

**FINDING THE LOCATION AND SHAPE OF BURIED
OBSTACLES USING ACOUSTIC OR
ELECTROMAGNETIC WAVES**

M.Sc. Thesis by

Tolga Ulaş GÜRBÜZ, B.Sc.

Department : Electronics and
Communication Engineering
Programme: Telecommunication
Engineering

JUNE 2007

**FINDING THE LOCATION AND SHAPE OF BURIED
OBSTACLES USING ACOUSTIC OR
ELECTROMAGNETIC WAVES**

**M.Sc. Thesis by
Tolga Ulaş GÜRBÜZ, B.Sc.
(504011255)**

**Date of submission : 7 May 2007
Date of defence examination: 13 June 2007**

**Supervisor (Chairman): Prof. Dr. İbrahim AKDUMAN
Members of the Examining Committee Assoc. Prof.Dr. Ali YAPAR (İ.T.U.)
Assoc. Prof.Dr. İbrahim TEKİN (S.U.)**

JUNE 2007

**GÖMÜLÜ İLETKEN CİSİMLERİN KONUMLARININ VE
ŞEKİLLERİNİN AKUSTİK VEYA ELEKTROMAGNETİK
DALGALAR KULLANILARAK SAPTANMASI**

**YÜKSEK LİSANS TEZİ
Müh. Tolga Ulaş GÜRBÜZ
(504011255)**

**Tezin Enstitüye Verildiği Tarih : 7 Mayıs 2007
Tezin Savunulduğu Tarih : 13 Haziran 2007**

**Tez Danışmanı : Prof.Dr. İbrahim AKDUMAN
Diğer Jüri Üyeleri Doç.Dr. Ali YAPAR (İ.T.Ü.)
Doç. Dr. İbrahim TEKİN (S.Ü.)**

HAZİRAN 2007

PREFACE

The inverse scattering problem is the problem of determining the characteristics of an object (its shape, internal constitution, etc.) from measurement data of radiation or particles scattered from the object. The problems related to bodies buried in a half-space are interesting and important examples of these problems. In many practical applications such as determination of underground tunnels and pipelines, detection of surface and land mines, observation of the seismic behavior of underground, etc , one usually faces with the problem of determining the properties of the object from a far field data or data obtained by the measurements performed only in the half-space not containing the body. Thus, these interesting and difficult problems become more and more attractive for researchers from different backgrounds.

This thesis proposes a method which yields good resolution in the reconstruction of the location and shape of a buried obstacle, with a less number of measurements. The mathematical similarities of electromagnetic and acoustic fields give us a chance to propose a method which can be implemented by using both EM and acoustic fields. For this thesis, we restricted our attention to the simplest case of two dimensions, which serves as a first step to the more involved and computationally much more expensive three-dimensional realistic situation.

This work could not be carried out without Prof. İbrahim AKDUMAN's help. I am grateful to him for his continuous support and encouragement.

June 2007

Tolga Ulaş GÜRBÜZ

INDEX

LIST OF ABBREVIATIONS	v
LIST OF FIGURES	vi
SUMMARY	vii
ÖZET	viii
1. INTRODUCTION	1
2. MODELLING THE FORWARD PROBLEM	4
2.1 Construction of the Green's Function	4
2.2 Far Field Properties of the Green's Function for a Layered Background Medium	6
2.3 Solution of the Scattering Problem	9
2.3.1 Green's formula for the layered medium	9
2.3.2 Rellich's lemma and uniqueness for the direct problem	11
2.3.3 An integral equation approach to solve the scattering problem	14
2.3.4 Solution of the integral equation via the Nyström method	15
3. A POINT SOURCE METHOD FOR RECONSTRUCTIONS	20
3.1 The Reciprocity Relation	20
3.2 Reconstruction Algorithm	21
4. NUMERICAL RESULTS	23
5. CONCLUSION	28
REFERENCES	29
BIOGRAPHY	31

LIST OF ABBREVIATIONS

PEC : Perfect Electric Conductor

LIST OF FIGURES

	<u>Page No</u>
Figure 1.1 : Geometry of the problem.....	2
Figure 2.1 : Regularity line of integration for the inverse Fourier transform..	6
Figure 2.2 : The mapping of C_1 onto the α - plane.....	7
Figure 3.1 : Illustration for the domain of approximation	21
Figure 4.1 : The original and reconstructed field for a source at z	24
Figure 4.2 : The original and reconstructed field for a different source point.	25
Figure 4.3 : Results of tresholding and logical operations	26

FINDING THE LOCATION AND SHAPE OF BURIED OBSTACLES USING ACOUSTIC OR ELECTROMAGNETIC WAVES

SUMMARY

In this thesis, we consider the problem of reconstructing the location and shape of a bounded perfectly conducting obstacle which is buried in a homogeneous half space. We restrict our attention to the simplest case of two dimensions, which serves as a first step to the more involved and computationally much more expensive three-dimensional realistic situation. We deal both the forward and inverse scattering problems for a buried obstacle.

First we represent the Green's function for the layered medium. Then, scattered field is expressed as a combined single- and double-layer potential in terms of the Green's function and a potential density. For the boundary values of this scattered field expression, an integral equation is obtained with the help of the Dirichlet boundary condition and jump relations. This integral equation is a singular boundary equation and is numerically solved for the potential density by using the Nyström method which is based on the straightforward approximation of the integrals by quadratures formulas. Thus, we obtain an approximating function which gives us the discrete values of the potential density function. By using this function, one can easily calculate the discrete values of the field scattered from a known obstacle.

The inverse scattering problem is then to recover the location and shape of the obstacle Ω through scattered field measurements along the line Λ for a line source illumination from the upper half-space. We establish a point-source method for the reconstruction of the location and shape of the obstacle. The method is based on the determination of the total field in the half-space containing the body through the measurement data. This is done by establishing the reciprocity relation between the scattered and incident field. Then one reconstructs the object by observing the variation of the total field in the region where the body is located. Finally, we describe some efficient implementation of the method and show numerical examples in two dimensions.

GÖMÜLÜ İLETKEN CİSİMLERİN KONUMLARININ VE ŞEKİLLERİNİN AKUSTİK VEYA ELEKTROMAGNETİK DALGALAR KULLANILARAK BELİRLENMESİ

ÖZET

Bu tezde, homojen bir yarı-uzaya gömülü mükemmel iletken bir cismin konumunun ve şeklinin belirlenmesi problemi ele alınmıştır. Gerçeğe daha yakın olan ancak çok daha yoğun hesaplama gerektiren üç boyutlu duruma giriş niteliğinde olmak üzere, iki boyutlu daha basit durum üzerinde durulmuştur. Gömülü bir cisim için geçerli olan düz ve ters saçılma problemlerinin her ikisi de ele alınmıştır.

Önce katmanlı ortam için Green fonksiyonu elde edilmiş, ardından saçılan alan, birleşik tek ve çift tabaka potansiyeller olarak, Green fonksiyonu ve potansiyel yoğunluğu cinsinden ifade edilmiştir. Bu saçılan alan ifadesinin sınır değerleri için, Dirichlet sınır koşulu ve sıçrama bağıntıları kullanılarak bir integral denklem elde edilmiştir. Bu denklem tekil bir sınır denklemdir ve integrallerin ayrık yaklaşıklarının kullanılması esasına dayanan Nyström yöntemi aracılığıyla sayısal olarak çözülmüştür. Böylelikle, potansiyel yoğunluğunun ayrık değerlerini veren bir yaklaşık potansiyel fonksiyonu elde edilir. Bu fonksiyon kullanılarak, bilinen bir cisimden saçılan alanın büyüklüğü kolaylıkla hesaplanabilir.

Ters saçılma problemi ise, üst yarı-uzayda bulunan bir çizgi kaynak tarafından aydınlatılan gömülü bir Ω cisminin konumunun ve şeklinin, bir Λ doğrusu boyunca yapılan saçılan alan ölçümleriyle saptanmasıdır. Cismin konumunu ve şeklini elde edebilmek için bir nokta-kaynak yöntemi tanıtılmıştır. Bu yöntem, cismin bulunduğu yarı-uzaydaki toplam alanın, ölçülen alan kullanılarak saptanmasına dayanır. Bu, saçılan alan ve gelen alan arasındaki karşılıklı ilişki kullanılarak yapılmıştır. Böylece, cismin gömülü olduğu bölgedeki toplam alan değişimi gözlenerek cismin konumu ve şekli saptanabilir. En sonunda, yöntemin gerçekleştirilmesinin kimi etkin biçimleri gösterilmiş ve iki boyutlu durum için yapılan sayısal çalışmaların sonuçlarından örnekler verilmiştir.

1. INTRODUCTION

During the last two decades inverse scattering problems connected with bodies buried in limited host mediums with known physical properties have become more and more attractive for researchers from different backgrounds such as mathematics, geophysics, acoustics, electromagnetics, etc. The reason of this intensive interest is the wide range of practical applications of these problems such as determination of underground tunnels and pipelines, detection of surface and land mines, observation of the seismic behaviour of underground, etc. The aim in such a problem consists of finding the geometrical as well as the physical properties of an inaccessible buried object by observing its effect on the propagation of certain waves. The problems related to bodies buried in a half-space are interesting and important examples of these problems. In such a case, one has to achieve the reconstruction from the limited data which is obtained by the illuminations and measurements performed only in the half-space not containing the body. This incompleteness of the data affects the behaviour of the problem very seriously. Here, we do not attempt to list publications in the open literature. For a detailed review we refer the readers to [2] and [7].

The main objective of this thesis is to propose a method which yields good resolution with a less number of measurements compared to other techniques in the open literature. To this aim, we consider the problem of reconstructing bounded sound-soft or perfectly conducting objects which are buried in a homogeneous half space. Measurements of the acoustic or electromagnetic field for the scattering of a fixed point-source illumination are taken on some surface above the ground. Then, we establish a point-source method for the reconstruction of the location and shape of the obstacle. The method is based on the determination of the total field in the half-space containing the body through the measurement data. This is done by establishing the reciprocity relation between the scattered and incident field. Then one reconstructs the object by observing the variation of the total field in the region where the body is located. We describe some efficient implementation of the method and show numerical examples in two dimensions.

For this thesis, we will restrict our attention to the simplest case of two dimensions, which serves as a first step to the more involved and computationally much more expensive three-dimensional realistic situation. Two-dimensional problems appear in applications with some symmetry along a geometrical axis.

Let us first consider the electromagnetic case. If we assume symmetry of the whole problem with respect to translations along the x_3 -axis, then we can model the electromagnetic field using the third component $u := E_3$ of the electric field vector. We consider a perfect conductor Ω which is buried in a lower homogeneous half space

$$H_2 := \{x = (x_1, x_2) \in \mathbb{R}^2, x_2 < 0\} \quad (1.1)$$

with the wave number κ_2 which is different from the wave number κ_1 of the upper homogeneous half space

$$H_1 := \{x = (x_1, x_2) \in \mathbb{R}^2, x_2 > 0\} \quad (1.2)$$

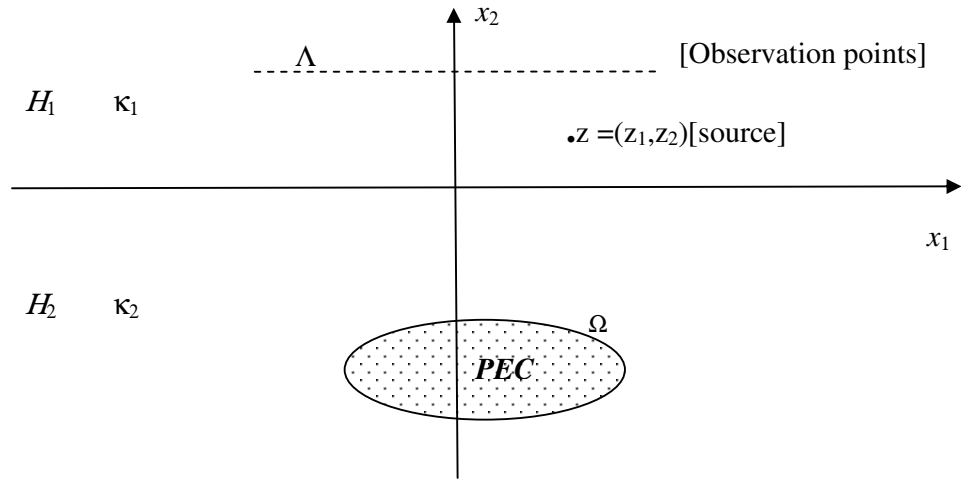


Figure 1.1 Geometry of the problem: Buried obstacle Ω in the lower half- space illuminated by a line source located at the point z

Here, for convenience we will assume that the domain Ω has boundary of class C^2 . The total field $u = u^i + u^s$ for scattering of some wave $u^i(x) = G(x, z)$ with source point in z is a solution to the Helmholtz equation

$$\Delta u(x) + \kappa_j^2 u(x) = 0, \quad j=1,2 \quad (1.3)$$

in $H_j \setminus (\bar{D} \cup \{z\})$ with the wave number κ_j . The scattered field u^s satisfies the Sommerfeld radiation condition

$$\sqrt{r} \left(\frac{\partial u^s}{\partial r} - i \kappa_j u^s \right) \rightarrow 0, \quad r = |x| \rightarrow \infty, \quad x \in H_j, \quad j=1,2, \quad (1.4)$$

uniformly for all directions $\hat{x} = x/|x|$. For perfectly conducting obstacles we know that the tangential component of the total electric field vanishes on the boundary. This leads to the Dirichlet boundary condition

$$u^s = -u^i \quad \text{on} \quad \partial\Omega. \quad (1.5)$$

Here, for simplicity, we assume that the normal derivative of u^s on $\partial\Omega$ exists. For boundaries of class C^2 this condition can be removed as in [4] for the homogeneous case using regularity properties of the solutions to the exterior Dirichlet problem. On the interface $\partial H_1 = \partial H_2$ we assume that both the function u and its normal components $\partial u / \partial \nu$ are continuous, i.e. we have

$$\lim_{h \rightarrow +0} (u(x + h\nu(x)) - u(x - h\nu(x))) = 0, \quad (1.6)$$

$$\lim_{h \rightarrow +0} e_2 \cdot (\nabla u(x + h\nu(x)) - \nabla u(x - h\nu(x))) = 0, \quad (1.7)$$

Uniformly on the plane $\Gamma_0 := \{x \in \mathbb{R}^2, x_2 = 0\}$. Note that the problem stated by (1.3) to (1.7) is the same for the sound-soft obstacle in acoustics.

The inverse scattering problem is then to recover the location and shape of the obstacle Ω through scattered field measurements along the line Λ for a line source illumination from the upper half-space.

The organization of the thesis is as follows. In Section 2, we give a general formulation of the forward scattering problem which in Section 3 we give a point source method for the reconstruction. In Section 4, numerical results are presented for both direct and inverse scattering problems. Finally a conclusion is made in Section 5.

2. MODELLING THE FORWARD PROBLEM

This section serves to fulfill two successive goals. First, we will briefly describe the construction of the Green's function G for the layered half-space problem. Here, we follow the approach suggested by Akduman and Alkumru [1]. Since a) in [1] only special cases are treated and b) we need to show several properties of the Green's function which are needed for the integral equation approach, we will give a concise derivation of these properties in this section.

The second goal is to give the solution of scattering of an incident line- source from the perfect conductor Ω .

2.1 Construction of the Green's Function

The Green's function G for the half- space problem is defined as the solution to the equation

$$\Delta_x G(x, z) + \kappa_j^2 G(x, z) = -\delta(x - z), \quad x \in H_j, \quad j=1,2, \quad (2.1)$$

such that $G(.,z)$ satisfies the continuity conditions (1.6) and (1.7) on Γ_0 and the radiation condition (1.4). In this equation $z \in \mathbb{R}^2$ is an arbitrary parameter and δ denotes Dirac's delta distribution. In order to find a suitable expression of G , now consider its Fourier transform with respect to x_1 , say $\hat{G}(v, x_2; z_2)$, which is the solution of the following problems:

$$\frac{d^2 \hat{G}}{dx_2^2} - (v^2 - \kappa_j^2) \hat{G} = -e^{-ivz_1} \delta(x_2 - z_2), \quad j=1,2, \quad (2.2)$$

$$\hat{G} \quad \text{and} \quad \frac{\partial \hat{G}}{\partial x_2} \quad \text{are continuous on} \quad x_2 = 0 \quad (2.3)$$

$$|\hat{G}| \rightarrow 0 \quad \text{as} \quad |x| \rightarrow \infty \quad (2.4)$$

After some straightforward calculations one can show that the function $\hat{G}$ has the following expressions:

$$\hat{G} = \frac{1}{2\gamma_1} e^{-i\nu z_1} \begin{cases} e^{-\gamma_1 |x_2 - z_2|} + R e^{-\gamma_1 (x_2 + z_2)}, & x_2 > 0, \quad z_2 > 0 \\ T e^{-\gamma_1 z_2 + \gamma_2 x_2}, & x_2 < 0, \quad z_2 > 0 \end{cases} \quad (2.5)$$

and

$$\hat{G} = \frac{1}{2\gamma_2} e^{-i\nu z_1} \begin{cases} T' e^{\gamma_2 z_2 - \gamma_1 x_2}, & x_2 > 0, \quad z_2 < 0 \\ e^{-\gamma_2 |x_2 - z_2|} + R' e^{\gamma_2 (x_2 + z_2)}, & x_2 < 0, \quad z_2 < 0 \end{cases} \quad (2.6)$$

where

$$R = \frac{\gamma_2 - \gamma_1}{\gamma_1 + \gamma_2}, \quad T = \frac{2\gamma_1}{\gamma_1 + \gamma_2} \quad (2.7)$$

$$R' = \frac{\gamma_1 - \gamma_2}{\gamma_1 + \gamma_2}, \quad T' = \frac{2\gamma_2}{\gamma_1 + \gamma_2} \quad (2.8)$$

In (2.5-8) the functions γ_1 and γ_2 stand for the square roots

$$\gamma_1(\nu) = \sqrt{\nu^2 - \kappa_1^2}, \quad \gamma_2(\nu) = \sqrt{\nu^2 - \kappa_2^2} \quad (2.9)$$

which are defined in complex ν -plane cut as shown in Figure 1.1 with the conditions

$$\gamma_j(0) = -i\kappa_j, \quad j=1,2. \quad (2.10)$$

Then the explicit expressions of the Green's function can be obtained through the well known inverse transform integral, namely;

$$G(x; z) = \frac{1}{2\pi} \int_{-\infty}^{\infty} \hat{G}(\nu, x_2; z) e^{i\nu x_1} d\nu \quad (2.11)$$

