Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Stable Radial Basis Function Interpolation for Multi-Physics Simulation Applications

David Sommer

Course of Study: Informatik

Examiner: Prof. Dr. rer. nat. habil. Miriam Mehl

Supervisor: Dipl.-Ing. Florian Lindner

Commenced: 2018-05-09

Completed: 2018-11-09

Abstract

In multi-physics simulation applications there is a need for some kind of middleware between distinct simulations. The preCICE project ([4]) aims to provide such a software. One key component of this is to interpolate data from one mesh to another. There are several possibilities for doing this, however in this bachelor thesis I will primarily consider radial basis functions. These have several advantages like for instance being oblivious of topological information of the meshes, thus working on arbitrary point clouds, but also some drawbacks such as bad numerical stability. There is some parameter that can be tuned to trade interpolation accuracy against stability, but recent results showed that there is a way to get good numerical stability and good accuracy at the same time. This thesis will focus on this method for improving numerical stability, the **RBF-QR** method. An implementation is given in Python and C++ with the latter being able to be used in highly parallel applications.

Contents

1 Multi-Physics Applications **5**
- 1.1 Nearest Neighbor Mapping . . . 5
- 1.2 Nearest Projection Mapping . . . 5

2 Basic RBF Interpolation **6**
- 2.1 Radialsymmetric functions . . . 6
- 2.2 Interpolation . . . 6
- 2.3 Condition . . . 8
  - 2.3.1 Improving condition by adding a polynomial . . . 11
  - 2.3.2 Improving condition by rescaling the interpolant . . . 13
  - 2.3.3 Improving condition by using a variable shape parameter 13
  - 2.3.4 Improving condition by using a preconditioner . . . 13

3 RBF-QR Interpolation **14**
- 3.1 RBF and polynomial interpolation . . . 14
- 3.2 Higher dimension . . . 15
- 3.3 RBF-QR on the unit sphere . . . 15
  - 3.3.1 A better basis . . . 15
  - 3.3.2 Spherical Harmonics . . . 15
  - 3.3.3 SPH Expansions . . . 15
  - 3.3.4 SPH expansion expressed in matrix form . . . 16
- 3.4 Nonperiodic cartesian space . . . 17
- 3.5 A better Expansion for Gaussian RBFs . . . 18
- 3.6 Polar coordinates . . . 20
- 3.7 Chebyshev polynomials . . . 20
  - 3.7.1 1-D Final Expansion . . . 21
  - 3.7.2 2-D Final Expansion . . . 22
  - 3.7.3 3-D Final Expansion . . . 23
- 3.8 The final algorithm . . . 24
- 3.9 Stability of the RBF-QR algorithm . . . 25

4 Implementation **25**
- 4.1 The Python implementation . . . 26
- 4.2 The C++ Implementation . . . 28
  - 4.2.1 Memory and communication model . . . 28
  - 4.2.2 MPI . . . 29
  - 4.2.3 Eigen . . . 29
  - 4.2.4 PETSc . . . 30
  - 4.2.5 Elemental . . . 30
  - 4.2.6 Architecture . . . 30
  - 4.2.7 Parallel Implementation . . . 30
- 4.3 Index transformation . . . 33
- 4.4 Computing $\tilde{R}$ . . . 34
  - 4.4.1 Computing $\tilde{R}$ for big K . . . 35
- 4.5 Determining j_{max} . . . 35
- 4.6 Parallel QR factorization . . . 36

5 Results **40**
5.1 Choosing the right ϵ 40
5.2 Chebyshev nodes 40
5.3 Condition and Accuracy 43
5.3.1 Condition in the small-ϵ domain 43
5.3.2 Condition and accuracy in one dimension 44
5.3.3 Condition and accuracy in two dimensions 47
5.4 Performance 49
5.4.1 Scaling to multiple nodes 49
5.4.2 Varying shape parameter 51
5.4.3 Variable mesh size 51

Appendices **53**

A 3d index transformation **53**

1 Multi-Physics Applications

With the advent of more and more powerful compute architectures there are increasingly bigger simulations possible. For simulating these big systems there are different kinds of PDEs to solve for different parts of the system. For instance a solid structure will have a PDE that looks nothing like the PDE of a fluid streaming along it. This kind of simulation is called *Multi-Physics Application*. In multi-physics applications there are two ways of using solvers to simulate a system. The first way is to simply include all the required equations within one system and to solve it with a single solver. The other way is to specify each simulation separately. This second approach is more complicated, as this means having multiple solvers with each one having its own mesh and needing to exchange data like heat, pressure, displacement, etc. However the advantage of having specialized solvers for each part of the system might outweigh the disadvantages. preCICE provides a library that a wide range of solvers can directly use to exchange data at the boundary surfaces. Exchanging the data requires data mapping, for which multiple methods can be used. A desirable feature of a data mapping algorithm is to be able to operate on arbitrary point clouds, without needing any (topological) information about the underlying mesh. preCICE does not need this information, but it can be provided optionally to enable some more sophisticated methods like Nearest Projection Mapping. One rather simple method to achieve data mapping on arbitrary point clouds is to use Nearest Neighbor Mapping.

1.1 Nearest Neighbor Mapping

For some input points $x_i \in \Xi$, some function $f : \mathbb{R}^d \to \mathbb{R}$ and some output points $y_i \in \mathbb{R}^d$, define

$$S_{\text{NN}}(y_i) = f(\operatorname*{arg\,min}_{x_i \in \Xi} ||y_i - x_i||)$$

as the interpolant which simply uses the nearest neighbor of each output mesh point to interpolate.
Nearest Neighbor Mapping is conceptually simple but does not always yield good results. Especially on non-matching mesh geometries the algorithm might give very inexact results. A more sophisticated approach is given by the *Nearest Projection Mapping* algorithm.

1.2 Nearest Projection Mapping

Nearest Projection Mapping works in two steps. First, some input point is projected onto a geometric primitive in the output mesh (Edge, Triangle) by using an orthogonal projection. This yields a point that is coplanar with the geometric primitive. For this point, barycentric coordinates are calculated and used as weights for interpolation onto the output mesh. However, this requires some topological information about the output mesh, which might or might not be given. Fortunately there is an algorithm that increases accuracy (in contrast to Nearest Neighbor Mapping) without needing any topological information, called *RBF Interpolation*

2 Basic RBF Interpolation

2.1 Radialsymmetric functions

Let $x \in \mathbb{R}^d$ and let $||\cdot||$ denote a norm on $\mathbb{R}^d$. Functions that fulfill

$$\Phi(x) = \Phi(||x||)$$

are called **radial basis functions** (RBF). Unless noted otherwise the euclidean norm $||x||_2 = \sqrt{x_1^2 + \ldots x_d^2}$ is used.
For interpolation the shifted variant

$$\Phi(||x - x_0||)$$

with some $x_0 \in \mathbb{R}^d$, often called *center*, will be used.

2.2 Interpolation

For interpolation of a function $f : \mathbb{R}^d \to \mathbb{R}$, known on some centers $x_i \in \Xi$, a radial basis function $\Phi(x)$ is used. There are multiple basis functions to choose from, some more widely used choices are listed in Figure 1. Of these, the Gaussian basis function is especially interesting because it will play an important role for the RBF-QR algorithm in section 3. It can be seen in Figure 2 for two dimensions.

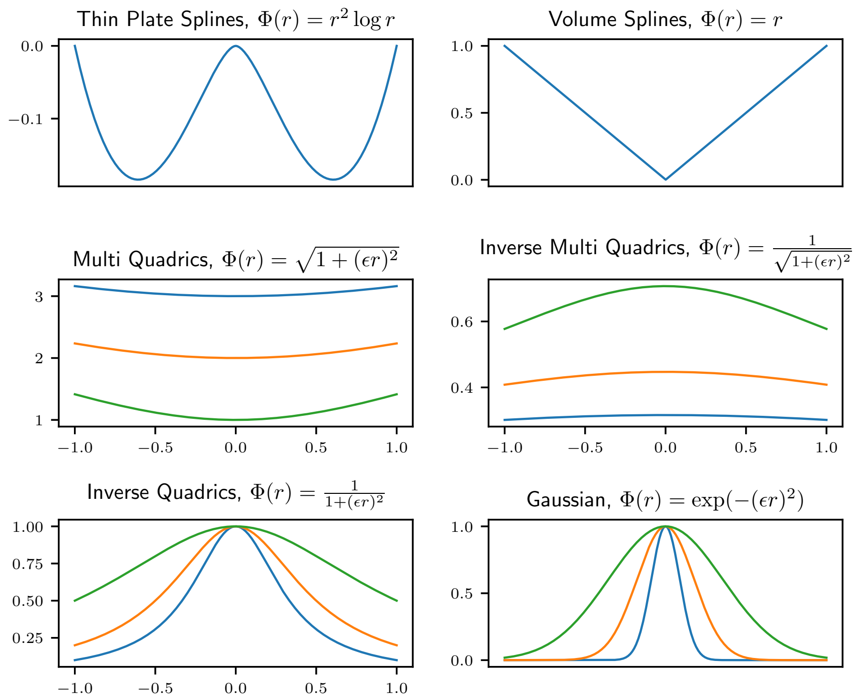

Figure 1: **Different basis functions**
Some basis functions are parameterized with a parameter ϵ that controls the shape of the function.

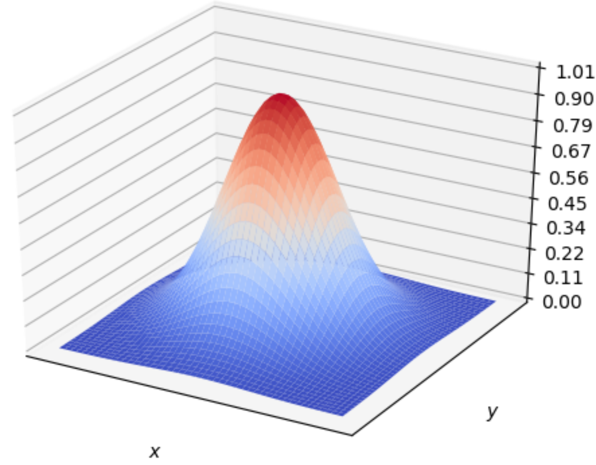

Figure 2: **Gaussian radial basis function in 2-D**
Basis function $\Phi(x, y) = e^{-\epsilon^2(x^2+y^2)}$

A linear combination of shifted versions of these basis functions is used:

$$S(x) = \sum_{i=0}^{N-1} \lambda_i \Phi(||x - x_i||) \tag{1}$$

$\lambda \in \mathbb{R}^N$ must satisfy

$$S|_\Xi \overset{!}{=} f|_\Xi$$

which can also be expressed as a linear system:

$$\underbrace{\begin{pmatrix} \Phi(||x_0 - x_0||) & \dots & \Phi(||x_{N-1} - x_0||) \\ \vdots & \ddots & \vdots \\ \Phi(||x_{N-1} - x_0||) & \dots & \Phi(||x_{N-1} - x_{N-1}||) \end{pmatrix}}_{=:A} \begin{pmatrix} \lambda_0 \\ \vdots \\ \lambda_{N-1} \end{pmatrix} \overset{!}{=} \begin{pmatrix} f_0 \\ \vdots \\ f_{N-1} \end{pmatrix} \tag{2}$$

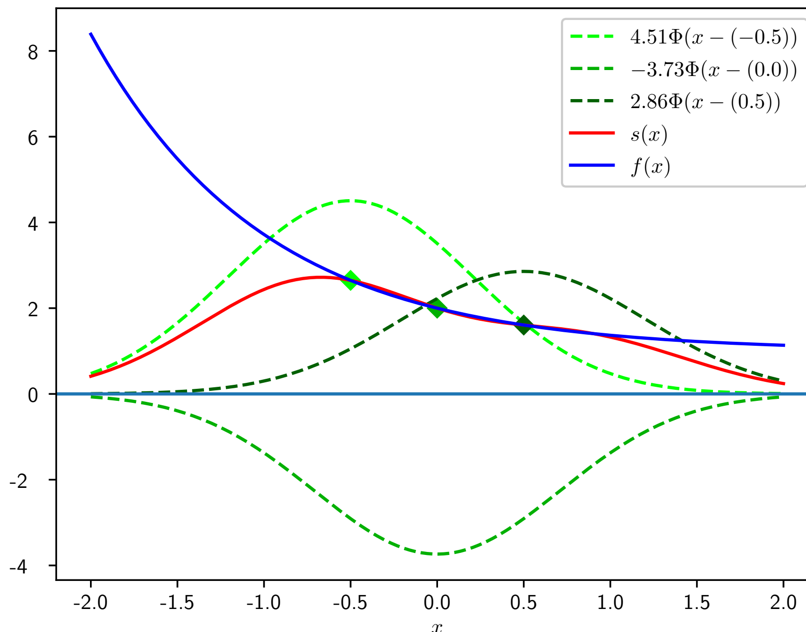

Figure 3: **Basic RBF interpolation with Gaussian basis functions**
RBF interpolation with $\epsilon = 1$, interpolating $f(x) = e^{-x} + 1$ at $\{-0.5, 0, 0.5\}$

Figure 3 shows how the interpolant s interpolates f by using Gaussian basis functions. It can be seen that s interpolates f exactly at the points $\{-0.5, 0, 0.5\}$ and because f is "sufficiently smooth", the error is rather small even for as few as 3 basis functions. The matrix A has some interesting properties. Firstly it is symmetric, which results directly from the symmetry of the basis functions. Furthermore for many RBF choices A is positive definite and therefore nonsingular for arbitrary centers.

2.3 Condition

Condition of matrices plays an important role when discussing radial basis function interpolation, so some basics are outlined in the following.
The condition of a nonsingular matrix $A \in \mathbb{R}^{N \times N}$ is defined as:

$$\kappa(A) = ||A||\ ||A^{-1}|| \tag{3}$$

for some norm $||\cdot||$. If A is normal (every symmetric matrix is normal) and $||\cdot||_2$ (the spectral norm) is chosen, (3) becomes

$$\kappa_{||\cdot||_2}(A) = \left|\frac{\lambda_{\max}(A)}{\lambda_{\min}(A)}\right| \tag{4}$$

so the condition is the ratio of the biggest to the smallest eigenvalue of A.
Many radial basis function choices are parameterized with a parameter ϵ, also referred to as the *shape parameter*. This parameter can control the condition of the resulting interpolation matrix. In 2005 Schaback ([15]) showed that the eigenvalues of the interpolation matrix A follow a pattern depending on ϵ for most RBF choices. In particular, all of the parameterized basis functions from Figure 1 (the last four) obey this pattern. They showed that in d dimensions the matrix A has

$$\binom{k+d-1}{d-1} \qquad (k \in \mathbb{N}_0)$$

eigenvalues of size $\mathcal{O}(\epsilon^{2k})$. The pattern is also given in table 1.

Power of ϵ	0	2	4	6
1-D	1	1	1	1
2-D	1	2	3	4
3-D	1	3	6	10
On the surface of a sphere	1	3	5	7

Table 1: Number of eigenvalues of size $\mathcal{O}(\epsilon^0)$, $\mathcal{O}(\epsilon^2)$, $\mathcal{O}(\epsilon^4)$, ...

