Wendland Functions
A C++ Code to Compute Them

Carlos Argáez[1], Sigurdur Hafstein[1] and Peter Giesl[2]
[1]*Faculty of Physical Sciences, University of Iceland, 107 Reykjavík, Iceland*
[2]*Department of Mathematics, University of Sussex, Falmer BN1 9QH, U.K.*

Keywords: Radial Basis Functions, Wendland Functions, Compact Support.

Abstract: In this paper we present a code in C++ to compute Wendland functions for arbitrary smoothness parameters. Wendland functions are compactly supported Radial Basis Functions that are used for interpolation of data or solving Partial Differential Equations with mesh-free collocation. For the computations of Lyapunov functions using Wendland functions their derivatives are also needed so we include this in the code. Wendland functions with a few fixed smoothness parameters are included in some C++ libraries, but for the general case the only code freely available was implemented in MAPLE taking advantage of the computer algebra system. The aim of this contribution is to allow scientists to use Wendland functions in their C++ code without having to implement them themselves. The computed Wendland functions are polynomials and their coefficients are computed and stored in a vector, which allows for efficient computation of their values using the Horner scheme.

1 INTRODUCTION

Mesh-free methods, particularly based upon Radial Basis Functions, provide a powerful tool for solving interpolation and generalized interpolation problems efficiently and in any dimension (Buhmann, 2003; Fasshauer, 2007; Schaback and Wendland, 2006). An interpolation problem is, given N pairwise distinct sites $\mathbf{x}_1, \ldots, \mathbf{x}_N \in \mathbb{R}^n$ (collocation points) and corresponding values $f_1, \ldots, f_N \in \mathbb{R}$ to find a function $f\colon \mathbb{R}^n \to \mathbb{R}$ satisfying $f(\mathbf{x}_j) = f_j$ for all $j = 1, \ldots, N$.

For a generalized interpolation problem, instead of prescribing the values of the function at given sites, one is given the values of linear functionals applied to the function. These include linear Partial Differential Equations with boundary values. Error estimates between the generalized interpolant and the true solution are available and depend on the fill distance, measuring how dense the collocation points are. These methods are particularly suited for high-dimensional problems and the use of scattered collocation points, as they do not require the collocation points to lie on a structured grid. They have been successfully used in various applications such as geography, engineering, neural networks, machine learning and image processing. Further applications include solving Partial Differential Equations (Fornberg and Flyer, 2015), the construction of Lyapunov functions in dynamical systems (Giesl, 2007), and high-dimensional integration (Dick et al., 2013).

A (generalized) interpolation problem looks for a function in a function space, often a Reproducing Kernel Hilbert Space. The corresponding kernel can be defined by a Radial Basis Function. Compactly supported Radial Basis Functions are given by the family of Wendland functions, which are polynomials on their support. They have two parameters and are defined recursively.

In this paper we present a numerical code in C++ to explicitly compute the Wendland function with any given parameters. Another code, written in MAPLE, has been given previously in (Zhu, 2012; Schaback, 2011). This code takes advantage of MAPLE subroutines like integration, factor and simplify, etc. Furthermore, libMesh (Kirk et al., 2006) and FOAM-FSI (Blom et al.,) provide C++ libraries that can be used to evaluate and provide Wendland functions. However, these just provide few limited cases of Wendland functions. In our case, we build a C++ code that includes all necessary operations to compute any Wendland function.

In Section 2 we give an overview over generalized interpolation and Wendland functions. In Section 3 we describe the algorithm used in the code, and in

Argáez, C., Hafstein, S. and Giesl, P.
Wendland Functions - A C++ Code to Compute Them.
DOI: 10.5220/0006441303230330
In *Proceedings of the 7th International Conference on Simulation and Modeling Methodologies, Technologies and Applications (SIMULTECH 2017)*, pages 323-330
ISBN: 978-989-758-265-3

Section 4 we give some examples of computed Wendland functions with our code. The full C++ code is presented in the appendix.

2 GENERALIZED INTERPOLATION AND WENDLAND FUNCTIONS

To discuss generalized interpolation in more detail, we let $H \subset C(\mathbb{R}^n, \mathbb{R})$ be a Hilbert space of functions $f : \mathbb{R}^n \to \mathbb{R}$, which are at least continuous. Its dual H^* consists of all functionals $\lambda : H \to \mathbb{R}$, which are linear and continuous. Examples of such functionals include Dirac's delta distribution $\delta_{\mathbf{x}_0}(f) = f(\mathbf{x}_0)$, evaluating the function at the point $\mathbf{x}_0 \in \mathbb{R}^n$ and differential operators evaluated at a point, e.g., $\lambda = \delta_{\mathbf{x}_0} \circ \frac{\partial}{\partial x_j}$.