Substituting $\hat{G}(\nu, x_2; z)$ into (2.11) and rearranging the resulting expression yields

$$G(x; z) = \begin{cases} \frac{i}{4} H_0^{(1)}(k_1 |x-z|) + \frac{1}{2\pi} \int_{-\infty}^{\infty} \frac{1}{2\gamma_1} \frac{\gamma_1 - \gamma_2}{\gamma_1 + \gamma_2} e^{-\gamma_1(x_2+z_2)} e^{i\nu(x_1-z_1)} d\nu, & x_2 > 0, z_2 > 0 \\ \frac{1}{2\pi} \int_{-\infty}^{\infty} \frac{1}{\gamma_1 + \gamma_2} e^{-\gamma_1 z_2} e^{-\gamma_2 x_2} e^{i\nu(x_1-z_1)} d\nu, & x_2 < 0, z_2 > 0 \end{cases} \quad (2.12)$$

$$G(x; z) = \begin{cases} \frac{1}{2\pi} \int_{-\infty}^{\infty} \frac{1}{\gamma_1 + \gamma_2} e^{\gamma_2 z_2} e^{-\gamma_1 x_2} e^{i\nu(x_1-z_1)} d\nu, & x_2 > 0, z_2 < 0 \\ \frac{i}{4} H_0^{(1)}(k_2 |x-z|) + \frac{1}{2\pi} \int_{-\infty}^{\infty} \frac{1}{2\gamma_1} \frac{\gamma_1 - \gamma_2}{\gamma_1 + \gamma_2} e^{\gamma_2(x_2+z_2)} e^{i\nu(x_1-z_1)} d\nu, & x_2 < 0, z_2 < 0 \end{cases} \quad (2.13)$$

where $H_0^{(1)}$ is the Hankel function of the first kind of order 0. From (2.12) and (2.13) we can easily see that the Green's function G is symmetrical, i.e. $G(x; z) = G(z; x)$ and has the property

$$G(x; z) = \hat{G}(|x_1 - z_1|, x_2, z_2), \quad x, z \in \mathbb{R}^2. \quad (2.14)$$

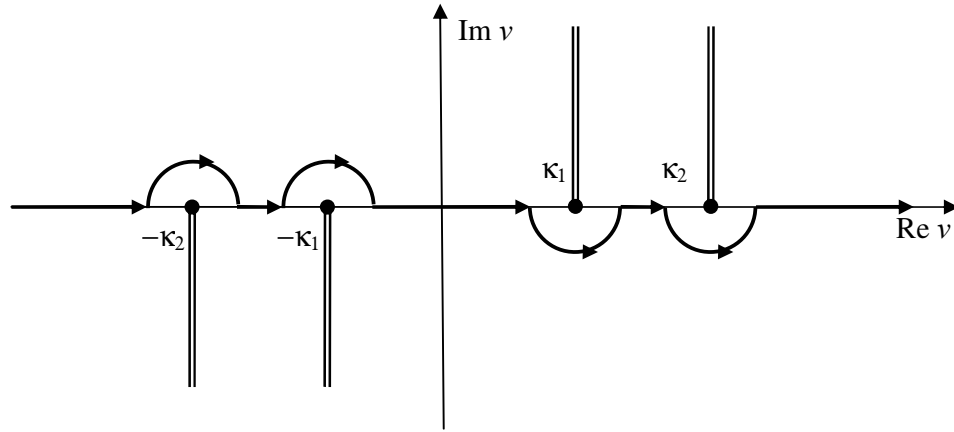


Figure 2.1 Regularity Line of, i.e., the path of integration for the inverse Fourier transform to calculate the Green's function.

2.2 Far Field Properties of the Green's Function for a Layered Background Medium

The far field expression of the Green's function given by (2.12) and (2.13) satisfies the radiation condition (1.4) and continuity conditions (1.6) and (1.7) uniformly. To show this property we will consider the case $x_2 > 0$, $z_2 < 0$ where the Green's function is given by

$$G(x; z) = \frac{1}{2\pi} \int_{-\infty}^{\infty} \frac{1}{\gamma_1 + \gamma_2} e^{\gamma_2 z_2 - \gamma_1 x_2} e^{iV(x_1 - z_1)} dV \quad (2.15)$$

and calculate $G(x; z)$ for $|x| \rightarrow \infty$. To this aim let us first make the following substitutions

$$x_1 = |x| \cos \phi, \quad x_2 = |x| \sin \phi, \quad \phi \in (0, \pi) \quad (2.16)$$

$$v = \kappa_2 \cos \alpha, \quad \gamma_2 = -i\kappa_2 \sin \alpha, \quad \alpha \in C^* \quad (2.17)$$

where C^* is the line illustrated in Figure 2.2.

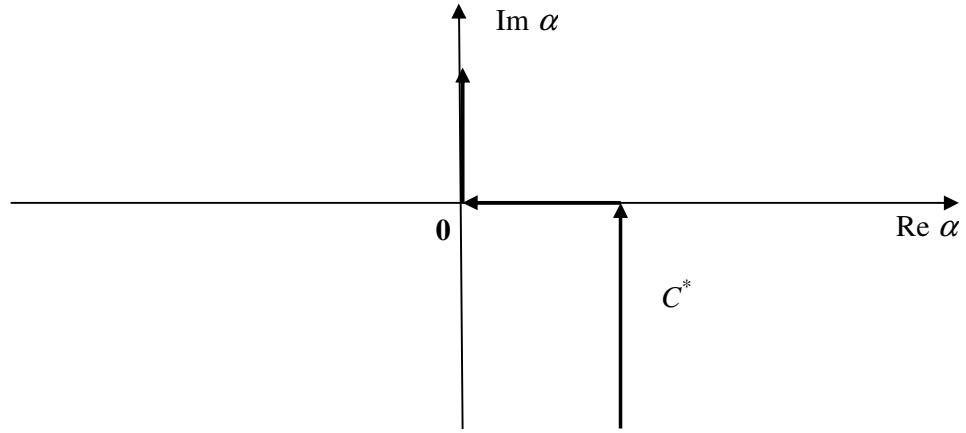


Figure 2.2 The mapping of C_1 onto the α - plane.

After some simple straightforward computations we get

$$G(|x|, \phi; z) = \frac{-i\kappa_2}{2\pi} \int_{C^*} f(\alpha) e^{|x|g(\alpha)} d\alpha \quad (2.18)$$

where

$$g(\alpha) = i(\sqrt{\kappa_1^2 - \kappa_2^2 \cos^2 \alpha} \sin \phi + \kappa_2 \cos \alpha \cos \phi) \quad (2.19)$$

$$f(\alpha) = \frac{\sin \alpha}{\sqrt{\kappa_1^2 - \kappa_2^2 \cos^2 \alpha} + \kappa_2 \sin \alpha} e^{-i\kappa_2(z_1 \cos \alpha + z_2 \sin \alpha)} \quad (2.20)$$

The integral in (2.18) is now in the Standard form susceptible to be evaluated for large values of $|x|$ asymptotically by the classical saddle-point technique [11]. Indeed the saddle-point occurs at:

$$\cos \alpha_s = \frac{\kappa_1}{\kappa_2} \cos \phi \quad \text{Re } \alpha_s \in (0, \pi). \quad (2.21)$$

Then by expanding the function $f(\alpha)$ into Taylor series around the saddle point α_s namely;

$$f(\alpha) = f(\alpha_s) + f'(\alpha_s)(\alpha - \alpha_s) + \frac{1}{2} f''(\alpha_s)(\alpha - \alpha_s)^2 + \dots \quad (2.22)$$

and following the known procedure for the saddle-point technique we obtain

$$\begin{aligned} \lim_{|x| \rightarrow \infty} G(|x|, \phi; z) &= \frac{e^{ik_1|x|}}{|x|} \frac{e^{i\frac{\pi}{4}} \sqrt{\kappa_1}}{\sqrt{2\pi}} \frac{\sin \phi}{\sin \alpha_s} f(\alpha_s) \\ &+ \frac{e^{i\kappa_1|x|}}{|x|^{\frac{3}{2}}} \frac{-ie^{i\frac{\pi}{4}} \kappa_1^{\frac{3}{2}}}{2\sqrt{2\pi\kappa_2^2}} \frac{\sin^2 \phi}{\sin^2 \alpha_s} f''(\alpha_s) \\ &+ O\left(\frac{1}{|x|^{\frac{5}{2}}}\right) \end{aligned} \quad (2.23)$$

The first term in (2.23) is called the far field expression of the Green's function. Note that the function $f(\alpha)$ is a regular function of α_s and its all derivatives exist and regular at that point.

On the other hand by the same analysis we can show that

$$\lim_{|x| \rightarrow \infty} \left| G(|x|, \phi; z) - \frac{e^{i\kappa_1|x|}}{|x|} - \frac{e^{i\frac{\pi}{4}} \sqrt{\kappa_1}}{\sqrt{2\pi}} \frac{\sin \phi}{\sin \alpha_s} f(\alpha_s) \right| \quad (2.24)$$

$$\leq \max_{\xi \in (\alpha_s, \alpha)} |f'''(\xi)| \frac{1}{2\sqrt{2\pi}} \frac{\kappa_1 \sin^2 \phi}{\kappa_2^2 \sin^2 \alpha_s} \frac{1}{|x|^{\frac{3}{2}}} \quad (2.25)$$

2.3 Solution of the Scattering Problem

For an incident field u^i given as a point-source

$$u^i(x) := G(x; z), \quad x \neq z \in \mathbb{R}^2 \quad (2.26)$$

with source-point z in the upper half-space H_1 , we search for a scattered field u^s which satisfies the Helmholtz equation (1.3), the radiation condition (1.4), the boundary condition (1.5) and the continuity properties (1.6) and (1.7).

Here we represent u^i as a function of x only, but as easily seen it is also a function of source point z . For simplicity it is preferred not to emphasize the effect of z in the following equations but it is obvious that changing the source point z will change $u^i(x)$ at any observation point and related terms as well.

2.3.1 Green's formula for the layered medium

As a tool we need Green's formula for the layered background medium. In particular, we need to pay attention to the interface Γ_0 between the lower and upper half space.

THEOREM 1. Let $\Omega \subset H_2$ be a bounded domain of class C^2 and let ν denote the unit normal vector to the boundary $\partial\Omega$ directed to the exterior of Ω . Consider a radiating field u such that $u|_{H_1} \in C^2(H_1)$ and $u|_{H_2} \in C^2(H_2 \setminus \overline{\Omega})$ which is a solution to the Helmholtz equation (2.1). We assume that u is continuous in $\mathbb{R}^2 \setminus \Omega$ and that it has continuous normal derivatives on the surface Γ_0 in the sense of (1.7) and that it possesses a normal derivative on the boundary $\partial\Omega$ in the sense that the limit

$$\frac{\partial u}{\partial \nu}(x) = \lim_{h \rightarrow +0} \nu(x) \cdot \text{grad } u(x - h\nu(x)), \quad x \in \partial\Omega \quad (2.27)$$

exists uniformly on $\partial\Omega$. Then we have Green's formula

$$u(x) = \int_{\partial\Omega} \left\{ u(y) \frac{\partial G(x, y)}{\partial \nu(y)} - \frac{\partial u}{\partial \nu}(y) G(x, y) \right\} ds(y), \quad x \in \mathbb{R}^2 \setminus \overline{\Omega}. \quad (2.28)$$

Proof. By B_R we denote a ball of radius R around the origin. We will apply the well-known Green's Theorem for a homogeneous background

$$\int_{\partial M} \left\{ \frac{\partial u}{\partial \nu}(y) G(x, y) - u(y) \frac{\partial G(x, y)}{\partial \nu(y)} \right\} ds(y) = \begin{cases} u(x), & x \in M \\ 0, & x \in \mathbb{R}^2 \setminus \overline{M} \end{cases} \quad (2.29)$$

in the half-ball $M = B_R \cap H_1$ and in the set $M = B_R \cap H_2 \setminus \Omega$, where ν is chosen such that it directs into the exterior of the domain under consideration. By continuity of the field $G(., y)$ and its normal derivative on Γ_0 the integrals over $M \cap \Gamma_0$ cancel each other and we obtain

$$\begin{aligned} u(x) = & \int_{\partial \Omega} \left\{ u(y) \frac{\partial G(x, y)}{\partial \nu(y)} - \frac{\partial u}{\partial \nu}(y) G(x, y) \right\} ds(y) \\ & + \int_{\partial B_R} \left\{ \frac{\partial u}{\partial \nu}(y) G(x, y) - u(y) \frac{\partial G(x, y)}{\partial \nu(y)} \right\} ds(y), \quad x \in \mathbb{R}^2 \setminus \overline{\Omega}. \end{aligned} \quad (2.30)$$

where the signs in the second integral arise from the choice of the normal derivative on ∂B_R which points into the exterior of B_R . We will show that the second integral converges to zero for $R \rightarrow \infty$. To this end we first show that

$$\int_{\partial B_R} |u|^2 ds = O(1), \quad R \rightarrow \infty. \quad (2.31)$$

We use

$$\left| \frac{\partial u}{\partial \nu} \right|^2 + \kappa_j^2 |u|^2 + 2\kappa_j \operatorname{Im}(u \frac{\partial \bar{u}}{\partial \nu}) = \left| \frac{\partial u}{\partial \nu} - i\kappa_j u \right|^2 \quad (2.32)$$

for each of the half-spaces H_j , $j=1,2$ and Green's first theorem in the sets $M = B_R \cap H_1$ and $M = B_R \cap H_2 \setminus \Omega$ together with the continuity of u and its normal derivative on Γ_0 to obtain

$$\begin{aligned} \int_{\partial B_R} u \frac{\partial \bar{u}}{\partial \nu} ds = & \int_{\partial \Omega} u \frac{\partial \bar{u}}{\partial \nu} ds - \int_{\partial B_R \cap H_1} (\kappa_1^2 |u|^2 + |\operatorname{grad} u|^2) dy \\ & - \int_{\partial B_R \cap H_2 \setminus \Omega} (\kappa_2^2 |u|^2 + |\operatorname{grad} u|^2) dy \end{aligned} \quad (2.33)$$

We define $\kappa(y) = \kappa_j$ for $y \in H_j$, $j = 1, 2$ and integrate $1/\kappa(y)$ times (2.32) over ∂B_R .

From the radiation condition (1.4) the integral on the right-hand side tends to zero for $R \rightarrow \infty$. Inserting the imaginary part of (2.33) into (2.32) we obtain

$$\lim_{r \rightarrow \infty} \int_{\partial B_R} \frac{1}{\kappa(y)} \left| \frac{\partial u(y)}{\partial \nu} \right|^2 + \kappa(y) |u(y)|^2 ds(y) = -2 \operatorname{Im} \int_{\partial \Omega} u \frac{\partial \bar{u}}{\partial \nu} ds. \quad (2.34)$$

Both terms on the left-hand side of (2.34) are nonnegative, thus they must be individually bounded for $R \rightarrow \infty$. We use the estimate

$$\int_{\partial B_R} \kappa(y) |u|^2 ds \geq \min_{j=1,2} \kappa_j \int_{\partial B_R} |u|^2 ds \quad (2.35)$$

to obtain (2.31). Finally, we decompose the second integral in (2.30) into

$$I_1 := \int_{\partial B_R} u(y) \left\{ \frac{\partial G(x, y)}{\partial \nu(y)} - i \kappa(y) G(x, y) \right\} ds(y) \quad (2.36)$$

and

$$I_2 := \int_{\partial B_R} G(x, y) \left\{ \frac{\partial u}{\partial \nu}(y) - i \kappa(y) u(y) \right\} ds(y). \quad (2.37)$$

From the radiation condition, equation (2.31) and the decay of the Green's function we obtain $I_1 \rightarrow 0$ and $I_2 \rightarrow 0$ for $r \rightarrow \infty$. Hence,

$$\int_{\partial B_R} \left\{ \frac{\partial u}{\partial \nu}(y) G(x, y) - u(y) \frac{\partial G(x, y)}{\partial \nu(y)} \right\} ds(y) = I_2 - I_1 \rightarrow 0, \quad r \rightarrow \infty \quad (2.38)$$

and the proof is complete.

2.3.2 Rellich's lemma and uniqueness for the direct problem

To derive uniqueness of the solution to the direct scattering problem usually Rellich's lemma is the basic ingredient. Here, we need a version of Rellich's lemma for the layered background. We remark that the following result has been derived in a more general setting by Kristensson [9], who also gives references to related and earlier work. Since for our simple situation the proof may be simplified considerably, in the following we will show a simplified derivation of the relevant uniqueness

properties. Also, using the typical form (1.4) of the radiation condition we build a bridge towards the representation used in [4].

THEOREM 2 (Rellich's lemma) We consider a solution u to the Helmholtz equation (1.3) in $IR^2 \setminus \overline{\Omega}$ which satisfies the continuity conditions (1.6) and (1.7) on Γ_0 . If u satisfies

$$\lim_{r \rightarrow \infty} \int_{|x|=r} |u(x)|^2 ds = 0, \quad (2.39)$$

then $u = 0$ in $IR^2 \setminus \overline{\Omega}$.

Proof. We expand the function u in the exterior of some ball B_R into its Fourier Series

$$u(x) = \sum_{n=0}^{\infty} (a_n^{(+)}(r)e^{in\varphi} + a_n^{(-)}(r)e^{-in\varphi}), \quad \hat{x} = \frac{x}{|x|} \quad (2.40)$$

with respect to spherical harmonics, where we use polar coordinates $r = |x|$ and φ is the angel of x towards the x_1 – axis. The coefficients are given by

$$a_n^{(\pm)}(r) = \frac{1}{2\pi} \int_0^{2\pi} u(r\hat{x}) e^{\pm in\varphi} d\varphi \quad (2.41)$$

and satisfy the Parseval equality

$$\int_{\partial B_R} |u(x)|^2 ds = r \sum_{n=0}^{\infty} \sum_{\pm} |a_n^{(\pm)}(r)|^2. \quad (2.42)$$

We would like to twice differentiate the coefficients $a_n^{\pm}$ and apply a partial integration. However, for the second derivative and a partial integration we need to be careful with the behaviour of the second derivative of u at the interface Γ_0 . Existence and boundedness of the second derivative of $a_n^{\pm}$ can be derived by differentiation and partial integration applied to

$$a_{n,\varepsilon}^{(\pm)}(r) = \frac{1}{2\pi} \int_{\varepsilon}^{\pi-\varepsilon} u(r\hat{x}) e^{\pm in\varphi} d\varphi + \frac{1}{2\pi} \int_{\pi+\varepsilon}^{2\pi-\varepsilon} u(r\hat{x}) e^{\pm in\varphi} d\varphi \quad (2.43)$$

with a subsequent limit $\varepsilon \rightarrow 0$. Using the Laplace operator in polar coordinates

$$\Delta = \frac{1}{r} \frac{\partial}{\partial r} \left(r \frac{\partial}{\partial r} \right) + \frac{1}{r^2} \frac{\partial^2}{\partial \varphi^2} \quad (2.44)$$

and $\Delta u + \kappa^2 u = 0$ by partial integration we obtain

$$\left(r^2 \frac{\partial^2}{\partial r^2} + r \frac{\partial}{\partial r} + [\kappa^2(y)r^2 - n^2] \right) a_n^{(\pm)} = 0, \quad n \in \mathbb{N}. \quad (2.45)$$

Thus, the coefficients $a_n^{(\pm)}$ satisfy the Bessel equation and we have

$$a_n^{(\pm)} = \alpha^{(+)} H_n^{(1)}(\kappa r) + \alpha^{(-)} H_n^{(2)}(\kappa r) \quad (2.46)$$

with constants $\alpha^{(\pm)}$. Finally, we recall the asymptotic behaviour

$$H_n^{(1,2)}(t) = \sqrt{\frac{2}{\pi t}} e^{i(t - \frac{n\pi}{2} - \frac{\pi}{4})} \left\{ 1 + O\left(\frac{1}{t}\right) \right\}, \quad t \rightarrow \infty. \quad (2.47)$$

From (2.39), (2.42) and (2.47) we obtain $\alpha_n^{(\pm)} = 0$ and therefore $u \equiv 0$ outside of some sufficiently large sphere. By analyticity this yields $u = 0$ in $\mathbb{R}^2 \setminus \overline{\Omega}$ and the proof is complete.

