Approximation by harmonic polynomials

Term project

Andrey Petukhov

Advisors: Prof. Dr. Ralf Hiptmair
Andrea Moiola

Seminar für Angewandte Mathematik
Eidgenössische Techische Hohschule Zürich

February 2010

Abstract

In this work we studied the approximation of the fundamental solution of the Laplace operator in two-dimensional space $u = \log(|x|)$ by harmonic polynomials. We analyzed the best approximation in the semi-ring with fixed outer radius and inner radius ? tending to zero. We observed exponential convergence in the degree of polynomials used for approximation. However, with inner radius tending to zero the rate of convergence gets worse. The convergence still remains exponential but with some exponent for the degree of polynomials.

Contents

Introduction

In most practical problems exact analytic computations are impossible even if the input data is given analytically by some function. In this case one has to think of numerical computations and various methods could be used for it depending on the type of the problem. Numerical computations always raise the problem of approximation as they are a-priori non-exact. The main aim here is to determine how close the approximate solution will be to the exact one (of course supposing it exists and is unique under some conditions). Methods used for numerical computations often require substituting the analytically given functions of the input data by their approximations and one gets non-exactness even at this step. To know how well the function is approximated is extremely important as it further determines one of the limits for the accuracy of the final result. Thus the problem of function approximation itself turns out to be an important one and it should be treated carefully.

Various classes of functions are used for the approximation. One of the best ideas is to approximate the given function by a finite number of terms of its Fourier series in some basis. But to do it properly one should choose such basis with respect to a given domain where the function should be approximated. Sometimes it is rather difficult to do taking into consideration the fact that the basis should be analytically orthogonal. Not for every domain it is trivial to do. But nevertheless it turns out that the convergence can still be very good even if the basis is non-orthogonal (with respect to the given domain). One of such examples is the set of harmonic polynomials that are the system of solutions of the Laplace operator in two- or three-dimensional space. Such system is orthogonal if we choose the appropriate domain (such as a disk in 2D and a ball in 3D). But if we have a problem in another domain, for example a half-disk or half-ball, the system is no more orthogonal. But the convergence in this case still remains very good. How good is it really? As far as we know, there is no full and complete proof for the rate of convergence for such an approximation (especially in two-dimensional case). However some numerical experiments show that the convergence is very fast and is exponential in the degree of polynomial. To check it and perhaps get some more ideas about the general case we do within this work such an experiment. We consider here the following problem. We consider the function $u = \log|x|$ in the two-dimensional domain Ω: semi-ring lying in the right half-plane, with fixed outer radius $R = 1$ and inner radius ε (Fig.1). We approximate this function by a finite number of elements of the series in terms of harmonic polynomials. The set of harmonic polynomials with the degree less than n in two-dimensional case is given by $H_n =< r^j e^{\pm j\varphi} >_{j=0}^{n}$,

where (r, φ) are polar coordinates (the system of harmonic polynomials is of course linearly independent and forms a basis of the space H_n).

The function is analytic in the given domain but has singularity at zero (center of the semi-ring). This can somehow influence the rate of convergence as we decrease the inner radius ε, getting closer and closer to the singularity point. We will see how the rate of convergence really behaves for different values of ε.

In the first chapter we will give some preliminary analytic derivations and computations and finally come to the equivalent problem for the system of linear equations that was solved numerically. In the second chapter we discuss the numerical calculations taking into consideration the difficulties that give us this problem. In the third chapter we analyze the results of the numerical simulations and describe some problems that we had to cope with during the simulations.

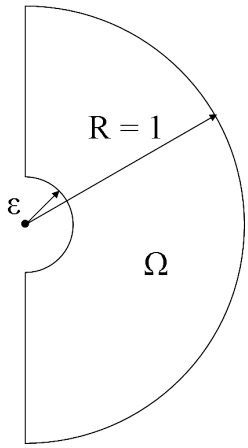

Fig.1. Computational domain.

Chapter 1

Preliminary derivations

1.1 Best approximation

We will look for the best approximation of the given function $u = \log|x|$ by harmonic polynomials from the set H_n. To get some estimates how close are really the original function and its approximation we should compute some norm of the difference between them. In principle we are free to choose the norm but it makes sense to choose the L^2– norm. In the mean-square sense (estimates in the L^2– norm) there exists a unique best approximation, so our problem is a-priori correct. Let p_n be some function, belonging to the approximation space H_n — the span of harmonic polynomials of all degrees up to n. Then the problem of obtaining the best approximation of a given function u in this space is formulated as a variational problem. We should find such function p_n in the approximation space that gives the minimum to the norm (or its square) of deviation from the function u:

$$f_n = ||u - p_n||^2_{L^2(\Omega)} \to \min\,. \tag{1}$$

In the space H_n we can consider the basis of harmonic polynomials $x_k = r^k e^{\pm k\varphi}, k = -n, ..., n$ as was in the definition of the space H_n, but we can also choose trigonometric form instead of exponentials: $a_i = r^i \sin i\varphi, b_j = r^j \cos j\varphi, i = 1, ..., n, j = 0, 1, ..., n$. We can simplify our problem taking into consideration the properties of the domain of approximation and the original function. The entire domain lies in the right half-plane and the original function is even (with respect to φ in polar coordinates). Therefore only coefficients at the even basis functions will be non-zero in the function that approximates the function u. Following this idea we choose the basis in the trigonometric form and we can say a-priori that all the coefficients at the sine functions will be zero and thus we can consider only cosine functions in the

basis of the approximation space. Since now we treat previously defined space H_n as a span of only even (in φ) harmonic polynomials in the trigonometric form. The best approximation of the function u of course belongs to this space due to the reasons given above.

The function $p_n \in H_n$ can be expanded with respect to the basis:

$$p_n = \sum_{i=0}^{n} \beta_i b_i. \tag{2}$$

Then we get from (1)

$$f_n = ||u - \sum_{i=0}^{n} \beta_i b_i||^2_{L^2(\Omega)} = (u,u)_{L^2(\Omega)} - 2\sum_{i=0}^{n} \beta_i (u, b_i)_{L^2(\Omega)} + \sum_{i=0}^{n}\sum_{j=0}^{n} \beta_i \beta_j (b_i, b_j)_{L^2(\Omega)}. \tag{3}$$

To find the minimum of this expression we differentiate it with respect to the coefficients $\beta_i, i = 0, ..., n$ and set all the derivatives to zero. We obtain the main system of equations for the coefficients that we will further solve numerically:

$$\sum_{k=0}^{n} \beta_k (b_j, b_k)_{L^2(\Omega)} = (u, b_j)_{L^2(\Omega)}, \qquad j = 0, ..., n. \tag{4}$$

