrained quadratic minimization problem. To this end we (artificially) introduce the quadratic functional

$$\begin{aligned} \boldsymbol{c}^T(\boldsymbol{y})G(\boldsymbol{y})\boldsymbol{c}(\boldsymbol{y}) &= \sum_{j=1}^{m}\sum_{k=1}^{m} c_j(\boldsymbol{y})c_k(\boldsymbol{y})G_{jk}(\boldsymbol{y}) \\ &= \sum_{j=1}^{m}\sum_{k=1}^{m} c_j(\boldsymbol{y})c_k(\boldsymbol{y})\langle p_j(\cdot-\boldsymbol{y}), p_k(\cdot-\boldsymbol{y})\rangle_{w_{\boldsymbol{y}}} \end{aligned}$$

which should be interpreted as the ($\boldsymbol{y}$-dependent) native space norm of the approximant $u(\boldsymbol{x},\boldsymbol{y}) = \sum_{j=1}^{m} c_j(\boldsymbol{y})p_j(\boldsymbol{x}-\boldsymbol{y})$. The Gram system (22.8) can be written in matrix-vector form as

$$G(\boldsymbol{y})\boldsymbol{c}(\boldsymbol{y}) = A(\boldsymbol{y})Q^{-1}(\boldsymbol{y})\boldsymbol{f}$$

where $Q(\boldsymbol{y})$ is the diagonal matrix of weight functions (22.16) and $A(\boldsymbol{y})$ is the matrix of polynomials (22.14) used earlier.

Minimization of the quadratic form subject to the linear side conditions is equivalent to minimization of (for fixed $\boldsymbol{y}$)

$$\frac{1}{2}\boldsymbol{c}^T(\boldsymbol{y})G(\boldsymbol{y})\boldsymbol{c}(\boldsymbol{y}) - \boldsymbol{\mu}^T(\boldsymbol{y})\left[G(\boldsymbol{y})\boldsymbol{c}(\boldsymbol{y}) - A(\boldsymbol{y})Q^{-1}(\boldsymbol{y})\boldsymbol{f}\right], \tag{22.31}$$

where $\boldsymbol{\mu}(\boldsymbol{y})$ is a vector of Lagrange multipliers.

The solution of the linear system resulting from the minimization problem (22.31) gives us

$$\begin{aligned} \boldsymbol{\mu}(\boldsymbol{y}) &= \left(G(\boldsymbol{y})G^{-1}(\boldsymbol{y})G^T(\boldsymbol{y})\right)^{-1} A(\boldsymbol{y})Q^{-1}(\boldsymbol{y})\boldsymbol{f} = G^{-T}(\boldsymbol{y})A(\boldsymbol{y})Q^{-1}(\boldsymbol{y})\boldsymbol{f} \\ \boldsymbol{c}(\boldsymbol{y}) &= G^{-1}(\boldsymbol{y})G^T(\boldsymbol{y})\boldsymbol{\mu}(\boldsymbol{y}) = \boldsymbol{\mu}(\boldsymbol{y}) \end{aligned}$$

so that — as in the case of radial basis function interpolation — the quadratic functional $\boldsymbol{c}^T(\boldsymbol{y})G(\boldsymbol{y})\boldsymbol{c}(\boldsymbol{y})$ is automatically minimized by solving only the Gram system $G(\boldsymbol{y})\boldsymbol{c}(\boldsymbol{y}) = \boldsymbol{f_p}(\boldsymbol{y})$.

22.7 Remarks

In the statistics literature the moving least squares approach is known as *local (polynomial) regression*. Good sources of information are the book [Fan and Gijbels (1996)] and the review article [Cleveland and Loader (1996)] according to which the basic ideas of local regression can be traced back at least to work done in the late 19th century by [Gram (1883); Woolhouse (1870); De Forest (1873); De Forest (1874)]. In particular, in the statistics literature one learns that the use

of least squares approximation is justified when the data $f_1,\ldots,f_N$ are normally distributed, whereas, if the noise in the data is not Gaussian, then other criteria should be used. See, *e.g.*, the survey article [Cleveland and Loader (1996)] for more details.

We close by establishing a connection between the polynomial reproduction constraints and certain moment conditions. Recall the polynomial reproduction constraints (22.14) in the Backus-Gilbert formulation

$$A(\boldsymbol{y})\Psi(\boldsymbol{x},\boldsymbol{y}) = \boldsymbol{p}(\boldsymbol{x}-\boldsymbol{y}).$$

By setting $\boldsymbol{y} = \boldsymbol{x}$ we get

$$A(\boldsymbol{x})\Psi(\boldsymbol{x},\boldsymbol{x}) = \boldsymbol{p}(\boldsymbol{0}). \tag{22.32}$$

Since we defined the kernels $K(\boldsymbol{x}_i,\boldsymbol{x}) = \Psi_i(\boldsymbol{x},\boldsymbol{x})$ earlier, and since we assume that the polynomials basis is such that $p_1(\boldsymbol{x}) \equiv 1$ and $p_j(\boldsymbol{0}) = 0$ for $j > 1$ we get from (22.32)

$$\sum_{i=1}^{N} p_k(\boldsymbol{x}_i - \boldsymbol{x})K(\boldsymbol{x}_i,\boldsymbol{x}) = \delta_{1k}, \qquad k = 1,\ldots,m.$$

This comprises a set of *discrete moment conditions* for the kernel K. Since there are only m conditions for the N kernel values $K(\boldsymbol{x}_i,\boldsymbol{x})$ at a fixed point $\boldsymbol{x}$, the kernel is not uniquely determined by these moment conditions. If, however, we add the least norm constraint from the Backus-Gilbert formulation, *i.e.*, we minimize

$$\frac{1}{2}\sum_{i=1}^{N} K^2(\boldsymbol{x}_i,\boldsymbol{x})\frac{1}{w(\boldsymbol{x}_i,\boldsymbol{x})} = \frac{1}{2}\|K(\cdot,\boldsymbol{x})\|^2_{1/w_{\boldsymbol{x}}},$$

then we get the standard minimum norm solution for underdetermined least squares problems. We also point out that we derived earlier (see Theorem 22.1) that the kernel $K(\cdot,\boldsymbol{x})$ is of the form

$$K(\cdot,\boldsymbol{x}) = \lambda_1(\cdot,\boldsymbol{x})w(\cdot,\boldsymbol{x})$$

with polynomial term $\lambda_1(\cdot,\boldsymbol{x})$ and weight function $w(\cdot,\boldsymbol{x})$. Thus, we have a unique solution once the weight function $w(\cdot,\boldsymbol{x})$ has been chosen.

The interpretation of MLS approximation with the help of moment matrices is used in the engineering literature (see, *e.g.*, [Li and Liu (2002)]), and also plays an essential role when connecting moving least squares approximation to the more efficient concept of *approximate approximation* [Maz'ya and Schmidt (2001)]. For a discussion of approximate moving least squares approximation see [Fasshauer (2002c); Fasshauer (2002d); Fasshauer (2003); Fasshauer (2004)] or Chapter 26.

Chapter 23

Examples of MLS Generating Functions

23.1 Shepard's Method

The moving least squares method in the case $d = 0$ (and therefore $m = 1$) with $p_1(\boldsymbol{x}) \equiv 1$ is usually referred to as *Shepard's method* [Shepard (1968)]. In the statistics literature Shepard's method is also known as a *kernel method* (see, *e.g.*, the papers from the 1950s and 60s [Rosenblatt (1956); Parzen (1962); Nadaraya (1964); Watson (1964)]). Using our notation we have

$$\mathcal{P}_f(\boldsymbol{x}) = c_1(\boldsymbol{x}).$$

The Gram "matrix" (22.9) consists of only one element

$$G(\boldsymbol{x}) = \langle p_1(\cdot - \boldsymbol{x}), p_1(\cdot - \boldsymbol{x}) \rangle_{w_{\boldsymbol{x}}} = \sum_{i=1}^{N} w(\boldsymbol{x}_i, \boldsymbol{x})$$

so that

$$G(\boldsymbol{x})\boldsymbol{c}(\boldsymbol{x}) = \boldsymbol{f_p}(\boldsymbol{x})$$

implies

$$c_1(\boldsymbol{x}) = \frac{\sum_{i=1}^{N} f(\boldsymbol{x}_i) w(\boldsymbol{x}_i, \boldsymbol{x})}{\sum_{j=1}^{N} w(\boldsymbol{x}_j, \boldsymbol{x})}. \tag{23.1}$$

If we compare (23.1) with the Backus-Gilbert quasi-interpolation formulation (22.11) we immediately see that the generating functions Ψ_i are given by

$$\Psi_i(\boldsymbol{x}, \boldsymbol{x}) = \frac{w(\boldsymbol{x}_i, \boldsymbol{x})}{\sum_{j=1}^{N} w(\boldsymbol{x}_j, \boldsymbol{x})}. \tag{23.2}$$

Since

$$\sum_{i=1}^{N} \Psi_i(\boldsymbol{x}, \boldsymbol{x}) = \sum_{i=1}^{N} \frac{w(\boldsymbol{x}_i, \boldsymbol{x})}{\sum_{j=1}^{N} w(\boldsymbol{x}_j, \boldsymbol{x})} \equiv 1 \tag{23.3}$$

independent of $\boldsymbol{x}$, Shepard's method is also known as a *partition of unity* method.

The weight functions can take many forms. In practice one usually takes a single basic weight function w which is then shifted to the data sites. Often the basic weight function is also a radial function (in either the Euclidean or maximum norm). We will use radial functions as basic weights in our numerical experiments below, *i.e.*,

$$w(\boldsymbol{x}_i, \boldsymbol{x}) = w(\|\boldsymbol{x} - \boldsymbol{x}_i\|)$$

where w is one of our (strictly positive definite) radial basic functions. Both compactly supported and globally supported functions will be used.

The dual Shepard basis is defined by (see (22.24))

$$G(\boldsymbol{y})\boldsymbol{\lambda}(\boldsymbol{x}, \boldsymbol{y}) = \boldsymbol{p}(\boldsymbol{x} - \boldsymbol{y})$$

so that

$$\lambda_1(\boldsymbol{x}, \boldsymbol{x}) = \frac{1}{\sum_{i=1}^{N} w(\boldsymbol{x}_i, \boldsymbol{x})},$$

and (*c.f.* (22.28))

$$\mathcal{P}_f(\boldsymbol{x}) = d_1(\boldsymbol{x})\lambda_1(\boldsymbol{x}, \boldsymbol{x}) \tag{23.4}$$

with

$$d_1(\boldsymbol{x}) = \langle f, p_1(\cdot - \boldsymbol{x})\rangle_{w_{\boldsymbol{x}}} = \sum_{i=1}^{N} f(\boldsymbol{x}_i) w(\boldsymbol{x}_i, \boldsymbol{x}).$$

The explicit dual representation of the Shepard approximant is, of course, identical to (23.1).

The generating functions (22.19) are defined as

$$\Psi_i(\boldsymbol{x}, \boldsymbol{x}) = w(\boldsymbol{x}_i, \boldsymbol{x})\lambda_1(\boldsymbol{x}, \boldsymbol{x}) p_1(\boldsymbol{x}_i - \boldsymbol{x}) = \frac{w(\boldsymbol{x}_i, \boldsymbol{x})}{\sum_{j=1}^{N} w(\boldsymbol{x}_j, \boldsymbol{x})},$$

which matches our earlier expression (23.2). This once more gives rise to the well-known quasi-interpolation formula for Shepard's method

$$\begin{aligned}\mathcal{P}_f(\boldsymbol{x}) &= \sum_{i=1}^{N} f(\boldsymbol{x}_i)\Psi_i(\boldsymbol{x}, \boldsymbol{x}) \\ &= \sum_{i=1}^{N} f(\boldsymbol{x}_i) \frac{w(\boldsymbol{x}_i, \boldsymbol{x})}{\sum_{j=1}^{N} w(\boldsymbol{x}_j, \boldsymbol{x})}.\end{aligned}$$

We now also have bi-orthogonality of the basis and dual basis, *i.e.*,

$$\langle \lambda_1(\cdot, \boldsymbol{x}), p_1(\cdot - \boldsymbol{x})\rangle_{w_{\boldsymbol{x}}} = 1.$$

Indeed

$$\begin{aligned}\langle \lambda_1(\cdot, \boldsymbol{x}), p_1(\cdot - \boldsymbol{x})\rangle_{w_{\boldsymbol{x}}} &= \sum_{i=1}^{N} \lambda_1(\boldsymbol{x}_i, \boldsymbol{x}) w(\boldsymbol{x}_i, \boldsymbol{x}) \\ &= \sum_{i=1}^{N} \frac{w(\boldsymbol{x}_i, \boldsymbol{x})}{\sum_{j=1}^{N} w(\boldsymbol{x}_j, \boldsymbol{x})} = 1.\end{aligned}$$

23.2 MLS Approximation with Nontrivial Polynomial Reproduction

While it is always possible to simply solve the local Gram systems (22.8) or (22.20) and therefore implicitly compute the generating functions Ψ_i, $i = 1, \ldots, N$, it is often of interest to have explicit formulas for Ψ_i. We saw in the previous section that in the case of reproduction of constants we arrive at Shepard's method which is valid independent of the space dimension. However, if the degree d of polynomial reproduction is nontrivial (*i.e.*, $d > 0$), then the size $m = \binom{d+s}{d}$ of the Gram systems, and therefore the resulting formulas for the generating functions will depend on the space dimension s.

We now present three examples with explicit formulas for the MLS generating functions.

To simplify the notation in the following examples we define the *moments*

$$\mu_{\boldsymbol{\alpha}} = \sum_{i=1}^{N} (\boldsymbol{x}_i - \boldsymbol{x})^{\boldsymbol{\alpha}} w(\boldsymbol{x}_i, \boldsymbol{x}), \qquad \boldsymbol{x} \in \mathbb{R}^s, \quad |\boldsymbol{\alpha}| \le 2d,$$

with $\boldsymbol{\alpha}$ a multi-index. These moments arise as entries in the Gram matrix $G(\boldsymbol{y})$ if we use a monomial basis and identify the reference point $\boldsymbol{y}$ with the evaluation point $\boldsymbol{x}$.

Example 23.1. We take $s = 1$, $d = 1$, and therefore $m = 2$. The set of data sites is given by $\mathcal{X} = \{x_1, \ldots, x_N\}$. We choose the standard monomial basis

$$\mathcal{U} = \text{span}\{p_1(x) = 1,\ p_2(x) = x\}.$$

Then the Gram matrix (22.20) is of the form

$$\begin{aligned}G(x) &= \begin{bmatrix} \langle p_1(\cdot - x), p_1(\cdot - x)\rangle_{w_x} & \langle p_1(\cdot - x), p_2(\cdot - x)\rangle_{w_x} \\ \langle p_2(\cdot - x), p_1(\cdot - x)\rangle_{w_x} & \langle p_2(\cdot - x), p_2(\cdot - x)\rangle_{w_x} \end{bmatrix} \\ &= \begin{bmatrix} \sum_{i=1}^{N} w(x_i, x) & \sum_{i=1}^{N} (x_i - x) w(x_i, x) \\ \sum_{i=1}^{N} (x_i - x) w(x_i, x) & \sum_{i=1}^{N} (x_i - x)^2 w(x_i, x) \end{bmatrix} \\ &= \begin{bmatrix} \mu_0 & \mu_1 \\ \mu_1 & \mu_2 \end{bmatrix},\end{aligned}$$

and the right-hand side of the Gram system is given by

$$\boldsymbol{p}(x-x) = \begin{bmatrix} p_1(0) \\ p_2(0) \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}.$$

Therefore, solution of the Gram system via Cramer's rule immediately yields

$$\lambda_1(x,x) = \frac{\mu_2}{\mu_0\mu_2 - \mu_1^2},$$
$$\lambda_2(x,x) = \frac{\mu_1}{\mu_1^2 - \mu_0\mu_2},$$

and according to (22.19) the generating functions for the Backus-Gilbert quasi-interpolant (MLS approximant) (22.11) are given by

$$\Psi_i(x,x) = w(x_i,x)\left[\lambda_1(x,x) + \lambda_2(x,x)(x_i - x)\right], \qquad i = 1,\ldots,N,$$

where the $w(x_i,\cdot)$ are arbitrary (positive) weight functions.

Example 23.2. We remain in the univariate case ($s = 1$) but increase the degree of polynomial reproduction to $d = 2$ so that $m = 3$. Again we take the standard monomial basis, *i.e.*,

$$\mathcal{U} = \text{span}\{p_1(x) = 1,\ p_2(x) = x,\ p_3(x) = x^2\}.$$

The 3×3 Gram system

$$\begin{bmatrix} \mu_0 & \mu_1 & \mu_2 \\ \mu_1 & \mu_2 & \mu_3 \\ \mu_2 & \mu_3 & \mu_4 \end{bmatrix} \begin{bmatrix} \lambda_1(x,x) \\ \lambda_2(x,x) \\ \lambda_3(x,x) \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

can then be solved analytically (*e.g.*, using Maple), and one obtains the Lagrange multipliers

$$\lambda_1(x,x) = \frac{\mu_2\mu_4 - \mu_3^2}{D},$$
$$\lambda_2(x,x) = \frac{\mu_2\mu_3 - \mu_1\mu_4}{D},$$
$$\lambda_3(x,x) = \frac{\mu_1\mu_3 - \mu_2^2}{D},$$

where $D = 2\mu_1\mu_2\mu_3 - \mu_0\mu_3^2 - \mu_2^3 - \mu_1^2\mu_4 + \mu_0\mu_2\mu_4$. Now the generating functions are given by

$$\Psi_i(x,x) = w(x_i,x)\left[\lambda_1(x,x) + \lambda_2(x,x)(x_i - x) + \lambda_3(x,x)(x_i - x)^2\right], \quad i = 1,\ldots,N.$$

Example 23.3. Reproduction of linear polynomials in 2D also leads to a 3×3 Gram system (since $s = 2$, $d = 1$, and therefore $m = 3$). Now the monomial basis is given by

$$\mathcal{U} = \text{span}\{p_1(x,y) = 1,\ p_2(x,y) = x,\ p_3(x,y) = y\},$$

where $\boldsymbol{x} = (x,y) \in \mathbb{R}^2$, and the Gram system looks like

$$\begin{bmatrix} \mu_{00} & \mu_{10} & \mu_{01} \\ \mu_{10} & \mu_{20} & \mu_{11} \\ \mu_{01} & \mu_{11} & \mu_{02} \end{bmatrix} \begin{bmatrix} \lambda_1(\boldsymbol{x},\boldsymbol{x}) \\ \lambda_2(\boldsymbol{x},\boldsymbol{x}) \\ \lambda_3(\boldsymbol{x},\boldsymbol{x}) \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}.$$

The Lagrange multipliers in this case turn out to be

$$\lambda_1(\boldsymbol{x},\boldsymbol{x}) = \frac{1}{D}\left[\mu_{11}^2 - \mu_{20}\mu_{02}\right],$$

$$\lambda_2(\boldsymbol{x},\boldsymbol{x}) = \frac{1}{D}\left[\mu_{10}\mu_{02} - \mu_{01}\mu_{11}\right],$$

$$\lambda_3(\boldsymbol{x},\boldsymbol{x}) = \frac{1}{D}\left[\mu_{20}\mu_{01} - \mu_{10}\mu_{11}\right],$$

with $D = \mu_{10}^2\mu_{02} + \mu_{20}\mu_{01}^2 - \mu_{00}\mu_{20}\mu_{02} - 2\mu_{10}\mu_{01}\mu_{11} + \mu_{00}\mu_{11}^2$. The generating functions Ψ_i, $i = 1,\ldots,N$, for this example are of the form

$$\Psi_i(\boldsymbol{x},\boldsymbol{x}) = w(\boldsymbol{x}_i,\boldsymbol{x})\left[\lambda_1(\boldsymbol{x},\boldsymbol{x}) + \lambda_2(\boldsymbol{x},\boldsymbol{x})(x_i - x) + \lambda_3(\boldsymbol{x},\boldsymbol{x})(y_i - y)\right].$$

While we can use a symbolic manipulation program such as Maple to solve the Gram system analytically for other choices of d and s, it is clear that the expressions for the generating functions quickly become very unwieldy. It may be tolerable to continue this approach for a 4×4 Gram system corresponding to reproduction of linear polynomials in 3D (or cubic polynomials in 1D), but already the case of quadratic reproduction in 2D (with $m = 6$) is much too complex to print here, and reproduction of quadratics in 3D (or cubics in 2D) requires even 10 Lagrange multipliers.

Chapter 24

MLS Approximation with MATLAB

24.1 Approximation with Shepard's Method

In this section we investigate approximation with Shepard's method

$$\mathcal{P}_f(\boldsymbol{x}) = \sum_{i=1}^{N} f(\boldsymbol{x}_i) \frac{w(\boldsymbol{x}_i, \boldsymbol{x})}{\sum_{j=1}^{N} w(\boldsymbol{x}_j, \boldsymbol{x})}, \qquad \boldsymbol{x} \in \mathbb{R}^s.$$

We will look at three sets of experiments. The first two sets will employ global Gaussian weights, *i.e.*, $w(\boldsymbol{x}_i, \boldsymbol{x}) = e^{-\varepsilon^2 \|\boldsymbol{x}-\boldsymbol{x}_i\|^2}$, in $\mathbb{R}^2$, while the third set of experiments is based on Wendland's compactly supported C^2 weights $w(\boldsymbol{x}_i, \boldsymbol{x}) = (1-\varepsilon\|\boldsymbol{x}-\boldsymbol{x}_i\|)_+^4 (4\varepsilon\|\boldsymbol{x}-\boldsymbol{x}_i\|+1)$ in $\mathbb{R}^s$ for $s = 1, 2, \ldots, 6$. Note that the Wendland weights are strictly positive definite basic functions only for $s \leq 3$ so that it would not be advisable to use them for RBF interpolation in higher dimensions. For MLS approximation, however, strict positive definiteness is not required. Here we only ask that the weights be positive on their support.

In our 2D experiments we take Franke's function (2.2) as our test function and sample it at uniformly spaced points in the unit square. In Table 24.1 we list the RMS-errors and computed convergence rates for both stationary and non-stationary approximation. Clearly, non-stationary approximation does not converge. This behavior is *exactly opposite* the convergence behavior of Gaussians in the RBF interpolation setting. There we have convergence in the non-stationary setting, but not in the stationary setting (*c.f.* Table 17.5 and Figure 17.12). The initial shape parameter for the stationary setting (and the fixed value in the non-stationary setting) is $\varepsilon = 3$. This value is subsequently multiplied by a factor of two for each halving of the fill-distance. We can see that Shepard's method seems to have a stationary approximation order of at least $\mathcal{O}(h)$. This will be verified theoretically in the next chapter.

The first two stationary Shepard approximations (based on 9 and 25 points, respectively) are displayed in the top part of Figure 24.1. It is apparent that the Shepard method has a smoothing effect. In order to emphasize this feature we provide Shepard approximations to noisy data (with 3% noise added to the

Table 24.1 2D stationary and non-stationary Shepard approximation with Gaussian weight function.

	stationary		non-stationary	
N	RMS-error	rate	RMS-error	rate
9	1.835110e-001		1.835110e-001	
25	5.885159e-002	1.6407	1.303771e-001	0.4932
81	2.299502e-002	1.3558	1.311538e-001	−0.0086
289	6.726166e-003	1.7735	1.315894e-001	−0.0048
1089	2.113604e-003	1.6701	1.320564e-001	−0.0051
4225	8.065893e-004	1.3898	1.323576e-001	−0.0033

Franke data) in the bottom part of Figure 24.1. On the left we display the Shepard approximation based on Gaussian weights with $\varepsilon = 48$ (corresponding to the entry in line 5 of Table 24.1). The RMS-error for this approximation is 1.820897e-002. The plot on the right of the bottom part of Figure 24.1 is the result of reducing ε to 12, and thus increasing the smoothing effect since "wider" weight functions are used, *i.e.*, more neighboring data are incorporated into the local regression fit. The corresponding RMS-error is 2.481218e-002. Note that even though the RMS-error is larger for the plot on the right, the surface is visually smoother. These examples should be compared to the data smoothing experiments in Chapter 19.

The MATLAB code for the 2D experiments is `Shepard2D.m` listed as Program 24.1. It is a little simpler than the RBF interpolation code used earlier since we do not need to assemble an interpolation matrix and no linear system needs to be solved. The evaluation matrix is assembled in two steps. First, on line 14, we compute the standard RBF evaluation matrix. In order to implement the Shepard scaling we note that all of the entries in row i of the evaluation matrix are scaled with the same sum, $\sum_{j=1}^{N} w(\boldsymbol{x}_j, \boldsymbol{x}_i)$. Therefore, on line 16, we perform the extra Shepard scaling with the help of `repmat` and a vector of `ones` which gives us all the necessary sums in the denominator. For the example with noisy data we add

```
f = f + 0.03*randn(size(f));
```

after line 11.

Program 24.1. `Shepard2D.m`

```
% Shepard2D
% Script that performs 2D Shepard approximation with global weights
% Calls on: DistanceMatrix, PlotSurf, PlotError2D
 1  rbf = @(e,r) exp(-(e*r).^2); ep = 5.5;
    % Define Franke's function as testfunction
 2  f1 = @(x,y) 0.75*exp(-((9*x-2).^2+(9*y-2).^2)/4);
 3  f2 = @(x,y) 0.75*exp(-((9*x+1).^2/49+(9*y+1).^2/10));
 4  f3 = @(x,y) 0.5*exp(-((9*x-7).^2+(9*y-3).^2)/4);
```


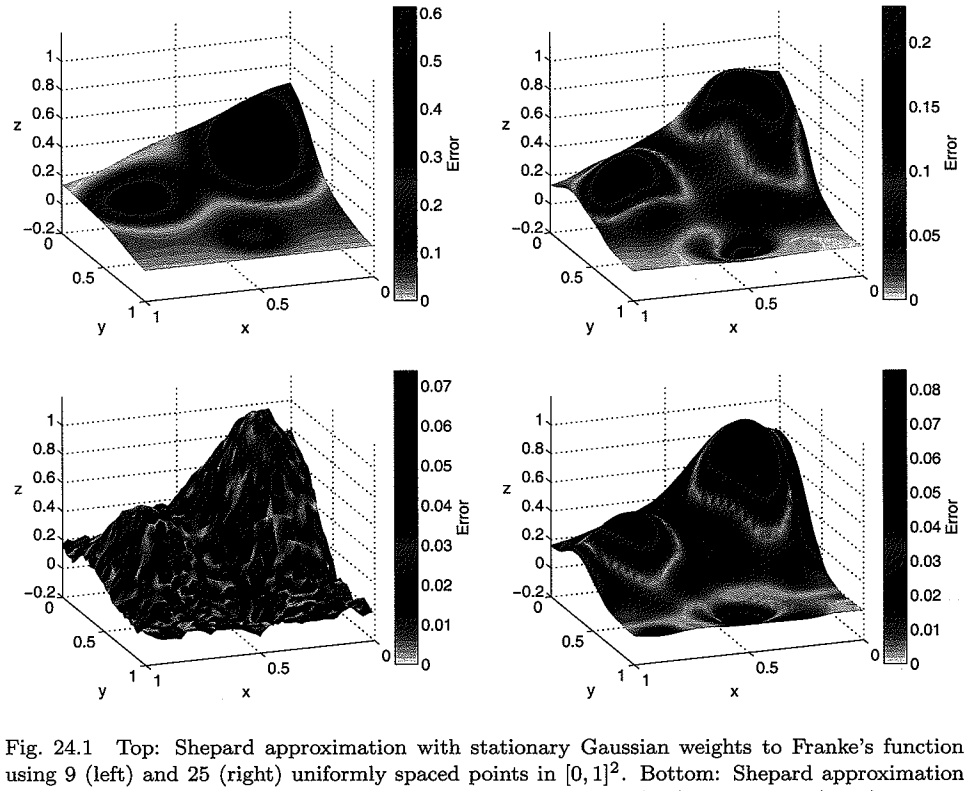

Fig. 24.1 Top: Shepard approximation with stationary Gaussian weights to Franke's function using 9 (left) and 25 (right) uniformly spaced points in $[0,1]^2$. Bottom: Shepard approximation with Gaussian weights to noisy Franke's function using $\varepsilon = 48$ (left) and $\varepsilon = 12$ (right) on 1089 uniformly spaced points in $[0,1]^2$.

```
 5  f4 = @(x,y) 0.2*exp(-((9*x-4).^2+(9*y-7).^2));
 6  testfunction = @(x,y) f1(x,y)+f2(x,y)+f3(x,y)-f4(x,y);
 7  N = 1089; gridtype = 'u';
 8  neval = 40;
    % Load data points
 9  name = sprintf('Data2D_%d%s',N,gridtype); load(name)
10  ctrs = dsites;
    % Create vector of function (data) values
11  f = testfunction(dsites(:,1),dsites(:,2));
    % Create neval-by-neval equally spaced evaluation locations
    % in the unit square
12  grid = linspace(0,1,neval); [xe,ye] = meshgrid(grid);
13  epoints = [xe(:) ye(:)];
    % Compute distance matrix between evaluation points and centers
14  DM_eval = DistanceMatrix(epoints,ctrs);
    % Compute evaluation matrix
15  EM = rbf(ep,DM_eval);
```

```
16  EM = EM./repmat(EM*ones(N,1),1,N);  % Shepard normalization
    % Compute quasi-interpolant
17  Pf = EM*f;
    % Compute exact solution, i.e.,
    % evaluate test function on evaluation points
18  exact = testfunction(epoints(:,1),epoints(:,2));
    % Compute errors on evaluation grid
19  maxerr = norm(Pf-exact,inf);
20  rms_err = norm(Pf-exact)/neval;
21  fprintf('RMS error:     %e\n', rms_err)
22  fprintf('Maximum error: %e\n', maxerr)
    % Plot interpolant
23  PlotSurf(xe,ye,Pf,neval,exact,maxerr,[160,20]);
    % Plot absolute error
24  PlotError2D(xe,ye,Pf,exact,maxerr,neval,[160,20]);
```

The next group of experiments (reported in Tables 24.2 and 24.3) should be compared to the distance matrix fits of Chapter 1. The data are again sampled from the test function

$$f_s(\boldsymbol{x}) = 4^s \prod_{d=1}^{s} x_d(1-x_d), \qquad \boldsymbol{x} = (x_1, \ldots, x_s) \in [0,1]^s,$$

which is parametrized by the space dimension s and coded in the MATLAB subroutine `testfunction.m` (see Program C.1).

Since we are using the compactly supported weights $w(\boldsymbol{x}_i, \boldsymbol{x}) = (1-\varepsilon\|\boldsymbol{x}-\boldsymbol{x}_i\|)_+^4 (4\varepsilon\|\boldsymbol{x}-\boldsymbol{x}_i\|+1)$ we can keep the evaluation matrix sparse and are therefore able to deal with much larger data sets. Again, we employ a stationary approximation scheme, *i.e.*, the scale parameter ε for the support of the weight functions is (inversely) proportional to the fill-distance. In fact, we take $\varepsilon = 2^k + 1$ (see line 4 of Program 24.2), where k also determines the number $N = (2^k+1)^s$ of data points used (*c.f.* line 3). These points are taken to be Halton points in $[0,1]^s$ (*c.f.* line 6).

In Tables 24.2 and 24.3 we list RMS-errors (computed on various grids of equally spaced points, *i.e.*, mostly on the boundary of the unit cube in higher dimensions) for a series of approximation problems in $\mathbb{R}^s$ for $s = 1, 2, \ldots, 6$. Again, the approximation order for Shepard's method seems to be roughly $\mathcal{O}(h)$ independent of the space dimension s.

The MATLAB code `Shepard_CS.m` is listed as Program 24.2. This time the evaluation matrix is a sparse matrix which we also compute in two steps. The standard RBF evaluation matrix is computed as for our earlier CSRBF computations on lines 10 and 11 using the subroutine `DistanceMatrixCSRBF.m` (see Chapter 12). This time the Shepard scaling is performed on line 12 with the help of a diagonal matrix stored in the `spdiags` format.

Table 24.2 Shepard fit in 1D–3D with compactly supported weight function.

	1D		2D		3D	
k	N	RMS-error	N	RMS-error	N	RMS-error
1	3	2.844398e-001	9	2.388314e-001	27	1.912827e-001
2	5	1.665656e-001	25	1.271941e-001	125	1.249589e-001
3	9	8.796073e-002	81	7.107988e-002	729	6.898182e-002
4	17	3.970488e-002	289	3.944228e-002	4913	3.718816e-002
5	33	1.738869e-002	1089	3.109722e-002	35937	2.838763e-002
6	65	7.535727e-003	4225	1.888361e-002	274625	9.837855e-003
7	129	3.418925e-003	16641	2.831294e-002		
8	257	1.615694e-003	66049	2.667914e-003		
9	513	7.872903e-004	263169	2.352736e-003		
10	1025	3.884881e-004				
11	2049	1.940611e-004				
12	4097	9.699922e-005				

Table 24.3 Shepard fit in 4D–6D with compactly supported weight function.

	4D		5D		6D	
k	N	RMS-error	N	RMS-error	N	RMS-error
1	81	1.310311e-001	243	8.936253e-002	729	6.372675e-002
2	625	8.943469e-002	3125	6.344921e-002	15625	3.910649e-002
3	6561	4.599027e-002	59049	3.492958e-002	531441	2.234389e-002
4	83521	2.523581e-002				

Program 24.2. `Shepard_CS.m`

```
% Shepard_CS
% Script that performs Shepard approximation for arbitrary
% space dimensions s using sparse matrices
% Calls on: DistanceMatrixCSRBF, MakeSDGrid, testfunction
% Uses:     haltonseq (written by Daniel Dougherty from
%           MATLAB Central File Exchange)
    % Wendland C2 weight function
 1  rbf = @(e,r) r.^4.*(5*spones(r)-4*r);
 2  s = 6;   % Space dimension s
    % Number of Halton data points
 3  k = 3; N = (2^k+1)^s;
 4  ep = 2^k+1; % Scale parameter for basis function
 5  neval = 4; M = neval^s;
    % Compute data sites as Halton points
 6  dsites = haltonseq(N,s);
 7  ctrs = dsites;
    % Create vector of function (data) values
```

```
 8  f = testfunction(s,dsites);
    % Create neval^s equally spaced evaluation locations in
    % the s-dimensional unit cube
 9  epoints = MakeSDGrid(s,neval);
    % Compute evaluation matrix, i.e.,
    % matrix of values of generating functions
10  DM_eval = DistanceMatrixCSRBF(epoints,ctrs,ep);
11  EM = rbf(ep,DM_eval);
    % Shepard scaling
12  EM = spdiags(1./(EM*ones(N,1)),0,M,M)*EM;
    % Compute quasi-interpolant
13  Pf = EM*f;
    % Compute exact solution, i.e.,
    % evaluate test function on evaluation points
14  exact = testfunction(s,epoints);
    % Compute errors on evaluation grid
15  maxerr = norm(Pf-exact,inf);
16  rms_err = norm(Pf-exact)/sqrt(M);
17  fprintf('RMS error:     %e\n', rms_err)
18  fprintf('Maximum error: %e\n', maxerr)
```

24.2 MLS Approximation with Linear Reproduction

In general one would expect a moving least squares method that reproduces linear polynomials to be more accurate than one that reproduces only constants such as Shepard's method. We illustrate this in Table 24.4 where we again use the C^2 compactly supported weight functions $w(\boldsymbol{x}_i, \boldsymbol{x}) = (1 - \varepsilon\|\boldsymbol{x} - \boldsymbol{x}_i\|)_+^4 (4\varepsilon\|\boldsymbol{x} - \boldsymbol{x}_i\| + 1)$ in a stationary approximation setting with a base $\varepsilon = 3$ for $N = 9$ data points (and then scaled inversely proportional to the fill-distance) to approximate data sampled from Franke's function at uniformly spaced points in $[0, 1]^2$. We can see that the MLS method with linear reproduction has a numerical approximation order of $\mathcal{O}(h^2)$. This will be justified theoretically in the next chapter.

Instead of solving a Gram system for each evaluation point as suggested in (22.8) or (22.20) we use the explicit formulas for the Lagrange multipliers and generating functions given for the 2D case in Example 23.3. These formulas are implemented in MATLAB in the subroutine `LinearScaling2D_CS.m` and listed in Program 24.3. In particular, this subroutine is written to deal with compactly supported weight functions and thus uses sparse matrices. As in the assembly of sparse interpolation matrices we make use of *kd*-trees.

We build a *kd*-tree of all of the centers for the weight functions and find — for each evaluation point — those centers whose support overlaps the evaluation point. The input to `LinearScaling2D_CS.m` are `epoints` (an $N \times s$ matrix representing a set of N data sites in $\mathbb{R}^s$), `ctrs` (an $M \times s$ matrix representing a set of M centers

Table 24.4 2D stationary MLS approximation to Franke's function at uniformly spaced points in $[0, 1]^2$ with linear precision using compactly supported weight function.

N	RMS-error	rate
9	1.789573e-001	
25	7.089522e-002	1.3359
81	2.691693e-002	1.3972
289	7.516377e-003	1.8404
1089	1.944252e-003	1.9508
4225	4.903575e-004	1.9873
16641	1.228639e-004	1.9968
66049	3.072866e-005	1.9994
263169	7.658656e-006	2.0044
1050625	1.921486e-006	1.9949

for the weight functions in $\mathbb{R}^s$), `rbf` (an anonymous or inline function defining the RBF weight function), and `ep` (the scale parameter that determines the size of the support of the weight functions). As always, wide functions result from a small value of `ep`, *i.e.*, the size of the support of the weight function is given by `1/ep`. The output of `LinearScaling2D_CS.m` is an $N \times M$ sparse matrix `Phi` that contains the MLS generating functions centered at the points given by `center` and evaluated at the evaluation points in `epoints`. Note that the compactly supported basic function needs to be supplied in its standard (unshifted) form, *e.g.*, as

```
rbf = @(e,r) max(spones(r)-e*r,0).^4.*(4*e*r+spones(r));
```

for the C^2 Wendland function $\varphi_{3,1}(r) = (1 - r)_+^4(4r + 1)$.

For each evaluation point (see the loop over `i` from line 10 to line 33) we compute the six different moments (entries of the Gram matrix G, *c.f.* Example 23.3) required for the computation of the Lagrange multipliers on lines 14–20. Then the values of the Lagrange multipliers and of the generating functions at the ith evaluation point are computed on lines 21–24. The final sparse matrix `Phi` is assembled on line 35.