The following uniqueness theorem has been published by Cutzach and Hazard [6] 1998 using the result of [9]. Here, we base it on the above version of Rellich's lemma.

THEOREM 3 (Uniqueness) The exterior Dirichlet problem for a buried object in the half space H_2 has at most one solution.

Proof. From the Dirichlet boundary condition and the existence of the normal derivative by equation (2.34) we obtain

$$\lim_{r \rightarrow \infty} \int_{|x|=r} \frac{1}{\kappa(y)} \left| \frac{\partial u(y)}{\partial \nu} \right|^2 + \kappa(y) |u(y)|^2 ds(y) = 0. \quad (2.48)$$

Taking the maximum of $\kappa(y)$ this yields

$$\lim_{r \rightarrow \infty} \int_{|x|=r} |u(y)|^2 ds(y) = 0 \quad (2.49)$$

By Rellich's lemma we derive $u = 0$ in $IR^2 \setminus \overline{\Omega}$ and the proof is complete.

2.3.3 An integral equation approach to solve the scattering problem

We search for u^s as a combined double and single-layer potential

$$u(x) = \int_{\partial\Omega} \left\{ \frac{\partial G(x, y)}{\partial \nu(y)} - iG(x, y) \right\} \varphi(y) ds(y), \quad x \in IR^2 \quad (2.50)$$

with exterior normal vector $\nu(y)$ at a point $y \in \partial\Omega$ and density $\varphi \in C(\partial\Omega)$. Since the Green's function G satisfies the Helmholtz equation, the radiation condition and the continuity properties, we will only need to satisfy the boundary condition (1.5) by an appropriate choice of the density φ .

We remark that in H_2 the singularity of G coincides with the singularity of the free-space Green's function $\frac{i}{4} H_0^{(1)}(k_2|x-z|)$ for the wave number κ_2 . Thus the continuity properties and jump-relations of the single- and double-layer potentials (see [3]) can be transferred to our case and we obtain for the boundary values of u the formula

$$u_{\pm}(x) = \pm \frac{1}{2} \varphi(x) + \int_{\partial\Omega} \left\{ \frac{\partial G(x, y)}{\partial \nu(y)} - iG(x, y) \right\} \varphi(y) ds(y), \quad x \in \partial\Omega. \quad (2.51)$$

With the help of the single-layer potential operator

$$(S\varphi)(x) := 2 \int_{\partial\Omega} G(x, y) \varphi(y) ds(y), \quad x \in IR^2 \quad (2.52)$$

and the double-layer potential operator

$$(K\varphi)(x) := 2 \int_{\partial\Omega} \frac{\partial G(x, y)}{\partial \nu(y)} \varphi(y) ds(y), \quad x \in IR^2 \quad (2.53)$$

we obtain the boundary integral equation

$$(I + K - iS) \varphi = -2u^i \text{ on } \partial\Omega. \quad (2.54)$$

2.3.4 Solution of the integral equation via the Nyström Method

Equation (2.54) is a singular boundary integral equation. In order to solve such an equation, an efficient numerical method is needed. Since it seems to be safe to state that the boundary curves in most practical applications are either analytic or piecewise analytic with corners, we restrict our attention to the analytic case where we recommend the Nyström method based on appropriately weighted numerical quadratures on an equidistant mesh.

We first describe the necessary parameterization of the integral equation (2.54) in the two-dimensional case. We assume that the boundary curve $\partial\Omega$ possesses a regular analytic and 2π -periodic parametric representation of the form

$$x(t) = (x_1(t), x_2(t)), \quad 0 \leq t \leq 2\pi, \quad (2.55)$$

in counterclockwise orientation satisfying $[x'_1(t)]^2 + [x'_2(t)]^2 > 0$ for all t . Then by straightforward calculations using

$$H_1^{(1)} = -H_0^{(1)}, \quad (2.56)$$

$$\frac{\partial G(x, y)}{\partial \nu(y)} = \text{grad}_y G(x, y) \cdot \nu(y), \quad (2.57)$$

$$ds(y) = \sqrt{[x'_1(\tau)]^2 + [x'_2(\tau)]^2} d\tau \quad (2.58)$$

we transform (2.54) into the parametric form

$$\psi(t) - \int_0^{2\pi} \{L(t, \tau) + i\eta M(t, \tau) + N(t, \tau)\} \psi(\tau) d\tau = g(t), \quad 0 \leq t \leq 2\pi \quad (2.59)$$

where we have set $\psi(t) := \varphi(x(t))$, $g(t) := -2u^i(x(t))$ and the kernels are given by

$$L(t, \tau) := \frac{ik_2}{2} \{x'_2(\tau)[x_1(\tau) - x_1(t)] - x'_1(\tau)[x_2(\tau) - x_2(t)]\} \frac{H_1^{(1)}(k_2 r(t, \tau))}{r(t, \tau)} \quad (2.60)$$

$$M(t, \tau) := \frac{i}{2} H_0^{(1)}(k_2 r(t, \tau)) \sqrt{[x'_1(\tau)]^2 + [x'_2(\tau)]^2} \quad (2.61)$$

$$N(t, \tau) = - \left\{ (T_1(t, \tau)x_2'(\tau) - T_2(t, \tau)x_1'(\tau) - i \left[\frac{1}{2\pi} \int_{-\infty}^{\infty} \frac{1}{2\gamma_2} \frac{\gamma_2 - \gamma_1}{\gamma_1 + \gamma_2} e^{\gamma_2(x_2(t) + x_2(\tau))} e^{iV(x_1(t) - x_1(\tau))} dV \right] \right\} \\ \times 2\sqrt{[x_1'(\tau)]^2 + [x_2'(\tau)]^2} \quad (2.62)$$

for $t \neq \tau$. Here, we have set

$$r(t, \tau) := \sqrt{[x_1(t) - x_1(\tau)]^2 + [x_2(t) - x_2(\tau)]^2} \quad (2.63)$$

and

$$T_1(t, \tau) = I_1(x(t), x(\tau)), \quad (2.64)$$

$$T_2(t, \tau) = I_2(x(t), x(\tau)) \quad (2.65)$$

which are the parametric representations of the integrals

$$I_1(x, y) := - \frac{i}{2\pi} \int_{-\infty}^{\infty} \frac{V}{2\gamma_2} \frac{\gamma_2 - \gamma_1}{\gamma_1 + \gamma_2} e^{\gamma_2(x_2 + y_2)} e^{iV(x_1 - y_1)} dV \quad (2.66)$$

$$I_2(x, y) := \frac{1}{2\pi} \int_{-\infty}^{\infty} \frac{1}{2} \frac{\gamma_2 - \gamma_1}{\gamma_1 + \gamma_2} e^{\gamma_2(x_2 + y_2)} e^{iV(x_1 - y_1)} dV \quad (2.67)$$

on the boundary $\partial\Omega$, respectively. It should be noted that we used the second part of (2.13) to express the Green's function in (2.54) since equation (2.54) is valid on $\partial\Omega$ where both x_2 and y_2 are in the lower half-space H_2 .

It is observed that the kernels L and M have logarithmic singularities at $t = \tau$. Hence for their proper numerical treatment [4], we split the kernels into

$$L(t, \tau) = L_1(t, \tau) \ln(4 \sin^2 \frac{t - \tau}{2}) + L_2(t, \tau), \quad (2.68)$$

$$M(t, \tau) = M_1(t, \tau) \ln(4 \sin^2 \frac{t - \tau}{2}) + M_2(t, \tau), \quad (2.69)$$

where

$$L_1(t, \tau) := \frac{k_2}{2\pi} \{x_2'(\tau)[x_1(\tau) - x_1(t)] - x_1'(\tau)[x_2(\tau) - x_2(t)]\} \frac{J_1(k_2 r(t, \tau))}{r(t, \tau)}, \quad (2.70)$$

$$L_2(t, \tau) := L(t, \tau) - L_1(t, \tau) \ln(4 \sin^2 \frac{t - \tau}{2}), \quad (2.71)$$

$$M_1(t, \tau) := -\frac{1}{2} J_0(k_2 r(t, \tau)) \sqrt{[x'_1(\tau)]^2 + [x'_2(\tau)]^2}, \quad (2.72)$$

$$M_2(t, \tau) := M(t, \tau) - M_1(t, \tau) \ln(4 \sin^2 \frac{t - \tau}{2}) \quad (2.73)$$

The kernels L_1, L_2, M_1 and M_2 turn out to be analytic. In particular, using series expansions of Bessel and Neumann functions [4] we can deduce the diagonal terms

$$L_2(t, t) = L(t, t) = \frac{1}{2\pi} \frac{x'_1(t)x''_2(t) - x'_2(t)x''_1(t)}{[x'_1(t)]^2 + [x'_2(t)]^2} \quad (2.74)$$

$$M_2(t, t) = \left\{ \frac{i}{2} - \frac{C}{\pi} - \frac{1}{2\pi} \ln \left(\frac{k^2}{4} \{ [x'_1(t)]^2 + [x'_2(t)]^2 \} \right) \right\} \sqrt{[x'_1(t)]^2 + [x'_2(t)]^2} \quad (2.75)$$

for $0 \leq t \leq 2\pi$. We note that despite the continuity of the kernel L , for numerical accuracy it is advantageous to separate the logarithmic part of L since the derivatives of L fail to be continuous at $t = \tau$. Hence, we have to numerically solve an integral equation of the form

$$\psi(t) - \int_0^{2\pi} K(t, \tau) \psi(\tau) d\tau = g(t), \quad 0 \leq t \leq 2\pi, \quad (2.76)$$

where the kernel can be written in the form

$$K(t, \tau) = K_1(t, \tau) \ln(4 \sin^2 \frac{t - \tau}{2}) + K_2(t, \tau) + K_3(t, \tau) \quad (2.77)$$

with analytic functions

$$K_1(t, \tau) = L_1(t, \tau) + iM_1(t, \tau), \quad (2.78)$$

$$K_2(t, \tau) = L_2(t, \tau) + iM_2(t, \tau) \quad (2.79)$$

$$K_3(t, \tau) = N(t, \tau) \quad (2.80)$$

and with an analytic right hand side g . Here we wish to point out that it is essential to split off the logarithmic singularity in a fashion which preserves the 2π -periodicity for the kernels K_1 and K_2 .

The Nyström method consists in the straightforward approximation of the integrals by quadratures formulas. In our case, for the 2π -periodic integrands, we choose an equidistant set of knots $t_j := \pi j / n$, $j = 0, \dots, 2n-1$, and use the quadrature rule

$$\int_0^{2\pi} \ln(4 \sin^2 \frac{t-\tau}{2}) f(\tau) d\tau \approx \sum_{j=0}^{2n-1} R_j^{(n)}(t) f(t_j), \quad 0 \leq t \leq 2\pi, \quad (2.81)$$

with the quadrature weights given by

$$R_j^{(n)}(t) := -\frac{2\pi}{n} \sum_{m=1}^{n-1} \frac{1}{m} \cos m(t-t_j) - \frac{\pi}{n^2} \cos n(t-t_j), \quad j = 0, \dots, 2n-1 \quad (2.82)$$

and the trapezoidal rule

$$\int_0^{2\pi} f(\tau) d\tau \approx \frac{\pi}{n} \sum_{j=0}^{2n-1} f(t_j). \quad (2.83)$$

Both these numerical integration formulas are obtained by replacing the integrand f by its trigonometric interpolation polynomial and then integrating exactly. The errors for the quadrature rules decrease at least exponentially when the number $2n$ of knots is increased. Of course, it is also possible to use quadrature rules different from (2.81) and (2.83) obtained from other approximations for the integrand f . However, due to their simplicity and high approximation order, the application of (2.81) and (2.83) is recommended. [4]

In the Nyström method, the integral equation (2.76) is replaced by the approximating equation

$$\psi^{(n)}(t) - \sum_{j=0}^{2n-1} \left\{ R_j^{(n)}(t) K_1(t, t_j) + \frac{\pi}{n} K_2(t, t_j) + \frac{\pi}{n} K_3(t, t_j) \right\} \psi^{(n)}(t_j) = g(t) \quad 0 \leq t \leq 2\pi \quad (2.84)$$

Equation (2.84) is obtained from (2.76) by applying the quadrature rule (2.81) to $f = K_1(t, \cdot) \psi$ and (2.83) to $f = K_2(t, \cdot) \psi$ and $f = K_3(t, \cdot) \psi$. The solution of (2.84)

reduces to solving a finite dimensional linear system. In particular for any solution of (2.84) the values $\psi_i^{(n)} = \psi^{(n)}(t_i)$, $i = 0, \dots, 2n-1$, at the quadrature points trivially satisfy the linear system

$$\psi_i^{(n)}(t) - \sum_{j=0}^{2n-1} \left\{ R_{|i-j|}^{(n)}(t) K_1(t, t_j) + \frac{\pi}{n} K_2(t_i, t_j) + \frac{\pi}{n} K_3(t_i, t_j) \right\} \psi_j^{(n)} = g(t_i) \quad (2.85)$$

for $i = 0, \dots, 2n-1$, where

$$R_j^{(n)} := R_j^{(n)}(0) = -\frac{2\pi}{n} \sum_{m=1}^{n-1} \frac{1}{m} \cos \frac{mj\pi}{n} - \frac{(-1)^j \pi}{n^2}, \quad j = 0, \dots, 2n-1 \quad (2.86)$$

Conversely, given a solution $\psi_i^{(n)}$, $i = 0, \dots, 2n-1$, of the system (2.85) the function $\psi^{(n)}$ defined by

$$\psi^{(n)}(t) := \sum_{j=0}^{2n-1} \left\{ R_j^{(n)}(t) K_1(t, t_j) + \frac{\pi}{n} K_2(t, t_j) + \frac{\pi}{n} K_3(t, t_j) \right\} \psi_j^{(n)} + g(t) \quad 0 \leq t \leq 2\pi \quad (2.87)$$

is readily seen to satisfy the approximating equation (2.84). The formula (2.87) may be viewed as a natural interpolation of the values $\psi_i^{(n)}$, $i = 0, \dots, 2n-1$ at the quadrature points to obtain the approximating function $\psi^{(n)}$ and goes back to Nyström.

3 A POINT-SOURCE METHOD FOR RECONSTRUCTIONS

The point-source method in its simplest version reconstructs the scattered field u^s near the scatterer and tries to find the boundary $\partial\Omega$ as a minimum curve of the total field u , which is known to vanish on $\partial\Omega$. The reconstruction of u^s is based on the reciprocity relation and on some back-projection operator introduced by Potthast R.[12],[13],[14]. We will now introduce the necessary tools for the layered background and describe the steps in which a reconstruction can be obtained.

3.1 The Reciprocity Relation

Reciprocity describes the time-reversal symmetry in scattering, i.e. the role of sender and receiver may be exchanged. Mathematically, reciprocity can be proven by an application of the second Green's theorem using the partial differential equation under consideration, the boundary condition and the radiation conditions. For scattering of the point-source $G(.,z)$ with source-point $z \in \mathbb{R}^2 \setminus \Omega$ by the obstacle Ω we will denote the scattered field by $G^s(.,z)$.

THEOREM 4 For scattering of point-sources $G(.,z)$ by the scatterer Ω we have

$$G^s(x,z) = G^s(z,x) \quad (3.1)$$

for all $x, z \in \mathbb{R}^2 \setminus \overline{\Omega}$.

Proof. By an application of Green's second theorem we obtain

$$\int_{\partial\Omega} \left\{ G^s(y,z) \frac{\partial G^s(x,y)}{\partial \nu(y)} - \frac{\partial G^s(y,z)}{\partial \nu(y)} G^s(x,y) \right\} ds(y) = 0 \quad (3.2)$$

for $x, z \in \mathbb{R}^2 \setminus \overline{\Omega}$. Green's formula applied to $u = G^s(.,z)$ reads

$$G^S(x, z) = \int_{\partial\Omega} \left\{ G^S(y, z) \frac{\partial G(x, y)}{\partial \nu(y)} - \frac{\partial G^S(y, z)}{\partial \nu(y)} G(x, y) \right\} ds(y), \quad x, z \in \mathbb{R}^2 \setminus \overline{\Omega} \quad (3.3)$$

Adding the two equations and using the boundary condition we obtain

$$G^S(x, z) = \int_{\partial\Omega} G^S(y, z) \frac{\partial G^{tot}(x, y)}{\partial \nu(y)} ds(y), \quad x, z \in \mathbb{R}^2 \setminus \overline{\Omega} \quad (3.4)$$

with $G^{tot} = G + G^S$. Again from Green's theorem we derive

$$\int_{\partial\Omega} \left\{ G(y, z) \frac{\partial G(x, y)}{\partial \nu(y)} - \frac{\partial G(y, z)}{\partial \nu(y)} G(x, y) \right\} ds(y) = 0 \quad (3.5)$$

and thus

$$G^S(x, z) = - \int_{\partial\Omega} G(y, z) \frac{\partial G^{tot}(x, y)}{\partial \nu(y)} ds(y), \quad x, z \in \mathbb{R}^2 \setminus \overline{\Omega} \quad (3.6)$$

Finally, using the boundary condition $G^S(y, z) = -G(y, z)$ we obtain (3.1).