It is a system of linear algebraic equations. We can also rewrite this system as a matrix equation

$$A\vec{\beta} = \vec{\mu} \tag{4a}$$

with unknown vector of coefficients $\vec{\beta} = (\beta_0, \beta_1, ..., \beta_n)^T$ and vector of the right-hand side $\vec{\mu} = \Big((u, b_0)_{L^2(\Omega)}, (u, b_1)_{L^2(\Omega)}, ..., (u, b_n)_{L^2(\Omega)}\Big)^T$ The matrix of this system is the mass matrix of the basis, i.e. the matrix of scalar products of the elements of the basis: $(A)_{ij} = (b_i, b_j)_{L^2(\Omega)}$. We should note that this matrix is symmetric and positive definite. These properties determine the method that will be used to obtain the solution of this system. One of the best ways to deal with the system with symmetric positive-definite matrix is to apply Cholesky decomposition. This method will be used in our case. Nevertheless there are still some difficulties with its implementation and we will speak about it in the next chapters. It is also necessary to rewrite the expression (3) for norm of the approximation error in the vector form

$$f_n = ||u - \sum_{i=0}^{n} \beta_i b_i||^2_{L^2(\Omega)} = ||u - \vec{\beta}\vec{b}(r,\varphi)||^2_{L^2(\Omega)} = (u,u)_{L^2(\Omega)} - 2\vec{\beta}^T\vec{\mu} + \vec{\beta}^T A\vec{\beta}, \tag{5}$$

where $\vec{b}(r, \varphi)$ is a vector-function, the elements of which are the basis functions. The matrix A and the vector $\vec{\mu}$ are the matrix and the right-hand

side of the system and we will compute their elements in the next section. The vector $\vec{\beta}$ is the vector of coefficients of the function that approximates the original one, with respect to the basis of harmonic polynomials. Using the system (3a) to compute the vector of coefficients we obtain the best approximation and the minimum possible approximation error. The only term left here is the term $(u,u)_{L^2(\Omega)}$. We will compute it analytically in the next section.

1.2 Analytic computations of coefficients

We must now compute all the coefficients of the system we obtained above. All of them are scalar products of some functions in the space $L^2(\Omega)$ and to compute them we are to compute integrals over the domain Ω. In general they can be computed numerically, but in our case it is possible to provide here complete analytic calculations. The main technique to be used here is integration by parts. We have to calculate integrals in the polar coordinates of two main types. Integrals of the first type look like the integral

$$\mu_j = \int_\Omega r^j \cos(j\varphi)\log(r) r \mathrm{d}r\mathrm{d}\varphi, \quad j = 0, 1, ..., n. \tag{6}$$

These integrals correspond to the L^2–scalar products of the original function u and the basis functions b_j and stand for the elements of the vector $\vec{\mu}$ of the right-hand side of our equation. Integrals of the second type are L^2–scalar products of the elements of the basis and stand for the elements of the matrix A of the equation:

$$(A)_{ij} = \int_\Omega r^j \cos(j\varphi) r^k \cos(k\varphi) r \mathrm{d}r\mathrm{d}\varphi, \quad j, k = 0, 1, ..., n. \tag{7}$$

In both cases the integrals in r and φ can be taken separately as the function under the integral can be presented as multiplication of two functions depending only on one of the two variables, and the domain of integration has the boundaries, which can be described by equations with only one of the two variables in polar coordinates. The equations of the boundaries are (see Fig.1) $r = \varepsilon, r = 1$ and $\varphi = -\pi/2, \varphi = \pi/2$.

Compute first the integrals of the type (6). The integrals in φ is computed easily:

$$\int_{-\pi/2}^{\pi/2} \cos(j\varphi)\mathrm{d}\varphi = \begin{cases} \pi, & \text{if } j = 0; \\ \frac{2}{j}(-1)^j, & \text{if } j \text{ is odd}; \\ 0, & \text{if } j \text{ is even}. \end{cases}$$

To compute the integrals in r we have to use integration by parts. Then the calculations are also trivial:

$$\int\limits_{\varepsilon}^{1} r^{j+1}\log r\mathrm{d}\varphi = -\frac{\varepsilon^{j+2}\log\varepsilon}{j+2} - \frac{1-\varepsilon^{j+2}}{(j+2)^2}.$$

Combining these two integrals together and introducing a new index k instead of j to take into consideration the fact that the integral vanishes when j is odd, we can write a final formula for the integral of the first type (see (6)):

$$\mu_j = \begin{cases} -\pi\left(\frac{\varepsilon^2\log\varepsilon}{2} - \frac{1-\varepsilon^2}{4}\right), & \text{for } j=0; \\ \frac{2\cdot(-1)^k}{2k-1}\left(\frac{\varepsilon^{2k+1}\log\varepsilon}{2k+1} + \frac{1-\varepsilon^{2k+1}}{(2k+1)^2}\right), & \text{for } k=1,2,...,\left[\frac{n+1}{2}\right], j=2k-1. \end{cases} \tag{8}$$

Now we compute the elements of the matrix A. Computing the integrals in r is trivial and we can immediately write the result:

$$\int\limits_{\varepsilon}^{1} r^{i+j+1}\mathrm{d}\varphi = \frac{1-\varepsilon^{i+j+2}}{i+j+2}.$$

To compute the integrals in φ we use some trigonometric formulas to transform the product of the cosine functions into the sum of cosine functions:

$$\cos i\varphi\cos j\varphi = \frac{1}{2}\left(\cos(i+j)\varphi + \cos(i-j)\varphi\right).$$

Then everything becomes simple and we get (for $i \neq j$)

$$\int\limits_{-\pi/2}^{\pi/2} \cos(i\varphi)\cos(j\varphi)\mathrm{d}\varphi = \frac{1}{i+j}\sin\left[(i+j)\frac{\pi}{2}\right] + \frac{1}{i-j}\sin\left[(i-j)\frac{\pi}{2}\right].$$

For $i=j$ we get, using the formula for the cosine of the double argument

$$\int\limits_{-\pi/2}^{\pi/2} \cos^2(i\varphi)\mathrm{d}\varphi = \begin{cases} \pi, & \text{for } i=0; \\ \frac{\pi}{2}, & \text{for } i\neq 0. \end{cases}$$

Combining these results together we get the final formula for the elements of the matrix A:

$$(A)_{ij} = \begin{cases} \pi\frac{1-\varepsilon^2}{2}, & \text{for } i=j=0; \\ \frac{\pi}{2}\frac{1-\varepsilon^{2i+2}}{2i+2}, & \text{for } i=j=1,2,...,n; \\ \frac{1-\varepsilon^{i+j+2}}{i+j+2}\left[\frac{1}{i+j}\sin\left[(i+j)\frac{\pi}{2}\right] + \frac{1}{i-j}\sin\left[(i-j)\frac{\pi}{2}\right]\right], & \text{for } i\neq j, i,j=1,2,...,n. \end{cases} \tag{9}$$

We should notice here that the basis we use is not orthogonal and thus the matrix A is not diagonal. This will give rise to some difficulties in numerical simulations and we will give more detail in the next chapter. Here we will only mention that non-orthogonality results from the particular choice of the domain Ω, which is a semi-ring (but not a ring for example). The same basis considered on the full ring becomes orthogonal. In the case of a semi-ring we have to somehow cope with the non-orthogonality of the basis.

Now all the coefficients are computed and we are ready to proceed to obtaining numerical solution of the system (4a). But before we do it we will compute the term $(u,u)_{L^2(\Omega)}$ left in the expression for the norm of the approximation error that we gave above. This term is represented by the following integral:

$$(u,u)_{L^2(\Omega)} = \int\limits_{-\pi/2}^{\pi/2} \mathrm{d}\varphi \int\limits_{\varepsilon}^{1} r\mathrm{d}r \log^2 r.$$

It can be easily computed analytically by taking integration by parts in r two times. We will finally get

$$(u,u)_{L^2(\Omega)} = -\pi\frac{\varepsilon^2}{2}(\log\varepsilon)^2 + \pi\frac{\varepsilon^2}{2}\log\varepsilon(\log\varepsilon) + \pi\frac{1-\varepsilon^2}{4}. \tag{10}$$

Now all preliminary computations are done and we can proceed to numerical simulation. It should be mentioned that all the computations till now were done analytically. This of course will help us to get rid of quadrature errors for example and will involve only evaluation errors that have the order of machine precision.