Program 24.3. `LinearScaling2D_CS.m`

```
% Phi = LinearScaling2D_CS(epoints,ctrs,rbf,ep)
% Forms a sparse matrix of scaled generating functions for MLS
% approximation with linear reproduction.
% Uses:         k-D tree package by Guy Shechter from
%               MATLAB Central File Exchange
 1  function Phi = LinearScaling2D_CS(epoints,ctrs,rbf,ep)
 2  [N,s] = size(epoints); [M,s] = size(ctrs);
 3  alpha = [0 0; 1 0; 0 1; 1 1; 2 0; 0 2];
    % Build k-D tree for centers
```

```
 4  [tmp,tmp,Tree] = kdtree(ctrs,[]);
    % For each eval. point, find centers whose support overlap it
 5  support = 1/ep; mu = zeros(6);
    % Modify the following line for optimum performance
 6  veclength = round(support*N*M/4);
 7  rowidx = zeros(1,veclength); colidx = zeros(1,veclength);
 8  validx = zeros(1,veclength);
 9  istart = 1; iend = 0;
10  for i = 1:N
11     [pts,dist,idx] = kdrangequery(Tree,epoints(i,:),support);
12     newlen = length(idx);
       % Vector of basis functions
13     Phi_i = rbf(ep,dist');
       % Compute all 6 moments for i-th evaluation point
14     for j=1:6
15        x_to_alpha = 1;
16        for coord=1:s
17a          x_to_alpha = x_to_alpha .*(ctrs(idx,coord)-...
17b            repmat(epoints(i,coord),newlen,1)).^alpha(j,coord);
18        end
19        mu(j) = Phi_i*x_to_alpha;
20     end
21     L1=(mu(4)^2-mu(5)*mu(6)); L2=(mu(2)*mu(6)-mu(3)*mu(4));
22     L3=(mu(5)*mu(3)-mu(2)*mu(4));
23a    scaling = L1*repmat(1,newlen,1) + ...
23b          L2*(ctrs(idx,1)-repmat(epoints(i,1),newlen,1))+...
23c          L3*(ctrs(idx,2)-repmat(epoints(i,2),newlen,1));
24a    denom = mu(2)^2*mu(6)+mu(5)*mu(3)^2-mu(1)*mu(5)*mu(6)-...
24b            2*mu(2)*mu(3)*mu(4)+mu(1)*mu(4)^2;
25     if (denom ~= 0)
26        scaling = scaling/denom;
27        iend = iend + newlen;
28        rowidx(istart:iend) = repmat(i,1,newlen);
29        colidx(istart:iend) = idx';
30        validx(istart:iend) = Phi_i.*scaling';
31        istart = istart + newlen;
32     end
33  end
34  filled = find(rowidx); % only those actually filled
35  Phi = sparse(rowidx(filled),colidx(filled),validx(filled),N,M);
    % Free memory
36  clear rowidx colidx validx; kdtree([],[],Tree);
```


With all the difficult coding relegated to the subroutine `LinearScaling2D_CS.m` the main program `LinearMLS2D_CS.m` is rather simple. It is listed in Program 24.4. The version of the code listed here is adapted to read a data file that contains both data sites (in the variable `dsites`) and function values (in the variable `rhs`). For the example displayed in Figure 24.2 we used an actual set of 1:250,000-scale Digital Elevation Model (DEM) data from the U.S. Geological Survey, namely the data set `Dubuque-E` which contains elevation data from a region in northeastern Iowa, northwestern Illinois, and southwestern Wisconsin. The data set is available on the worldwide web at `http://edcsgs9.cr.usgs.gov/glis/hyper/guide/1_dgr_demfig/-states/IL.html`. The original data set contains 1201×1201 uniformly spaced measurements ranging in height from 178 meters to 426 meters. We converted the DEM data format with the utility `DEM2XYZN` that can be downloaded from `http://data.geocomm.com/dem/dem2xyzn/`, selected only the northeastern quadrant of $601 \times 601 = 361201$ elevation values, and scaled the x and y coordinates that originally defined a square with a side length of about 35 miles to the unit square. The resulting data set `Data2D_DubuqueNE` is included on the enclosed CD. An MLS approximation with linear reproduction based on the C^2 compactly supported Wendland weight used earlier was evaluated on a grid of 60×60 equally spaced points and is displayed in Figure 24.2. We use a shape parameter of $\varepsilon = 30$ to determine the support size of the weight functions.

Program 24.4. `LinearMLS2D_CS.m`

```
% LinearMLS2D_CS
% Script that performs MLS approximation with linear reproduction
% using sparse matrices
% Calls on: LinearScaling2D_CS
 1  rbf = @(e,r) max(spones(r)-e*r,0).^4.*(4*e*r+spones(r));
 2  ep = 30; neval = 60;
    % Load data points and rhs
 3  load('Data2D_DubuqueNE');
 4  ctrs = dsites;
    % Create neval-by-neval equally spaced evaluation locations
    % in the unit square
 5  grid = linspace(0,1,neval); [xe,ye] = meshgrid(grid);
 6  epoints = [xe(:) ye(:)];
    % Compute evaluation matrix
 7  EM = LinearScaling2D_CS(epoints,ctrs,rbf,ep);
    % Compute MLS approximation   (rhs read from data file)
 8  Pf = EM*rhs;
 9  figure   % Plot MLS fit
10  surfplot = surf(xe,ye,reshape(Pf,neval,neval));
11  set(surfplot,'FaceColor','interp','EdgeColor','none')
12  view([15,35]); camlight; lighting gouraud; colormap summer
```

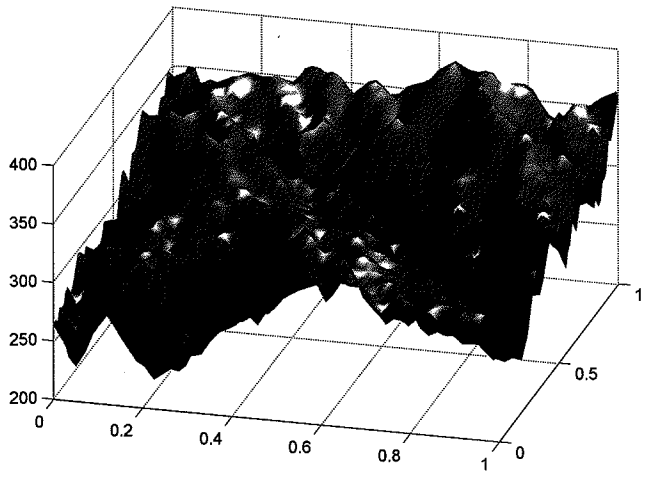

Fig. 24.2 MLS approximation using compactly supported weight for elevation data in northwestern Illinois.

A much simpler (but also much slower) implementation of MLS approximation with linear reproduction is given as Program 24.5. In this implementation we solve the local least squares problem for each evaluation point $\boldsymbol{x}$, *i.e.*, we obtain the MLS approximation at the point $\boldsymbol{x}$ as

$$\mathcal{P}f(\boldsymbol{x}) = u(\boldsymbol{x}, \boldsymbol{x}) = \sum_{j=1}^{m} c_j(\boldsymbol{x}) p_j(\boldsymbol{x} - \boldsymbol{x}) = c_1(\boldsymbol{x}),$$

where the coefficients are found by solving the Gram system (*c.f.* 22.8)

$$G(\boldsymbol{x})c(\boldsymbol{x}) = \boldsymbol{f_p}(\boldsymbol{x}).$$

The solution of these systems (for each evaluation point $\boldsymbol{x}$) is computed on line 18 of the program, with the Gram matrix built as

$$G(\boldsymbol{x}) = P^T(\boldsymbol{x}) W(\boldsymbol{x}) P(\boldsymbol{x}),$$

and the polynomial matrix P with entries $P_{ij}(\boldsymbol{x}) = p_j(\boldsymbol{x}_i - \boldsymbol{x})$ and diagonal weight matrix computed on lines 16 and 17, respectively. The subroutine `LinearScaling2D_CS.m` is not required by this program. Compare this program with Program 24.1.

Program 24.5. `LinearMLS2D_GramSolve.m`

```
% LinearMLS2D_GramSolve
% Script that performs MLS approximation with linear reproduction
% by solving local Gram systems
% Calls on: DistanceMatrix, PlotSurf, PlotError2D
```



```
 1  rbf = @(e,r) exp(-(e*r).^2); ep = 5.5;
    % Define Franke's function as testfunction
 2  f1 = @(x,y) 0.75*exp(-((9*x-2).^2+(9*y-2).^2)/4);
 3  f2 = @(x,y) 0.75*exp(-((9*x+1).^2/49+(9*y+1).^2/10));
 4  f3 = @(x,y) 0.5*exp(-((9*x-7).^2+(9*y-3).^2)/4);
 5  f4 = @(x,y) 0.2*exp(-((9*x-4).^2+(9*y-7).^2));
 6  testfunction = @(x,y) f1(x,y)+f2(x,y)+f3(x,y)-f4(x,y);
 7  N = 1089; gridtype = 'u';
 8  neval = 40;
    % Load data points
 9  name = sprintf('Data2D_%d%s',N,gridtype); load(name)
10  ctrs = dsites;
    % Create vector of function (data) values
11  f = testfunction(dsites(:,1),dsites(:,2));
    % Create neval-by-neval equally spaced evaluation locations
    % in the unit square
12  grid = linspace(0,1,neval); [xe,ye] = meshgrid(grid);
13  epoints = [xe(:) ye(:)];
    % Compute MLS approximation with shifted basis polynomials
14  Pf = zeros(neval^2,1);
15  for j=1:neval^2
16     P = [ones(N,1) dsites-repmat(epoints(j,:),N,1)];
17     W = diag(rbf(ep,DistanceMatrix(epoints(j,:),ctrs)));
18     c = (P'*W*P)\(P'*W*f);
19     Pf(j) = c(1);
20  end
    % Compute exact solution, i.e.,
    % evaluate test function on evaluation points
21  exact = testfunction(epoints(:,1),epoints(:,2));
    % Compute errors on evaluation grid
22  maxerr = norm(Pf-exact,inf);
23  rms_err = norm(Pf-exact)/neval;
    % Plot interpolant
24  PlotSurf(xe,ye,Pf,neval,exact,maxerr,[160,20]);
    % Plot absolute error
25  PlotError2D(xe,ye,Pf,exact,maxerr,neval,[160,20]);
```

Alternatively, we could have coded lines 14–20 with an unshifted polynomial basis (*c.f.* the discussion in Chapter 22), in which case we would have

```
14  P = [ones(N,1) dsites];
15  Pf = zeros(neval^2,1);
16  for j=1:neval^2
```

```
17      W = diag(rbf(ep,DistanceMatrix(epoints(j,:),ctrs)));
18      c = (P'*W*P)\(P'*W*f);
19      Pf(j) = [1 epoints(j,:)]*c;
20  end
```

in Program 24.5. This requires setting up the matrix P only once. However, the computations are numerically not as stable. Also, additional computation of the approximant on line 19 is required via a dot product.

24.3 Plots of Basis-Dual Basis Pairs

We now provide some plots of the polynomial basis functions p for the standard MLS approach, the dual basis functions λ (Lagrange multipliers), and the generating functions Ψ from the Backus-Gilbert approach for a one-dimensional example with $\mathcal{X}$ being the set of 11 equally spaced points in $[0,1]$. We take $m = 3$, *i.e.*, we consider the case that ensures reproduction of quadratic polynomials. The weight function is taken to be the standard Gaussian radial function scaled with a shape parameter $\varepsilon = 5$. In the only exception in this book, these plots were created with Maple using the program `MLSDualBases.mws` included in Appendix C.

The three basis polynomials $p_1(x) = 1$, $p_2(x) = x$, and $p_3(x) = x^2$ are shown in Figure 24.3, whereas the dual basis functions λ_1, λ_2, and λ_3 are displayed in Figure 24.4.

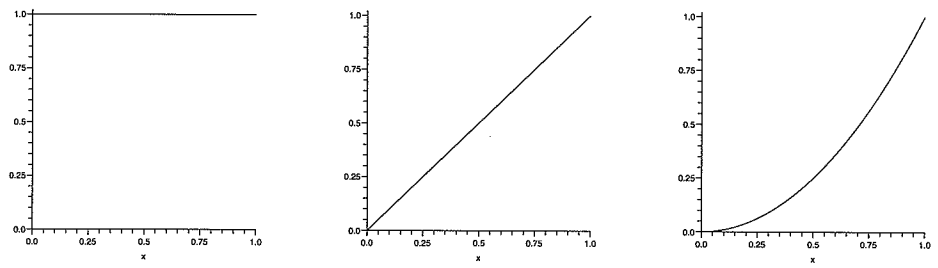

Fig. 24.3 Plot of the three polynomial basis functions for moving least squares approximation with quadratic reproduction on $[0,1]$.

In Figure 24.5 we plot three MLS generating functions (solid) together with the corresponding generating functions from the approximate moving least squares approach (dashed) described in Chapter 26. The generating functions for the approximate MLS approach are Laguerre-Gaussians (*c.f.* Section 4.2). While the standard MLS generating functions reflect the fact that the data is given on a finite interval, the generating functions for approximate MLS approximation are all just shifts of

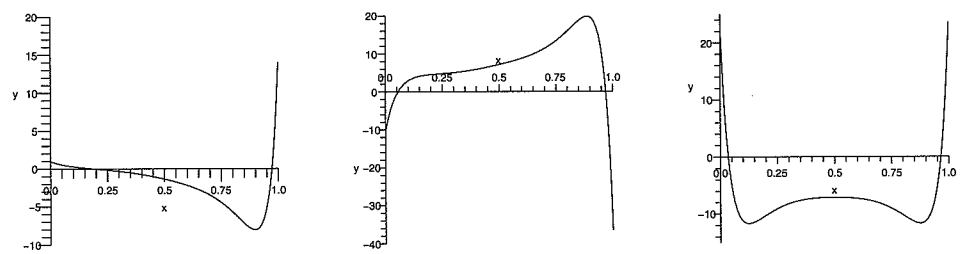

Fig. 24.4 Plot of the three dual basis functions for moving least squares approximation with quadratic reproduction for 11 equally spaced points in $[0,1]$.

the one function

$$\Psi(\boldsymbol{x},\boldsymbol{y}) = \frac{1}{\sqrt{\mathcal{D}\pi}}\left(\frac{3}{2} - \frac{\|\boldsymbol{x}-\boldsymbol{y}\|^2}{\mathcal{D}h^2}\right) e^{-\frac{\|\boldsymbol{x}-\boldsymbol{y}\|^2}{\mathcal{D}h^2}}$$

to the center points $\boldsymbol{y}$. Here we identify the scale parameter $\mathcal{D}$ with our shape parameter ε for the weight function via $\varepsilon = \frac{1}{\sqrt{\mathcal{D}}h}$. For this example with 11 points in $[0,1]$ we have $h = 1/10$, so that $\varepsilon = 5$ corresponds to a value of $\mathcal{D} = 4$.

In the center of the interval, where the influence of the boundary is minimal, the two types of generating functions are almost identical (see the right plot in Figure 24.5).

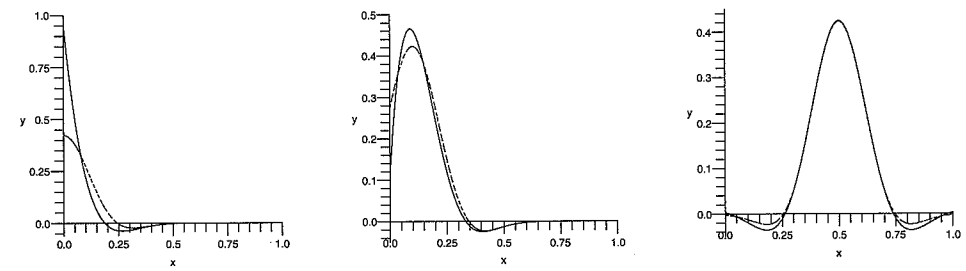

Fig. 24.5 Standard MLS generating functions (solid) and approximate MLS generating functions (dashed) centered at three of the 11 equally spaced points in $[0,1]$.

If the data points are no longer equally spaced, the Lagrange functions and generating functions are also less uniform. Figures 24.6 and 24.7 illustrate this dependence on the data distribution for 11 Halton points in $[0,1]$.

Finally, we provide plots of MLS generating functions for the case of reproduction of linear polynomials in 2D (see Figure 24.8). These plots were created with the MATLAB program `LinearMLS2D_CS.m` (see Program 24.4) by plotting column j of the evaluation matrix `EM` corresponding to the values of the jth generating function. We used the C^2 Wendland weights $w(\boldsymbol{x}_i,\boldsymbol{x}) = (1-\varepsilon\|\boldsymbol{x}-\boldsymbol{x}_i\|)_+^4\,(4\varepsilon\|\boldsymbol{x}-\boldsymbol{x}_i\|+1)$ with $\varepsilon = 5$.

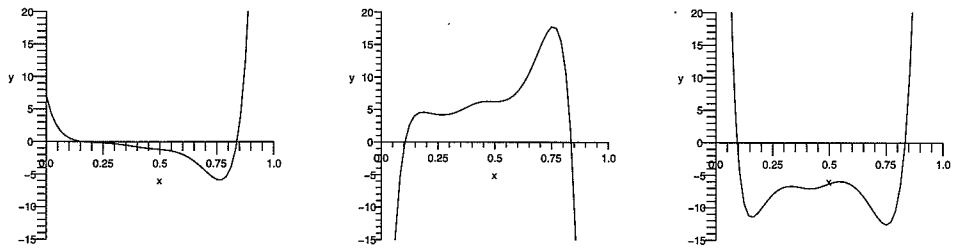

Fig. 24.6 Plot of the three dual basis functions for moving least squares approximation with quadratic reproduction for 11 Halton points in $[0, 1]$.

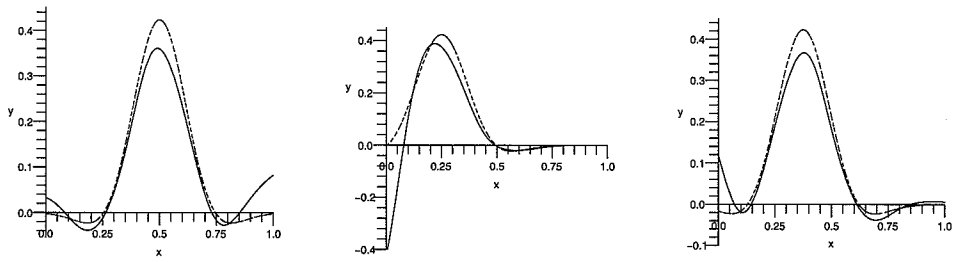

Fig. 24.7 Standard MLS generating functions (solid) and approximate MLS generating functions (dashed) centered at three of the 11 Halton points in $[0, 1]$.

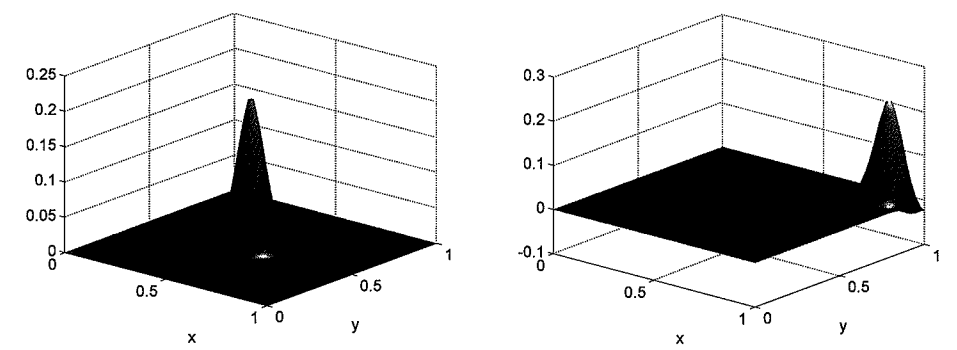

Fig. 24.8 MLS generating functions for linear reproduction centered at two of 289 uniformly spaced data sites in $[0, 1]^2$.

Chapter 25

Error Bounds for Moving Least Squares Approximation

25.1 Approximation Order of Moving Least Squares

Since the moving least squares approximants can be written as quasi-interpolants we can use standard techniques to derive their point-wise error estimates. The standard argument proceeds as follows. Let f be a given (smooth) function that generates the data, *i.e.*, $f_1 = f(\boldsymbol{x}_1), \ldots, f_N = f(\boldsymbol{x}_N)$, and let p be an arbitrary polynomial. Moreover, assume that the moving least squares approximant is given in the form

$$\mathcal{P}_f(\boldsymbol{x}) = \sum_{i=1}^{N} f(\boldsymbol{x}_i)\Psi_i(\boldsymbol{x}, \boldsymbol{x})$$

with the generating functions Ψ_i satisfying the polynomial reproduction property

$$\sum_{i=1}^{N} p(\boldsymbol{x}_i)\Psi_i(\boldsymbol{x}, \boldsymbol{x}) = p(\boldsymbol{x}), \quad \text{for all } p \in \Pi_d^s,$$

as described in Chapter 22. Then, due to the polynomial reproduction property of the generating functions, we get

$$\begin{aligned} |f(\boldsymbol{x}) - \mathcal{P}_f(\boldsymbol{x})| &\le |f(\boldsymbol{x}) - p(\boldsymbol{x})| + |p(\boldsymbol{x}) - \sum_{i=1}^{N} f(\boldsymbol{x}_i)\Psi_i(\boldsymbol{x}, \boldsymbol{x})| \\ &= |f(\boldsymbol{x}) - p(\boldsymbol{x})| + |\sum_{i=1}^{N} p(\boldsymbol{x}_i)\Psi_i(\boldsymbol{x}, \boldsymbol{x}) - \sum_{i=1}^{N} f(\boldsymbol{x}_i)\Psi_i(\boldsymbol{x}, \boldsymbol{x})|. \end{aligned}$$

Combination of the two sum and the definition of the discrete maximum norm yield

$$\begin{aligned} |f(\boldsymbol{x}) - \mathcal{P}_f(\boldsymbol{x})| &\le |f(\boldsymbol{x}) - p(\boldsymbol{x})| + \sum_{i=1}^{N} |p(\boldsymbol{x}_i) - f(\boldsymbol{x}_i)||\Psi_i(\boldsymbol{x}, \boldsymbol{x})| \\ &\le \|f - p\|_\infty \left[1 + \sum_{i=1}^{N} |\Psi_i(\boldsymbol{x}, \boldsymbol{x})|\right]. \end{aligned} \tag{25.1}$$

We see that in order to refine the error estimate we now have to answer two questions:

- How well do polynomials approximate f? This can be achieved with standard Taylor expansions.
- Are the generating functions bounded? The expression $\sum_{i=1}^{N} |\Psi_i(\boldsymbol{x}, \boldsymbol{x})|$ is known as the (value of the) *Lebesgue function*, and finding a bound for the Lebesgue function is the main task that remains.

By taking the polynomial p above to be the Taylor polynomial of total degree d for f at $\boldsymbol{x}$, the remainder term immediately yields an estimate of the form

$$\|f - p\|_\infty \leq C_1 h^{d+1} \max_{\boldsymbol{\xi} \in \Omega} |D^{\boldsymbol{\alpha}} f(\boldsymbol{\xi})|, \qquad |\boldsymbol{\alpha}| = d + 1. \tag{25.2}$$

Thus, if we can establish a uniform bound for the Lebesgue function, then (25.1) and (25.2) will result in

$$|f(\boldsymbol{x}) - \mathcal{P}_f(\boldsymbol{x})| \leq C h^{d+1} \max_{\boldsymbol{\xi} \in \Omega} |D^{\boldsymbol{\alpha}} f(\boldsymbol{\xi})|, \qquad |\boldsymbol{\alpha}| = d + 1,$$

which shows that moving least squares approximation with polynomial reproduction of degree d has approximation order $\mathcal{O}(h^{d+1})$.

For Shepard's method, *i.e.*, moving least squares approximation with constant reproduction (*i.e.*, $m = 1$ or $d = 0$), we saw above that the generating functions are of the form

$$\Psi_i(\boldsymbol{x}, \boldsymbol{x}) = \frac{w(\boldsymbol{x}_i, \boldsymbol{x})}{\sum_{j=1}^{N} w(\boldsymbol{x}_j, \boldsymbol{x})}$$

and form a partition of unity (see (23.3)). Therefore the Lebesgue function admits the uniform bound

$$\sum_{i=1}^{N} |\Psi_i(\boldsymbol{x}, \boldsymbol{x})| = 1.$$

This shows that Shepard's method has approximation order $\mathcal{O}(h)$.

Bounding the Lebesgue function in the general case is more involved and is the subject of the papers [Levin (1998); Wendland (2001a)]. As indicated above, this results in approximation order $\mathcal{O}(h^{d+1})$ for a moving least squares method that reproduces polynomials of degree d. In both papers the authors assumed that the weight function is compactly supported, and that the support size is scaled proportional to the fill distance. The following theorem paraphrases the results of [Levin (1998); Wendland (2001a)].

Theorem 25.1. *Let $\Omega \subset \mathbb{R}^s$. If $f \in C^{d+1}(\Omega)$, $\{\boldsymbol{x}_i : i = 1, \ldots, N\} \subset \Omega$ are quasi-uniformly distributed with fill distance h, the weight functions $w_i(\boldsymbol{x}) = w(\boldsymbol{x}_i, \boldsymbol{x})$ are compactly supported with support size $\rho_i = ch$ ($c = \text{const.}$), and if polynomials in Π_d^s are reproduced according to (22.13), then the scale of MLS approximations*

$$\mathcal{P}_f^{(h)}(\boldsymbol{x}) = \sum_{i=1}^{N} f(\boldsymbol{x}_i) \Psi\left(\frac{\boldsymbol{x} - \boldsymbol{x}_i}{\rho_i}\right), \tag{25.3}$$

with generating functions $\Psi(\boldsymbol{x} - \boldsymbol{x}_i) = \Psi_i(\boldsymbol{x}, \boldsymbol{x})$ determined via (22.17)–(22.19) satisfies

$$\|f - \mathcal{P}_f^{(h)}\|_\infty \leq C h^{d+1} \max_{\boldsymbol{\xi} \in \Omega} |D^{\boldsymbol{\alpha}} f(\boldsymbol{\xi})|, \qquad |\boldsymbol{\alpha}| = d + 1.$$

It should be possible to arrive at similar estimates if the weight function only decays fast enough (see, *e.g.*, the survey [de Boor (1993)]).

Aside from this constraint on the weight function (which essentially corresponds to a stationary approximation scheme), the choice of weight function w does not play a role in determining the approximation order of the moving least squares method. As noted earlier, it only determines the smoothness of $\mathcal{P}_f$. For example, in the paper [Cleveland and Loader (1996)] from the statistics literature on local regression the authors state that often "the choice [of weight function] is not too critical", and the use of the so-called *tri-cube*

$$w_i(\boldsymbol{x}) = (1 - \|\boldsymbol{x} - \boldsymbol{x}_i\|^3)_+^3, \qquad \boldsymbol{x} \in \mathbb{R}^s,$$

is suggested. Of course, many other weight functions such as (radial) B-splines or any of the (bell-shaped) radial basis functions studied earlier can also be used. If the weight function is compactly supported, then the generating functions Ψ_i will be so, too. This leads to computationally efficient methods since the Gram matrix $G(\boldsymbol{x})$ will be sparse.

An interesting question is also the size of the support of the different local weight functions. Obviously, a fixed support size for all weight functions is possible. However, this will cause serious problems as soon as the data are not uniformly distributed. Therefore, in the arguments in [Levin (1998); Wendland (2001a)] the assumption is made that the data are at least quasi-uniformly distributed. Another choice for the support size of the individual weight functions is based on the number of nearest neighbors, *i.e.*, the support size is chosen so that each of the local weight functions contains the same number of centers in its support. A third possibility is suggested in [Schaback (2000b)] where the author proposes to use another moving least squares approximation based on (equally spaced) auxiliary points to determine a smooth function δ such that at each evaluation point $\boldsymbol{x}$ the radius of the support of the weight function is given by $\delta(\boldsymbol{x})$. However, convergence estimates for these latter two choices do not exist.

Sobolev error estimates are provided for moving least squares approximation with compactly supported weight functions in [Armentano (2001)]. The rates obtained in that paper are not in terms of the fill distance but instead in terms of the support size ρ of the weight function. Moreover, it is assumed that for general s and $m = \binom{s+d}{d}$ the local Lagrange functions are bounded. As mentioned above, this is the hard part, and in [Armentano (2001)] such bounds are only provided in the case $s = 2$ with $d = 1$ or $d = 2$. However, if combined with the general bounds for the Lebesgue function provided by Wendland, the paper [Armentano (2001)] yields the following estimates:

$$|f(\boldsymbol{x}) - \mathcal{P}_f(\boldsymbol{x})| \leq C \rho^{d+1} \max_{\boldsymbol{\xi} \in \Omega} |D^{\boldsymbol{\alpha}} f(\boldsymbol{\xi})|, \qquad |\boldsymbol{\alpha}| = d + 1,$$

but also

$$|\nabla(f-\mathcal{P}_f)(\boldsymbol{x})| \le C\rho^d \max_{\boldsymbol{\xi}\in\Omega} |D^{\boldsymbol{\alpha}} f(\boldsymbol{\xi})|, \qquad |\boldsymbol{\alpha}| = d+1.$$

In the weaker (local) L_2-norm we have

$$\|f-\mathcal{P}_f\|_{L_2(B_j\cap\Omega)} \le C\rho^{d+1}|f|_{W_2^{d+1}(B_j\cap\Omega)}$$

and

$$\|\nabla(f-\mathcal{P}_f)\|_{L_2(B_j\cap\Omega)} \le C\rho^{d}|f|_{W_2^{d+1}(B_j\cap\Omega)},$$

where the balls B_j provide a finite cover of the domain Ω, *i.e.*, $\Omega \subseteq \bigcup_j B_j$, and the number of overlapping balls is bounded. Here $W_2^d(\Omega)$ is the Sobolev space of functions whose derivatives up to order d are in L_2 (*c.f.* Chapter 13).

We close this chapter by pointing out that early error estimates for some special cases were provided in [Farwig (1987); Farwig (1991)].

Chapter 26

Approximate Moving Least Squares Approximation

26.1 High-order Shepard Methods via Moment Conditions

While we mentioned earlier that the weight function does not have an effect on the approximation order results for Shepard's method (cf. Theorem 25.1), wee now present some heuristic considerations for obtaining higher-order Shepard methods. This will result in certain conditions on the moments of the weight function w.

Recall that using our shifted monomial representation we obtain $\mathcal{P}_f(\boldsymbol{x}) = c_1(\boldsymbol{x})$ (see (22.5)) for any degree d. Therefore, if we can find weight functions $w(\boldsymbol{x}_i,\cdot)$ such that the first row (and first column) of the Gram matrix $G(\boldsymbol{x})$ becomes $[\sum_i w(\boldsymbol{x}_i,\boldsymbol{x}),0,\ldots,0]^T = [\langle 1,1\rangle_{w_{\boldsymbol{x}}},0,\ldots,0]^T$, then $c_1(\boldsymbol{x}) = \langle f,1\rangle_{w_{\boldsymbol{x}}}/\langle 1,1\rangle_{w_{\boldsymbol{x}}}$ via (22.8). Thus,

$$\mathcal{P}_f(\boldsymbol{x}) = \sum_{i=1}^N f(\boldsymbol{x}_i)\frac{w(\boldsymbol{x}_i,\boldsymbol{x})}{\sum_{j=1}^N w(\boldsymbol{x}_j,\boldsymbol{x})},$$

but now — by construction — the method has approximation order $\mathcal{O}(h^{d+1})$ (instead of the mere $\mathcal{O}(h)$ of Shepard's method which permits the use of *arbitrary* weights).

We therefore use the *discrete moments* (*c.f.* Section 23.2)

$$\mu_{\boldsymbol{\alpha}} = \sum_{i=1}^N (\boldsymbol{x}_i-\boldsymbol{x})^{\boldsymbol{\alpha}} w(\boldsymbol{x}_i,\boldsymbol{x}), \qquad \boldsymbol{x}\in\mathbb{R}^s, \qquad 0\le|\boldsymbol{\alpha}|\le d, \tag{26.1}$$

where $\boldsymbol{\alpha}$ is a multi-index, and then demand

$$\mu_{\boldsymbol{\alpha}} = \delta_{\boldsymbol{\alpha}\mathbf{0}}, \qquad 0\le|\boldsymbol{\alpha}|\le d. \tag{26.2}$$

As mentioned in Chapter 23 these moments provide the entries of the Gram matrix $G(\boldsymbol{x})$. The condition $\mu_{\mathbf{0}} = 1$ ensures that the weights $w(\boldsymbol{x}_i,\cdot)$ form a *partition of unity*, and we end up with a quasi-interpolant of the form

$$\mathcal{P}_f(\boldsymbol{x}) = \sum_{i=1}^N f(\boldsymbol{x}_i) w(\boldsymbol{x}_i,\boldsymbol{x}).$$

We can summarize our heuristics in

Proposition 26.1. *Let $\Omega \subset \mathbb{R}^s$. If $f \in C^{d+1}(\Omega)$, the data sites $\{\boldsymbol{x}_i : i = 1, \ldots, N\} \subset \Omega$ are quasi-uniformly distributed with fill distance h, the weight functions $w(\boldsymbol{x}_i, \cdot) = w(\boldsymbol{x}_i - \boldsymbol{x})$ are compactly supported with support size $\rho_i = ch$ ($c = \text{const.}$), and the discrete moment conditions*

$$\mu_{\boldsymbol{\alpha}} = \delta_{\boldsymbol{\alpha}\mathbf{0}}, \qquad 0 \le |\boldsymbol{\alpha}| \le d,$$

are satisfied, then

$$\mathcal{P}_f^{(h)}(\boldsymbol{x}) = \sum_{i=1}^{N} f(\boldsymbol{x}_i) w\left(\frac{\boldsymbol{x}_i - \boldsymbol{x}}{\rho_i}\right)$$

has approximation order $\mathcal{O}(h^{d+1})$.

The discrete moment conditions in Proposition 26.1 lead us to the following interpretation of the weight function w:

$$w(\boldsymbol{x}, \boldsymbol{y}) = w_0(\boldsymbol{x}, \boldsymbol{y}) q(\boldsymbol{x} - \boldsymbol{y}), \qquad q \in \Pi_d^s, \tag{26.3}$$

where w_0 is a new (arbitrary) weight function, and q is a polynomial of degree d orthonormal in the sense of the inner product (22.2) with respect to w_0. This view is also taken by Liu and co-workers (see, *e.g.*, [Li and Liu (2002)]) where q is called the *correction function.*

The discrete moment conditions in Proposition 26.1 are difficult to satisfy analytically, as is the construction of discrete multivariate orthogonal polynomials as needed for (26.3). In order to obtain quasi-interpolants for arbitrary space dimensions and scattered data sites we consider the concept of approximate approximation in the next section. There one satisfies continuous moment conditions while ensuring that the discrete moment conditions are almost satisfied.

26.2 Approximate Approximation

We now give the main results on approximate approximation relevant to our work. The concept of *approximate approximation* was first introduced by Maz'ya in the early 1990s (see [Maz'ya (1991); Maz'ya (1994)]). To keep the discussion transparent we restrict ourselves to results for regular center distributions. However, irregularly spaced data are also allowed by the theory (see, *e.g.*, [Maz'ya and Schmidt (2001); Lanzara *et al.* (2006)]) and have been investigated in practice (see, *e.g.*, [Fasshauer (2004)] or [Lanzara *et al.* (2006)]). In [Maz'ya and Schmidt (2001)] the authors present a quasi-interpolation scheme of the form

$$\mathcal{M}_f^{(h)}(\boldsymbol{x}) = \mathcal{D}^{-s/2} \sum_{\boldsymbol{\nu} \in \mathbb{Z}^s} f(\boldsymbol{x}_{\boldsymbol{\nu}}) \Psi\left(\frac{\boldsymbol{x} - \boldsymbol{x}_{\boldsymbol{\nu}}}{\sqrt{\mathcal{D}} h}\right), \tag{26.4}$$

where the data sites $\boldsymbol{x}_{\boldsymbol{\nu}} = h\boldsymbol{\nu}$ are required to lie on a regular s-dimensional grid with gridsize h. The parameter $\mathcal{D}$ scales the support of the generating function Ψ. As we

will see below, the parameter $\mathcal{D}$ can be chosen to make a so-called *saturation error* so small that it does not affect numerical computations. The generating function is required to satisfy the *continuous moment conditions*

$$\int_{\mathbb{R}^s} \boldsymbol{y}^{\boldsymbol{\alpha}} \Psi(\boldsymbol{y}) d\boldsymbol{y} = \delta_{\boldsymbol{\alpha}\mathbf{0}}, \qquad 0 \le |\boldsymbol{\alpha}| \le d. \tag{26.5}$$

This is the continuous analog of (26.2).

The following approximate approximation result is due to Maz'ya and Schmidt (see, *e.g.*, [Maz'ya and Schmidt (2001)]).

Theorem 26.1. *Let $f \in C^{d+1}(\mathbb{R}^s)$, $\{\boldsymbol{x}_{\boldsymbol{\nu}} : \boldsymbol{\nu} \in \mathbb{Z}^s\} \subset \mathbb{R}^s$ and let Ψ be a continuous generating function which satisfies the moment conditions (26.5) along with the decay requirement*

$$|\Psi(\boldsymbol{x})| \le c_K (1 + \|\boldsymbol{x}\|^2)^{-K/2}, \qquad \boldsymbol{x} \in \mathbb{R}^s,$$

where c_K is some constant, $K > d + s + 1$ and d is the desired degree of polynomial reproduction. Then

$$\|f - \mathcal{M}_f^{(h)}\|_\infty = \mathcal{O}(h^{d+1}) + E_0(\Psi, \mathcal{D}). \tag{26.6}$$

Using Poisson's summation formula one can bound the *saturation error* E_0 by (see Lemma 2.1 in [Maz'ya and Schmidt (1996)])

$$E_0(\Psi, \mathcal{D}) \le \sum_{\boldsymbol{\nu} \in \mathbb{Z}^s \setminus \{\mathbf{0}\}} \mathcal{F}\Psi(\sqrt{\mathcal{D}}\boldsymbol{\nu}). \tag{26.7}$$

For this result the Fourier transform of Ψ is defined via

$$\mathcal{F}\Psi(\boldsymbol{\omega}) = \int_{\mathbb{R}^s} \Psi(\boldsymbol{x}) e^{-2\pi i \boldsymbol{x} \cdot \boldsymbol{\omega}} d\boldsymbol{x}$$

with $\boldsymbol{x} \cdot \boldsymbol{\omega}$ the standard Euclidean inner product of $\boldsymbol{x}$ and $\boldsymbol{\omega}$ in $\mathbb{R}^s$. The saturation error can be interpreted as the discrepancy between the continuous and discrete moment conditions, and its influence can be controlled by the choice of the parameter $\mathcal{D}$ in (26.4).