3.2 Reconstruction algorithm

We measure the scattered field u^S in $n + 1$ points

$$\Lambda_n = \left\{ \left(a + \frac{(b-a)j}{n}, h \right), \quad j = 0, \dots, n \right\} \quad (3.7)$$

at height h above the ground, i.e. above the line Γ_0 . Consider a point $x \in H_2 \setminus \overline{\Omega}$. We choose a domain of approximation $G_x \subset H_2$ such that $\Omega \subset G_x$ and $x \notin \overline{G_x}$, see Figure 3.1. Here, we assume that the homogeneous interior Dirichlet problem for the Helmholtz equation in the domain G_x does have only the trivial solution $u \equiv 0$.

To construct the back-projection operator of the point source method we first approximate the Green's function $G(\cdot, x)$ on ∂G_x by a superposition of Green's function $G(\cdot, y)$ with

$$y \in \Lambda := \{(a + (b - a)t, h), \quad t \in [0, 1]\} \quad (3.8)$$

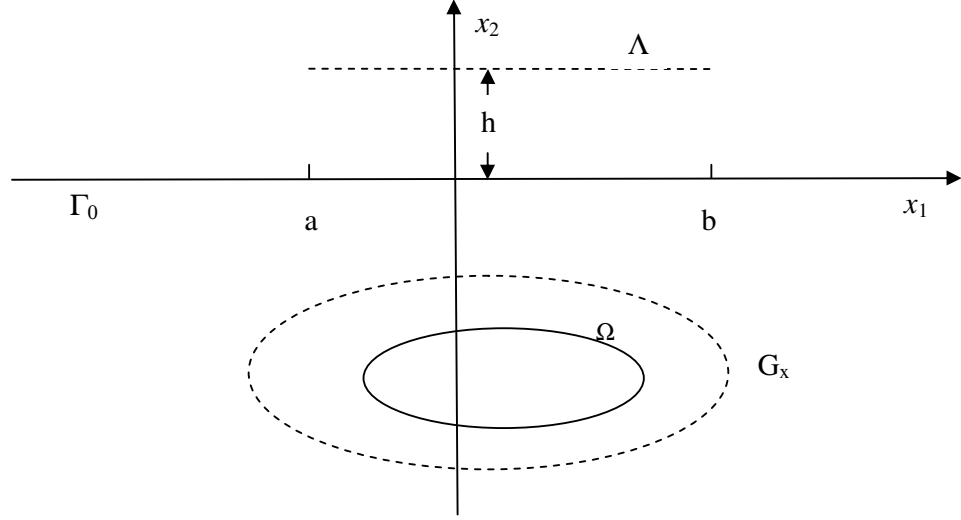


Figure 3.1 Illustration for the domain of approximation used for the point source method.

i.e. we construct a density $g(., x) \in L^2(\Lambda)$ such that

$$G(\xi, x) \approx \int_{\Lambda} G(\xi, y) g(y, x) ds(y), \quad \xi \in \partial G_x. \quad (3.9)$$

First, by the boundedness of the interior Dirichlet problem for the Helmholtz equation in Ω , this approximation is also valid on the boundary of the domain $\Omega \subset G_x$.

Second, by the linearity and boundedness of the scattering problem, we obtain an approximation

$$G^s(\xi, x) \approx \int_{\Lambda} G^s(\xi, y) g(y, x) ds(y), \quad \xi \in \mathbb{R}^2 \setminus \overline{\Omega} \quad (3.10)$$

for the corresponding scattered fields. By the use of the reciprocity relation given by Theorem 4 we derive

$$G^s(x, \xi) \approx \int_{\Lambda} G^s(y, \xi) g(y, x) ds(y), \quad \xi \in \mathbb{R}^2 \setminus \overline{\Omega}. \quad (3.11)$$

Finally, we use this approximation for our incident wave $G(.,z)$ with source point $\xi = z$. Our data is given by $G^S(y, z)$ for $y \in \Lambda$. For each point $x \in \mathcal{G}$ on a grid $\mathcal{G}$ covering the space where the unknown object Ω might be located, we calculate $G_{rec}^S(x, z)$ by the right-hand side of (3.11). For the following images in Figure 4.1 we will plot the total field

$$G(x, z) + G_{rec}^S(x, z), \quad x \in \mathcal{G}, \quad (3.12)$$

or its reconstruction, respectively. The unknown object can be found searching the minimum points of the total field on the grid $\mathcal{G}$.

4. NUMERICAL RESULTS

In this section we present some numerical results in order to have an idea of the applicability as well as the effectiveness of the method given above. For the numerical treatment of the inverse problem we first solved the direct problem by boundary integral equations. Then, the scattered field was obtained by evaluation of the corresponding boundary layer potential on the measurement line. For simplicity, we used a Nyström method ignoring the singularity which yields linear convergence for the solution of the boundary integral equation. For the Green's function G we employed a straightforward implementation of the Fourier transform using the trapezoidal rule. All programs were both written in MATLAB and in C++ and run on regular PCs. The C++ code took less than a minute to produce the forward data. For that reason, the code for C++ is given in Appendix.

Second, for each point z in a grid $\mathcal{G}$ we solved the integral equation (3.9) by the Tikhonov regularization

$$g_\alpha(.,z) := (\alpha I + H^* H)^{-1} H^* G(.,z). \quad (4.1)$$

Here, H is the operator

$$(H\varphi)(x) := \int_{\Lambda} G(x,y)\varphi(y)ds(y), \quad x \in \partial G_x, \quad (4.2)$$

H^* is the L^2 -adjoint of H and α is the regularization parameter. Then, we calculated

$$G_\alpha^S(z) := \int_{\Lambda} G^S(y,\xi)g(y,z)ds(y) \quad (4.3)$$

following equation (3.11). In C++ the operation for each $z \in \mathcal{G}$ could be performed in fractions of a second and the time for the algorithm was proportional to the resolution of the images which are shown in the figures.

For some simple examples we consider an elliptical body with boundary

$$\partial\Omega = \{(\cos(t) + 1, 0.5 * \sin(t) - 3), \quad t \in [0, 2\pi]\}. \quad (4.4)$$

The first image of Figure 4.1 shows a plot of the total field for scattering of a point source $G(.,z)$, $z = (-5, 10)$, located in the upper half space with $\kappa_2=3$. The second image is just a portion of the first image. The last image demonstrates the reconstructed total field using the point source method with some domain of approximation given by translations of $\partial\Omega = \{(3 * \cos(t) + 1, 1.5 * \sin(t) - 1.6), \quad t \in [0, 2\pi]\}$ by x . We used 20 measurements of the scattered field on a line Λ from $a = -8$ to $b = 8$ and $h = 1$. Due to the fixed orientation of the domain of approximation we find only the upper part of the boundary $\partial\Omega$.

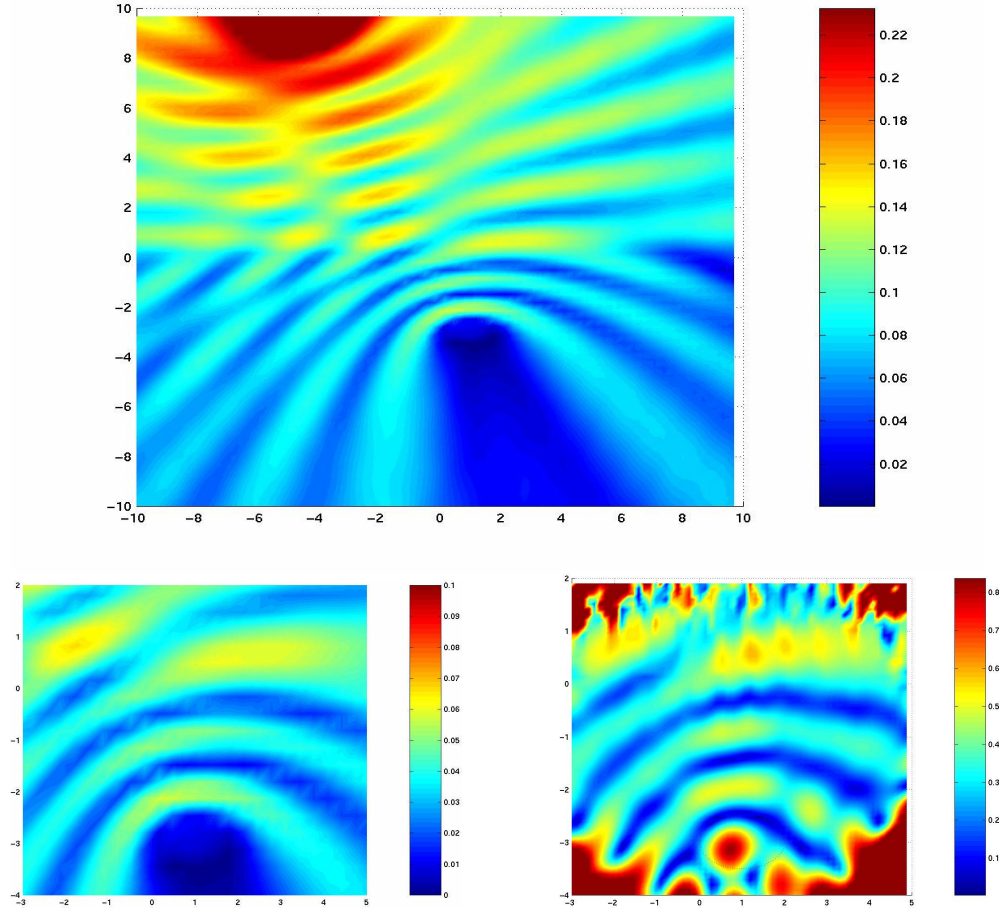


Figure 4.1 Plot of the original and reconstructed total field for scattering of a point source $G(.,z)$, $z = (-10, 10)$, located in the upper half space with $\kappa_1 = 2$ by an obstacle $\partial\Omega = \{(\cos(t) + 1, 0.5 * \sin(t) - 3), \quad t \in [0, 2\pi]\}$ buried in a lower half space with $\kappa_2 = 3$

Figure 4.2 illustrates the total and the reconstructed fields respectively, for the obstacle given above and the source location $z = (5, 10)$.

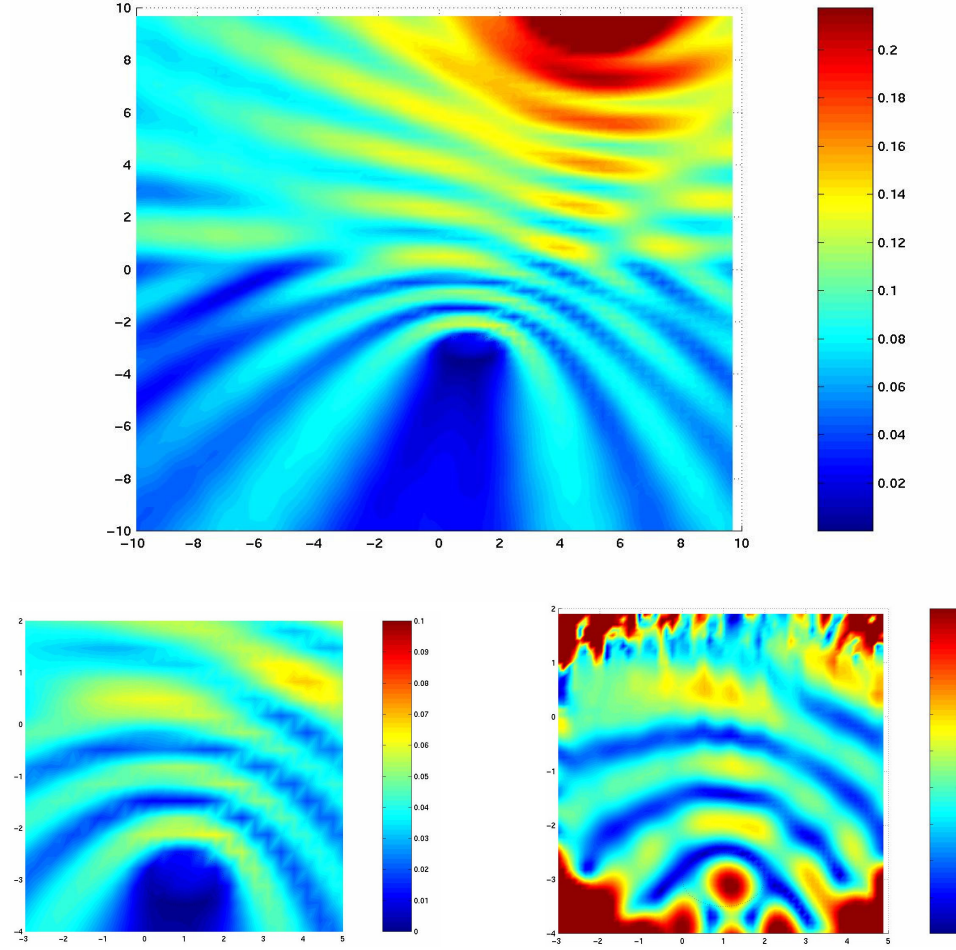


Figure 4.2 Plot of the original and reconstructed total field for scattering of a point source $G(., z)$, $z = (5, 10)$, located in the upper half space with $\kappa_1 = 2$ by an obstacle used in Figure 4.1 buried in a lower half space with $\kappa_2 = 3$

We also performed some tresholding operation to the images to extract the minima and a logical ‘and’ operation to combine the results for two different illuminations. A result is shown in Figure 4.3. Logical image operations for obstacle reconstructions have been used by Luke and Potthast [10] for homogeneous backgrounds. Here, it would be of high interest to employ the same techniques for the two-layered background.

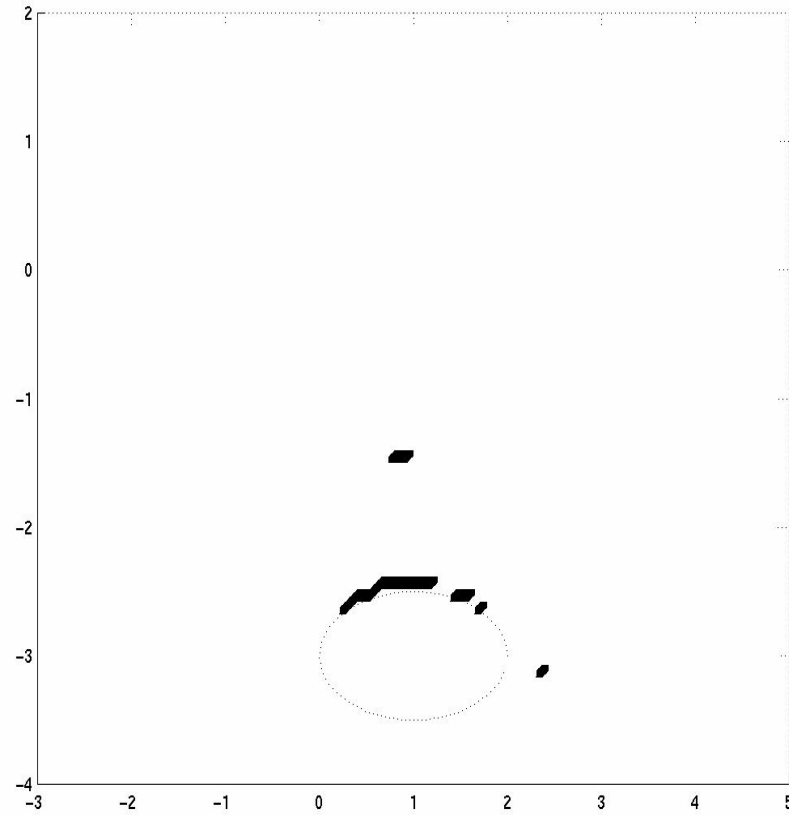
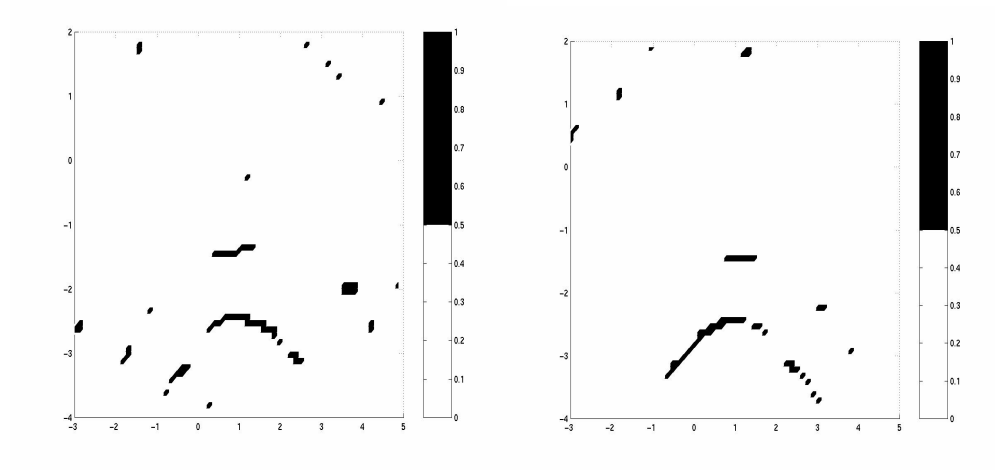


Figure 4.3 The first two images are thresholded versions of the reconstructed fields from the Figures 4.1 and 4.2. The third image is obtained from the first two images by a logical ‘and’ operation.

5. CONCLUSION

In this thesis, we proposed a method to reconstruct the location and shape of a buried obstacle which is illuminated by a source located in the half-space not containing the body. We restricted our attention to the simplest case of two dimensions, which serves as a first step to the more involved and computationally much more expensive three-dimensional realistic situation. However, the treatment of our method in two-dimensions is also valuable, since two-dimensional problems appear in real-life applications with some symmetry along a geometrical axis such as determination of pipelines...etc. It should also be noted that, in order to use the Nyström method in Section 4, we assumed that the boundary of the obstacle can be represented parametrically.

In the numerical implementation of the method, the time for the algorithm was proportional to the resolution of the reconstruction images. For an effective application of the algorithm, adaptive strategies need to be employed, which leads to further research. Here, we restricted our attention to the applicability of the method for reconstructions. Our method yields good resolution with a less number of measurements compared to other techniques in the open literature.

Since we demonstrated our method for a perfectly conducting obstacle, the method can be generalized for different boundary conditions.