The generalized interpolation problem is: given N linearly independent functionals $\lambda_1, \ldots, \lambda_N \in H^*$ and corresponding values $f_1, \ldots, f_N \in \mathbb{R}$, a generalized interpolant is a sufficiently smooth function $f : \mathbb{R}^n \to \mathbb{R}$ satisfying $\lambda_j(f) = f_j$ for all $j = 1, \ldots, N$. Note that the usual interpolation problem is a special case with $\lambda_j = \delta_{\mathbf{x}_j}$.

The norm-minimal interpolant is the interpolant that, in addition, minimizes the norm of the Hilbert space, namely $\min\{\|f\|_H : \lambda_j(f) = f_j, 1 \le j \le N\}$.

It is well-known (Wendland, 2005) that the norm-minimal interpolant can be represented as a linear combination of the Riesz representers $v_j \in H$ of the functionals. If the Hilbert space is a Reproducing Kernel Hilbert Space, then these Riesz representers can be expressed in a simpler way.

A Reproducing Kernel Hilbert Space is a Hilbert space H with a reproducing kernel $\Phi : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$ such that

1. $\Phi(\cdot, \mathbf{x}) \in H$ for all $\mathbf{x} \in \mathbb{R}^n$
2. $g(\mathbf{x}) = \langle g, \Phi(\cdot, \mathbf{x}) \rangle_H$ for all $g \in H$ and $\mathbf{x} \in \mathbb{R}^n$

where $\langle \cdot, \cdot, \rangle$ denotes the inner product of the Hilbert space H.

Then the Riesz representers are given $v_j = \lambda_j^{\mathbf{y}} \Phi(\cdot, \mathbf{y})$, i.e. the functional applied to one of the arguments of the reproducing kernel. Hence, the norm-minimal interpolant can be written as $f(\mathbf{x}) = \sum_{j=1}^{N} \beta_j \lambda_j^{\mathbf{y}} \Phi(\mathbf{x}, \mathbf{y})$, where the coefficients β_j are determined by the interpolation conditions $\lambda_j(f) = f_j$, $1 \le j \le N$.

Often kernels depend on the difference between the two arguments, i.e. $\Phi(\mathbf{x}, \mathbf{y}) := \phi(\mathbf{x} - \mathbf{y})$. Radial Basis Functions are radial functions ϕ, namely $\phi(\mathbf{x} - \mathbf{y}) := \Psi(\|\mathbf{x} - \mathbf{y}\|)$. There are a variety of Radial Basis Functions, including the Gaussians, multiquadrics, inverse multiquadrics and thin plate splines. Each of them defines a different Reproducing Kernel Hilbert Space.

Compactly supported Radial Basis Functions, which are polynomials on their support, were considered in (Wu, 1995; Schaback and Wu, 1996), and are now called Wendland functions (Wendland, 1995; Wendland, 1998); for a discussion of compactly supported Radial Basis Functions see (Zhu, 2012). The corresponding Reproducing Kernel Hilbert Space is norm-equivalent to a Sobolev space.

In this paper, we describe a new algorithm to define compute Wendland Function of any order.

The Wendland functions $\psi_{l,k}$, where $l \in \mathbb{N}$ and $k \in \mathbb{N}_0$ are defined recursively by (Wendland, 1995)

$$\begin{aligned} \psi_{l,0}(r) &= (1-r)_+^l \\ \psi_{l,k+1}(r) &= \int_r^1 t\psi_{l,k}(t)\,\mathrm{d}t \text{ for } k = 0, 1, \ldots \end{aligned}$$

where $x_+ = \begin{cases} x \text{ for } x > 0 \\ 0 \text{ for } x \le 0 \end{cases}$.