If we use radial generating functions, then we can use the formula for the Fourier transform of a radial function (see Theorem B.1 in Appendix B) to compute the leading term of (26.7), and therefore obtain an estimate for $\mathcal{D}$ for any desired saturation error. If $\mathcal{D}$ is chosen large enough, then the saturation error will be smaller than the machine accuracy for any given computer, and therefore not noticeable in numerical computations. This means, that even though — theoretically — the quasi-interpolation scheme (26.4) fails to converge, it does converge for all practical numerical purposes. Moreover, we can have an arbitrarily high rate of convergence, $d+1$, by picking a generating function Ψ with sufficiently many vanishing moments.

We point out that the error bound (26.6) is formulated for gridded data on an unbounded domain. An analogous result also holds for finite grids (see [Ivanov *et al.* (1999)]); and similar results for scattered data on bounded domains can be found in, *e.g.*, [Lanzara *et al.* (2006); Maz'ya and Schmidt (2001)]. However, in this case the function f (defining the "data") is required to be compactly supported on the finite domain, or appropriately mollified.

26.3 Construction of Generating Functions for Approximate MLS Approximation

In order to construct the generating functions for the approximate MLS approach we assume the generating function Ψ to be radial and of the form

$$\Psi(\boldsymbol{x}) = \psi_0(\|\boldsymbol{x}\|^2)q(\|\boldsymbol{x}\|^2).$$

Therefore, the continuous moment conditions become (*c.f.* (26.5))

$$\int_{\mathbb{R}^s} \|\boldsymbol{x}\|^{2k} q(\|\boldsymbol{x}\|^2)\psi_0(\|\boldsymbol{x}\|^2)dx = \delta_{k0}, \qquad 0 \le k \le d, \tag{26.8}$$

and we need to determine the univariate polynomial q of degree d accordingly to get approximation order $\mathcal{O}(h^{2d+2})$.

By using s-dimensional spherical coordinates we can rewrite the integral in (26.8) as

$$\begin{aligned}
&\int_0^\infty \int_0^{2\pi} \int_0^\pi \cdots \int_0^\pi r^{2k} q(r^2)\psi_0(r^2) r^{s-1} \sin^{s-2}\phi_1 \ldots \sin\phi_{s-2} d\phi_1 \ldots d\phi_{s-2} d\theta dr \\
&\quad = 2\pi \int_0^\pi \sin^{s-2}\phi_1 d\phi_1 \ldots \int_0^\pi \sin\phi_{s-2} d\phi_{s-2} \int_0^\infty r^{2k} q(r^2)\psi_0(r^2) r^{s-1} dr \\
&\quad = 2\pi \prod_{m=1}^{s-2} \int_0^\pi \sin^m\phi d\phi \int_0^\infty r^{2k} q(r^2)\psi_0(r^2) r^{s-1} dr.
\end{aligned} \tag{26.9}$$

The substitution $y = r^2$ converts the last integral to

$$\int_0^\infty r^{2k} q(r^2)\psi_0(r^2) r^{s-1} dr = \frac{1}{2}\int_0^\infty y^k q(y)\psi_0(y) y^{(s-2)/2} dy, \tag{26.10}$$

and one can show that

$$\pi \prod_{m=1}^{s-2} \int_0^\pi \sin^m\phi d\phi = \frac{\pi^{s/2}}{\Gamma(s/2)}. \tag{26.11}$$

Combining (26.9), (26.10) and (26.11) gives us

$$\int_{\mathbb{R}^s} \|\boldsymbol{x}\|^{2k} q(\|\boldsymbol{x}\|^2)\psi_0(\|\boldsymbol{x}\|^2)dx = \frac{\pi^{s/2}}{\Gamma(s/2)} \int_0^\infty y^{k-1} q(y)\psi_0(y) y^{s/2} dy,$$

and therefore we now have obtained a set of one-dimensional orthogonality conditions

$$\frac{\pi^{s/2}}{\Gamma(s/2)} \int_0^\infty y^{k-1} q(y)\psi_0(y) y^{s/2} dy = \delta_{k0}, \qquad 0 \le k \le d \tag{26.12}$$

that can be used to determine the generating function

$$\Psi(\boldsymbol{x}) = \psi_0(\|\boldsymbol{x}\|^2)q(\|\boldsymbol{x}\|^2)$$

once we have chosen an initial weight ψ_0. Moreover, we know that this construction ensures Ψ to have approximate approximation order $\mathcal{O}(h^{2d+2})$.

Thus, the strategy for constructing generating functions for higher-order approximate MLS approxmation is as follows:

(1) Pick an arbitrary (univariate) weight function ψ_0.
(2) Compute the coefficients of $q \in \Pi_d^1$ via the (univariate) moment conditions (26.12).
(3) This leads to the (multivariate) radial generating function $\Psi(\boldsymbol{x}) = \psi_0(\|\boldsymbol{x}\|^2)q(\|\boldsymbol{x}\|^2)$ to be used in the quasi-interpolant (26.4).

Example 26.1. Probably the most aesthetic example is given by $\psi_0(y) = e^{-y}$ so that the basic generating function (corresponding to $d = 0$, *i.e.*, with approximate approximation order $\mathcal{O}(h^2)$ is found by assuming that the polynomial q is a constant, *i.e.*, $q(y) \equiv a_0$. Then (26.12) becomes

$$\int_0^\infty a_0 y^{(s-2)/2} e^{-y} dy = \frac{\Gamma(s/2)}{\pi^{s/2}},$$

which leads to $a_0 = \pi^{-s/2}$, so that we have

$$\Psi(\boldsymbol{x}) = \frac{1}{\sqrt{\pi^s}} e^{-\|\boldsymbol{x}\|^2}$$

— an appropriately scaled Gaussian.

If we take $d = 1$ (to obtain approximate approximation order $\mathcal{O}(h^4)$) and assume q to be a linear polynomial of the form $q(y) = a_0 + a_1 y$, then there are two moment conditions, namely

$$\int_0^\infty (a_0 + a_1 y) y^{(s-2)/2} e^{-y} dy = \frac{\Gamma(s/2)}{\pi^{s/2}},$$

$$\int_0^\infty (a_0 + a_1 y) y^{s/2} e^{-y} dy = 0,$$

or

$$\begin{aligned}
a_0\Gamma(s/2) + a_1\Gamma((s+2)/2) &= \frac{\Gamma(s/2)}{\pi^{s/2}}, \\
a_0\Gamma((s+2)/2) + a_1\Gamma((s+4)/2) &= 0.
\end{aligned}$$

We can solve this system of linear equations and obtain

$$a_0 = \frac{s+2}{2\sqrt{\pi^s}}, \qquad a_1 = \frac{-1}{\sqrt{\pi^s}}.$$

In the special case $s = 2$ this yields $a_0 = \frac{2}{\pi}$ and $a_1 = -\frac{1}{\pi}$, so that

$$\Psi(\boldsymbol{x}) = \frac{1}{\pi}\left(2 - \|\boldsymbol{x}\|^2\right) e^{-\|\boldsymbol{x}\|^2}.$$

In general, the polynomials q turn out to be (univariate) generalized Laguerre polynomials $L_d^{s/2}$ of degree d (*c.f.* Section 4.2) which are known to be orthogonal on the interval $[0, \infty)$ with respect to the weight function $y^{s/2}e^{-y}$. Therefore the generating functions for the approximate MLS approximation method are the Laguerre-Gaussians

$$\Psi(\boldsymbol{x}) = \frac{1}{\sqrt{\pi^s}} e^{-\|\boldsymbol{x}\|^2} L_d^{s/2}(\|\boldsymbol{x}\|^2).$$

In particular, Table 4.1 contains specific examples for $s = 1, 2, 3$ and $d = 1$ and 2 except for the scale factor $1/\sqrt{\pi^s}$. Figure 4.1 shows plots of the generating functions in the cases $s = 1$, $d = 2$, and $s = 2$, $d = 2$.

Example 26.2. If we use the function $\psi_0(y) = \left(1-\sqrt{y}\right)_+^4 (4\sqrt{y}+1)$ as initial weight, then we can perform calculations analogous to those above. For $d = 0$ and $s = 2$ we get

$$\Psi(\boldsymbol{x}) = \frac{7}{\pi}\psi_0(\|\boldsymbol{x}\|^2) = \frac{7}{\pi}\left(1-\|\boldsymbol{x}\|\right)_+^4 (4\|\boldsymbol{x}\|+1), \qquad \boldsymbol{x} \in \mathbb{R}^2, \tag{26.13}$$

with approximation order $\mathcal{O}(h^2)$. Except for the factor $7/\pi$ this generating function corresponds to Wendland's compactly supported C^2 radial basic function (*c.f.* Table 11.1).

Using the same function ψ_0 for $d = 1$ and $s = 2$ we obtain

$$\Psi(\boldsymbol{x}) = \psi_0(\|\boldsymbol{x}\|^2)\frac{7}{\pi}\left(\frac{504}{229} - \frac{1980}{229}\|\boldsymbol{x}\|^2\right) \tag{26.14}$$

with approximation order $\mathcal{O}(h^4)$. This function is displayed in the left plot of Figure 26.1. See Examples 27.2 and 27.4 for more special cases based on this initial weight function.

Example 26.3. Many other choices for the initial weight function ψ_0 are possible. For example, we can take $\psi_0(y) = (1-y)^\alpha(1+y)^\beta$, the weight function for univariate Jacobi polynomials (which are orthogonal on $[-1,1]$). Since the integral defining the orthogonality relations contains an extra factor of $y^{(2k+s-2)/2}$, the moment conditions (26.12) do not yield Jacobi polynomials. However, the resulting generating functions for approximate MLS approximation can still be rather simple. In Table 26.1 we list several such examples for $s = 2$, $\beta = 0$ and various combinations of d and α. Note that these functions are also compactly supported, *i.e.*, they are defined to be zero for $\|\boldsymbol{x}\| > 1$.

Table 26.1 Approximate MLS generating functions Ψ based on $\psi_0(y) = (1-y)^\alpha$, $y \in [-1,1]$ for various choices of d and α.

d	$\alpha = 2$	$\alpha = 5/2$
0	$\frac{3}{\pi}(1-\|\boldsymbol{x}\|^2)^2$	$\frac{7}{2\pi}(1-\|\boldsymbol{x}\|^2)^{5/2}$
1	$\frac{4}{\pi}\left(2-5\|\boldsymbol{x}\|^2\right)(1-\|\boldsymbol{x}\|^2)^2$	$\frac{9}{4\pi}\left(4-11\|\boldsymbol{x}\|^2\right)(1-\|\boldsymbol{x}\|^2)^{5/2}$
2	$\frac{15}{\pi}\left(1-6\|\boldsymbol{x}\|^2+7\|\boldsymbol{x}\|^4\right)(1-\|\boldsymbol{x}\|^2)^2$	$\frac{33}{16\pi}\left(8-52\|\boldsymbol{x}\|^2+65\|\boldsymbol{x}\|^4\right)(1-\|\boldsymbol{x}\|^2)^{5/2}$

The function $\Psi(\boldsymbol{x}) = \frac{4}{\pi}\left(2-5\|\boldsymbol{x}\|^2\right)(1-\|\boldsymbol{x}\|^2)^2$ is displayed in the right plot of Figure 26.1.

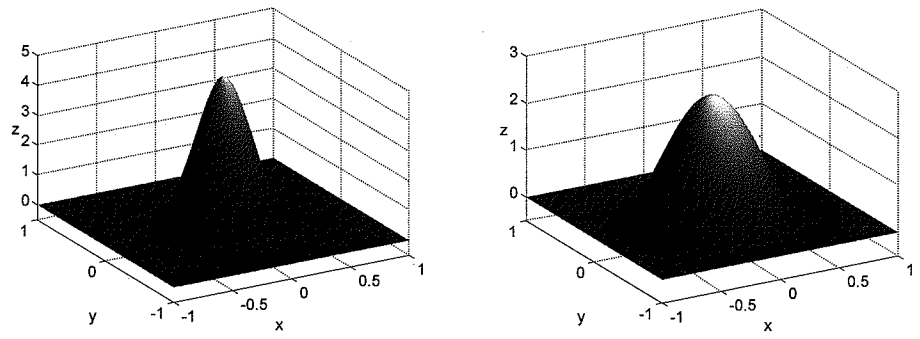

Fig. 26.1 Compactly supported generating functions for approximate linear reproduction. $\Psi(\boldsymbol{x}) = \frac{7}{\pi}\left(\frac{504}{229} - \frac{1980}{229}\|\boldsymbol{x}\|^2\right)(1-\|\boldsymbol{x}\|)_+^4 (4\|\boldsymbol{x}\|+1)$ (left) and $\Psi(\boldsymbol{x}) = \frac{4}{\pi}\left(2-5\|\boldsymbol{x}\|^2\right)(1-\|\boldsymbol{x}\|^2)^2$ (right) centered at the origin in $\mathbb{R}^2$.

Chapter 27

Numerical Experiments for Approximate MLS Approximation

In this chapter we present a series of experiments for approximate MLS approximation with both globally supported Laguerre-Gaussian generating functions as well as with compactly supported generating functions based on the initial weight $\psi_0(y) = \left(1-\sqrt{y}\right)_+^4 (4\sqrt{y}+1)$ as in Example 26.2 of the previous chapter.

27.1 Univariate Experiments

Example 27.1. We begin with univariate globally supported Laguerre-Gaussians. These functions are listed in Table 4.1 except for the scaling factor $1/\sqrt{\pi}$ required for the 1D case. In the left plot of Figure 27.1 we illustrate the effect the scaling parameter $\mathcal{D}$ has on the convergence behavior for Gaussian generating functions. We use a mollified univariate Franke-like function of the form

$$f(x) = 15e^{-\frac{1}{1-(2x-1)^2}}\left(\frac{3}{4}e^{-\frac{(9x-2)^2}{4}} + \frac{3}{4}e^{-\frac{(9x+1)^2}{49}} + \frac{1}{2}e^{-\frac{(9x-7)^2}{4}} - \frac{1}{5}e^{-(9x-4)^2}\right)$$

as test function. For each choice of $\mathcal{D} \in \{0.4, 0.8, 1.2, 1.6, 2.0\}$ we use a sequence of grids of $N = 2^k + 1$ (with $k = 1, \ldots, 14$) equally spaced points in $[0, 1]$ at which we sample the test function. The approximant is computed via

$$\mathcal{P}_f(x) = \frac{1}{\sqrt{\pi\mathcal{D}}} \sum_{i=1}^{N} f(x_i) e^{-\frac{(x-x_i)^2}{\mathcal{D}h^2}}, \qquad x \in [0, 1],$$

where $h = 1/(N-1)$. This corresponds to our usual shape parameter ε having a value of

$$\varepsilon = \frac{1}{\sqrt{\mathcal{D}}h} = \frac{N-1}{\sqrt{\mathcal{D}}} = \frac{2^k}{\sqrt{\mathcal{D}}},$$

i.e., we are in the regime of stationary approximation. The effect of $\mathcal{D}$ is clearly visible in the figure. A value of $\mathcal{D} \geq 2$ exhibits an approximation order of $\mathcal{O}(h^2)$ throughout the range of our experiments, while smaller values allow the saturation error to creep in at earlier stages.

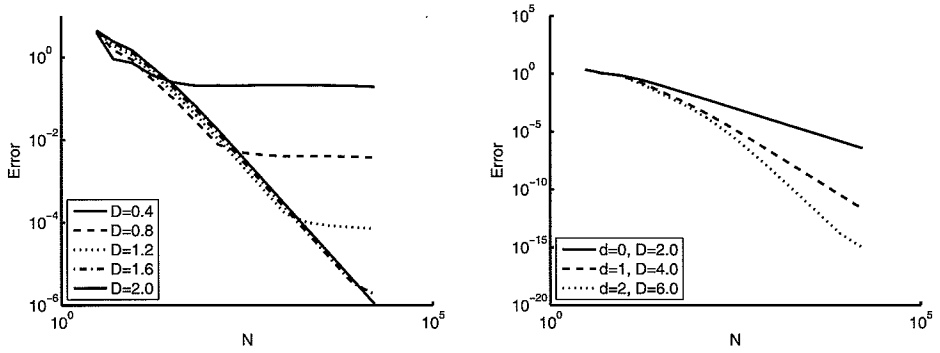

Fig. 27.1 Convergence of 1D approximate MLS approximation. The left plot shows the effect of various choices of $\mathcal{D}$ on the convergence behavior of Gaussians. The right plot illustrates the convergence of Laguerre-Gaussians for various values of d.

In the right plot of Figure 27.1 we compare the approximation orders achievable with the Laguerre-Gaussians of orders $d = 0, 1, 2$ in 1D. The respective values of $\mathcal{D}$ are $\mathcal{D} = 2, 4, 6$. The steepest sections of the curves correspond to approximate approximation orders of $\mathcal{O}(h^{2.0})$, $\mathcal{O}(h^{4.0})$, and $\mathcal{O}(h^{5.99})$, respectively — a perfect match with the rates predicted by the theory. Notice that for the second-order Laguerre-Gaussian we have convergence all the way to machine accuracy.

The MATLAB program `ApproxMLSApprox1D.m` (see Program 27.1) was used to generate the right plot in Figure 27.1. We define the three different Laguerre-Gaussian generating functions as members of a MATLAB cell array `rbf` and place the corresponding values of $\mathcal{D}$ to be used with each of the functions in the vector `D` (see lines 1–4). The univariate Franke-like test function is defined in lines 5–10. This function is mollified so that it goes to zero smoothly at the boundaries of the interval. The program contains two for-loops. The first is over the three different generating functions (corresponding to approximate constant, linear and quadratic reproduction, respectively). The inner loop performs a series of experiments with an increasing number N of data. Here we perform 14 iterations with N ranging from $N = 3$ to $N = 16385$.

For applications of approximate MLS approximation we limit ourselves to uniformly spaced data since there are presently no robust methods for dealing with nonuniform data (see [Lanzara *et al.* (2006); Maz'ya and Schmidt (2001)] for a theoretical approach to non-uniform data, and [Fasshauer (2004); Lanzara *et al.* (2006)] for some numerical experiments). All we need in order to compute the approximant is the evaluation matrix `EM` computed on line 23, which is then multiplied by the function values `f` and scaled by the factor $\mathcal{D}^{-s/2}$ on line 24. The commands needed to produce the plot are included on lines 15, 27 and 29–31.

Program 27.1. `ApproxMLSApprox1D.m`

```
% ApproxMLSApprox1D
% Script that performs 1D approximate MLS approximation
% Calls on: DistanceMatrix
    % Laguerre-Gaussians for 1D
 1  rbf{1} = @(e,r) exp(-(e*r).^2)/sqrt(pi);
 2  rbf{2} = @(e,r) exp(-(e*r).^2)/sqrt(pi).*(1.5-(e*r).^2);
 3a rbf{3} = @(e,r) exp(-(e*r).^2)/sqrt(pi).*...
 3b          (1.875-2.5*(e*r).^2+0.5*(e*r).^4);
 4  D = [2, 4, 6];   % Scale parameters for generating functions
    % Define Franke-like function as testfunction
 5  f1 = @(x) 0.75*exp(-(9*x-2).^2/4);
 6  f2 = @(x) 0.75*exp(-(9*x+1).^2/49);
 7  f3 = @(x) 0.5*exp(-(9*x-7).^2/4);
 8  f4 = @(x) 0.2*exp(-(9*x-4).^2);
 9  moll = @(x) 15*exp(-1./(1-4*(x-0.5).^2));
10  testfunction = @(x) moll(x).*(f1(x)+f2(x)+f3(x)-f4(x));
11  maxlevel = 14;   % number of iterations
12  M = 200;   % to create M evaluation points in unit interval
13  xe = linspace(0,1,M); epoints = xe(:);
14  exact = testfunction(epoints);
15  figure; hold on; cword = cellstr(['r- ';'g--';'b: ']);
16  for i=1:length(D)
17     for k=1:maxlevel
18        N(k) = (2^k+1); ep = (N(k)-1)/sqrt(D(i));
19        name = sprintf('Data1D_%du', N(k));  load(name);
20        ctrs = dsites;
          % Create vector of function values
21        f = testfunction(dsites);
          % Compute evaluation matrix
22        DM = DistanceMatrix(epoints,ctrs);
23        EM = rbf{i}(ep,DM);
          % Compute approximate MLS approximation
24        Pf = EM*f/sqrt(D(i));
          % Compute RMS error on evaluation grid
25        rms_err(k) = norm(Pf-exact)/sqrt(M);
26     end
27     plot(N,rms_err,cword{i});
28  end
29  set(gca,'XScale','log','YScale','log','Fontsize',14)
30  legend('d=0, D=2.0','d=1, D=4.0','d=2, D=6.0',3);
31  xlabel('N');  ylabel('Error');  hold off
```

Example 27.2. In the second set of experiments we use compactly supported generating functions with initial weight $\psi_0(y) = \left(1-\sqrt{y}\right)_+^4 (4\sqrt{y}+1)$. In the univariate case these functions are for $d = 0, 1, 2$

$$\begin{aligned}
\Psi(x) &= \frac{3}{2}\left(1-|x|\right)_+^4 (4|x|+1)\,,\\
\Psi(x) &= \frac{1}{3}\left(7-35|x|^2\right)\left(1-|x|\right)_+^4 (4|x|+1)\,, \qquad (27.1)\\
\Psi(x) &= \frac{105}{11993}\left(350-3960|x|^2+7293|x|^4\right)\left(1-|x|\right)_+^4 (4|x|+1)\,.
\end{aligned}$$

These functions can be computed as in Section 26.3 of the previous chapter. An approximate approximation order of $\mathcal{O}(h^2)$ for the first function in (27.1) is illustrated in the left plot of Figure 27.2. Note that a rather large value of $\mathcal{D}$ (namely $\mathcal{D} \approx 500$) is required to make the saturation error so small that it no longer affects our experiments. Since the shape parameter ε determines the support radius of our generating functions, and since $\varepsilon = 1/\sqrt{\mathcal{D}}h = (N-1)/\sqrt{\mathcal{D}}$, we see that the evaluation matrix will be completely dense until N grows above approximately 25. Furthermore, for such a large value of $\mathcal{D}$ there is a visible smoothing effect (very slow convergence) during the first few iterations with $N = 3, 5, 9, 17, 33, 65$.

In the right plot of Figure 27.2 we compare the approximation orders achievable with the univariate compactly supported generating functions (27.1) of orders $d = 0, 1, 2$. The respective values of $\mathcal{D}$ required to prevent the saturation error from corrupting our experiments are $\mathcal{D} = 500, 5000, 20000$, respectively. Note that $\mathcal{D} = 20000$ implies that the evaluation matrix will be dense (and therefore computationally inefficient) until N reaches about 150. Thus, it is not until the very last iterations with $N = 8193$ and $N = 16385$ that we get to take real advantage of the compact support of the generating function, *i.e.*, have sparse sums. The steepest sections of the curves correspond to approximate approximation orders of $\mathcal{O}(h^{2.0})$, $\mathcal{O}(h^{3.91})$, and $\mathcal{O}(h^{5.45})$, respectively.

The MATLAB code for the compactly supported experiments can be written in two ways. One possibility would be to rewrite the generating functions in shifted form as explained in Chapter 12, and then use the sparse code `DistanceMatrixCSRBF.m`. However, as just explained, we cannot take much advantage of the sparsity usually associated with compactly supported functions. Therefore, we use essentially the same code as in Program 27.1. The only changes needed are the substitution of the definition of the compactly supported generating functions for the Laguerre-Gaussians along with appropriate values for `D`.

Our experiments seem to suggest that there is no point in using compactly supported generating functions for univariate approximate MLS approximation. The results using Laguerre-Gaussians are far more accurate, and due to the large value of $\mathcal{D}$ required for the compactly supported functions they do not offer an advantage in terms of computational complexity, either.

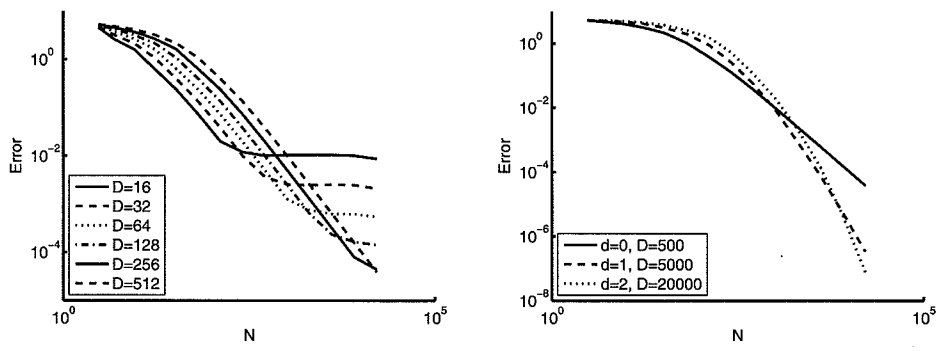

Fig. 27.2 Convergence of 1D approximate MLS approximation with compactly supported generating functions. The left plot shows the effect of various choices of $\mathcal{D}$ on the convergence behavior for the first function in (27.1). The right plot illustrates the convergence for the three functions in (27.1).

27.2 Bivariate Experiments

Example 27.3. This example is similar to the second part of Example 27.1. Now we use bivariate Laguerre-Gaussians with scale factor $1/\pi$. The test data are sampled from a bivariate mollified Franke function, *i.e.*, we multiply Franke's function (2.2) by the mollifier

$$g(x,y) = 15e^{-\frac{1}{1-(2x-1)^2}} e^{-\frac{1}{1-(2y-1)^2}}.$$

The data sites are uniform grids of $(2^k+1)^2$ points (with $k = 1, \ldots, 5$) in the unit square. The values for the scale parameter $\mathcal{D}$ used are $\mathcal{D} = 1, 2, 2.5$. The steepest sections of the error curves correspond to approximate approximation orders of $\mathcal{O}(h^{1.83})$, $\mathcal{O}(h^{2.80})$, and $\mathcal{O}(h^{3.00})$, respectively. Note that these rates do not match the theoretically predicted orders. We will see that we can achieve the theoretically predicted orders by using more data sites. This, however, will require special evaluation techniques to deal with the large sums efficiently (see the next chapter).

Example 27.4. The bivariate compactly supported generating functions with initial weight $\psi_0(y) = \left(1-\sqrt{y}\right)_+^4 (4\sqrt{y}+1)$ providing approximate reproduction of constants, linear and quadratic polynomials, respectively, are (*c.f.* Example 26.2)

$$\begin{aligned}
\Psi(\boldsymbol{x}) &= \frac{7}{\pi}\left(1-\|\boldsymbol{x}\|\right)_+^4 (4\|\boldsymbol{x}\|+1)\,,\\
\Psi(\boldsymbol{x}) &= \frac{252}{229\pi}\left(14-55\|\boldsymbol{x}\|^2\right)\left(1-\|\boldsymbol{x}\|\right)_+^4 (4\|\boldsymbol{x}\|+1)\,,\\
\Psi(\boldsymbol{x}) &= \frac{693}{112901\pi}\left(4179-37050\|\boldsymbol{x}\|^2+59605\|\boldsymbol{x}\|^4\right)\left(1-\|\boldsymbol{x}\|\right)_+^4 (4\|\boldsymbol{x}\|+1)\,.
\end{aligned}$$

The values we use for the scale parameter are $\mathcal{D} = 20, 40, 80$, respectively. The steepest sections of the error curves correspond to approximate approximation or-

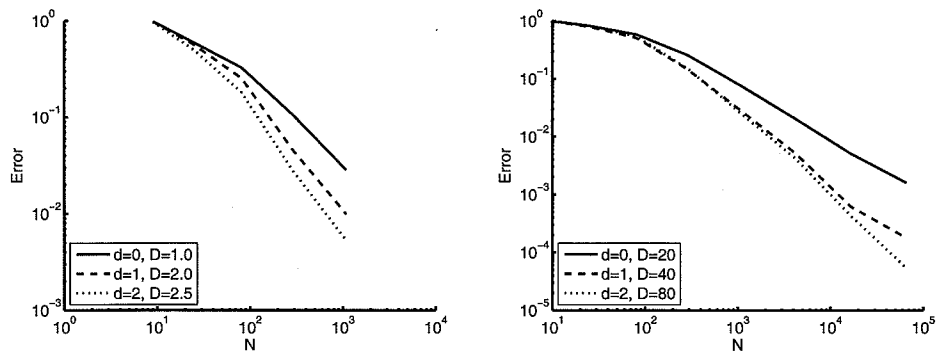

Fig. 27.3 Convergence of 2D approximate MLS approximation with Laguerre-Gaussians (left) and compactly supported (right) generating functions for various values of d.

ders of $\mathcal{O}(h^{1.96})$, $\mathcal{O}(h^{3.03})$, and $\mathcal{O}(h^{3.26})$, respectively. Again, we do not obtain a match with the theoretically predicted orders, even though we used up to $N = 66049$ data points.

The bivariate case provides a more level playing field for the compactly supported generating functions. We can take advantage of the compact support and obtain more accurate results at a reasonable cost. However, another alternative way of obtaining highly accurate multivariate approximate MLS approximations is presented in the next chapter.

Chapter 28

Fast Fourier Transforms

28.1 NFFT

In the recent papers [Kunis *et al.* (2002); Nieslony *et al.* (2004); Potts and Steidl (2003)] use of the *fast Fourier transform for non-uniformly spaced points* was suggested as an efficient way to solve and evaluate radial basis function problems. The C++ software package NFFT by the authors is available for free download [Kunis and Potts (2002)]. A discussion of the actual NFFT software would go beyond the scope of this book. Instead, we briefly describe how to use NFFTs and FFTs to simultaneously evaluate expansions of the form

$$\mathcal{P}_f(\boldsymbol{y}_j) = \sum_{k=1}^{N} c_k \Phi(\boldsymbol{y}_j - \boldsymbol{x}_k) \tag{28.1}$$

at many evaluation points $\boldsymbol{y}_j$, $j = 1, \ldots, M$. Note that this covers not only approximate MLS approximations, but also the evaluation of other quasi-interpolants as well as the evaluation of RBF interpolants.

Direct summation of (28.1) requires $\mathcal{O}(MN)$ operations, while it can be shown that use of the NFFT reduces the cost to $\mathcal{O}(M + N)$ operations. Therefore, as is always the case with fast Fourier transforms, use of the algorithm will pay off for sufficiently many evaluations.

In their papers Nieslony, Potts and Steidl distinguish between basic functions Φ that are singular and those that are non-singular. Singular basic functions are C^∞ everywhere except at the origin and include examples such as

$$\frac{1}{r}, \ \frac{1}{r^2}, \ \log r, \ r^2 \log r,$$

where $r = \|\cdot\|$. Non-singular basic functions are smooth everywhere such as Gaussians and (inverse) multiquadrics. We will restrict our discussion to this latter class.

The basic idea for the following algorithm is remarkably simple. It relies on the fact that the exponential $e^{-\alpha(\boldsymbol{y}_j - \boldsymbol{x}_k)}$ can be written as $e^{-\alpha \boldsymbol{y}_j} e^{\alpha \boldsymbol{x}_k}$. Moreover, the method applies to arbitrary basic functions (which is in strong contrast to the fast

multipole type methods discussed in Chapter 35. One starts out by approximating the (arbitrary, but smooth) basic function Φ using standard Fourier series, *i.e.*,

$$\Phi(\boldsymbol{x}) \approx \sum_{\boldsymbol{\ell} \in I_n} b_{\boldsymbol{\ell}} e^{2\pi i \boldsymbol{\ell} \cdot \boldsymbol{x}} \tag{28.2}$$

with $\boldsymbol{\ell}$ a multi-index in the index set $I_n = \left[-\frac{n}{2}, \frac{n}{2}\right)^s$. The coefficients $b_{\boldsymbol{\ell}}$ are found by the discrete inverse Fourier transform

$$b_{\boldsymbol{\ell}} = \frac{1}{n^s} \sum_{\boldsymbol{k} \in I_n} \Phi\left(\frac{\boldsymbol{k}}{n}\right) e^{-2\pi i \boldsymbol{k} \cdot \boldsymbol{\ell}/n}. \tag{28.3}$$

Numerically, this task is accomplished with software for the standard (inverse) FFT (*e.g.*, [FFTW]).

Remark 28.1. Note that this definition of the Fourier transform (as well as the one below) is different from the one used throughout the rest of this book. However, in order to stay closer to the software packages, we adopt the notation used there.

Using the representation (28.2) of the basic function Φ we can rewrite (28.1) as

$$\begin{aligned} \mathcal{P}_f(\boldsymbol{y}_j) &\approx \sum_{k=1}^{N} c_k \sum_{\boldsymbol{\ell} \in I_n} b_{\boldsymbol{\ell}} e^{2\pi i \boldsymbol{\ell} \cdot (\boldsymbol{y}_j - \boldsymbol{x}_k)} \\ &= \sum_{\boldsymbol{\ell} \in I_n} b_{\boldsymbol{\ell}} \sum_{k=1}^{N} c_k e^{2\pi i \boldsymbol{\ell} \cdot (\boldsymbol{y}_j - \boldsymbol{x}_k)} \end{aligned}$$

Now, the exponential is split using the above mentioned property, *i.e.*,

$$\mathcal{P}_f(\boldsymbol{y}_j) \approx \sum_{\boldsymbol{\ell} \in I_n} b_{\boldsymbol{\ell}} \sum_{k=1}^{N} c_k e^{-2\pi i \boldsymbol{\ell} \cdot \boldsymbol{x}_k} e^{2\pi i \boldsymbol{\ell} \cdot \boldsymbol{y}_j}.$$

This, however, can be viewed as a fast Fourier transform at the non-uniformly spaced points $\boldsymbol{y}_j$, *i.e.*,

$$\mathcal{P}_f(\boldsymbol{y}_j) \approx \sum_{\boldsymbol{\ell} \in I_n} d_{\boldsymbol{\ell}} e^{2\pi i \boldsymbol{\ell} \cdot \boldsymbol{y}_j}.$$

where the coefficients $d_{\boldsymbol{\ell}} = b_{\boldsymbol{\ell}} a_{\boldsymbol{\ell}}$ with

$$a_{\boldsymbol{\ell}} = \sum_{k=1}^{N} c_k e^{-2\pi i \boldsymbol{\ell} \cdot \boldsymbol{x}_k},$$

which in turn is nothing but an inverse discrete Fourier transform at the non-uniformly spaced points $\boldsymbol{x}_k$. These latter two transforms are dealt with numerically using the NFFT software.

Together, for the case of non-singular basic functions Φ, we have the following algorithm.

Algorithm 28.1. Fast Fourier transform evaluation

For $\boldsymbol{\ell} \in I_n$

Compute the coefficients

$$b_{\boldsymbol{\ell}} = \frac{1}{n^s} \sum_{\boldsymbol{k} \in I_n} \Phi\left(\frac{\boldsymbol{k}}{n}\right) e^{-2\pi i \boldsymbol{k} \cdot \boldsymbol{\ell}/n}$$

by inverse FFT.

Compute the coefficients

$$a_{\boldsymbol{\ell}} = \sum_{k=1}^{N} c_k e^{-2\pi i \boldsymbol{\ell} \cdot \boldsymbol{x}_k}$$

by inverse NFFT.

Compute the coefficients $d_{\boldsymbol{\ell}} = a_{\boldsymbol{\ell}} b_{\boldsymbol{\ell}}$.

end

For $1 \le j \le M$

Compute the values

$$\mathcal{P}_f(\boldsymbol{y}_j) \approx \sum_{\boldsymbol{\ell} \in I_n} d_{\boldsymbol{\ell}} e^{2\pi i \boldsymbol{\ell} \cdot \boldsymbol{y}_j}$$

by NFFT.

end

In the papers [Kunis *et al.* (2002); Nieslony *et al.* (2004); Potts and Steidl (2003)] the authors suggest a special boundary regularization in case the basic function does not decay fast enough, *i.e.*, the basic function is large near the boundary of the domain. However, for our experiments with Laguerre-Gaussians reported in the next section this is not an issue.

Of course, this method will only provide an approximation of the expansion (28.1) and error estimates are provided in the literature (see, *e.g.*, [Nieslony *et al.* (2004)]).

While we only illustrate the use of (N)FFTs for the evaluation of radial sums it should be clear that this method can also be coupled with any other algorithm that is based on evaluation of fast summation at non-uniform points (such as the preconditioned GMRES algorithm of Section 34.3, the "greedy" algorithm of Section 33.1, or the Faul-Powell algorithm of Section 33.2).

28.2 Approximate MLS Approximation via Non-uniform Fast Fourier Transforms

A few examples that illustrate the use of fast Fourier transforms for the evaluation of approximate moving least squares approximations (quasi-interpolants) are

taken from the paper [Fasshauer and Zhang (2004)]. We deviate from our usual policy of providing only results of experiments based on MATLAB code and include Figures 28.1–28.3, which were obtained with C++ software incorporating the NFFT library.

We use the following mollified Franke-type function in the unit cube $[0,1]^s$ in space dimensions $s = 1, 2, 3$:

$$\begin{aligned} f(x_1,x_2,x_3) = \frac{3}{4}\Big[&\exp(-\frac{(9x_1-2)^2}{4} - \frac{(9x_2-2)^2}{4} - \frac{(9x_3-2)^2}{4}) \\ &+ \exp(-\frac{(9x_1+1)^2}{49} - \frac{(9x_2+1)^2}{10} - \frac{(9x_3+1)^2}{29})\Big] \\ &+ \frac{1}{2}\exp(-\frac{(9x_1-7)^2}{4} - (9x_2-3)^2 - \frac{(9x_3-5)^2}{2}) \\ &- \frac{1}{5}\exp(-(9x_1-4)^2 - (9x_2-7)^2 - (9x_3-5)^2), \end{aligned}$$

$$g(x_1,x_2,x_3) = 15 f(x_1,x_2,x_3) \prod_{i=1}^{3} \exp(\frac{-1}{1-(2x_i-1)^2}),$$

where x_1, x_2, x_3 are used according to the space dimension s. The data sites are $N = (2^k+1)^s$ equally spaced points in the unit cube, while the errors are computed at M evaluation points randomly distributed in $[0,1]^s$ with $M = 32768$ for $s = 1$, $M = 262144$ for $s = 2$, and $M = 2146689$ for $s = 3$.

In our experiments we use $n = 4N^{1/s}$ in (28.3) for all computations except for the very last experiments in 2D and 3D. We do not have an automated strategy for choosing n. However, the values just mentioned yield satisfactory results and go along with the values suggested by Theorems 3.1 and 3.4 of [Nieslony *et al.* (2004)]. In all experiments displayed in Figures 28.1–28.3, the scale parameter $\mathcal{D}$ is taken to be 3.0.

The left plots in Figures 28.1–28.3 show the maximum error versus the number of centers N on a logarithmic scale for the three types of Laguerre-Gaussian generating functions of Table 4.1. This illustrates that the approximation does converge well (almost reaching the rates predicted by the theory) as we increase the number of data sites.

The presence of the saturation error is clearly visible in Figure 28.1. The plots on the right compare the cost of direct summation versus that for NFFT summation, and show that the efficiency is greatly improved by the use of the NFFT. Due to their long duration some of the computational times for the direct summation were omitted.