REFERENCES

- [1] **Akduman, I. and Alkumru, A.**, 1995. A generalized ART algorithm for inverse scattering problems related to buried cylindrical bodies, *Inverse Problems*, **11**, 1125-1136.
- [2] **Colton, D. Coyle, J. and Monk, P.**, 2000. Recent Developments in Inverse Acoustic Scattering Theory, *SIAM Review* **42**, No. 3, 369-414.
- [3] **Colton, D. and Kress, R.**, 1983. *Integral Equation Methods in Scattering Theory*, Wiley, New York.
- [4] **Colton, D. and Kress, R.**, 1998. *Inverse Acoustic and Electromagnetic Scattering Theory*, Springer Berlin, Heidelberg, New York.
- [5] **Coyle, J.**, 2000. An inverse electromagnetic scattering problem in a two-layered background, *Inverse Problems*, **16**, 275-292.
- [6] **Cutzach, P.-M. and Hazard, C.**, 1998. Existence, Uniqueness and Analyticity Properties for Electromagnetic Scattering in a Two-layered Medium, *Math. Meth. Appl. Sci.*, **21**, 433-461.
- [7] **Duchene, B. and Lesselier, D.**, 1996. Ch.10, Review of Radio Science 1993-1996, pp. 235-268, Ed. R. Stone, Oxford University Press, Oxford.
- [8] **Duchene, B., Lesselier, D. and Tabbara, W.**, 1987 Acoustical imaging of 2D fluid targets buried in half-space: A diffraction tomography approach, *IEEE Trans. Ultrason. Ferroel. Freq. Control* **UFFFC**, **34**, 540-549.
- [9] **Kristensson, G.**, 1980. A uniqueness theorem for Helmholtz equation: penetrable media with an infinite interface, *SIAM J. Math. Anal.*, **11**, 1104-1117.
- [10] **Luke, D.R. and Potthast, R.**, "Limited aperture image processing in inverse scattering", (submitted for publication).
- [11] **Marcuwitz, N. and Felsen, L.B.**, 1973. *Radiation and Scattering of Waves*. Englewood Cliffs, NJ: Prentice-Hall.
- [12] **Potthast, R.**, 1996. A fast new method to solve inverse scattering problems, *Inverse Problems*, **12**, 731-742.
- [13] **Potthast, R.**, 1998, A point-source method for inverse acoustic and electromagnetic obstacle scattering problems, *IMA Jour. Appl. Math.*, **61**, 119-140.

- [14] **Potthast, R.**, 2001, Point sources and multipoles in inverse scattering, Chapman & Hall Research Notes, London.

BIOGRAPHY

Tolga Ulaş Gürbüz was born in Burdur, Turkey in 1980. He received his primary education in Burdur Şeker Primary School and secondary education in Burdur Anatolian High School, Eskişehir Science School and Burdur Cumhuriyet High School. He graduated from Yeditepe University Department of Electrical and Electronics Engineering in 2001. He is currently a student of the M.Sc. program in Istanbul Technical University Electronics and Communication Department.

APPENDIX

dirburied_1.c: Source Code for the direct problem

```
//-----  
//  
//      dirburied_1.c  
//  
// Program to calculate the acoustic scattered and total  
// field for scattering by a PEC(sound-soft obstacle) in two  
// dimensions buried in a lower half space.  
//  
//  
//  
// Parameters with typical values:  
//  
// n=20;    number of points on the boundary of the obstacle  
//          to solve an integral equation of the first kind  
// nnf=20;   number of points on the measurement line  
// z1=-10.0; x-coordinate of source point of incident field  
// z2=10.0;  y-coordinate of source point of incident field  
// k1=1.0;   wave number for upper half-space  
// k2=2.0;   wave number for lower half-space  
// a=-4.0;   parameters for the measurement line  
// b=4.0;    defined by  $\{(y, mh), a \leq y \leq b\}$   
// mh=1.0;   height of measurement line above the ground  
// plotpar=1; if plotpar=1 the field total field u will be  
//            calculated and saved in u_re_c.mat, u_im_c.mat  
//            if plotpar=<>1 only the values on the measure-  
//            ment line nf_c.mat will be produced.  
// alpha=1e-8; regularization parameter for the integral  
//            equation of the first kind.  
//  
// If a plot of u is prepared, the following parameters are  
// needed:  
//  
// a1=-10;   left and right, lower and upper boundary  
// b1=10;    of the sampling domain  
// a2=-10;    $\{(x, y); a1 \leq x \leq b1, a2 \leq z \leq b2\}$   
// b2=10;  
// n1=41;    number of sampling points in x-direction  
// n2=41;    number of sampling points in y-direction  
//  
//
```

```

//
//-----

#include <iostream.h>
#include "matrix-classic.h"
#include "supp.h"
#include "math.h"
#include "fundsol.h"

//-----
//
//   main()
//
//-----

main(int argc, char *argv[]){

    //-----
    // Deklarations
    //-----
    int n,nnf,plotpar;
    double z1,z2,k1,k2,a,b,mh,alpha;

    //-----
    // Parameters (see explanaitons above)
    //-----

    n=20;
    nnf=20;
    z1=-10.0;
    z2=10.0;
    k1=1.0;
    k2=2.0;
    a=-4.0;
    b=4.0;
    mh=1.0;
    plotpar=1;
    alpha=1e-8;

    // if plotpar = 1 the following parameters are needed
    int a1,b1,a2,b2,n1,n2;
    a1=-10;
    b1=10;
    a2=-10;
    b2=10;
    n1=21;
    n2=21;

    //-----

```

```

// Greetings and parameter output
//-----
cout << "-----\n";
system("date");
cout << "\n";
cout << "This is the program d i r b u r i e d _ 1 \n";
cout << "Calculating the scattered field for scattering\n";
cout << "of a point source by some buried obstacle\n";

cout << "\n";
cout << " n      = " << n << "\n";
cout << " nnf    = " << nnf << "\n";
cout << " z1     = " << z1 << "\n";
cout << " z2     = " << z2 << "\n";
cout << " k1     = " << k1 << "\n";
cout << " k2     = " << k2 << "\n";
cout << " a      = " << a << "\n";
cout << " b      = " << b << "\n";
cout << " mh     = " << mh << "\n";
cout << " plotpar = " << plotpar << "\n";
cout << " alpha  = " << alpha << "\n";

//-----
// points on the boundary of the domain
//-----

double h,t1;
realVector xd1,xd2;
xd1=realVector(n);
xd2=realVector(n);
h=2*pi/(1.0*n);
for(int j=0; j<n; j++){
    t1=h*j;
    *(xd1.start + j) = cos(t1) + 1;
    *(xd2.start + j) = 0.5*sin(t1)-3;
}
saveRealVectorMatlabFormat( xd1, "xd1_c.mat");
saveRealVectorMatlabFormat( xd2, "xd2_c.mat");

//-----
// auxilliary domain for IGL of the first kind
//-----

realVector xg1,xg2;
xg1=realVector(n);
xg2=realVector(n);
double fac=0.9;
for(int j=0; j<n; j++){
    t1=h*j;
    *(xg1.start + j) = fac*cos(t1) + 1;

```

```

        *(xg2.start + j) = fac*0.5*sin(t1)-3;
    }
    saveRealVectorMatlabFormat( xg1, "xg1_c.mat");
    saveRealVectorMatlabFormat( xg2, "xg2_c.mat");

//-----
// setup matrix for direct problem
//-----

cout << "\n";
cout << "- setting up matrix\n";
matrixClassic mat;
mat=matrixClassic(n,n);
for (int j=0; j<n; j++){
    for (int l=0; l<n; l++){
        *(mat.start + n*j + l) =
            fundsol(*(xd1.start+j), *(xd2.start+j),
                    *(xg1.start+l), *(xg2.start+l),
                    k1,k2);
    };
};
// saveMatrixClassicMatlabFormat(mat,"mat_re_c.mat", "mat_im_c.mat");

//-----
// calculate right-hand side
//-----

cout << "- calculating the right-hand side\n";
complexVector rs;
rs=complexVector(n);
for (int j=0; j<n; j++){
    *(rs.start + j) =
        (-1.0)*fundsol(*(xd1.start+j), *(xd2.start+j),
                        z1, z2, k1, k2);
};

// saveComplexVectorMatlabFormat( rs, "rs_re_c.mat", "rs_im_c.mat");

//-----
// solve IGL of the first kind using Tikhonov regularization
//-----

matrixClassic eye,tik,tikInv,matAdj;
eye=identityMatrixClassic(n);
tik=matrixClassic(n,n);
tikInv=matrixClassic(n,n);
matAdj=matrixClassic(n,n);
complexVector den;

```

```

matAdj=adj(mat);
tik=alpha*eye + matAdj*mat;
tikInv=gauss(tik);
den=tikInv*matAdj*rs;

// saveComplexVectorMatlabFormat( den, "den_re_c.mat", "den_im_c.mat");

//-----
// calculate the scattered field on the measurement line
//-----

cout << "- calculating the scattered field\n";

double x1,x2;
complexVector nf;
nf=complexVector(nnf);
for(int j=0; j<nnf; j++){
    x1=a+(b-a)/nnf*j;
    x2=mh;
    *(nf.start+j)=complex(0.0,0.0);
    for(int l=0; l<n; l++){
        *(nf.start+j)=*(nf.start+j)+ (*(den.start+l))*
            fundsol(x1, x2, *(xg1.start+l), *(xg2.start+l), k1, k2);
    }
}

saveComplexVectorMatlabFormat( nf, "nf_re_c.mat", "nf_im_c.mat");

//-----
// prepare plot of total field
//-----

if( plotpar == 1){
    cout << "- calculating total field for plot\n";

    cout << "\n";
    cout << " a1  = " << a1 << "\n";
    cout << " b1  = " << b1 << "\n";
    cout << " a2  = " << a2 << "\n";
    cout << " b2  = " << b2 << "\n";
    cout << " n1  = " << n1 << "\n";
    cout << " n2  = " << n2 << "\n";

    double h1,h2;
    h1=(1.0*b1-1.0*a1)/(1.0*n1);
    h2=(1.0*b2-1.0*a2)/(1.0*n2);

    cout << " h1  = " << h1 << "\n";
    cout << " h2  = " << h2 << "\n";
}

```

```

matrixClassic inc, us, u;
inc=matrixClassic(n1,n2);
us=matrixClassic(n1,n2);
u=matrixClassic(n1,n2);

for(int j=0; j<n1; j++){
    for(int l=0; l<n2; l++){
        x1=a1+h1*j;
        x2=a2+h2*l;

        *(inc.start+n2*j+l)=fundsol(x1,x2,z1,z2,k1,k2);
        *(us.start+n2*j+l)=complex(0.0,0.0);
        for(int xi=0; xi<n; xi++){
            *(us.start+n2*j+l)=*(us.start+n2*j+l) +
                (*(den.start+xi))* fundsol(x1,x2,
                    *(xg1.start+xi),*(xg2.start+xi),k1,k2);
        }
        *(u.start+n2*j+l)=*(inc.start+n2*j+l) +
            *(us.start+n2*j+l);
    }
}
cout << "\n";
saveMatrixClassicMatlabFormat(inc,"inc_re_c.mat", "inc_im_c.mat");
cout << "- saved inc_re_c.mat and inc_im_c.mat\n";
saveMatrixClassicMatlabFormat(us,"us_re_c.mat", "us_im_c.mat");
cout << "- saved us_re_c.mat and us_im_c.mat\n";

realVector ex,ey;
ex=realVector(n1);
ey=realVector(n2);
for(int j=0; j<n1; j++){
    *(ex.start+j)=a1+h1*j;
}
for(int l=0; l<n2; l++){
    *(ey.start+l)=a2+h2*l;
}

//-----
// save for further use (Matlab format)
//-----

saveMatrixClassicMatlabFormat(u,"u_re_c.mat", "u_im_c.mat");
cout << "- saved u_re_c.mat and u_im_c.mat\n";
saveRealVectorMatlabFormat( ex, "ex_c.mat");
cout << "- saved ex_c.mat\n";
saveRealVectorMatlabFormat( ey, "ey_c.mat");
cout << "- saved ey_c.mat\n";
cout << " a plot of abs(u) will be produced using makeimage.m\n";
cout << " via the matlab system\n";

```

```

        // plotpar condition ends
    }

    cout << "\n";
    system("date");
    cout << "-----\n";

    //-----
    //
    //-----

    // End
}

```

dirburied_1.c: Source Code for the inverse problem

```

//-----
//
//      invburied_1.c
//
// Program to calculate the reconstructed total EM (acoustic)
// field for scattering by PEC(sound-soft) obstacle in two
// dimensions buried in a lower half space from measurements
// on a line above the ground.
//
//
// Parameters with typical values:
//
// nnf=20;   number of points on the measurement line
//           it should coincide with the vector nf of
//           measurements which needs to be stored in the
//           files nf_re_c.mat and nf_im_c.mat in the
//           same directory
// z1=-10.0; x-coordinate of source point of incident field
// z2=10.0;  y-coordinate of source point of incident field
// k1=1.0;   wave number for upper half-space
// k2=2.0;   wave number for lower half-space
// a=-4.0;   parameters for the measurement line

```

```

// b=4.0;    defined by {(y,mh), a <= y <= b}
// mh=1.0;    height of measurement line above the ground
// plotpar=1; if plotpar=1 the field total field u will be
//             calculated and saved in u_re_c.mat, u_im_c.mat
//             if plotpar=<>1 only the values on the measure-
//             ment line nf_c.mat will be produced.
// alpha=1e-8; regularization parameter for the integral
//             equation of the first kind used in the
//             point source method
//
// r=4.0;    radius parameter for the domain of approximation
// eps=0.1;    distance of the source point from the domain
//             of approximation
//
// For the plot the following parameters are needed:
//
// a1=-10;    left and right, lower and upper boundary
// b1=10;    of the sampling domain
// a2=-10;    {(x,y); a1 <= x <= b1, a2 <= z <= b2}
// b2=10;
// n1=41;    number of sampling points in x-direction
// n2=41;    number of sampling points in y-direction
//
//
//-----

```

```

#include <iostream.h>
#include "matrix-classic.h"
#include "supp.h"
#include "math.h"
#include "fundsol.h"

```

```

//-----
//
//  main()
//
//-----

```

```

main(int argc, char *argv[]){

```

```

//-----
// parameters (see explanations above)
//-----

```

```

int nnf,plotpar;
double z1,z2,k1,k2,a,b,mh,alpha;

```

```

nnf=20;
z1=-10.0;

```

```

z2=10.0;
k1=1.0;
k2=2.0;
a=-4.0;
b=4.0;
mh=1.0;
plotpar=1;
alpha=1e-8;

double r,eps;
r=4.0;
eps=0.1;

int a1,b1,a2,b2,n1,n2;
a1=-3;
b1=5;
a2=-4;
b2=2;
n1=20;
n2=20;

//-----
// greetings and parameter output
//-----

cout << "-----\n";
system("date");
cout << "\n";
cout << "This is the program i n v b u r i e d _ 1 \n";
cout << "Reconstructing the scattered field for scattering\n";
cout << "of a point source by some buried obstacle\n";
cout << "from its values on some upper line\n";

cout << "\n";
cout << " nnf = " << nnf << "\n";
cout << " z1 = " << z1 << "\n";
cout << " z2 = " << z2 << "\n";
cout << " k1 = " << k1 << "\n";
cout << " k2 = " << k2 << "\n";
cout << " a = " << a << "\n";
cout << " b = " << b << "\n";
cout << " mh = " << mh << "\n";
cout << " plotpar = " << plotpar << "\n";

cout << "\n";
cout << " a1 = " << a1 << "\n";
cout << " b1 = " << b1 << "\n";
cout << " a2 = " << a2 << "\n";
cout << " b2 = " << b2 << "\n";
cout << " n1 = " << n1 << "\n";

```

```

cout << " n2    = " << n2 << "\n";

//-----
// read in far field data
//-----

realVector nf_re, nf_im;
complexVector nf;
nf_re=realVector(nnf);
nf_im=realVector(nnf);
nf=complexVector(nnf);

ifstream in("nf_re_c.mat");
if (!in) cout << "Error: cannot open input file nf_re_c.mat\n";

for(int t=0; t<nnf; t++){
    in >> *(nf_re.start+t);
}
in.close();

ifstream in2("nf_im_c.mat");
if (!in2) cout << "Error: cannot open input file nf_im_c.mat\n";

for(int t=0; t<nnf; t++){
    in2 >> *(nf_im.start+t);
}
in2.close();

for(int t=0; t<nnf; t++){
    *(nf.start+t)=complex(*(nf_re.start+t),*(nf_im.start+t));
}

//-----
// points on the upper line
//-----

realVector xnf1,xnf2;
xnf1=realVector(nnf);
xnf2=realVector(nnf);
for(int j=0; j<nnf; j++){
    *(xnf1.start + j) = a+(b-a)/nnf*j;
    *(xnf2.start + j) = mh;
}
// saveRealVectorMatlabFormat( xnf1, "xnf1_c.mat");
// saveRealVectorMatlabFormat( xnf2, "xnf2_c.mat");

//-----
// domain of approximation
//-----

```

```

double h,t1;
realVector ax1,ax2;
ax1=realVector(nnf);
ax2=realVector(nnf);
h=2*pi/(1.0*nnf);
for(int j=0; j<nnf; j++){
    t1=h*j;
    *(ax1.start + j) = r*cos(t1);
    *(ax2.start + j) = r*0.5*sin(t1)-0.5*r-eps;
}
// saveRealVectorMatlabFormat( ax1, "ax1_c.mat");
// saveRealVectorMatlabFormat( ax2, "ax2_c.mat");

//-----
// evaluation points ex and ey
//-----

double h1,h2;
h1=(1.0*b1-1.0*a1)/(1.0*n1);
h2=(1.0*b2-1.0*a2)/(1.0*n2);

cout << " h1  = " << h1 << "\n";
cout << " h2  = " << h2 << "\n";
cout << "\n";

realVector ex,ey;
ex=realVector(n1);
ey=realVector(n2);
for(int j=0; j<n1; j++){
    *(ex.start+j)=a1+h1*j;
}
for(int l=0; l<n2; l++){
    *(ey.start+l)=a2+h2*l;
}

saveRealVectorMatlabFormat( ex, "ex_c.mat");
saveRealVectorMatlabFormat( ey, "ey_c.mat");

//-----
// for each evaluation point set up the matrix and
// calculate the backprojection
//-----

matrixClassic hergmat;
hergmat=matrixClassic(nnf,nnf);
complexVector rs;
rs=complexVector(nnf);

```

```

matrixClassic eye, tik, tikInv, hergmatAdj;
eye=identityMatrixClassic(nnf);
tik=matrixClassic(nnf,nnf);
tikInv=matrixClassic(nnf,nnf);
hergmatAdj=matrixClassic(nnf,nnf);
complexVector den;
den=complexVector(nnf);

matrixClassic inc, usa, u;
inc=matrixClassic(n1,n2);
usa=matrixClassic(n1,n2);
u=matrixClassic(n1,n2);

cout << "- counting from (0,0) to (" << n1-1 << ", " << n2-1 << "):\n";

for(int l1=0; l1<n1; l1++){
    for(int l2=0; l2<n2; l2++){

        cout << "\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b";
        cout << " (" << l1 << ", " << l2 << ")";
        cout.flush();

        //-----
        // matrix for calculation of density g
        //-----

        for (int j=0; j<nnf; j++){
            for (int l=0; l<nnf; l++){
                *(hergmat.start + nnf*j + l) =
                    fundsol(*(ax1.start+j)+*(ex.start+l1),
                        *(ax2.start+j)+*(ey.start+l2),
                        *(xnf1.start+l), *(xnf2.start+l),
                        k1,k2);
            };
        };

        //-----
        // right-hand side for calculation of density g
        //-----

        for (int j=0; j<nnf; j++){
            *(rs.start + j) =
                fundsol(*(ax1.start+j)+*(ex.start+l1),
                    *(ax2.start+j)+*(ey.start+l2),
                    *(ex.start+l1), *(ey.start+l2),
                    k1, k2);
        };
    }
}