This table is to be read by considering successively higher powers of ϵ until all N eigenvalues are accounted for.
For instance, a Matrix with $N = 4$ in 2-D would have one eigenvalue of size ϵ^0, two eigenvalues of size ϵ^2 and one eigenvalue of size ϵ^4. From here it becomes evident that there is some severe ill-conditioning in the matrix A.
Changing the shape parameter influences how the basis functions look and changes properties of the interpolations such as condition and accuracy. Consider the Gaussian basis functions in Figure 4

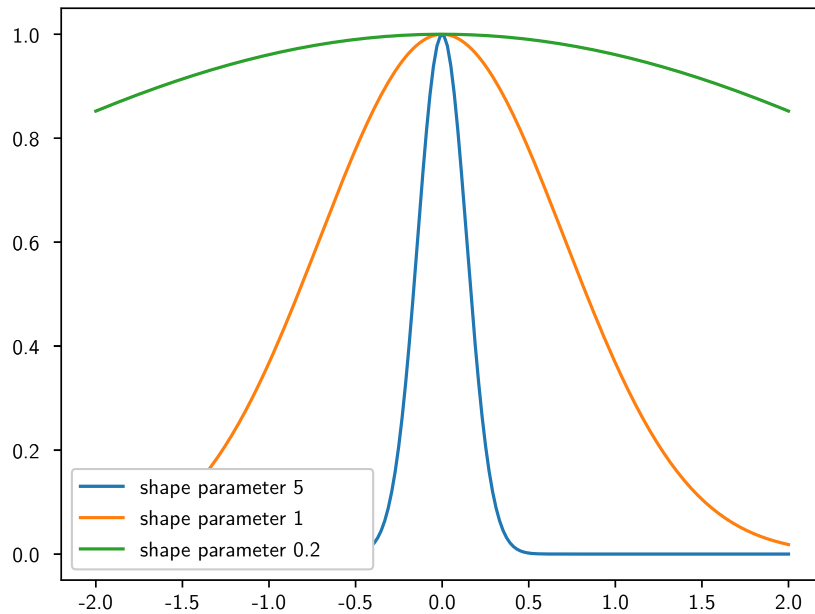

Figure 4: **Gaussian RBF for different shape parameters**
Choosing a small ϵ yields flat, global basis functions, whereas choosing a big ϵ yields tall, local basis functions.

Defining a cutoff radius for the basis function leads to some entries in A being 0, thus improving condition. This leads to a trade-off between accuracy and condition.

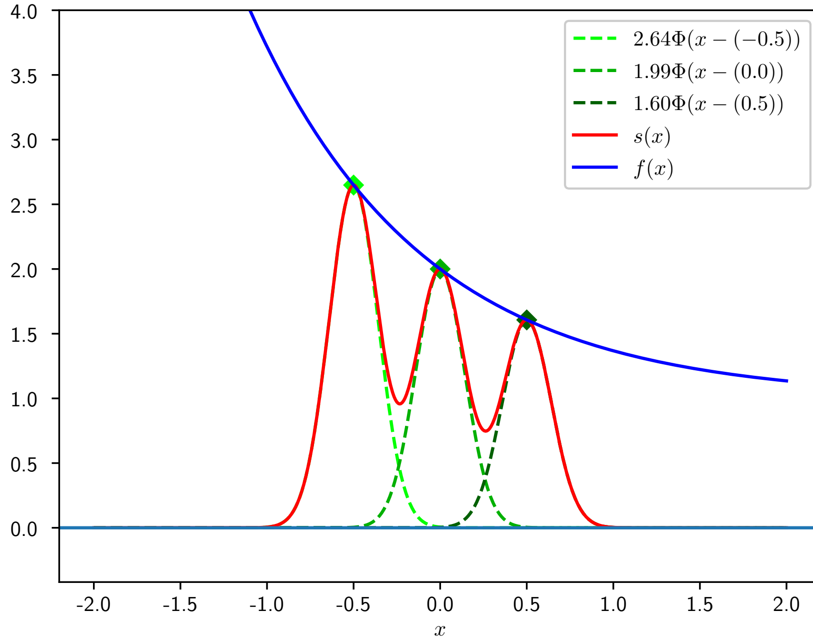

Figure 5: **Interpolation with bigger ϵ**
Interpolating $f(x) = e^{-x} + 1$ at $\{-0.5, 0, 0.5\}$ with $\epsilon = 5$. The red interpolant does not match the original function (blue) very well, thus accuracy of the interpolant is bad.

In figure 5 one can see that choosing a bigger value for ϵ yields skinnier basis functions at the expense of accuracy. Note how in figure 3 every basis function is global, whereas in figure 5 basis functions vanish at every center except the one center around which they are defined, thus creating an interpolation matrix A that is diagonal.
Finding the optimal shape parameter has been discussed in many previous works ([8]) and some basic terms are defined in section 5.1.

2.3.1 Improving condition by adding a polynomial

Adding a polynomial to the interpolation can improve condition and for some RBF choices it is even necessary in order to make sure the matrix A is unisolvent. This section is based on [14]. However the principle is very widely used, for instance [3] describes a similar approach. The principle will be outlined for first order polynomials.
The basic idea is to replace (1) by:

$$S(x) = \sum_{i=0}^{n} \gamma_i \Phi(||x - x_i||) + \beta_0 + \beta^T x \tag{5}$$

with $\beta \in \mathbb{R}^d$ and $\beta_0 \in \mathbb{R}$. The interpolation conditions are the usual conditions $S|_\Xi \stackrel{!}{=} f|_\Xi$ and some additional conditions for the polynomial:

$$\sum_{i=0}^{n} \gamma_i = 0 \text{ and } \sum_{i=0}^{n} \gamma_i x_i^{(j)} = 0 \tag{6}$$

for all $1 \leq j \leq d$ and with $x_i^{(j)}$ denoting the j-th coefficient of x_i.
The authors of [14] described two ways of including these additional requirements. The first is the *integrated polynomial*: As usual let x_i denote some data points. Let $f = (f(x_0), \ldots, f(x_{N-1}))^T$ denote the function values at these points:

$$\begin{pmatrix} 0 & Q^T \\ Q & P \end{pmatrix} \underbrace{\begin{pmatrix} \beta \\ \gamma \end{pmatrix}}_{=:p} = \begin{pmatrix} 0 \\ f \end{pmatrix}$$

with the usual interpolation matrix $P_{i,j} = \Phi(||x_i - x_j||_2)$ and an additional Matrix Q where the i-th row of Q is $(1\, x_i^{(1)} \ldots x_i^{(d)})$. The additional requirements are "integrated" inside the interpolation matrix.
For evaluation the matrix E is constructed by

$$E = \begin{pmatrix} V & P' \end{pmatrix}$$

where P' is given by $P'_{i,j} = \Phi(||y_i - x_j||_2)$ and $V_{i,} = (1\, y_i^{(0)} \ldots y_i^{(d)})$ with y_i denoting some output mesh node. The output values are then computed by

$$S = E \cdot p$$

The other way is the *separated polynomial*:
First solve the least-squares-problem

$$\begin{bmatrix} Q \end{bmatrix} \cdot [\beta] \approx \begin{bmatrix} f \end{bmatrix}$$

then subtract the polynomial values from f and solve for γ:

$$P \cdot \gamma = f - Q \cdot \beta$$

For evaluation the polynomial values have to be added again:

$$S = P' \cdot \gamma + V \cdot \beta$$

2.3.2 Improving condition by rescaling the interpolant

Rescaling the interpolant is a rather simple extension to RBF interpolation but has been shown in [6] to yield good results. One uses

$$\bar{S}(x) = \frac{S(x)}{S_1(x)} \tag{7}$$

with $s_1(x)$ being the interpolant for the constant function $g \equiv 1$ as the new interpolant. The algorithm is described in [6] and results are also found in [14].

2.3.3 Improving condition by using a variable shape parameter

Many meshes are not regular and equidistant. Setting a fixed shape parameter will therefore result in some basis functions spanning across many nodes and other basis functions across few. Furthermore one would usually want to flatten the basis functions a bit towards the boundary of a mesh, because some of their support will be outside the mesh. This would suggest using a variable shape parameter, which can be set for each basis function individually. However one would need some kind of local density measure to set this shape parameter usefully. This information can possibly be given by the input mesh but this would compromise one of the biggest strength of RBF interpolation: To work on arbitrary scattered data. Computing this information later is rather expensive as well and finding good algorithms is a nontrivial problem.

2.3.4 Improving condition by using a preconditioner

Precondition of linear systems is an approach that is used universally when solving linear systems. Instead of solving $Ax = B$, one solves

$$AP^{-1}Px = b$$

by solving the linear system

$$AP^{-1}y = b$$

and then solving

$$Px = y$$

Alternatively one can solve

$$PAx = Pb$$

The first way is called *right preconditioning* and the second way is called *left preconditioning* (depending on the side of A that P is multiplied to). The linear operator should be cheap to invert and the product $P^{-1}A$ should be better conditioned than A. Determining a "good" operator P is not easy. Iterative methods like *Jacobi method* or the *conjugate gradient method* are widely used. They provide operators P that are cheap to multiply with A and improve condition iteratively by applying the preconditioning principle multiple times until a sufficiently good condition is reached. However, often times these methods require a significant amount of computation. Improving condition by using preconditioners is discussed for instance in [9] or [2].

3 RBF-QR Interpolation

3.1 RBF and polynomial interpolation

In 2002, Driscoll and Fornberg ([7]) discovered that if radial basis functions meet some "simple conditions", they (rather surprisingly) converge to Lagrange polynomials in 1D. More formally they showed:
Let N distinct data points in 1-D be given. Suppose the basis function

$$\Phi(r,\epsilon) = a_0 + \epsilon^2 a_1 r^2 + \epsilon^4 a_2 r^4 + \dots \tag{8}$$

is such that the RBF system (2) has a solution for all $\epsilon > 0$. For integer n define the symmetric matrices G_{2n-1} and G_{2n} by:

$$G_{2n-1} = \begin{bmatrix} \binom{0}{0}a_0 & \binom{2}{2}a_1 & \dots & \binom{2n-2}{2n-2}a_{n-1} \\ \binom{2}{0}a_1 & \binom{4}{2}a_2 & \dots & \binom{2n}{2n-2}a_n \\ \vdots & \vdots & & \vdots \\ \binom{2n-2}{0}a_{n-1} & \binom{2n}{2}a_n & \dots & \binom{4n-4}{2n-2}a_{2n-2} \end{bmatrix}_{n\times n}$$

$$G_{2n} = \begin{bmatrix} \binom{2}{1}a_1 & \binom{4}{3}a_2 & \dots & \binom{2n}{2n-1}a_n \\ \binom{4}{1}a_2 & \binom{6}{3}a_3 & \dots & \binom{2n+2}{2n-1}a_{n+1} \\ \vdots & \vdots & & \vdots \\ \binom{2n}{1}a_n & \binom{2n+2}{3}a_{n+1} & \dots & \binom{4n-2}{2n-1}a_{2n-1} \end{bmatrix}_{n\times n}$$

If G_{n-1} and G_N are nonsingular, then the RBF interpolant $s(x,\epsilon)$, defined by

$$s(x,\epsilon) = \sum_{k=1}^{N} \lambda_k \Phi(||x - x_k||, \epsilon) \tag{9}$$

satisfies

$$\lim_{\epsilon\to 0} s(x,\epsilon) = L_N(x) \tag{10}$$

where

$$L_{N-1}(x) := \sum_{j=0}^{N-1} f_j l_j(x)$$

denotes the Lagrance interpolating polynomial for $f = (f_0, \dots, f_{N-1})$ on the nodes. This interpolating polynomial is defined with lagrange polynomials

$$l_j(x) := \prod_{\substack{0\le m\le N-1 \\ m\neq j}} \frac{x - x_m}{x_j - x_m}$$

The proof can be found in [7].
For infinitely smooth RBF choices (with gaussian RBFs being one of them) there is an expansion (8) with a taylor series, as will be seen later.

3.2 Higher dimension

In 2005, Larsson and Fornberg ([13])found a more general result applicable in higher dimensions. They were able to show that in the $\epsilon \to 0$ limit, depending on the node layout and the choice of the radial basis function, the RBF interpolant converges to polynomial interpolation. Moreover they were able to find a pattern for eigenvalues in n-D.

3.3 RBF-QR on the unit sphere

In 2007, Fornberg and Piret ([11]) found a method for interpolating with radial basis functions, when the nodes are scattered on the unit sphere, using spherical harmonics. This method is generalized to the nonperiodic euclidean space in section 3.4, however some key concepts of the algorithms are already outlined in this section.

3.3.1 A better basis

They key concept of RBF-QR is to change the basis from the bad basis of shifted RBFs to a better basis. This is not fundamentally new: When doing polynomial interpolation one often prefers the chebyshev basis $\{T_0(x), T_1(x), \dots\}$ over the monomial basis $\{1, x, x^2, \dots\}$. Both span the same space, but the latter leads to better conditioned interpolation.

3.3.2 Spherical Harmonics

For the RBF-QR algorithm on the unit sphere, so called **Spherical Harmonics (SPH)** are used. A class of functions that is defined as (assuming $(x, y, z) \in S^2$ with S^2 being the unit sphere):

$$Y_\mu^\nu(x,y,z) = \begin{cases} \sqrt{\frac{2\mu+1}{4\pi}}\sqrt{\frac{(\mu-\nu)!}{(\mu+\nu)!}}P_\mu^\nu(z)\cos(\nu\tan^{-1}(\frac{y}{x})) & v = 0, 1, \dots, \mu \\ \sqrt{\frac{2\mu+1}{4\pi}}\sqrt{\frac{(\mu-\nu)!}{(\mu+\nu)!}}P_\mu^{-\nu}(z)\sin(-\nu\tan^{-1}(\frac{y}{x})) & v = -\mu, \dots, -1 \end{cases}$$

where P_μ^ν are associated Legendre functions:

$$P_n^m(x) = (-1)^m(1-x^2)^{m/2}\frac{d^m}{dx^m}P_n(x)$$

with $P_n(x)$ being the Legendre polynomial of degree n:

$$P_n(x) = \frac{1}{2^n n!}\left(\frac{d^n}{dx^n}(x^2-1)^n\right)$$

3.3.3 SPH Expansions

These SPHs can be used to express an arbitrary function defined on the unit sphere:

$$s(x,y,z) = \sum_{\mu=0}^{\infty}\sum_{\nu=-\mu}^{\mu} c_{\mu,\nu}Y_\mu^\nu(x,y,z)$$

Hubbert and Baxter [1] give explicit expressions for the expansion of radial basis functions:

$$\Phi(||x - x_i||) = \sum_{\mu=0}^{\infty} {\sum_{\nu=-\mu}^{\mu}}' \{c_{\mu,\epsilon}\epsilon^{2\mu} Y_\mu^\nu(x_i)\} Y_\mu^\nu(x) \qquad (11)$$

Where $\sum'$ means that the term at $\nu = 0$ is halved. The coefficients take closed algebraic forms, for instance Gaussian radial basis functions yield

$$c_{\mu,\epsilon} = \frac{4\pi^{3/2}}{\epsilon^{2\mu+1}} e^{-2\epsilon^2} I_{\mu+1/2}(2\epsilon^2)$$

With I_α being modified bessel functions of the first kind.