The experiments presented in this section show that it is not unreasonable to approximate large data sets with globally supported generating functions. In fact, the accuracy achieved with the global functions is far superior to that which we obtained with the compactly supported functions for similar problems in the previous chapter.

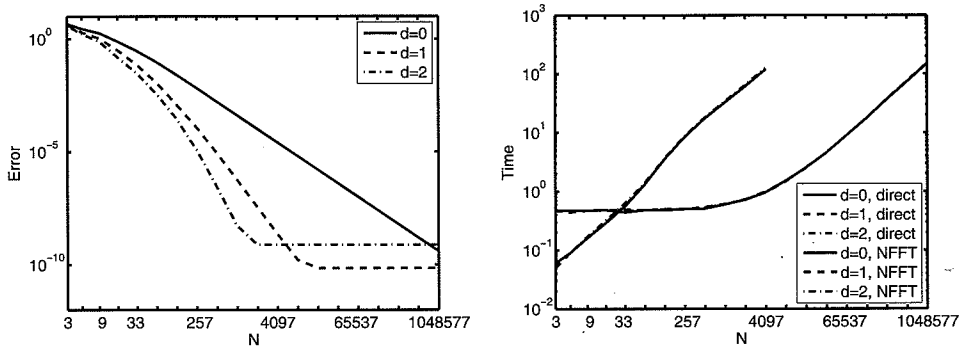

Fig. 28.1 Convergence and execution times for 1D example (Gaussian, linear and quadratic Laguerre-Gaussian generating functions).

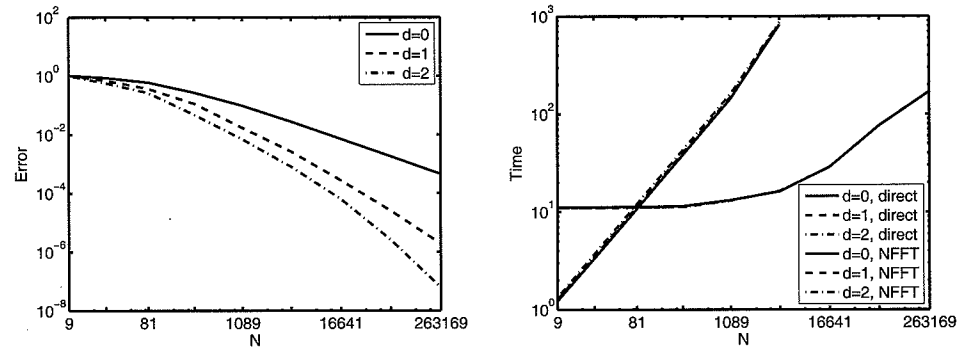

Fig. 28.2 Convergence and execution times for 2D example (Gaussian, linear and quadratic Laguerre-Gaussian generating functions).

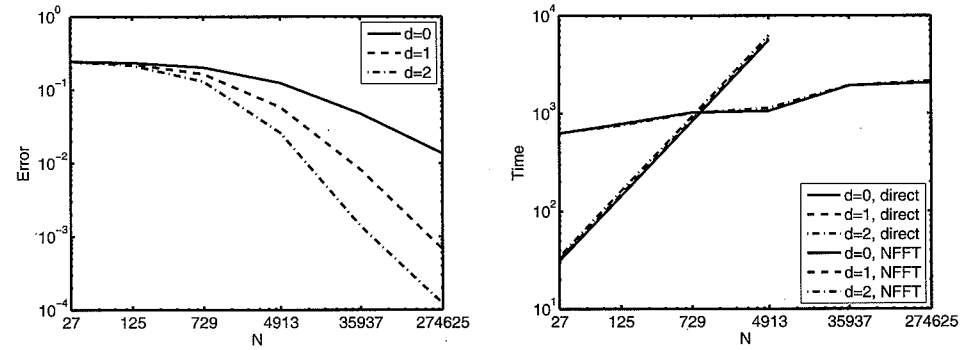

Fig. 28.3 Convergence and (predicted) execution times for 3D example (Gaussian, linear and quadratic Laguerre-Gaussian generating functions).

Chapter 29

Partition of Unity Methods

Another possibility for fast computation with meshfree approximation methods is the *partition of unity* method. This approach offers a simple way to decompose a large problem into many small problems while at the same time ensuring that the accuracy obtained for the local fits is carried over to the global fit.

29.1 Theory

The partition of unity method was suggested in [Babuška and Melenk (1997); Melenk and Babuška (1996)] in the mid 1990s in the context of meshfree Galerkin methods for the solution of partial differential equations (see Chapters 44 and 45 for a discussion of an RBF-based Galerkin approach). In the scattered data fitting context the paper [Franke (1977)] already contains a similar algorithm. We base the presentation in this section on the paper [Wendland (2002a)].

The basic idea for the partition of unity method is to start with a partition of the open and bounded domain $\Omega \subseteq \mathbb{R}^s$ into M subdomains Ω_j such that $\bigcup_{j=1}^{M} \Omega_j \supseteq \Omega$ with some mild overlap among the subdomains. Associated with these subdomains we choose a *partition of unity*, *i.e.*, a family of compactly supported, non-negative, continuous functions w_j supported on the closure of Ω_j such that at every point $\boldsymbol{x}$ in Ω we have

$$\sum_{j=1}^{M} w_j(\boldsymbol{x}) = 1. \tag{29.1}$$

Now, for every subdomain Ω_j we construct a local approximation u_j (*e.g.*, a radial basis function interpolant), and then form the global approximant to the data on the entire domain Ω via

$$\mathcal{P}_f(\boldsymbol{x}) = \sum_{j=1}^{M} u_j(\boldsymbol{x}) w_j(\boldsymbol{x}), \qquad \boldsymbol{x} \in \Omega. \tag{29.2}$$

Note that if the local fits interpolate at a given data point $\boldsymbol{x}_\ell$, *i.e.*, $u_j(\boldsymbol{x}_\ell) = f(\boldsymbol{x}_\ell)$,

then the global approximant also interpolates at this point:

$$\begin{aligned}\mathcal{P}_f(\boldsymbol{x}_\ell) &= \sum_{j=1}^{M} u_j(\boldsymbol{x}_\ell) w_j(\boldsymbol{x}_\ell) \\ &= \sum_{j=1}^{M} f(\boldsymbol{x}_\ell) w_j(\boldsymbol{x}_\ell) = f(\boldsymbol{x}_\ell).\end{aligned}$$

The last equality holds due to the partition of unity property (29.1). Technically, the second sum will most likely not run over the full set of indices, $1, \ldots, M$, since $\boldsymbol{x}_\ell$ will lie only in the support of some of the w_j. Of course, this does not change the result.

In order to be able to formulate error bounds we need some technical conditions. We require the partition of unity functions to be *k-stable*, *i.e.*, we require that each $w_j \in C^k(\mathbb{R}^s)$ satisfies for every multi-index $\boldsymbol{\alpha}$ with $|\boldsymbol{\alpha}| \leq k$ the inequality

$$\|D^{\boldsymbol{\alpha}} w_j\|_{L_\infty(\Omega_j)} \leq C_{\boldsymbol{\alpha}} / \delta_j^{|\boldsymbol{\alpha}|},$$

where $C_{\boldsymbol{\alpha}}$ is some positive constant, and $\delta_j = \operatorname{diam}(\Omega_j)$.

In order to understand the following approximation theorem from [Wendland (2002a)] we need to define the space $C^k_\beta(\mathbb{R}^s)$ of all C^k functions f whose derivatives of order $|\boldsymbol{\alpha}| = k$ satisfy $D^{\boldsymbol{\alpha}} f(\boldsymbol{x}) = \mathcal{O}(\|\boldsymbol{x}\|_2^\beta)$ for $\|\boldsymbol{x}\|_2 \to 0$.

Theorem 29.1. *Suppose $\Omega \subseteq \mathbb{R}^s$ is open and bounded, and let $\mathcal{X} = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N\} \subseteq \Omega$. Let $\Phi \in C^k_\beta(\mathbb{R}^s)$ be strictly conditionally positive definite of order m. Let $\{\Omega_j\}$ be a regular covering for $(\Omega, \mathcal{X})$ and let $\{w_j\}$ be k-stable for $\{\Omega_j\}$. Then the error between $f \in \mathcal{N}_\Phi(\Omega)$ and its partition of unity interpolant (29.2) with $u_j \in \operatorname{span}\{\Phi(\cdot, \boldsymbol{x}) : \boldsymbol{x} \in \mathcal{X} \cap \Omega_j\} + \Pi^s_{m-1}$ can be bounded by*

$$|D^{\boldsymbol{\alpha}} f(\boldsymbol{x}) - D^{\boldsymbol{\alpha}} \mathcal{P}_f(\boldsymbol{x})| \leq C h_{\mathcal{X},\Omega}^{\frac{k+\beta}{2} - |\boldsymbol{\alpha}|} |f|_{\mathcal{N}_\Phi(\Omega)},$$

for all $\boldsymbol{x} \in \Omega$ and all $|\boldsymbol{\alpha}| \leq k/2$.

The regularity assumptions on the subdomains Ω_j are:

- For every $\boldsymbol{x} \in \Omega$ the number of subdomains Ω_j with $\boldsymbol{x} \in \Omega_j$ is bounded by a global constant K.
- Every subdomain Ω_j satisfies an interior cone condition (*c.f.* Definition 14.2).
- The local fill distances $h_{\mathcal{X}_j,\Omega_j}$ are uniformly bounded by the global fill distance $h_{\mathcal{X},\Omega}$, where $\mathcal{X}_j = \mathcal{X} \cap \Omega_j$.

If we compare this with the global error estimates from Chapter 15 we see that the partition of unity preserves the local approximation order for the global fit. Thus, we can efficiently compute large RBF interpolants by solving many small RBF interpolation problems (in parallel if we wish) and then glue them together with the global partition of unity $\{w_j\}$.

A simple way to obtain a partition of unity is via a Shepard approximant (*c.f.* Chapter 23). Therefore, we can think of the partition of unity method as a Shepard method with higher-order data. Namely, the "data" are now given by the local approximations u_j instead of just the values $f(\boldsymbol{x}_j)$. The benefits of this kind of approach seem to have first been realized in [Franke (1977)].

29.2 Partition of Unity Approximation with MATLAB

The MATLAB program for the partition of unity approximation based on local RBF interpolants is again rather similar to our earlier programs. The main difference is that we now also have to create the subdomains Ω_j (which we will do as overlapping circles) and the associated partition of unity $\{w_j\}$ (for which we use a Shepard method based on Wendland's compactly supported RBFs) .

The compactly supported radial weight functions on line 1 of Program 29.1 and the RBF on line 2 are used in the shifted form $\widetilde{\varphi}_{s,k} = \varphi_{s,k}(1 - \cdot)$ (cf. Table 11.1). Note that we use the *kd*-tree routines to build two trees: a data tree, and an evaluation tree. Inside the loop over all partition of unity cells (lines 27–38) we first use `kdrangequery` to find all data sites in cell j, build the local interpolation matrix based on these points, and then repeat the process for the evaluation points. Note that the contributions to the final global fit are accumulated cell by cell (see line 36).

Program 29.1. `PU2D_CS.m`

```
% PU2D_CS
% Script that performs partition of unity approximation using
% sparse matrices
% Calls on: DistanceMatrixCSRBF
% Uses:     k-D tree package by Guy Shechter
%           from MATLAB Central File Exchange
    % Weight function for global Shepard partition of unity weighting
 1  wf = @(e,r) r.^4.*(5*spones(r)-4*r);
    % RBF basis function for local RBF interpolation
 2  rbf = @(e,r) r.^4.*(5*spones(r)-4*r);
 3  ep = 0.1; % Parameter for local basis functions
    % Define Franke's function as testfunction
 4  f1 = @(x,y) 0.75*exp(-((9*x-2).^2+(9*y-2).^2)/4);
 5  f2 = @(x,y) 0.75*exp(-((9*x+1).^2/49+(9*y+1).^2/10));
 6  f3 = @(x,y) 0.5*exp(-((9*x-7).^2+(9*y-3).^2)/4);
 7  f4 = @(x,y) 0.2*exp(-((9*x-4).^2+(9*y-7).^2));
 8  testfunction = @(x,y) f1(x,y)+f2(x,y)+f3(x,y)-f4(x,y);
 9  N = 1089; gridtype = 'h';
    % Parameter for npu-by-npu grid of PU cells in unit square
10  npu = 16;
    % Parameter for neval-by-neval evaluation grid in unit square
```

```
11  neval = 40;
    % Load data points
12  name = sprintf('Data2D_%d%s',N,gridtype); load(name)
13  ctrs = dsites;
14  rhs = testfunction(dsites(:,1),dsites(:,2));
15  wep = npu; % Parameter for weight function
    % Create neval-by-neval equally spaced evaluation locations
    % in the unit square
16  grid = linspace(0,1,neval); [xe,ye] = meshgrid(grid);
17  epoints = [xe(:) ye(:)];
    % Create npu-by-npu equally spaced centers of PU cells in the
    % unit square
18  pugrid = linspace(0,1,npu); [xpu,ypu] = meshgrid(pugrid);
19  cellctrs = [xpu(:) ypu(:)];
20  cellradius = 1/wep;
    % Compute Shepard evaluation matrix
21  DM_eval = DistanceMatrixCSRBF(epoints,cellctrs,wep);
22  SEM = wf(ep,DM_eval);
23  SEM = spdiags(1./(SEM*ones(npu^2,1)),0,neval^2,neval^2)*SEM;
    % Build k-D trees for data sites and evaluation points
24  [tmp,tmp,datatree] = kdtree(dsites,[]);
25  [tmp,tmp,evaltree] = kdtree(epoints,[]);
26  Pf = zeros(neval^2,1);  % initialize
27  for j=1:npu^2
    % Find data sites in cell j
28a   [pts,dist,idx] = kdrangequery(datatree,...
28b                   cellctrs(j,:),cellradius);
29    if (length(idx) > 0)
    % Build local interpolation matrix for cell j
30a     DM_data = DistanceMatrixCSRBF(dsites(idx,:),...
30b                           ctrs(idx,:),ep);
31      IM = rbf(ep,DM_data);
    % Find evaluation points in cell j
32a     [epts,edist,eidx] = kdrangequery(evaltree,...
32b                           cellctrs(j,:),cellradius);
    % Compute local evaluation matrix
33a     DM_eval = DistanceMatrixCSRBF(epoints(eidx,:),...
33b                           ctrs(idx,:),ep);
34      EM = rbf(ep,DM_eval);
    % Compute local RBF interpolant
35      localfit = EM * (IM\rhs(idx));
    % Accumulate global fit
```



```
36      Pf(eidx) = Pf(eidx) + localfit.*SEM(eidx,j);
37    end
38  end
    % Compute exact solution
39  exact = testfunction(epoints(:,1),epoints(:,2));
    % Compute maximum error on evaluation grid
40  maxerr = norm(Pf-exact,inf);
41  rms_err = norm(Pf-exact)/neval;
42  fprintf('RMS error:     %e\n', rms_err)
43  fprintf('Maximum error: %e\n', maxerr)
    % Plot interpolant
44  fview = [160,20];
45  PlotSurf(xe,ye,Pf,neval,exact,maxerr,fview);
    % Plot maximum error
46  PlotError2D(xe,ye,Pf,exact,maxerr,neval,fview);
```

In Tables 29.1 and 29.2 we illustrate how the local convergence order is maintained globally for a partition of unity based on Wendland's C^2 compactly supported RBFs. In both tables the local approximations are computed via either the compactly supported C^2 functions of Wendland, or via Gaussians (that are globally supported on the local subdomains). The data were sampled from Franke's function at various sets of uniformly spaced points (in Table 29.1) and Halton points (in Table 29.2) in the unit square. The M local subdomains were given by circles centered at equally spaced points in the unit square. We can see that (especially on the uniformly spaced data sites) the partition of unity method reflects the approximation behavior of the local methods. For the compactly supported Wendland functions we obtain $\mathcal{O}(h^2)$ throughout our series of experiments (with a theoretically predicted local order of $\mathcal{O}(h^{3/2})$), whereas for the Gaussians we obtain an approximation behavior in places vaguely suggestive of exponential convergence. For these experiments the fill distances for the sets of Halton points were not estimated via (2.4). Instead, we assumed that the fill distance decreases by a factor of two from one iteration to the next, as it does in the case of uniformly distributed points.

A relatively large uniform value of the shape parameter ε was used for the Gaussians on the uniform data sets in Table 29.1 to obtain the exponential convergence results. Use of the same value of ε on the Halton data sets results in RMS-errors for the Gaussians that are *worse* than those for local interpolants based on compactly supported RBFs (see Table 29.2). For the local interpolants based on the Wendland functions we used a large support radius of $\rho = 1/\varepsilon = 10$ in accordance to the observations made in Chapter 17. The main reason for the relatively poor quality of the local interpolants based on Gaussians in the Halton setting is that all of the computations with the Wendland functions are performed with well-conditioned interpolation matrices (in spite of the large support radius, *i.e.*, flat basis functions).

Note that the RMS-error for local Gaussian interpolation in the last row of

Table 29.1 2D partition of unity approximation on uniform points with Wendland's C^2 compactly supported functions for partition of unity and local Wendland and Gaussian interpolants.

		Wendland			Gaussian		
N	M	RMS-error	rate	ε	RMS-error	rate	ε
9	1	3.475816e-001		0.1	3.703960e-001		6
25	4	1.301854e-001	1.4168	0.1	1.229046e-001	1.5915	6
81	16	8.089165e-003	4.0084	0.1	2.413852e-002	2.3481	6
289	64	1.183369e-003	2.7731	0.1	4.551325e-003	2.4070	6
1089	256	2.716542e-004	2.1231	0.1	5.393771e-004	3.0769	6
4225	1024	6.795949e-005	1.9990	0.1	1.112507e-005	5.5994	6
16641	4096	1.697195e-005	2.0015	0.1	2.031757e-006	2.4530	6
66049	16384	4.241184e-006	2.0006	0.1	1.798274e-007	3.4980	6
263169	65536	9.778231e-007	2.1168	0.1	2.657751e-008	2.7583	6
1050625	262144	2.557991e-007	1.9346	0.1	1.844820e-009	3.8487	6

Table 29.2 2D partition of unity approximation on Halton points with Wendland's C^2 compactly supported functions for partition of unity and local Wendland and Gaussian interpolants.

		Wendland			Gaussian		
N	M	RMS-error	rate	ε	RMS-error	rate	ε
9	1	3.325975e-001		0.1	3.384117e-001		6
25	4	1.355396e-001	1.2951	0.1	1.383321e-001	1.2906	6
81	16	1.035963e-002	3.7097	0.1	3.640953e-002	1.9257	6
289	64	2.569458e-003	2.0114	0.1	1.030984e-002	1.8203	6
1089	256	5.860966e-004	2.1323	0.1	3.543463e-003	1.5408	6
4225	1024	2.703318e-004	1.1164	0.1	1.045103e-003	1.7615	6
16641	4096	7.701234e-005	1.8116	0.1	3.896345e-004	1.4235	6
66049	16384	4.492321e-005	0.7776	0.1	5.012220e-005	2.9586	6
263169	65536	1.589134e-005	1.4992	0.1	1.631609e-005	1.6192	6
1050625	262144	1.032629e-006	3.9438	0.1	2.000238e-006	3.0281	6

Table 29.1 presents the best approximation of Franke's test function reported in this book. However, in Table 17.5 we needed only $N = 4225$ uniformly spaced points for a global Gaussian interpolant with $\varepsilon = 6.3$ to achieve an RMS-error of 7.371879e-009. Of course, it may be possible to obtain even better approximations with other RBFs. We do not claim that Gaussians are the "best" RBFs. However, we do recommend the partition of unity approach for the solution of large interpolation or approximation problems since it is relatively simple to implement and its execution is quite efficient.

Chapter 30

Approximation of Point Cloud Data in 3D

30.1 A General Approach via Implicit Surfaces

A common problem in computer graphics and computer aided design (CAD) is the reconstruction of a three-dimensional surface defined in terms of *point cloud data*, *i.e.*, as a set of unorganized, irregular points in 3D. For example, this could be laser range data obtained for the purpose of computer modeling of a complicated 3D object. Such applications also arise, *e.g.*, in computer graphics or in medical imaging. An approach to obtaining a surface that fits the given 3D point cloud data that has recently become rather popular (see, *e.g.*, [Carr *et al.* (1997); Carr *et al.* (2001); Morse *et al.* (2001); Ohtake *et al.* (2003a); Ohtake *et al.* (2003b); Turk and O'Brien (2002); Wendland (2002b)]) is based on the use of *implicit surfaces* defined in terms of some meshfree approximation method such as an RBF interpolant or an MLS approximant.

More precisely, given data of the form $\{\boldsymbol{x}_i = (x_i, y_i, z_i) \in \mathbb{R}^3,\ i = 1, \ldots, N\}$ assumed to come from some two-dimensional manifold $\mathcal{M}$ (*i.e.*, a surface in $\mathbb{R}^3$), we seek another surface $\mathcal{M}^*$ that is a reasonable approximation to $\mathcal{M}$. For the implicit surface approach we think of $\mathcal{M}$ as the surface of all points (x, y, z) that satisfy the implicit equation

$$f(x, y, z) = 0$$

for some function f. Thus, the function f implicitly defines the surface $\mathcal{M}$. In other words, the equation $f(x, y, z) = 0$ defines the zero iso-surface of the trivariate function f and therefore this iso-surface coincides with $\mathcal{M}$.

As so often before, we will construct the surface $\mathcal{M}^*$ via interpolation. Obviously, if we only specify the interpolant to be zero at the data points, then we will obtain a zero interpolant, and will not be able to extract a meaningful iso-surface. Therefore, the key to finding an approximation to the trivariate function f from the given data points $\boldsymbol{x}_i$, $i = 1, \ldots, N$, is to add an extra set of *off-surface points* to the data so that we can then compute a (solid) three-dimensional interpolant $\mathcal{P}_f$ to the total set of points, *i.e.*, the surface points plus the auxiliary off-surface points. This will result in a nontrivial interpolant, and we will then be able to extract its

zero iso-surface. To illustrate this technique we discuss how this idea works in the 2D setting in the next section.

The addition of off-surface points results in a problem of the same type as the scattered data approximation problems discussed in earlier chapters. In particular, if the data sets are large, it is advisable to use either a local radial basis interpolant, a moving-least squares approach, or a partition of unity interpolant. However, there are also implicit point cloud interpolants based on fast evaluation algorithms with global RBFs (see, *e.g.*, [Carr *et al.* (2001)]).

The surface reconstruction problem consists of three sub-problems:

(1) Construct the extra off-surface points.
(2) Find the trivariate meshfree approximant to the augmented data set.
(3) Render the iso-surface (zero-contour) of the fit computed in step (2).

In order to keep the discussion as simple as possible we assume that, in addition to the point cloud data, we are also given a set of surface normals $\boldsymbol{n}_i = (n_i^x, n_i^y, n_i^z)$ to the surface $\mathcal{M}$ at the points $\boldsymbol{x}_i = (x_i, y_i, z_i)$. If these normals are not explicitly given, there are techniques available that can be used to estimate the normals (see, *e.g.*, the discussion in [Wendland (2002b)]). Once we have the (oriented) surface normals, we construct the extra off-surface points by marching a small distance along the surface normal, *i.e.*, we obtain for each data point (x_i, y_i, z_i) two additional off-surface points. One point lies "outside" the manifold $\mathcal{M}$ and is given by

$$(x_{N+i}, y_{N+i}, z_{N+i}) = \boldsymbol{x}_i + \delta \boldsymbol{n}_i = (x_i + \delta n_i^x, y_i + \delta n_i^y, y_i + \delta n_i^z),$$

and the other point lies "inside" $\mathcal{M}$ and is given by

$$(x_{2N+i}, y_{2N+i}, z_{2N+i}) = \boldsymbol{x}_i - \delta \boldsymbol{n}_i = (x_i - \delta n_i^x, y_i - \delta n_i^y, y_i - \delta n_i^z).$$

Here δ is a small step size (whose specific magnitude can be rather critical for a good surface fit, see [Carr *et al.* (2001)]). In particular, if δ is chosen too large, then this can easily result in self-intersecting inner or outer auxiliary surfaces. In our simple MATLAB implementation in the next section we uniformly take δ to be 1% of the maximum dimension of the bounding box of the data as suggested in [Wendland (2002b)].

Once we have created the auxiliary data, the interpolant is computed by determining a function $\mathcal{P}_f$ whose zero contour interpolates the given point cloud data, and whose "inner" and "outer" offset contours interpolate the augmented data, *i.e.*,

$$\begin{aligned} \mathcal{P}_f(\boldsymbol{x}_i) &= 0, \qquad i = 1, \ldots, N, \\ \mathcal{P}_f(\boldsymbol{x}_i) &= 1, \qquad i = N+1, \ldots, 2N, \\ \mathcal{P}_f(\boldsymbol{x}_i) &= -1, \qquad i = 2N+1, \ldots, 3N. \end{aligned}$$

The values of ± 1 for the auxiliary data are arbitrary. Their precise value is not as critical as the choice of δ.

For the third step we also use a very simple solution, namely we just render the resulting approximating surface $\mathcal{M}^*$ as the zero contour of the 3D interpolant. In

MATLAB this can be accomplished with the command `isosurface` (or `contour` for 2D problems). This provides a rough picture of the implicit surface interpolant, but may also lead to some rendering artifacts that can be avoided with more sophisticated rendering procedures. For more serious applications one usually employs some variation of a *ray tracing* or *marching cube* algorithm (see the discussion in the references listed above).

The implicit surface representation has the advantage that the surface normals of the approximating surface $\mathcal{M}^*$ can be explicitly and analytically calculated as the gradients of the trivariate function $\mathcal{P}_f$, *i.e.*, $\boldsymbol{n}(\boldsymbol{x}) = \nabla \mathcal{P}_f(\boldsymbol{x})$.

Since in practice the data (*e.g.*, obtained by laser range scanners) is subject to measurement errors it is often beneficial if an additional smoothing procedure is employed. One can either use the ridge regression approach suggested in Chapter 19, or use a moving least squares approximation instead of an RBF interpolant. Another implicit smoothing technique was suggested in [Beatson and Bui (2003)]. Noisy data can also be dealt with by using a multilevel technique such as suggested in [Ohtake *et al.* (2003b)]. We discuss multilevel interpolation and approximation algorithms in Chapter 32.

30.2 An Illustration in 2D

Since the 3D point cloud interpolation problem requires an interpolant to points viewed as samples of a trivariate function whose graph is a 4D hypersurface, the visualization of the individual steps of the construction of the final iso-surface is problematic. We therefore illustrate the process with an analogous two-dimensional problem, *i.e.*, we assume we are given points (taken from a closed curve $\mathcal{C}$) in the plane, and it is our goal to find an interpolating curve $\mathcal{C}^*$. Below we present both MATLAB code and several figures.

Program 30.1. `PointCloud2D.m`

```
% PointCloud2D
% Script that fits a curve to 2D point cloud
% Calls on: DistanceMatrix
% Uses:     haltonseq (written by Daniel Dougherty
%                      from MATLAB Central File Exchange)
    % Gaussian RBF
 1  rbf = @(e,r) exp(-(e*r).^2); ep = 3.5;
 2  N = 81;   % number of data points
 3  neval = 40; % to create neval-by-neval evaluation grid
 4  t = 2*pi*haltonseq(N,1); dsites = [cos(t) sin(t)];
 5  x = (2+sin(t)).*cos(t); y = (2+cos(t)).*sin(t);
 6  nx = (2+cos(t)).*cos(t)-sin(t).^2;
```

```
 7  ny = (2+sin(t)).*sin(t)-cos(t).^2;
 8  dsites = [x y]; normals = [nx ny];
    % Produce auxiliary points along normals "inside" and "outside"
 9  bmin = min(dsites,[],1);  bmax = max(dsites,[],1);
10  bdim = max(bmax-bmin);
    % Distance along normal at which to place new points
11  delta = bdim/100;
    % Create new points
12  dsites(N+1:2*N,:) = dsites(1:N,:) + delta*normals;
13  dsites(2*N+1:3*N,:) = dsites(1:N,:) - delta*normals;
    % "original" points have rhs=0,
    % "inside" points have rhs=-1, "outside" points have rhs=1
14  rhs = [zeros(N,1); ones(N,1); -ones(N,1)];
    % Let centers coincide with data sites
15  ctrs = dsites;
    % Compute new bounding box
16  bmin = min(dsites,[],1);  bmax = max(dsites,[],1);
    % Create neval-by-neval equally spaced evaluation locations
    % in bounding box
17  xgrid = linspace(bmin(1),bmax(1),neval);
18  ygrid = linspace(bmin(2),bmax(2),neval);
19  [xe,ye] = meshgrid(xgrid,ygrid);
20  epoints = [xe(:) ye(:)];
21  DM_eval = DistanceMatrix(epoints,ctrs);
22  EM = rbf(ep,DM_eval);
23  DM_data = DistanceMatrix(dsites,ctrs);
24  IM = rbf(ep,DM_data);
25  Pf = EM * (IM\rhs);
    % Plot extended data with 2D-fit Pf
26  figure; hold on; view([-30,30])
27  plot3(dsites(:,1),dsites(:,2),rhs,'r.','markersize',20);
28  mesh(xe,ye,reshape(Pf,neval,neval));
29  axis tight; hold off
    % Plot data sites with interpolant (zero contour of 2D-fit Pf)
30  figure; hold on
31  plot(dsites(1:N,1),dsites(1:N,2),'bo');
32  contour(xe,ye,reshape(Pf,neval,neval),[0 0],'r');
33  hold off
```

In the MATLAB program `PointCloud2D.m` (see Program 30.1) we create test data on lines 4–8 by sampling a parametric curve (in polar coordinates) at irregular parameter values `t`. These points are displayed in the left plot of Figure 30.1.

Since we use a known representation of the curve to generate the point cloud it is also possible to obtain an exact normal vector associated with each data point (see lines 6–8).

The strategy for creation of the auxiliary points is the same as above, *i.e.*, we add data points "inside" and "outside" the given point set. This is done by placing these points along the normal vector at each original data point (see lines 12 and 13). The distance along the normal at which the auxiliary points are placed is taken to be 1% of the size of the maximum dimension of the bounding box of the original data (see lines 9–11).

Next, the problem is turned into a full 2D interpolation problem (whose solution has a 3D graph) by adding function values (of the unknown bivariate function f whose zero-level iso-curve will be the desired interpolating curve) at the extended data set. We assign a value of 0 to each original data point, and a value of 1 or -1 to "outside" or "inside" points, respectively. This is done on line 14 of the code and the resulting data is displayed in the right plot of Figure 30.1 (*c.f.* also line 27 of the code).

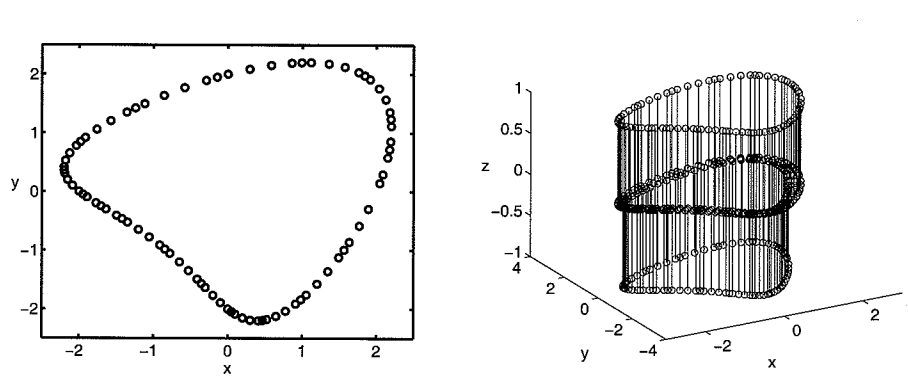

Fig. 30.1 Point cloud data (left) and extended "inner" and "outer" data (right) for implicit curve with 81 non-uniform data points.

Now we can solve the problem just like any of our 2D interpolation problems discussed earlier. In fact, in Program 30.1 we use straightforward RBF interpolation with Gaussian RBFs (see lines 1 and 19–25).

Finally, the zero contour of the resulting surface is extracted on line 32 using the `contour` command. In the left plot of Figure 30.2 we display a surface plot of the bivariate RBF interpolant to the extended data set (obtained via the `mesh` command on line 28), and in the right plot we show the final interpolating curve along with the original data.

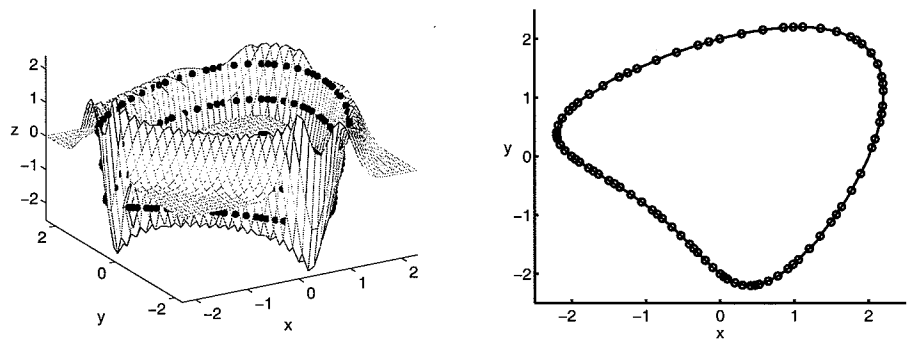

Fig. 30.2 Surface fit (left) and zero contour for 81 non-uniform data points.

30.3 A Simplistic Implementation in 3D via Partition of Unity Approximation in MATLAB

In the MATLAB program `PointCloud3D_PUCS` (see Program 30.2) we present a fairly simple implementation of the partition of unity approach to interpolation of point cloud data in 3D. The partition of unity is created with a Shepard approximant as in the previous chapter. As in Program 29.1 we use Wendland's C^2 compactly supported function as both the Shepard weights and for the local RBF interpolants.

The data sets used in our examples correspond to various resolutions of the Stanford bunny available on the world-wide web at `http://graphics.stanford.edu/data/3Dscanrep/`. Data sets consisting of 35947, 8171, 1889, and 453 points are included in the file `bunny.tar.gz`. The normals for this kind of PLY data can be computed with the utility `normalsply` from the package `ply.tar.gz` provided by Greg Turk, and available on the world-wide web at `http://www.cc.gatech.edu/projects/large_models/ply.html`. Results obtained with the `PointCloud3D_PUCS` for the 453 and 8171 point cloud sets are displayed in Figure 30.3. Processed data files are included on the enclosed CD.

Many parts of the MATLAB code for `PointCloud3D_PUCS` are similar to Program 29.1. The bunny data set including point normals is loaded on line 6, and the bounding box for the point cloud and its maximum dimension are computed on lines 7–8. The off-surface points are added in lines 10–14. Then the right-hand side for the augmented (3D) interpolation problem is defined on line 15 (assigning a value of 0 for the on-surface data points, and a value of ± 1 for the "outside" and "inside" off-surface points), and we recompute the bounding box for the augmented data on line 16.

We have found that a reasonable value for the radius of the partition of unity subdomains seems to be given by the maximal dimension of the bounding box divided by the cube root of the number, M, of subdomains, *i.e.*, $\text{diam}(\Omega_j) = 1/w_\varepsilon$ with $w_\varepsilon = \sqrt[3]{M}/\text{bdim}$ (see lines 9 and 28).

The main part of the program (lines 29–46) is almost identical to lines 21–38 of Program 29.1. On lines 47–55 we add the code that creates and displays the zero-contour iso-surface for the 3D (solid) interpolant `Pf` along with the point cloud data.

Program 30.2. `PointCloud3D_PUCS.m`

```
% PointCloud3D_PUCS
% Script that fits a surface to 3D point cloud using partition of
% unity approximation with sparse matrices
% Calls on: CSEvalMatrix
% Uses:     k-D tree package by Guy Shechter
%           from MATLAB Central File Exchange
   % Weight function for global Shepard partition of unity weighting
 1 wf = @(e,r) r.^4.*(5*spones(r)-4*r);
   % The RBF basis function for local RBF interpolation
 2 rbf = @(e,r) r.^4.*(5*spones(r)-4*r);
 3 ep = 1; % Parameter for basis function
   % Parameter for npu-by-npu-by-npu grid of PU cells
 4 npu = 8;
   % Parameter for npu-by-npu-by-npu grid of PU cells
 5 neval = 25;
   % Load data points and compute bounding box
 6 load('Data3D_Bunny3');  N = size(dsites,1);
 7 bmin = min(dsites,[],1);  bmax = max(dsites,[],1);
 8 bdim = max(bmax-bmin);
 9 wep = npu/bdim;
   % Add auxiliary points along normals "inside" and "outside"
   % Find points with nonzero normal vectors and count them
10 withnormals = find(normals(:,1)|normals(:,2)|normals(:,3));
11 addpoints = length(withnormals);
   % Distance along normal at which to place new points
12 delta = bdim/100;
   % Create new points
13a dsites(N+1:N+addpoints,:) = ...
13b      dsites(withnormals,:) + delta*normals(withnormals,:);
14a dsites(N+addpoints+1:N+2*addpoints,:) = ...
14b      dsites(withnormals,:) - delta*normals(withnormals,:);
   % Interpolant is implicit surface, i.e.,
   % "original" points have rhs=0, "inside" rhs=-1, "outside" rhs=1
15 rhs = [zeros(N,1); ones(addpoints,1); -ones(addpoints,1)];
   % Compute new bounding box
16 bmin = min(dsites,[],1);  bmax = max(dsites,[],1);
```

```
17  ctrs = dsites;
    % Create neval-by-neval-by-neval equally spaced evaluation
    % locations in bounding box
18  xgrid = linspace(bmin(1),bmax(1),neval);
19  ygrid = linspace(bmin(2),bmax(2),neval);
20  zgrid = linspace(bmin(3),bmax(3),neval);
21  [xe,ye,ze] = meshgrid(xgrid,ygrid,zgrid);
22  epoints = [xe(:) ye(:) ze(:)];
    % Create npu-by-npu-by-npu equally spaced centers of PU cells
    % in bounding box
23  puxgrid = linspace(bmin(1),bmax(1),npu);
24  puygrid = linspace(bmin(2),bmax(2),npu);
25  puzgrid = linspace(bmin(3),bmax(3),npu);
26  [xpu,ypu,zpu] = meshgrid(puxgrid,puygrid,puzgrid);
27  cellctrs = [xpu(:) ypu(:) zpu(:)];
28  cellradius = 1/wep;
    % Compute Shepard evaluation matrix
29  DM_eval = DistanceMatrixCSRBF(epoints,cellctrs,wep);
30  SEM = wf(wep,DM_eval);
31  SEM = spdiags(1./(SEM*ones(npu^3,1)),0,neval^3,neval^3)*SEM;
    % Build k-D trees for data sites and evaluation points
32  [tmp,tmp,datatree] = kdtree(dsites,[]);
33  [tmp,tmp,evaltree] = kdtree(epoints,[]);
34  Pf = zeros(neval^3,1); % initialize
35  for j=1:npu^3
       % Find data sites in cell j
36a    [pts,dist,idx] = kdrangequery(datatree,...
36b                     cellctrs(j,:),cellradius);
37     if (length(idx) > 0)
          % Build local interpolation matrix for cell j
38a       DM_data = DistanceMatrixCSRBF(dsites(idx,:),...
38b                 ctrs(idx,:),ep);
39        IM = rbf(ep,DM_data);
          % Find evaluation points in cell j
40a       [epts,edist,eidx] = kdrangequery(evaltree,...
40b                           cellctrs(j,:),cellradius);
          % Compute local evaluation matrix
41a       DM_eval = DistanceMatrixCSRBF(epoints(eidx,:),...
41b                 ctrs(idx,:),ep);
42        EM = rbf(ep,DM_eval);
          % Compute local RBF interpolant
43        localfit = EM * (IM\rhs(idx));
```



```
          % Accumulate global fit
44        Pf(eidx) = Pf(eidx) + localfit.*SEM(eidx,j);
45     end
46  end
    % Plot data sites with interpolant (zero contour of 3D-fit Pf)
47  figure; hold on
48  plot3(dsites(1:N,1),dsites(1:N,2),dsites(1:N,3),'bo');
49a pfit = patch(isosurface(xe,ye,ze,...
49b                       reshape(Pf,neval,neval,neval),0));
50  isonormals(xe,ye,ze,reshape(Pf,neval,neval,neval),pfit)
51a set(pfit,'FaceLighting','gouraud','FaceColor',...
51b     'red','EdgeColor','none');
52  light('Position',[0 0 1],'Style','infinite');
53  daspect([1 1 1]); view([0,90]);
54  axis([bmin(1) bmax(1) bmin(2) bmax(2) bmin(3) bmax(3)]);
55  axis off;  hold off
```

In Figure 30.3 we display partition of unity fits based on local RBF interpolants built with compactly supported Wendland's C^2 basis functions. The point cloud data sets consist of 453 points (top plots in Figure 30.3) and 8171 points (bottom plots in Figure 30.3). The augmented data sets for the 3D interpolants are almost three times as large (since not every data point has a normal vector associated with it). On the left we show the fitted surface along with the data points, and on the right the fit is displayed by itself. For the plots in the bottom part of Figure 30.3 the simple iso-surface plot in MATLAB does a surprisingly good job. In other situations, however, it causes some artifacts such as the extra surface fragment near the bunny's ear in the top part of Figure 30.3.