```

```

//-----
// solution of the IGL
//-----

hergmatAdj=adj(hergmat);
tik=alpha*eye + hergmatAdj*hergmat;
tikInv=gauss(tik);
den=tikInv*hergmatAdj*rs;

*(inc.start+n2*l1+l2)=fundsol(*(ex.start+l1),*(ey.start+l2),z1,z2,k1,k2);
*(usa.start+n2*l1+l2)=complex(0.0,0.0);
for(int xi=0; xi<nnf; xi++){
    *(usa.start+n2*l1+l2)=*(usa.start+n2*l1+l2) +
        (*(nf.start+xi))*(*(den.start+xi));
}
*(u.start+n2*l1+l2)=*(inc.start+n2*l1+l2) + (*(usa.start+n2*l1+l2));
}

//-----
// save for further use (Matlab format)
//-----

saveMatrixClassicMatlabFormat(inc,"inc_re_c.mat", "inc_im_c.mat");
saveMatrixClassicMatlabFormat(usa,"usa_re_c.mat", "usa_im_c.mat");
saveMatrixClassicMatlabFormat(u,"u_re_c.mat", "u_im_c.mat");
cout << "- saved inc_re_c, inc_im_c, usa_re_c, usa_im_c, u_re_c and u_im_c.\n";

cout << "\n";
system("date");
cout << "-----\n";

//-----
//
//-----

// End
}

```

complex.h Subprogram

```
#ifndef complex_h
#define complex_h

#include <iostream.h>
#include <fstream.h>
#include <iomanip.h>
#include <math.h>

//*****
// class complex
//*****

class complex{

public:

    double re, im;

    complex (double r=0, double i=0){
        re = r;
        im = i;
    }

//*****
// output of a complex number, alternative: <<
//*****

    void print() {
        cout << "(" << re << ", " << im << ")";
    }

}; //class complex

//*****
// the output operator for a complex number
//*****

ostream& operator << (ostream& s, complex a1){
    s << "(" << a1.re << ", " << a1.im << ")";
    return s;
}
```

```

//*****
// the input operator for a complex number
// c1=(c1.re, c1.im) in this form
//*****

istream& operator >> (istream& s, complex& c1){
    long double rpl=0, ipl=0;
    double rp=0, ip=0;
    char c;

    s >> c;
    if (c == '('){
        s >> rpl >> c;
        if (c == ',') s >> ipl >> c;
        if (c != ')') {
            cout << "Error: you did not type a complex number!";
            s.clear(ios::badbit);
        }
    }else{
        s.putback(c);
        s>>rpl;
    }

    if(s) c1=complex((double)rpl,(double)ipl);
    return s;
}

//*****
// the Sum of two complex numbers
//*****

complex operator +( complex a1 , complex a2 ) {
    return complex (a1.re + a2.re, a1.im + a2.im);
}

//*****
// subtract a complex number
//*****

complex operator -( complex a1 , complex a2 ) {
    return complex(a1.re - a2.re, a1.im - a2.im);
}

//*****
// multiply two complex numbers
//*****

complex operator *( complex a1 , complex a2 ) {
    return complex(a1.re*a2.re-a1.im*a2.im,

```

```

        a1.re*a2.im+a1.im*a2.re);
    }

//*****
// multiply a double and a complex number
//*****

complex operator *( double d , complex a2 ) {
    return complex(d*(a2.re), d*(a2.im));
}

complex operator *( complex a2, double a1 ) {
    return complex (a1*a2.re, a1*a2.im);
}

//*****
// Quotient of two complex numbers
//*****

complex operator /( complex a1, complex a2) {
    return complex ((a1.re * a2.re + a1.im * a2.im)/(a2.re*a2.re + a2.im*a2.im),
        (a1.im * a2.re - a1.re * a2.im)/(a2.re*a2.re + a2.im*a2.im));
}

//*****
// euclidean norm of a complex number
//*****

double norm(complex a1){
    return sqrt( pow(a1.re,2) + pow(a1.im,2));
}

//*****
// absolute value of a complex number == euclidean norm
//*****

double abs(complex a1){
    return sqrt( pow(a1.re,2) + pow(a1.im,2));
}

//*****
// the conjugate of a complex number
//*****

complex conj( complex a1){
    return complex( a1.re, -a1.im);
}
//*****

```

```

// the real part of a complex number
//*****

double realPart( complex a1){
    return a1.re;
}
//*****

// the imaginary part of a complex number
//*****

double imaginaryPart( complex a1){
    return a1.im;
}

//*****

// the exponential function of a complex number
//*****

complex exp(complex a1){
    return complex( exp(a1.re) * cos(a1.im), exp(a1.re) * sin(a1.im));
}

//*****
****
//
//  class complexVector
//
//*****
****

class complexVector{

public:
    int n;
    complex *start;

// Initialization of the point-vector

    complexVector(){
        n=1;
        start = new complex[1];
    };

    complexVector(int ni){
        n=ni;
        start = new complex[n];
        for(int j=0; j<n; j++){
            *(start + j) = complex();
        };
    };
};

```

```

    };
};

//-----
//
// the output operator for complexVector
//
//-----

ostream& operator << (ostream& s, complexVector cv1){
    int k, n=cv1.n;
    s << "\n";
    s << " ComplexVector size: n = " << n << ",  \n";
    k=0;
    for (int j=0; j<n; j++){
        k++;
        s << " " << setprecision(8)<< (*(cv1.start + j));
        s.flush();
        if( k == 4){
            s << "\n";
            k=0;
        };
    };//for j
    cout << "\n";
    return s;
}

//-----
//
// save a complexVector in a file in Mathematica-format
//
//-----
//
//void saveComplexVectorMathematicaFormat( complexVector cv1,
//                                         char* fileName="complexVector.mat"){
//
// char ch;
//
// ofstream totemp("temp.mat");
// if (!totemp) cout << "Error: cannot open output file" << fileName << "\n";
//
// totemp << "{";
// for(int j=0; j<cv1.n; j++){
//     totemp << " ";
//     totemp << (*(cv1.start + j)).re<< " + I*";
//     totemp << (*(cv1.start + j)).im;
//     if(j<cv1.n-1){

```

```

//  totemp << ",";
//  };
//  };//j
//  totemp << "}";
//  totemp.close();
//
//  ifstream from("temp.mat");
//  ofstream to(fileName);
//  while(from.get(ch)){
//      if(ch == 'e'){
//          to << "*10^";
//      }else{
//          to << ch;
//      };
//  };
//  };
//  from.close();
//  to.close();
//  system("rm temp.mat");
//
//  cout << "    |sl The complexVector is saved in ";
//  cout << fileName << " in Mathematica-format.\n";
//}
//
//-----
//
//  save a complexVector in a file in Matlab-format
//
//-----

void saveComplexVectorMatlabFormat( complexVector cv1,
                                   char* fileName1="Vector_re.mat",
                                   char* fileName2="cVector_im.mat"
                                   ){

    char ch;

    ofstream totemp1(fileName1);
    if (!totemp1) cout << "Error: cannot open output file" << fileName1 << "\n";

    ofstream totemp2(fileName2);
    if (!totemp2) cout << "Error: cannot open output file" << fileName2 << "\n";

    for(int j=0; j<cv1.n; j++){
        totemp1 << (*(cv1.start + j)).re << "\n";
        totemp2 << (*(cv1.start + j)).im << "\n";
    };//j
    totemp1.close();
    totemp2.close();

}

```

```

//*****
****
//
//  class realVector
//
//*****
****

class realVector{

public:
    int n;
    double *start;

// Initialization of the point-vector

    realVector(){
        n=1;
        start = new double[1];
    };

    realVector(int ni){
        n=ni;
        start = new double[n];
        for(int j=0; j<n; j++){
            *(start + j) = 0.0;
        };
    };
};

//-----
//
//  the output operator for realVector
//
//-----

ostream& operator << (ostream& s, realVector cv1){
    int k, n=cv1.n;
    s << "\n";
    s << " RealVector size: n = " << n << ",  \n";
    k=0;
    for (int j=0; j<n; j++){
        k++;
        s << " " << setprecision(8) << (*(cv1.start + j));
        s.flush();
        if( k == 4){
            s << "\n";

```

```

        k=0;
    };
};//for j
cout << "\n";
return s;
}

//-----
//
// save a realVector in a file in Matlab-format
//
//-----

void saveRealVectorMatlabFormat( realVector cv1,
                                char* fileName1="vector.mat"
                                ){

    char ch;
    int n=cv1.n;

    ofstream totemp(fileName1);
    if (!totemp) cout << "Error: cannot open output file" << fileName1 << "\n";

    for(int t=0; t<n; t++){
        totemp << *(cv1.start + t) << "\n";
    };//j
    totemp.close();
}
#endif //complex_h

```

fundsol.h Subprogram for the fundamental solution

```

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "complex.h"

//*****
//
// Fundamental solution for two-layered medium
//
// source point (z1,z2)
// evaluation point (x1,x2)
// wave number for upper half-space k1
// wave number for lower half-space k2
//
//
//*****

complex fundsol(double x1,double x2,double z1,double z2,double k1,double k2)

```

```

{
long m1,m2,m3,m4,m5;
double a,eps,arg,aa,bb;
complex fund1,fundsol,ci,hankel,fund2;
complex fundref( double x1 , double x2 , double z1 , double z2 ,
    double k1 , double k2 , double a , double eps , long m1 , long m2 ,
        long m3 , long m4);
complex fundtr( double x1 , double x2 , double z1 , double z2 ,
    double k1 , double k2 , double a , long m1);
ci=complex(0.0,1.0);

m1=10;
m2=10;
m3=10;
m4=10;
m5=100;
a=2*max(k1,k2);
eps=0.1;

if( x2>0 && z2>0 )
{
    arg=k1*sqrt(pow((x1-z1),2)+pow((x2-z2),2));
    fund1=0.25*ci*complex(j0(arg),y0(arg));
    fundsol=fund1+fundref(x1,x2,z1,z2,k1,k2,a,eps,m1,m2,m3,m4);
}

else if ( x2<0 && z2>0 )
{
    fundsol=fundtr(x1,x2,z1,z2,k1,k2,a,m5);
}

else if ( x2>0 && z2<0 )
{
    fundsol=fundtr(x1,-x2,z1,-z2,k2,k1,a,m5);
}

else
{
    arg=k2*sqrt(pow((x1-z1),2)+pow((x2-z2),2));
    fund1=0.25*ci*complex(j0(arg),y0(arg));

```

```

    fundsol=fund1+fundref(x1,-x2,z1,-z2,k2,k1,a,eps,m1,m2,m3,m4);
}

return fundsol;
}

```

```

complex fundref( double x1 , double x2 , double z1 , double z2 ,
double k1 , double k2 , double a , double eps , long m1 , long m2 ,
long m3 , long m4)

{
long i1;
double aa,bb,pi,h1,h2,h3,nu11,nu21,nu31,heps,theta11,heps1,theta21;
complex top11,ker11,top13,top14,int1;
complex top21,ker21,top23,top24,int2;
complex top31,ker31,top33,top34,int3;
complex top41,nu41,ker41,top43,top44,int4;
complex top51,nu51,ker51,top53,top54,int5;
complex ar1,ar2,kc1,kc2,nu43,nu44,nu53,nu54,ci,fundr;
complex kernelref(double x1 , double x2 , double z1 , double z2 ,
double k1 , double k2 , double nu);
complex kernelref1(double x1 , double x2 , double z1 , double z2 ,
double k1 , double k2 , complex nu );
pi=3.141592653589793;
h1=(-k1-eps+a)/m1;
h2=(2*k1-2*eps)/m2;
h3=(a-k1-eps)/m3;
ci=complex(0.0,1.0);
kc1=complex(-k1,0.0);
kc2=complex(k1,0.0);

    top11=complex(0.0,0.0);
    for( i1=1 ; i1<=m1-1 ; i1++)
    {
        nu11=-a+i1*h1;
        ker11=kernelref(x1,x2,z1,z2,k1,k2,nu11);

        top11=top11+ker11;

    }

    top13= kernelref(x1,x2,z1,z2,k1,k2,-a);

    top14=kernelref(x1,x2,z1,z2,k1,k2,-k1-eps);

int1=h1*(top11+0.5*top13+0.5*top14);

```

```

top21=complex(0.0,0.0);
for( i1=1 ; i1<=m2-1 ; i1++)
{
    nu21=-k1+eps+i1*h2;
    ker21=kernelref(x1,x2,z1,z2,k1,k2,nu21);
    top21=top21+ker21;
}
top23= kernelref(x1,x2,z1,z2,k1,k2,-k1+eps);
top24=kernelref(x1,x2,z1,z2,k1,k2,k1-eps);
int2=h2*(top21+0.5*top23+0.5*top24);

top31=complex(0.0,0.0);
for( i1=1 ; i1<=m3-1 ; i1++)
{
    nu31=k1+eps+i1*h3;
    ker31=kernelref(x1,x2,z1,z2,k1,k2,nu31);
    top31=top31+ker31;
}
top33=kernelref(x1,x2,z1,z2,k1,k2,k1+eps);
top34=kernelref(x1,x2,z1,z2,k1,k2,a);
int3=h3*(top31+0.5*top33+0.5*top34);

heps=-pi/m4;

top41=complex(0.0,0.0);
for( i1=1 ; i1<=m4-1 ; i1++)
{
    theta11=pi+i1*heps;
    ar1=theta11*ci;
    nu41=kc1+eps*exp(ar1);
    ker41=kernelref1(x1,x2,z1,z2,k1,k2,nu41);
    top41=top41+ker41*ci*eps*exp(ar1);
}

nu43=complex(-k1-eps,0.0000001);
nu44=complex(-k1+eps,0.0);
top43=-1.*ci*eps*kernelref1(x1,x2,z1,z2,k1,k2,nu43);

top44=ci*eps*kernelref1(x1,x2,z1,z2,k1,k2,nu44);
int4=heps*(top41+0.5*top43+0.5*top44);

heps1=pi/m4;

```

```

        top51=complex(0.0,0.0);
        for( i1=1 ; i1<=m4-1 ; i1++)
        {
            theta21=pi+i1*heps1;
            ar2=theta21*ci;
            nu51=kc2+eps*exp(ar2);
            ker51=kernelref1(x1,x2,z1,z2,k1,k2,nu51);
            top51=top51+ker51*ci*eps*exp(ar2);
        }
        nu53=complex(k1-eps,0.0);
        nu54=complex(k1+eps,-0.0000001);
        top53=-1.*ci*eps*kernelref1(x1,x2,z1,z2,k1,k2,nu53);
        top54=ci*eps*kernelref1(x1,x2,z1,z2,k1,k2,nu54);
        int5=heps1*(top51+0.5*top53+0.5*top54);

        fundr=(1./(2.*pi))*(int1+int2+int3+int4+int5);

        return fundr;
    }

    complex fundtr( double x1 , double x2 , double z1 , double z2 ,
                    double k1 , double k2 , double a , long m1)
    {
        long i1;
        double h1,nu11,pi;
        complex top11,ker11,top13,top14,int1;
        complex fund;
        complex kerneltr(double x1 , double x2 , double z1 , double z2 ,
                        double k1 , double k2 , double nu);

        pi=3.141592653589793;

        h1=2.0*a/m1;
        top11=complex(0.0,0.0);
        for( i1=1 ; i1<=m1-1 ; i1++)
        {
            nu11=-a+i1*h1;
            ker11=kerneltr(x1,x2,z1,z2,k1,k2,nu11);
            top11=top11+ker11;
        }
        top13=kerneltr(x1,x2,z1,z2,k1,k2,-a);
        top14=kerneltr(x1,x2,z1,z2,k1,k2,a);
        int1=h1*(top11+0.5*top13+0.5*top14);
        fund=(1./(2.*pi))*int1;
        return fund;
    }

    complex kernelref(double x1 , double x2 , double z1 , double z2 ,
                    double k1 , double k2 , double nu)
    {

```

```

    double aa,bb;
    complex ci,g1,g2,r,e1,e2,kerref,p1;
    complex gamma( double k , double nu );
    ci=complex(0.,1.);
    g1=gamma(k1,nu);
    g2=gamma(k2,nu);
    r=(g1-g2)/(g1+g2);
    p1=-1.*(x2+z2)*g1;
    e1=r*exp(p1);
    e2=exp(nu*(x1-z1)*ci);
    kerref=e1*e2/(2.*g1);
    return kerref;
}

complex kernelref1(double x1 , double x2 , double z1 , double z2 ,
                  double k1 , double k2 , complex nu)
{
    double aa,bb;
    complex ci,g1,g2,r,e1,e2,kerref1,p1;
    complex gammac( double k , complex nu );
    ci=complex(0.,1.);

    g1=gammac(k1,nu);
    g2=gammac(k2,nu);
    r=(g1-g2)/(g1+g2);
    p1=-1.*(x2+z2)*g1;
    e1=r*exp(p1);
    e2=exp(nu*(x1-z1)*ci);
    kerref1=e1*e2/(2.*g1);
    return kerref1;
}

complex kerneltr(double x1 , double x2 , double z1 , double z2 ,
                double k1 , double k2 , double nu)
{
    complex g1,g2,t,e1,e2,kertr,ci;
    complex gamma( double k , double nu );
    ci=complex(0.,1.);
    g1=gamma(k1,nu);
    g2=gamma(k2,nu);
    t=complex(1.,0.)/(g1+g2);
    e1=t*exp(-1.*z2*g1+x2*g2);
    e2=exp(nu*(x1-z1)*ci);
    kertr=e1*e2;
    return kertr;
}

complex gamma( double k , double nu )

```

```

{
complex ga,nu1;
nu1=complex(nu,0.0);

    if( abs(nu1) > k )
    {
        ga=complex(sqrt(nu*nu-k*k),0.0);
    }
    else
    {
        ga=complex(0.,-sqrt(k*k-nu*nu));
    }
    return ga;
}

complex gammac( double k , complex nu )
{
double pi,x3,x4,x5,x6,x8,aa,bb;
complex ci,x1,x2,x7,ggc;
ci=complex(0.,1.);
pi=3.141592653589793;
x1=nu-complex(k,0.0);
x2=nu+complex(k,0.0);
x3=abs(x1);
x4=abs(x2);

if ( imaginaryPart(nu) < 0. )
{
    x5=2.*pi+atan2(imaginaryPart(x1),realPart(x1));
}
else
{
    x5=atan2(imaginaryPart(x1),realPart(x1));
}
x6=atan2(imaginaryPart(x2),realPart(x2));
x7=exp(0.5*(x5+x6)*ci);
x8=sqrt(x3*x4);
ggc=-1.*x8*x7;
return ggc;
}