Chapter 2

Numerical experiment and programming

Proceeding to numerical computations we should recall which quantitative results are necessary for us. The main aim is to determine the rate of convergence of the best approximation with respect to the degree of polynomials (harmonic polynomials) used for approximation, for different values of ε. This means that for a given ε we have, first, to compute the coefficients of the best approximation with respect to the basis of harmonic polynomials, for different degrees n, that means we have to solve the main system (4a) with the matrix A and right-hand side vector $\vec{\mu}$ (see (8-9)) and obtain the vector of coefficients $\vec{\beta}$. Second, we are to compute the approximation errors using (5), (10) and also (8-9) for the matrix A and vector $\vec{\mu}$ (for various degrees of approximation polynomials). Thus we obtain, for a given ε, the table of values - the dependence of the approximation error on the degree of polynomial. Determining the character of this dependence we will determine the rate of convergence of approximation with respect to the degree of polynomial. This process should be repeated for different values of ε so that we could observe how the rate of convergence depends on that, how close we get to the singularity of the function (recall that ε is the radius of the small circle near the singularity of the original function).

The most difficult and important part here is to obtain the vector of coefficients $\vec{\beta}$. There are some problems that raise in these computations that are rather difficult to overcome. We will now discuss it in more detail.

2.1 Approaches to obtaining numerical solution

Computing the vector of coefficients in our case is, from the mathematical point of view, solving the system of linear algebraic equations (4) that has a matrix representation (4a). There exist a great number of methods to be applied for obtaining solution of a system of such a kind. The choice of the method is determined by the properties of the system, in particular of the matrix of the system.

We recall that in our case the matrix A in (4a) is the mass matrix of the chosen basis of harmonic polynomials. This implies that A is a symmetric positive-definite matrix (recall here, that the basis we have chosen is not orthogonal and thus the matrix is not diagonal). There is a special way to deal with such matrices that reduces the number of operations (in comparison with standard Gauss method for general matrices) and increases the speed of computations. This is a method of Cholesky decomposition. The matrix A is presented in the form $A = R^T R$, where R is an upper-triangular matrix and R^T is a lower-triangular matrix, respectively. The original problem (omitting the vector arrows) $A\beta = R^T R\beta = \mu$ is reduced to solving sequentially two systems: $R^T\xi = \mu$ and then $R\beta = \xi$. Both systems have triangular matrices and thus can be solved easily by obtaining the elements of the unknown vector sequentially.

At the moment everything seems rather simple, but in fact the situation is much more difficult. The problem we face when solving the system (4a) numerically by means of Cholesky decomposition is that it turns out to be unstable. The reason for that is that the matrix A becomes ill-conditioned with the increase of the degree of polynomial used for approximation. For example, when implementing the described algorithm in MATLAB, we got great instability as we approached the degree $n = 20 \sim 25$, depending on the value of ε. At this point the Cholesky decomposition could't be computed as the matrix (due to numerical effects) was not considered to be positive-definite any more. This comes exactly from the ill-conditionality of the matrix of the system. There are some ways to deal with this problem. The first possible approach is to introduce some regularization term. This will definitely dive us some solution but it is not the thing we are interested in. Regularization gives us one more source of error. As the degree of approximation polynomial increases, the matrix becomes more and more ill-conditioned and we have to use greater regularization that gives us, of course, greater regularization error. Therefore in this case we cannot determine the real rate of convergence of the approximation. This means that we should avoid regularization in our

computations.

To introduce the next approach we should understand what really happens when at some point the Cholesky decomposition cannot be computed. In fact the reason for that is that we somehow reach the limit of precision, the machine precision (double precision in MATLAB implementation). The values that are computed have at this point the same order as the as the errors of computations. This leads to the situation when at some point square roots that should be computed within the Cholesky decomposition cannot be calculated (in real numbers) as the program has to take a square root of a negative number. The possible way to deal with such a problem is to increase the precision of our computations considering a larger number of decimal digits. This gives the idea of implementation of high-precision library. To realize this idea we came from MATLAB implementation to C++ implementation and we applied the ARPREC library (ARbitrary PRECision Computation Package), designed by a group of scientists led by David H. Bailey from Lawrence Berkeley National Laboratory (see [3]). This is a software package for performing arbitrary precision arithmetic. Application of this library gives us a possibility to perform computations with any preassigned precision (up to thousand digits, instead of 16 digits in double precision computations). At a first glance this will completely solve all our problems but unfortunately we faced new problems with application of this library. We will described them in more detail in the next section. We now give some general results of our computations and and we will give more concrete results on rates of convergence in different cases in the next chapter.

2.2 Algorithm implementation

We implemented the algorithm described above in this chapter in two programming media - in MATLAB with double precision and then in C++ with double and multiple precision. All the codes are presented in the appendix. The program codes for double precision computations look exactly the same for MATLAB and C++ implementation, the only difference is in the algorithm used for computing Cholesky decomposition. In MATLAB we use the "backslash" for solving linear system. This operation tests the matrix for being positive-definite and if it is then the method of Cholesky decomposition is applied. MATLAB uses the the algorithm from LAPACK for computing the Cholesky decomposition. This algorithm includes a lot of rescaling, so it is in some sense adapted for computing the decomposition of the poorly-conditioned matrices. In C++ computations we used the common formulas for Cholesky decomposition without adaptivity and rescaling,

which can be found in any book on numerical methods (for example [1]). At the same time in both MATLAB and C++ double precision computations we got exactly the same limit for the degree of polynomial, at which Cholesky decomposition cannot be performed any more. This limit for double precision calculations is, as mentioned above, $n = 20 \sim 25$, depending on the value of ε - the limit value n becomes larger as ε tends to zero. The only difference is that in MATLAB the results look a bit smarter as we approach this limit. This means that if we construct an approximation function from the computed vector of coefficients with respect to the chosen basis of harmonic polynomials, we get some oscillations when approaching the limit degree of polynomial if we use non-adapted simple algorithm (as in C++) for Cholesky decomposition. The approximation error nevertheless remains the same. If we replace the standard algorithm for Cholesky decomposition used in MATLAB by the algorithm without adaptivity (but also implemented in MATLAB), which we used in C++, we get exactly the same results. This verifies the code in C++ and allows us to proceed to the next step - implementation of arbitrary precision arithmetic.

The ARPREC library is written in such a way that the previous code, written for double precision arithmetic, remains the same with only change of type of variables from "double" to a special type for variables with arbitrary precision arithmetic. All the functions, such as sine, cosine or logarithm, are defined for such variables. There are also special functions that compute such constants as π, e and others with the required precision. This makes the use of the library rather simple. The precision (notably the number of decimal digits) is determined at the beginning of all computations and may be chosen easily up to 1000. Due to this we expect that irrespective of the algorithm used for computing Cholesky decomposition we will get fine results almost for any degree of polynomial (but not too large, so that the computed values have the order of 10^{1000} that is the order of the machine precision in this case) used for approximation, even if the matrix becomes very ill-conditioned. Unfortunately, for some reason we don't obtain the results we expect. We managed to go to larger degrees of polynomials (in comparison with double precision arithmetic), but at some point again got the limit. The limit in this case has moved to $n = 45 \sim 60$, depending on the value of ε. The problem was the same as for double precision computations - the Cholesky decomposition could not be computed due to the same problem as we described earlier. It seems that at some point we lose the precision of our calculations. Moreover, if we prescribe different number of digits (more than that for double precision) the results remain the same. It seems that something is computed with double precision instead of that, prescribed beforehand. Unfortunately we did not manage to find this very point, and this

problem remains unsolved. Nevertheless we managed to extend the limit at least twice to greater degrees n and obtained the rate of convergence that holds at greater range of degrees. We now leave the problem with implementation the arbitrary precision library unsolved and proceed to discussing the quantitative results of our simulations.