For the top plots in Figure 30.3 we used $8^3 = 256$ subdomains and $25^3 = 15625$ evaluation points in the bounding box of the 3D data, and for those on the bottom of Figure 30.3 we used $32^3 = 32768$ subdomains and $50^3 = 125000$ evaluation points.

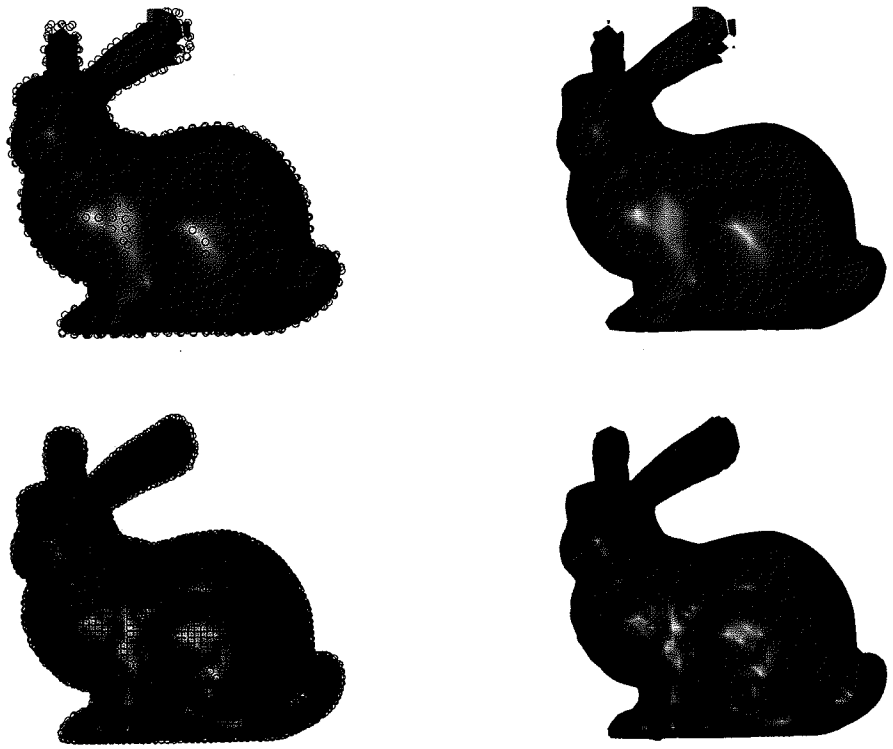

Fig. 30.3 Partition of unity implicit surface interpolant to Stanford bunny with 453 (top) and 8171 (bottom) data points.

Chapter 31

Fixed Level Residual Iteration

In the next few chapters we will look at various versions of residual iteration. The basic idea of using an iterative algorithm in which one takes advantage of the residual of an initial approximation to obtain a more accurate solution is well-known in many branches of mathematics.

31.1 Iterative Refinement

For example, in numerical linear algebra this process is known as *iterative refinement* (see, *e.g.*, [Kincaid and Cheney (2002)]). We might be interested in solving a system of linear equations $A\boldsymbol{x} = \boldsymbol{b}$, and obtain a (numerical) solution $\boldsymbol{x}_0$ by applying an algorithm such as Gaussian elimination. We can then compute the residual $\boldsymbol{r} = \boldsymbol{b} - A\boldsymbol{x}_0$, and realize that it is related to the error, $\boldsymbol{e} = \boldsymbol{x} - \boldsymbol{x}_0$, via the relation

$$A\boldsymbol{e} = A\boldsymbol{x} - A\boldsymbol{x}_0 = \boldsymbol{r}. \tag{31.1}$$

Thus, by adding the (numerical) solution $\boldsymbol{e}_0$ of equation (31.1) to the initial solution $\boldsymbol{x}_0$ one expects to improve the initial approximation to $\boldsymbol{x}_1 = \boldsymbol{x}_0 + \boldsymbol{e}_0$ (since the true solution $\boldsymbol{x} = \boldsymbol{x}_0 + \boldsymbol{e}$). Of course, this procedure can be repeated iteratively. This leads to the algorithm

Algorithm 31.1. Iterative refinement

(1) Compute an approximate solution $\boldsymbol{x}_0$ of $A\boldsymbol{x} = \boldsymbol{b}$.
(2) For $k = 1, 2, \ldots$ do
  (a) Compute the residual $\boldsymbol{r}_k = \boldsymbol{b} - A\boldsymbol{x}_{k-1}$.
  (b) Solve $A\boldsymbol{e}_k = \boldsymbol{r}_k$.
  (c) Update $\boldsymbol{x}_k = \boldsymbol{x}_{k-1} + \boldsymbol{e}_k$.

We can rewrite the last statement in the algorithm as

$$\boldsymbol{x}_k = \boldsymbol{x}_{k-1} + B(\boldsymbol{b} - A\boldsymbol{x}_{k-1}), \tag{31.2}$$

where B is an *approximate (or numerical) inverse* of A characterized by the property that $\|I - BA\| < 1$ for some matrix norm. This condition allows us to express the

exact inverse of A by a Neumann series, *i.e.*,

$$A^{-1} = (BA)^{-1}B = \left[\sum_{j=0}^{\infty}(I - BA)^j\right] B.$$

Therefore the exact solution of $A\boldsymbol{x} = \boldsymbol{b}$ can be written in the form

$$\boldsymbol{x} = A^{-1}\boldsymbol{b} = \left[\sum_{j=0}^{\infty}(I - BA)^j\right] B\boldsymbol{b}. \tag{31.3}$$

For the iterative refinement algorithm, on the other hand, we have from (31.2) and the fact that $\boldsymbol{x}_0 = B\boldsymbol{b}$, that

$$\begin{aligned}\boldsymbol{x}_k &= \boldsymbol{x}_{k-1} + B(\boldsymbol{b} - A\boldsymbol{x}_{k-1})\\ &= (I - BA)\boldsymbol{x}_{k-1} + B\boldsymbol{b}\\ &= (I - BA)\boldsymbol{x}_{k-1} + \boldsymbol{x}_0.\end{aligned} \tag{31.4}$$

We can recursively substitute this relation back in for $\boldsymbol{x}_{k-1}$, $\boldsymbol{x}_{k-2}$, etc., and obtain

$$\boldsymbol{x}_k = \left[\sum_{j=0}^{k}(I - BA)^j\right] \boldsymbol{x}_0 = \left[\sum_{j=0}^{k}(I - BA)^j\right] B\boldsymbol{b}. \tag{31.5}$$

It is now easy to see that the iterates $\boldsymbol{x}_k$ of the refinement algorithm converge to the exact solution $\boldsymbol{x}$. We simply look at the difference $\boldsymbol{x} - \boldsymbol{x}_k$ at level k, *i.e.*, from (31.3) and (31.5) we obtain

$$\boldsymbol{x} - \boldsymbol{x}_k = \left[\sum_{j=k+1}^{\infty}(I - BA)^j\right] B\boldsymbol{b},$$

whose norm goes to zero for $k \to \infty$ since $\|I - BA\| < 1$ by the assumption made on the approximate inverse B.

We now apply these ideas to RBF interpolation and MLS approximation. In this and the following chapters we will consider three different scenarios:

- Fixed level iteration, *i.e.*, the iterative refinement algorithm is performed on a fixed set of data points $\mathcal{X}$.
- Multilevel iteration, *i.e.*, we work with a nested sequence of data sets $\mathcal{X}_0 \subseteq \mathcal{X}_1 \subseteq \cdots \subseteq \mathcal{X}$.
- Adaptive iteration, *i.e.*, residual iteration is performed on adaptively chosen subsets of $\mathcal{X}$, *e.g.*, by starting with some small subset of $\mathcal{X}$ and then adding one point at a time from the remainder of $\mathcal{X}$ that is determined to be "optimal".

Note that the third approach is similar to the adaptive knot insertion algorithm of Chapter 21.

31.2 Fixed Level Iteration

The simplest setting for a meshfree residual iteration algorithm arises when we fix the data sites $\mathcal{X} = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N\}$ throughout the iterative procedure. For this setting the iterative refinement algorithm from linear algebra can be adapted in a straightforward way. We first discuss residual iteration for quasi-interpolants (or approximate MLS approximants) based on (radial) generating functions Ψ_j, $j = 1, \ldots, N$.

If we keep the same set of generating functions for all steps of the iteration then we are performing *non-stationary approximation* and we obtain

Algorithm 31.2. Fixed level residual iteration based on quasi-interpolation

(1) Compute an initial approximation $\mathcal{P}_f^{(0)}$ to the data $\{(\boldsymbol{x}_j, f(\boldsymbol{x}_j)),\ j = 1, \ldots, N\}$ of the form $\mathcal{P}_f^{(0)}(\boldsymbol{x}) = \sum_{j=1}^{N} f(\boldsymbol{x}_j)\Psi_j(\boldsymbol{x})$.

(2) For $k = 1, 2, \ldots$ do

(a) Compute the residuals $r_k(\boldsymbol{x}_j) = f(\boldsymbol{x}_j) - \mathcal{P}_f^{(k-1)}(\boldsymbol{x}_j)$ for all $j = 1, \ldots, N$.

(b) Compute the correction $u(\boldsymbol{x}) = \sum_{j=1}^{N} r_k(\boldsymbol{x}_j)\Psi_j(\boldsymbol{x})$.

(c) Update $\mathcal{P}_f^{(k)}(\boldsymbol{x}) = \mathcal{P}_f^{(k-1)}(\boldsymbol{x}) + u(\boldsymbol{x})$.

As for the iterative refinement algorithm, we can rewrite the last line of the algorithm as

$$\mathcal{P}_f^{(k)}(\boldsymbol{x}) = \mathcal{P}_f^{(k-1)}(\boldsymbol{x}) + \sum_{j=1}^{N}\left[f(\boldsymbol{x}_j) - \mathcal{P}_f^{(k-1)}(\boldsymbol{x}_j)\right]\Psi_j(\boldsymbol{x}).$$

Now we restrict the evaluation of the approximation to the data sites only. Thus, we have

$$\mathcal{P}_f^{(k)}(\boldsymbol{x}_i) = \mathcal{P}_f^{(k-1)}(\boldsymbol{x}_i) + \sum_{j=1}^{N}\left[f(\boldsymbol{x}_j) - \mathcal{P}_f^{(k-1)}(\boldsymbol{x}_j)\right]\Psi_j(\boldsymbol{x}_i), \qquad i = 1, \ldots, N. \tag{31.6}$$

Next we collect all of these N equations into one single matrix-vector equation by introducing the vectors $\boldsymbol{f} = [f(\boldsymbol{x}_1), f(\boldsymbol{x}_2), \ldots, f(\boldsymbol{x}_N)]^T$ and $\Psi = [\Psi_1, \Psi_2, \ldots, \Psi_N]^T$. This allows us to rewrite the initial approximant in matrix-vector form

$$\mathcal{P}_f^{(0)}(\boldsymbol{x}) = \Psi^T(\boldsymbol{x})\boldsymbol{f}. \tag{31.7}$$

Moreover, evaluation of the vector Ψ of generating functions at the data sites $\boldsymbol{x}_i$, $i = 1, \ldots, N$ gives rise to a matrix A with rows $\Psi^T(\boldsymbol{x}_i)$, $i = 1, \ldots, N$. Therefore, (31.6) now becomes

$$\boldsymbol{\mathcal{P}_f}^{(k)} = \boldsymbol{\mathcal{P}_f}^{(k-1)} + A(\boldsymbol{f} - \boldsymbol{\mathcal{P}_f}^{(k-1)}),$$

where we interpret $\mathcal{P}_f{}^{(k)}$ as a vector of values of the approximant at the data sites, *i.e.*, $\mathcal{P}_f{}^{(k)} = [\mathcal{P}_f^{(k)}(\boldsymbol{x}_1), \ldots, \mathcal{P}_f^{(k)}(\boldsymbol{x}_N)]^T$.

Next we follow analogous steps as in our discussion of iterative refinement above. Thus

$$\begin{aligned} \mathcal{P}_f{}^{(k)} &= \mathcal{P}_f{}^{(k-1)} + A(\boldsymbol{f} - \mathcal{P}_f{}^{(k-1)}) \\ &= (I - A)\mathcal{P}_f{}^{(k-1)} + A\boldsymbol{f} \\ &= (I - A)\mathcal{P}_f{}^{(k-1)} + \mathcal{P}_f{}^{(0)}, \end{aligned} \tag{31.8}$$

since (31.7) implies that on the data sites we have $\mathcal{P}_f{}^{(0)} = A\boldsymbol{f}$. Now we can again recursively substitute back in and obtain

$$\mathcal{P}_f{}^{(k)} = \left[\sum_{j=0}^{k}(I - A)^j\right]\mathcal{P}_f{}^{(0)} = \left[\sum_{j=0}^{k}(I - A)^j\right]A\boldsymbol{f}. \tag{31.9}$$

Note that here we have to deal only with the matrix A since the computation of the correction in the algorithm does not require the solution of a linear system.

As before, the sum $\sum_{j=0}^{k}(I - A)^j$ can be seen as a truncated Neumann series expansion for the inverse of the matrix A. If we demand that $\|I - A\| < 1$, then the matrix $\left(\sum_{j=0}^{k}(I - A)^j\right)$ is an approximate inverse of A which converges to A^{-1} since $\|I - A\|^k \to 0$ for $k \to \infty$. More details (such as sufficient conditions under which $\|I - A\| < 1$) are given in [Fasshauer and Zhang (2006)].

In order to establish a connection between iterated (approximate) MLS approximation and RBF interpolation we assume the matrix A to be positive definite and generated by radial basis functions $\Phi_j = \varphi(\|\cdot - \boldsymbol{x}_j\|)$ as in our discussions in earlier chapters. Then A corresponds to an RBF interpolation matrix, and we see that the iterated (approximate) MLS approximation converges to the RBF interpolant provided the same function spaces are used, *i.e.*, $\text{span}\{\Psi_j,\ j = 1, \ldots, N\} = \text{span}\{\Phi_j,\ j = 1, \ldots, N\}$.

In particular, we have established

Theorem 31.1. *Assume $\Psi_1, \ldots, \Psi_N$ are strictly positive definite (radial) generating functions for approximate MLS approximation as discussed in Chapter 26. Then the residual iteration fit of Algorithm 31.2 based on approximate MLS approximation with these generating functions converges to the RBF interpolant based on the same basis functions $\Psi_1, \ldots, \Psi_N$ provided the matrix A with entries $A_{ij} = \Psi_j(\boldsymbol{x}_i)$ satisfies $\|I - A\| < 1$.*

A sufficient condition for A to satisfy $\|I - A\| < 1$ was given in [Fasshauer and Zhang (2006)]. As long as the maximum row sum of A is small enough, *i.e.*,

$$\max_{i=1,2,\ldots,N}\left\{\sum_{j=1}^{N}|A_{i,j}|\right\} < 2,$$

we have convergence of the residual iteration algorithm. This condition is closely related to the Lebesgue function of the RBF interpolant. For example, it is not hard to see that Shepard generating functions satisfy this condition since each row sum is equal to one due to the partition of unity property of the Shepard functions. For other types of functions the condition can be satisfied by an appropriate scaling of the basic function with a sufficiently small shape parameter. However, if ε is taken too small, then the algorithm converges very slowly. A series of experiments analyzing the behavior of the algorithm are presented in [Fasshauer and Zhang (2006)] and also in Section 31.4 below.

The question of whether the approximate MLS generating functions are strictly positive definite has been irrelevant up to this point. However, in order to make the connection between AMLS approximation and RBF interpolation as stated in Theorem 31.1 it is important to find AMLS generating functions that satisfy this additional condition. Of course, any (appropriately normalized) strictly positive definite function can serve as a second-order accurate AMLS generating function. However, it is an open question for which of these functions their higher-order generating functions computed according to our discussion in Chapter 26 are also strictly positive definite.

The family of Laguerre-Gaussians (4.2) provides one example of generating/basis functions that can be used to illustrate Theorem 31.1 (see the numerical experiments below) since their Fourier transforms are positive (see (4.3)).

31.3 Modifications of the Basic Fixed Level Iteration Algorithm

If we start from the interpolation end, then the interpolation conditions $\mathcal{P}_f(\boldsymbol{x}_i) = f(\boldsymbol{x}_i)$ tell us that we need to solve the linear system $A\boldsymbol{c} = \boldsymbol{f}$ in order to find the coefficients of the RBF expansion

$$\mathcal{P}_f(\boldsymbol{x}) = \sum_{j=1}^{N} c_j\Phi_j(\boldsymbol{x}).$$

Following the same iterative procedure as above (*c.f.* (31.4)) this leads to

$$\boldsymbol{c}_k = \boldsymbol{c}_{k-1} + B(\boldsymbol{f} - A\boldsymbol{c}_{k-1}) \tag{31.10}$$

$$= \sum_{j=0}^{k}(I - BA)^j B\boldsymbol{f}, \tag{31.11}$$

where B is an approximate inverse of A as in Section 31.1 and we let $\boldsymbol{c}_0 = B\boldsymbol{f}$. Here $\boldsymbol{c}_k$ is the k-th step approximation to the coefficient vector $\boldsymbol{c} = [c_1, \ldots, c_N]^T$.

Equation (31.10) can also be rewritten as

$$\boldsymbol{c}_k = (I - BA)\boldsymbol{c}_{k-1} + B\boldsymbol{f},$$

and therefore corresponds to a standard stationary iteration for the solution of linear systems (see, *e.g.*, p. 620 of [Meyer (2000)]). The splitting matrices such that $A = M - N$ are $M = B^{-1}$, $N = B^{-1} - A$, and $H = M^{-1}N = (I - BA)$.

On the other hand, (31.11) gives us an interpretation of the residual iteration as a Krylov subspace method with the Krylov subspaces generated by the matrix $I - BA$ and the vector $B\boldsymbol{f}$.

In the quasi-interpolation formulation the corresponding formulas are given by (31.9), *i.e.*,

$$\mathcal{P}_{\boldsymbol{f}}{}^{(k)} = \left[\sum_{j=0}^{k}(I-A)^j\right] A\boldsymbol{f} \tag{31.12}$$

and can also be interpreted as a Krylov subspace iteration with the Krylov subspaces generated by the matrix $I - A$ and the vector $A\boldsymbol{f}$. Note, however, that in (31.11) we are computing the coefficients of the RBF interpolant, while in (31.12) we are directly computing an approximation to the interpolant.

A natural problem associated with Krylov subspace methods is the determination of coefficients (search directions) α_j such that $\sum_{j=0}^{k}\alpha_j(I-A)^jA\boldsymbol{f}$ converges faster than the generic method with $\alpha_j = 1$ discussed above. Some related work is discussed in the context of the Faul-Powell algorithm in Section 33.2.

We conclude our discussion of modifications of the basic fixed level residual iteration algorithm by noting that the usual stationary approximation method cannot be applied within the fixed level iteration paradigm since we do not have a change in data density that can be used as a guide to re-scale the basis functions. However, it is possible to generalize the non-stationary algorithm to a more general setting in which we change the approximation space from one step to the next. As in the non-stationary setting we can only apply this strategy with approximation methods since an interpolation method will immediately lead to a zero initial residual. For example, one could devise an algorithm in which we use cross-validation at each iteration step to determine the optimal shape parameter (or support size) for the next residual correction. Such an algorithm would also fit into the category of adaptive iterations as discussed below.

31.4 Iterated Approximate MLS Approximation in MATLAB

We now illustrate the fixed level residual iteration algorithm with some MATLAB experiments based on the iteration of approximate MLS approximants with Gaussian generating functions. To obtain some test data we use Franke's function (2.2) on 289 Halton points in the unit square.

In our earlier discussion of approximate MLS approximation we limited ourselves mostly to the case of uniformly spaced data. This was due to the fact, that for non-uniformly spaced data one needs to scale the generating functions individually according to the local variation in the data density in order to maintain the approximate approximation orders stated in Theorem 26.1. Now the convergence result of Theorem 31.1 shows that we no longer need to feel bound by those limitations.

Iteration will automatically improve the approximate MLS fit also on non-uniform data. On the other hand, this observation suggests that the use of a uniform shape parameter for RBF interpolation is most likely not the ideal strategy to obtain highly accurate RBF fits. While a few experiments of RBF interpolation with varying shape parameters exist in the literature (see, *e.g.*, [Kansa and Carlson (1992); Bozzini *et al.* (2002); Fornberg and Zuev (2006)]), the theory for this case is only rudimentary [Bozzini *et al.* (2002)].

The MATLAB code for our examples is provided in Program 31.1. Since we are iterating the approximate MLS approximation we define the scale of the generating functions in terms of the parameter $\mathcal{D}$ (see line 2). However, since the RBF (Gaussian) is defined with the parameter ε we convert $\mathcal{D}$ to ε based on the formula $\varepsilon = 1/(\sqrt{\mathcal{D}}h)$. We approximate h (even for non-uniform Halton points) by $h = 1/(\sqrt{N}-1)$, where N is the number of data points (in 2D).

In contrast to previous programs we now require two sets of evaluation points. The usual `epoints` that we employ for error computation and plotting along with another set `respoints`, the points at which we evaluate the residuals during the iterative procedure. These points coincide with the data points (see line 13). The iteration on lines 23–28 is equivalent to the formulation in Algorithm 31.2 above.

Program 31.1. `Iterated_MLSApproxApprox2D.m`

```
% Iterated_MLSApproxApprox2D
% Script that performs iterated approximate MLS approximation
% Calls on: DistanceMatrix
 1  rbf = @(e,r) exp(-(e*r).^2);
 2  D = 64/9;  % Parameter for basis function
    % Define Franke's function as testfunction
 3  f1 = @(x,y) 0.75*exp(-((9*x-2).^2+(9*y-2).^2)/4);
 4  f2 = @(x,y) 0.75*exp(-((9*x+1).^2/49+(9*y+1).^2/10));
 5  f3 = @(x,y) 0.5*exp(-((9*x-7).^2+(9*y-3).^2)/4);
 6  f4 = @(x,y) 0.2*exp(-((9*x-4).^2+(9*y-7).^2));
 7  testfunction = @(x,y) f1(x,y)+f2(x,y)+f3(x,y)-f4(x,y);
 8  neval = 40;
 9  N = 289; gridtype = 'h';
    % Convert D to epsilon for use with basis function definition
10  h = 1/(sqrt(N)-1); ep = 1/(sqrt(D)*h);
    % Number of levels for multilevel iteration
11  maxlevel = 10000;
    % Load data points
12  name = sprintf('Data2D_%d%s',N,gridtype); load(name)
13  respoints = dsites; ctrs = dsites;
    % Create neval-by-neval equally spaced evaluation locations
    % in the unit square
```

```
14  grid = linspace(0,1,neval); [xe,ye] = meshgrid(grid);
15  epoints = [xe(:) ye(:)];
    % Compute exact solution
16  exact = testfunction(epoints(:,1),epoints(:,2));
    % Compute evaluation matrix directly based on the distances
    % between the evaluation points and centers
17  DM = DistanceMatrix(epoints,ctrs);
18  EM = rbf(ep,DM)/(pi*D);
    % Compute - for all levels - evaluation matrices for
    % residuals directly based on the distances between the
    % next finer points (respoints) and centers
19  DM = DistanceMatrix(respoints,ctrs);
20  RM = rbf(ep,DM)/(pi*D);
21  Pf = zeros(neval^2,1);  %  initialize
    % Create vector of function values (initial residual),
22  rhs = testfunction(dsites(:,1),dsites(:,2));
23  for level=1:maxlevel
        % Evaluate on evaluation points
        % (for error computation and plotting)
24      Pf = Pf + EM*rhs;
        % Compute new residual
25      rhs = rhs - RM*rhs;
        % Compute errors on evaluation grid
26      maxerr(level) = norm(Pf-exact,inf);
27      rms_err(level) = norm(Pf-exact)/neval;
28  end
29  figure; semilogy(1:maxlevel,maxerr,'b',1:maxlevel,rms_err,'r');
```

According to the experiments shown in Figure 2.5 the optimal shape parameter for Gaussian interpolation to Franke's function on 289 Halton points is close to $\varepsilon = 6$. The corresponding value of $\mathcal{D}$ for the Gaussian as an approximate MLS generating function is $\mathcal{D} = 64/9$ (since $\varepsilon = 1/(\sqrt{\mathcal{D}}h)$, and we approximate h for the non-uniform Halton points by the value we would have for uniform points, *i.e.*, $h = 1/(\sqrt{N} - 1)$. For this value of the shape parameter we see the convergence behavior of the approximate MLS residual iteration in the left plot of Figure 31.1. The final maximum error after 10000 iterations is 9.921772e-002 (top/solid curve), and the RMS error is 3.939342e-003 (lower/dashed curve). For comparison, the errors for the corresponding RBF interpolant (which is the theoretical limit of the residual iteration) are 3.238735e-002 for the maximum error and 1.074443e-003 for the RMS error. These errors are included as horizontal straight lines in the left plot of Figure 31.1.

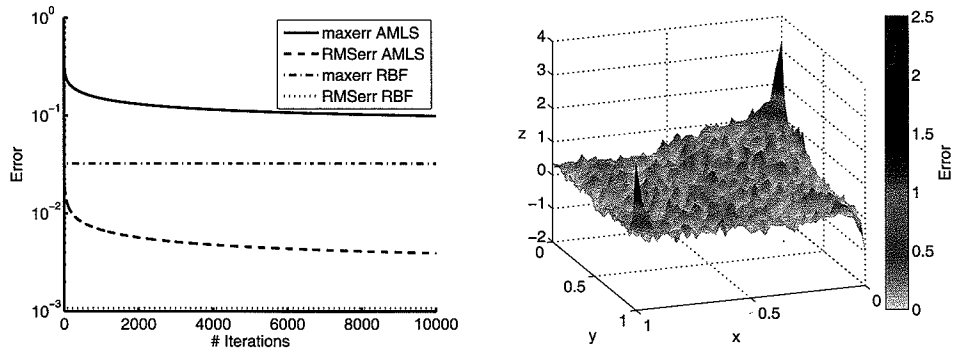

Fig. 31.1 Convergence for iterated MLS approximation based on Gaussian generating functions with $\mathcal{D} = 64/9$ ($\varepsilon = 6$) to data sampled from Franke's function at 289 Halton points (left), and fit for an RBF interpolant based on Gaussians with $\varepsilon = 1$ (right).

An advantage of the residual iteration is that it allows us to compute radial basis approximations also for values of the shape parameter for which the interpolation matrix is very ill-conditioned. For example, the right plot of Figure 31.1 shows an RBF interpolant to the 289 Halton samples of Franke's function based on Gaussians with shape parameter $\varepsilon = 1$. The reciprocal condition number estimate provided by MATLAB for the interpolation matrix for this problem is `RCOND = 2.132739e-020`.

In these ill-conditioned cases convergence of the residual iteration to the limit is rather slow, but the approximations can be computed very stably. In the left plot of Figure 31.2 we show the convergence behavior for the residual iteration with approximate MLS approximants based on Gaussian generating functions with $\mathcal{D} = 256$ (corresponding to $\varepsilon = 1$). Note that the approximation errors for the iterative scheme quickly drop below the maximum error of 2.507017e+000 and RMS error of 2.186992e-001 of the mostly meaningless interpolant. The corresponding fit for the iterative method is displayed in the right plot of the figure. While this fit is not very accurate, it is still much more reliable than the fit consisting of mostly numerical noise shown in the right plot of Figure 31.1. The final errors after 10000 iterations are 2.933448e-001 for the maximum error and 8.470775e-002 for the RMS error.

It is of interest to note that the residual iteration shows the most dramatic error improvement during the first few iterations. Thus, only a few iterations of approximate MLS approximation are required to obtain a reasonable (and stably computable) approximation to the RBF interpolant. Moreover, we emphasize again that while our discussion of approximate MLS approximation was mostly limited to the case of uniform data (at least for most practical purposes), this limitation no longer exists for the residual iteration algorithm.

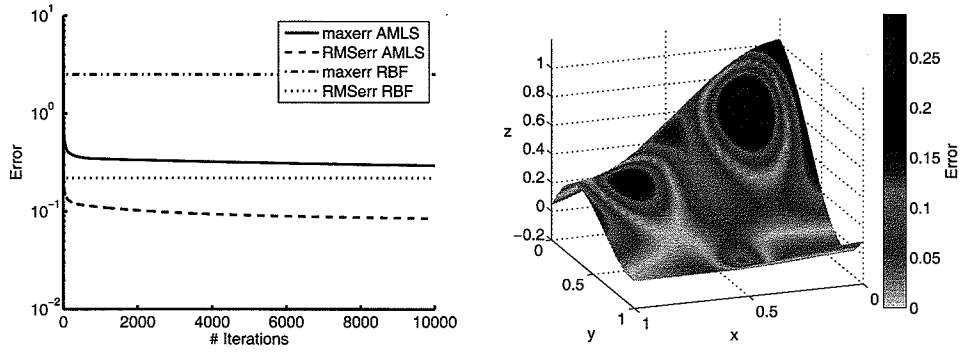

Fig. 31.2 Convergence and final fit for iterated approximate MLS approximation based on 289 Halton points and Gaussians with $\mathcal{D} = 256$ ($\varepsilon = 1$).

31.5 Iterated Shepard Approximation

The use of other approximation methods within the fixed level residual iteration algorithm such as regular MLS approximation or RBF least squares approximation is also possible. We point out, however, that the use of RBF interpolation does not make any sense in the context of fixed level residual iteration since the residuals on the data are automatically zero if we perform interpolation of the data.

If we want to use regular MLS approximation instead of approximate MLS approximation in the residual iteration, then, for Shepard's method, this means replacing line 18 in Program 31.1 by

```
18a EM = rbf(ep,DM);
18b EM = EM./repmat(EM*ones(N,1),1,N);     % Shepard normalization
```

and line 20 by

```
20a RM = rbf(ep,DM);
20b RM = RM./repmat(RM*ones(N,1),1,N);     % Shepard normalization
```

Note that we can now no longer claim that the limit of the iterated Shepard approximant is given by the RBF interpolant based on the Shepard weights as basis functions. In fact, the iterated Shepard approximant becomes more accurate than the RBF interpolant. For example, if we take Gaussian weight functions with $\varepsilon = 6$ or $\varepsilon = 1$ as above, then the corresponding convergence behavior for the iterated Shepard approximant is displayed in Figure 31.3. Moreover, for the example with $\varepsilon = 6$, 45 iterations of the Shepard approximant result in a smaller maximum error than the RBF interpolant, while 3515 iterations are required to push the RMS error for the Shepard iteration below 1.074443e-003. After 10000 iterations with

Shepard approximants the maximum error is 8.760946e-003 and the RMS error is 7.889609e-004. For the $\varepsilon = 1$ example the final errors (after 10000 iterations) are 2.664303e-001 for the maximum error and 8.169080e-002 for the RMS error. This example is very similar to the AMLS example displayed in Figure 31.2. Again, the most significant part of the error improvement occurs during the first 10–20 iterations.

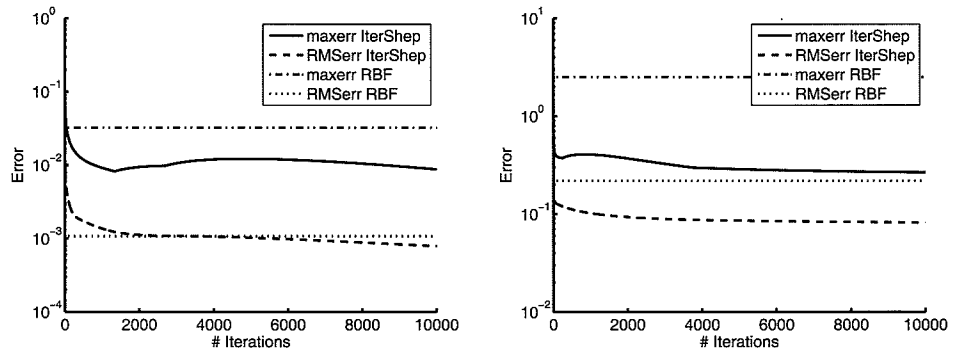

Fig. 31.3 Convergence for iterated Shepard approximation based on 289 Halton points and Gaussian weights with $\varepsilon = 6$ (left), and $\varepsilon = 1$ (right).

For other data sets and other values of ε residual iteration may converge faster. For example, in Figure 31.4 we used 1089 data points (again taken from Franke's function) and a value of $\varepsilon = 16$. On the left we show the convergence behavior for iterated approximate MLS with Gaussian generating functions, and on the right for iterated Shepard approximation with Gaussian weights. Both graphs contain the errors for RBF interpolation with Gaussian basis functions for comparison. We note that the approximate MLS iteration approaches the RBF interpolant faster than in the previous examples. Moreover, both errors for iterated Shepard approximation are smaller than those for the interpolant after only three iterations. In fact, after 10 iterations the maximum error for the iterated Shepard approximation is one order of magnitude smaller than that for the RBF interpolant.

A detailed study of the dependence of the convergence of the fixed-level residual iteration algorithm on the shape parameter ε and more examples are provided in [Fasshauer and Zhang (2006)].

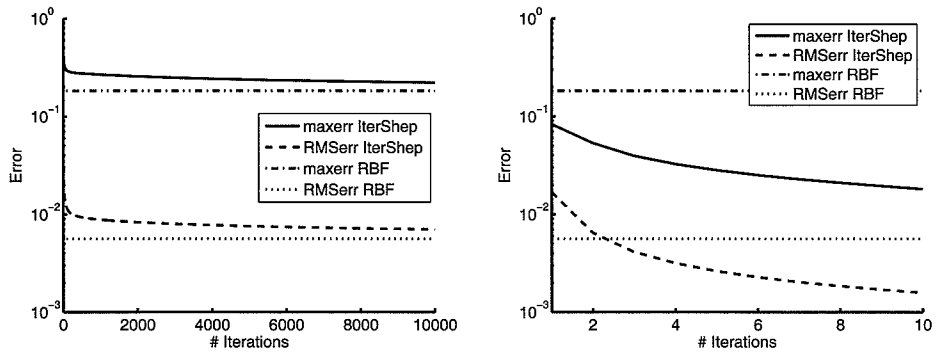

Fig. 31.4 Convergence for iterated approximate MLS approximation (left) and Shepard approximation (right) based on 1089 Halton points and Gaussian weights with $\varepsilon = 16$.

Chapter 32

Multilevel Iteration

As we saw in Chapters 12 and 16 that there is a trade-off principle for interpolation with compactly supported radial functions. Namely, using a non-stationary approach we can obtain good approximation results at the cost of increasing computational complexity, while with a stationary approach one has an efficient approximation method. However, stationary approximation with compactly supported radial functions is saturated, *i.e.*, it provides only limited convergence.

32.1 Stationary Multilevel Interpolation

In order to combine the advantages of both approaches for interpolation with compactly supported radial functions described above, Schaback suggested the use of a *multilevel* stationary scheme. This scheme was implemented first in [Floater and Iske (1996b)] and later studied by a number of other researchers (see, *e.g.*, [Chen *et al.* (2002); Fasshauer and Jerome (1999); Hales and Levesley (2002); Hartmann (1998); Iske (2001); Narcowich *et al.* (1999); Wendland (1999a)]).

In contrast to the fixed level iteration of the previous chapter we will now use a nested sequence $\mathcal{X}_1 \subset \cdots \subset \mathcal{X}_K = \mathcal{X} \subset \mathbb{R}^s$ of point sets with increasingly greater data density, *i.e.*, smaller fill distance $h_{\mathcal{X},\Omega}$. The basic idea of the stationary multilevel interpolation algorithm is to scale the size of the support of the basis functions with the fill distance, but to interpolate to residuals on progressively refined sets of centers. This method has all of the combined benefits of the interpolation methods for compactly supported RBFs referred to earlier: it is computationally efficient (can be performed in $\mathcal{O}(N)$ operations), well-conditioned, and appears to be convergent.

An algorithm for multilevel interpolation is as follows:

Algorithm 32.1. Stationary multilevel interpolation

(1) Create nested point sets $\mathcal{X}_1 \subset \cdots \subset \mathcal{X}_K = \mathcal{X} \subset \mathbb{R}^s$, and initialize $\mathcal{P}_f(\boldsymbol{x}) = 0$.
(2) For $k = 1, 2, \ldots, K$ do
  (a) Solve $u(\boldsymbol{x}) = f(\boldsymbol{x}) - \mathcal{P}_f(\boldsymbol{x})$ on $\mathcal{X}_k$.