3.3.4 SPH expansion expressed in matrix form

These expansions can be expressed in matrix form:

$$\Phi(x) = \begin{bmatrix} \Phi(||x - x_0||) \\ \Phi(||x - x_1||) \\ \vdots \\ \Phi(||x - x_{N-1}||) \end{bmatrix} = B \cdot Y(x)$$

with

$$B = \begin{bmatrix} \frac{c_{0,\epsilon}}{2} Y_0^0(x_0) & \frac{\epsilon^2 c_{1,\epsilon}}{1} Y_1^{-1}(x_0) & \frac{\epsilon^2 c_{1,\epsilon}}{2} Y_1^0(x_0) & \frac{\epsilon^2 c_{1,\epsilon}}{1} Y_1^1(x_0) & \dots \\ \frac{c_{0,\epsilon}}{2} Y_0^0(x_1) & \frac{\epsilon^2 c_{1,\epsilon}}{1} Y_1^{-1}(x_1) & \frac{\epsilon^2 c_{1,\epsilon}}{2} Y_1^0(x_1) & \frac{\epsilon^2 c_{1,\epsilon}}{1} Y_1^1(x_1) & \dots \\ \vdots & \vdots & \vdots & \vdots & \dots \\ \frac{c_{0,\epsilon}}{2} Y_0^0(x_{N-1}) & \frac{\epsilon^2 c_{1,\epsilon}}{1} Y_1^{-1}(x_{N-1}) & \frac{\epsilon^2 c_{1,\epsilon}}{2} Y_1^0(x_{N-1}) & \frac{\epsilon^2 c_{1,\epsilon}}{1} Y_1^1(x_{N-1}) & \dots \end{bmatrix}$$

and

$$Y(x) = \begin{bmatrix} Y_0^0(x) \\ Y_1^{-1}(x) \\ Y_1^0(x) \\ Y_1^1(x) \\ Y_2^{-2}(x) \\ Y_2^{-1}(x) \\ Y_2^0(x) \\ Y_2^1(x) \\ Y_2^2(x) \\ \vdots \end{bmatrix}$$

with $B \in \mathbb{R}^{N\times\infty}$ and $Y(x) \in \mathbb{R}^\infty$ Note that this idea is somewhat incomplete as the resulting matrices would be infinitely big due to the infinite expansion (11). Therefore these expansions must be truncated at some point.
Now split $B = CE$ (this can be done analytically by simply factoring out the increasing powers of ϵ).

$$B = \begin{bmatrix} & & \\ & C & \\ & & \end{bmatrix} \cdot \begin{bmatrix} \epsilon^0 & & & & & \\ & \epsilon^2 & & & & \\ & & \epsilon^2 & & & \\ & & & \epsilon^2 & & \\ & & & & \epsilon^4 & \\ & & & & & \ddots \end{bmatrix}$$

This is a very important step in the algorithm because it separates the (well-conditioned) matrix C from the powers of ϵ. The powers of ϵ are responsible for the bad conditioning of the original matrix, so they must be handled in a special way, which will be seen later.
Next split $C = QR$:

$$B = \begin{bmatrix} & & \\ & Q & \\ & & \end{bmatrix} \begin{bmatrix} r_{1,1} & r_{1,2} & r_{1,3} \\ & r_{2,2} & r_{2,3} \\ & & \ddots \end{bmatrix} \begin{bmatrix} \epsilon^0 & & \\ & \epsilon^2 & \\ & & \ddots \end{bmatrix}$$

However, neither C nor R are square. The expansion will always have more than N elements, so B (as well as C and R) will have more columns than rows. Even though a QR-decomposition is well-defined on arbitrary matrices, this will put some limitations on the algorithm to be used for computing this decomposition. More on that will be discussed in 4.6.
If B is multiplied with any nonsingular matrix from the left, the basis functions change but the space that is spanned by them stays the same.
Let $E_N \in \mathbb{R}^{N \times N}$ be the first N powers of E. Consider the new basis:

$$\Psi(x) := E_N^{-1} Q^T \Phi(x) = E_N^{-1} REY(x)$$

The matrix $E_N^{-1} RE$ has some convenient properties in that it is upper triangular, and only a few superdiagonals contain low powers of ϵ. Another important property is the way the powers of ϵ enter the expansion. The pattern of powers of ϵ exactly matches the sequence in Table 1. This is important because then conditioning improves at the same pace as it would be worsening, therefore staying invariant with ϵ. For instance if $N = 4$ the original matrix would have a smallest eigenvalue of size $\mathcal{O}(\epsilon^2)$ (and the biggest eigenvalue of size $\mathcal{O}(1)$ as always), but the matrix E would have powers of epsilon up to ϵ^2 as well, therefore extracting powers of ϵ at the exact same rate as they would worsen the condition of the resulting matrix.

3.4 Nonperiodic cartesian space

The principle from section 3.3 shall now be applied to the Gaussian basis function in nonperiodic cartesian space.The original basis is given by

$$\Phi(r) = e^{-(\epsilon r)^2}$$

Now, a polynomial expansion is needed. A taylor expansion would be a natural choice. Fortunately the function e^x is usually defined by its taylor series, so it's trivial to get:

$$e^x = \sum_{k=0}^{\infty} \frac{x^k}{k!}$$

The taylor expansion for the Gaussian basis function follows directly:

$$e^{-(\epsilon r)^2} = \sum_{k=0}^{\infty} \frac{(-(\epsilon r)^2)^k}{k!} = \sum_{k=0}^{\infty} \frac{(-\epsilon^2)^k}{k!} r^{2k}$$

The shifted variant in nonperiodic euclidean space is needed:

$$e^{-(\epsilon||x-x_0||)} = \sum_{k=0}^{\infty} \frac{(-\epsilon^2)^k}{k!}(||x-x_0||)^{2k} = \sum_{k=0}^{\infty} \frac{(-\epsilon^2)^k}{k!}((x-x_0)\cdot(x-x_0))^k$$

assuming the euclidean norm ($||x|| = \sqrt{x_0 + \cdots + x_n}$) is used (I will assume this for the remainder of this thesis unless noted otherwise). The next step would now be to check if this expansion does indeed improve conditioning at a sufficient rate. In other words, if the matrix E were constructed, which pattern of ϵ^{2k} would it have? Every expansion function is multiplied by a power of ϵ which would in the next step be factored out into E, so expansion functions are grouped by their power of ϵ. This reveals the following pattern in 1-D:

$$\{\{1\}, \{x, x^2\}, \{x^3, x^4\}, \dots\}$$

The sequence of how powers of ϵ enter E would then be

$$\{1, 2, 2, 2, \dots\}$$

In 2-D:

$$\{\{1\}, \{x^2, x, y, y^2\}, \{x^4, x^3, x^2y, x^2y^2, xy, xy^2, y^3, y^4\}\}$$

yielding

$$\{1, 4, 8, 12, 16, \dots\}$$

In 3-D:

$$\{\{1\}, \{x^2, x, y^2, y, z^2, z\},$$
$$\{x^4, x^3, x^2y^2, x^2y, x^2z^2, x^2z, xy^2, xy, x^2z^2, xz, y^4, y^3, y^2z^2, y^2z, yz^2, yz, z^4, z^3\}\}$$

yielding

$$\{1, 6, 18, \dots\}$$

None of these sequences match any of the sequences in Table 1. This means that this expansion does not eliminate ill-conditioning at the same rate as it occurs. For instance in 3-D, with $N = 20$ there are eigenvalues up to $\mathcal{O}(\epsilon^6)$. However, because only 20 new basis functions are brought in, ill-conditioning is only eliminated up to $\mathcal{O}(\epsilon^4)$. This effect worsens for bigger N at a disastrous rate. So naively applying the RBF-QR algorithm for arbitrary basis functions does not seem to work.
Fortunately Fornberg, Larsson and Flyer found an expansion for gaussian RBFs that matches the sequences in Table 1 exactly.

3.5 A better Expansion for Gaussian RBFs

In the following the Gaussian basis functions (centered around x_0) will be reconsidered:

$$\Phi(x, x_0) = e^{-(\epsilon||x-x_0||)^2} = e^{-\epsilon^2(x-x_0)\cdot(x-x_0)} = e^{-\epsilon^2(x\cdot x)}e^{-\epsilon^2(x_0\cdot x_0)}e^{2\epsilon^2(x_0\cdot x)} \tag{12}$$

Now look at the Factors independently:

$$e^{-\epsilon^2(x_0\cdot x_0)}$$

This is a constant factor, so it has no influence on the expansion.

$$e^{-\epsilon^2(x \cdot x)} \tag{13}$$

The authors of [10] called this factor "harmless as $\epsilon \to 0$", This will be motivated in the following. Section 3.7 will show that the mesh needs to be transformed to fit into the unit sphere, so it can be assumed that $||x|| \leq 1$. This means that

$$e^{-\epsilon^2(x \cdot x)} \geq e^{-\epsilon^2}$$

Now, having a (global) minimum for (13), the error of assuming that $e^{-\epsilon^2(x \cdot x)} \approx 1$ can be quantified as

$$E(\epsilon) = 1 - e^{-\epsilon^2}$$

Figure 6 shows how fast $E(x)$ converges to 0 as $\epsilon \to 0$

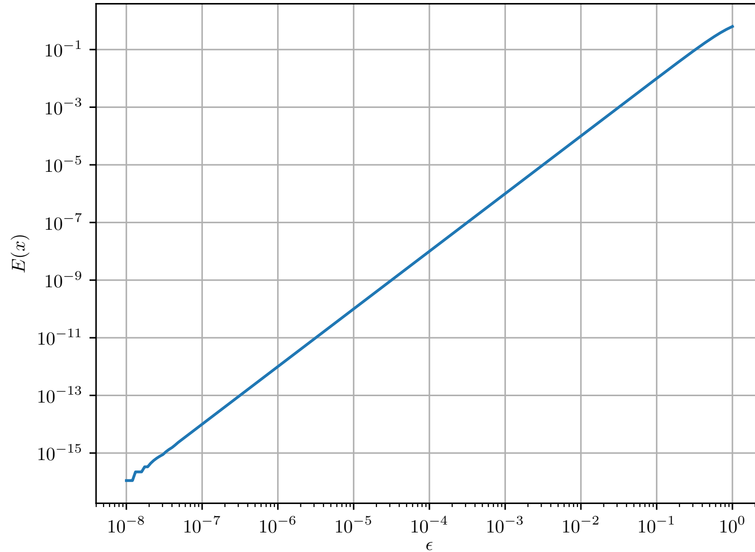

Figure 6: **Error $E(x)$ for different values of ϵ**

It can be seen that for $\epsilon \leq 10^{-8}$ the error is of the same magnitude as machine precision.
The last factor $e^{2\epsilon^2(x_0 \cdot x)}$ can be expressed in terms of a Taylor series:

$$e^{2\epsilon^2(x_0 \cdot x)} = \sum_{i=0}^{\infty} \frac{(2\epsilon^2)^i}{i!}(x \cdot x_0)^i$$

With $(x \cdot x_0)^i$ yielding new basis functions at the exact same rate as Table 1 predicts.

3.6 Polar coordinates

When using the RBF-QR method, one has to handle rather high-degree-monomials. This problem is not very uncommon, and can for instance be mitigated by using low-order piecewise polynomials. Unfortunately, this is not an option here. The authors of [10] proposed transforming the expansion for the Gaussian basis function (12) to polar coordinates. The full transformation can be found in the paper. I will just restate the new expansion and new basis. Let (r, θ) denote a point in polar coordinates. In 2-D this then becomes:

$$\begin{aligned}
\Phi(r,\theta,r_0,\theta_0) = 2\cdot e^{-\epsilon^2 r_0^2}\cdot e^{-\epsilon^2 r^2}[\\
(\epsilon^2 r_0 r)^0 \left\{\frac{1}{2}\cdot\frac{1}{0!0!}\Theta_0\right\} \\
+ (\epsilon^2 r_0 r)^2 \left\{\frac{1}{2}\cdot\frac{1}{1!1!}\Theta_0 + \frac{1}{2!0!}\Theta_2\right\} \\
+ (\epsilon^2 r_0 r)^4 \left\{\frac{1}{2}\cdot\frac{1}{2!2!}\Theta_0 + \frac{1}{3!1!}\Theta_2 + \frac{1}{4!0!}\Theta_4\right\} \\
+\cdots+ \\
(\epsilon^2 r_0 r)^1 \left\{\cdot\frac{1}{1!0!}\Theta_1\right\} \\
+ (\epsilon^2 r_0 r)^3 \left\{\cdot\frac{1}{2!1!}\Theta_1 + \frac{1}{3!0!}\Theta_3\right\} \\
+ (\epsilon^2 r_0 r)^5 \left\{\cdot\frac{1}{3!2!}\Theta_1 + \frac{1}{4!1!}\Theta_3 + \frac{1}{5!0!}\Theta_5\right\} \\
+\dots \\
] \qquad (14)
\end{aligned}$$

with $\Theta_m = (\cos m\theta_0 \cos m\theta + \sin m\theta_0 \sin m\theta)$ and the expansion split for even and odd powers of ϵ^2 ($\{1, \epsilon^4, \epsilon^8, \dots\}$ and $\{\epsilon^2, \epsilon^6, \epsilon^{10} \dots\}$). The new basis functions are

$$\begin{aligned}
e^{-\epsilon^2 r^2}\{ \\
\{1\}, \\
r\{\cos\theta, \sin\theta\}. \\
r^2\{1, \cos 2\theta, \sin 2\theta\}, \\
r^3\{\cos\theta, \sin\theta, \cos 3\theta, \sin 3\theta\}, \\
\dots\} \qquad (15)
\end{aligned}$$

3.7 Chebyshev polynomials

The authors of [10] even went one step further and expressed (12) in terms of chebyshev polynomials, thus improving condition even further. I will again just outline the final expressions, but this time for all dimensions, as those are the final expansions. Chebyshev polynomials are defined on $[-1, 1]$ by:

$$T_0(x) = 1$$

$$T_1(x) = x$$

$$T_{n+1}(x) = 2x\,T_n(x) - T_{n-1}(x)$$

Or equivalently:

$$T_n(x) = \cos(n \arccos(x)) \tag{16}$$

3.7.1 1-D Final Expansion

In 1-D the expansion is

$$\Phi_k(x) = \sum_{j=0}^{\infty} d_j c_j(x_k) \tilde{T}_j(x) \tag{17}$$

with expansion functions

$$\tilde{T}_j(x) = e^{-\epsilon^2 x^2} T_j(x)$$

These expansion functions can be seen in figure 7 and figure 8 for different values of ϵ.

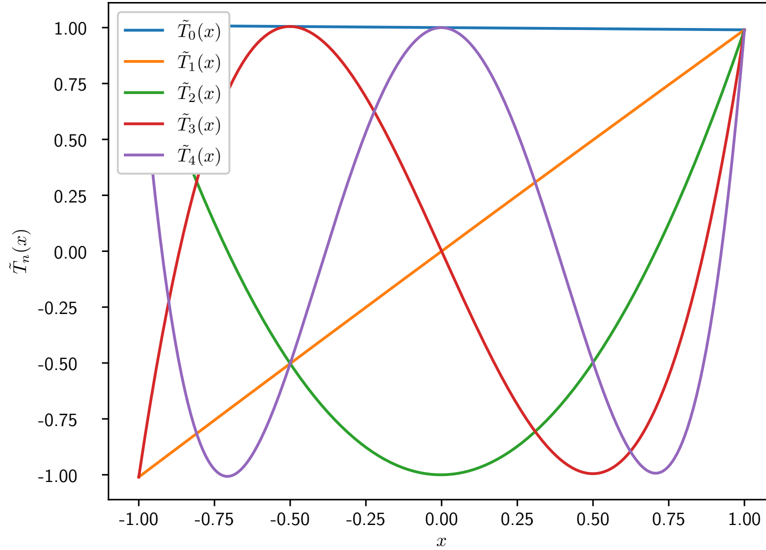

Figure 7: **New basis functions for a small ϵ**
Choosing $\epsilon = 0.1$ yields basis functions that are close to chebyshev polynomials.