Chapter 3

Computational results

3.1 Rates of convergence

The main results we are to present here are the rates of convergence of approximation with respect to the degree of polynomial used for approximation. We present the computational results for various values of ε. We will now refer to the results of C++ arbitrary precision implementation. In most cases we preserve 100 decimal digits, but as we mentioned above the results (for some reason) do not change if we consider 50 of 500 digits for example.

Speaking generally we got the results that justified our expectations in terms of the rate of convergence. For all cases we got exponential convergence in the degree of polynomial. But still the quantitative description of the convergence varies, depending on the values of ε (recall that it determines how close we get to the singularity of the function).

Consider first a sufficiently large value $\varepsilon = 0.9$. We are now very far away from singularity.For this value of ε we clearly observe exponential convergence of approximation to the original function in the degree of polynomial, with respect to the L^2–norm (Fig.2). The blue line contains the "experimental curve" and the red one is a linear approximation for it. We clearly see that all experimental (numerically computed) points almost lie on this line. At $n > 35, n = 36 \sim 37$ we get the numerical limit of computations for this value of ε that we described above, but up to this limit we observe very good convergence that is indeed exponential: $||u - u_n||_{L^2(\Omega)} \sim e^{-n}$. Here u is the original function and u_n — its best approximation by harmonic polynomials with highest degree n.

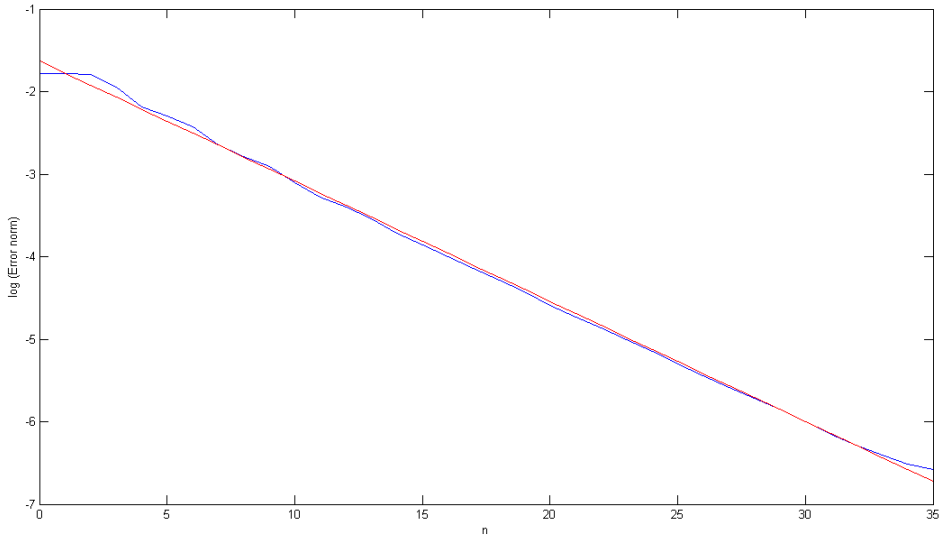

Fig.2. Norm of the approximation error (logarithmic scale); $\varepsilon = 0.9$.

Consider now a smaller value $\varepsilon = 0.5$. In this case we again observe very good exponential convergence, but at the same time the results are slightly different from the previous case. The experimental curve still can be very good approximated by a line, but now it becomes more a "curve" than a straight line (Fig.3).

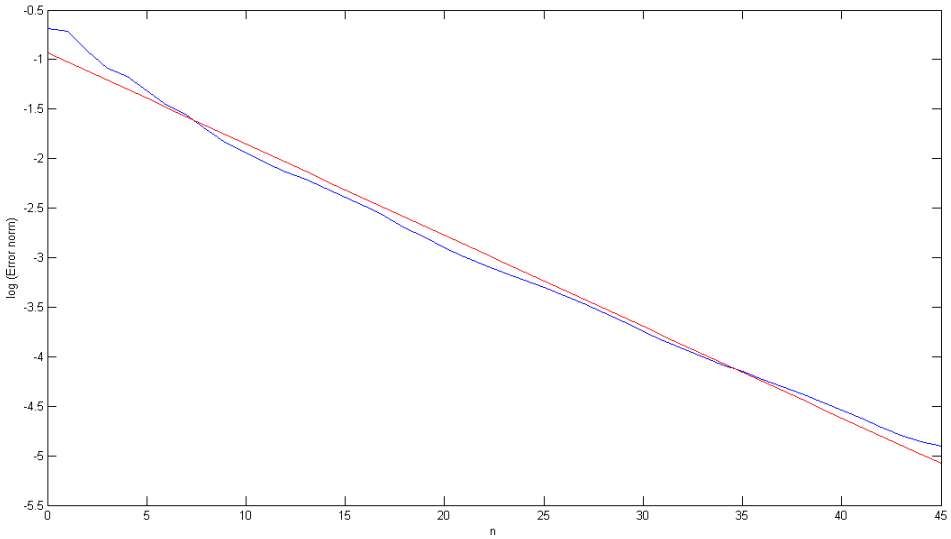

Fig.3. Norm of the approximation error (logarithmic scale); $\varepsilon = 0.5$.

We now decrease ε further and consider the value $\varepsilon = 0.1$. Expecting the same model convergence we obtain in this case the results, which are not accurate any more. At the Fig.4 we plot the dependence of the logarithm of the error norm on the degree of polynomial, but we notice that it is really not a straight line any more. The experimental curve deviates from the model line too far, so that we can not treat this model line as a right dependence. At the same time we see that the experimental line only gets some curvature but in principle the character of this dependence does not change drastically. This means that the convergence still remains exponential and does not change to algebraic for example. But now we get some additional exponent at the degree of polynomial n. Therefore we consider the model that seems to give a better description to the observed results: $||u - u_n||_{L^2(\Omega)} \sim e^{-n^\alpha}$ with some $0 < \alpha < 1$. The idea of exponential convergence with some exponent parameter has been already mention is some papers (see for example [8]). Considering in this case $\alpha = \frac{1}{2}$ we plot the square of the logarithm of the error norm instead of the logarithm itself (Fig.5). After that we again observe good correspondence between experimental curve and model line, but now a new model is used, which is really a simple generalization of the notion of exponential convergence.