(b) Update $\mathcal{P}_f(\boldsymbol{x}) \leftarrow \mathcal{P}_f(\boldsymbol{x}) + u(\boldsymbol{x})$.

The representation of the update u at step k is a radial basis function expansion of the form

$$u(\boldsymbol{x}) = \sum_{\boldsymbol{x}_j \in \mathcal{X}_k} c_j^{(k)} \varphi\left(\frac{\|\boldsymbol{x} - \boldsymbol{x}_j\|}{\rho_k}\right)$$

with φ a (compactly supported) basic function and its support scale $\rho_k \simeq h_{\mathcal{X}_k,\Omega}$. This requires the solution of a linear system whose size is determined by the number of points in $\mathcal{X}_k$.

Unfortunately, so far there are only limited theoretical results concerning the convergence of this multilevel algorithm. In [Narcowich *et al.* (1999)] the authors show that a related algorithm (in which additional boundary conditions are imposed) converges at least linearly. Hartmann analyzes the multilevel algorithm in his Ph.D. thesis [Hartmann (1998)]. He shows at least linear convergence for multilevel interpolation on a regular lattice for various radial basis functions. Similar results are obtained in [Hales and Levesley (2002)] for (globally supported) polyharmonic splines, *i.e.*, thin plate splines and radial powers. In this context linear convergence is to be interpreted as an improvement of the form

$$\|f - \mathcal{P}_f^{(k)}\| \le C\|f - \mathcal{P}_f^{(k-1)}\|, \qquad k = 1, \ldots, K,$$

where $\mathcal{P}_f^{(k)}$ denotes the interpolant at level k, and $C < 1$ is a positive (level-independent) constant.

The main difficulty in proving the convergence of the multilevel algorithm is the fact that the approximation space changes from one level to the next. The approximation spaces are not nested (as they usually are for wavelets). This means that the native space norm changes from one level to the next. Hales and Levesley avoid this problem by scaling the (uniformly spaced) data instead of the basis functions. Then the fact that polyharmonic splines are in a certain sense homogeneous (see Section 34.4) simplifies the analysis. This fact was also used in [Wendland (2005a)] to prove linear convergence for multilevel (scattered data) interpolation based on thin plate splines.

Another approach to multilevel interpolation was recently suggested by Opfer (see [Opfer (2004); Opfer (2006)]). He constructs so-called *multiscale kernels* that have the information from different resolution levels built into a single function. These kernels are built as tensor products of scaling functions that form a wavelet-like multiresolution analysis. Opfer provides error bounds for interpolation with these kernels analogous to those in Theorem 15.3. He also demonstrates how multiscale kernels can be used for scattered data interpolation, and for image decomposition and compression.

32.2 A MATLAB Implementation of Stationary Multilevel Interpolation

The MATLAB program `ML_CSRBF3D.m` for stationary multilevel interpolation with compactly supported RBFs is displayed as Program 32.1. In this program we use the compactly supported RBF $\varphi(r) = (1-r)_+^6(35r^2 + 18r + 3)$ of Wendland. This function is C^4 and strictly positive definite and radial on $\mathbb{R}^3$. The program shows a number of differences compared to our previous codes. On line 2 we define the maximum number of levels K we wish to use, and then define the corresponding scale parameters ε for the basic function. Since we load data sets consisting of $(2^k+1)^3$ uniformly spaced data points in the unit cube (where k runs from 1 to K, see lines 13–17), the fill distance $h_{\mathcal{X}_k,\Omega}$ changes by a factor of two from one level to the next. Therefore, in order to guarantee a stationary interpolation scheme, the scale parameter ε needs to change by a factor of two from one level to the next, also. This is achieved in line 2 of the code where we also use an additional factor of 0.7 to uniformly scale all values of ε. Note that here ε corresponds to a support radius $\rho = 1/\varepsilon$.

In each iteration we solve one interpolation problem (see line 18–22). The expansion coefficients `coef` are stored in one component of a MATLAB cell array. Similarly, we need to keep the centers for all levels in memory (again using a cell array `ctrs`, see line 17). The right-hand side for the interpolation problem is given by the data (values of the test function on the initial data set) in the first iteration, and in later iterations by the residual, *i.e.*, the difference between the data (values of the test function on the present grid) and the values of the fit on the present grid. These values are computed at the end of the previous iteration and stored in the vector `Rf` (see lines 28–32). This extra evaluation (in addition to the evaluation on a separate evaluation grid for error monitoring and plotting purposes, see lines 34–39) adds to the complexity (and inefficiency) of the present code. Moreover, the evaluation of the residual on the present grid requires us to keep evaluation matrices for all levels in memory (in the cell array `RM`, see lines 24–27). Note that the evaluation points (`respoints`) for the residuals change with every iteration and need not be kept in storage. An alternative approach would be to evaluate all residuals on a common (fine) grid, *e.g.*, the evaluation grid. This, however, would make (the evaluation of) the initial iterates rather expensive.

Finally, on lines 42–48 we keep track of the RMS error, and compute the rate of convergence from one level to the next. The commands that generate plots of the data sites, interpolant and error are given on lines 49–54.

Program 32.1. `ML_CSRBF3D.m`

```
% ML_CSRBF3D
% Script that performs multilevel RBF Interpolation using
% sparse matrices
```

```
% Calls on: DistancMatrixCSRBF
   % Wendland C4
 1 rbf = @(e,r) r.^6.*(35*r.^2-88*r+56*spones(r));
   % Number of levels with epsilons for stationary interpolation
 2 K = 4; ep = 0.7*2.^[0:K-1];
 3 testfunction = @(x,y,z) 64*x.*(1-x).*y.*(1-y).*z.*(1-z);
 4 gridtype = 'u'; % Type of data points: 'u'=uniform
 5 neval = 10; M = neval^3;
 6 grid=linspace(0,1,neval); [xe,ye,ze]=meshgrid(grid);
 7 epoints=[xe(:) ye(:) ze(:)];
 8 exact = testfunction(epoints(:,1),epoints(:,2),epoints(:,3));
 9 isomin = 0.1; isomax = 1; isostep = .1;
10 xslice = .25:.25:1; yslice = 1; zslice = [0,0.5];
11 Rf_old = zeros(27,1);    % initialize
12 for k=1:K
13    N1 = (2^k+1)^3;    N2 = (2^(k+1)+1)^3;
14    name1 = sprintf('Data3D_%d%s',N1,gridtype);
15    name2 = sprintf('Data3D_%d%s',N2,gridtype);
16    load(name2);     respoints = dsites;
17    load(name1);     ctrs{k} = dsites;
   % Compute right-hand side (= residual)
18    Tf = testfunction(dsites(:,1),dsites(:,2),dsites(:,3));
19    rhs = Tf - Rf_old;
20    DM_data = DistanceMatrixCSRBF(dsites,ctrs{k},ep(k));
21    IM = rbf(ep(k),DM_data);
      % Compute coefficients for RBF interpolant to detail level
22    coef{k} = IM\rhs;
23    if (k < K)
         % Compute - for all levels - evaluation matrices for
         % residuals directly
24       for j=1:k
25          DM_res = DistanceMatrixCSRBF(respoints,ctrs{j},ep(j));
26          RM{j} = rbf(ep(j),DM_res);
27       end
         % Evaluate RBF interpolant (sum of all previous fits
         % evaluated on current grid)
28       Rf = zeros(N2,1);
29       for j=1:k
30          Rf = Rf + RM{j}*coef{j};
31       end
32       Rf_old = Rf;
33    end
```



```
34    DM_eval = DistanceMatrixCSRBF(epoints,ctrs{k},ep(k));
35    EM = rbf(ep(k),DM_eval);
36    Pf = EM*coef{k};
37    if (k > 1)
38       Pf = Pf_old + Pf;
39    end
40    Pf_old = Pf;
41    maxerr = norm(Pf-exact,inf);
42    rms_err = norm(Pf-exact)/sqrt(M);
43    fprintf('RMS error:     %e\n', rms_err)
44    if (k > 1)
45       rms_rate = log(rms_err_old/rms_err)/log(2);
46       fprintf('RMS rate:      %f\n', rms_rate)
47    end
48    rms_err_old = rms_err;
49    figure
50    plot3(dsites(:,1),dsites(:,2),dsites(:,3),'bo');
51a   PlotIsosurf(xe,ye,ze,Pf,neval,exact,maxerr,isomin,...
51b        isostep,isomax);
52    PlotSlices(xe,ye,ze,Pf,neval,xslice,yslice,zslice);
53a   PlotErrorSlices(xe,ye,ze,Pf,exact,neval,...
53b        xslice,yslice,zslice);
54 end
```

In Figure 32.1 we display four data sets used in the 3D multilevel experiment. The corresponding 3D multilevel interpolants are shown as iso-surfaces in Figure 32.2. Note that the test function $f(x,y,z) = 64x(1-x)y(1-y)z(1-z)$ is a three-dimensional "bump" function, and therefore only the outermost iso-surface (corresponding to the function value 0.1) is visible. Therefore, we also display three-dimensional slice plots of the absolute error in Figure 32.3. Both the iso-surfaces and the slice plots are color coded according to the absolute error.

In Table 32.1 we list the corresponding RMS errors and observed convergence rates for the 3D multilevel experiment.

Table 32.1 3D stationary multilevel interpolation with $\varphi(r) = (1-r)_+^6(35r^2+18r+3)$.

mesh	RMS-error	rate	% nonzero	time
$3\times3\times3$	1.005315e-001		92.32	0.16
$5\times5\times5$	2.764907e-002	1.8623	36.99	0.56
$9\times9\times9$	2.626864e-003	3.3958	8.88	13.45
$17\times17\times17$	5.706061e-004	2.2028	1.57	73.50

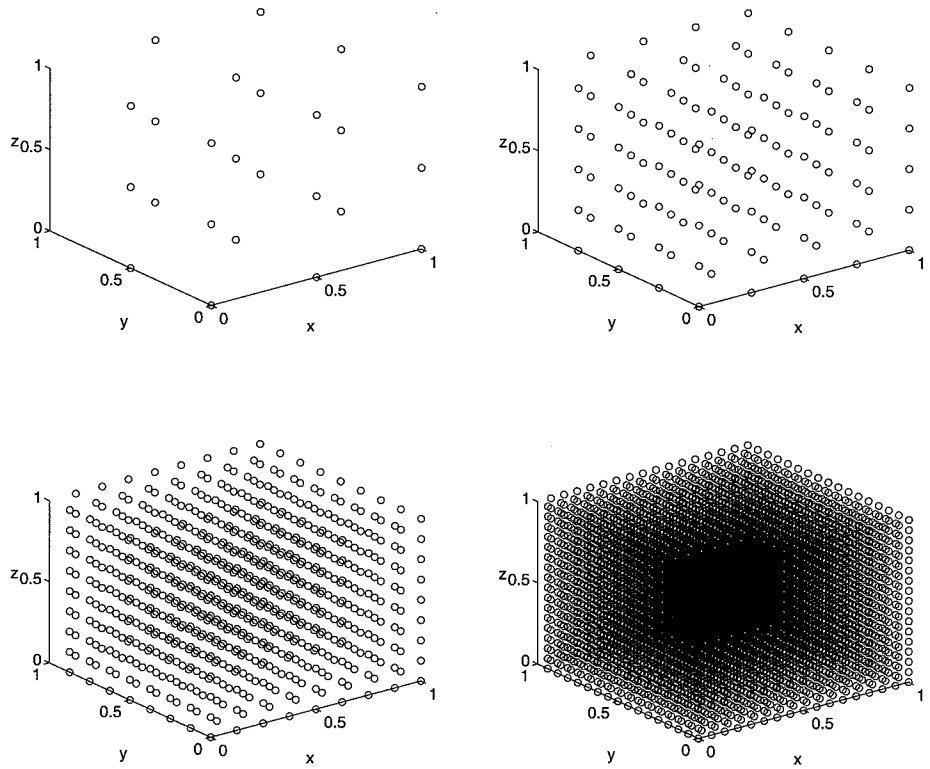

Fig. 32.1 Uniform point sets in the unit cube with 27, 125, 729, and 4913 points (top left to bottom right).

Next we present a two-dimensional multilevel interpolation experiment with scattered data. The data were obtained in the ASCII VRML format from `http://amba.charite.de/~ksch/spsm/beet1s.wrl.gz`. The data set provides a digitized surface model of a bust of the famous German composer Ludwig van Beethoven. The original data set consists of 2663 points in the unit square. In order to provide a nested set of data sites for the multilevel algorithm we use the program `Thin.m` provided in Appendix C. The processed data files are included on the enclosed CD. The resulting point sets are displayed in Figure 32.4. A much more detailed discussion of thinning algorithms for scattered data is presented in [Iske (2004)]. The MATLAB program used to generate the multilevel interpolants in Figure 32.5 is essentially the same as Program 32.1. The main difference is that the data values are also read from the data file instead of being generated by a test function. For rendering purposes the interpolants are evaluated on an 80×80 grid of equally spaced points in the unit square.

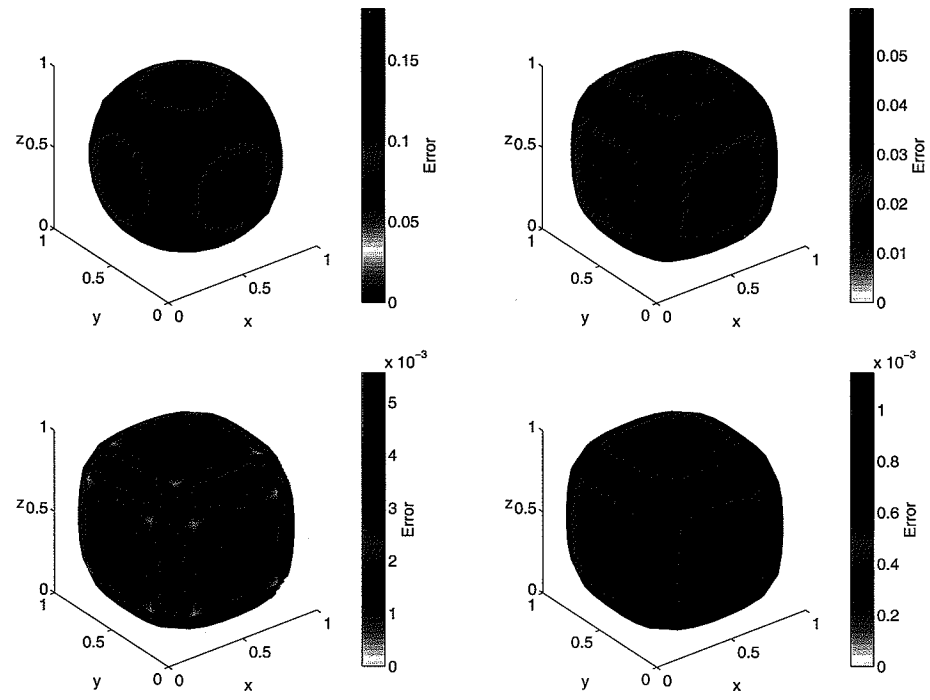

Fig. 32.2 Iso-surface plots for multilevel interpolants at levels 1, 2, 3 and 4 (top left to bottom right) false -colored by absolute error .

32.3 Stationary Multilevel Approximation

The same basic multilevel algorithm can also be used for other approximation methods. In [Fasshauer (2002c)] the idea was applied to moving least squares methods and approximate moving least squares methods. Experiments similar to those of [Fasshauer (2002c)] are now repeated here. In order to be able to provide a comparison between the multilevel interpolation and approximation algorithms we begin with one more example for multilevel interpolation.

We obtain the data for the following numerical examples by sampling a mollified Franke function f at uniformly spaced points in the unit square $[0,1]^2$, *i.e.*,

$$f(x,y) = 15 \exp\left(\frac{-1}{1-(2x-1)^2}\right) \exp\left(\frac{-1}{1-(2y-1)^2}\right) F(x,y),$$

where F denotes Franke's function (2.2).

In Table 32.2 we list the benchmark results for multilevel RBF interpolation with the compactly supported function $\varphi_{3,1}(r) = (1-r)_+^4 (4r+1)$. We again use an initial scale factor of 0.7 for the shape parameter ε. Since the shape parameter ε is equal to the reciprocal of the support scale ρ this means that the initial support scale ρ_1 is chosen so that the (univariate) basic function is fairly wide. Subsequent

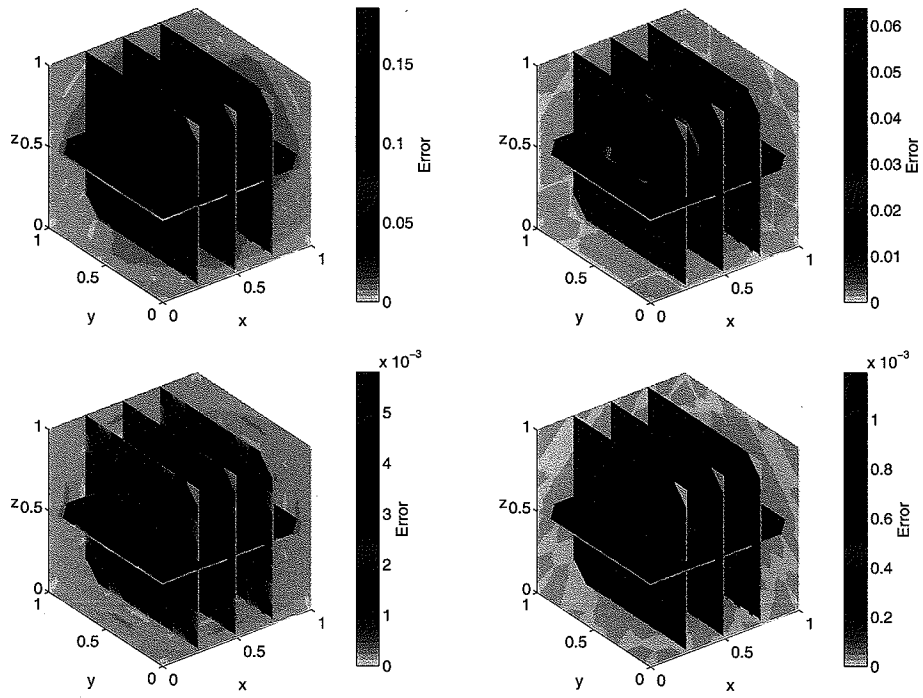

Fig. 32.3 Slice plots of absolute errors for multilevel interpolants at levels 1, 2, 3 and 4 (top left to bottom right).

support scales are successively divided by two (*i.e.*, ε is multiplied by two) — just as the fill distance is halved on successive computational grids $\mathcal{X}_k$.

Table 32.2 2D stationary multilevel interpolation with $\varphi(r) = (1-r)_+^4(4r+1)$ at equally spaced points in $[0,1]^2$.

mesh	RMS-error	rate	% nonzero	time
3×3	2.498505e-001		100	0.13
5×5	7.695304e-002	1.6990	57.76	0.16
9×9	2.092849e-002	1.8785	23.18	0.20
17×17	1.145664e-003	4.1912	7.47	0.42
33×33	1.376035e-004	3.0576	2.13	1.64
65×65	3.303559e-005	2.0584	0.57	7.98
129×129	2.149123e-006	3.9422	0.15	5.98

In Table 32.2 we list the RMS errors computed on a 40×40 uniform evaluation grid along with the percentage of non-zero entries in the RBF interpolation matrix and the computer time required for each iteration. The first four multilevel interpolants are shown in Figure 32.6.

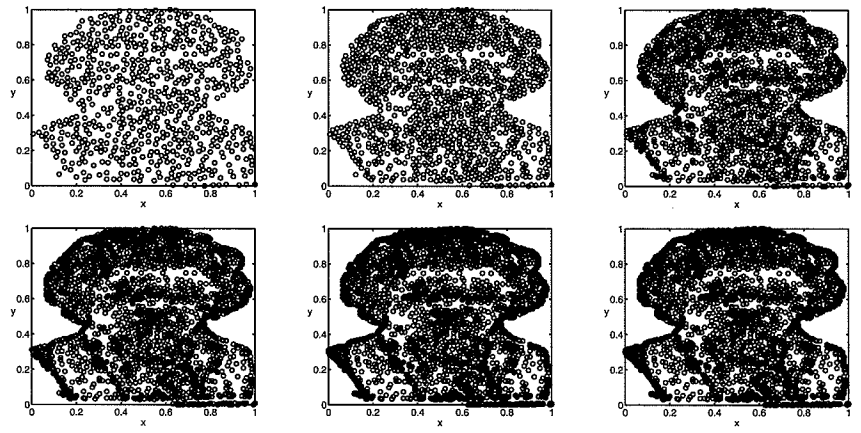

Fig. 32.4 Thinned data sets for Beethoven's head. For top left to bottom right: 163, 663, 1163, 1663, 2163, and 2663 points.

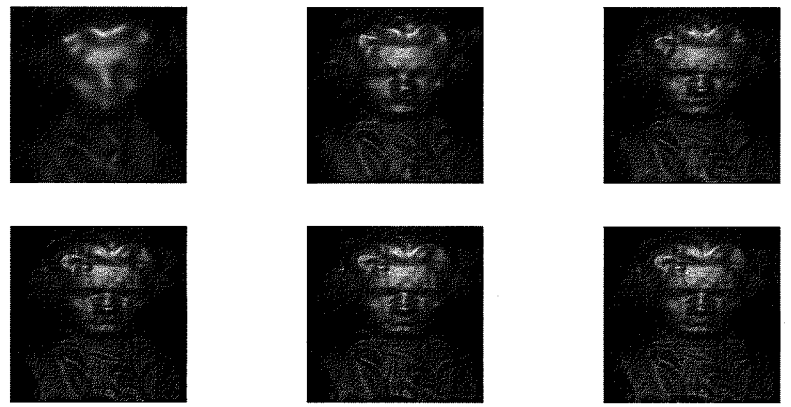

Fig. 32.5 Multilevel interpolants to Beethoven data. For top left to bottom right: 163, 663, 1163, 1663, 2163, and 2663 points.

Next we replace RBF interpolation at each step of the multilevel residual iteration algorithm by standard moving least squares approximation. Table 32.3 illustrates the performance of the multilevel algorithm for Shepard's method and a moving least squares approximation with linear precision, both based on the compactly supported weight function $\varphi_{3,1}(r) = (1-r)_+^4(4r+1)$. The support scaling is the same as in the previous multilevel interpolation example. The MATLAB code for these examples is omitted as it is very similar to that of Program 32.1. We note, however, that our implementation of the linear precision variant based on Program 24.3 is rather inefficient when compared to the other multilevel examples presented here.

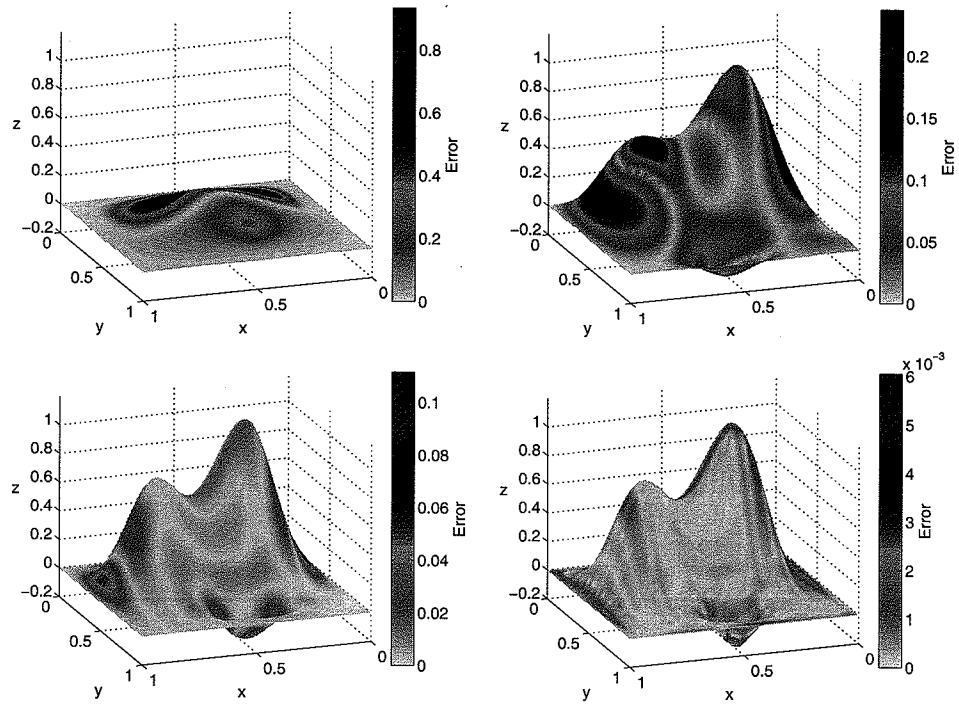

Fig. 32.6 The first four interpolants from Table 32.2.

There seems to be no theoretical investigation of the convergence properties of the multilevel algorithm for moving least squares approximation in the literature.

Table 32.3 2D multilevel MLS approximation with $\varphi(r) = (1-r)_+^4(4r+1)$.

	Shepard			linear precision		
mesh	RMS-error	rate	time	RMS-error	rate	time
3×3	2.776569e-001		0.16	2.812184e-001		0.91
5×5	1.615753e-001	0.7811	0.14	1.481365e-001	0.9248	0.97
9×9	7.519432e-002	1.1035	0.19	7.015497e-002	1.0783	1.36
17×17	1.858696e-002	2.0163	0.39	1.932368e-002	1.8602	3.09
33×33	3.581720e-003	2.3756	1.56	2.639418e-003	2.8721	11.45
65×65	5.458943e-004	2.7140	7.67	3.426412e-004	2.9454	89.64
129×129	1.047351e-004	2.3819	0.36	3.936786e-005	3.1216	1.17

As a third part of this example we use approximate MLS approximation at each level of the residual iteration algorithm. We use the generating functions $\Psi(r) = \frac{7}{\pi}(1-r)_+^4(4r+1)$ (giving rise to an approximate partition of unity) and $\Psi(r) = \frac{252}{229\pi}(1-r)_+^4(4r+1)(14-55r^2)$ (giving rise to an approximate partition of

unity with one vanishing moment). Recall that we constructed these functions in Example 26.2 starting with the initial weight function $\psi_0(y) = \left(1-\sqrt{y}\right)_+^4(4\sqrt{y}+1)$. As scale parameter we take $\mathcal{D} = 400/49$. This corresponds to the same scaling as in the other examples since $\varepsilon = 1/(\sqrt{\mathcal{D}}h)$ and the initial fill distance for the 3×3 grid is $h = 1/2$.

Table 32.4 Multilevel approximate MLS approximation with basic function $\psi_0(y) = (1-\sqrt{y})_+^4(4\sqrt{y}+1)$ and $\mathcal{D} = 400/49$.

	basic method			1 vanishing moment		
mesh	RMS-error	rate	time	RMS-error	rate	time
3×3	2.803247e-001		0.13	2.630763e-001		0.13
5×5	1.578062e-001	0.8289	0.16	9.133669e-002	1.5262	0.16
9×9	7.483597e-002	1.0764	0.20	2.783120e-002	1.7145	0.22
17×17	1.784522e-002	2.0682	0.39	3.399671e-003	3.0332	0.45
33×33	2.468958e-003	2.8536	1.52	4.359882e-004	2.9630	1.81
65×65	3.637815e-004	2.7628	7.30	7.856778e-005	2.4723	9.05
129×129	5.636161e-005	2.6903	0.39	2.460906e-005	1.6747	0.38

If we compare the numbers from the three different approaches listed in Tables 32.2–32.4 we see that none of the approximation methods yield more accurate results than the interpolation method. It is surprising, however, that the approximate MLS methods perform better than the regular MLS methods. For noisy data the approximate MLS method would be preferable.

32.4 Multilevel Interpolation with Globally Supported RBFs

So far we have concentrated on the use of compactly supported functions within the multilevel residual iteration algorithm. For globally supported functions we learned in earlier chapters that we can obtain good approximation order estimates in the non-stationary setting. One reason for our focus on compactly supported functions in this chapter is that if we consider the use of globally supported functions in a non-stationary multilevel interpolation framework, then we see that nothing is gained by the multilevel approach (provided we limit ourselves to the solution of linear problems such as the interpolation problems discussed above).

More precisely, if the meshes $\mathcal{X}_k$ at the different levels are nested and the shape parameter ε in the globally supported functions is kept fixed through all levels k, then the function spaces $\mathcal{S}_k = \operatorname{span}\{\varphi(\|\cdot - \boldsymbol{x}_j^{(k)}\|) : \boldsymbol{x}_j^{(k)} \in \mathcal{X}_k\}$ are also nested. Consequently, the richest space $\mathcal{S} = \bigcup_{k=1}^K \mathcal{S}_k$, which is used when all updates have been performed at the finest level, is equal to the space $\mathcal{S}_K$ on the finest mesh $\mathcal{X}_K$. Thus, a direct fit at the finest level K uses the same approximation space, and will therefore yield the same quality of fit, as the multilevel algorithm using all of the meshes $\mathcal{X}_k$, $k = 1, \ldots, K$. However, the multilevel algorithm requires all the

additional (unnecessary) intermediate work on the coarser meshes.

It therefore follows

Theorem 32.1 (Rule 1). *Consider a linear problem of the form $Lu = f$ on $\Omega \subseteq \mathbb{R}^s$, let $\mathcal{X}_1, \ldots, \mathcal{X}_K$ be a nested sequence of point sets in Ω, and let φ be a globally supported RBF with fixed shape parameter ε for all $k = 1, \ldots, K$. Then the approximate solution u_K obtained by the multilevel interpolation algorithm is the same as the solution of the problem $Lu = f$ on the fixed level $\mathcal{X}_K$ using the space $\mathcal{S}_K$, i.e., the use of globally supported RBFs with fixed value of ε within a multilevel residual iteration algorithm for linear problems is pointless.*

Here L could even be a linear differential operator (as used in later chapters). However, for the present discussion of interpolation problems we are only interested in $L = I$.

If one varies the parameter ε with the levels k (*i.e.*, one departs from the non-stationary regime) then the function space $\mathcal{S} = \bigcup_{k=1}^{K} \mathcal{S}_k$ used for the final fit with a multilevel algorithm will be richer than the space $\mathcal{S}_K$ used directly for the finest level $\mathcal{X}_K$ alone. This is clear since the spaces $\mathcal{S}_k$, $k = 1, \ldots, K$, are no longer nested. This implies that, for a "good" sequence of ε-values, one can expect to obtain more accurate fits using the multilevel framework.

This is summarized in

Corollary 32.1 (Rule 2). *The multilevel residual iteration algorithm for linear problems has the potential of being more accurate than a direct fit if the parameter ε is varied with the levels.*

We now illustrate Rules 1 and 2 with a scattered data fitting problem in $\mathbb{R}^2$. We use Franke's function (2.2) on the unit square as our test function. We take the (radial) basic function to be a multiquadric $\varphi(r) = \sqrt{1 + (\varepsilon r)^2}$. The point sets $\mathcal{X}_k$ are given by $(2^k + 1)^2$ equally spaced points in the unit square and are therefore nested with fill-distance $h_{\mathcal{X}_k,\Omega} = 1/2^k$.

In all of our numerical examples we list RMS-errors calculated on a fine evaluation mesh of 40×40 uniformly spaced points in the unit square.

For the first example we fix the multiquadric shape parameter at $\varepsilon = 10/3$ throughout. The errors and rates in Table 32.5 indicate the well-known spectral convergence behavior of multiquadrics. The last row in the table also shows that the parameter ε is too small for this point set and the matrix is so ill-conditioned that the approximation is starting to be contaminated by roundoff errors. According to Rule 1 there is no difference between using the multilevel algorithm and a direct fit on N points. This can also be observed numerically.

In order to illustrate Rule 2 we repeat the above example, but now take $\varepsilon = \sqrt{N}/2$ (essentially a stationary approach). This time we list what happens with the multilevel algorithm and compare this to the results obtained by computing the

Table 32.5 MQ fit to Franke's function with fixed value of $\varepsilon = 10/3$.

mesh	RMS-error	rate
3×3	1.802052e-001	
5×5	2.807009e-002	2.6825
9×9	4.009608e-003	2.8075
17×17	3.885488e-005	6.6892
33×33	2.294910e-008	10.7254
65×65	1.511150e-008	0.6028

approximation directly in one step on the sets $\mathcal{X}_k$, $k = 1, \ldots, 6$ (*c.f.* Table 32.6).

Table 32.6 Multilevel MQ and direct fits to Franke's function with variable value of $\varepsilon = \sqrt{N}/2$.

		multilevel		direct	
mesh	ε	RMS-error	rate	RMS-error	rate
3×3	.667	1.847122e-001		1.847122e-001	
5×5	.400	3.128037e-002	2.5619	3.105411e-002	2.5724
9×9	.222	4.288046e-003	2.8669	4.036275e-003	2.9437
17×17	.118	1.598555e-004	4.7455	1.004206e-004	5.3289
33×33	.061	1.187541e-005	3.7507	1.919679e-005	2.3871
65×65	.031	1.310485e-006	3.1798	4.700632e-006	2.0299

Note that, with the varying parameter ε, up to 289 points the direct approach is more accurate, but then the multilevel approach does a better job. This is due to the fact that the matrices for the denser point sets become increasingly ill-conditioned (even with the adjusted ε-value) and therefore the direct fits with 1089 or 4225 points are likely to be inaccurate. With the multilevel algorithm the fits on the finer grids act only as "corrections" to the coarse grid fits computed earlier. This agrees with the philosophy of the multilevel algorithm, and here is where the richer function space pays off.

Finally, it is also possible to combine the fixed level iteration of the previous chapter with the multilevel iteration of this chapter, *i.e.*, we can replace the interpolation steps in the multilevel scheme by fixed level iteration of an appropriate approximation method. We do not report any such experiments here.

Chapter 33

Adaptive Iteration

The two adaptive algorithms discussed in this chapter were both conceived to yield an approximate solution to the RBF interpolation problem. However, they have some similarity with the least squares knot insertion algorithm of Chapter 21 as well as with the iterative algorithms of the previous two chapters. The contents of this chapter are based mostly on the papers [Faul and Powell (1999); Faul and Powell (2000); Schaback and Wendland (2000a); Schaback and Wendland (2000b)] and the book [Wendland (2005a)].

33.1 A Greedy Adaptive Algorithm

We concentrate on systems for strictly positive definite functions (variations for strictly conditionally positive definite functions also exist). One of the central ingredients (and main differences to the previous iterative algorithms) is the use of the native space inner product discussed in Chapter 13. As always, we assume that our data sites are $\mathcal{X} = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N\}$, but now we also consider a second set $\mathcal{Y} \subseteq \mathcal{X}$.

If we let $\mathcal{P}_f^{\mathcal{Y}}$ be the interpolant to f on $\mathcal{Y} \subseteq \mathcal{X}$, then $\langle f - \mathcal{P}_f^{\mathcal{Y}}, \mathcal{P}_f^{\mathcal{Y}} \rangle_{\mathcal{N}_\Phi(\Omega)} = 0$ by Lemma 18.1 (with $u = f$) and we obtain the energy split (see Corollary 18.1)

$$\|f\|^2_{\mathcal{N}_\Phi(\Omega)} = \|f - \mathcal{P}_f^{\mathcal{Y}}\|^2_{\mathcal{N}_\Phi(\Omega)} + \|\mathcal{P}_f^{\mathcal{Y}}\|^2_{\mathcal{N}_\Phi(\Omega)}.$$

One possible point of view is now to consider an iteration on residuals. To this end we pretend to start with our desired interpolant $r_0 = \mathcal{P}_f^{\mathcal{X}}$ on the entire set $\mathcal{X}$, and an appropriate sequence of sets $\mathcal{Y}_k$, $k = 0, 1, \ldots$ (we will discuss some possible heuristics for choosing these sets later). Then, just as in our earlier residual iterations, we iteratively define

$$r_{k+1} = r_k - \mathcal{P}_{r_k}^{\mathcal{Y}_k}, \qquad k = 0, 1, \ldots. \tag{33.1}$$

Now, the energy splitting identity with $f = r_k$ gives us

$$\|r_k\|^2_{\mathcal{N}_\Phi(\Omega)} = \|r_k - \mathcal{P}_{r_k}^{\mathcal{Y}_k}\|^2_{\mathcal{N}_\Phi(\Omega)} + \|\mathcal{P}_{r_k}^{\mathcal{Y}_k}\|^2_{\mathcal{N}_\Phi(\Omega)} \tag{33.2}$$

or, using the iteration formula (33.1),

$$\|r_k\|^2_{\mathcal{N}_\Phi(\Omega)} = \|r_{k+1}\|^2_{\mathcal{N}_\Phi(\Omega)} + \|r_k - r_{k+1}\|^2_{\mathcal{N}_\Phi(\Omega)}. \tag{33.3}$$

Therefore, using (33.1) and (33.3), we have the following telescoping sum for the partial sums of the norm of the residual updates $\mathcal{P}_{r_k}^{\mathcal{Y}_k}$

$$\begin{aligned}\sum_{k=0}^{K} \|\mathcal{P}_{r_k}^{\mathcal{Y}_k}\|_{\mathcal{N}_\Phi(\Omega)}^2 &= \sum_{k=0}^{K} \|r_k - r_{k+1}\|_{\mathcal{N}_\Phi(\Omega)}^2 \\ &= \sum_{k=0}^{K} \left\{ \|r_k\|_{\mathcal{N}_\Phi(\Omega)}^2 - \|r_{k+1}\|_{\mathcal{N}_\Phi(\Omega)}^2 \right\} \\ &= \|r_0\|_{\mathcal{N}_\Phi(\Omega)}^2 - \|r_{K+1}\|_{\mathcal{N}_\Phi(\Omega)}^2 \le \|r_0\|_{\mathcal{N}_\Phi(\Omega)}^2.\end{aligned}$$

This estimate shows that the sequence of partial sums is monotone increasing and bounded, and therefore convergent — even for a poor choice of the sets $\mathcal{Y}_k$. If we can show that the residuals r_k converge to zero, then we would have that the iteratively computed approximation

$$u_{K+1} = \sum_{k=0}^{K} \mathcal{P}_{r_k}^{\mathcal{Y}_k} = \sum_{k=0}^{K} (r_k - r_{k+1}) = r_0 - r_{K+1} \tag{33.4}$$

converges to the original interpolant $r_0 = \mathcal{P}_f^{\mathcal{X}}$.

While this residual iteration algorithm has some structural similarities with the fixed level algorithm of Chapter 31 we now are considering a way to efficiently compute the interpolant $\mathcal{P}_f^{\mathcal{X}}$ on some fine set $\mathcal{X}$ by using an iteration on subsets of the data. Earlier we approximated the interpolant by iterating an *approximation method* on the full data set, whereas now we are approximating the interpolant by iterating an *interpolation method* on nested (increasing) adaptively chosen subsets of the data.