So, if we want to evaluate a Wendland function for given k and l at the point cr, where $c > 0$ is a fixed constant, we obtain:

$$\psi_{l,k}(cr) = \underbrace{\int_{cr}^1 t_k \int_{t_k}^1 t_{k-1} \ldots \int_{t_2}^1 t_1 \psi_{l,0}(t_1) \overbrace{\mathrm{d}t_1 \ldots \mathrm{d}t_k}^{k \text{ times}}}_{k \text{ times}} \tag{1}$$

3 ALGORITHM

3.1 Function `wendlandfunction`

The Wendland function is built up from recursive integrations of a polynomial. As seen in (1), the parameters that define the polynomial's degree are k and l, where l denotes the degree of the initial polynomial and k is the number of recursive integrations. We will write the algorithm as a C++ program and will use the libraries Blitz++ and Boost.

In our calculations, we will focus on the calculation of the polynomial on the support; note that the support of $\psi_{l,k}(r)$ is $[0, 1]$. Furthermore, it is seen in (1), that in each step we multiply the previous polynomial by a factor t. This product will increase the degree of the polynomial by one and the integration will also increase the polynomial's degree by one. Hence, the final degree of the Wendland function will be $l + 1 + 2 \cdot k$.

Below we will present the steps to construct the Wendland function.

1. First, we compute the coefficients of the binomial expansion of $(1-r)^l$ using Pascal's Triangle. The coefficients are stored in an array (a vector) whose indexing corresponds to the degree of the respective polynomial term. In more detail, at the n-th position we store the coefficient of the term r^n for $n = 0,\dots,l$. The dimension of the storage in this part of the code will be $l+1$.
2. Once the binomial expansion is computed, the first multiplication $t(1-t)^l$ must be performed. This multiplication will only increase each term's degree by one, without affecting the coefficients, hence, we move the coefficients from the previous iteration into the next position in the array. So, the coefficient of t^n will become the coefficient of t^{n+1} for $n = 0,\dots,l$.
3. The integration will also increase the length of the array by one. The difference is that in this case the coefficients will be integrated using the simple formula (without the integration constant):
$$\int at^n \mathrm{d}t = \frac{at^{n+1}}{n+1}$$
Therefore, the coefficient of t^n will be divided by $n+1$ and stored in the position $n+1$ of the new array.
4. The limits of the integration are r and 1. Evaluating a polynomial at 1 is achieved by adding up all coefficients. The lower limit, r will preserve the terms of the polynomial, but we need to change the sign. Altogether, the new polynomial will be
$$\underbrace{N(r)}_{\text{new polynomial}} = \underbrace{P(1)}_{\text{polynomial evaluated at 1}} - \underbrace{P(r)}_{\text{original polynomial}}$$
5. After that, repeat steps 2. to 4. until the last integration is completed.
6. Finally, we want to evaluate the function $\psi_{l,k}(cr)$ with fixed positive constant $c > 0$. Hence, at the last integration, we consider the evaluation of cr at the lower limit only.

In practice, when the Wendland functions are computed, they tend to be presented with integer coefficients, i.e. the integration coefficients must be scaled by their Least Common Multiplayer (LCM). The coefficients in C++ are given as `double` and to be able to obtain the LCM one needs integers. Furthermore, the library to compute the LCM is a Boost library that requires to clearly define the type of the variable. So, the integrate coefficient are stored as a different array defined as `long long unsigned`. Once the LCM is obtained, the Greatest Common Divisor (GCD) is also computed to make sure that the factor to make the coefficients integers is the least.

The code that computes the Wendland functions is divided in three parts and can be found in the appendix.

1. The first step computes the coefficients of Pascal's Triangle for the degree l. If $k = 0$, then the Wendland function will be given by $(1-cr)^l$ for any given value of c. If $k \neq 0$ then the c parameter will be not considered in this iteration.
2. The next step is to keep track of iterations up to $k-1$, in this section of the code the product and the integration will be carried out. The vector `momentaneo`, i.e.,"momentanous" is created to store the numerators momentanously.
   The term that corresponds to the evaluation of the upper limit 1, equals the sum of all the terms. Since in a hand-made calculation, it would be necessary to keep track of the denominators as well, following that logic, the denominator is equal to the LCM, i.e. the `divisors(0)=mcm` of all the elements of divisors.
3. The last iteration is the most difficult one. It is given when the value of the last iteration k occurs and $k \neq 0$. It follows the same logic: the first vector is the product that equals the coefficients of the last integration but is stored in the previous entry plus one. Now the integration is considered with the c parameter, namely `integral(i+1)= pow(c,i+1)*product(i)/(i+1);`.
   In the next step, the divisors are computed, then the LCM and finally `divisors(0)=mcm;` in this case, the upper limit 1 will be as before, but now we are required to divide by the corresponding c^n: `integral(0)-=integral(i)/pow(c,i);`
   Yet again the corresponding numerator will be `numerators(N+2*k-l-2-j) =round(integral(0)*divisors(0));` where the function round ensures that the number is integer. In order to compute the GCD it is necessary to figure out that all the numerators are correct. However, if one is not interested in making the coefficient integers, the C++ function here presented is able to give the corresponding Wendland functions for double coefficients. The option is a boolean variable, if equals true then the Wendland function is scaled. Notice that when $k = 0$ the c parameter is computed automatically according to their value.
   Furthermore, when one is interested in using a non-integer c, one can just activate this options with a second boolean variable that considers c as integers only when it is true. This is because when scaling the polynomial coefficients, one should be carefully to scale them at the proper time.