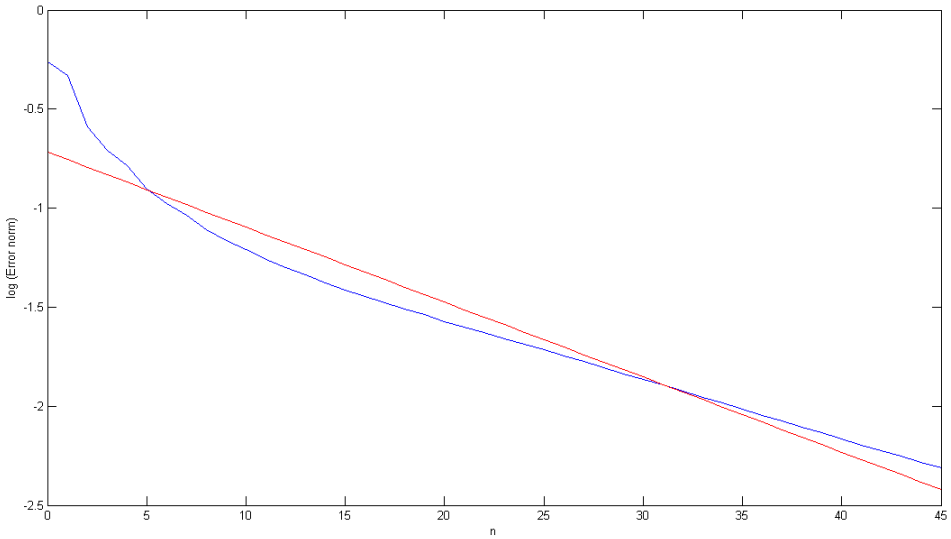

Fig.4. Norm of the approximation error (logarithmic scale); $\varepsilon = 0.1$.

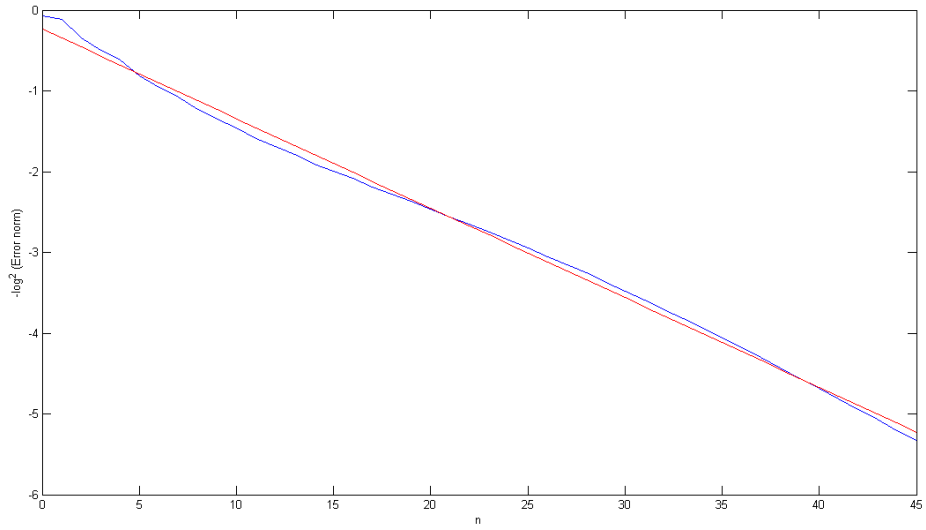

Fig.5. Norm of the approximation error ("squared" logarithmic scale); $\varepsilon = 0.1$.

Further decrease of ε will also involve the decrease of the exponent parameter α. For $\varepsilon = 0.05$ the right choice of this parameter is $\alpha = \frac{1}{3}$ and thus the dependence of the third power of the logarithm of the approximation error is linear with respect to the degree of polynomial (Fig.6).

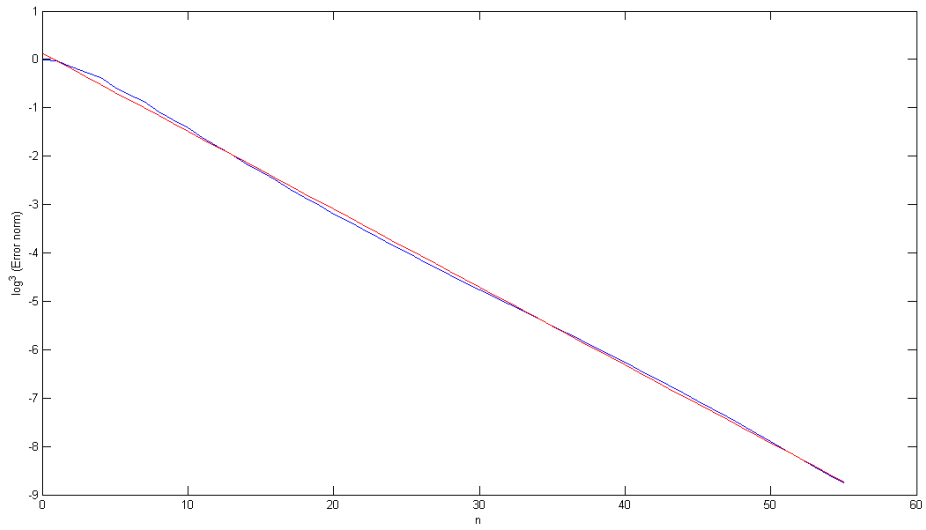

Fig.6. Norm of the approximation error ("cubed" logarithmic scale); $\varepsilon = 0.05$.

If we decrease ε further we will get the right value $\alpha = \frac{1}{4}$ for $\varepsilon = 10^{-4}$ (Fig.7).

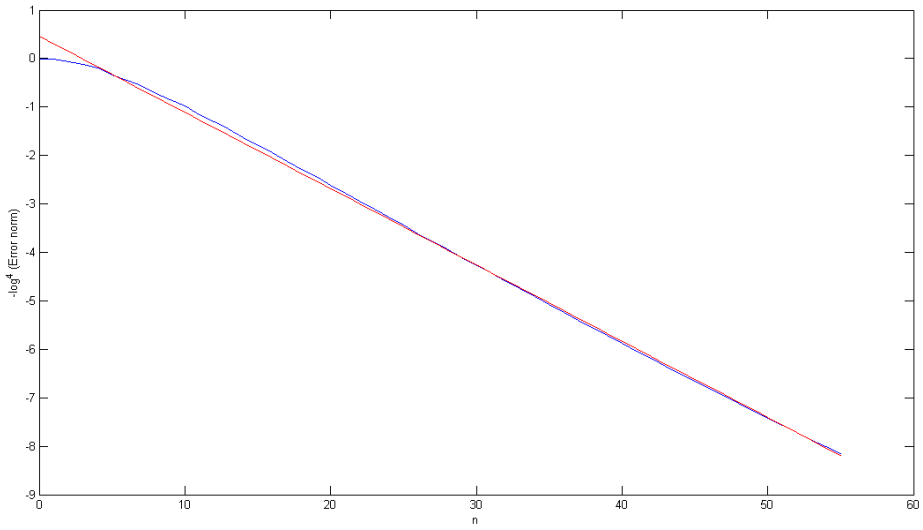

Fig.7. Norm of the approximation error ("4th power" logarithmic scale); $\varepsilon = 10^{-4}$.

At the same time if get even close to singularity point, decreasing ε even more, we will not get the values of parameter α less than $\frac{1}{4}$. Even for $\varepsilon = 10^{-6}$ and $\varepsilon = 10^{-10}$ the choice $\alpha = \frac{1}{4}$ appears to be exactly right. In this situation the value $\alpha = \frac{1}{4}$ seems to be a kind of a limit value. It does not matter how close we get to the singularity point the approximation will still exponentially converge with the exponent parameter not less than $\frac{1}{4}$.

3.2 Distribution of coefficients

The next important result, which can be obtained from the approximation problem, is determination of the coefficient distribution. At this point we should again recall that the basis of harmonic polynomials that we use in our computations is non-orthogonal. If it were, there would be almost no question about the distribution of coefficient. In case of an orthogonal (orthonormal, that is even better) the coefficients of the best approximation are simply the Fourier coefficients and they decrease fast with the degree of approximating polynomial tending to infinity. But in our case the basis in not orthogonal and this leads to completely different distribution of coefficients.