The present method also has some similarities with the multilevel algorithms of Chapter 32. However, now we are interested in computing the interpolant $\mathcal{P}_f^{\mathcal{X}}$ on the set $\mathcal{X}$ based on a single function Φ, whereas earlier, our final interpolant was the result of using the spaces $\bigcup_{k=1}^{K} \mathcal{N}_{\Phi_k}(\Omega)$, where Φ_k was an appropriately scaled version of the basic function Φ. Moreover, the goal in Chapter 32 was to approximate f, not $\mathcal{P}_f$.

In order to prove convergence of the residual iteration, let us assume that we can find sets of points $\mathcal{Y}_k$ such that at step k at least some fixed percentage of the energy of the residual is picked up by its interpolant, *i.e.*,

$$\|\mathcal{P}_{r_k}^{\mathcal{Y}_k}\|_{\mathcal{N}_\Phi(\Omega)}^2 \ge \gamma \|r_k\|_{\mathcal{N}_\Phi(\Omega)}^2 \tag{33.5}$$

with some fixed $\gamma \in (0, 1]$. Then (33.3) and the iteration formula (33.1) imply

$$\|r_{k+1}\|_{\mathcal{N}_\Phi(\Omega)}^2 = \|r_k\|_{\mathcal{N}_\Phi(\Omega)}^2 - \|\mathcal{P}_{r_k}^{\mathcal{Y}_k}\|_{\mathcal{N}_\Phi(\Omega)}^2,$$

and therefore

$$\|r_{k+1}\|_{\mathcal{N}_\Phi(\Omega)}^2 \le \|r_k\|_{\mathcal{N}_\Phi(\Omega)}^2 - \gamma \|r_k\|_{\mathcal{N}_\Phi(\Omega)}^2 = (1-\gamma)\|r_k\|_{\mathcal{N}_\Phi(\Omega)}^2.$$

Applying this bound recursively yields

Theorem 33.1. *If the choice of sets $\mathcal{Y}_k$ satisfies (33.5), then the residual iteration (33.4) converges linearly in the native space norm, and after K steps of iterative refinement there is an error bound*

$$\|r_0 - u_K\|_{\mathcal{N}_\Phi(\Omega)}^2 = \|r_K\|_{\mathcal{N}_\Phi(\Omega)}^2 \le (1-\gamma)^K \|r_0\|_{\mathcal{N}_\Phi(\Omega)}^2.$$

This theorem has various limitations. In particular, the norm involves the function Φ which makes it difficult to find sets $\mathcal{Y}_k$ that satisfy (33.5). Moreover, the native space norm of the initial residual r_0 is not known, either. Therefore, using an equivalent discrete norm on the set $\mathcal{X}$, Schaback and Wendland establish an estimate of the form

$$\|r_0 - u_K\|_{\mathcal{X}}^2 \le \frac{C}{c} \left(1 - \delta \frac{c^2}{C^2}\right)^{K/2} \|r_0\|_{\mathcal{X}}^2,$$

where c and C are constants denoting the norm equivalence, *i.e.*,

$$c\|u\|_{\mathcal{X}} \le \|u\|_{\mathcal{N}_\Phi(\Omega)} \le C\|u\|_{\mathcal{X}}$$

for any $u \in \mathcal{N}_\Phi(\Omega)$, and where δ is a constant analogous to γ (but based on use of the discrete norm $\|\cdot\|_{\mathcal{X}}$ in (33.5)). In fact, any discrete ℓ_p norm on $\mathcal{X}$ can be used. In the implementation below we will use the maximum norm.

In [Schaback and Wendland (2000b)] a basic version of this algorithm — where the sets $\mathcal{Y}_k$ consist of a single point — is described and tested. The resulting approximation yields the *best K-term approximation* to the interpolant. This idea is related to the concept of *greedy approximation algorithms* (see, *e.g.*, [Temlyakov (1998)]) and *sparse approximation* (see, *e.g.*, [Girosi (1998)]).

If the set $\mathcal{Y}_k$ consists of only a single point $\boldsymbol{y}_k$, then the partial interpolant $\mathcal{P}_{r_k}^{\mathcal{Y}_k}$ is particularly simple, namely

$$\mathcal{P}_{r_k}^{\mathcal{Y}_k} = \beta \Phi(\cdot, \boldsymbol{y}_k)$$

with

$$\beta = \frac{r_k(\boldsymbol{y}_k)}{\Phi(\boldsymbol{y}_k, \boldsymbol{y}_k)}.$$

This follows immediately from the usual RBF expansion (which consists of only one term here) and the interpolation condition $\mathcal{P}_{r_k}^{\mathcal{Y}_k}(\boldsymbol{y}_k) = r_k(\boldsymbol{y}_k)$.

The point $\boldsymbol{y}_k$ is picked to be the point in $\mathcal{X}$ where the residual is largest, *i.e.*, $|r_k(\boldsymbol{y}_k)| = \|r_k\|_\infty$. This choice of "set" $\mathcal{Y}_k$ certainly satisfies the constraint (33.5) since Φ is strictly positive definite and therefore has its maximum at the origin (cf. Property (4) in Theorem 3.1). Moreover, the interpolation problem is (approximately) solved without having to invert any linear systems. The algorithm can be summarized as

Algorithm 33.1. Greedy one-point algorithm

Input data locations $\mathcal{X}$, associated values of f, tolerance tol > 0
Set initial residual $r_0 = \mathcal{P}_f^{\mathcal{X}}$, initialize $u_0 = 0$, $e = \infty$, $k = 0$

Choose starting point $\boldsymbol{y}_k \in \mathcal{X}$
While $e >$ tol do
  Set $\beta = \dfrac{r_k(\boldsymbol{y}_k)}{\Phi(\boldsymbol{y}_k, \boldsymbol{y}_k)}$
  For $1 \le i \le N$ do
    $r_{k+1}(\boldsymbol{x}_i) = r_k(\boldsymbol{x}_i) - \beta\Phi(\boldsymbol{x}_i, \boldsymbol{y}_k)$
    $u_{k+1}(\boldsymbol{x}_i) = u_k(\boldsymbol{x}_i) + \beta\Phi(\boldsymbol{x}_i, \boldsymbol{y}_k)$
  end
  Find $e = \max_{\mathcal{X}} |r_{k+1}|$ and the point $\boldsymbol{y}_{k+1}$ where it occurs
  Increment $k = k + 1$
end

A MATLAB implementation of the greedy one-point algorithm is presented as Program 33.1. The implementation is quite straightforward using our function `DistanceMatrix` in conjunction with the anonymous function `rbf` to compute both $\Phi(\boldsymbol{y}_k, \boldsymbol{y}_k)$ (on lines 18 and 19) and $\Phi(\boldsymbol{x}_i, \boldsymbol{y}_k)$ needed for the updates of the residual r_{k+1} and the approximation u_{k+1} on lines 21–24. The algorithm demands that we compute the residuals r_k on the data sites. On the other hand, the partial approximants u_k to the interpolant can be evaluated anywhere. If we choose to do this also at the data sites, then we are required to use a plotting routine that differs from our usual one (such as `trisurf` built on a triangulation of the data sites obtained with the help of `delaunayn`). We instead choose to follow the same procedure as in all of our other programs, *i.e.*, to evaluate u_k on a 40×40 grid of equally spaced points. This has been implemented on lines 21–25 of the program. Note that the updating procedure has been vectorized in MATLAB allowing us to avoid the for-loop over i in the algorithm.

It is important to realize that we never actually compute the initial residual $r_0 = \mathcal{P}_f^{\mathcal{X}}$. All we require are the *values* of r_0 on the grid $\mathcal{X}$ of data sites. However, since $\mathcal{P}_f^{\mathcal{X}}|_{\mathcal{X}} = f|_{\mathcal{X}}$ the values $r_0(\boldsymbol{x}_i)$ are given by the interpolation data $f(\boldsymbol{x}_i)$ (see line 13 of the code). Moreover, since the sets $\mathcal{Y}_k$ are subsets of $\mathcal{X}$ the value $r_k(\boldsymbol{y}_k)$ required to determine β is actually one of the current residual values (see line 20 of the code).

The final approximation to the interpolant and the approximation error are plotted with the commands given on lines 36–38. The commands for the plots of the points $\boldsymbol{y}_k$ selected by the algorithm and the norm of the residual displayed in Figures 33.1 and 33.3 are included on lines 39 and 40.

Program 33.1. `RBFGreedyOnePoint2D.m`

```
% RBFGreedyOnePoint2D
% Script that performs greedy one point algorithm for adaptive
% 2D RBF interpolation
% Calls on: DistanceMatrix
```



```
1  rbf = @(e,r) exp(-(e*r).^2);      % Gaussian RBF
2  ep = 5.5;  % Parameter for basis function
   % Define Franke's function as testfunction
3  f1 = @(x,y) 0.75*exp(-((9*x-2).^2+(9*y-2).^2)/4);
4  f2 = @(x,y) 0.75*exp(-((9*x+1).^2/49+(9*y+1).^2/10));
5  f3 = @(x,y) 0.5*exp(-((9*x-7).^2+(9*y-3).^2)/4);
6  f4 = @(x,y) 0.2*exp(-((9*x-4).^2+(9*y-7).^2));
7  testfunction = @(x,y) f1(x,y)+f2(x,y)+f3(x,y)-f4(x,y);
   % Number and type of data points
8  N = 16641; gridtype = 'h';
9  neval = 40; grid = linspace(0,1,neval);
10 [xe,ye] = meshgrid(grid); epoints = [xe(:) ye(:)];
   % Tolerance; stopping criterion
11 tol = 1e-5; kmax = 1000;
   % Load data points
12 name = sprintf('Data2D_%d%s',N,gridtype); load(name)
   % Initialize residual and fit
13 r_old = testfunction(dsites(:,1),dsites(:,2));
14 u_old = 0;
15 k = 1; maxres(k) = 999999;
   % Use an (arbitrary) initial point
16 ykidx = (N+1)/2; yk(k,:) = dsites(ykidx,:);
17 while (maxres(k) > tol && k < kmax)
      % Evaluate basis function at yk
18    DM_data = DistanceMatrix(yk(k,:),yk(k,:));
19    IM = rbf(ep,DM_data);
20    beta = r_old(ykidx)/IM;
      % Compute evaluation matrices for residual and fit
21    DM_res = DistanceMatrix(dsites,yk(k,:));
22    RM = rbf(ep,DM_res);
23    DM_eval = DistanceMatrix(epoints,yk(k,:));
24    EM = rbf(ep,DM_eval);
      % Update residual and fit
25    r = r_old - beta*RM;     u = u_old + beta*EM;
      % Find new point to add
26    [sr,idx] = sort(abs(r));
27    maxres(k+1) = sr(end);
28    ykidx = idx(end);     yk(k+1,:) = dsites(ykidx,:);
29    r_old = r;     u_old = u;
30    k = k + 1;
31 end
   % Compute exact solution
```

```
exact = testfunction(epoints(:,1),epoints(:,2));
maxerr = norm(u-exact,inf); rms_err = norm(u-exact)/neval;
fprintf('RMS error:     %e\n', rms_err)
fprintf('Maximum error: %e\n', maxerr)
fview = [160,20]; % viewing angles for plot
PlotSurf(xe,ye,u,neval,exact,maxerr,fview);
PlotError2D(xe,ye,u,exact,maxerr,neval,fview);
figure; plot(yk(:,1),yk(:,2),'ro')
figure; semilogy(maxres,'b');
```

To illustrate the greedy one-point algorithm we perform two experiments. Both tests use data obtained by sampling Franke's function at 16641 Halton points in $[0,1]^2$. However, the first test is based on Gaussians, while the second one uses inverse multiquadrics. For both tests we use the same shape parameter $\varepsilon = 5.5$. This results in the inverse multiquadrics having a more global influence than the Gaussians. This effect is clearly evident in the first few approximations to the interpolants in Figures 33.2 and 33.4.

Figure 33.4, in particular, shows that the greedy algorithm enforces interpolation of the data only on the most recent set $\mathcal{Y}_k$ (*i.e.*, for the one-point algorithm studied here only at a single point). If one wants to maintain the interpolation achieved in previous iterations, then the sets $\mathcal{Y}_k$ should be nested. This, however, would have a significant effect on the execution time of the algorithm since the matrices at each step would increase in size.

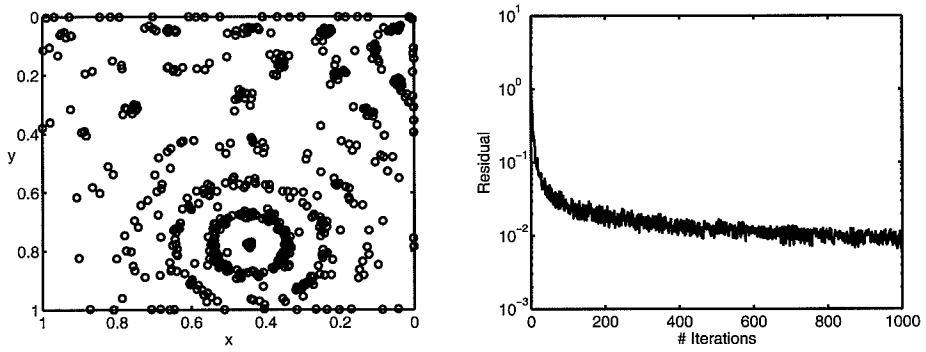

Fig. 33.1 1000 selected points and residual for greedy one point algorithm with Gaussian RBFs and $N = 16641$ data points.

In order to obtain our approximate interpolants we used a tolerance of 10^{-5} along with an additional upper limit of `kmax=1000` on the number of iterations. For both tests the algorithm uses up all 1000 iterations. The final maximum residual for Gaussians is `maxres` $= 0.0075$, while for inverse MQs we have `maxres` $= 0.0035$. In both cases there occurred several multiple point selections. Contrary to interpolation problems based on the solution of a linear system, multiple point selections do not pose a problem here.

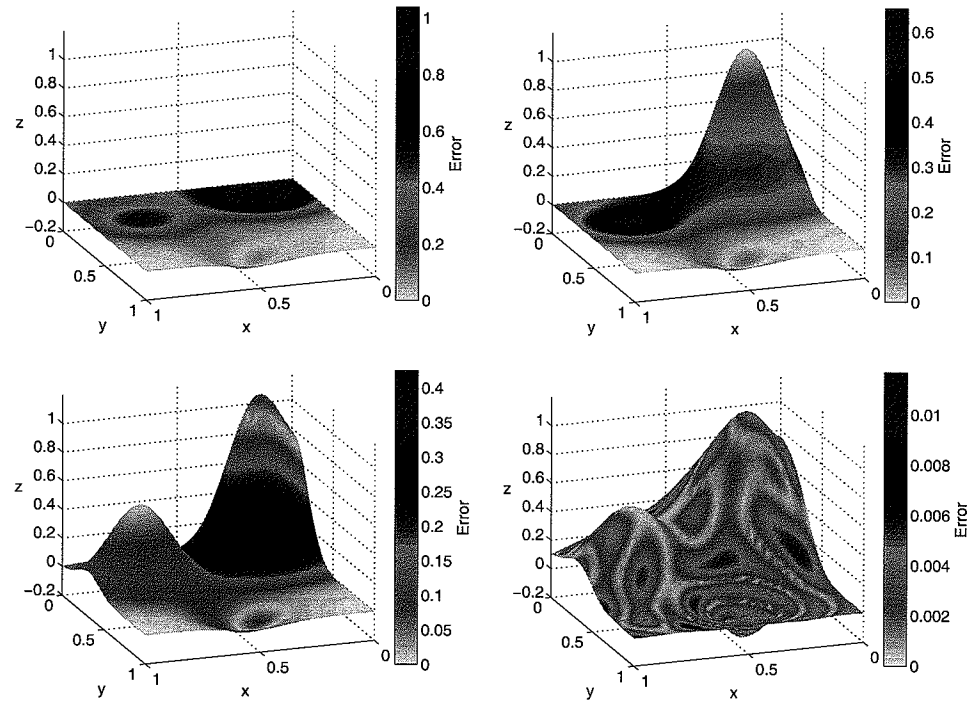

Fig. 33.2 Fits of Franke's function for greedy one point algorithm with Gaussian RBFs and $N = 16641$ data points. Top left to bottom right: 1 point, 2 points, 4 points, final fit with 1000 points.

One advantage of this very simple algorithm is that no linear systems need to be solved. This allows us to approximate the interpolants for large data sets even for globally supported basis functions, and also with small values of ε (and therefore an associated ill-conditioned interpolation matrix). One should not expect too much in this case, however, as the results in Figure 33.5 show where we used a value of $\varepsilon = 0.1$ for the shape parameter. As with the fixed level iteration of approximate MLS approximants based on flat generating functions, a lot of smoothing occurs so that the convergence to the RBF interpolant is very slow.

Moreover, in the pseudo-code of the algorithm matrix-vector multiplications are not required, either. However, MATLAB allows for a vectorization of the for-loop which does result in two matrix-vector multiplications.

For practical situations, *e.g.*, for smooth radial basis functions and densely distributed points in $\mathcal{X}$ the convergence can be rather slow. The simple greedy algorithm described above is extended in [Schaback and Wendland (2000b)] to a version that adaptively uses basis functions of varying scales.

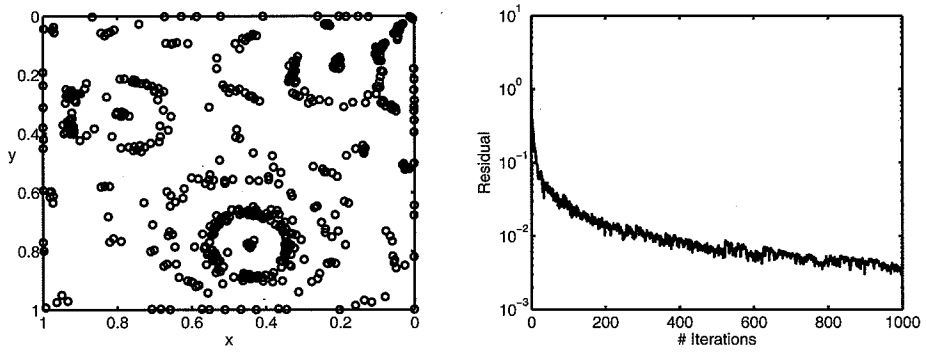

Fig. 33.3 1000 selected points and residual for greedy one point algorithm with IMQ RBFs and $N = 16641$ data points.

33.2 The Faul-Powell Algorithm

Another iterative algorithm was suggested in [Faul and Powell (1999); Faul and Powell (2000)]. From our earlier discussions we know that it is possible to express the radial basis function interpolant in terms of cardinal functions u_j^*, $j = 1, \ldots, N$, *i.e.*,

$$\mathcal{P}_f(\boldsymbol{x}) = \sum_{j=1}^{N} f(\boldsymbol{x}_j) u_j^*(\boldsymbol{x}).$$

The basic idea of the Faul-Powell algorithm is to use *approximate cardinal functions* Ψ_j instead. Of course, this will only give an approximate value for the interpolant, and therefore an iteration on the residuals is suggested to improve the accuracy of this approximation.

The basic philosophy of this algorithm is very similar to that of the fixed level iteration of Chapter 31. In particular, the Faul-Powell algorithm can also be interpreted as a Krylov subspace method. However, instead of taking approximate MLS generating functions, the approximate cardinal functions Ψ_j, $j = 1, \ldots, N$, are determined as linear combinations of the basis functions $\Phi(\cdot, \boldsymbol{x}_\ell)$ for the interpolant, *i.e.*,

$$\Psi_j = \sum_{\ell \in \mathcal{L}_j} b_{j\ell} \Phi(\cdot, \boldsymbol{x}_\ell), \tag{33.6}$$

where $\mathcal{L}_j$ is an index set consisting of n ($n \approx 50$) indices that are used to determine the approximate cardinal function. For example, the n nearest neighbors of $\boldsymbol{x}_j$ will usually do. In general, the choice of index sets allows much freedom, and this is the reason why we include the algorithm in this chapter on adaptive iterative methods. Also, as pointed out at the end of this section, there is a certain duality between the Faul-Powell algorithm and the greedy algorithm of the previous section.

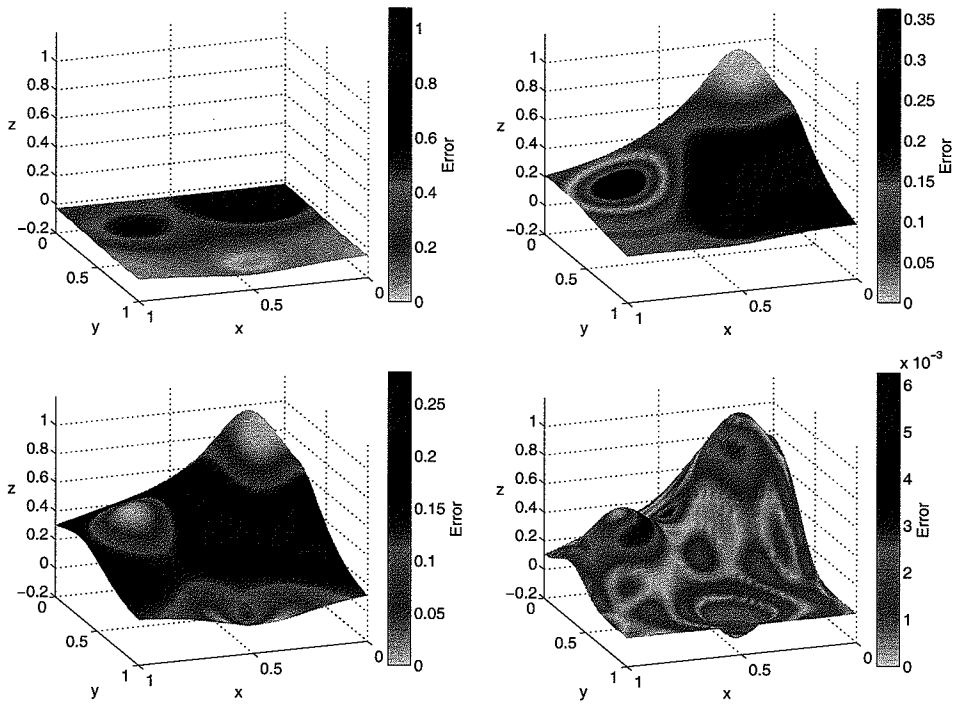

Fig. 33.4 Fits of Franke's function for greedy one point algorithm with IMQ RBFs and $N = 16641$ data points. Top left to bottom right: 1 point, 2 points, 4 points, final fit with 1000 points.

For every $j = 1, \ldots, N$, the coefficients $b_{j\ell}$ are found as solution of the (relatively small) $n \times n$ linear system

$$\Psi_j(\boldsymbol{x}_i) = \delta_{jk}, \qquad i \in \mathcal{L}_j. \tag{33.7}$$

These approximate cardinal functions are computed in a pre-processing step.

As before, in its simplest form the residual iteration can be formulated as

$$u^{(0)}(\boldsymbol{x}) = \sum_{j=1}^{N} f(\boldsymbol{x}_j) \Psi_j(\boldsymbol{x})$$

$$u^{(k+1)}(\boldsymbol{x}) = u^{(k)}(\boldsymbol{x}) + \sum_{j=1}^{N} \left[f(\boldsymbol{x}_j) - u^{(k)}(\boldsymbol{x}_j) \right] \Psi_j(\boldsymbol{x}), \qquad k = 0, 1, \ldots.$$

Instead of adding the contribution of all approximate cardinal functions at the same time, this is done in a three-step process in the Faul-Powell algorithm. To this end, we choose index sets $\mathcal{L}_j$, $j = 1, \ldots, N-n$, such that $\mathcal{L}_j \subseteq \{j, j+1, \ldots, N\}$ while making sure that $j \in \mathcal{L}_j$. Also, if one wants to use this algorithm to approximate the interpolant based on conditionally positive definite functions of order m, then one needs to ensure that the corresponding centers form an $(m-1)$-unisolvent set and append a polynomial to the local expansion (33.6).

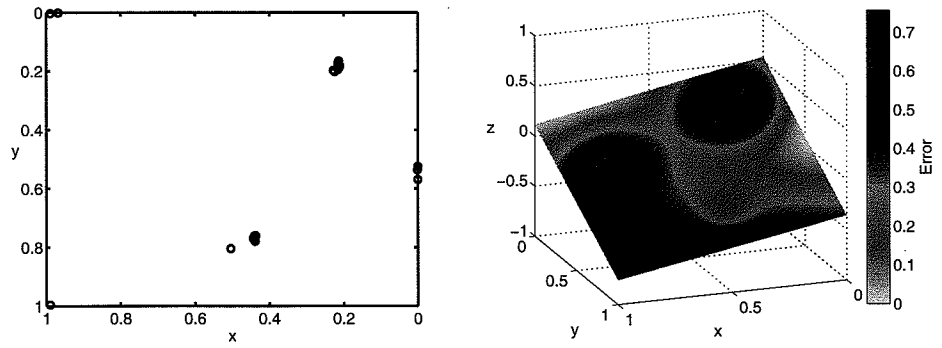

Fig. 33.5 1000 selected points (only 20 of them distinct) and fit of Franke's function for greedy one point algorithm with flat Gaussian RBFs ($\varepsilon = 0.1$) and $N = 16641$ data points.

Now, in the first step we define $u_0^{(k)} = u^{(k)}$, and then iterate

$$u_j^{(k)} = u_{j-1}^{(k)} + \theta_j^{(k)} \Psi_j, \qquad j = 1, \ldots, N-n, \tag{33.8}$$

with

$$\theta_j^{(k)} = \frac{\langle \mathcal{P}_f - u_{j-1}^{(k)}, \Psi_j \rangle_{\mathcal{N}_\Phi(\Omega)}}{\langle \Psi_j, \Psi_j \rangle_{\mathcal{N}_\Phi(\Omega)}}. \tag{33.9}$$

The stepsize $\theta_j^{(k)}$ is chosen so that the native space best approximation to the residual $\mathcal{P}_f - u_{j-1}^{(k)}$ from the space spanned by the approximate cardinal functions Ψ_j is added. Using the representation (33.6) of Ψ_j in terms of the global basis $\{\Phi(\cdot, \boldsymbol{x}_i) : \ i = 1, \ldots, N\}$, the reproducing kernel property of Φ, and the (local) cardinality property (33.7) of Ψ_j we can calculate the denominator of (33.9) as

$$\begin{aligned}
\langle \Psi_j, \Psi_j \rangle_{\mathcal{N}_\Phi(\Omega)} &= \langle \Psi_j, \sum_{\ell \in \mathcal{L}_j} b_{j\ell} \Phi(\cdot, \boldsymbol{x}_\ell) \rangle_{\mathcal{N}_\Phi(\Omega)} \\
&= \sum_{\ell \in \mathcal{L}_j} b_{j\ell} \langle \Psi_j, \Phi(\cdot, \boldsymbol{x}_\ell) \rangle_{\mathcal{N}_\Phi(\Omega)} \\
&= \sum_{\ell \in \mathcal{L}_j} b_{j\ell} \Psi_j(\boldsymbol{x}_\ell) = b_{jj}
\end{aligned}$$

since we have $j \in \mathcal{L}_j$ by construction of the index set $\mathcal{L}_j$. Similarly, we get for the numerator

$$\begin{aligned}
\langle \mathcal{P}_f - u_{j-1}^{(k)}, \Psi_j \rangle_{\mathcal{N}_\Phi(\Omega)} &= \langle \mathcal{P}_f - u_{j-1}^{(k)}, \sum_{\ell \in \mathcal{L}_j} b_{j\ell} \Phi(\cdot, \boldsymbol{x}_\ell) \rangle_{\mathcal{N}_\Phi(\Omega)} \\
&= \sum_{\ell \in \mathcal{L}_j} b_{j\ell} \langle \mathcal{P}_f - u_{j-1}^{(k)}, \Phi(\cdot, \boldsymbol{x}_\ell) \rangle_{\mathcal{N}_\Phi(\Omega)} \\
&= \sum_{\ell \in \mathcal{L}_j} b_{j\ell} \left(\mathcal{P}_f - u_{j-1}^{(k)} \right) (\boldsymbol{x}_\ell)
\end{aligned}$$

$$= \sum_{\ell \in \mathcal{L}_j} b_{j\ell} \left(f(\boldsymbol{x}_\ell) - u_{j-1}^{(k)}(\boldsymbol{x}_\ell) \right).$$

Therefore (33.8) and (33.9) can be written as

$$u_j^{(k)} = u_{j-1}^{(k)} + \frac{\Psi_j}{b_{jj}} \sum_{\ell \in \mathcal{L}_j} b_{j\ell} \left(f(\boldsymbol{x}_\ell) - u_{j-1}^{(k)}(\boldsymbol{x}_\ell) \right), \qquad j = 1, \ldots, N-n.$$

In the second step of the Faul-Powell algorithm the residual is interpolated on the remaining n points (collected via the index set $\mathcal{L}^*$). Thus, we find a function $v^{(k)}$ in $\operatorname{span}\{\Phi(\cdot, \boldsymbol{x}_j) : \ j \in \mathcal{L}^*\}$ such that

$$v^{(k)}(\boldsymbol{x}_i) = f(\boldsymbol{x}_i) - u_{N-n}^{(k)}(\boldsymbol{x}_i), \qquad i \in \mathcal{L}^*,$$

and the approximation is updated, *i.e.*,

$$u^{(k+1)} = u_{N-n}^{(k)} + v^{(k)}.$$

In the third step the residuals are updated, *i.e.*,

$$r_i^{(k+1)} = f(\boldsymbol{x}_i) - u^{(k+1)}(\boldsymbol{x}_i), \qquad i = 1, \ldots, N. \tag{33.10}$$

The outer iteration (on k) is now repeated unless the largest of these residuals is small enough.

We can summarize this algorithm as

Algorithm 33.2. Faul-Powell algorithm

Input data locations $\mathcal{X} = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N\}$, associated values of f, and tolerance $\text{tol} > 0$

Pre-processing step

Choose n

For $1 \le j \le N-n$ do

Determine the index set $\mathcal{L}_j$

Find the coefficients $b_{j\ell}$ of the approximate cardinal function Ψ_j by solving

$$\Psi_j(\boldsymbol{x}_i) = \delta_{jk}, \qquad i \in \mathcal{L}_j$$

end

Set $k = 0$ and $u_0^{(k)} = 0$

Initialize residuals $r_i^{(k)} = f(\boldsymbol{x}_i), \ i = 1, \ldots, N$

Set $e = \max_{i=1,\ldots,N} |r_i^{(k)}|$

While $e > \text{tol}$ do

For $1 \le j \le N-n$ do

Update

$$u_j^{(k)} = u_{j-1}^{(k)} + \frac{\Psi_j}{b_{jj}} \sum_{\ell \in \mathcal{L}_j} b_{j\ell} \left(f(\boldsymbol{x}_\ell) - u_{j-1}^{(k)}(\boldsymbol{x}_\ell) \right)$$

end
Solve the interpolation problem

$$v^{(k)}(\boldsymbol{x}_i) = f(\boldsymbol{x}_i) - u^{(k)}_{N-n}(\boldsymbol{x}_i), \qquad i \in \mathcal{L}^*$$

Update the approximation

$$u^{(k+1)}_0 = u^{(k)}_{N-n} + v^{(k)}$$

Compute new residuals $r_i^{(k+1)} = f(\boldsymbol{x}_i) - u_0^{(k+1)}(\boldsymbol{x}_i),\ i = 1, \ldots, N$
Set new value for $e = \max_{i=1,\ldots,N} |r_i^{(k+1)}|$
Increment $k = k + 1$

end

Faul and Powell prove that this algorithm converges to the solution of the original interpolation problem. Similar to some of the other algorithms, one needs to make sure that the residuals are evaluated efficiently by using, *e.g.*, a fast multipole expansion, fast Fourier transform, or compactly supported functions.

In its most basic form the Krylov subspace algorithm of Faul and Powell can also be explained as a dual approach to the greedy residual iteration algorithm of Schaback and Wendland. Instead of defining appropriate sets of points $\mathcal{Y}_k$, in the Faul and Powell algorithm one picks certain subspaces U_k of the native space. In particular, if U_k is the one-dimensional space $U_k = \text{span}\{\Psi_k\}$ (where Ψ_k is a local approximation to the cardinal function) we get the algorithm described above. For more details see [Schaback and Wendland (2000b)].

We leave the implementation of this algorithm to the reader.

Chapter 34

Improving the Condition Number of the Interpolation Matrix

In Chapter 16 we noted that the system matrices arising in scattered data interpolation with radial basis functions tend to become very ill-conditioned as the minimal separation distance $q_{\mathcal{X}}$ between the data sites $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N$, is reduced. Therefore it is natural to devise strategies to prevent such instabilities by either preconditioning the system, or by finding a better basis for the approximation space we are using. The former approach is standard procedure in numerical linear algebra, and in fact we can use any of the well-established methods (such as preconditioned conjugate gradient iteration) to improve the stability and convergence of the interpolation systems that arise for strictly positive definite functions. In particular, the sparse systems that arise in (multilevel) interpolation with compactly supported radial basis functions can be solved efficiently with the preconditioned conjugate gradient method. However, in our implementation (see the discussion in Section 12.1) we use MATLAB's `sparse` function which takes advantage of state-of-the-art direct methods for sparse linear systems.

The second approach to improving the condition number of the interpolation system, *i.e.*, the idea of using a more stable basis, is well known from univariate polynomial and spline interpolation. The Lagrange basis functions for univariate polynomial interpolation are the ideal basis if we are interested in stably solving the interpolation equations since the resulting interpolation matrix is the identity matrix (which is certainly much better conditioned than, *e.g.*, the Vandermonde matrix that we get if we use a monomial basis). Similarly, B-splines give rise to diagonally dominant, sparse system matrices which are much easier to deal with than the matrices we would get if we were to represent a spline interpolant using the alternative truncated power basis. Both of these examples are studied in great detail in standard numerical analysis texts (see, *e.g.*, [Kincaid and Cheney (2002)]) or in the literature on splines (see, *e.g.*, [Schumaker (1981)]). We will address an analogous approach for radial basis functions in Section 34.4 below.

Before we describe any of the specialized preconditioning procedures for radial basis function interpolation matrices we give two examples presented in the early RBF paper [Jackson (1989a)] to illustrate the effects of and motivation for preconditioning in the context of radial basis functions.

34.1 Preconditioning: Two Simple Examples

Example 34.1. Let $s = 1$ and consider interpolation based on $\varphi(r) = r$ with no polynomial terms added. As data sites we choose $\mathcal{X} = \{1, 2, \ldots, 10\}$. This leads to the system matrix

$$A = \begin{bmatrix} 0&1&2&3&\ldots&9 \\ 1&0&1&2&\ldots&8 \\ 2&1&0&1&\ldots&7 \\ 3&2&1&0&\ldots&6 \\ \vdots&\vdots&\vdots&\vdots&\ddots&\vdots \\ 9&8&7&6&\ldots&0 \end{bmatrix}$$

with ℓ_2-condition number $\text{cond}(A) \approx 67$. Instead of solving the linear system $A\boldsymbol{c} = \boldsymbol{y}$, where $\boldsymbol{y} = [y_1, \ldots, y_{10}]^T \in \mathbb{R}^{10}$ is a vector of given real numbers (data values), we can find a suitable matrix B to pre-multiply both sides of the equation such that the system is simpler to solve. Ideally, the new system matrix BA should be the identity matrix, *i.e.*, B should be an *approximate inverse* of A. Thus, having found an appropriate matrix B, we must now solve the linear system $BA\boldsymbol{c} = B\boldsymbol{y}$. The matrix B is usually referred to as the *preconditioner* of the linear system. For the matrix A above we can choose a preconditioner B as

$$B = \begin{bmatrix} 1&0&0&0&\ldots&0&0 \\ \frac{1}{2}&-1&\frac{1}{2}&0&\ldots&0&0 \\ 0&\frac{1}{2}&-1&\frac{1}{2}&\ldots&0&0 \\ 0&0&\frac{1}{2}&-1&\ldots&0&0 \\ \vdots&\vdots&\vdots&\vdots&\ddots&\vdots&\vdots \\ 0&0&0&0&\ldots&-1&\frac{1}{2} \\ 0&0&0&0&\ldots&0&1 \end{bmatrix}.$$

This leads to the following preconditioned system matrix

$$BA = \begin{bmatrix} 0&1&2&\ldots&8&9 \\ 0&1&0&\ldots&0&0 \\ 0&0&1&\ldots&0&0 \\ \vdots&\vdots&\vdots&\ddots&\vdots&\vdots \\ 0&0&0&\ldots&1&0 \\ 9&8&7&\ldots&1&0 \end{bmatrix}$$

in the system $BA\boldsymbol{c} = B\boldsymbol{y}$. Note that BA is almost an identity matrix. One can easily check that now $\text{cond}(BA) \approx 45$.

The motivation for this choice of B is the following. The function $\varphi(r) = r$ or $\Phi(x) = |x|$ is a fundamental solution of the Laplacian Δ ($= \frac{d^2}{dx^2}$ in the one-dimensional case), *i.e.*

$$\Delta\Phi(x) = \frac{d^2}{dx^2}|x| = \frac{1}{2}\delta_0(x),$$

where δ_0 is the Dirac delta function centered at zero. Thus, B is chosen as a discretization of the Laplacian with special choices at the endpoints of the data set.

Example 34.2. For non-uniformly distributed data we can use a different discretization of the Laplacian Δ for each row of B. To see this, let $s = 1$, $\mathcal{X} = \{1, \frac{3}{2}, \frac{5}{2}, 4, \frac{9}{2}\}$, and again consider interpolation with the radial function $\varphi(r) = r$. Then

$$A = \begin{bmatrix} 0&\frac{1}{2}&\frac{3}{2}&3&\frac{7}{2} \\ \frac{1}{2}&0&1&\frac{5}{2}&3 \\ \frac{3}{2}&1&0&\frac{3}{2}&2 \\ 3&\frac{5}{2}&\frac{3}{2}&0&\frac{1}{2} \\ \frac{7}{2}&3&2&\frac{1}{2}&0 \end{bmatrix}$$

with $\text{cond}(A) \approx 18.15$, and if we choose

$$B = \begin{bmatrix} 1&0&0&0&0 \\ 1&-\frac{3}{2}&\frac{1}{2}&0&0 \\ 0&\frac{1}{2}&-\frac{5}{6}&\frac{1}{3}&0 \\ 0&0&\frac{1}{3}&-\frac{4}{3}&1 \\ 0&0&0&0&1 \end{bmatrix},$$

based on second-order backward differences of the points in $\mathcal{X}$, then the preconditioned system to be solved becomes

$$\begin{bmatrix} 0&\frac{1}{2}&\frac{3}{2}&3&\frac{7}{2} \\ 0&1&0&0&0 \\ 0&0&1&0&0 \\ 0&0&0&1&0 \\ \frac{7}{2}&3&2&\frac{1}{2}&0 \end{bmatrix} \boldsymbol{c} = B\boldsymbol{y}.$$

Once more, this system is almost trivial to solve and has an improved condition number of $\text{cond}(BA) \approx 8.94$.