```

fundsol2.h Subprogram for the fundamental solution in two-layered medium

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "complex.h"

/*****
//
// Fundamental solution for two-layered medium
//
// source point (z1,z2)
// evaluation point (x1,x2)
// wave number for upper half-space k1
// wave number for lower half-space k2
//
//
*****/

complex fundsol(double x1,double x2,double z1,double z2,double k1,double k2)

{
long m1,m2,m3,m4,m5;
double a,eps,arg,aa,bb;
complex fund1,fundsol,ci,hankel,fund2;
complex fundref( double x1 , double x2 , double z1 , double z2 ,
double k1 , double k2 , double a , double eps , long m1 , long m2 ,
long m3 , long m4);
complex fundtr( double x1 , double x2 , double z1 , double z2 ,
double k1 , double k2 , double a , long m1);
ci=complex(0.0,1.0);

m1=50;
m2=50;
m3=50;
m4=50;
m5=100;
a=2*(k1+k2);
eps=0.1;
```

```

if( x2>0 && z2>0 )
{

    arg=k1*sqrt(pow((x1-z1),2)+pow((x2-z2),2));
    fund1=0.25*ci*complex(j0(arg),y0(arg));
    fundsol=fund1+fundref(x1,x2,z1,z2,k1,k2,a,eps,m1,m2,m3,m4);
}

else if ( x2<0 && z2>0 )
{

    fundsol=fundtr(x1,x2,z1,z2,k1,k2,a,m5);
}

else if ( x2>0 && z2<0 )
{

    fundsol=fundtr(x1,-x2,z1,-z2,k2,k1,a,m5);
}

else

{
    arg=k2*sqrt(pow((x1-z1),2)+pow((x2-z2),2));
    fund1=0.25*ci*complex(j0(arg),y0(arg));

    fundsol=fund1+fundref(x1,-x2,z1,-z2,k2,k1,a,eps,m1,m2,m3,m4);
}

return fundsol;
}

complex fundref( double x1 , double x2 , double z1 , double z2 ,
double k1 , double k2 , double a , double eps , long m1 , long m2 ,
long m3 , long m4)

{
long i1;
double aa,bb,pi,h1,h2,h3,nu11,nu21,nu31,heps,theta11,heps1,theta21;
complex top11,ker11,top13,top14,int1;
complex top21,ker21,top23,top24,int2;
complex top31,ker31,top33,top34,int3;
complex top41,nu41,ker41,top43,top44,int4;
complex top51,nu51,ker51,top53,top54,int5;
complex ar1,ar2,kc1,kc2,nu43,nu44,nu53,nu54,ci,fundr;
complex kernelref(double x1 , double x2 , double z1 , double z2 ,
double k1 , double k2 , double nu);
complex kernelref1(double x1 , double x2 , double z1 , double z2 ,

```

```

double k1 , double k2 , complex nu );
pi=3.141592653589793;
h1=(-k1-eps+a)/m1;
h2=(2*k1-2*eps)/m2;
h3=(a-k1-eps)/m3;
ci=complex(0.0,1.0);
kc1=complex(-k1,0.0);
kc2=complex(k1,0.0);

    top11=complex(0.0,0.0);
    for( i1=1 ; i1<=m1-1 ; i1++)
    {
        nu11=-a+i1*h1;
        ker11=kernelref(x1,x2,z1,z2,k1,k2,nu11);

        top11=top11+ker11;

    }

    top13= kernelref(x1,x2,z1,z2,k1,k2,-a);

    top14=kernelref(x1,x2,z1,z2,k1,k2,-k1-eps);
int1=h1*(top11+0.5*top13+0.5*top14);

    top21=complex(0.0,0.0);
    for( i1=1 ; i1<=m2-1 ; i1++)
    {

        nu21=-k1+eps+i1*h2;
        ker21=kernelref(x1,x2,z1,z2,k1,k2,nu21);
        top21=top21+ker21;
    }
    top23= kernelref(x1,x2,z1,z2,k1,k2,-k1+eps);
    top24=kernelref(x1,x2,z1,z2,k1,k2,k1-eps);
int2=h2*(top21+0.5*top23+0.5*top24);

    top31=complex(0.0,0.0);
    for( i1=1 ; i1<=m3-1 ; i1++)
    {

        nu31=k1+eps+i1*h3;
        ker31=kernelref(x1,x2,z1,z2,k1,k2,nu31);
        top31=top31+ker31;
    }
    top33=kernelref(x1,x2,z1,z2,k1,k2,k1+eps);
    top34=kernelref(x1,x2,z1,z2,k1,k2,a);
int3=h3*(top31+0.5*top33+0.5*top34);

```

```

heps=-pi/m4;

    top41=complex(0.0,0.0);
    for( i1=1 ; i1<=m4-1 ; i1++)
    {
        theta11=pi+i1*heps;
        ar1=theta11*ci;
        nu41=kc1+eps*exp(ar1);
        ker41=kernelref1(x1,x2,z1,z2,k1,k2,nu41);
        top41=top41+ker41*ci*eps*exp(ar1);
    }

    nu43=complex(-k1-eps,0.0000001);
    nu44=complex(-k1+eps,0.0);
    top43=-1.*ci*eps*kernelref1(x1,x2,z1,z2,k1,k2,nu43);

    top44=ci*eps*kernelref1(x1,x2,z1,z2,k1,k2,nu44);
    int4=heps*(top41+0.5*top43+0.5*top44);

    heps1=pi/m4;

    top51=complex(0.0,0.0);
    for( i1=1 ; i1<=m4-1 ; i1++)
    {
        theta21=pi+i1*heps1;
        ar2=theta21*ci;
        nu51=kc2+eps*exp(ar2);
        ker51=kernelref1(x1,x2,z1,z2,k1,k2,nu51);
        top51=top51+ker51*ci*eps*exp(ar2);
    }
    nu53=complex(k1-eps,0.0);
    nu54=complex(k1+eps,-0.0000001);
    top53=-1.*ci*eps*kernelref1(x1,x2,z1,z2,k1,k2,nu53);
    top54=ci*eps*kernelref1(x1,x2,z1,z2,k1,k2,nu54);
    int5=heps1*(top51+0.5*top53+0.5*top54);

    fundr=(1./(2.*pi))*(int1+int2+int3+int4+int5);

    return fundr;
}

complex fundtr( double x1 , double x2 , double z1 , double z2 ,
                double k1 , double k2 , double a , long m1)
{

```

```

long i1;
double h1,nu11,pi;
complex top11,ker11,top13,top14,int1;
complex fund;
complex kerneltr(double x1 , double x2 , double z1 , double z2 ,
                 double k1 , double k2 , double nu);

pi=3.141592653589793;

h1=2.0*a/m1;
top11=complex(0.0,0.0);
for( i1=1 ; i1<=m1-1 ; i1++)
{
    nu11=-a+i1*h1;
    ker11=kerneltr(x1,x2,z1,z2,k1,k2,nu11);
    top11=top11+ker11;
}
top13=kerneltr(x1,x2,z1,z2,k1,k2,-a);
top14=kerneltr(x1,x2,z1,z2,k1,k2,a);
int1=h1*(top11+0.5*top13+0.5*top14);
fund=(1./(2.*pi))*int1;
return fund;
}

complex kernelref(double x1 , double x2 , double z1 , double z2 ,
                 double k1 , double k2 , double nu)
{
    double aa,bb;
    complex ci,g1,g2,r,e1,e2,kerref,p1;
    complex gamma( double k , double nu );
    ci=complex(0.,1.);
    g1=gamma(k1,nu);
    g2=gamma(k2,nu);
    r=(g1-g2)/(g1+g2);
    p1=-1.*(x2+z2)*g1;
    e1=r*exp(p1);
    e2=exp(nu*(x1-z1)*ci);
    kerref=e1*e2/(2.*g1);
    return kerref;
}

complex kernelref1(double x1 , double x2 , double z1 , double z2 ,
                  double k1 , double k2 , complex nu)
{
    double aa,bb;
    complex ci,g1,g2,r,e1,e2,kerref1,p1;
    complex gammac( double k , complex nu );
    ci=complex(0.,1.);

```

```

    g1=gamma(k1,nu);
    g2=gamma(k2,nu);
    r=(g1-g2)/(g1+g2);
    p1=-1.*(x2+z2)*g1;
    e1=r*exp(p1);
    e2=exp(nu*(x1-z1)*ci);
    kerref1=e1*e2/(2.*g1);
    return kerref1;
}

complex kerneltr(double x1 , double x2 , double z1 , double z2 ,
                double k1 , double k2 , double nu)
{
    complex g1,g2,t,e1,e2,kertr,ci;
    complex gamma( double k , double nu );
    ci=complex(0.,1.);
    g1=gamma(k1,nu);
    g2=gamma(k2,nu);
    t=complex(1.,0.)/(g1+g2);
    e1=t*exp(-1.*z2*g1+x2*g2);
    e2=exp(nu*(x1-z1)*ci);
    kertr=e1*e2;
    return kertr;
}

complex gamma( double k , double nu )
{
    complex ga,nu1;
    nu1=complex(nu,0.0);

    if( abs(nu1) > k )
    {
        ga=complex(sqrt(nu*nu-k*k),0.0);
    }
    else
    {
        ga=complex(0.,-sqrt(k*k-nu*nu));
    }
    return ga;
}

complex gammac( double k , complex nu )
{
    double pi,x3,x4,x5,x6,x8,aa,bb;
    complex ci,x1,x2,x7,ggc;
    ci=complex(0.,1.);
    pi=3.141592653589793;
    x1=nu-complex(k,0.0);
    x2=nu+complex(k,0.0);

```

```

x3=abs(x1);
x4=abs(x2);

if ( imaginaryPart(nu) < 0. )
{
    x5=2.*pi+atan2(imaginaryPart(x1),realPart(x1));

}
else
{
    x5=atan2(imaginaryPart(x1),realPart(x1));

}
x6=atan2(imaginaryPart(x2),realPart(x2));
x7=exp(0.5*(x5+x6)*ci);
x8=sqrt(x3*x4);
ggc=-1.*x8*x7;
return ggc;
}

```

matrix-classic.h Subprogram

```

#include <iostream.h>
#include <fstream.h>
#include <iomanip.h>
#include "complex.h"
#include "supp.h"

//----- class matrixClassic -----
//
// A classical matrix consists of the integers n and m
// which define the size of the matrix and an array of
// complex elements
//
//-----

//-----
// class matrixClassic
//-----

class matrixClassic{

public:
    int n;

```

```

int m;
complex* start;

// Initialization of the matrix

matrixClassic(){
    n=1;
    m=1;
    start=new complex[1];
    *start=complex(0.0, 0.0);
};

matrixClassic(int ni, int mi){
    n=ni;
    m=mi;
    start = new complex[ni*mi];
    for(int i=0; i<ni; i++){
        for(int j=0; j<mi; j++){
            *(start + mi*i + j) = complex(0.0,0.0);
        };
    };
};

//-----
// print() to print the matrix, alternative: <<
//-----

void print(){
    int k;
    cout << "\n-----\n";
    cout << "n: " << n << " ";
    cout << "m: " << m << "\n\n";
    for (int i=0; i<n; i++){
        k=0;
        for (int j=0; j<m; j++){
            k++;
            cout << "(";
            cout << (*(start + m*i + j)).re << ", " << (*(start + m*i + j)).im;
            cout << ") \t";
            if( k == 10){
                cout << "\n";
                k=0;
            };
        }
        cout << "\n\n";
    };
};
}; //class matrixClassic

//-----

```

```

//
// the output operator for a classical Matrix
//
//-----

ostream& operator << (ostream& s, matrixClassic a1){

    int k, n=a1.n, m=a1.m;
    s << "\n";
    s << "Matrix size: n = " << n << ",   ";
    s << "m = " << m << ",\n\n";
    if( m != 1){
        for (int i=0; i<n; i++){
            k=0;
            for (int j=0; j<m; j++){
                k++;
                s << setprecision(8)<< *(a1.start + m*i + j) << "\t";
                s.flush();
                if( k == 4){
                    s << "\n";
                    k=0;
                };
            };//for j
            s<< "\n\n";
        };//for i
    }
    return s;
}
else{
    k=0;
    for (int i=0; i<n; i++){
        s << setprecision(8)<< *(a1.start + i) << "\t";
        s.flush();
        if( k == 4){
            s << "\n";
            k=0;
        };
    };//for i
};
}

//-----
//
// save a matrix in a file
//
//-----

void saveMatrixClassic( matrixClassic a1, char* fileName="mat.dat"){

    ofstream to(fileName);
    if (!to) cout << "Error: cannot open output file" << fileName << "\n";
}

```

```

to << a1.n << " ";
to << a1.m << " ";

for(int i=0; i<a1.n; i++){
    for(int j=0; j<a1.m; j++){
        to << *(a1.start + a1.m * i + j) << " \n";
    };
};
to.close();
cout << "    \n Matrix saved in " << fileName << ".\n";
}

//-----
//
// save a matrix in a file
//
//-----

void saveMatrixClassicNoComment( matrixClassic a1, char* fileName="mat.dat"){

    ofstream to(fileName);
    if (!to) cout << "Error: cannot open output file" << fileName << "\n";

    to << a1.n << " ";
    to << a1.m << " ";

    for(int i=0; i<a1.n; i++){
        for(int j=0; j<a1.m; j++){
            to << *(a1.start + a1.m * i + j) << " \n";
        };
    };
    to.close();

}

//-----
//
// save a matrix in a file in Mathematica-format
//
//-----

//void saveMatrixClassicMathematicaFormat( matrixClassic a1, char*
fileName="matrix.mat"){
//
// char ch;
//
// ofstream totemp("temp.mat");
// if (!totemp) cout << "Error: cannot open output file" << fileName << "\n";
//

```

```

// totemp << "{";
//
// for(int i=0; i<a1.n; i++){
//   if(a1.m != 1){
//     totemp << "{";
//   };
//   for(int j=0; j<a1.m; j++){
//     totemp << (*(a1.start + a1.m * i + j)).re << "+ I * ";
//     totemp << (*(a1.start + a1.m * i + j)).im;
//     if(j < a1.m-1){
//       totemp << ",";
//     };
//   };//j
//   if(a1.m != 1){
//     if(i < a1.n-1){
//       totemp << ",";
//     }else{
//       totemp << "}}";
//     };
//   }else{
//     if(i < a1.n-1){
//       totemp << ",";
//     }else{
//       totemp << "}";
//     };
//   };
// };//i
// totemp.close();
//
// ifstream from("temp.mat");
// ofstream to(fileName);
// while(from.get(ch)){
//   if(ch == 'e'){
//     to << "*10^";
//   }else{
//     to << ch;
//   };
// };
// from.close();
// to.close();
// system("rm temp.mat");
//
// // cout << "    | The matrix is saved in " << fileName << " in Mathematica-
format.\n";
//}
///-----
//
// save a matrix in a file in Matlab-format
//
//-----

```

```

void saveMatrixClassicMatlabFormat( matrixClassic a1,
                                   char* fileName1="matrix_re.mat",
                                   char* fileName2="matrix_im.mat"
                                   ){

    char ch;
    int n=a1.n, m=a1.m;

    ofstream to(fileName1);
    if (!to) cout << "Error: cannot open output file" << fileName1 << "\n";

    for(int i=0; i<n; i++){
        for(int j=0; j<m; j++){
            to << (*(a1.start + m*i + j)).re << " ";
        };//j
        to << "\n";
    };//i
    to.close();

    ofstream to2(fileName2);
    if (!to) cout << "Error: cannot open output file" << fileName1 << "\n";

    for(int i=0; i<n; i++){
        for(int j=0; j<m; j++){
            to2 << (*(a1.start + m*i + j)).im << " ";
        };//j
        to2 << "\n";
    };//i
    to2.close();

}

//-----
//
//  get a matrix from a file
//
//-----

void getMatrixClassic( matrixClassic &a1, char* fileName){

    int n1,m1;

    ifstream from(fileName);
    if (!from) cout << "Error: cannot open file" << fileName << "\n";

    from >> n1;
    from >> m1;
    a1=matrixClassic(n1,m1);

```

```

    for(int i=0; i<a1.n; i++){
        for(int j=0; j<a1.m; j++){
            from >> *(a1.start + a1.m * i + j);
        };
    };
    from.close();
    cout << "      |r| The matrix is read from " << fileName << ".\n";
}

//-----
//
//  get a matrix from a file
//
//-----

void getMatrixClassicPlainFormat( matrixClassic &a1, char* fileName){

    int n1,m1;

    ifstream from(fileName);
    if (!from) cout << "Error: cannot open file" << fileName << ".\n";

    from >> n1;
    from >> m1;
    a1=matrixClassic(n1,m1);

    for(int i=0; i<a1.n; i++){
        for(int j=0; j<a1.m; j++){
            from >> (*(a1.start + a1.m * i + j)).re;
            from >> (*(a1.start + a1.m * i + j)).im;
        };
    };
    from.close();
    cout << "      |r| The matrix is read from " << fileName << ".\n";
}

//-----
//
//  Initiate the identity matrix
//
//-----

matrixClassic identityMatrixClassic(int ni){

    matrixClassic a3i=matrixClassic(ni,ni);

    for(int i=0; i<ni; i++){
        *(a3i.start + a3i.m*i + i) = complex(1.0,0.0);
    };
}

```

```

        return a3i;
    }

//-----
//
//  Add two matrices using +
//
//-----

matrixClassic operator +( matrixClassic a1i, matrixClassic a2i){

    if (a1i.n == a2i.n){
        if( a1i.m == a2i.m){

            matrixClassic a3i=matrixClassic(a1i.n,a1i.m);

            for (int i=0; i<a3i.n; i++){
                for (int j=0; j<a3i.m; j++){
                    *(a3i.start + a3i.n*i + j) =  *(a1i.start + a1i.n*i + j)
                                                    + *(a2i.start + a2i.n*i + j);
                };
            };
            return a3i;

        }else{
            cout << "Error: you try to add matrices of different size!\n\n";
        }
    }
}

```