The shape of the distribution in essence does not depend on the degree of polynomial, it in fact remains the same. There are two main properties of this distribution. The first one is that the signs of coefficients alternates (Fig.8).

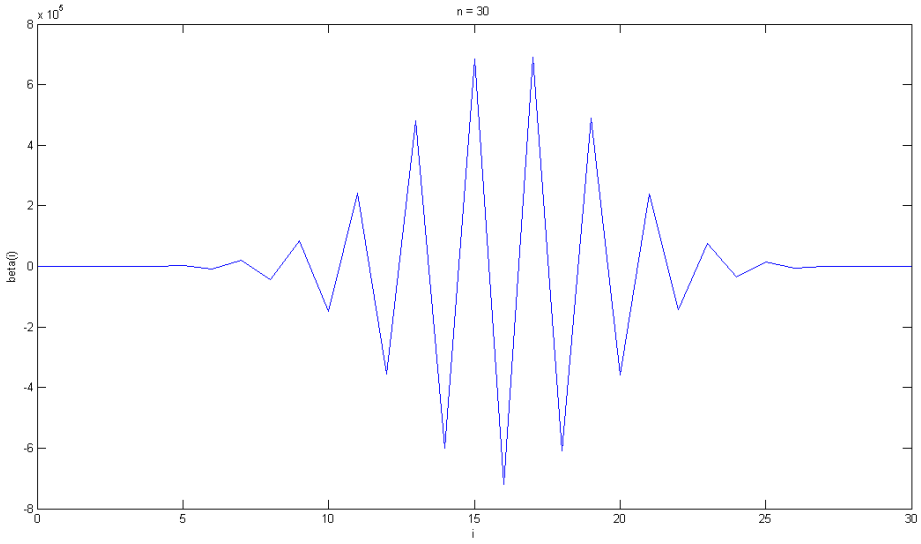

Fig.8. Approximation coefficients, $\varepsilon = 0.5, n = 30$.

For example, all odd coefficients are negative and all even coefficients are positive. At the same time the distribution of the modulus of coefficients (and this is the second property) resembles the Gaussian distribution (Fig.9). The red line corresponds to the experimentally obtained coefficient distribution and the green one - to the Gaussian approximation.

The picture is almost symmetric within the range of degrees of polynomials for a given maximum degree n. If the distribution were Gaussian this could give an idea and a possibility to compute the coefficients without solving the linear system (which has an ill-conditioned matrix and thus this problem is rather difficult), but choosing the parameters of the Gaussian distribution for the coefficients instead. However the distribution is not exactly Gaussian and the modulus of coefficients decreases faster than the Gaussian curve, as we can see in the logarithmic scale (Fig.10). The question about the coefficient distribution still remains open at the moment.

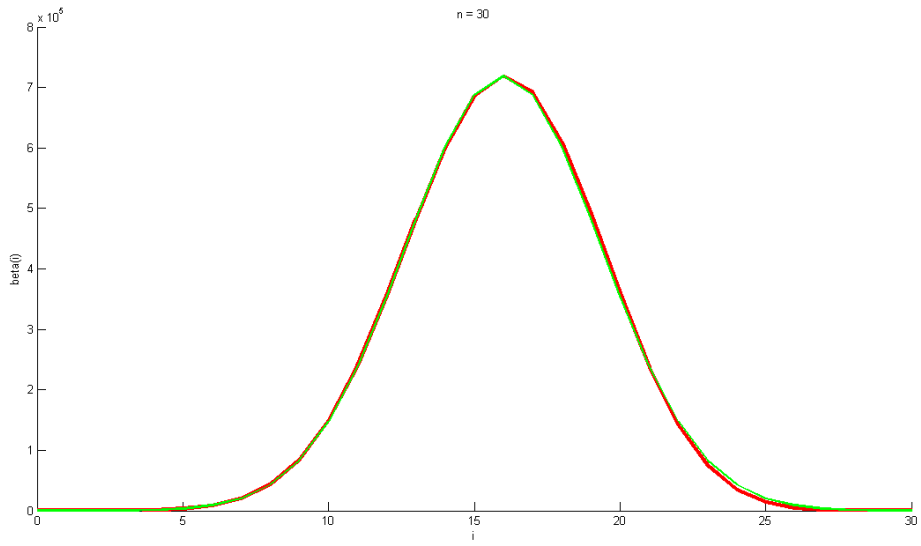

Fig.9. Modulus of approximation coefficients, $\varepsilon = 0.5, n = 30$.

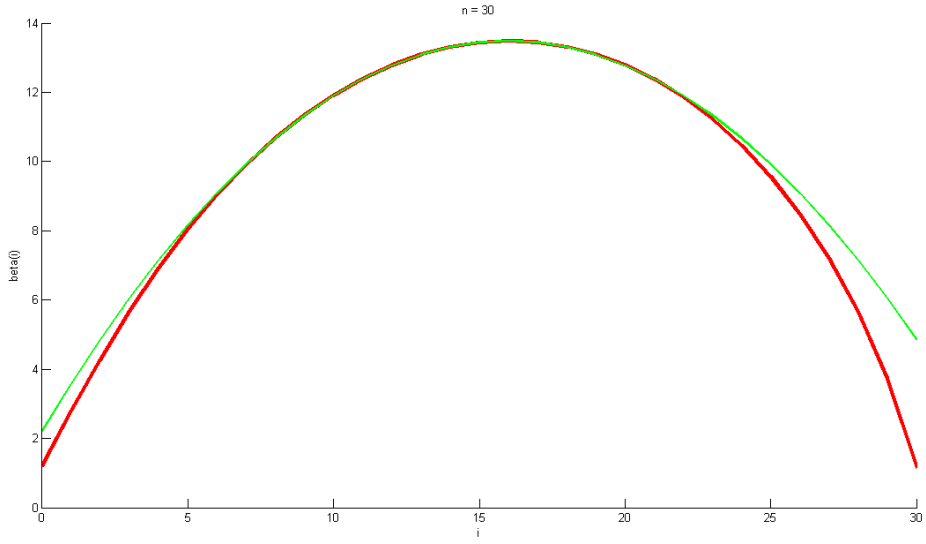

Fig.10. Modulus of approximation coefficients, logarithmic scale $\varepsilon = 0.5, n = 30$.

Chapter 4

Conclusion

Within this work we analyzed the convergence of an approximation of a function, which has a singularity at some point, by harmonic polynomials in two-dimensional case. The singularity point was not included it the computational domain and therefore the original function remains analytic in this domain. We analyzed the rate of convergence of such approximation for different case depending on that, how close to the approximation domain is the singularity point of the original function. In all the cases we observed exponential convergence in the degree of polynomial used for approximation, but in some of them - with an additional exponent parameter. The approximation error may be generally described by the formula

$$||u - u_n||_{L^2(\Omega)} \sim e^{-n^\alpha} \tag{11}$$

with some $0 < \alpha < 1$. The parameter tends to 1 if we are far from the singularity point and it decreases as the domain get closer to the singularity point. Nevertheless we observed a limit for α that clearly hold in our numerical experiments. The exponent parameter α does not go below $\frac{1}{4}$ even if get very close to the singularity point ($\varepsilon \sim 10^{-10}$ and even smaller).