3.2 AUXILIARY FUNCTIONS ψ_1, ψ_2

Wendland functions find important applications in algorithms to compute Lyapunov functions (Giesl, 2007). In those, the computation of the derivatives of Wendland functions is mandatory. For a given Wendland function $\psi_{l,k}(r) =: \psi(r)$ we define the auxiliary functions ψ_1 and ψ_2 as follows for $r > 0$:

$$\begin{aligned} \psi_1(r) &= \frac{\frac{d}{dr}\psi(r)}{r} \\ \psi_2(r) &= \frac{\frac{d}{dr}\psi_1(r)}{r} \end{aligned} \tag{2}$$

Therefore to be able to implement (2), one must notice that for a given Wendland function the function ψ_1 will be a polynomial of two degrees lower than the original Wendland function. For $\psi_2(r)$ one should follow the same logic. For $r > 0$, $\psi_2(r)$ will be two orders lower than $\psi_1(r)$. When evaluating it, before calling the functions `wendlandpsi1` and `wendlandpsi2` one could test r to find whether it is $r > 0$ or $r = 0$. The full C++ codes are presented in the appendix.

3.3 GENERAL DISCUSSION

This code produces high order Wendland functions that are stored in a very efficient way by terms of an array. Such storage allows us to use Wendland functions as many times as required without re-computing. Furthermore, this allows to evaluate Wendland function in very efficient ways, e.g., by means of Horner's method to evaluate polynomials. In general, the computation of a high order Wendland function with this code takes a tenth of a millisecond. There are some advantages and disadvantages that need to be discussed. On the one hand, the limit of performance of our code will depend on the capability of any particular computer to storage `long long unsigned` variables. Therefore, although this codes is designed to provide any Wendland function, it has the disadvantage of reaching only the limit of the local computer where it is run. On the other hand, it is preferable to have a code that computes Wendland functions rather than a library that would, in general, be very limited in capabilities. To the best of our knowledge current libraries only provide a limited selection of Wendland functions.

4 EXAMPLES

All the results here presented, have been tested and compared with Mathematica.

For $c = 1$

$\psi_{3,9}(r) = 2431 - 30030r^2 + 171171r^4 - 596904r^6$
$+1424430r^8 - 2469012r^{10} + 3233230r^{12}$
$-3325608r^{14} + 2909907r^{16} - 3233230r^{18}$
$+2752512r^{19} - 969969r^{20} + 131072r^{21}$

$\psi_{4,9}(r) = 221 - 3003r^2 + 19019r^4 - 74613r^6$
$+203490r^8 - 411502r^{10} + 646646r^{12}$
$-831402r^{14} + 969969r^{16} - 1616615r^{18}$
$+1835008r^{19} - 969969r^{20} + 262144r^{21}$
$-29393r^{22}$

$\psi_{5,9}(r) = 221 - 3289r^2 + 23023r^4$
$-100947r^6 + 312018r^8 - 728042r^{10}$
$+1352078r^{12} - 2124694r^{14} + 3187041r^{16}$
$-7436429r^{18} + 10551296r^{19}$
$-7436429r^{20} + 3014656r^{21}$
$-676039r^{22} + 65536r^{23}$

$\psi_{6,9}(r) = 1105 - 17940r^2 + 138138r^4$
$-672980r^6 + 2340135r^8 - 6240360r^{10}$
$+13520780r^{12} - 25496328r^{14}$
$+47805615r^{16} - 148728580r^{18}$
$+253231104r^{19} - 223092870r^{20}$
$+120586240r^{21} - 40562340r^{22}$
$+7864320r^{23} - 676039r^{24}$