```

//-----
//
//  Subtract two matrices using -
//
//-----

matrixClassic operator -( matrixClassic a1i, matrixClassic a2i){

    if (a1i.n == a2i.n){
        if( a1i.m == a2i.m){

            matrixClassic a3i=matrixClassic(a1i.n,a1i.m);

            for (int i=0; i<a3i.n; i++){
                for (int j=0; j<a3i.m; j++){
                    *(a3i.start + a3i.n*i + j) =  *(a1i.start + a1i.n*i + j)
                                                    - *(a2i.start + a2i.n*i + j);
                };
            };

```

```

    };
    return a3i;

    }else{
    cout << "Error: you try to add matrices of different size!\n\n";
    }
}

//-----
//
//  Multiply a double with a matrix using *
//
//-----

matrixClassic operator *(double d, matrixClassic a1i){

    matrixClassic a3i=matrixClassic(a1i.n,a1i.m);

    for (int i=0; i<a3i.n; i++){
        for (int j=0; j<a3i.m; j++){
            *(a3i.start + a3i.m*i + j)=d * (*(a1i.start + a1i.m*i + j));
        };
    };
    return a3i;

}

//-----
//
//  Multiply a matrix with a complexVector using *
//
//-----

complexVector operator *( matrixClassic a1i, complexVector cv){

    if (a1i.m == cv.n){

        complexVector res=complexVector(a1i.n);
        for (int i=0; i<res.n; i++){
            for (int k=0; k<cv.n; k++){
                *(res.start + i) = *(res.start + i) +
                    ( *(a1i.start + a1i.m*i + k)
                    * *(cv.start + k) );
            };
        };
        return res;

    }else{

```

```

        cout << "Error: you try to multiply a matrix with a vector of incompatible
size!\n\n";
    };
}

//-----
//
//  Multiply two matrices using *
//
//-----

matrixClassic operator *( matrixClassic a1i, matrixClassic a2i){

    if (a1i.m == a2i.n){

        matrixClassic aOut=matrixClassic(a1i.n,a2i.m);
        for (int i=0; i<aOut.n; i++){
            for (int j=0; j<aOut.m; j++){
                for (int k=0; k<a2i.n; k++){
                    *(aOut.start + aOut.m*i + j) = *(aOut.start + aOut.m*i + j) +
                        ( *(a1i.start + a1i.m*i + k)
                          * *(a2i.start + a2i.m*k + j) );
                };
            };
        };
        return aOut;

    }else{
        cout << "Error: you try to multiply matrices of incompatible size!\n\n";
    };
}

//-----
//
//  Take the power of a matrix using ^
//
//-----

matrixClassic operator ^( matrixClassic a1i, int power){

    if( a1i.n == a1i.m){

        matrixClassic ati=identityMatrixClassic(a1i.n);

        for (int i=0; i<power; i++){
            ati = ati * a1i;
        };
        return ati;
    }
}

```

```

    }else{
        cout << "Error: you try to take a power of a non-square-matrix!\n\n";
    };
}

```

```

//-----
//
// The L^2-Norm of a matrix
//
//-----

```

```

double normL2(matrixClassic ai){

    double sum=0;

    for(int i=0; i< ai.n; i++){
        for(int j=0; j<ai.m; j++){
            sum=sum + power((*(ai.start + ai.m*i + j)).re,2)
                + power((*(ai.start + ai.m*i + j)).im,2);
        };
    };
    return sqrt(sum);
}

```

```

//-----
//
// The C^0-Norm of a matrix
//
//-----

```

```

double normC0(matrixClassic ai){

    double max=0;

    for(int i=0; i< ai.n; i++){
        for(int j=0; j<ai.m; j++){
            if( abs(*(ai.start + ai.m*i + j)) > max){
                max=abs(*(ai.start + ai.m*i + j));
            };
        };
    };
    return max;
}

```

```

//-----
//
// the adjoint of a matrixClassic
//

```

```
//-----
matrixClassic adj(matrixClassic a1){

    matrixClassic aOut=matrixClassic(a1.m,a1.n);
    int n=a1.n;
    int m=a1.m;
    for (int i=0; i<aOut.n; i++){
        for (int j=0; j<aOut.m; j++){
            *(aOut.start + m*i + j)=conj( *(a1.start + n*j + i) );
        };
    };
    return aOut;

}

```

```
//-----
//
// the transposed matrix
//
//-----

```

```
matrixClassic transpose(matrixClassic a1i){

    matrixClassic a3i=matrixClassic(a1i.m,a1i.n);

    for (int i=0; i<a1i.n; i++){
        for (int j=0; j<a1i.m; j++){
            *(a3i.start + a3i.m*j + i)= *(a1i.start + a1i.m*i + j);
        };
    };
    return a3i;

}

```

```
//-----
// Definition of nodes for the SearchSet:
// each node contains the adress of the next node and its value,
// if nde is zero there is no next node.
//-----

```

```
class node{

public:
    node *prevNode;
    node *nextNode;
    int value;

```

```

// initiation of class node

node(node* nde1=0, node* nde2=0, int vle1=0){
    prevNode=nde1;
    nextNode=nde2;
    value=vle1;
};

};

//-----
// print the values of the SearchSet starting with node
//-----

void printNode(node *nde){

    int k=0;
    cout << "thisNode: " << (int)nde << " ";
    cout << "prevNode: " << (int)nde->prevNode << " ";
    cout << "nextNode: " << (int)nde->nextNode << " ";
    cout << "value: " << nde->value << " \n";

    if(nde->nextNode != 0){
        nde=nde->nextNode;
        printNode(nde);
        k++;
    }else{
        cout << "\n";
    };

}

//-----
// We set up a Nodelist (SearchSet for the Gauss-Algorithm)
//-----

node* makeSearchSet(int n){

    node *tempNode1=new node();
    tempNode1->value=1;
    node *startNode=tempNode1;

    for(int k=2; k<= n; k++){

        node *tempNode2= new node();
        tempNode1->nextNode=tempNode2;
        tempNode2->prevNode=tempNode1;
        tempNode2->value=k;
        tempNode1=tempNode2;
    };

}

```

```

    return startNode;
}

//-----
// Delete a node of the nodelist
//-----

void deleteNode(node* &delNode, node* anfang=0){

    node *tempNode;

    if( delNode!= 0){
        if( (delNode->nextNode) == 0){
            if( delNode->prevNode == 0){
                //the only node
                delete delNode;
            }else{
                //the last node
                (delNode->prevNode)->nextNode=delNode->nextNode;
                delete delNode;
            };
        }else{
            if( (delNode->prevNode) == 0){
                //the first node
                tempNode=delNode->nextNode;
                delNode->value=(delNode->nextNode)->value;
                delNode->nextNode=(delNode->nextNode)->nextNode;
                if( delNode->nextNode != 0){
                    (delNode->nextNode)->prevNode=delNode;
                }
                delete tempNode;
            };
        }else{
            //not the first or the last node
            (delNode->prevNode)->nextNode=delNode->nextNode;
            (delNode->nextNode)->prevNode=delNode->prevNode;
            delete delNode;
        };
    };
}

}

//-----
//
// Compute the inverse of the matrix using the Gauss-Algorithm
//
//-----

```

```

matrixClassic gauss(matrixClassic ai, char* str="gauss.mat"){

    node *searchSetNode=makeSearchSet(ai.n);
    node *tempNode, *biggestElementNode;
    complex biggestElement, ch1, sum;
    double dTest;
    int biggestElementIndex;
    matrixClassic aWork=matrixClassic(ai.n, 2 * ai.m);
    matrixClassic aInv=matrixClassic(ai.n, ai.m);
    complex tempVec[2*ai.m];

    // cout << "  lgauss! Hello! This is the classical Gauss-Algorithm!\n";
    // cout << "  lgauss! I am computing the inverse of your matrix....\n";

    // if we got a square-matrix
    if( ai.n == ai.m){

        //set up the working matrix
        for(int i=0; i<ai.n; i++){
            *(aWork.start + aWork.m * i + aWork.n + i)=complex(1,0);
            for(int j=0; j<ai.n; j++){
                *(aWork.start + aWork.m * i + j)=*(ai.start + ai.m * i + j);
            };
        };

        // cout << "  lgauss! Status report, counting from 0 to " << ai.n << ":\n";
        // cout << "  lgauss! l";

        //open testfile
        // ofstream totemp("gausstest.dat");
        // if (!ttemp) cout << "Error: cannot open output file" << "gausstest.dat" << "\n";

        for(int j=0; j<ai.n; j++){

            //if( j % 4 == 0){
            //cout << j << "l";
            //cout.flush();
            //};

            // Search the biggest element in row j
            // take only elements into account which are part of the searchSet!

            tempNode=searchSetNode;
            biggestElement=complex(0,0);

            while( ((int)tempNode) != 0){
                if( norm(*(aWork.start + aWork.m*((tempNode->value)-1) + j)) >
norm(biggestElement)){
                    biggestElement=*(aWork.start + aWork.m*((tempNode->value)-1) + j);
                    biggestElementIndex=(tempNode->value)-1;
                }
            }
        }
    }
}

```

```

        biggestElementNode=tempNode;
        tempNode=tempNode->nextNode;
    }else{
        tempNode=tempNode->nextNode;
    };
};

if(biggestElementNode != 0){
    deleteNode(biggestElementNode);
}else{
    cout << "Error with biggestElementNode!";
};

for(int k=0; k<aWork.m; k++){
    *(tempVec + k)= *(aWork.start + aWork.m*biggestElementIndex + k);
};

for(int l=0; l<aWork.n; l++){
    ch1=*(aWork.start + aWork.m*l + j);

    for(int k=j; k<aWork.m; k++){
        if( l != biggestElementIndex){
            *(aWork.start + aWork.m*l + k)=*(aWork.start + aWork.m*l + k)
            - ( *(tempVec + k) ) * ch1 / ( *(tempVec + j) );
        }else{
            *(aWork.start + aWork.m*l + k)=*(tempVec + k)
            / ( *(tempVec + j) );
        };
    };//for k
};//for l

};//for j

// cout << ai.n << "\n  lgaussl Transforming permutations...\n";

for(int l=0; l< ai.n; l++){
    for(int k=0; k<ai.n; k++){
        sum=complex(0.0, 0.0);
        for(int p=0; p<ai.n; p++){
            sum=sum + (*(aWork.start + aWork.m * p + l ))
            * (*(aWork.start + aWork.m * p + aWork.n + k ));
        };//for p
        *(aInv.start + aInv.n * l + k)=sum;
    };//for k
};//for l

dTest=normL2(aInv * ai - identityMatrixClassic(ai.n));
if(dTest > 0.00001){
    cout << "  lgaussl numerical errors in matrix inversion (dtest = " << dTest <<
    ")!!! \n";
}

```

```

    }else{
        // cout << " |gauss| Precision of the matrix inversion: " << dTest << "\n";
    };

    // saveMatrixClassicMathematicaFormat(aInv, str);
    // cout << " |gauss| Leaving Gauss-Algorithm.\n";

    //delete &aWork;

    //close test-file
    // totemp.close();
    return aInv;

}

}

//-----
//
// Compute the inverse of the matrix using the Gauss2-Algorithm
//
//-----

void gauss2(matrixClassic ai, matrixClassic &aInv,
            double &gaussTest, char* str="gauss.mat"){

    node *searchSetNode=makeSearchSet(ai.n);
    node *tempNode, *biggestElementNode;
    complex biggestElement, ch1, sum;
    int biggestElementIndex;
    matrixClassic aWork=matrixClassic(ai.n, 2 * ai.m);
    complex tempVec[2*ai.m];

    // if we got a square-matrix
    if( ai.n == ai.m){

        //set up the working matrix
        for(int i=0; i<ai.n; i++){
            *(aWork.start + aWork.m * i + aWork.n + i)=complex(1,0);
            for(int j=0; j<ai.n; j++){
                *(aWork.start + aWork.m * i + j)=*(ai.start + ai.m * i + j);
            };
        };

        //open testfile
        // ofstream totemp("gausstest.dat");
        // if (!totemp) cout << "Error: cannot open output file" << "gausstest.dat" << "\n";
    }
}

```

```

for(int j=0; j<ai.n; j++){

    // Search the biggest element in row j
    // take only elements into account which are part of the searchSet!

    tempNode=searchSetNode;
    biggestElement=complex(0,0);

    while(tempNode != 0){
        if( norm(*(aWork.start + aWork.m* tempNode->value + j)) >
norm(biggestElement)){
            biggestElement=*(aWork.start + aWork.m* tempNode->value + j);
            biggestElementIndex=tempNode->value;
            biggestElementNode=tempNode;
            tempNode=tempNode->nextNode;
        }else{
            tempNode=tempNode->nextNode;
        }
    };

    if(biggestElementNode != 0){
        deleteNode(biggestElementNode);
    };

    for(int k=0; k<aWork.m; k++){
        *(tempVec + k)= *(aWork.start + aWork.m*biggestElementIndex + k);
    };

    for(int l=0; l<aWork.n; l++){
        ch1=*(aWork.start + aWork.m*l + j);
        for(int k=j; k<aWork.m; k++){
            if( l != biggestElementIndex){
                *(aWork.start + aWork.m*l + k)=*(aWork.start + aWork.m*l + k)
                - ( *(tempVec + k) ) * ch1 / ( *(tempVec + j) );
            }else{
                *(aWork.start + aWork.m*l + k)=*(tempVec + k)
                / ( *(tempVec + j) );
            };
        };//for k
    };//for l

};//for j

for(int l=0; l< ai.n; l++){
    for(int k=0; k<ai.n; k++){
        sum=complex(0.0, 0.0);
        for(int p=0; p<ai.n; p++){
            sum=sum + (*(aWork.start + aWork.m * p + l ))
            * (*(aWork.start + aWork.m * p + aWork.n + k ));

```

```

        };//for p
        *(aInv.start + aInv.n * l + k)=sum;
    };//for k
};//for l

gaussTest=normL2(aInv * ai - identityMatrixClassic(ai.n));
if(gaussTest > 0.01){
    cout << "||" << gaussTest << "||";
    cout.flush();
}else{

};

delete &aWork;

}else{
    cout << "Error: you try to compute the inverse of a non-square matrix!";
};

}

```

supp.h Subprogram

```

#if !defined(supp)
#define supp

const double pi=3.14159;
const double EulerGamma=0.577215665;

//*****
// Maximum of double
//*****

double max(double a, double b){

    if(a > b){
        return a;
    }else{
        return b;
    }
}

```

```

}

//*****
//
//
//*****
int ipower(int bi, int ei){

    int it=1;

    if( ei == 0){
        it=1;
    }else{
        if( ei > 0){
            for(int k=0; k<ei; k++){
                it=it*bi;
            };
        }else{
            cout << "Error: you try to use ipower with a wrong exponent!";
        };
    };
    return it;
}

//*****
//
//
//*****

double power(double x, int e){

    double p=1.0;

    if( e == 0){
        p=1.0;
    }else{
        if( e > 0){
            for(int k=0; k<e; k++){
                p=p*x;
            };
        }else{
            x=1.0/x;
            p=power(x, -e);
        };
    };
    return p;
}

```

```

//*****
// Get an integer out of a string-variable
//*****

int stringToInteger(char* str){

    int k=0;
    int sum=0;

    while( *(str + k) > 47 && *(str + k) < 58){
        sum = 10*sum + ((int)*(str + k)-48);
        k++;
    };

    return sum;

}

//*****
// Get an double out of a string-variable
//*****

double stringToDouble(char* str){

    int k=0;
    int mark;
    double sum=0, sign;

    if( *str == '-'){
        sign=-1.0;
        k++;
    }else{
        sign=1.0;
    };
    while( *(str + k) > 47 && *(str + k) < 58){
        sum = 10*sum + ((int)*(str + k)-48);
        k++;
    };
    if(*(str+k) == '.'){
        mark=k;
        k++;
        while( *(str + k) > 47 && *(str + k) < 58){
            sum = sum + power(10.0,(mark-k))*((int)*(str + k)-48);
            k++;
        };
    }else{
        if(*(str+k) == 'e'){
            k++;
            cout << "beep";
        }
    }
}

```

```

    });

    return sign*sum;

}

//*****
// signum
//*****

int signum(double d){

    int s;
    if( d > 0){
        s=1;
    }else{
        if(d == 0){
            s=0;
        }else{
            s=-1;
        };
    };
    return s;
}

//*****
//
// compareString
//
//*****

int compareString( char* s1, char* s2){

    int k=0;

    while( *(s1+k) == *(s2+k) && *(s1+k) != '\n' && *(s2+k) != '\n' &&
           *(s1+k) != '\0' && *(s2+k) != '\0' ){
        k++;
    };

    if( (*(s1+k) == '\0' && *(s2+k) == '\0') ||
        (*(s1+k) == '\n' && *(s2+k) == '\0') ||
        (*(s1+k) == '\n' && *(s2+k) == '\n')){
        return 1;
    }else{
        return 0;
    };
}

```

```

//*****
// copyString
//*****

void copyString( char* s1, char* s2){

    int k=0;

    while( *(s1+k) != '\n' && *(s2+k) != '\n' &&
           *(s1+k) != '\0' && *(s2+k) != '\0' ){
        *(s1+k)=*(s2+k);
        k++;
    };

}

int min(int i1, int i2){

    int i3;
    if( i1 > i2){
        i3=i2;
    }else{
        i3=i1;
    };
    return i3;
}

int max(int i1, int i2){

    int i3;
    if( i1 > i2){
        i3=i1;
    }else{
        i3=i2;
    };
    return i3;
}

//*****
// round
//*****

double round(double d1, int n){

    double d2, dr;
    long int n2;

    d2=power(10,n)*d1;

```

```

    n2=(long int)d2;
    dr=(1.0*n2)/(1.0*power(10,n));
    return dr;
}

int round(double d1){

    long int n2;
    n2=(long int)d1;
    return n2;
}

double abs(double d1){

    double d2;
    if( d1 < 0){
        d2 = -1.0*d1;
    }else{
        d2=d1;
    };
    return d2;
}

//*****
//  class doubleVector
//*****

class doubleVector{

public:
    int n;
    double *start;

// Initialization of the point-vector

    doubleVector(){
        n=1;
        start = new double[1];
    };

    doubleVector(int ni){
        n=ni;
        start = new double[n];
        for(int j=0; j<n; j++){
            *(start + j) = 0.0;
        };
    };
};

//-----

```

```

// the output operator for doubleVector
//-----

ostream& operator << (ostream& s, doubleVector dv1){
    int k, n=dv1.n;
    s << "\n";
    s << "DoubleVector size: n = " << n << ",  \n";
    k=0;
    for (int j=0; j<n; j++){
        k++;
        s << " " << setprecision(4)<< (*(dv1.start + j));
        s.flush();
        if( k == 8){
            s << "\n";
            k=0;
        };
    };//for j
    cout << "\n";
    return s;
}
//*****
// mod()
//*****
int mod(int k, int n){

    int res;

    res=k%n;
    if( res<0){
        res=res+n;
    };
    return res;
}
//*****
#endif // _supp_h

```