The question with the coefficient distribution however remains unsolved. The distribution seems very much to be like Gaussian, but in the logarithmic scale as we saw above the coefficients decrease even faster than the Gaussian distribution. At the same time we faced some problems when applying the ARPREC library for arbitrary precision computations. They may have also influenced the coefficient distribution. We got the idea that at some point we lose accuracy and some computations are performed with lower precision. However this should be checked carefully and this is the problem for future investigations.

Bibliography

1. Kalitkin N.N. Numerical methods. Moscow: Nauka, 1978, (in Russian).

2. A. Moiola, R. Hiptmair and I. Perugia. Approximation by plane waves. Research report 2009-27, SAM, ETH Zürich. August 2009.

3. ARPREC: an arbitrary precision computational package. David H. Bailey, Yozo Hida, Xiaoye S. Li and Brandon Thompson. Lawrence Berkeley National Laboratory. Berkeley. CA94720. 2002.

4. J. L. Walsh, The approximation of harmonic functions by harmonic polynomials and by harmonic rational functions, Bull. Amer. Math. Soc., 35 (1929), pp. 499–544.

5. Interpolation and approximation by rational functions in the complex domain, Fifth edition. American Mathematical Society Colloquium Publications, Vol. XX, American Mathematical Society, Providence, R.I., 1969.

6. J. L. Walsh, W. E. Sewell, H. M. Elliott. On the Degree of Polynomial Approximation to Harmonic and Analytic Functions. Transactions of the American Mathematical Society, Vol. 67, No. 2 (Nov., 1949), pp. 381-420.

7. J. L. Walsh. An Interpolation Problem for Harmonic Functions. American Journal of Mathematics, Vol. 76, No. 1 (Jan., 1954), pp. 259-272.

8. V. Andrievskii, Uniform harmonic approximation on compact sets in R^k, $k \geq 3$, SIAM J. Math. Anal., 24 (1993), pp. 216–222.

Appendix

Here we present the codes that were elaborated for computer implementation of the described above algorithms.

C++ codes for arbitrary precision arithmetic

(See also [3] for information on ARPREC)

<header_ all.h>

```
# include <arprec/mp_ real.h>
// Functions dealing with norms
mp_ real lognorm(mp_ real eps);
mp_ real error_ norm(mp_ real *beta, mp_ real *f_ right, mp_ real
**matr_ A,mp_ real eps, int n);
// Functions fou solving linear system
mp_ real* f_ right(int n, mp_ real eps);
mp_ real** matrix_ A(int n, mp_ real eps);
mp_ real **cholesky(mp_ real**matr,int n);
mp_ real *solve_ down(mp_ real**m_ down,mp_ real*y,int n);
mp_ real *solve_ up(mp_ real**m_ up,mp_ real*y,int n);
```

<compute_ norms.cpp>

```
//# include <math.h>
# include <arprec/mp_ real.h>
# include <iostream>
# include "Poly_ arprec/header_ all.h"
using namespace std;
```

```
// Compute the norm of the function u(x) = ln( abs(x) ), x is a 2D vector,
in L2(omega),
// eps - inner radius of the semi-ring, the outer radius equals 1.
mp_ real lognorm(mp_ real eps){
//mp::mp_ init(3);
mp_ real z;
mp_ real pi;
mp_ real::mppi(pi);
z = pi*pow(eps,2)/2*( -pow(log(eps),2) + log(eps) ) + pi*(1-pow(eps,2))/4;
cout « "Lognorm = " « z « endl « endl;
return z;
};
mp_ real error_ norm(mp_ real *beta, mp_ real *f_ right, mp_ real
**matr_ A, mp_ real eps, int n){
//mp::mp_ init(3);
mp_ real I1("0.00");
mp_ real I2("0.00");
mp_ real I3("0.00");
I1 = lognorm(eps);
for(int i=0;i<n+1;i++){
I2 = I2 + beta[i]*f_ right[i];
};
cout « "I2 = " « I2 « endl;
mp_ real s("0");
for(int j=0;j<n+1;j++){
s = 0;
for(int k=0;k<n+1;k++){
s = s + matr_ A[j][k]*beta[k];
};
I3 = I3 + beta[j]*s;
};
cout « "I3 = " « I3 « endl;
cout « "I2-I3 = " « I2-I3 « endl;
if(I1 - 2*I2 + I3 >=0){
return sqrt(I1 - 2*I2 + I3);
}
else{
return sqrt(-(I1 - 2*I2 + I3));
};
};
```

<lin_ syst.cpp>

```
# include <arprec/mp_ real.h>
# include <iostream>
# include "Poly_ arprec/header_ all.h"
using namespace std;
// Compute the vector of the right side: right(j) = (u, b(j))
mp_ real* f_ right(int n, mp_ real eps){
//mp::mp_ init(3);
mp_ real pi;
mp_ real::mppi(pi);
mp_ real *z = new mp_ real[n+1];
for(int k = 0; k<n+1; k++){
z[k] = 0; };
z[0] = -pi/2*pow(eps,2)*log(eps) - pi*(1-pow(eps,2))/4;
for (int k = 1;k<=(n+1)/2;k++){
z[2*k-1] = 2*pow(-1.0,k)/(2*k-1) * ( pow(eps,2*k+1)*log(eps)/(2*k+1) +
(1-pow(eps,2*k+1))/(2*k+1)/(2*k+1) );
};
return z;
};
// Compute the matrix of the main equation: A(i,j) = (b(i), b(j))
mp_ real **matrix_ A(int n, mp_ real eps){
mp_ real pi;
mp_ real::mppi(pi);
mp_ real **z = new mp_ real*[n+1];
for(int k=0;k<n+1;k++){
z[k] = new mp_ real [n+1];
};
z[0][0] = pi/2*(1-pow(eps,2));
for(int i = 0;i<n+1;i++){
for(int j = 0;j<n+1;j++){
if ( (i == j) & & ( (i!=0) || (j!=0) )){
z[i][j] = pi/2*( (1-pow(eps,(2*i+2)))/(2*i+2) );
};
if (i != j){
z[i][j] = (1-pow(eps,(i+j+2)))/(i+j+2) * ( ( (1/(mp_ real)(i+j))*sin(pi*(i+j)/2)
+ 1/(mp_ real)(i-j)*sin(pi*(i-j)/2) ));
```

```
};
};
};
return z;
};
// Compute Cholesky decomposition of the matrix A
mp_ real **cholesky(mp_ real**matr,int n){
//mp::mp_ init(3);
mp_ real **z = new mp_ real*[n+1];
for(int k=0;k<n+1;k++){
z[k] = new mp_ real [n+1];
};
mp_ real alpha = pow(10.0,-15);
for(int i=0;i<n+1;i++){
for(int j=0;j<n+1;j++){
z[i][j] = 0;
};
};
mp_ real s1("0");
mp_ real s2("0");
for(int j=0;j<n+1;j++){
s1 = 0;
if(j!=0){
for(int l=0;l<j;l++){
s1 = s1 + z[j][l]*z[j][l];
};
};
z[j][j] = pow(matr[j][j] - s1, .5);
for(int i=j+1;i<n+1;i++){
s2 = 0;
if(j!=0){
for(int k=0;k<j;k++){
s2 = s2 + z[j][k]*z[i][k];
};
};
z[i][j] = (matr[i][j]-s2)/z[j][j];
};
};
return z;
}
// Compute solution of linear system Rx = y with lower triangular matrix
```

```
R
mp_ real *solve_ down(mp_ real**m_ down,mp_ real*y,int n){
//mp::mp_ init(3);
mp_ real *z = new mp_ real[n+1];
for(int k = 0; k<n+1; k++){
z[k] = 0;
};
z[0] = y[0]/m_ down[0][0];
mp_ real s("0");
for(int i=1;i<n+1;i++){
s = 0; for(int j=0;j<i;j++){
s = s + m_ down[i][j]*z[j];
};
z[i] = (y[i]-s)/m_ down[i][i];
};
return z;
};
// Compute solution of linear system R'z = x with lower triangular matrix
R
// !!! R is obtained through Cholesky decomposition
mp_ real *solve_ up(mp_ real**m_ down,mp_ real*y,int n){
mp_ real *z = new mp_ real[n+1];
for(int k = 0; k<n+1; k++){
z[k] = 0;
};
z[n] = y[n]/m_ down[n][n];
mp_ real s("0");
for(int i=n-1;i>=0;i–){
s = 0; for(int j=n;j>i;j–){
s = s + m_ down[j][i]*z[j];
};
z[i] = (y[i]-s)/m_ down[i][i];
};
return z;
};
```