$\psi_{7,6}(r) = 11 - 171r^2 + 1292r^4$
$-6460r^6 + 25194r^8 - 92378r^{10}$
$+554268r^{12} - 1323008r^{13}$
$+1662804r^{14} - 1323008r^{15}$
$+692835r^{16} - 233472r^{17}$
$+46189r^{18} - 4096r^{19}$

$\psi_{7,9}(r) = 221 - 3900r^2 + 32890r^4$
$-177100r^6 + 688275r^8 - 2080120r^{10}$
$+5200300r^{12} - 11589240r^{14}$
$+26558675r^{16} - 106234700r^{18}$
$+211025920r^{19} - 223092870r^{20}$
$+150732800r^{21} - 67603900r^{22}$
$+19660800r^{23} - 3380195r^{24}$
$+262144r^{25}$

$\psi_{8,9}(r) = 17 - 325r^2 + 2990r^4$
$-17710r^6 + 76475r^8 - 260015r^{10}$
$+742900r^{12} - 1931540r^{14}$
$+5311735r^{16} - 26558675r^{18}$
$+60293120r^{19} - 74364290r^{20}$
$+60293120r^{21} - 33801950r^{22}$
$+13107200r^{23} - 3380195r^{24}$
$+524288r^{25} - 37145r^{26}$

For $c = 2$

$\psi_{5,4}(r) = 7 - 312r^2 + 6864r^4 - 109824r^6$
$+2306304r^8 - 9371648r^9 + 18450432r^{10}$
$-20447232r^{11} + 12300288r^{12} - 3145728r^{13}$

For $c = 3$

$$\psi_{5,4}(r) = 7 - 702r^2 + 34749r^4 - 1250964r^6 + 59108049r^8 - 360277632r^9 + 1063944882r^{10} - 1768635648r^{11} + 1595917323r^{12} - 612220032r^{13}$$

For $c = 4$

$$\psi_{5,4}(r) = 7 - 1248r^2 + 109824r^4 - 7028736r^6 + 590413824r^8 - 4798283776r^9 + 18893242368r^{10} - 41875931136r^{11} + 50381979648r^{12} - 25769803776r^{13}$$

Next, we show how the graphs of these equations look for $c = 1$.

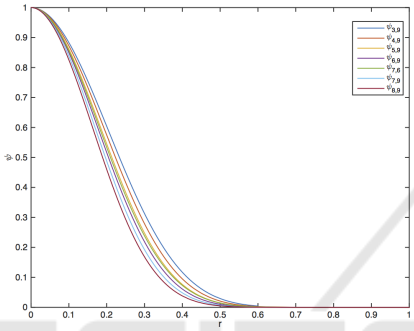

Figure 1: Different Wendland functions obtained with this code. All of them have been normalised and they are presented over all their support $[0, 1]$.

5 APPLICATIONS

Wendland functions have important applications to compute Lyapunov functions (Giesl, 2007). In the following, we give examples of computing Lyapunov functions for a particular system with low and high order Wendland functions, $\psi_{5,3}$ and $\psi_{9,7}$ respectively.

We consider the dynamical system given by

$$\begin{pmatrix} \dot{x} \\ \dot{y} \end{pmatrix} = \begin{pmatrix} -x(x^2+y^2-1/4)(x^2+y^2-1) - y \\ -y(x^2+y^2-1/4)(x^2+y^2-1) + x. \end{pmatrix}, \quad (3)$$

System (3) has a repelling periodic orbit with radius $1/2$, an asymptotically stable periodic orbit with radius 1 and the origin is a stable equilibrium.

Obtaining a vector plot of system (3) with Mathematica (Wolfram Research, Inc.,), it is possible to notice how the system behaves, Figure 2.

Now, the computed Lyapunov functions are shown in Figure 3, while the contour lines for system (3) are presented in Figure 4.

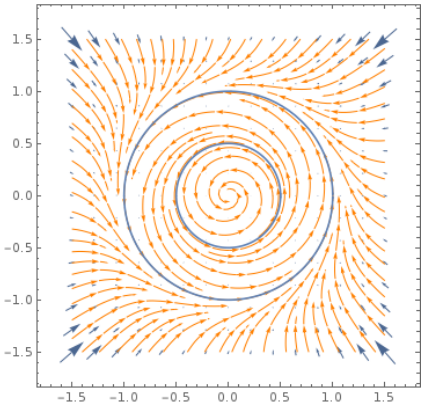

Figure 2: Vector plot for system (3). The blue line show both the asymptotically stable periodic orbit with radius 1 and the repelling periodic orbit with radius $1/2$.