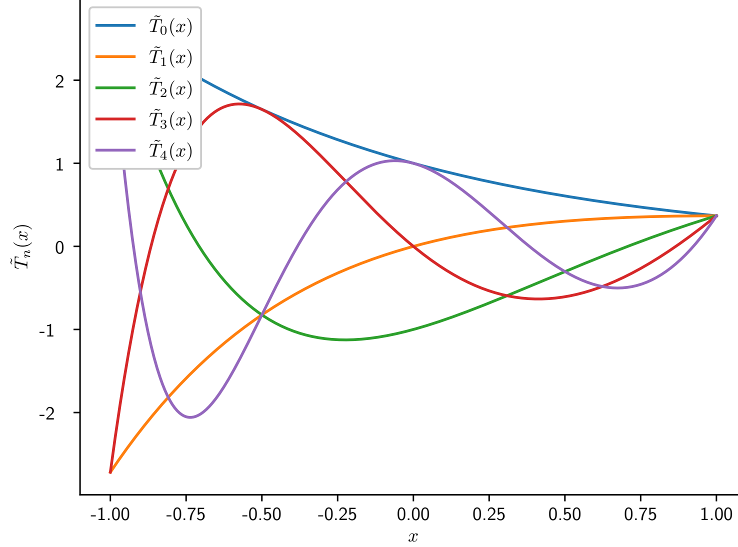

Figure 8: **New basis functions for a bigger ϵ**
Choosing $\epsilon = 1$ yields basis functions that are not quite chebyshev polynomials. Note however that for $x \to 0$, they still converge to chebyshev polynomials (This can also be seen from factor (13))

Scale factors and coefficients are given by

$$d_j = \frac{2\epsilon^{2j}}{j!} \tag{18}$$

$$c_j(x_k) = t_j e^{-\epsilon^2 x_k^2} x_k^j {}_0F_1([], j+1, \epsilon^4 x_k^2)$$

$$t_j = \begin{cases} \frac{1}{2} & j = 0 \\ 1 & j > 0 \end{cases}$$

3.7.2 2-D Final Expansion

In 2-D the expansion is more complex:

$$\begin{aligned} \Phi_k(x) &= \sum_{j=0}^{\infty} \sum_{m=0}^{\frac{j-p_j}{2}} d_{j,m} c_{j,m}(x_k) T^c_{j,m}(x) \\ &+ \sum_{j=0}^{\infty} \sum_{m=1-p_j}^{\frac{j-p_j}{2}} d_{j,m} s_{j,m}(x_k) T^s_{j,m}(x) \end{aligned} \tag{19}$$

$$p_j = \begin{cases} 1 & \text{if } j \text{ is odd} \\ 0 & \text{else} \end{cases}$$

Expansion functions are (input points must be given in polar coordinates $x_k = (r_k, \theta_k)$):

$$\begin{cases} T^c_{j,m}(x) = e^{-\epsilon^2 r^2} r^{2m} T_{j-2m}(r)\cos((2m+p_j)\theta), \\ T^s_{j,m}(x) = e^{-\epsilon^2 r^2} r^{2m} T_{j-2m}(r)\sin((2m+p_j)\theta), 2m+p_j \neq 0 \end{cases}$$

with coefficients

$$c_{j,m}(x_k) = b_{2m+p_j} t_{j-2m} e^{-\epsilon^2 r_k^2} r_k^j \cos((2m+p_j)\theta_k) {}_1F_2(\alpha_{j,m}, \beta_{j,m}, \epsilon^4 r_k^2)$$

$$s_{j,m}(x_k) = b_{2m+p_j} t_{j-2m} e^{-\epsilon^2 r_k^2} r_k^j \sin((2m+p_j)\theta_k) {}_1F_2(\alpha_{j,m}, \beta_{j,m}, \epsilon^4 r_k^2)$$

with helping coefficients $b_0 = 1$ and $b_m = 2,\ m > 0$ as well as $t_0 = \frac{1}{2}$ and $t_j = 1,\ j > 0$. Parameters for the hypergeometric function ${}_1F_2$ are: $a_{j,m} = \frac{j-2m+p_j+1}{2}$ and $\beta_{j,m} = \left[j - 2m + 1k, \frac{j+2m+p_j+2}{2}\right]$
The scaling coefficients are:

$$d_{j,m} = \frac{\epsilon^{2j}}{2^{j-2m-1}\left(\frac{j+2m+p_j}{2}\right)!\left(\frac{j-2m-p_j}{2}\right)!} \tag{20}$$

3.7.3 3-D Final Expansion

In the following, the *normalized associated legendre polynomials* will be defined by

$$P_n^m(x) = (-1)^m \sqrt{\frac{\left(n+\frac{1}{2}\right)(n-m)!}{(n+m)!}} P_n^{(m)}(x)$$

where $P_n^{(m)}$ denote the (unnormalized) associated legendre polynomials

$$P_n^{(m)}(x) = \frac{(-1)^m}{2^n n!}(1-x^2)^{m/2}\frac{d^{n+m}}{dx^{n+m}}(x^2-1)^n$$

The expansion in 3-D is given by

$$\Phi_k(x) = \sum_{j=0}^{\infty}\sum_{m=0}^{\frac{j-p_j}{2}} d_{j,m} \sum_{\nu=-(2m+p_j)}^{2m+p_j} c_{j,m,\nu}(x_k) T_{j,m,\nu}(x) \tag{21}$$

with expansion functions

$$T_{j,m,\nu}(x) = e^{-\epsilon^2 r^2} r^{2m} Y^{\nu}_{2m+p_j}(\theta, \Phi) T_{j-2m}(r)$$

with spherical coordinates defined with θ as the colatitude ($\theta = 0$ is the north pole) and

$$Y_\mu^\nu(\theta,\Phi) = P_\mu^\nu(\cos\theta)\cos(\nu\Phi),\ \nu = 0,\ldots,\mu$$

$$Y_\mu^{-\nu}(\theta,\Phi) = P_\mu^\nu(\cos\theta)\sin(\nu\Phi),\ \nu = 1,\ldots,\mu$$

with $P_\mu^\nu(z)$ being normalized associated Legendre functions. The scaling coefficients are given by

$$d_{j,m} = 2^{3+p_j+4m}\epsilon^{2j}\frac{\left(\frac{j+p_j+2m}{2}\right)!}{\left(\frac{j-p_j-2m}{2}\right)!\,(j+1+p_j+2m)!}$$

and coefficients becoming

$$c_{j,m,\mu}(x_k) = t_{j-2m} y_\mu e^{-\epsilon^2 r_k^2} r_k^j Y_{2m+p_j}^\nu (\theta_k, \Phi_k) {}_2F_3(\rho_{j,m}, \sigma_{j,m}, \epsilon^4 r_k^2)$$

where $y_0 = \frac{1}{2}$ and $y_\nu = 1,\ \nu > 0$ and the parameters to the hypergeometric function ${}_2F_3$ are $\rho_{j,m} = \left[\frac{tj-2m+1}{2}, \frac{j-2m+2}{2}\right]$ and $\sigma_{j,m} = \left[j - 2m + 1, \frac{j-2m+p_j+2}{2}. \frac{j+2m+p_j+3}{2}\right]$. When implementing these expansions, some issues will arise that will be discussed in section 4.

3.8 The final algorithm

The Algorithm can be expressed in matrix form, but there is still one issue with the previous expansions: They are infinite, so these expansions have to be truncated at some $j = j_{max}$. Determining this j_{max} will be discussed in section 4.5. Define K (d denotes the number of dimensions)[1]:

$$K := \begin{pmatrix} j_{max} + d \\ d \end{pmatrix} \geq N \tag{22}$$

First, express the original basis Φ as

$$\Phi(x) = C \cdot D \cdot T(x) \tag{23}$$

In 1-D the matrices are given by $(\Phi(x))_k := \phi_k(x) := \phi(||x - x_k||)$, $C_{k,j} := c_j(x_k)$, $D_{i,i} := d_i$, $(T(x))_i := \tilde{T}_i(x)$ and
$\Phi(x) \in \mathbb{R}^N$, $C \in \mathbb{R}^{N \times K}$, $D \in \mathbb{R}^{K \times K}$, $T(x) \in \mathbb{R}^K$.
Now split

$$C = Q \cdot [R_1\ R_2]$$

with $R_1 \in \mathbb{R}^{N \times N}$ and upper triangular (and $R_2 \in \mathbb{R}^{N \times (K-N)}$) and rewrite D as

$$D = \begin{bmatrix} D_1 & 0 \\ 0 & D_2 \end{bmatrix}$$

with $D_1 \in \mathbb{R}^{N \times N}$ and $D_2 \in \mathbb{R}^{(K-N) \times (K-N)}$
This way, (23) can be written as

$$\begin{aligned} \Phi(x) &= Q \cdot [R_1\ R_2] \begin{bmatrix} D_1 & 0 \\ 0 & D_2 \end{bmatrix} \cdot T(x) \\ &= Q \cdot [R_1 D_1\ R_2 D_2] \cdot T(x) \end{aligned} \tag{24}$$

Now multiply from the left with $D_1^{-1} R_1^{-1} Q^T$ to obtain a new basis that spans the same manifold as the original basis:

$$\begin{aligned} \Psi(x) :&= D_1^{-1} R_1^{-1} Q^T \Phi(x) \\ &= \begin{bmatrix} I & D_1^{-1} R_1^{-1} R_2 D_2 \end{bmatrix} \cdot T(x) \end{aligned} \tag{25}$$

Computing $D_1^{-1} R_1^{-1} R_2 D_2$ is a bit involved and will be discussed in section 4.4. For now, simply define $\tilde{R} := D_1^{-1} R_1^{-1} R_2 D_2$ and rewrite (25) as

$$\Psi(x) = \begin{bmatrix} I\ \tilde{R} \end{bmatrix} T(x) \tag{26}$$

[1] This is the M from [10, equation (5.1)], but M is used ambiguously there

Now compute

$$\begin{aligned} A &:= [\Psi(y_0) \cdots \Psi(y_{M-1})]^T \\ &= [T(y_0) \cdots T(y_{M-1})]^T \begin{bmatrix} I \\ \tilde{R}^T \end{bmatrix} \\ &= T_1^T + T_2^T \tilde{R}^T \end{aligned} \tag{27}$$

Where $(T_1)_{i,j} := \tilde{T}_i(y_j)$, $T_1 \in \mathbb{R}^{N \times M}$ and
$(T_2)_{i,j} := \tilde{T}_{N+i}(y_j)$, $T_2 \in \mathbb{R}^{(K-N) \times M}$
Then solve

$$A\lambda = f$$

for interpolation weights λ. Use those to compute the interpolant

$$s(\dot{x}) = \Psi(\dot{x})^T \lambda$$

at any given point $\dot{x}$.

3.9 Stability of the RBF-QR algorithm

The QR decomposition is inherently stable and the matrix C on which it is performed contains entries of size $\mathcal{O}(1)$. Another important part of the algorithm is the matrix $\tilde{R}$. Here it is interesting to look at how the powers of ϵ enter the matrix. They are contained in D_1^{-1} and D_2. The pattern how they enter $\tilde{R}$ is described by the following schema:

$$\begin{bmatrix} \epsilon^{2N} & \epsilon^{2N+2} & \cdots & \epsilon^{2K} \\ \epsilon^{2N-2} & \epsilon^{2N} & \cdots & \epsilon^{2K-2} \\ \vdots & \vdots & \ddots & \vdots \\ \epsilon^{0} & \epsilon^{2} & \cdots & \epsilon^{2N} \end{bmatrix}$$

So it can be seen that $\tilde{R}$ is mostly lower trapezoidal, as entries get successively smaller towards the upper right edge. This makes $\tilde{R}$ (and therefore A) well-conditioned.

4 Implementation

The implementation of the algorithm was done in two steps. First, it was implemented in Python in a very idiomatic way, with hardly any optimizations applied. The second step was then to implement it in C++.

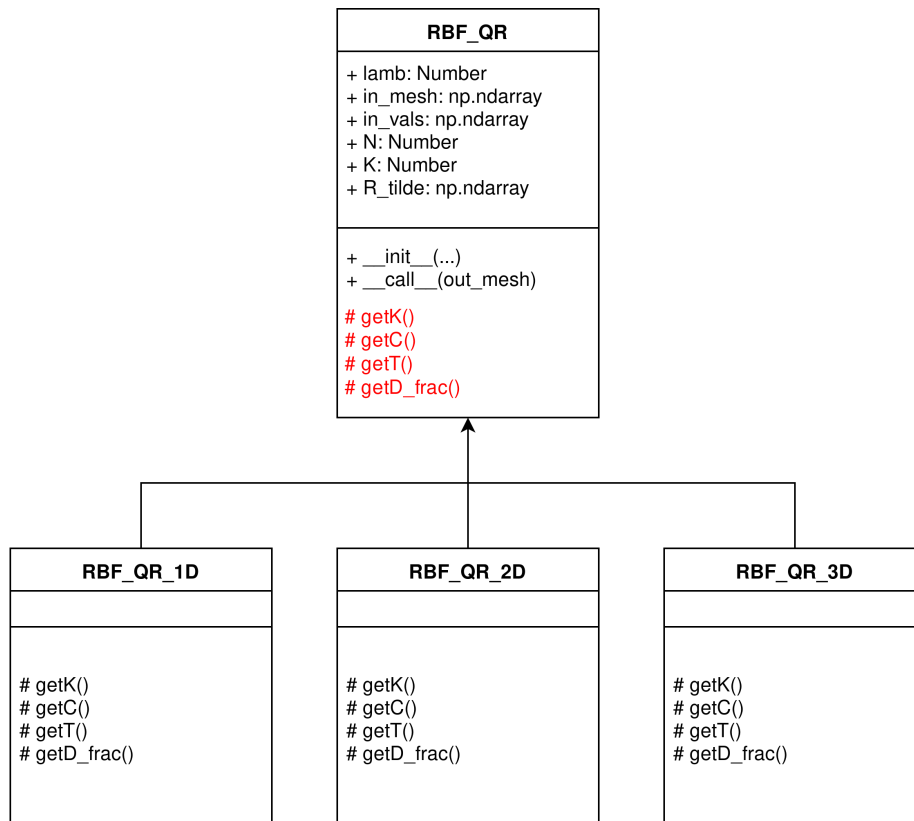

Figure 9: **The class hierarchy of the python implementation**
Red functions are pure virtual functions.

4.1 The Python implementation

The python implementation can be found in the public Github repository PyRBF[2]. It uses SciPy[3] for linear algebra and special functions. The code is organized into four classes that can be seen in Figure 9.

The function `getK()` computes j_{max} and then returns K (more on that in section 4.5). `getC()` assembles the C matrix according to the respective expansion. Some difficulties that arise when assembling C are given in section 4.3. `getT` is responsible for getting the new expansion functions ($T(x)$). This is done by actually returning a list of functions (however the C++ implementation uses a more efficient approach). The function `getD_frac()` returns a special matrix D_{frac} that is constructed by multiplying the ϵ-powers of D_1^{-1} in a column vector with a row vector consisting of the powers of ϵ in D_2. For instance in

[2]https://github.com/floli/PyRBF
[3]https://scipy.org/

1-D this yields:

$$D_{\text{frac, 1D}} = \begin{pmatrix} \epsilon^0 \\ \epsilon^{-2} \\ \vdots \\ \epsilon^{-2N} \end{pmatrix} \cdot \begin{pmatrix} \epsilon^{2N} & \epsilon^{2N+2} & \dots & \epsilon^{2K} \end{pmatrix} \tag{28}$$

This matrix has the same shape as $R_1^{-1}R_2$ and the entrywise product (Hadamard product) $D_{\text{frac}} \circ (R_1^{-1}R_2)$ equals $\tilde{R}$.

The rest of the algorithm is split into an *offline phase* (`__init__(...)`) and an *online phase* (`__call__(...)`). The offline phase consists of determining K, assembling C, computing $QR = C$, computing $\tilde{R}$, computing $A = T_1^T + T_2^T \tilde{R}^T$ and solving $A\lambda = f$ for lambda. The online phase then only evaluates the interpolant $s(\hat{x})$ on some output mesh. The offline phase is computationally way more expensive than the online phase, but if the input mesh and the input values do not change it only has to be performed once. Then the values can be interpolated onto any output mesh cheaply. The implementation of `__init__(...)` and `__call__(...)` is rather straightforward as it only requires calling the respective SciPy routines. In fact the entire code of `__init__(...)` is given in listing 1 and consists of as few as 15 lines of code. The parameters `scale` and `translate` indicate another issue with the RBF-QR algorithm, because it only works on for points x with $||x|| < 1$ (this is because of the chebyshev polynomials used). Therefore the mesh is scaled and translated to fit into this space and after the mapping is computed, the inverse scaling and translation is done. NumPy (part of SciPy) uses the `@`-operator to express matrix multiplication and the `*`-operator expresses entrywise multiplication (and applying broadcasting rules if the shapes don't match).

Listing 1: The offline phase in python

```python
def __init__(self, shape_param, in_mesh, in_vals):
  self.shape_param, self.in_mesh, self.in_vals
        = shape_param, np.copy(in_mesh), np.copy(in_vals)
  in_mesh = self.in_mesh  # update reference after copy
  self.N = M = N = in_mesh.shape[1]
  # Step 1: Compute jmax and K
  self.K = K = self._get_K(np.float64)
  # Step 2: Assemble C
  C = self._get_C()
  # Step 3: QR decomposition of C and R_tilde
  Q, R = np.linalg.qr(C)
  R_dot = solve_triangular(R[:, :N], R[:, N:K])
  D_fraction = self._get_D_frac()
  R_tilde = R_dot * D_fraction
  # Step 4: Evaluate expansion functions on in_mesh and compute A
  T = np.empty((K, M))
  T_dynamic = self._get_T()
  for i in range(K):
    T[i, :] = T_dynamic[i](in_mesh)
  self.A = A = T[:N, :].T + T[N:K, :].T @ R_tilde.T
```

```
  # Step 5: Solve for lambda
  self.lamb = np.linalg.solve(A, in_vals)
  # Step 6: Prepare evaluation
  self.I_R_tilde = np.hstack((np.identity(N), R_tilde))
```

The code for the online phase is even shorter and is found in listing 2. It basically only performs a matrix multiplication $[I\,\tilde{R}] \cdot T(\dot{x}) \cdot \lambda$

Listing 2: The online phase in python

```
def __call__(self, out_mesh):
  # Step 6: Evaluate
  out_length = out_mesh.shape[1]
  T_out = np.empty((self.K, out_length))
  T_dynamic = self._get_T()
  for i in range(self.K):
  T_out[i, :] = T_dynamic[i](out_mesh)
  Psi_out = self.I_R_tilde @ T_out
  predicition = Psi_out.T @ self.lamb
  return predicition
```

4.2 The C++ Implementation

The python implementation has three major drawbacks: It is potentially slow because every entry in the matrices is assembled by python. The other problem is about scalability: The python code is inherently single-threaded. Thirdly the preCICE project is written in C++, so this mostly rules out python for the final implementation. The C++ Implementation must be capable of running inside a distributed parallel application like preCICE which assumes a special execution model.