34.2 Early Preconditioners

Ill-conditioning of the interpolation matrices was identified as a serious problem very early, and Nira Dyn along with some of her co-workers (see, *e.g.*, [Dyn (1987); Dyn (1989); Dyn and Levin (1983); Dyn *et al.* (1986)]) provided some of the first preconditioning strategies tailored especially to radial basis function interpolants.

For the following discussion we consider the general interpolation problem that includes polynomial reproduction (see Chapter 6). Therefore, we have to solve the following system of linear equations

$$\begin{bmatrix} A & P \\ P^T & O \end{bmatrix} \begin{bmatrix} \boldsymbol{c} \\ \boldsymbol{d} \end{bmatrix} = \begin{bmatrix} \boldsymbol{y} \\ \boldsymbol{0} \end{bmatrix}, \tag{34.1}$$

with the individual pieces given by $A_{jk} = \varphi(\|\boldsymbol{x}_j - \boldsymbol{x}_k\|)$, $j,k = 1,\ldots,N$, $P_{j\ell} = p_\ell(\boldsymbol{x}_j)$, $j = 1,\ldots,N$, $\ell = 1,\ldots,M$, $\boldsymbol{c} = [c_1,\ldots,c_N]^T$, $\boldsymbol{d} = [d_1,\ldots,d_M]^T$, $\boldsymbol{y} = [y_1,\ldots,y_N]^T$, O an $M \times M$ zero matrix, and $\mathbf{0}$ a zero vector of length M with $M = \dim \Pi^s_{m-1}$. Here, as discussed earlier, φ should be strictly conditionally positive definite of order m and radial on $\mathbb{R}^s$ and the set $\mathcal{X} = \{\boldsymbol{x}_1,\ldots,\boldsymbol{x}_N\}$ should be $(m-1)$-unisolvent.

The preconditioning scheme proposed by Dyn and her co-workers is a generalization of the simple differencing scheme discussed above. It is motivated by the fact that the polyharmonic splines (*i.e.*, thin plate splines and radial powers)

$$\varphi(r) = \begin{cases} r^{2k-s}\log r, & s \text{ even}, \\ r^{2k-s}, & s \text{ odd}, \end{cases}$$

$2k > s$, are fundamental solutions of the k-th iterated Laplacian in $\mathbb{R}^s$, *i.e.*

$$\Delta^k \varphi(\|\boldsymbol{x}\|) = c\delta_{\mathbf{0}}(\boldsymbol{x}),$$

where $\delta_{\mathbf{0}}$ is the Dirac delta function centered at the origin, and c is an appropriate constant.

For the (inverse) multiquadrics $\varphi(r) = (1+r^2)^{\pm 1/2}$, which are also discussed in the papers mentioned above, application of the Laplacian yields a similar limiting behavior, *i.e.*

$$\lim_{r\to\infty} \Delta^k \varphi(r) = 0,$$

and for $r \to 0$

$$\Delta^k \varphi(r) \gg 1.$$

One now wants to discretize the Laplacian on the (irregular) mesh given by the (scattered) data sites in $\mathcal{X}$. To this end the authors of [Dyn *et al.* (1986)] suggest the following procedure for the case of scattered data interpolation over $\mathbb{R}^2$.

(1) Start with a triangulation of the set $\mathcal{X}$, *e.g.*, the will do. This triangulation can be visualized as follows.

   (a) Begin with the points in $\mathcal{X}$ and construct their or *Voronoi diagram*. The Dirichlet tile of a particular point $\boldsymbol{x}$ is that subset of points in $\mathbb{R}^2$ which are closer to $\boldsymbol{x}$ than to any other point in $\mathcal{X}$. The dashed lines in Figure 34.1 denote the Dirichlet tesselation for the set of 25 Halton points (circles) in $[0,1]^2$.

   (b) Construct the Delaunay triangulation, which is the dual of the Dirichlet tesselation, *i.e.*, connect all strong neighbors in the Dirichlet tesselation, *i.e.*, points whose tiles share a common edge. The solid lines in Figure 34.1 denote the corresponding Delaunay triangulation of the 25 Halton points.

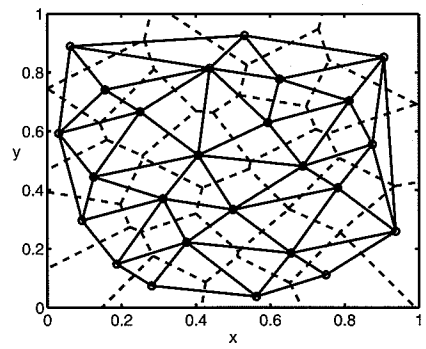

Fig. 34.1 Dirichlet tesselation (dashed lines) and corresponding Delaunay triangulation (solid lines) of 25 Halton points (circles).

(2) Discretize the Laplacian on this triangulation. In order to also take into account the boundary points Dyn, Levin and Rippa instead use a discretization of an iterated Green's formula which has the space Π^2_{m-1} as its null space. The necessary partial derivatives are then approximated on the triangulation using certain sets of vertices of the triangulation. (three points for first order partials, six for second order).

Figure 34.1 was created in MATLAB using the commands

```
load('Data2D_25h')
tes = delaunayn(dsites);
triplot(tes,dsites(:,1),dsites(:,2),'k-')
hold on
[vx, vy] = voronoi(dsites(:,1),dsites(:,2),tes);
plot(dsites(:,1),dsites(:,2),'ko',vx,vy,'k--')
axis([0 1 0 1])
```

As in our other MATLAB examples, the file `Data2D_25h` contains the coordinates of the 25 Halton points in the array `dsites`.

The discretization described above yields the matrix $B = (b_{ji})_{j,i=1}^N$ as the preconditioning matrix in a way analogous to the previous section. We now obtain

$$(BA)_{jk} = \sum_{i=1}^{N} b_{ji}\varphi(\|\boldsymbol{x}_i - \boldsymbol{x}_k\|) \approx \Delta^m \varphi(\|\cdot - \boldsymbol{x}_k\|)(\boldsymbol{x}_j), \quad j,k = 1,\ldots,N. \tag{34.2}$$

This matrix has the property that the entries close to the diagonal are large compared to those away from the diagonal, which decay to zero as the distance between the two points involved goes to infinity. Since the construction of B (in step 2 above) ensures that part of the preconditioned block matrix vanishes, namely $BP = O$,

one must now solve the non-square system

$$\begin{bmatrix} BA \\ P^T \end{bmatrix} \boldsymbol{c} = \begin{bmatrix} B\boldsymbol{y} \\ \boldsymbol{0} \end{bmatrix}.$$

Actually, the top part of the system, the square system $BA\boldsymbol{c} = B\boldsymbol{y}$, is singular, but it is shown in the paper [Dyn *et al.* (1986)] that the additional constraints $P^T\boldsymbol{c} = 0$ guarantee existence of a unique solution. Furthermore, the coefficients $\boldsymbol{d}$ in the original expansion of the interpolant $\mathcal{P}_f$ can be obtained by solving

$$P\boldsymbol{d} = \boldsymbol{y} - A\boldsymbol{c},$$

i.e., by fitting the polynomial part of the expansion to the residual $\boldsymbol{y} - A\boldsymbol{c}$.

The approach just described leads to localized basis functions Ψ that are linear combinations of the original basis functions φ. More precisely,

$$\Psi_j(\boldsymbol{x}) = \sum_{i=1}^{N} b_{ji}\varphi(\|\boldsymbol{x} - \boldsymbol{x}_i\|) \approx \Delta^m\varphi(\|\cdot - \boldsymbol{x}_j\|)(\boldsymbol{x}), \tag{34.3}$$

where the coefficients b_{ji} are determined via the discretization described above.

The localized basis functions Ψ_j, $j = 1, \ldots, N$, (see (34.3)) can be viewed as an alternative (better conditioned) basis for the approximation space spanned by the functions $\Phi_j = \varphi(\|\cdot - \boldsymbol{x}_j\|)$. We will come back to this idea in Section 34.4.

In [Dyn *et al.* (1986)] the authors describe how the preconditioned matrices can be used efficiently in conjunction with various iterative schemes such as Chebyshev iteration or a version of the conjugate gradient method. The authors also mention smoothing of noisy data, or low-pass filtering as other applications for this preconditioning scheme.

The effectiveness of the above preconditioning strategy was illustrated with some numerical examples in [Dyn *et al.* (1986)]. We list some of their results in Table 34.1. Thin plate splines and multiquadrics were tested on two different data sets (grid I and grid II) in $\mathbb{R}^2$. The shape parameter ε for the multiquadrics was chosen to be the reciprocal of the average mesh size. A linear term was added for thin plate splines, and a constant for multiquadrics.

Table 34.1 Condition numbers without and with preconditioning.

φ	N	grid I orig.	grid I precond.	grid II orig.	grid II precond.
TPS	49	1181	4.3	1885	3.4
	121	6764	5.1	12633	3.9
MQ	49	7274	69.2	17059	222.8
	121	10556	126.0	107333	576.0

One can see that the most dramatic improvement is achieved for thin plate splines. This is to be expected since the method described above is tailored to these functions. As noted earlier, for multiquadrics an application of the Laplacian does not yield the delta function, but for values of r close to zero gives just relatively large values.

Another early preconditioning strategy was suggested in [Powell (1994a)]. Powell uses Householder transformations to convert the matrix of the interpolation system (34.1) to a symmetric positive definite matrix, and then uses the conjugate gradient method. However, Powell reports that this method is not particularly effective for large thin plate spline interpolation problems in $\mathbb{R}^2$.

In [Baxter (1992a); Baxter (2002)] preconditioned conjugate gradient methods for solving the interpolation problem are discussed in the case when Gaussians or multiquadrics are used on a regular grid. The resulting matrices are Toeplitz matrices, and a large body of literature exists for dealing with matrices having this special structure (see, *e.g.*, [Chan and Strang (1989)]).

34.3 Preconditioned GMRES via Approximate Cardinal Functions

More recently, Beatson, Cherrie and Mouat [Beatson *et al.* (1999)] proposed a preconditioner for the iterative solution of radial basis function interpolation systems in conjunction with the GMRES method of [Saad and Schultz (1986)]. The GMRES method is a general purpose iterative solver that can be applied to nonsymmetric (nondefinite) systems. For fast convergence the matrix should be preconditioned such that its eigenvalues are clustered around one and away from the origin. Obviously, if the basis functions for the radial basis function space were cardinal functions, then the matrix would be the identity matrix with all its eigenvalues equal to one. Therefore, the GMRES method would converge in a single iteration. Consequently, the preconditioning strategy employed by the authors of [Beatson *et al.* (1999)] for the GMRES method is to obtain a preconditioning matrix B that is close to the inverse of A.

Since it is too expensive to find the true cardinal basis (this would involve at least as much work as solving the interpolation problem), the idea pursued in [Beatson *et al.* (1999)] (and suggested earlier in [Beatson *et al.* (1996); Beatson and Powell (1993)]) is to find *approximate* cardinal functions similar to the functions Ψ_j in the previous subsection. Now, however, there is also an emphasis on efficiency, *i.e.*, we are interested in *local* approximate cardinal functions, if possible (*c.f.* also the use of approximate cardinal functions in the Faul-Powell algorithm of Section 33.2). Several different strategies for the construction of these approximate cardinal functions were suggested in [Beatson *et al.* (1999)]. We will now explain the basic idea.

Given the centers $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N$ for the basis functions in the RBF interpolant

$$\mathcal{P}_f(\boldsymbol{x}) = \sum_{j=1}^{N} c_j\varphi(\|\boldsymbol{x} - \boldsymbol{x}_j),$$

the j-th approximate cardinal function is given as a linear combination of the basis

functions $\Phi_i = \varphi(\|\cdot - \boldsymbol{x}_i\|)$, where i runs over (some subset of) $\{1,\ldots,N\}$, *i.e.*,

$$\Psi_j = \sum_{i=1}^{N} b_{ji}\varphi(\|\cdot - \boldsymbol{x}_i\|) + p_j. \tag{34.4}$$

Here p_j is a polynomial in Π^s_{m-1} that is used only in the conditionally positive definite case, and the coefficients b_{ji} satisfy the usual conditions

$$\sum_{i=1}^{N} b_{ji}p_j(\boldsymbol{x}_i) = 0 \qquad \text{for all } p_j \in \Pi^s_{m-1}. \tag{34.5}$$

The key feature in designing the approximate cardinal functions is to have only a few $n \ll N$ coefficients in (34.4) to be nonzero. In that case the functions Ψ_j are found by solving small $n \times n$ linear systems, which is much more efficient than dealing with the original $N \times N$ system. For example, in [Beatson *et al.* (1999)] the authors use $n \approx 50$ for problems involving up to 10000 centers. The resulting preconditioned system is of the same form as the earlier preconditioner (34.2), *i.e.*, we now have to solve the preconditioned problem

$$(BA)\boldsymbol{c} = B\boldsymbol{y},$$

where the entries of the matrix BA are just $\Psi_j(\boldsymbol{x}_k)$, $j,k = 1,\ldots,N$.

The simplest strategy for determining the coefficients b_{ji} is to select the n nearest neighbors of $\boldsymbol{x}_j$, and to find b_{ji} by solving the (local) cardinal interpolation problem

$$\Psi_j(\boldsymbol{x}_i) = \delta_{ij}, \qquad i = 1,\ldots,n,$$

subject to the moment constraint (34.5) listed above. Here δ_{ij} is the Kronecker-delta, so that Ψ_j is one at $\boldsymbol{x}_j$ and zero at all of the neighboring centers $\boldsymbol{x}_i$.

This basic strategy is improved by adding so-called *special points* that are distributed (very sparsely) throughout the domain (for example near corners of the domain, or at other significant locations).

A few numerical results for thin plate spline and multiquadric interpolation in $\mathbb{R}^2$ from [Beatson *et al.* (1999)] are listed in Table 34.2. The condition numbers are ℓ_2-condition numbers, and the points were randomly distributed in the unit square. The "local precond." column uses the $n = 50$ nearest neighbors to determine the approximate cardinal functions, whereas the right-most column uses the 41 nearest neighbors plus nine special points placed uniformly in the unit square. The effect of the preconditioning on the performance of the GMRES algorithm is, *e.g.*, a reduction from 103 to 8 iterations for the 289 point data set for thin plate splines, or from 145 iterations to 11 for multiquadrics.

An extension of the ideas of [Beatson *et al.* (1999)] to linear systems arising in the collocation solution of partial differential equations (see Chapter 38) was explored in Mouat's Ph.D. thesis [Mouat (2001)] and also in the recent paper [Ling and Kansa (2005)].

Table 34.2 Condition numbers without and with preconditioning.

φ	N	unprecond.	local precond.	local precond. w/special
TPS	289	4.005e+006	1.464e+003	5.721e+000
	1089	2.753e+008	6.359e+005	1.818e+002
	4225	2.605e+009	2.381e+006	1.040e+006
MQ	289	1.506e+008	3.185e+003	2.639e+002
	1089	2.154e+009	8.125e+005	5.234e+004
	4225	3.734e+010	1.390e+007	4.071e+004

34.4 Change of Basis

As pointed out at the beginning of this chapter, another approach to obtaining a better conditioned interpolation system is to work with a different basis for the approximation space. While this idea is implicitly addressed in the preconditioning strategies discussed above, we will now make it our primary goal to find a better conditioned basis for the RBF approximation space. Univariate piecewise linear splines and natural cubic splines can be interpreted as radial basis functions, and we know that B-splines form stable bases for those spaces. Therefore, it should be possible to generalize this idea for other RBFs.

The process of finding a "better" basis for conditionally positive definite radial basis functions is closely connected to finding the reproducing kernel of the associated native space. Since we did not elaborate on the construction of native spaces for conditionally positive definite functions earlier, we will now present the relevant formulas without going into any further details. In particular, for polyharmonic splines we will be able to find a basis that is in a certain sense *homogeneous*, and therefore the condition number of the related interpolation matrix will depend only on the number N of data points, but *not* on their separation distance (*c.f.* the discussion in Chapter 16). This approach was suggested by Beatson, Light and Billings [Beatson *et al.* (2000)], and has its roots in [Sibson and Stone (1991)].

Let Φ be a strictly conditionally positive definite kernel of order m, and $\mathcal{X} = \{\boldsymbol{x}_1,\ldots,\boldsymbol{x}_N\} \subset \Omega \subset \mathbb{R}^s$ be an $(m-1)$-unisolvent set of centers. Then the reproducing kernel for the native space $\mathcal{N}_\Phi(\Omega)$ is given by

$$\begin{aligned} K(\boldsymbol{x},\boldsymbol{y}) = \Phi(\boldsymbol{x},\boldsymbol{y}) &- \sum_{k=1}^{M} p_k(\boldsymbol{x})\Phi(\boldsymbol{x}_k,\boldsymbol{y}) - \sum_{\ell=1}^{M} p_\ell(\boldsymbol{y})\Phi(\boldsymbol{x},\boldsymbol{x}_\ell) \\ &+ \sum_{k=1}^{M}\sum_{\ell=1}^{M} p_k(\boldsymbol{x})p_\ell(\boldsymbol{y})\Phi(\boldsymbol{x}_k,\boldsymbol{x}_\ell) + \sum_{\ell=1}^{M} p_\ell(\boldsymbol{x})p_\ell(\boldsymbol{y}), \end{aligned} \tag{34.6}$$

where the points $\{\boldsymbol{x}_1,\ldots,\boldsymbol{x}_M\}$ comprise an $(m-1)$-unisolvent subset of $\mathcal{X}$ and the polynomials p_k, $k = 1,\ldots,M$, form a *cardinal* basis for Π^s_{m-1} on this subset whose dimension is $M = \binom{s+m-1}{m-1}$, *i.e.*,

$$p_\ell(\boldsymbol{x}_k) = \delta_{k,\ell}, \qquad k,\ell = 1,\ldots,M.$$

This formulation of the reproducing kernel for the conditionally positive definite case also appears in the statistics literature in the context of *kriging* (see, *e.g.*, [Berlinet and Thomas-Agnan (2004)]). In that context the kernel K is a covariance kernel associated with the generalized covariance Φ. These two kernels give rise to the kriging equations and dual kriging equations, respectively.

An immediate consequence of having found the reproducing kernel K is that we can express the radial basis function interpolant to values of some function f given on $\mathcal{X}$ in the form

$$\mathcal{P}_f(\boldsymbol{x}) = \sum_{j=1}^{N} c_j K(\boldsymbol{x}, \boldsymbol{x}_j), \qquad \boldsymbol{x} \in \mathbb{R}^s.$$

Note that the kernel K used here is a strictly positive definite kernel (since it is a reproducing kernel) with built-in polynomial precision. The coefficients c_j are determined by satisfying the interpolation conditions

$$\mathcal{P}_f(\boldsymbol{x}_i) = f(\boldsymbol{x}_i), \qquad i = 1, \ldots, N.$$

We will see below (in Tables 34.3 and 34.4) that this basis already performs "better" (*i.e.*, is better conditioned) than the standard basis $\{\Phi(\cdot, \boldsymbol{x}_1), \ldots, \Phi(\cdot, \boldsymbol{x}_N)\}$ if we keep the number of centers fixed, and vary only their separation distance.

To obtain the homogeneous basis referred to above we modify K by subtracting the tensor product polynomial, *i.e.*,

$$\kappa(\boldsymbol{x}, \boldsymbol{y}) = K(\boldsymbol{x}, \boldsymbol{y}) - \sum_{\ell=1}^{M} p_\ell(\boldsymbol{x}) p_\ell(\boldsymbol{y}).$$

Now, if $\boldsymbol{y}$ denotes any one of the points $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_M$ in the $(m-1)$-unisolvent subset of $\mathcal{X}$ used in the construction of K above, then we have

$$\begin{aligned}\kappa(\cdot, \boldsymbol{y}) &= \Phi(\cdot, \boldsymbol{y}) - \sum_{k=1}^{M} p_k(\cdot)\Phi(\boldsymbol{x}_k, \boldsymbol{y}) - \sum_{\ell=1}^{M} p_\ell(\boldsymbol{y})\Phi(\cdot, \boldsymbol{x}_\ell) \\ &\quad + \sum_{k=1}^{M}\sum_{\ell=1}^{M} p_k(\cdot) p_\ell(\boldsymbol{y}) \Phi(\boldsymbol{x}_k, \boldsymbol{x}_\ell) \\ &= \Phi(\cdot, \boldsymbol{y}) - \sum_{k=1}^{M} p_k(\cdot)\Phi(\boldsymbol{x}_k, \boldsymbol{y}) - \Phi(\cdot, \boldsymbol{y}) + \sum_{k=1}^{M} p_k(\cdot)\Phi(\boldsymbol{x}_k, \boldsymbol{y}) = 0\end{aligned}$$

since the polynomials p_k are cardinal on $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_M$, *i.e.*, only one of the $p_k(y)$ will "survive".

This means that the functions $\kappa(\cdot, \boldsymbol{x}_j)$, $j = 1, \ldots, N$, cannot be used as a basis of our approximation space. Instead we need to remove the points used to define the cardinal polynomials above from the set of centers used for κ. Once we do this it turns out that the matrix C with entries $C_{i,j} = \kappa(\boldsymbol{x}_i, \boldsymbol{x}_j)$, $i, j = M+1, \ldots, N$, is positive definite, and therefore we obtain the following basis

$$\{p_1, \ldots, p_M\} \cup \{\kappa(\cdot, \boldsymbol{x}_{M+1}), \ldots, \kappa(\cdot, \boldsymbol{x}_N)\}$$

for the space $\operatorname{span}\{\Phi(\cdot, \boldsymbol{x}_1), \ldots, \Phi(\cdot, \boldsymbol{x}_N)\}$. Therefore the interpolant can be represented in the form

$$\mathcal{P}_f(\boldsymbol{x}) = \sum_{j=1}^{M} d_j p_j(\boldsymbol{x}) + \sum_{k=M+1}^{N} c_k \kappa(\boldsymbol{x}, \boldsymbol{x}_k), \qquad \boldsymbol{x} \in \mathbb{R}^s. \tag{34.7}$$

The coefficients are determined as usual by enforcing the interpolation conditions $\mathcal{P}_f(\boldsymbol{x}_i) = f(\boldsymbol{x}_i)$, $i = 1, \ldots, N$. Since the polynomials p_j are cardinal on $\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_M\}$ and κ was shown to be zero if centered at these points, this leads to the following linear system

$$\begin{bmatrix} I & O \\ P^T & C \end{bmatrix} \begin{bmatrix} \boldsymbol{d} \\ \boldsymbol{c} \end{bmatrix} = \begin{bmatrix} \boldsymbol{y}_p \\ \boldsymbol{y}_\kappa \end{bmatrix}, \tag{34.8}$$

with I an $M \times M$ identity matrix, O an $M \times (N-M)$ zero matrix, C as above, $P_{ij} = p_j(\boldsymbol{x}_i)$, $j = 1, \ldots, M$, $i = M+1, \ldots, N$, $\boldsymbol{c} = [c_{M+1}, \ldots, c_N]^T$, $\boldsymbol{d} = [d_1, \ldots, d_M]^T$, and the right-hand side vectors $\boldsymbol{y}_p = [f(\boldsymbol{x}_1), \ldots, f(\boldsymbol{x}_M)]^T$ and $\boldsymbol{y}_\kappa = [f(\boldsymbol{x}_{M+1}), \ldots, f(\boldsymbol{x}_N)]^T$. The identity block (cardinality of the polynomial basis functions) implies that the coefficient vector $\boldsymbol{d}$ is given by

$$d_j = f(\boldsymbol{x}_j), \qquad j = 1, \ldots, M,$$

and therefore the system (34.8) can be solved as

$$C\boldsymbol{c} = \boldsymbol{y}_\kappa - P^T \boldsymbol{d}. \tag{34.9}$$

As mentioned above, one can show that the matrix C is symmetric and positive definite.

Most importantly, for polyharmonic splines, the ℓ_2-condition number of the matrix C is invariant under a uniform scaling of the centers, *i.e.*, if $C^h = (\kappa(h\boldsymbol{x}_i, h\boldsymbol{x}_j))$, then

$$\operatorname{cond}(C^h) = \operatorname{cond}(C).$$

This is proved to varying degrees of detail in the papers [Beatson *et al.* (2000); Iske (2003a)] and the book [Wendland (2005a)].

Example 34.3. The simplest example is given by the polyharmonic spline $\varphi(r) = r$. In this case $M = 1$ so that the only polynomial term is given by the constant $p \equiv 1$. For simplicity we use the origin as a special point. Using these conventions we have the following three representations of the various kernels:

$$\begin{aligned}\Phi(\boldsymbol{x}, \boldsymbol{y}) &= \|\boldsymbol{x} - \boldsymbol{y}\|, \\ K(\boldsymbol{x}, \boldsymbol{y}) &= \|\boldsymbol{x} - \boldsymbol{y}\| - \|\boldsymbol{y}\| - \|\boldsymbol{x}\| + 1, \\ \kappa(\boldsymbol{x}, \boldsymbol{y}) &= \|\boldsymbol{x} - \boldsymbol{y}\| - \|\boldsymbol{y}\| - \|\boldsymbol{x}\|.\end{aligned}$$

Note that in this case the condition number of the matrix C associated with the kernel κ is clearly invariant under uniform scaling of the problem. However, the matrix A associated with the basic norm RBF Φ enjoys the same invariance. It is only when we add the polynomial blocks P and P^T to ensure reproduction of constants that the condition number of the resulting block matrix will vary greatly with the problem scaling. A similar dependence of the condition number of the system matrix on the scaling is associated with the kernel K.

34.5 Effect of the "Better" Basis on the Condition Number of the Interpolation Matrix

We reproduce some numerical experiments from [Beatson *et al.* (2000)] based on the use of thin plate splines in $\mathbb{R}^2$. We compute the ℓ_2-condition numbers of the interpolation matrix for the three different approaches mentioned above, *i.e.*, using the standard basis consisting of functions $\Phi(\cdot, \boldsymbol{x}_j)$ and monomials (obtained with the MATLAB program `RBFInterpolation2Dlinear.m` of Chapter 6), using the reproducing kernels $K(\cdot, \boldsymbol{x}_j)$, and using the matrix C based on the kernel κ. The matrix for the kernels $K(\cdot, \boldsymbol{x}_j)$ is computed with the program `tpsK.m` provided in Program 34.1. The three polynomial cardinal functions are based on the three corners $(0,0)$, $(0,1)$, and $(1,0)$ of the unit square, *i.e.*,

$$p_1(\boldsymbol{z}) = 1 - z_1 - z_2,$$
$$p_2(\boldsymbol{z}) = z_1,$$
$$p_3(\boldsymbol{z}) = z_2,$$

where $\boldsymbol{z} = (z_1, z_2) \in \mathbb{R}^2$.

The program `tpsK.m` is completely vectorized, *i.e.*, we input arrays of points `x` and `y`, and create the entire matrix with entries $K(\boldsymbol{x}_i, \boldsymbol{x}_j)$, $i, j = 1, \ldots, N$ (denoted by `rbf` in the program). We assemble the matrix according to the terms in (34.6). On lines 3 and 4 we fill two matrices, `px` and `py`, whose columns contain the values of the polynomials p_1, p_2 and p_3 (defined as separate functions at the end of the program) at all of the points in `x` and `y`, respectively. The first term of (34.6), the matrix $\Phi(\boldsymbol{x}, \boldsymbol{y})$, is assembled on line 5 where we call the subroutine `tps.m` listed as Program C.4 in Appendix C. Next, on lines 6–11 we add the next two sums from (34.6) simultaneously. The double sum is added to the matrix `rbf` on lines 12–17, and finally the tenor product polynomial term is computed and added on lines 18–20.

Program 34.1. `tpsK.m`

```
% rbf = tpsK(x,y)
% Computes matrix for thin plate spline kernel K with
% linear polynomials cardinal on (0,0), (1,0), (0,1)
% Calls on: tps
 1  function rbf = tpsK(x,y)
    % Define points for cardinal polynomials
 2  ppoints = [0 0; 1 0; 0 1];
 3  px = [p1(x) p2(x) p3(x)];
 4  py = [p1(y) p2(y) p3(y)];
 5  r = DistanceMatrix(x,y); rbf = tps(1,r);
 6  for k=1:3
 7     r = DistanceMatrix(ppoints(k,:),y);
```



```
 8     rbf = rbf - px(:,k)*tps(1,r);
 9     r = DistanceMatrix(x,ppoints(k,:));
10     rbf = rbf - tps(1,r)*py(:,k)';
11  end
12  for j=1:3
13     for k=1:3
14        r = DistanceMatrix(ppoints(j,:),ppoints(k,:));
15        rbf = rbf + px(:,j)*py(:,k)'*tps(1,r);
16     end
17  end
18  for k=1:3
19     rbf = rbf + px(:,k)*py(:,k)';
20  end
21  return
    % The cardinal polynomials
22  function w = p1(z)
23  w = 1 - z(:,1) - z(:,2);
24  return
25  function w = p2(z)
26  w = z(:,1);
27  return
28  function w = p3(z)
29  w = z(:,2);
30  return
```

Since Program 34.1 produces the entire interpolation (or evaluation) matrix, we can use Program 2.1 and replace lines 13 and 14 by

```
IM = tpsK(dsites,ctrs);
```

and lines 15 and 16 by

```
EM = tpsK(epoints,ctrs);
```

in order to solve the interpolation problem in this case.

The matrix C is obtained in a similar fashion by using a program `tpsH.m` that is identical to `tpsK.m` except that lines 18–20 are removed. In addition, we need to remove the corner points $(0,0)$, $(1,0)$, and $(0,1)$ from the `ctrs` and `dsites` in the driver program.

In the first experiment (illustrated in Table 34.3) the problem is formulated on the unit square $[0,1]^2$. Here both the number of points and the separation distance vary from one row in the table to the next. The three different columns list the ℓ_2-condition numbers of the interpolation matrix for the three different approaches mentioned above. With this setup all three methods perform comparably.

Table 34.3 Condition numbers for different thin plate spline bases on $[0,1]^2$ with increasing number of points and varying separation distance.

spacing h	standard matrix	reproducing kernel	homogeneous matrix
1/8	3.515800e+003	1.839030e+004	7.583833e+003
1/16	3.893850e+004	2.651373e+005	1.108581e+005
1/32	5.136252e+005	4.000679e+006	1.686431e+006
1/64	7.618277e+006	6.202918e+007	2.626402e+007

In the second experiment (shown in Table 34.4) the number of points is kept fixed at 5×5 equally spaced points. However, the domain is scaled to the square $[0,a]^2$ with scale parameter a (this can easily be done using the same programs as above by introducing a scale parameter at the appropriate places, see also Program 34.2). The effect of this is that only the separation distance $q_{\mathcal{X}}$ changes from one row to the next in the table. Now, clearly, the two new methods show less dependence on the separation distance, with the condition number of the homogeneous matrix C being completely insensitive to the re-scaling as claimed earlier.

Table 34.4 Condition numbers for different thin plate spline bases on $[0,a]^2$ with fixed number of 25 points and varying separation distance.

scale parameter a	standard matrix	reproducing kernel	homogeneous matrix
0.001	2.434883e+008	8.463509e+008	5.493771e+002
0.01	2.436378e+006	8.464002e+006	5.493771e+002
0.1	2.517866e+004	8.513354e+004	5.493771e+002
1.0	3.645782e+002	1.366035e+003	5.493771e+002
10	1.874215e+006	1.260864e+003	5.493771e+002
100	1.151990e+011	1.139634e+005	5.493771e+002
1000	3.548239e+015	1.138572e+007	5.493771e+002

We close this section by pointing out that Iske an co-workers take advantage of the scale invariance of polyharmonic splines (and thin plate splines in particular) in the construction of a numerical multiscale solver for transport problems (see, *e.g.*, [Behrens *et al.* (2002)]).

34.6 Effect of the "Better" Basis on the Accuracy of the Interpolant

In this section we provide an example illustrating the surprising fact that for polyharmonic splines not only the homogeneous kernel κ can be used successfully for poorly scaled problems, but also the standard kernel Φ.

Example 34.4. We use the thin plate spline basic function $\varphi(r) = r^2 \log r$ and a scaled version of Franke's testfunction to generate test data on a 5×5 uniform grid in the square $[0,a]^2$ as in Table 34.4. However, now the scale parameter a will range from 10^{-9} to 10^9. We will present condition numbers and root-mean-square errors computed on a 40×40 uniform grid for the three different kernels discussed previously. We list only the MATLAB code for the homogeneous case since the two other programs are very similar to previous ones. The function `tpsH.m` called by Program 34.2 is almost the same as Program 34.1 listed above. The required modifications are noted there.

Many parts of Program 34.2 are familiar. However, in order to deal with the kernel κ and the associated matrix C we need to define the special points at which the cardinal polynomials are defined. This is done on line 9, where the special points are taken as three corners of the scaled unit square. Since the kernel is the zero function when centered at these points they need to be removed from the set of centers. This is accomplished on lines 11, 12 and 14. The scaling of the problem happens on lines 8, 9 and 13. The scale also enters in a number of other places such as the definition of the evaluation grid on line 15, computation of the right-hand side on lines 17–19, and the computation of the exact solution on line 26. In contrast to most of our other interpolation programs here we compute the interpolation and evaluation matrices with a single subroutine (*c.f.* the calls to `tpsH` on lines 20 and 22). Note that the scale parameter a is passed to `tpsH`. Equations 34.9 and 34.7 for the solution and evaluation of the interpolant are implemented together on line 25. Finally, the three cardinal polynomials are coded on lines 35–43. Since these polynomials are defined on the unit square they need to be called with re-scaled arguments (cf. lines 17 and 23).

Program 34.2. `RBFInterpolation2DtpsH.m`

```
% RBFInterpolation2DtpsH
% Script that performs 2D TPS interpolation with homogeneous kernel
% Calls on: tpsH
 1  function RBFInterpolation2DtpsH
    % Define Franke's function as testfunction
 2  f1 = @(x,y) 0.75*exp(-((9*x-2).^2+(9*y-2).^2)/4);
 3  f2 = @(x,y) 0.75*exp(-((9*x+1).^2/49+(9*y+1).^2/10));
 4  f3 = @(x,y) 0.5*exp(-((9*x-7).^2+(9*y-3).^2)/4);
 5  f4 = @(x,y) 0.2*exp(-((9*x-4).^2+(9*y-7).^2));
 6  testfunction = @(x,y) f1(x,y)+f2(x,y)+f3(x,y)-f4(x,y);
 7  N = 25; gridtype = 'u';
 8  a = 1e9;
 9  ppoints = a*[0 0; 1 0; 0 1];
    % Load data points
10  name = sprintf('Data2D_%d%s',N,gridtype); load(name)
    % Remove (0,0), (1,0), (0,1) to work with C matrix
11a remove = [find(dsites(:,1)==0 & dsites(:,2)==0);...
```

```
11b      find(dsites(:,1)==1 & dsites(:,2)==0);...
11c      find(dsites(:,1)==0 & dsites(:,2)==1)]
12  dsites(remove,:) = [];
    % Scale problem to square [0,a]^2
13  dsites = a*dsites;
    % Let centers coincide with data sites
14  ctrs=dsites;
15  neval = 40; grid = linspace(0,a,neval);
16  [xe,ye] = meshgrid(grid); epoints = [xe(:) ye(:)];
    % Create right-hand side for homogeneous problem
17  DP = [p1(dsites/a) p2(dsites/a) p3(dsites/a)]';
18  d = testfunction(ppoints(:,1)/a,ppoints(:,2)/a);
19  rhs = testfunction(dsites(:,1)/a,dsites(:,2)/a) - DP'*d;
    % Compute interpolation matrix for the special case of TPS
    % native space kernel (no need to add polynomials)
20  IM = tpsH(dsites,ctrs,a);
    % Compute condition number of interpolation matrix
21  fprintf('l2-condition          : %e\n', cond(IM))
    % Compute evaluation matrix
22  EM = tpsH(epoints,ctrs,a);
23  EP = [p1(epoints/a) p2(epoints/a) p3(epoints/a)];
24  EM = [EM EP];
    % Compute RBF interpolant
25  Pf = EM * [(IM\rhs); d];
    % Compute exact solution
26  exact = testfunction(epoints(:,1)/a,epoints(:,2)/a);
    % Compute errors on evaluation grid
27  maxerr = norm(Pf-exact,inf);
28  rms_err = norm(Pf-exact)/neval;
29  fprintf('RMS error:     %e\n', rms_err)
30  fprintf('Maximum error: %e\n', maxerr)
31  fview = [160,20];  % viewing angles for plot
32  PlotSurf(xe,ye,Pf,neval,exact,maxerr,fview);
33  PlotError2D(xe,ye,Pf,exact,maxerr,neval,fview);
34  return
    % The cardinal polynomials
35  function w = p1(z)
36  w = 1 - z(:,1) - z(:,2);
37  return
38  function w = p2(z)
39  w = z(:,1);
40  return
```



```
41  function w = p3(z)
42  w = z(:,2);
43  return
```

In Table 34.5 we list the root-mean-square errors resulting from the three different interpolation methods. The scaling of the domain was chosen more extreme than in Table 34.4 so that the sensitivity of the reproducing kernel K becomes clearly visible. Its condition number of the interpolation matrix for $a = 10^{-9}$ was 5.354134e+018, while for $a = 10^9$ it was 6.994062e+019. While both of these condition numbers are clearly very high and therefore indicate that we might expect numerical difficulties solving a problem on these length scales, the other two methods (standard TPS basis functions and the homogeneous kernel κ) perform perfectly throughout the entire range of scalings. Moreover, the condition numbers for the standard TPS interpolation matrix are much higher than for the K kernel: 7.299408e+021 for $a = 10^{-9}$, and even 2.749537e+038 for $a = 10^9$. Nevertheless, the standard TPS interpolant does *not* suffer from instability due to this ill-conditioning. The same is true for all other tests we have performed with standard polyharmonic spline interpolants (such as, *e.g.*, the norm basic function).

Table 34.5 RMS errors for different thin plate spline interpolants on $[0, a]^2$ with fixed number of 25 points and varying separation distance.

scale parameter a	standard matrix	reproducing kernel	homogeneous matrix
10^{-9}	2.969478e-002	NAN	2.969478e-002
10^{-6}	2.969478e-002	2.970740e-002	2.969478e-002
10^{-3}	2.969478e-002	2.969478e-002	2.969478e-002
1.0	2.969478e-002	2.969478e-002	2.969478e-002
10^3	2.969478e-002	2.969478e-002	2.969478e-002
10^6	2.969478e-002	2.969218e-002	2.969478e-002
10^9	2.969478e-002	1.207446e+003	2.969478e-002