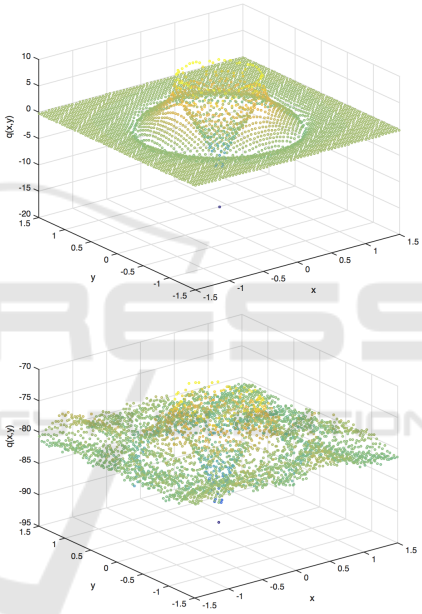

Figure 3: Approximated Lyapunov functions for system (3). The upper figure shows the Lyapunov function using $\psi_{5,3}(r)$ and the lower one shows it for $\psi_{9,7}(r)$.

6 CONCLUSIONS

In this paper we have presented an efficient implementation to compute the Wendland function [1] $\psi_{l,k}(cr)$ for any given parameters $l \in \mathbb{N}$, $k \in \mathbb{N}_0$ and $c > 0$. Wendland functions are compactly supported Radial Basis Functions that are used to solve (generalized) interpolation problems in high dimensions. These include solving PDE boundary value problems which are used in many important applications.

[1]The code can be downloaded directly from: https://notendur.hi.is/~carlos/codes/WendlandFunction-CarlosArgaez2017.zip.

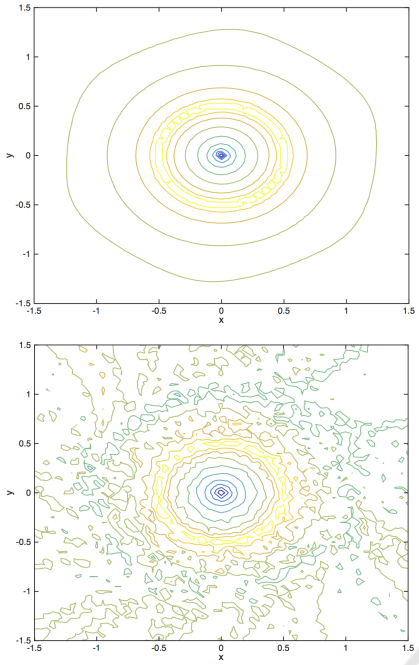

Figure 4: Contour lines for the approximated Lyapunov functions for system (3). The upper figure shows the contour lines for the Lyapunov function using $\psi_{5,3}(r)$ and the lower one shows them for $\psi_{9,7}(r)$.

ACKNOWLEDGEMENTS

First author in this paper is supported by the Icelandic Research Fund (Rannís) grant number 163074-052, Complete Lyapunov functions: Efficient numerical computation. Special thanks to Dr. Jean-Claude Berthet for all his good comments and advices on C++.

REFERENCES

Blom, D., Cardiff, P., Gillebaart, T., t. Hofstede, E., and Kazemi-Kamyab, V. Foam-fsi project.

Buhmann, M. D. (2003). *Radial basis functions: theory and implementations*, volume 12 of *Cambridge Monographs on Applied and Computational Mathematics*. Cambridge University Press, Cambridge.

Dick, J., Kuo, F. Y., and Sloan, I. H. (2013). High-dimensional integration: the quasi-Monte Carlo way. *Acta Numer.*, 22:133–288.

Fasshauer, G. E. (2007). *Meshfree approximation methods with MATLAB*, volume 6 of *Interdisciplinary Mathematical Sciences*. World Scientific Publishing Co. Pte. Ltd., Hackensack, NJ. With 1 CD-ROM (Windows, Macintosh and UNIX).

Fornberg, B. and Flyer, N. (2015). Solving PDEs with radial basis functions. *Acta Numer.*, 24:215–258.

Giesl, P. (2007). *Construction of Global Lyapunov Functions Using Radial Basis Functions*. Lecture Notes in Math. 1904, Springer.