4.2.1 Memory and communication model

For massively parallel computations one can no longer assume that there is some memory that every process can randomly access. Thus one must assume *distributed memory*. An example is outlined in figure 10. Every node consists of one or more processors, which share some kind of memory. Reads and writes to this memory are assumed to be rather fast and random access is possible. When processors of different nodes want to communicate with each other, some kind of transmission medium has to be used. This is assumed to be rather slow, and there is no random access possible. Instead *message passing* is used. Each processor can communicate with another processor by sending messages. This model has some implications on how to parallelize algorithms, because in parallel algorithms one will rather think in terms of message passing instead of synchronization.

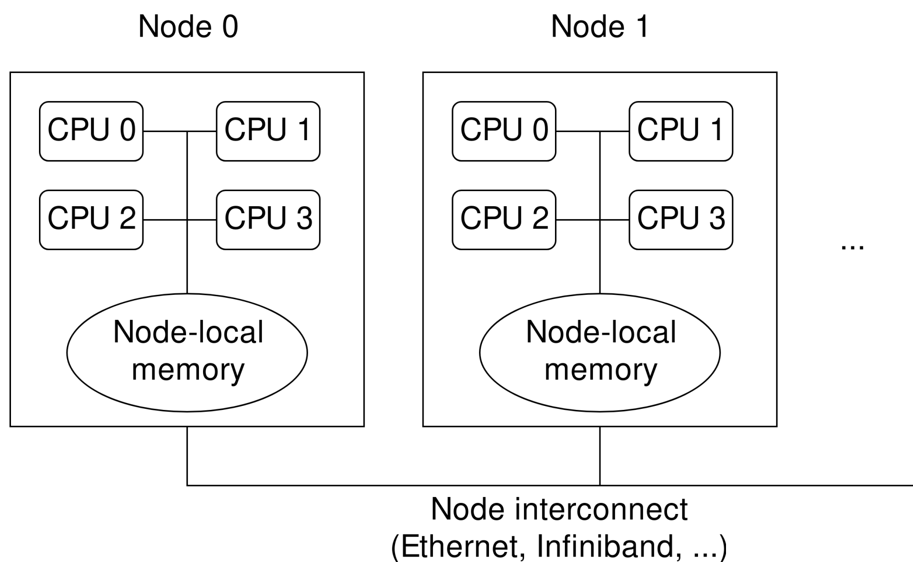

Figure 10: **Distributed memory**
Each CPU has some memory shared with other CPUs and some memory that is remote. Sometimes communication between CPUs can use shared memory, but sometimes the node interconnect has to be used, which is usually much slower than shared memory. Thus avoiding communication is a major goal when designing a parallel algorithm.

4.2.2 MPI

For sending these messages the Message Passing Interface[4] standard became the de-facto standard for HPC applications.
In MPI message passing is performed between processes (also called *ranks* there). The programmer usually only specifies the message and MPI handles the actual transmission via shared memory, IPC, Ethernet, Infiniband, etc. Aside from simple send/receive routines there is also some collective communication possible. Two of these collective routines were used and shall be described briefly here:
`MPI_Gatherv(...)` collects data from all ranks to one rank.
`MPI_Allgatherv(...)` lets every rank specify some data which is then exchanged in a way that every rank has the data of every rank.
More information on these routines is given in [12].

4.2.3 Eigen

Eigen[5] is a library for (sequential) linear algebra. It is widely used in scientific applications, but in this application it is only used wherever no parallel algorithm was available.

[4]https://www.mpi-forum.org/
[5]http://eigen.tuxfamily.org

4.2.4 PETSc

PETSc[6] is a library for parallel solution of partial differential equations. As such it has some algorithms for solving linear system in a parallel way. It is already used in preCICE and therefore using it will not introduce new dependencies to preCICE. The C++ implementation also uses PETSc to some extent, but unfortunately PETSc is specialized on solving PDEs so it does not have much support for dense linear algebra.

4.2.5 Elemental

Elemental[7] is yet another library for parallel linear algebra, but with support for dense linear algebra. It is used as an optional feature for the implementation, as preCICE does not use Elemental yet. For evaluation elemental has also been used, but proved to be problematic as well, as can be seen in section **??**.

4.2.6 Architecture

The architecture of the C++ implementation is quite similar to figure 9. However in the python implementation the virtual functions returned a full matrix, whereas in the C++ Implementation they only return one entry (they are given a row and a column as parameter). `getT()` can now simply be implemented as `getT(int index, VectorXd x)` (`VectorXd` is a vector of arbitrary size containing doubles in Eigen), actually evaluating the respective function at some point `x`. This is possible because C++ does not necessary need virtual functions to realize inheritance, but can instead perform some kind of "compile-time-inheritance" (CRTP). This has the advantage that the compiler can see through the function call and possibly inline/vectorize the matrix assembly.

4.2.7 Parallel Implementation

The C++ implementation tries to stay as close to the python implementation as possible, but sometimes has to diverge in order to apply optimizations or because of parallelization. When looking at the performance of the algorithm, there are two major places for optimization (they are the routines with $\mathcal{O}(N^3)$ complexity, the rest is $\mathcal{O}(N^2)$). The first is the QR factorization. Unfortunately PETSc does not provide a routine to compute a general QR factorization, so I had to roll my own implementation (actually, PETSc has routines which use QR factorizations internally but they don't expose the matrices publicly and don't construct the full matrix explicitly). Details can be found in section 4.6. The other place is when λ is computed from $A\lambda = f$. Solving linear systems in a stable and fast way is a delicate problem, far beyond the scope of this thesis, but since A is rather well-conditioned a fast algorithm like LU-factorization or even simple gauss elimination (both with column pivoting) would probably be sufficient. However PETSc only provides a solver based on a QR-factorization, called `KSPLSQR`, for dense linear algebra. It is a general solver that is also applicable for least-squares problems. In section **??** the performance of this solver can be seen. Figure 11 shows how the algorithm is implemented in C++. There are quite a few steps of the algorithm implemented in a serial fashion, which is due to the fact that

[6]https://www.mcs.anl.gov/petsc/
[7]http://libelemental.org/

PETSc does not support multiplying dense matrix in parallel. Instead, I focused on parallelizing the expensive parts of the computation.

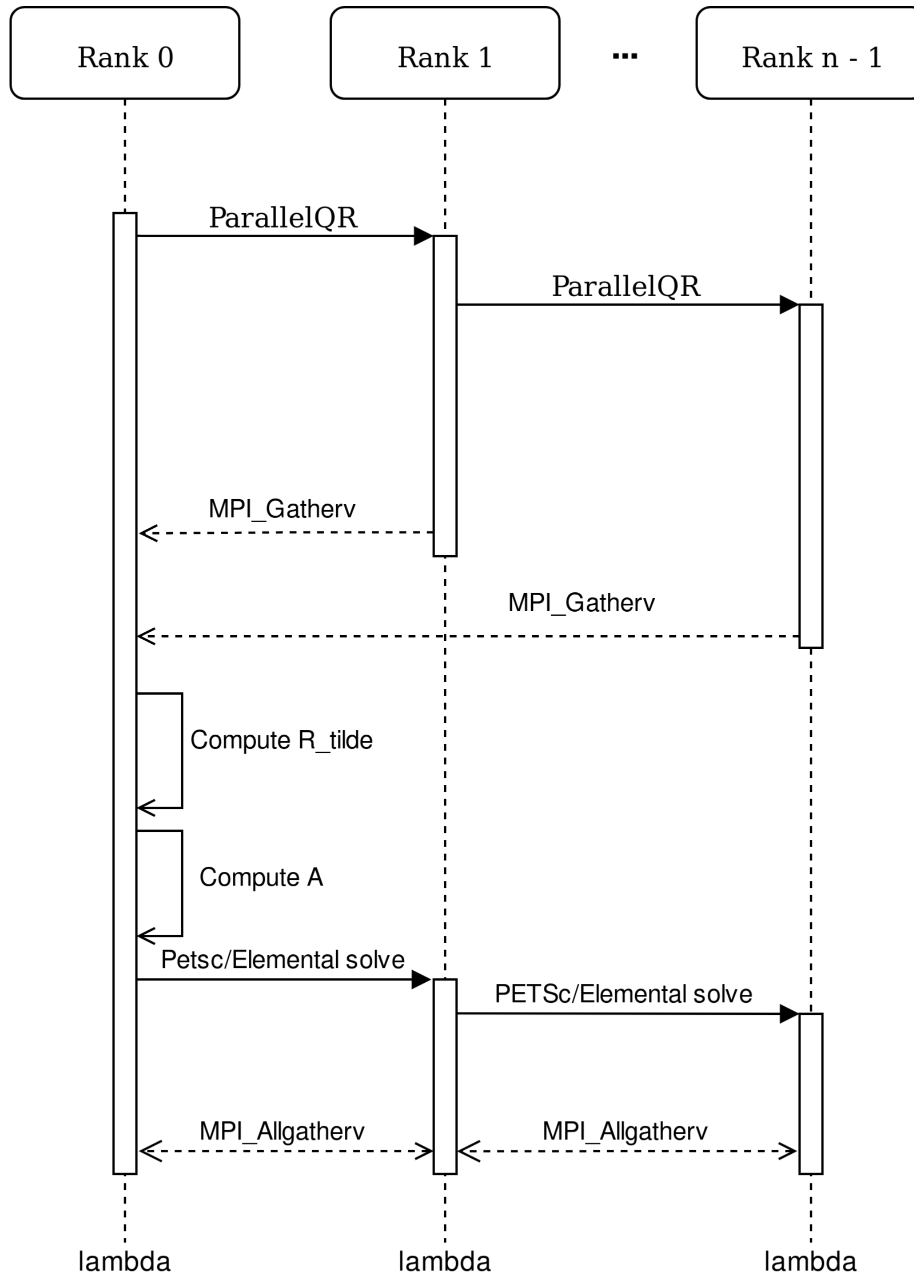

Figure 11: **The offline phase in C++**
The Parallel QR decomposition is executed first. This combines assembling and factoring the matrix. The results are then collected at rank 0, which then computes $\tilde{R}$ and A in a serial fashion. The equation $A\lambda = f$ is then solved in a parallel fashion, which is done by using PETSc or Elemental.

4.3 Index transformation

When assembling the Matrix C the question of how to map the expansion to C arises. Usually some index i denoting the number of terms so far is given. One is interested in the highest j that is reached within expansion (19). In 2-D, the number of terms are connected to the highest j in the following way:

$$s(j_m) := \sum_{j=0}^{j_m} \sum_{m=0}^{\frac{j-p_j}{2}} 1 + \sum_{j=0}^{j_m} \sum_{m=1-p_j}^{\frac{j-p_j}{2}} \tag{29}$$

$$= \sum_{j=0}^{j_m} \left(\frac{j-p_j}{2} + 1 + \frac{j-p_j}{2} + p_j \right)$$

$$= \sum_{j=0}^{j_m} (j+1)$$

$$= \frac{(j_m+1)(j_m+2)}{2} \tag{30}$$

$s(j_m)$ gives the maximum number of terms that can be reached if $j \leq j_m$. Any number of terms can be expressed as:

$$n = s(j_m) + m$$

for some $j_m \in \mathbb{Z}$ and some $m < j_m$. This makes it possible to express the matrices from section 3.8 for the 2-D case in the following way:

$$D_{\frac{j(j+1)}{2}+m, \frac{j(j+1)}{2}+m} = \begin{cases} d_{j,m} & m \leq \frac{j-p_j}{2} \\ d_{j,m-\frac{j+p_j}{2}} & \text{else.} \end{cases}$$

$$C_{k,\frac{j(j+1)}{2}+m} = \begin{cases} c_{j,m}(x_k) & m \leq \frac{j-p_j}{2} \\ s_{j,m-\frac{j+p_j}{2}}(x_k) & \text{else.} \end{cases}$$

$$T(x)_{\frac{j(j+1)}{2}+m} = \begin{cases} T^c_{j,m}(x) & m \leq \frac{j-p_j}{2} \\ T^s_{j,m-\frac{j+p}{2}}(x) & \text{else} \end{cases}$$

Some way to compute j and m from a given index i when assembling $C_{k,i}$ is needed. A simple solution would be to maintain an array of all needed indices j, m inside an array of size K. This is also the way the authors of [10] implemented this issue in their attached MATLAB Code. However, this means having an additional lookup in the loop assembling C.

Fortunately there is another possibility. One just needs to invert $s(n-1) = \frac{n(n+1)}{2}$ using the identity

$$\forall y \in \mathbb{Z}: \ \max \left\{ j \in \mathbb{Z} \middle| \frac{j(j+1)}{2} \leq y \right\} = \left\lfloor \frac{1}{2} \left(\sqrt{1+8y} - 1 \right) \right\rfloor \tag{31}$$

to determine j. Then m is given by $m = i - s(j)$.

For 3-D one can derive similar formulas by applying this principle but unfortunately the resulting equations become really unwieldy (as they are the solution to a general cubic equation). The derivation is outlined in appendix A but is not used in the implementation.

4.4 Computing $\tilde{R}$

When naively computing $\tilde{R} = D_1^{-1} R_1^{-1} R_2 D_2$ a lot of cancellation and overflow would occur due to the powers of ϵ contained in D_1^{-1} and D_2, thus negating the positive effects of the algorithm.
Define $\dot{R}, \tilde{R} \in \mathbb{R}^{N\times(K-N)}$:

$$\dot{R} := R_1^{-1} R_2$$
$$S := \dot{R} D_2 = R_1^{-1} R_2 D_2$$
$$\tilde{R} := D_1^{-1} S = D_1^{-1} R_1^{-1} R_2 D_2$$

$\dot{R}$ can be computed by backward substitution. S and R must be obtained analytically, as this is where the powers of ϵ enter. (Note that $(D_2)_{i,i} = d_{N+i}$):

$$S_{i,j} = \dot{R}_{i,j} d_{N+j}$$

$$\tilde{R}_{i,j} = \dot{R}_{i,j} \frac{1}{d_i} d_{N+j} \tag{32}$$

In 1-D this yields together with (18):

$$\tilde{R}_{i,j} = \dot{R}_{i,j} \frac{i!}{2\epsilon^{2i}} * \frac{2\epsilon^{2(N+j)}}{(N+j)!} = \dot{R}_{i,j} \frac{\epsilon^{2(N+j-i)}}{(N+j-i)!} \tag{33}$$

In 2-D, the equation for $\tilde{R}$ gets really unwieldy, so instead only $\frac{d_{N+j}}{d_i}$ (without the index shift applied) is considered:

$$\begin{aligned}\frac{d_{\frac{j_1(j_1+1)}{2}+m_1}}{d_{\frac{j_2(j_2+1)}{2}+m_2}} &= \frac{\epsilon^{2j_1} 2^{j_2-2m_2-1} \left(\frac{j_2+2m_2+p_{j_2}}{2}\right)! \left(\frac{j_2-2m_2-p_{j_2}}{2}\right)!}{2^{j_1-2m_1-1} \left(\frac{j_1+2m_1+p_{j_1}}{2}\right)! \left(\frac{j_1-2m_1-p_{j_1}}{2}\right)! \epsilon^{2j_2}} \\ &= \frac{\epsilon^{2(j_1-j_2)} \left(\frac{j_2+2m_2+p_{j_2}}{2}\right)! \left(\frac{j_2-2m_2-p_{j_2}}{2}\right)!}{2^{j_1-2m_1-j_2+2m_2} \left(\frac{j_1+2m_1+p_{j_1}}{2}\right)! \left(\frac{j_1-2m_1-p_{j_1}}{2}\right)!}\end{aligned} \tag{34}$$

In 3-D the equation (34) becomes

$$\frac{d_{j_1,m_1}}{d_{j_2,m_2}} = \frac{2^{3+p_{j_1}+4*m_1} \epsilon^{2j_1} \left(\frac{j_1+p_{j_1}+2m_1}{2}\right)! \left(\frac{j_2-p_{j_2}-2m_2}{2}\right)! (j_2+1+p_{j_2}+2m_2)!}{2^{3+p_{j_2}+4m_2} \epsilon^{2j_2} \left(\frac{j_2+p_{j_2}+2m_2}{2}\right)! \left(\frac{j_1-p_{j_1}-2m_1}{2}\right)! (j_1+1+p_{j_1}+2m_1)!}$$

$$\begin{aligned}\frac{d_{j_1,m_1}}{d_{j_2,m_2}} &= 2^{p_{j_1}-p_{j_2}+4*(m_1-m_2)} \epsilon^{2(j_1-j_2)} \\ &\quad \frac{\left(\frac{j_1+p_{j_1}+2m_1}{2}\right)! \left(\frac{j_2-p_{j_2}-2m_2}{2}\right)! (j_2+1+p_{j_2}+2m_2)!}{\left(\frac{j_2+p_{j_2}+2m_2}{2}\right)! \left(\frac{j_1-p_{j_1}-2m_1}{2}\right)! (j_1+1+p_{j_1}+2m_1)!}\end{aligned} \tag{35}$$

4.4.1 Computing $\tilde{R}$ for big K

When calculating $\tilde{R}$ for large K, a lot of cancellation occurs. In the following, a way of computing $\tilde{R}$ even for large K is described.
Consider terms like

$$a^{(i)} := \prod_{k=1}^{i} a(k)$$

For instance a^b can be written as:

$$\prod_{k=1}^{b} a$$

And $a!$ would be:

$$\prod_{k=1}^{a} k$$

A product $a_1^{(i_1)} * a_2^{(i_2)} * \dots a_n^{(i_n)}$ can then be calculated as

$$\prod_{m=1}^{n} a_m^{(i_m)} = \prod_{k=1}^{\max\{i_1,\dots,i_n\}} \prod_{m=1}^{n} \begin{cases} a_m(k) & k \leq i_m \\ 1 & \text{else.} \end{cases} \tag{36}$$

By using (36) computing $\tilde{R}$ becomes feasible even for large K (although it might be a bit slower than the naive approach).