<polyfit_ main.cpp>

```
# include <iostream>
```

```
# include <fstream>
# include <arprec/mp_ real.h>
# include <stdio.h>
# include <math.h>
# include "Poly_ arprec/header_ all.h"
using namespace std;
void main()
{
fstream fs_ e("Data/err_ norm.txt", ios::out);
fstream fs_ coeff("Data/coeff.txt", ios::out);
if (!fs_ e || !fs_ coeff){
cout « "Can't open file.\ n"; return;
}
else{
mp::mp_ init(100);
int n_ max = 45;
// Initial data
mp_ real eps = .1;
cout « "Epsilon = " « eps « endl;
for(int n=0;n<=n_ max;n++){
// Compute and print L2-norm of log-function in Omega
// mp_ real a = lognorm(eps);
// cout « "Norm of log = " « a « endl « endl;
// Compute right hand-side vector f
mp_ real *y;
y = f_ right(n, eps);
/* cout « "Right hand-side vector f" « endl;
for(int j = 0;j<n+1;j++){
cout « y[j] « endl;
};
cout « endl;
*/
// Compute matrix A
mp_ real **r;
r = matrix_ A(n, eps);
/* cout « "Matrix A" « endl;
for(int i = 0;i<n+1;i++){
for(int j = 0;j<n+1;j++){
//cout « r[j+(n+1)*1] « "";
cout « r[i][j] « "";
};
```

```
cout « endl;
};
cout « endl;
cout « endl;
*/
// Compute matrix chol(A)
mp_ real **q;
q = cholesky(r,n);
/* cout « "Chol (A)" « endl;
for(int i = 0;i<n+1;i++){
for(int j = 0;j<n+1;j++){
//cout « q[j+(n+1)*1] « "";
cout « q[i][j] « "";
};
cout « endl;
};
*/
cout « endl;
// Compute vector 'beta' of expansion coefficients
mp_ real *x = solve_ down(q,y,n);
mp_ real *beta = solve_ up(q,x,n);
for(int j = 0;j<n+1;j++){
//cout « beta[j] « endl;
fs_ coeff « beta[j] « endl; };
cout « endl;
// Compute approximation error in L2-norm mp_ real e_ norm = error_
norm(beta, y, r, eps, n);
cout « "Degree = " « n « endl;
cout « "Error norm = " « e_ norm « endl;
cout « endl;
cout « endl;
//FILE * ff = fopen("err_ norm.txt", "w");
//ff « e_ norm;
//fclose(ff);
fs_ e « n « " " « e_ norm « endl;
// Deallocate memory
if(x!=0)
{
delete[] x;
};
if(beta!=0)
```

```
{
delete[] beta;
};
if(y!=0)
{
delete[] y;
};
if(r!=0)
{
for(int k=0;k<n+1;k++){
delete[] r[k];
};
delete r;
};
if(q!=0)
{
for(int k=0;k<n+1;k++){
delete[] q[k];
};
delete q;
};
};
mp::mp_ finalize();
fs_ e.close();
fs_ coeff.close();
};
};
```

MATLAB codes for plotting the results computed within C++ implementation

<function fread_ error>

```
[f_ name2open f_ path2open] = uigetfile('Data\err_ norm.txt','MultiSelect','off');
if isequal(f_ name2open,0) | isequal(f_ path2open,0)
errordlg('No File');
else
s_ name2open = strcat(f_ path2open, f_ name2open);
```

```
end;
ff = fopen(s_ name2open, 'r'); A = fscanf(ff, '% i 10 ∧ % d x % g', [3 inf]);
fclose(ff);
n = A(1,:);
err = A(3,:).*10.∧( A(2,:) );
log_ err = log10(err);
% Exponent parameter
mm = 4;
figure;
plot(n,sign(log_ err).*log_ err.∧ mm);
r = polyfit(n,sign(log_ err).*log_ err.∧ mm,1);
approx_ line = r(1)*n + r(2);
hold on
plot(n,approx_ line,'r');
xlabel('n');
ylabel('-log∧4 (Error norm)');
figure;
plot(n,log_ err); p = polyfit(n,log_ err,1);
approx_ line = p(1)*n + p(2);
hold on
plot(n,approx_ line,'r');
xlabel('n');
ylabel('log (Error norm)');
```

<function fread_ coeff30>

```
[f_ name2open f_ path2open] = uigetfile('Data\coeff30.txt','MultiSelect','off');
if isequal(f_ name2open,0) | isequal(f_ path2open,0)
errordlg('No File');
else
s_ name2open = strcat(f_ path2open, f_ name2open);
ff = fopen(s_ name2open, 'r');
A = fscanf(ff, '10 ∧ % d x % g ',[2 inf]);
fclose(ff);
n = 0:size(A,2)-1;
coeff = A(2,:).*10.∧( A(1,:) );
figure;
eps = .1;
step = (1-eps)/100;
```

```
r = eps:step:1;
phi = -pi/2:pi/100:pi/2;
X = r'*cos(phi);
Y = r'*sin(phi);
Z = zeros(length(r),length(phi));
for i = 1:length(r)
for j = 1:length(phi)
for k = 1:length(coeff)
Z(i,j) = Z(i,j) + coeff(k) * b(k-1,r(i),phi(j));
end;
end;
end;
surf(X,Y,Z);
shading interp;
xlabel('x');
ylabel('y'); zlabel('C++ arprec approximation log(r)');
n2 = length(coeff)-1;
A = matrix_ A (n2, .5);
f = right (n2, .5);
z = A\f;
delta = z - coeff';
Z2 = zeros(length(r),length(phi));
for i = 1:length(r) for j = 1:length(phi)
for k = 1:length(coeff)
Z2(i,j) = Z2(i,j) + z(k) * b(k-1,r(i),phi(j));
end;
end;
end;
figure;
surf(X,Y,Z2);
shading interp;
xlabel('x');
ylabel('y');
zlabel('MatLab approximation log(r)');
Z3 = zeros(length(r),length(phi));
for i = 1:length(r)
for j = 1:length(phi)
for k = 1:length(coeff)
Z3(i,j) = Z3(i,j) + delta(k) * b(k-1,r(i),phi(j));
end;
end;
```

```
end;
figure;
surf(X,Y,Z3);
shading interp;
xlabel('x');
ylabel('y');
zlabel('Error2 of approximation log(r)');
end;
```