Kirk, B. S., Peterson, J. W., Stogner, R. H., and Carey, G. F. (2006). `libMesh`: A c++ library for parallel adaptive mesh refinement/coarsening simulations. *Engineering with Computers*, 22(3–4):237..254.

Schaback, R. (2011). The missing wendland functions. 34:67–81.

Schaback, R. and Wendland, H. (2006). Kernel techniques: from machine learning to meshless methods. *Acta Numer.*, 15:543–639.

Schaback, R. and Wu, Z. (1996). Operators on radial functions. *J. Comput. Appl. Math.*, 73(1-2):257–270.

Wendland, H. (1995). Piecewise polynomial, positive definite and compactly supported radial functions of minimal degree. *Adv. Comput. Math.*, 4(4):389–396.

Wendland, H. (1998). Error estimates for interpolation by compactly supported Radial Basis Functions of minimal degree. *J. Approx. Theory*, 93:258–272.

Wendland, H. (2005). *Scattered data approximation*, volume 17 of *Cambridge Monographs on Applied and Computational Mathematics*. Cambridge University Press, Cambridge.

Wolfram Research, Inc. Mathematica, wolframalpha.

Wu, Z. M. (1995). Compactly supported positive definite radial functions. *Adv. Comput. Math.*, 4(3):283–292.

Zhu, S.-X. (2012). Compactly supported radial basis functions: how and why? OCCAM Preprint Number 12/57, University of Oxford, Eprints Archive.