4.5 Determining j_{max}

The authors of [10] described a way of determining where to cut off the expansions (17), (19), (21):
The basic idea is to choose j_{max} so that the scaling coefficients d_i get truncated once the ratio formed by D (cf. (32)) exceeds machine precision `eps`:

$$\frac{\max\limits_{i \geq K} D_{i,i}}{\min\limits_{0 \leq i < N} D_{i,i}} < \texttt{eps} \tag{37}$$

If $\epsilon \leq 1$ in 1-D, or $\epsilon \leq \sqrt{2}$ in 2-D and 3-D, d_i is monotonically falling, thus the denominator becomes $D_{N,N}$. However, if this is not the case, a formula is given by:

$$\min_{0 \leq i < N} D_{i,i} = \begin{cases} \min(1, d_{j_N}) & \text{in 1-D} \\ \min(2, d_{j_N,0}) & \text{in 2-D} \\ \min(8, d_{j_N,0}) & \text{in 3-D} \end{cases}$$

Note that this estimation from [10, Appendix B.2] is a bit conservative. D is indexed from 0, but still $d_0 = 2$ in 1-D, for instance.
In two and three dimensions one has to consider the biggest scaling coefficient within each block of D as well[8]. For $\epsilon \leq 1$ the blocks are monotonically falling,[9]

[8] The formulation "blocks" may be a bit misleading as D is still a diagonal matrix. "block" refers to the part of the main diagonal that corresponds to one summand of the first sum in the expansion.

[9] Note that this again differs from [10, Appendix B.2], because j is indexed from 0, and $p_0 = 1$

and simply $\min_{0 \le i \le N} D_{i,i} = D_{N,N}$. Otherwise, [10] gives approximations by

$$\max_{0 \le m \le \frac{j-p_j}{2}} d_{j,m} \approx \begin{cases} e^{0.223j+0.212-0.657p_j} d_{j,0} & \text{in 2-D} \\ e^{0.223j-0.012-0.649p_j} d_{j,0} & \text{in 3-D} \end{cases}$$

Now one can simply increase j_{max} until the ratio in (37) exceeds machine precision to get j_{max}.

4.6 Parallel QR factorization

For a matrix $A \in \mathbb{R}^{m \times n}$ there exists some $Q \in \mathbb{R}^{m \times m}$ and some $R \in \mathbb{R}^{m \times n}$, so that

$$A = QR$$

with Q being orthogonal (that is $Q^T Q = I$) and R being upper triangular.
For computing a QR factorization there are three algorithms (and variations thereof) that are widely used: Gram-Schmidt, Householder transformations and Givens rotations. For the implementation the Givens rotations algorithm was used. It uses special orthogonal matrices (*Givens rotations*) that are defined by:

$$G_{i,j}(c,s) := \left(\begin{array}{ccc|c|ccc|c|ccc} 1 & & & 0 & & & & 0 & & & \\ & \ddots & & \vdots & & & & \vdots & & & \\ & & 1 & 0 & & & & 0 & & & \\ \hline 0 & \dots & 0 & c & 0 & \dots & 0 & s & 0 & \dots & 0 \\ \hline & & & 0 & 1 & & & 0 & & & \\ & & & \vdots & & \ddots & & \vdots & & & \\ & & & 0 & & & 1 & 0 & & & \\ \hline 0 & \dots & 0 & -s & 0 & \dots & 0 & c & 0 & \dots & 0 \\ \hline & & & 0 & & & & 0 & 1 & & \\ & & & \vdots & & & & \vdots & & \ddots & \\ & & & 0 & & & & 0 & & & 1 \end{array} \right) \begin{array}{l} \\ \\ \\ \longleftarrow \text{Row } i \\ \\ \\ \\ \longleftarrow \text{Row } j \\ \\ \\ \\ \end{array} \tag{38}$$

$$\begin{array}{cc} \uparrow & \uparrow \\ \text{Column } i & \text{Column } j \end{array}$$

The algorithm is outlined in algorithm 1. The transformed A becomes R and the product of all $G_{i,j}$ is Q.

```
for j = 1, ..., n do
    for i = j + 1, ..., m do
        if a_{i,j} ≠ 0 then
            r_{i,j} = sqrt(a_{j,j}^2 + a_{i,j}^2)
            c_{i,j} = a_{j,j} / r_{i,j}
            s_{i,j} = a_{i,j} / r_{i,j}
            A ← G_{j,i}(c_{i,j}, s_{i,j}) · A
        end
    end
end
```

Algorithm 1: QR factorization with Givens rotations

Note that in every iteration of the inner loop the entry $a_{i,j}$ is set to 0 (a proof and more details can be found in [5]). For parallelization, I have to assume *distributed memory*, which makes transferring data rather expensive. This algorithm leaves some room for parallelization, which becomes evident when looking at the data dependencies in figure 12. From this scheme one can derive a strict weak partial ordering (two elements can be ordered the same, but still be different) for the way the loops are iterated:

$$\begin{pmatrix} \cdot & & & & & \\ 1 & \cdot & & & & \\ 2 & 3 & \cdot & & & \\ 3 & 4 & 5 & \cdot & & \\ 4 & 5 & 6 & 7 & \cdot & \\ 5 & 6 & 7 & 8 & 9 & \cdot \end{pmatrix} \tag{39}$$

Every iteration order that preserves the order in (39) is valid. For parallelizing the algorithm there are two options. The first one shall be called *row-wise* and assigns each process one or more rows to set to 0. The second one shall be called *column-wise* and assigns each process one or more columns to set to 0. When looking at figure 12, one can see that in a row-wise parallel algorithm, there would be two rows transferred after each iteration of the inner loop (when setting $a_{i,j}$ to 0, rows i and j have to be transferred to the next rank), whereas in a columns-wise parallel layout, only one row has to be transferred to the next rank (for entry $a_{i,j}$ that is row j). Furthermore, the row that stays within the process is always the same row for each process, so for medium size matrices the memory might stay "hot" (it is cached). However when a process computes multiple columns, iterating row-wise is preferred, as this yields rows to the next process earlier, thus speeding up the pipeline. This is summarized in figure 13.

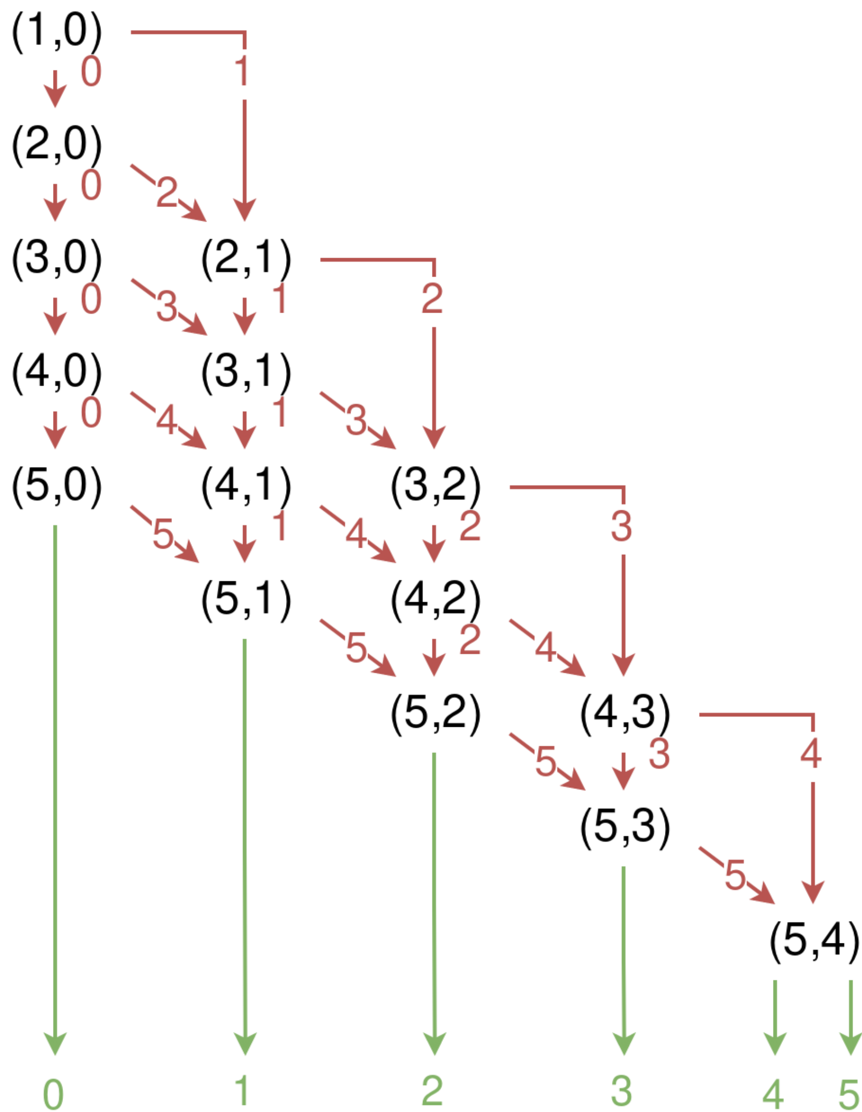

Figure 12: **Data dependencies in givens rotation algorithm**
Each tuple (i, j) indicates one iteration of the inner loop. Red arrows indicate where rows are to be transferred. Green arrows indicate where rows are finished. From these data dependencies there is already some inherent constraint visible regarding the execution order, as each iteration first needs to receive the respective rows. So for instance (2, 1) must be executed after (2, 0), but can be executed concurrently with (3, 0).

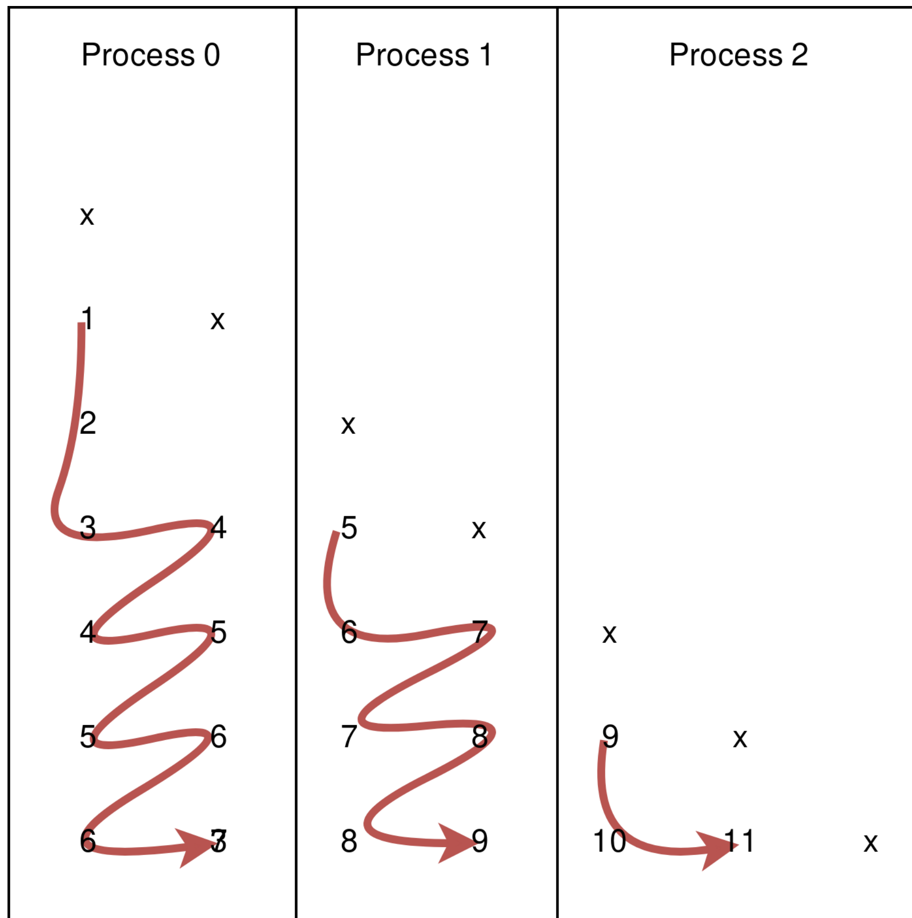

Figure 13: **A hybrid approach for givens iteration order**
This approach combines row-wise traversal with column-wise traversal. Each process "possesses" some subset of columns that is traversed row-wise. This ensures that data dependencies of the next process are fulfilled as early as possible. The communication for each process is bounded by $n * m$, where n denotes the number of rows and m denotes the number of columns. The overall communication is bounded by $\frac{n \cdot k \cdot m}{2}$, so there is definitely some communication overhead present.

5 Results

When investigating the results of RBF-QR, the question arises, what the algorithm should be compared to. One choice would be the unmodified RBF approach (In [10] it is referred to as *RBF-Direct*). However, there is a lightweight extension of RBF-Direct, which is RBF-Interpolation with an added polynomial. It has already been investigated and compared to RBF-Direct in [14] and it is also the way RBF-Interpolation is done in preCICE. Therefore this will be the algorithm that RBF-QR will be compared to.

5.1 Choosing the right ϵ

When choosing the "right" ϵ, one has to consider the size of the mesh too. This is why the shape parameter is sometimes also expressed in terms of nodes that are within the carrier of the basis function (c.f. [14]). For a given cutoff value - e.g. 10^{-9} - one can get ϵ from the number of nodes m within the carrier of the basis function by using

$$\epsilon = \frac{\sqrt{-\ln y_{min}}}{m \cdot h_{max}} \tag{40}$$

for some cut-off value y_{min} and the maximum distance between two adjacent centers being given by h_{max}.
For the RBF-QR algorithm the mesh has to be rescaled to lie within the unit sphere in d dimensions anyway, so scaling the shape parameter is less important. For comparison of RBF-QR with RBF-interpolation with polynomials, I will always use a mesh within the unit sphere. For a uniform mesh one can derive from (40):

$$\epsilon = \frac{(N-1)\sqrt{-\ln(y_{min})}}{m \cdot 2} \tag{41}$$

for N centers. For other node layouts, this formula is applicable as well, if I accept that instead of defining ϵ in terms of h_{max}, I will instead define it in terms of the maximum distance between centers.
There is a "minimal" shape parameter ϵ_{min} so that every basis function with $\epsilon \leq \epsilon_{min}$ is global on the input mesh. A formula for ϵ_{min} can be derived from (40) and is given by

$$\epsilon_{min}(y_{min}) = \frac{\sqrt{-\ln(y_{min})}}{2}$$

With $y_{min} = 10^{-9}$ for instance, it yields $\epsilon_{min} \approx 2.2761$. This is important, as one would expect no further improvement in accuracy below that value. However, this is not a good value to choose for RBF-QR, as discussed in section 5.3.1.