APPENDIX: THE CODE

The complete Wendland Function code is given below:

```
void wendlandfunction(double l,double k,
double c,bool b, bool d, Array<double, 1>&wdlf)
{
long long unsigned comparing;
long long unsigned mcm = 0;
long long unsigned mcd = 0;
long long unsigned N=l+1;
Array <double, 1> coeff((int)(N+2*k+1)),
 cpower((int)(N+2*k+1)),
 pascalcoeff((int)(N+2*k+1));
Array <double, 1> integral((int)(N+2*k+1)),
integralbackup((int)(N+2*k+1)),
momentaneo((int)(N+2*k+1));
Array < long long unsigned, 1>
divisors((int)(N+2*k+1)),
numerators(N+2*k);
Array <double, 1> product((int)(N+2*k));
wdlf.resize((int)(N+2*k+1));
numerators=1;
for(int j=0;j<=k;j++)
 {
 product.reference(Range(0,(int)(N+2*j-2)));
 integral.reference(Range(0,(int)(N+2*j-1)));
```

```
integralbackup.reference
(Range(0,(int)(N+2*j-1)));
divisors.reference(Range(0,(int)(N+2*j-1)));
divisors=1.0;
product=0.0;
integral=0.0;
cpower=0.0;
if(j==0)
 {
  if(k>0.0)
  {
   wdlf.reference(Range(0,l));
   for(int i=0;i<=l;i++)
    {
     integral.reference(Range(0,l));
     integralbackup.reference(Range(0,l));
     cpower(i)=i;
     double x=1;
     for(int h=0;h<=i;h++)
      {
       coeff(h)=pow(-1,h)*x;
        x = x * (i - h) / (h + 1);
       }
      }
     integral=coeff;
     for(int i=0; i<l+1; i++)
      {
       numerators(numerators.size()-l+i-1)
       =coeff(i);
      }
     }else{
     for(int i=0;i<=l;i++)
      {
       integral.reference(Range(0,l));
       integralbackup.reference(Range(0,l));
       cpower(i)=i;
       double x=1;
       for(int h=0;h<=i;h++)
        {
coeff(h)=pow(c,cpower(h))*pow(-1,h)*x;
x = x * (i - h) / (h + 1);
        }
      }
 integral=coeff;
               }
 momentaneo=1.0;
 }if((j>0) && (j<k)){
 for(int i=0; i<product.size()-1; i++)
  {
   product(i+1)=coeff(i);
  }
 for(int i=1; i<integral.size()-1; i++)
  {
  integral(i+1)=product(i)/(i+1);
  }
 for(int i=(int)integral.size()-1; i>=2; i--)
  {
  divisors(i)=(i)*momentaneo(i-2);
  }
mcm=divisors(0);
 for(int i=0; i<=divisors.size()-1; i++)
  {
    mcm
    =boost::math::lcm(mcm,divisors(i));
    }
 divisors(0)=mcm;
 for(int i=1; i<integral.size(); i++)
  {
   integral(0)-=integral(i);
   }
 integral=-1.0*integral;
 numerators(N+2*k-l-2-j)
 =round(integral(0)*divisors(0));
 coeff.reference
 (Range(0,(int)integral.size()-1));
 wdlf.reference
 (Range(0,(int)integral.size()-1));
 momentaneo.reference
 (Range(0,(int)integral.size()-1));
 momentaneo=divisors;
 coeff=integral;
  }if((j==k) && (k!=0)){
 for(int i=0; i<product.size()-1; i++)
  {
   product(i+1)=coeff(i);
  }
 for(int i=1; i<integral.size()-1; i++)
  {
   integral(i+1)=pow(c,i+1)*product(i)/(i+1);
  }
 for(int i=(int)integral.size()-1; i>=2; i--)
  {
   divisors(i)=(i)*momentaneo(i-2);
  }
 mcm=divisors(0);
 for(int i=0; i<=(int)divisors.size()-1; i++)
  {
   mcm=
  boost::math::lcm(mcm,divisors(i));
  }
divisors(0)=mcm;
 for(int i=1; i<integral.size(); i++)
  {
   integral(0)-=integral(i)/pow(c,i);
  }
  integral=-1.0*integral;
 numerators(N+2*k-l-2-j)
  =round(integral(0)*divisors(0));
 if(numerators(N+2*k-l-2-j)==0)
  {
   numerators(N+2*k-l-2-j)=1;
  }
 for(int i=0; i<integral.size()-1;i++)
  {
    if(integral(i)==0.0)
      {
       numerators(i)=integral(i);
      }
  }
mcm=divisors(0);
 for(int i=1; i<=divisors.size()-1; i++)
  {
   mcm=
  boost::math::lcm(mcm,divisors(i));
```

```
  }
 divisors(0)=mcm;
 numerators(0)=round(divisors(0)*integral(0));
 for(int i=0; i<=divisors.size()-1; i++)
  {
  mcd=
  boost::math::gcd(numerators(i),divisors(i));
  divisors(i)=divisors(i)/mcd;
  }
 mcm=divisors(0);
 for(int i=1; i<=divisors.size()-1; i++)
  {
   mcm=
  boost::math::lcm(mcm,divisors(i));
   }
  coeff.reference
  (Range(0,(int)integral.size()-1));
  wdlf.reference
  (Range(0,(int)integral.size()-1));
  momentaneo.reference
  (Range(0,(int)integral.size()-1));
  momentaneo=divisors;
  coeff=integral;
      }
 }
if(k==0)
 {
  if(b)
 {
 wdlf=integral;
  }else{
   wdlf=integral;
        }
 }else{
    integralbackup=integral;
    if(b)
    {
     cout << "mcm " << mcm << endl;
     for(int i=0; i<(int)integral.size();i++)
      {
  cout.precision(18);
  wdlf(i)=round(mcm*integral(i));
      }
    Array<double, 1> wdlfm((int)integral.size());
    wdlfm=wdlf;
    long long unsigned mcd=wdlfm(0);
    for(int i=1; i<(int)integral.size();i++)
     {
      if(wdlf(i)!=0.0)
       {
        comparing=abs(wdlf(i));
        mcd=
        boost::math::gcd(mcd,comparing);
       }
     }
   cout << "mcd " << mcd << endl;
  if(d)
   {
    for(int i=0; i<(int)integral.size();i++)
    {
    wdlf(i)=(wdlfm(i)/mcd);
    }
    }else{
    for(int i=0; i<(int)integral.size();i++)
     {
      wdlf(i)=mcm*(integralbackup(i)/mcd);
     }
         }
    }else{  wdlf=integral;  }
}
```

The code for the auxiliary function ψ_1:

```
void wendlandpsi1(Array<double, 1>
  &wdlfinput, Array<double, 1> &wdlf1)
{
  int dim=(int)wdlfinput.size();
  wdlf1.resize(dim-2);
  for(int i=dim-1; i>0; i--)
  {    wdlf1(i-2)=i*wdlfinput(i);   }
}
```

The code for the auxiliary function ψ_2:

```
void wendlandpsi2(Array<double, 1>
  &wdlf1, Array<double, 1> &wdlf2)
{
  int dim=(int)wdlf1.size();
  wdlf2.resize(dim-2);
  for(int i=dim-1; i>0; i--)
  {    wdlf2(i-2)=i*wdlf1(i);  }
}
```