5.2 Chebyshev nodes

For solving PDE's in real-world applications, the choice of a proper mesh is crucial. Even though RBF interpolation is a mesh-free method, the node layout given by a solver does have some influence on the interpolation quality. In this paper, two kinds of node layouts are considered - equidistant nodes and

chebyshev nodes (used in a gauss-chebyshev mesh). In one dimension those are given by

$$x_k = \cos\left(\frac{2k+1}{2n}\pi\right), \; k = 1, \ldots, n$$

on the interval $[-1, 1]$. An example is given by Figure 14. For arbitrary intervals one can use any affine transformation on these nodes, and for higher dimensions, every combination of chebyshev nodes in every dimension will be considered, which is also shown in Figure 15.

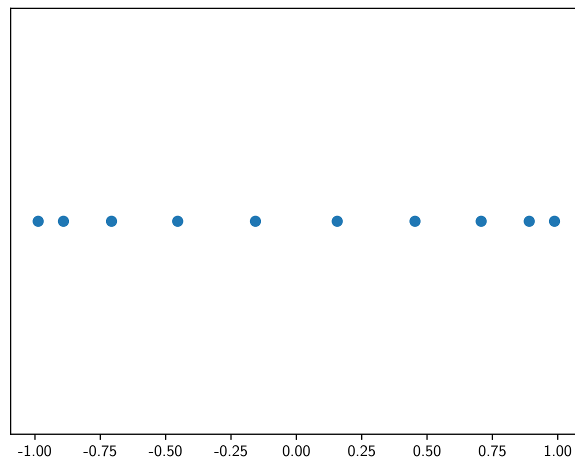

Figure 14: 1-D chebyshev nodes
One cell of order 10 is shown. Points are clustered towards the boundary.

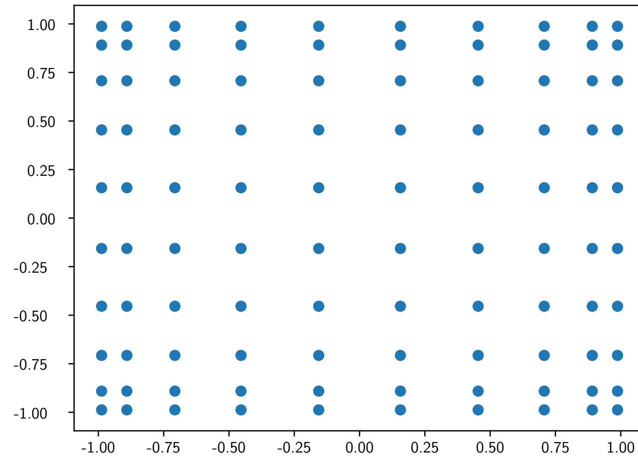

Figure 15: *2-D chebyshev nodes*
One cell of order 10 (in each dimension) is shown.

This node layout can usually be found within one *cell*, but a mesh consists of multiple cells (also called *elements* in FEM), so this node layout repeats within one mesh, as shown in figure 16.

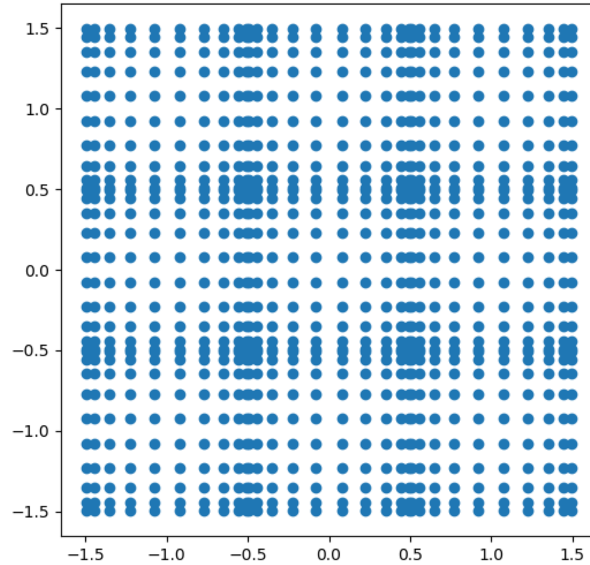

Figure 16: *2-D chebyshev nodes consisting of multiple elements*
In each dimension there are 3 cells with order 10 each. Overall, this makes 900 points split among 9 cells.

5.3 Condition and Accuracy

When the following sections refer to *condition*, for RBF interpolation with a polynomial the condition of the matrix of the linear system is meant (that is the matrix P). For RBF-QR, *condition* refers to the condition of A (from equation (27)). For interpolation with an added polynomial, the separated polynomial was used, which is also referred to as *Separated Poly* in the following sections.

5.3.1 Condition in the small-ϵ domain

When ϵ is made small, the basis functions become increasingly flat. Figure 17 shows the effect on the condition of A for the two algorithms.

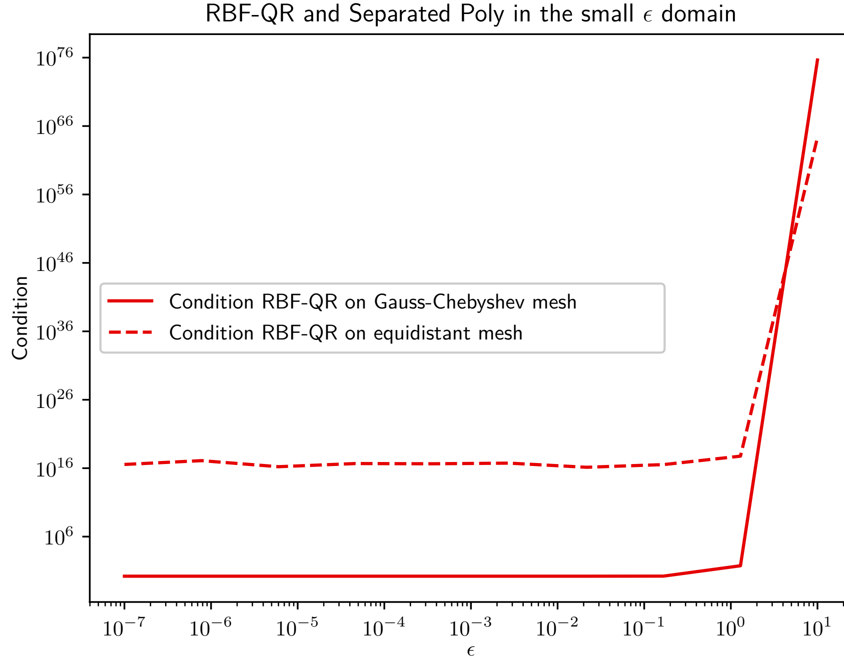

Figure 17: *Interpolation with varying* ϵ
Interpolating an arbitrary function on two different 1D-meshes with $N = 100$ nodes each.

One can make several observations from this Figure. Firstly, the condition stays invariant within $\epsilon < 1$. Interestingly, the condition deteriorates terribly when $\epsilon > 1$. This is not very surprising, because I previously assumed at several points that ϵ is rather small. For instance, when reconsidering figure 6, it can be seen that the factor (13) can no longer be assumed to be negligible. Secondly, the condition is rather bad if interpolation is done on equidistant nodes, but is almost perfect on gauss-chebyshev nodes. This observation can also be made from figure 18 and figure 19, where an explanation is given. For interpolation with RBF-QR $\epsilon = 10^{-5}$ is chosen, which ensures that the condition of RBF-QR does not worsen because of ϵ. For RBF interpolation with polynomials, $m = 5$ is chosen. This value yields a good trade-off between condition and accuracy, which has been shown in [14].

5.3.2 Condition and accuracy in one dimension

For comparison, the function

$$f(x) = e^{-|x-3|^2} + 2 \tag{42}$$

will be used.
The alternative algorithm will be RBF-interpolation with an added polynomial. The resulting linear system is solved by using a direct solver (LU). However,

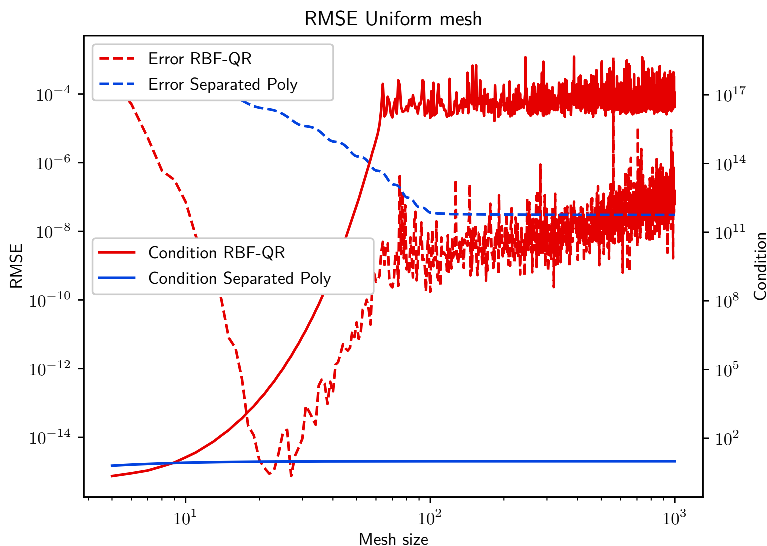

Figure 18: **Accuracy and condition of RBF-QR and RBF with a polynomial on an equidistant mesh**

For the separated polynomial RBF interpolation $m = 5$ was chosen, thus condition stays invariant with the mesh size. The condition of RBF-QR deteriorates very quickly with increasing mesh size and the error of RBF-QR also deteriorates when $N > 16$, and RBF-QR becomes even worse than RBF with polynomials for big values of N.

in real world applications one would very likely apply some preconditioner to the linear system first. Choosing such a preconditioner is a delicate problem. For interpolation on a uniform mesh, condition and accuracy can be seen in Figure 18. RBF-QR performs rather badly on big meshes, whereas the RBF algorithm accuracy stays invariant from $N = 100$ onwards. This is due to the fact that the basis function is scaled in a way where a fixed number of vertices is included for each basis function (c.f. section 5.1). In Figure 19 the same principle is applied for a gauss-chebyshev-mesh. (here $m = 5$ means that for RBF with polynomials *at least* 5 nodes are included within each basis function. This principle will again be applied in the 2-D case). Interestingly, the results are very different from Figure 18: RBF-QR accuracy improves very quickly until machine precision (where no further improvement is possible) and the condition stays very low even for big mesh sizes. The algorithm works almost perfectly. On the other hand, for the RBF algorithm with a polynomial, the condition number increases with the mesh size and accuracy only improves at a rather slow rate. The reason for the difference between Figure 18 and Figure 19 can be seen when looking at the new basis for 1-D. When ϵ is chosen to be small, the new basis functions are close to chebyshev polynomials. This means that in 1-D the new basis performs polynomial interpolation with chebyshev polynomials on gauss-chebyshev points. This has been seen to yield very good results in many previous works.

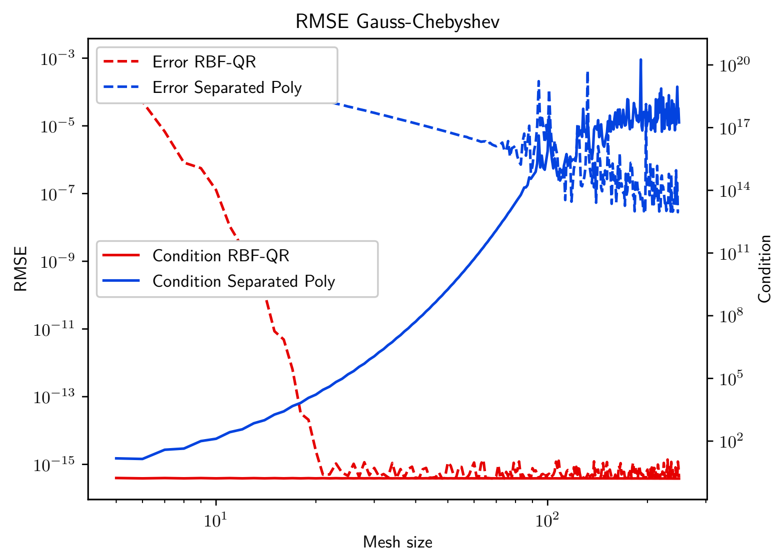

Figure 19: **Accuracy and condition of RBF-QR and RBF with a polynomial on a gauss-chebyshev-mesh**
For RBF-QR, condition and accuracy are almost perfect. The condition for RBF with a polynomial does not stay invariant with N, because each basis function must *at least* include 5 points. Thus, towards the boundary each basis function includes more than 5 points and the condition of the system worsens.

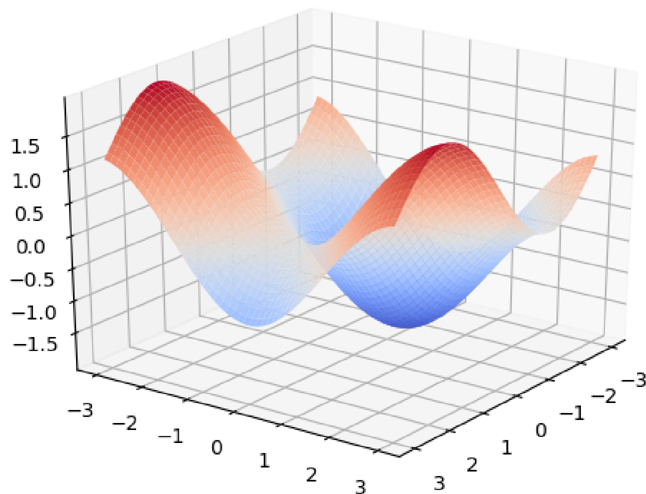

Figure 20: **The benchmark function for 2-D**

5.3.3 Condition and accuracy in two dimensions

In 2-D, a different function is used for comparison. It is given by

$$f(x, y) = \sin(x) - \cos(y) \tag{43}$$

and can be found in Figure 20. This function is again evaluated on an regular (equidistant) mesh first. Condition and accuracy for interpolation on a uniform mesh are shown in Figure 21. From this figure multiple observations can be made. The accuracy of RBF-QR improves at a steady pace when increasing the mesh size (Note that a mesh size of 50 in one dimension means having 2500 nodes). The condition stays invariant for RBF-QR at a high value which interestingly does not affect accuracy too much (considering that a direct (partial pivoting LU) solver was used). The results for RBF interpolation with an added polynomial are less surprising. The condition number increases with the number of points until 10^{15}, and accuracy improves at the same time. From a mesh size of 20 in one dimension onwards, neither condition nor accuracy change much anymore. This is because of the way ϵ is scaled. There are always 5 nodes included in one dimension.[10] An analogous investigation was done on gauss-chebyshev points in 2-D. The results are shown in Figure 22. For RBF-QR, not much has changed. The error improves by roughly one order of magnitude, the condition stays about the same. For interpolation with an added polynomial a

[10]The actual number of points contained within the (circular) carrier of the basis function varies, but can be safely assumed to be ≤ 25.

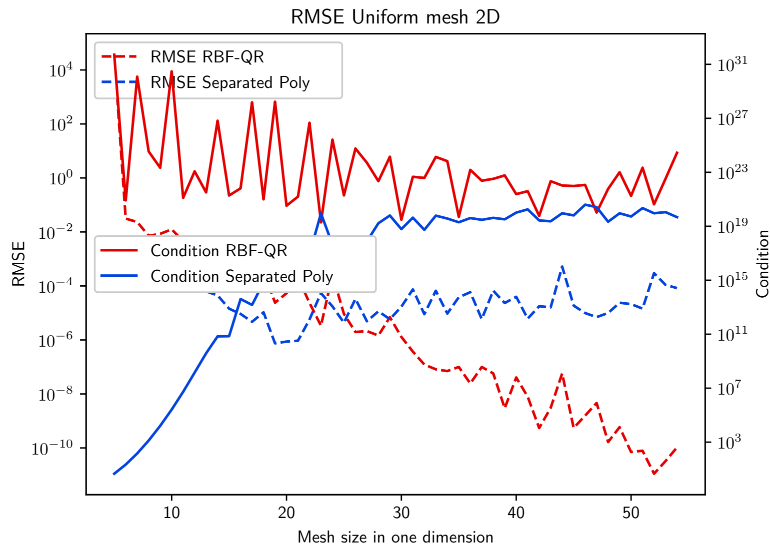

Figure 21: **Accuracy and condition in 2-D on a uniform mesh**
RBF-QR improves accuracy with the mesh size, even though the condition of A is rather bad. For RBF with an added polynomial, condition and accuracy are staying more or less invariant with increasing mesh size.

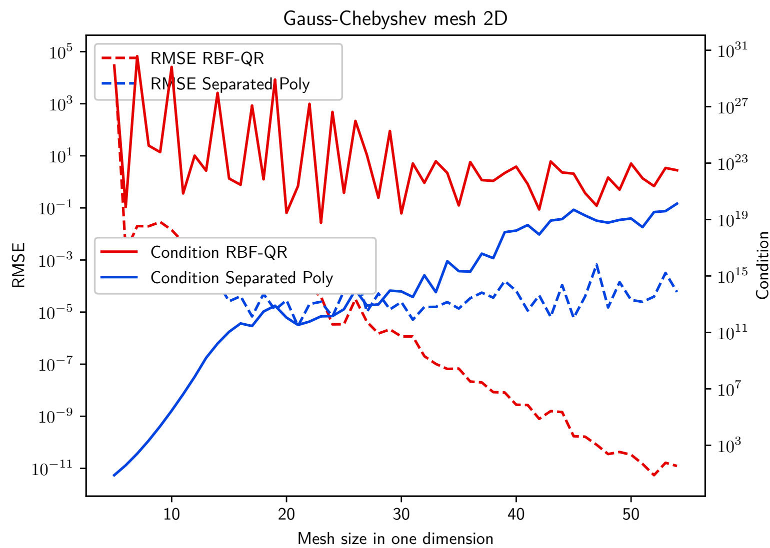

Figure 22: **Accuracy and condition in 2-D on a gauss-chebyshev mesh**
The results are quite similar to Figure 21, but this time the accuracy of RBF-QR is a bit better and the condition of the RBF interpolation with an added polynomial worsens with the mesh size.

similar effect as in 1-D occurs. When increasing the number of nodes the condition worsens, because towards the boundary, more points are included within the carrier of each basis function. However accuracy does not increase from this effect for some reason.

5.4 Performance

Performance is an important factor for a mapping algorithm, and in the following sections the performance of RBF-QR will be investigated. However, there are some severe restrictions to the overall efficiency of the algorithm, which will be discussed in the following as well. The performance of RBF-QR will be compared to RBF interpolation with an added polynomial. For the latter, the preCICE implementation will be used, with all compiler optimizations enabled. However, there were some restrictions as well.

5.4.1 Scaling to multiple nodes

Before considering the time needed to compute RBF-QR for different mesh sizes, it might be interesting how well the algorithm scales to different levels of parallelism. For this, a fixed 2D-mesh of with 20 nodes in each dimension (400 nodes overall) was used. The time was then measured for the two interesting parts of the algorithm: The QR factorization (as outlined in section 4.6) and the solver for the system $A\lambda = f$. In Figure 23 the results by using PETSc can be seen. There is no real performance gain from using more processors, which might be because dense linear algebra incurs a significant communication overhead. Also note that PETSc does not prioritize parallel dense linear algebra and usually advocates the use of Elemental for the solution of these systems. The performance of using Elemental can be seen in Figure 24. Elemental seems to perform worse on many ranks, which might hint at a bug within Elemental. Since neither of these libraries benefit very much from parallelism, the following sections will only investigate sequential performance.

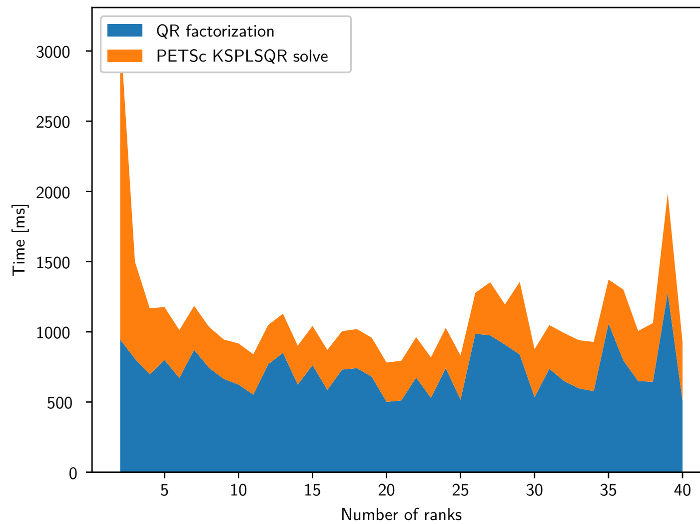

Figure 23: **Performance of RBF when scaled to different levels of parallelism, using PETSc**
A fixed problem size of $20 * 20 = 400$ nodes was used. The performance does not change much when increasing the number of ranks (therefore increasing parallelism).

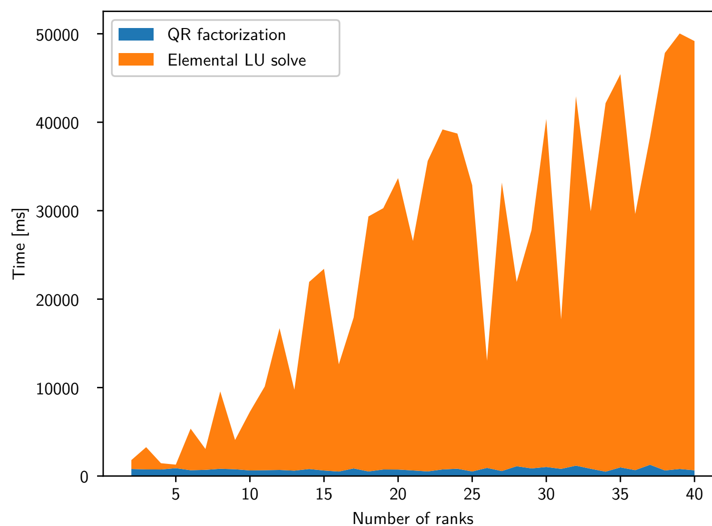

Figure 24: **Performance of RBF when scaled to different levels of parallelism, using Elemental**
A fixed problem size of $20 * 20 = 400$ nodes was used. The performance actually decreases severely when increasing the number of ranks.

5.4.2 Varying shape parameter

Changing the shape parameter ϵ has some performance implications for the traditional RBF algorithms, because the system to be solved is a sparse matrix. Decreasing the shape parameter means having more non-zero entries in the system matrix and therefore increasing computational cost. For RBF interpolation with added polynomial this can be seen from Figure 25. There, a fixed mesh of size $10 * 10$ is used and the basis functions are made increasingly flat. Once all basis functions are global, performance does not deteriorate any further. At this point the matrix P is no longer sparse, because all entries are now non-zero. For RBF-QR the shape parameter does not play a role, as dense linear algebra is used anyway.

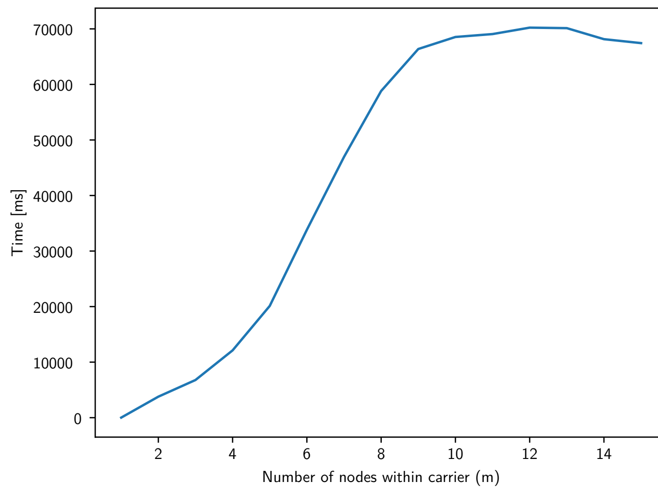

Figure 25: **Performance of RBF interpolation with added polynomial for different values of m**
A fixed problem size of $10 * 10 = 100$ nodes was used. The number of points included within the carrier of a basis function in one dimension (m) is used for the x-axis. The time increases until $m = 10$, where no further increase is visible.

5.4.3 Variable mesh size

The mesh size obviously has some effect on the runtime as well. Increasing the number of points means increasing the size of the matrices involved and thus increases runtime of the algorithm. Unfortunately the implementation of RBF-QR and the implementation of RBF interpolation with an added polynomial use very different data structures. This makes comparing these two algorithms rather hard. In particular, assembling the matrix turned out to be rather slow for RBF interpolation with an added polynomial, whereas RBF-QR takes more time to actually solve the system. An investigation how these effects might change for very big meshes could be interesting for future work. For this work i will therefore restrict measurements to RBF-QR. In Figure 26 the time needed

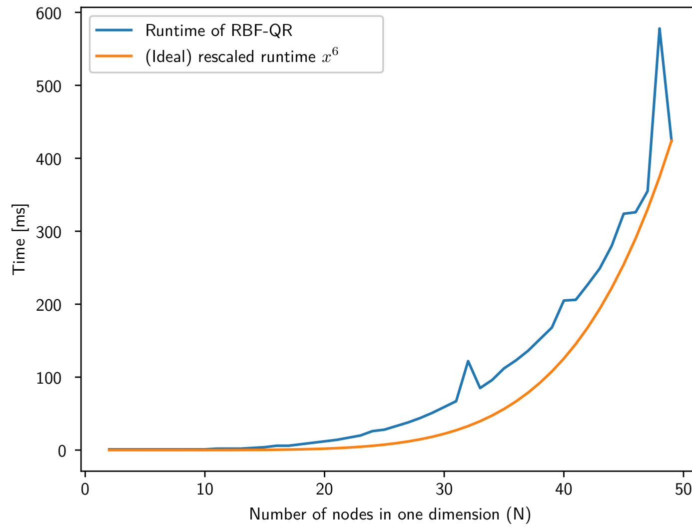

Figure 26: **Performance of RBF-QR for different mesh sizes**
As the mesh size increases, runtime increases as well. The (theoretical) runtime of $\mathcal{O}(x^6)$ is shown as well and rescaled to match the last value of RBF-QR.

to perform the offline phase is depicted. The theoretical runtime of the algorithm in would be cubic in the mesh size and thus one would expect $\mathcal{O}(x^6)$ time for a mesh with x points in each dimension. From Figure 26 one can see that the algorithm almost matches this time.

Appendices

A 3d index transformation

Here is a derivation of the index shift outlined in 4.3 in the 3D-case. This time the formula is given by

$$s(n) = \sum_{a=0}^{n} \sum_{b=0}^{\frac{j-p_j}{2}} \sum_{c=-(2b+p_a)}^{2b+p_a} 1 \tag{44}$$

$$= \sum_{a=0}^{n} \sum_{b=0}^{\frac{j-p_j}{2}} (4b + 2p_a + 1)$$

$$= \sum_{a=0}^{n} \sum_{b=0}^{\lfloor \frac{a}{2} \rfloor} (4b + 2p_a + 1)$$

Splitting into even and odd a's and halving them.

$$= \sum_{a=0}^{\lfloor \frac{n}{2} \rfloor} \sum_{b=0}^{a} (4b + 1) + \sum_{a=0}^{\lfloor \frac{n}{2} \rfloor} \sum_{b=0}^{a-1} (4b + 3)$$

$$= \sum_{a=0}^{\lfloor \frac{n}{2} \rfloor} \left((a+1) + 4\frac{a(a+1)}{2} + 3a + 4\frac{a(a-1)}{2} \right)$$

$$= \sum_{a=0}^{\lfloor \frac{n}{2} \rfloor} (4a^2 + 4a + 1)$$

Now look at

$$t(k) := \sum_{a=0}^{k} (4a^2 + 4a + 1) \tag{45}$$

Using square pyramidal number for $4a^2$ and gaussian sum formula for $4a$:

$$= \frac{1}{3}(k+1)(2k+1)(2k+3)$$

Getting t^{-1} means having to solve

$$y = \frac{1}{3}(k+1)(2k+1)(2k+3)$$

For $y > 1$ the expression

$$\frac{1}{3}(k+1)(2k+1)(2k+3) - y$$

only has one (real) root. It is given by

$$k = \frac{1}{2} \left(\frac{\sqrt[3]{\sqrt{3}\sqrt{243y^2 - 1} + 27y}}{\sqrt[3]{9}} + \frac{1}{\sqrt[3]{3}\sqrt[3]{\sqrt{3}\sqrt{243y^2 - 1} + 27y}} - 2 \right) \tag{46}$$

From here one could get $s^{-1}(n)$, but the resulting equation would be even more unwieldy, as there would be an additional distinction between even and odd n.

References

[1] Brad JC Baxter and Simon Hubbert. Radial basis functions for the sphere. In *Recent progress in multivariate approximation*, pages 33–47. Springer, 2001.

[2] Richard K Beatson, Jon B Cherrie, and Cameron T Mouat. Fast fitting of radial basis functions: Methods based on preconditioned gmres iteration. *Advances in Computational Mathematics*, 11(2-3):253–270, 1999.

[3] Martin D Buhmann. Radial basis functions. *Acta numerica*, 9:1–38, 2000.

[4] Hans-Joachim Bungartz, Florian Lindner, Bernhard Gatzhammer, Miriam Mehl, Klaudius Scheufele, Alexander Shukaev, and Benjamin Uekermann. preCICE – a fully parallel library for multi-physics surface coupling. *Computers and Fluids*, 141:250–258, 2016. Advances in Fluid-Structure Interaction.

[5] Wolfgang Dahmen and Arnold Reusken. *Numerik für Ingenieure und Naturwissenschaftler*. Springer-Verlag, 2006.

[6] Simone Deparis, Davide Forti, and Alfio Quarteroni. A rescaled localized radial basis function interpolation on non-cartesian and nonconforming grids. *SIAM Journal on Scientific Computing*, 36(6):A2745–A2762, 2014.

[7] Tobin A Driscoll and Bengt Fornberg. Interpolation in the limit of increasingly flat radial basis functions. *Computers & Mathematics with Applications*, 43(3-5):413–422, 2002.

[8] Gregory E Fasshauer and Jack G Zhang. On choosing "optimal" shape parameters for rbf approximation. *Numerical Algorithms*, 45(1-4):345–368, 2007.

[9] Gregory E Fasshauer and Jack G Zhang. Preconditioning of radial basis function interpolation systems via accelerated iterated approximate moving least squares approximation. In *Progress on Meshless Methods*, pages 57–75. Springer, 2009.

[10] Bengt Fornberg, Elisabeth Larsson, and Natasha Flyer. Stable computations with gaussian radial basis functions. *SIAM Journal on Scientific Computing*, 33(2):869–892, 2011.

[11] Bengt Fornberg and Cécile Piret. On choosing a radial basis function and a shape parameter when solving a convective pde on a sphere. *Journal of Computational Physics*, 227(5):2758–2780, 2008.

[12] Message P Forum. Mpi: A message-passing interface standard. Technical report, Knoxville, TN, USA, 1994.

[13] Elisabeth Larsson and Bengt Fornberg. Theoretical and computational aspects of multivariate interpolation with increasingly flat radial basis functions. *Computers & Mathematics with Applications*, 49(1):103–130, 2005.

[14] Florian Lindner, Miriam Mehl, and Benjamin Uekermann. Radial basis function interpolation for black-box multi-physics simulations. In *VII International Conference on Computational Methods for Coupled Problems in Science and Engineering*, pages 1–12, 2017.

[15] Robert Schaback. Multivariate interpolation by polynomials and radial basis functions. *Constructive Approximation*, 21(3):293–317, 2005.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature