Bioimpedance Measurements Using the Integrated Circuit AD5933

Bernt Jørgen Nordbotten

University of Oslo
Department of Physics
Electronics and Computer Science

Thesis for the degree of Master of Science

June 2008

Contents

Acknowledgement

I want to thank my advisors Ørjan G. Martinsen, Department of Physics, University of Oslo and Sverre Grimnes, Department of Clinical and Biomedical Engineering, Rikshospitalet, Oslo, for their valuable comments and guidance during the work on this thesis. Further I want to thank my fellow students for willingly having taken part as measurement objects in the whole body measurements. I also want to thank my brother Nils for proofreading the manuscript.

Oslo, June 2008

Bernt Jørgen Nordbotten

Abstract

This thesis gives a description of a prototype bioimpedance measurement system based on the integrated circuit AD5933. The prototype operates from 5 - 100 kHz and covers the impedance range 0.1kΩ - 10 MΩ in six subranges.

The system is operated from a PC, and the software required for operation and control has been developed.

Verification testing on R/C modules have shown that the calibration process and the signal level are critical issues with regard to operational performance. With carefull calibration the system operates well and whole body measurements on 11 persons have been performed with satisfactory and repeatable results. Statistical processing of the results using both a Standard Deviation approach and a Principal Components Analysis gave results identifying regions of whole body impedance values as well as indicators for outliers.

The measurement principle as such is very promising and with a promised updating of the IC the applicability will improve.

List of Figures

List of Tables

Chapter 1

Introduction

Bioimpedance measurements are becomming more and more used in medical situations, since there are becomming more and more applications where bioimpedance can be used. This thesis is using a fairly new integrated circuit for impedance measurements named AD5933. With this device it is possible to make cheap and small instruments. The AD5933 itself cost only about $19 or less, and is only available in a surface mounted package. So the system can also be made very compact, since the integrated circuit itself is small and requires few external parts as well.

1.1 Background and motivation

The body is a good electrical conductor. Its impedance can be measured by applying a weak signal and detecting the resulting current.

In this thesis, I consider the use of the AD5933 [2, 3] to simplify measurement procedures and make it possible to build easily portable equipment for operation in connection with a laptop. The AD5933 is a impedance converter with an internal frequency generator. The output excitation can be applied to an impedance, and the response signal will then be sampled and a Discrete Fourier Transform (DFT) is performed which returns a real an imaginary part which is used to calculate the impedance and phase.

We may foresee an interesting development with more advanced and flexible signal-processing optimized for bioimpedance measurements and evaluation of result. The AD5933 represent a step in this direction, and it is thus interesting to look into its application possibilities and shortcomings. Measurements of this type give information about the electrochemical processes in the tissue [1] an can be used for characterizing the tissue and for monitoring physiological changes. They may be used directly for detection of an illness like skin cancer [22].

Impedance measurements of the body is a noninvasive technique with low risk, and the equipment used is in general cheap, compared with hospital equipment for characterisation of body functionality.

Lately the advancement of IC-technology has opened for even further reduction in volume and prize [2] and operation in connection with laptops. Another development trend is the use of statistical interpolation of data.

The thesis work is focusing on utilization of this new technology, including making a prototype and developing necessary software for operation.

1.2 Goals

The main goal for this master thesis is to develop a bioimpedance measuring system using the integrated circuit AD5933 and perform verification tests on resistor-capacitor (RC) networks and living tissue to test the suitability of the system for whole body measurements and for a possible method for body composition.

1.2.1 Part goals

The main goal can be divided into the following part goals:

- Design and implement a prototype system based on the AD5933, using a microcontroller as interface between the AD5933 and the PC used for control and data collection.

- Develop software for the PC and microcontroller.

- Establish operational procedures for performing measurements with the system.

- Perform verification testing of the system on RC-networks.

- Analysis of results.

1.3 Structure of the thesis

This thesis is structured as follows:

Chapter 1 provides a background and motivation for the work, and presents the goals of this thesis.

Chapter 2 provides background information on the electrical properties of tissue that are of relevance to the usage of the device developed in this thesis. Relevant body impedance properties are also described.

Chapter 3 gives a description of the hardware of the system and the properties of the main modules and how they affect the operation of the system.

Chapter 4 discusses the software devloped during the work on this thesis. It also provides a brief description of third party software used during the work on this thesis.

Chapter 5 gives a prototype system specification and operational guidelines.

Chapter 6 cover the testing of the full prototype system including tests on resistor/capacitor networks and whole body measurements with analysis and discussion of results. Encountered problems associated with the built in properties of the AD5933 and its documentation are discussed.

Chapter 7 discusses possibilities for improvement of the prototype system, including using an upcoming improved version of the AD5933.

Chapter 8 presents final discussion and conclusions.

Appendixes

Appendix A gives the full C code for the microcontroller.

Appendix B gives the full Python code for the graphical interface.

Appendix C contains the table with all the body measurement results

Appendix D gives flow-diagrams for the developed software.

Chapter 2

Theoretical background

This chapter gives a background to the concept of bioimpedance and its use in body measurements, both for full body composition evaluation and study of some specific parts of the body.

The goal is to understand the potential of the system built and to give a background for the made measurements, in particular the total body measurements.

2.1 Bioimpedance and body composition

The measured response of a biomaterial (e.g., the total body, skin, muscle, fat, or blood), either dead or living, to an applied current is referred to as its bioimpedance. More specifically, the bioimpedance is the biomaterial's ability to oppose the applied current flow. Bioimpedance describes the passive electrical properties of biomaterial, and changes in the bioimpedance can reflect changes in the biomaterial (e.g., changes in water content, changes in the blood flow, nervous activity, galvanic skin response). The bioimpedance is a complex quantity, the reason for this is because the biomaterial not just oppose the applied current flow, it also phase-shifts the voltage with respect to the current in the time-domain caused by the built in capacitances at the cell membranes.

The body act as a conductor in combination with the capacitance of the cell membranes. The part of the body contributing most to the conduction process is referred to as the fat free mass (FFM). In addition we have the fat mass (FM) which only contribute a little to the electrical conduction process. If the FFM is determined from measurements, then FM is given by the total body weight minus the FFM. From the FFM we have one part from the extracellular water (ECW), another part intracellular water

(ICW), the protein and the bone structure with minerals. The total body water is the sum of ECW (extracellular water) and ICW (intracellular water). The body cell mass (BCM) which is the protein rich part of the body is not affected in catabolic states. The challenge is to make measurements allowing the best possible determination of these different compartments of the body.

The models describing the composition of the body are under further development. Ellis has presented a review [10] describing status and development trends. The basic and original 2-compartment (2-C) model distinguishing between fat free and fat is still in use. Going to the 3-C model FFM was divided into two parts, total water content and remaining solids (mainly protein and minerals). The 4-C model included protein and mineral compartments, and new measurements techniques like neutron activation analysis for body protein and dual energy X-ray absorption (DXA) for bone mineral content. This requires highly specialized equipment and safe operation.

Another 4-C model divides the fat free mass into body cell mass (BCM), extracellular water (ECW) and extracellular solids (ECS).

Ellis refers to a comprehensive 5-level model originally proposed by Wang et al. [11]. This model, which is shown in Figure 2.1, together with the 2-C model, operates with the levels elemental, molecular, cellular, tissue systems and the total body.

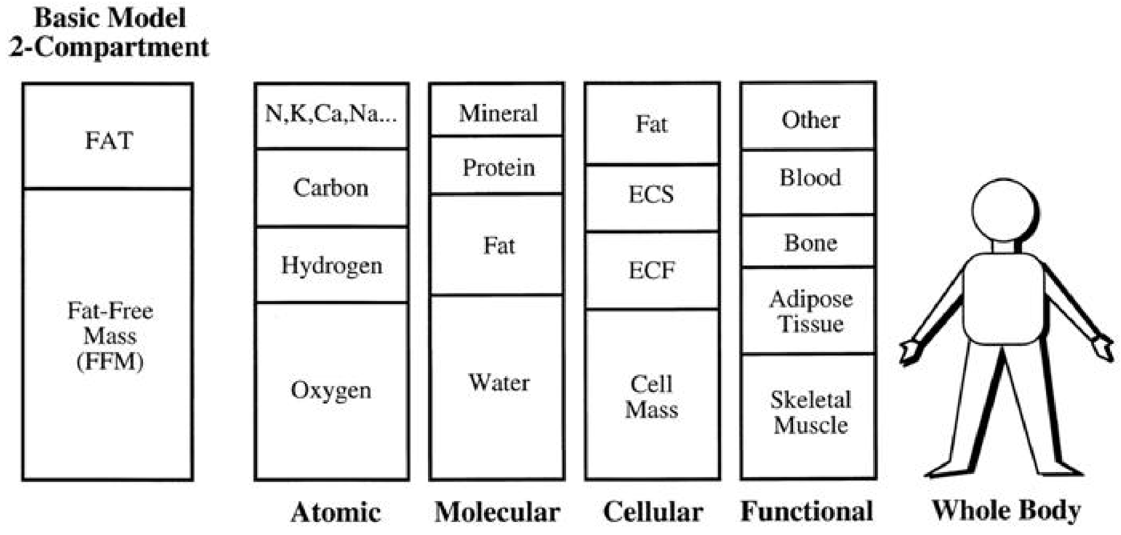


Figure 2.1: Body composition model [10]

2.2 Electrical properties of tissue

The conductivity of the body is ionic, the reason for this is the existence of ions in the intra- and extracellular liquid, two of the most important ions are Cl^- and Na^+. In ordinary electronics, conduction in metals is a flow of free electrons in the metal, the conduction in the body is quite different because the conduction is of the form that the ions are transported around in the intra- and extracellular liquid. This leads to concentration changes. Tissue is composed of cells, these cells have unpolare membrane. The cell membrane consists of two layers of fosfolipid, this fosfolipid has polar head, but the two tails consist of fat and is therefore unpolar, which makes the membrane a bad conductor for ions. This property gives the cells a capacitive property. Because of these capacitive properties of the cells, tissue can be seen on as a dielectric (but tissue can also be seen on as a conductor, muscle tissue is more like a conductor with capacitive properties, while stratum corneum is more like a dielectric with some conducting properties).

Because of the tissue's dielectric property, we can use a model of a capacitor to set up some basic expressions for the admittance and impedance (impedance $Z = \frac{1}{Y}$) for this capacitor. If we have a capacitor where A is the plate area and L is the distance between the plates, the admittance for this capacitor is then given by:

$$Y = G + j\omega C \tag{2.1}$$

Where G is the conductance given by $G = \sigma' A/L$ [S] The electrical properties of tissue vary considerable depending on it's structure. The complex impedance is given by [1].

$$Z = R + jX = R - \frac{j}{\omega C} \tag{2.2}$$

where the last part is valid for a R-C series combination.

$$Y = \frac{1}{Z} = G + j\omega C \tag{2.3}$$

Taking the variations of the properties into account the specific admittance called admittivity is given by

$$y = \sigma + j\omega\epsilon_0\epsilon_r \tag{2.4}$$

where σ is the conductivity of the tissue locally, ϵ_0 the dielectric constant of free space, ϵ_r the local tissue permittivity. To illustrate:

From 2.3 to 2.4 the relations

$$C = \epsilon_0 \epsilon_r \frac{A}{d}, \; R = \sigma \frac{A}{d} \tag{2.5}$$

can be used;

Similarly we have the specific impedance, called impedivity

$$z = \rho = \rho - \frac{j}{\epsilon_0 \epsilon_r \omega} \tag{2.6}$$

Eq. 2.5 and 2.6 shows that local variations in σ, ρ and ϵ_r will give variations in admittivity and impedivity. The electrical properties of tissue do in general vary with frequency and with variations in the permittivity ϵ_r, as illustrated in Figure 2.2. When tissue is damaged in any way, or

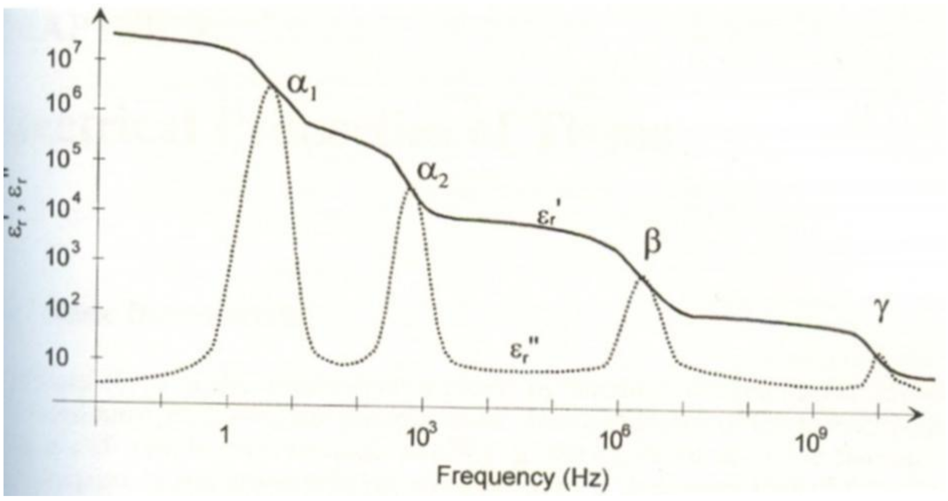


Figure 2.2: Dispersion regions for tissue [1], results in a frequency dependent permittivity

objects like tumors have developed, the tissue locally will have different properties including the dielectric properties. This leads to a modification of the electric field distribution and the impedivity is also changed. Measurements on normal and cancerous human breast tissues have led to the conclusion that there are significant differences in electrical impedance between normal and malignant human breast tissues [12]. The malignant tissues show a lower impedivity. The changes relative to healthy tissue are attributed to increased water and salt content, changed membrane permeability and packing density as well as orientation of the cells.

2.3 Impedance of the body and living tissue

Starting with a resistivity ρ (Ωm) and a conductivity σ=1/ρ (S/m) the total resistance of a cylinder of homogeneous material of length L and area A is given by

$$R = \rho * L/A = \rho * L^2/V \tag{2.7}$$

Where V is the volume of the water with conductive ions. The water volume is given by

$$V = \rho \frac{L^2}{A} \tag{2.8}$$

This equation was used by Hoffer et al. [5] making measurements on 20 volunteers with connections to the right hand and the left foot (obtaining the greatest length of the conductor). In addition, body height, weight and wrist circumference were measured. Measurements were performed at 100kHz. Total body water volume was also determined by radioisotope-dilution measurement and a regression equation on the form

$$TBW = AT^2/Z + B \tag{2.9}$$

was empirically derived from the volunteers. Comparing with 2.8 indicates that ρ is included in A and T^2 corresponds to L^2. B is not commented on in this article. In later development there has been focus on improving the algorithm as illustrated in Table 3 in the paper by Kyle et al. [7] where bioimpedance analysis equations reported in the literature between 1990 and 2004 are listed.

The algorithms are now more sophisticated, taking weight, age and gender into account in slightly different ways. Similar types of equations are listed for body cell mass (BCM), intercellular water (ICW), extracellular water (ECW), body fat (BF) and fat free mass (FFM). However there are few choices for ICW and BCM, meaning that ICW must be found from TBW and ECW, and that there may be more problems determining the body cell mass.

Our main task is to make measurements of the body or living tissue. The body consists mainly of water containing dissolved ions like Na^+ and K^+ making it highly conductive. There are two main types of current flowing as a response to an excitation voltage, the extracellular current and the intracellular current.

The cell membrane represents a capacitance and for low frequencies (dc) there will be no current penetration. At higher frequencies current flows in and out of the capacitors at the membrane borders. The body can be modeled as an intracellular part (resistor capacitance in series) in parallel with a resistor representing the extracellular part as shown in Figure 2.3. This model was originally proposed by

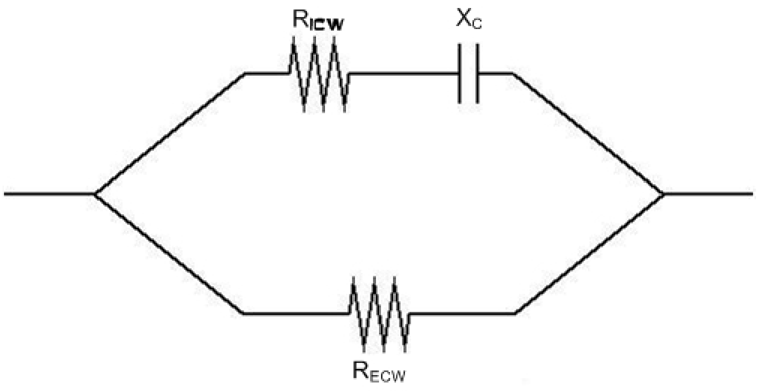


Figure 2.3: Model of the body represented as an electrical circuit

Fricke [6] and is referred to as Fricke's circuit. Here X_c represents capacitance of the membrane barriers, R_{ICW} the intracellular fluid (water) where conductivity is mainly provided by dissolved K^+ ions. For the extracellular resistance R_{ECW} its mainly Na^+ ions dissolved in water.

There are other more complex models consisting of several resistors and capacitances and a pure resistor capacitance series is also used. For the Fricke model the current at low frequencies is dominated by the extracellular contribution R_0. At high frequencies the body resistance reflects full contributions from both extracellular and intracellular contributions. The value is then R_∞ representing R_{ICW} and R_{ECW} in parallel. The resistance is commonly plotted versus the negative reactance value [4], in a Cole plot as shown in Figure 2.4. In the plot the effect of the dielectric relaxation τ_Z used in the Cole empirical equation 2.10

$$Z = R_\infty + \frac{R_0 - R_\infty}{1 + (j\omega\tau_Z)^\alpha} \tag{2.10}$$

resulting in a displacement of the center of the semicircle from the real axis. Kyle et al. [7] have reviewed the methods, used for bioelectric impedance analysis of the body in determining body composition. The basic methods used can be summarized as:

2.3.1 Single frequency bioelectrical impedance analysis (SF-BIA). For the single frequency measurement and analysis it is common to

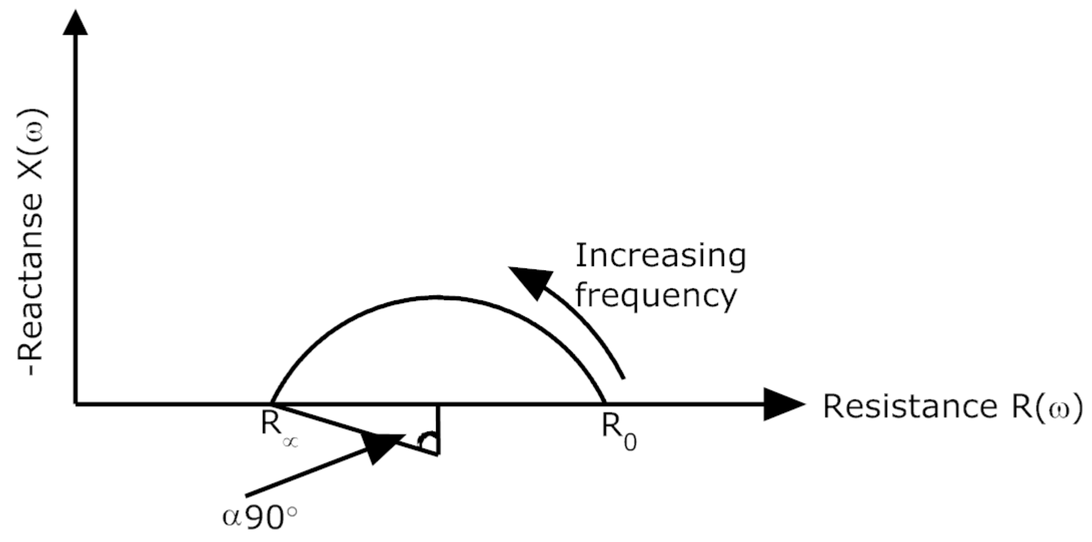


Figure 2.4: Illustration of the frequency dependency of the impedance of living tissue

make measurements at 50kHz using surface electrodes on hand and foot. At 50kHz the current passes through both the intra- and extracellular fluids with some individual variations. Thus the measured resistivity is a weighted sum of extracellular (full contribution) and intracellular water resistivities. SF-BIA is used to estimate FFM and TWB based on empirical equations. The conditions under which this type of equations are valid must be carefully observed and taken into account.

2.3.2 Multi frequency bioimpedance analysis (MF-BIA). This Method then includes impedance measurements at different frequencies, typically 1, 5, 50, 100, 200 and 500kHz. However it has been observed that reproducibility is poor for frequencies below 5kHz and above 200kHz, especially for the reactance at low frequencies [7, 9]. The AD5933 based device presented here does cover the interesting region up to 100kHz, and will be tested also in the multi frequency mode. The multi frequency approach allows evaluation of FFM, TBW, ICW and ECW.

2.3.3 Segmented bioelectrical impedance analysis. Bioimpedance is frequency dependent (because of the cell's capacitive property), and measuring the bioimpedance over several frequencies (henceforth bioimpedance spectroscopy) can give valuable information about tissue and membrane structures as well as intra- and extracellular liquid distributions. Because of the tissue's capacitive properties the bioimpedance will decrease with increasing frequency. While whole body bioimpedance analysis (BIA) focuses on the totality, the localized BIA looks at specified body segments like an arm or a foot. One could for instance imagine that a circulation problem or a damage in one of the feet would have an influence on the local impedance. This is a technique which is being investigated by NASA for surveyance and study of astronauts during preparation for space missions [13]. The problem is that the lack of gravitational force causes a redistribution of fluids in the body (i.e., from thighs toward head) which severely may affect performance.

2.3.4 Electrical Impedance Scanning. Electrical Impedance Scanning is a method used for investigation of local variations. One area of interest is searching for and investigation of possible malignant tissue, where the impedivity will locally be reduced relative to healthy surrounding. This problem seem to have been specially focused on in the USA, where there are restrictions on the use of mammographic screening for women under 40 years [14].

In the recent ICEBI-conference in Graz there were several papers focusing on the use of electrical impedance tomography (EIT) exemplified by [15, 16, 17].

Scanning measurements require a set of electrodes [18] as illustrated in 2.5. With excitation at one electrode pair, registration is performed at all the other electrodes. When one registration is ended the excitation is moved systematically to a new electrode position and so on. This creates an enormous amount of data, and the challenge is the data processing needed to give reliable results.

2.4 Application of bioimpedance measurements

2.5 Multivariate analysis

After the whole body measurements was completed I used multivariate analysis with the measured results, to see if there is a correlation between the measured data (impedance and phase) and the boddy mass index (BMI). For this I used a program called The Unscrambler from a company

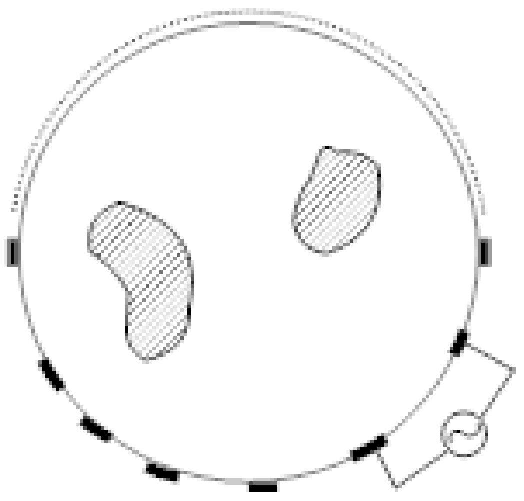


Figure 2.5: A measurement object, with regions exhibiting changed impedivity equipped with electrodes for excitation and detection.

named Camo. In the multivariate analysis I used a multivariate analysis named PLS1 regression analysis.

2.5.1 Partial Least Squares (PLS) regression analysis

PLS regression analysis is a method for relating the variations in one or several response variables (Y-variables) to the variations of several predictors (X-variables), with either explanatory or predictive purposes. So in the data set the BMI is the Y-variables while the impedance and the phase is the X variables which I want to find a correlation with the Y-variables, or so to say predict the Y-variables from the X-variables.

PLS1 regression analysis is a version of the PLS method with only one Y-variable, there are other versions like PLS2 that have 2 or more Y-variable's also but in my measurements I have only used one (BMI).

Chapter 3

A bioimpedance measurement system based on the integrated circuit AD5933

3.1 Description of the system and its main modules

The system to be built should be a complete system, from measurement to presentation of final results. Figure 3.1 shows a block diagram for the main modules of the bioimpedance measurement system consisting of the AD5933 card, a microcontroller for communication with the AD5933, and a laptop (PC) communicating with the microcontroller and presenting results. The software package developed during the work consists of one part for the microcontroller and one part for the PC. Outer connections to devices under calibration or measurement are included in the figure. The hardware blocks and their integration into the system are described in the following sections (i.e., 3.2 - 3.6), and the software package is described in Chapter4 with the full code in Appendix A for the microcontroller code and Appendix B for the graphical interface code.

3.2 The AD5933 integrated circuit and its functionality

The AD5933 is a high precision impedance converter system with an internal converter system and an internal DDS (Direct Digital Synthesis) frequency generator for provision of the signal used for excitation of

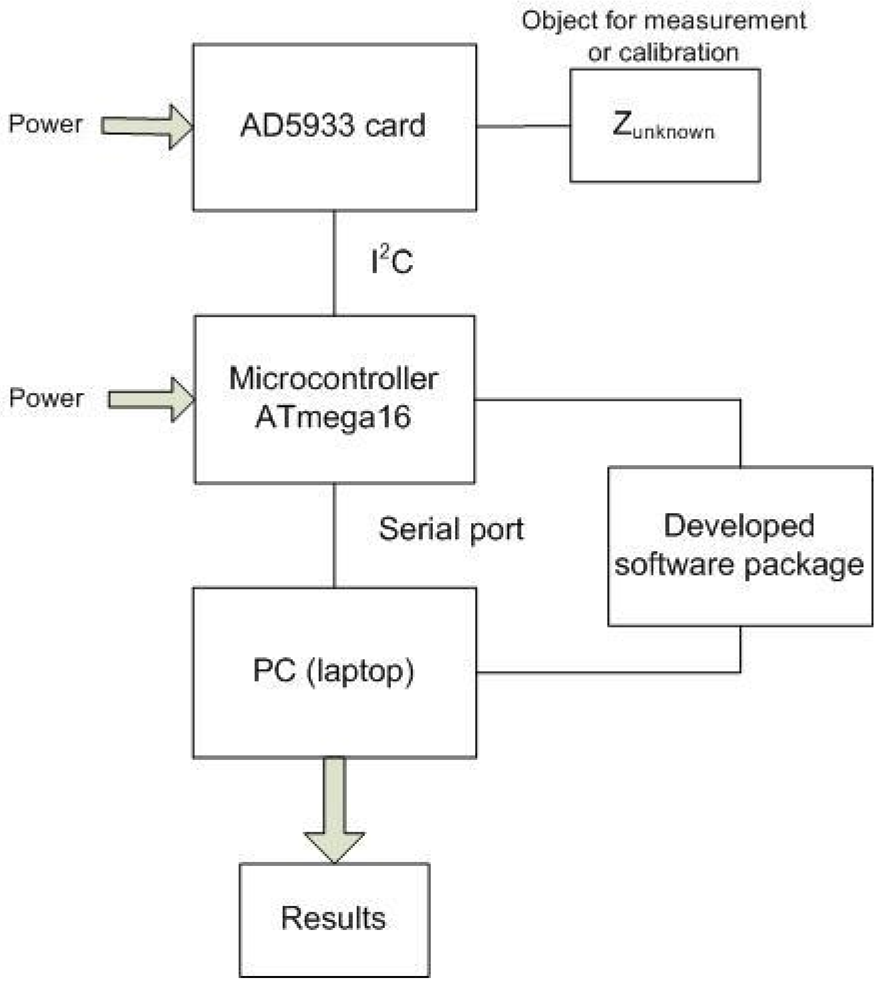


Figure 3.1: Main modules of the bioimpedance measurement system

the impedance being tested. The response signal from the impedance is amplified and then sampled by a 12 bit, 1 MSPS ADC (Analog to Digital Converter). A discrete Fourier transform (DFT) is performed by using 1024-point DFT processor included in the circuit. This DFT provides a real and imaginary number for each frequency.

The system can be used to perform impedance measurements from 0.1kΩ to 10MΩ. However, due to operational requirements on calibration and excitation voltage this measurement domain is divided into 6 ranges, range 1-6. The frequencies of operation is from 3kHz to 100kHz, where the low frequency is dependent on the reference oscillator frequency and stability. A block diagram of the IC AD5933 is shown in Figure 3.2.

Here are also two external components included, the device under test representing an impedance $Z(\omega) = R + jX(\omega)$, and the resistor R_{FB} which is the reference resistor for a selected measurement range. It is used for calibration with $Z(\omega)$ replaced with a R_{cal} of the same value as R_{FB}. R_{FB} is then kept for all measurements in the range where it has been used for calibration. From an operational point of view the AD5933 consists of

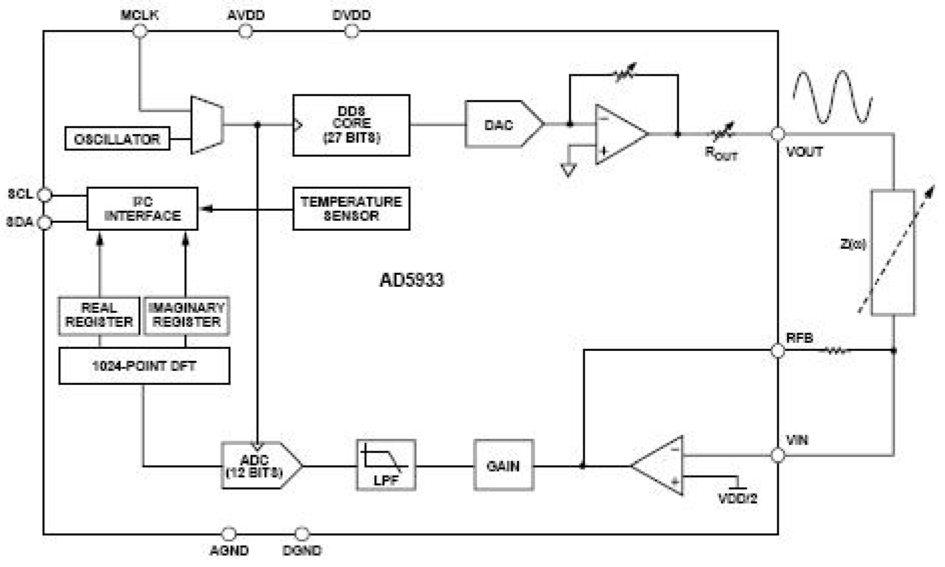


Figure 3.2: Functional block diagram of the AD5933 [2]

the following main parts:

- the transmitter part generating the signal employing a DDS using either an external oscillator (MCLK) or an internal reference oscillator, a DAC transforming the digital signal into a sinusoidal signal used as excitation signal for the device under test. The signal from the DAC is sent through a programmable gain stage where the peak-to-peak output voltage is selected. The output signal can be stepped in frequency from the operator's control, giving a start frequency, the frequency increment and the number of increments.

- when the device under test is excited by this voltage a response current is set up. This is the input to the current-voltage amplifier in the receive stage. This amplifier is followed by a programmable amplifier (PGA) with gain options of 1x and 5x. Figure 3.3 shows the receive and on board data handling stage in more detail (this figure is taken from the AD5933's datasheet but modified slightly). The feedback resistor R_{FB} (between V_{in} and RFB) is assumed identical with the resistor used for the calibration. The reason for this is discussed later. This resistor and the gain setting resistor of the PGA-stage determines the signal level at the ADC input. It must be kept within the linear range of the ADC (0V-V_{DD}) to avoid faulty operation. We also see that the voltage values in the system will change when $Z(\omega)$ is varied. This is compensated for by the possibilities for changing the two variable resistors in the transmit stage, the feedback resistor and the out resistor R$_{out}$. The system is then operated in 6 ranges with different output excitation voltages and different values of the calibration resistor and the feedback resistor R_{FB} of the current to voltage amplifier. The PGA gain is normally set to 1x. Going to 5x could cause problems with saturation because this will increase the voltage from the current-to-voltage amplifier, and if the voltage to the ADC goes out of it's linear range (0V-V$_{DD}$) the ADC will become saturated. The digital data from the ADC are fed into the digital signal processing part of the system where a discrete Fourier transform (DFT)

$$X(f) = \sum_{n=0}^{1023} \left(x(n) \left(\cos(n) - j \sin(n) \right) \right) \tag{3.1}$$

is performed using data from 1024 samples for each frequency point. The output data is stored in two 16 bits registers for real and imaginary components respectively in twos complement format. The impedance value for the device under test is calculated from these data, using a calibration procedure comparing the magnitude with a known resistor value.

3.3 The microcontroller and its functions

I have used the ATMEL AVR series of microcontrollers, because I had some experience with these from before and because they are well documented.

3.3.1 The ATmega16 microcontroller

I have used the ATmega16 microcontroller for this project, this microcontroller has large enough flash memory (16kB memory space) and has the

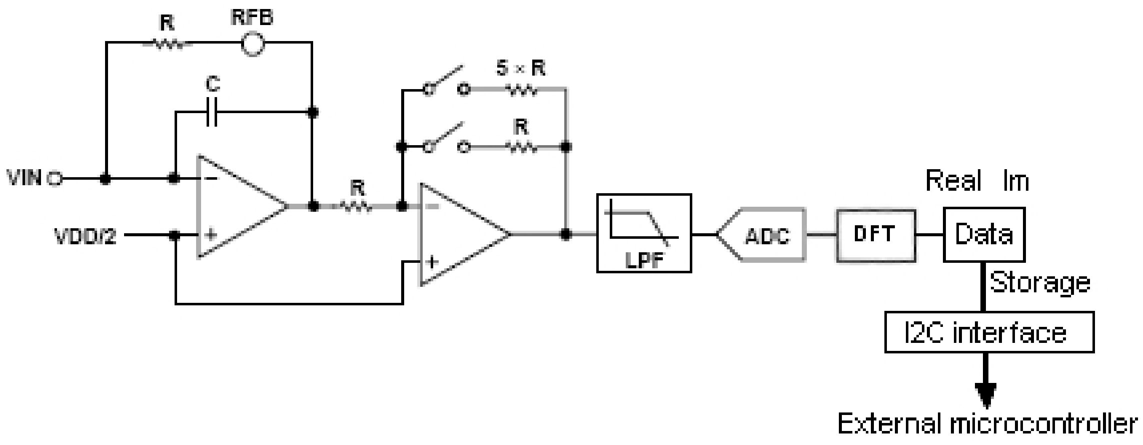


Figure 3.3: Details of the AD5933 receive and data handling stage

needed peripheral connections (serial port and I^2C/TWI). A discussion on different parts of the ATmega16 that was used in the project is given here together with a short introduction to ATmega16 in general. For features not used in the thesis work or for more details, I refer to the ATmega16 datasheet [19]. The ATmega16 8-bit microcontroller in the AVR family is based on the AVR enhanced RISC (Reduced Instruction Set Computer). With less instructions and compilators that are optimized for the RISC architecture, a RISC processor may be quicker than more complex processors because the simplified instructions can be executed faster.

Now I move on to the features used in the thesis work.

The Universal Synchronous and Asynchronous serial Receiver and Transmitter (USART)

The ATmega16 USART features were used to communicate with the PC via the serial port. On the STK500 start board there is an integrated circuit MAX202 that handles the transition from the transistor-transistor logic level to RS232 level.

The USART takes bytes of data and transmits the individual bits in sequential order, at the destination (PC) the bits are reassembled to a byte by the USART.

It is possible to use either synchronous or asynchronous mode, the synchronous mode require that the sender and receiver have a common clock, while in asynchronous mode this is not necessary. In asynchronous mode is used in the code.

The serial port is in a process of being obsolete and removed from new products. The serial port has been replaced by the USB for most applications. A possible solution to add the USB interface to this measurement system could be to use a integrated circuit FT232R from a company named FTDI. The FT232R converts the serial UART from the microcontroller to USB, this could be used instead of the MAX202. This would not require any change in the software but would require a proper driver to be installed on the PC to emulate the USB as a serial port.

Inter-Integrated Circuit (I^{2}C)

The IC AD5933 uses a I^{2}C bus for communication with a microcontroller, the I^{2}C bus is a synchronous bidirectional serial bus that provides an efficient method for data exchange between devices. In this setting the microcontroller is the master, while the AD5933 is the slave.

The master start a transmission by sending the 7-bit address of the slave and a write/read bit to the bus. The write/read bit is 0 when a write operation is being started, and 1 when a read operation is being started. Then the slave returns an ack for received address. If the write/read bit was set low, then a writing process is started by sending one or several bytes to a specified register location in the slave device, or if low a reading process is started by reading one or more bytes from a specified register location in the slave device. After the transmission is completed the master generates a stop condition on the bus which then ends the transmission between the master and the current slave. There is also some useful information on this in the AD5933 datasheet [2].

3.3.2 The development board STK500

The STK500 is a development kit for the Atmel AVR series of microcontrollers. With this board you can program most of the AVR microcontrollers and access the microcontrollers I/O with headers on the board. These headers can then be connected with wires to some peripheral, either on the board itself (push buttons, LED's, and serial port) or externally. I have in my thesis used the on board serial port connection and then connected to AD5933 externally with 3 wires (2 for I^2C and 1 for ground) going from the STK500 board to the breadboard and the AD5933. For more information about the STK500 starter board see [20]

3.4 The PC and its role in the controll system

The PC has the following main tasks in the controll of the system:

- It runs the graphical interface for operation of the system.
- It receives data from the microcontroller via the serial port.
- It controls calibration runs and stores calibration data.
- It performs correction of measured values based on the calibration data. We then obtain calibrated values for the real and imaginary parts from the DFT. The magnitude is given by eq. 3.2

  $$Magnitude = \sqrt{real^2 + im^2} \tag{3.2}$$

  and the phase is given by eq. 3.3.

  $$\theta = tan^{-1}\,(im/real) \tag{3.3}$$

- It stores and present the result.

3.5 The analog digital converter

This module has a 12 bit resolution and a sampling rate of 1 MSPS. It has turned out to be a critical module in the system due to the limited linear range of 0V - VDD. I f the signal in exceeds this value the ADC is in saturation and the data sent from the ADC to Digital Signal Processor performing the DFT are not representative for the values of the impedances being tested.

3.6 The direct digital synthesis (DDS) and frequency of operation

Direct Digital Synthesis is generating a function from digital values of the function generated for selected time intervals and stored in a random access memory (RAM). The reference clock controls the read out to the DAC which generates the analog output. The low pass filter following the DAC filters out higher frequency components. The reference clock is provided internally at 16.776MHz. An external clock reference may give better stability. The reference clock also controls sampling time of the receive part ADC. In this configuration the system is specified to operate from 100kHz and down to 5kHz, but we have seen functional operation down to 3kHz. The upper frequency is set by the low pass filter. For operation at lower frequencies than 5kHz, the problem is the sampling time of the ADC [3]. It is recommended to scale down the clock frequency using a clock frequency of 2MHz, the useful frequency range would be 5kHz-300Hz. The upper frequency of operation, and the sweep span is also reduced.

Chapter 4

Software for operation of the system

In this section I will describe the programs used, and also explain in some more detail my own developed software. Please also see Appendix D for a flowchart that shows how the software works.

4.1 Used software

Here I list all the programs used in the development, with a description of the programs and some of their main features.

4.1.1 WinAVR

This is not a single program, it's more like a collection of several programs and header files. The header files can be included in the code and then add some functions that can be used in the code (like functions for interrupts, i/o, delays, etc.), while the programs contain tools for compiling the code, simulating and also programming the compiled code into the microcontroller. There is also included an editor (Programmers Notepad). The compiler in the WinAVR is based upon the GNU GCC compiler.

4.1.2 AVR Studio 4

AVR Studio 4 is an Integrated Development Environment (IDE) for Atmels AVR series of microcontrollers. One can use AVR studio for writing, debugging and simulating code, and also program the code to the microcontroller. Personally I have only used the simulating and the programming features.

4.1.3 Notepad++

I have used this editor to write all my code in. It provides a tabbed interface, this way several files can open at the same time, and it's easy to switch between the different source files which are opened. I found this very practical since my C code is composed of several files (one file contains all the USART routines, the other contains all the I^2C routines and the last contains the main function). It also supports syntax highlighting, line numbering and some other features.

4.1.4 Tera Term

Tera Term is a program used for serial port communication, used for sending the start parameters to the microcontroller and receive the data. This way I could test the operationality of the I^2C connection. When I received the data I received the value from the real and imaginary register, which I then could calculate the impedance and phase from. This was a lengthy process. Once I had confirmed that the connection worked correctly I wrote a GUI that made this operation more user friendly, especially since the GUI automatically calculated impedance and phase, and then plotted the result.

4.1.5 Portmon

Portmon is a program that monitors ports on the computer, I used this in the start to debug some problems with the serial communication between the PC and the microcontroller. Portmon displays all sent and received data.

4.1.6 Python

I have written my GUI using the Python scripting language. Python offers wide support from libraries and is well suited for efficient prototyping.

4.2 Developed software

Here I will give a description of the code and provide detailed explanation of some key points in the code. There are also some comments with explanation but here some key points are considered.

4.2.1 C code for microcontroller

I split this section into three parts, so that each part explains parts from the cited program file. The reason for this is to avoid confusion about which part belongs to which file.

myusart.c & myusart.h

The file myusart.c is the file that contains all the function for using the USART, while the myusart.h contains some definitions and declarations of all the functions in myusart.c so that if I include the myusart.h in some code I can use all the functions in myusart.c. I do not think myusart.h needs any further explanation, so I go right into explaining the key parts of the code in myusart.c.

```
{
  UCSRB |= (1 << RXEN) | (1 << TXEN);
  UCSRC |= (1 << URSEL) | (1 << UCSZ0) | (1 << UCSZ1);

  UBRRL = BAUD_PRESCALE;
  UBRRH = (BAUD_PRESCALE >> 8);
  return 1;
}
```

So this is the function used to initialize the USART. On the second line I set the RXEN (receiver enable) and TXEN (transmitter enable) in the UCSRB (usart control and status register B) high, this will activate both USART receiver and transmitter. This also overrides the normal port operation of pin 0 and pin 1 of PORTD on the microcontroller. On the next line I set the URSEL (register select), UCSZ0 and UCSZ1 in the UCSRC high, setting URSEL high will activate writing to the UCSRC. The reason that I need to pull URSEL high is because UCSRC shares the I/O location with the UBRHH (usart baud rate register), and I can only access UCSRC if URSEL is pulled high. In table 4.1 all the bits in the UCSRC is shown, I also included the bits that I write to the register.

Table 4.1: Table of the bits in the UCSRC

Bit	URSEL	UMSEL	UPM1	UPM0	USBS	UCSZ1	UCSZ0	UCPOL
Writing	1	0	0	0	0	1	1	0

I will now explain further about each bit, except the URSEL bit which is already covered. The UMSEL (usart mode select) bit will if pulled high set the operation mode to synchronous, while leaving it low as I do will set the mode to asynchronous operation. In my design I have chosen asynchronous operation, mostly because it is simple to work with and I have some experience with it from before. UPM0 and UPM1 selects the parity mode, either none, even or odd parity. When both are left low like in my code, the parity function is disabled. If both were set high the mode would be set to odd parity, or if UPM1 were set high and UPM0 set low the mode would be set to even parity operation. The USBS sets the number of stop bits, if left low there will be 1 stop bit, and if pulled high there will be 2 stop bits. The shown setting of UCSZ1 and UCSZ0 sets the character size (number of data bits) for the receiver and transmitter to 8 bit. The UCPOL bit is only used in synchronous operation, so in my case I leave it low. In synchronous mode this bit sets the relationship between the transmission change and the data input sample (so one can select which option at either falling or rising CLK edge).

In the next two lines I write the desired baud rate code to the two baud rate registers, it should be noted that BAUD_PRESCALE is defined in myusart.h as:

```
#define BAUD_PRESCALE (((F_CPU / (USART_BAUDRATE * 16UL))) - 1)
```

Where F_CPU is defined as 3686400 in the makefile (this is the microcontrollers clock frequency), and USART_BAUDRATE is the desired baud rate which I have set to 9600. The UBRRL register should contain the eight least significant bits while the UBRRH should contain the four most significant bits. This is the reason I need to right shift the bits eight times to get the four most significant bits to the UBRRH register (hence the right shifting in the code). So with this I think I have explained this function, and I therefore move on to the next function which handles transmission of one byte.

```
void USART_transmit(char data)
{
  while ((UCSRA & (1 << UDRE)) == 0) {};
  UDR = data;
}
```

So what this function does is first to wait until UDRE (in the UCSRA register) is set to high. When UDRE is set to high this means that the buffer is empty and that one therefore can write new data to the USART. So after the UDRE bit is set high the UDR is assigned to a new value that is to be transmitted.
The next function handles receiving of one byte.

```
int USART_receive(void){
  while ((UCSRA & (1 << RXC)) == 0) {};
  return UDR;
}
```

This function first waits until the RXC bit is set to high, which then indicates that there is unread data in the receive buffer. When this happen I then return the received data.

The next function is more or less based on the function USART_transmit but it incorporates the possibility to transmit several bytes stored in a pointer.

```
void USART_CharTransmit(char* data)
{
  int n;
  n=0;
  while (1)
  {
    while ((UCSRA & (1 << UDRE)) == 0) {};
    UDR =*(data+n);
    n++;
    if(!(*(data+(n)))){
      break;
    }
  }
  USART_transmit('\n');
  USART_transmit('\r');
}
```

First I declare an int variable that is then set to zero, this int is going to be used as counter to increment the pointer so that the next byte in the pointer can be transmitted. A loop that is only exited when there are no more data in the pointer to be sent is then established. The transmission is done in the same way as in the USART_transmit function. At the end, the commands for new line and carriage return are transmitted.

The next and last function, myusart.c, handles receiving of several bytes. To know when to stop reading more data it is implemented so that it stops if the received byte is a carriage return.

```
char* USART_CharReceive(void){
  int i;
  char* data;
  char temp;
  data = (char*) malloc(30*sizeof(char));
  for(i=0; i<30; i++){
    temp =USART_receive();
    if(temp=='\r'){
      break;
    }
    *(data+i)=temp;
  }
  *(data +i)='\0';
  return data;
}
```

An int variable, a char pointer and a char variable are then declared. The char pointer will be used to store all the received data in, while the char variable temp will be used for temporary storing the data. The int variable will be used to increment the pointer so that each received data don't overwrite the last. I then allocate memory for the pointer and enter a for-loop, which for every iteration reads one byte and stores it in the temp variable. Then it checks if the temp variable contains the symbol for carriage return. If it does, the loop is exited without saving the carriage symbol, otherwise the received byte is saved to the pointed location. When the loop is finished the pointer is returned.

mytwi.c & mytwi.h

These two files contain all the functions used in the I^2C communication. I was without any experience with I^2C from before, so this is the part of the code I used the most time on to get it to work correctly. I start with describing the function that initializes the I^2C communication.

```
int TWI_init(void){
  TWBR=10;
  TWCR=0x04;
  return 1;
}
```

A value of ten is first assigned to the TWBR register, this register sets the frequency for the SCL. TWBR can be calculated from the following equation given in ATmega16's datasheet.

$$Freq_{SCL} = \frac{Freq_{CPU}}{16 + 2(TWBR) * 4^{TWPS}} \tag{4.1}$$

From eq. 4.1 we then get:

$$Freq_{SCL} * (16 + 2(TWBR) * 4^{TWPS}) = Freq_{CPU} \tag{4.2}$$

$$16 + 2(TWBR) * 4^{TWPS} = \frac{Freq_{CPU}}{Freq_{SCL}} \tag{4.3}$$

$$2(TWBR) * 4^{TWPS} = \frac{Freq_{CPU}}{Freq_{SCL}} - 16 \tag{4.4}$$

$$TWBR = \frac{\frac{Freq_{CPU}}{Freq_{SCL}} - 16}{2 * 4^{TWPS}} = \frac{\frac{Freq_{CPU} - 16 * Freq_{SCL}}{Freq_{SCL}}}{2 * 4^{TWPS}} \tag{4.5}$$

$$\underline{TWBR = \frac{Freq_{CPU} - 16 * Freq_{SCL}}{2 * 4^{TWPS} * Freq_{SCL}}} \tag{4.6}$$

So with this equation the value to be programmed to the TWBR can be calculated. I decided to use a SCL frequency of 100kHz. So I can then easily calculate the value to be programmed to the TWBR register by inserting all the values in the equation. TWPS can operate as a prescaler but that feature is not used. TWPS is set to zero in my calculations.

$$TWBR = \frac{3686400Hz - 16 * 100000Hz}{2 * 4^0 * 100000Hz} = \underline{\underline{10,432}} \tag{4.7}$$

I then know I have to program the value 10 to get a close SCL frequency to what I wanted, the frequency will be a little higher since the number is rounded down to an integer.

Next I set the TWCR register to the hex value 0x04, this sets the TWEN (TWI enable) bit high. When this is set high the TWI is enabled and takes the controll over the SDA and SCL pins.

The next function is just a loop that waits until the TWINT bit in the TWCR is cleared, since this indicates finished current job.

```
void TWI_wait(void){
  while(!(TWCR &(1<<TWINT))){

  }
}
```

So this loop goes until the TWINT bit is set, when this loop is finished the transmission is complete.

The next function handles sending the start condition to the bus that sets the microcontroller as the master on the bus.

```
unsigned char Send_start(void)
{
  TWCR=START;
  TWI_wait();
  if((TWSR & 0xF8)!=0x08 || (TWSR & 0xF8)!=0x10)
    return TWSR;
  return 0xFF;
}
```

START is defined in the code to the hex value 0xA4. For the microcontroller to claim to be Master it needs to sets the TWSTA bit in the TWCR register high. Then I wait until the TWINT flag is cleared since this indicates a finished transmission. Then the status register is checked to see if the start condition (or the repeated start condition) has been transmitted successfully. If it have, then 1 is returned, if not then the value of TWSR is returned (for possible use in debugging).

The next function issue a stop condition on the bus.

```
void TWI_stop(void){
  TWCR=Stop
}
```

Where Stop is defined in the code to the hex value 0x94. To generate a stop condition on the bus I must set the TWSTO bit in the TWCR register to high, which is done by assigning 0x94 to the TWCR register.

The next function I want to explain a bit deeper is the function that handles the transmission of the address of the slave device (in this case AD5933) on the bus.

```
unsigned char TWI_send_adr(unsigned char adr){
  TWDR=adr;
  TWCR=Trans;
  TWI_wait();
  if((TWSR & 0xF8)!= 0x18){
    return TWSR;
  }
  return 1;
}
```

Trans is defined in the code as the hex code 0x84. First I assign the address of the slave to the TWDR (TWI data register), the TWDR then contains the next byte to be transmitted. Before the address can be transmitted I need to clear the TWINT bit in the TWCR register by setting it to one. This will start the operation of the TWI and the byte in the TWDR will be transmitted on the bus. So the TWIN bit is cleared by setting the TWCR to the hex value 0x84. Then I need to wait until TWINT flag is set, since this indicates that TWI has finished the current job. To check if the transmission has been successful I also need to check if ACK from slave has been received. In this test I mask out the three last bits in the registers since these are not relevant for this test. If the test is successful I return 1, else I return the current value of TWSR.

The next function contains the routines for sending a byte to the TWI bus.

```
unsigned char TWI_send_byte(unsigned char data){
  TWDR=data;
  TWCR=Trans;
  TWI_wait();
  if((TWSR & 0xF8) != 0x28){
    return TWSR;
  }
  else{
    return 1;
  }
```

First I assign the data to be transmitted to the TWDR (TWI data register), data will then be the next byte to be transfered to the bus. For the data to be transmitted the TWINT flag needs to be cleared by setting it to one, which is done with $TWCR = Trans$. Trans is defined in the code to the hex value 0x84. Then I wait until the TWINT flag is cleared (which means that the byte has been transmitted), and then check if the status register (TWSR) of the bus contains an ack from the slave. If ack not received the value of the TWSR is returned, but if ack is received, 1 is returned to indicate success.

The next function contains the routines for setting the memory location in the AD5933 to write or read from.

```
unsigned char TWI_set_memloc(unsigned char mem_location){
Send_start();
TWI_send_adr(SLA_write);
TWI_send_byte(0xB0);
TWI_send_byte(mem_location);
return 1; /*Return 1 if succeeded*/
}
```

SLA_write is defined in the code to the hex value 0x1A. Calling the function Send_start will make the micocontroller the master on the bus. Next I send the address of the slave device and that I want to write to the slave device (so the last bit in the address is left zero to indicate that this is a write operation). Next I transfer the hex value 0xB0 to the bus, this will tell the AD5933 that the next byte is an address pointer (see page 26 in AD5933 manual). Then I transfer the register location that I want to read or write to. Then I return 1 back to signalize that the operation has been completed.

The next function handles writing 1 byte to the desired address (register) in the AD5933.

```
unsigned char TWI_byte_write(unsigned char reg_addr,\\
unsigned char data){
  Send_start();
  TWI_send_adr(SLA_write);
  TWI_send_byte(reg_addr);
  TWI_send_byte(data);
  TWI_stop();
  return 1;
}
```

The two first lines in the function are the same as the last function so they won't explain any further. Next I the send the address for the register I wish to write to, followed by the data I want to write to that register. At last the function TWI_stop() which will generate a stop condition on the bus is called.

The next function handles writing several bytes (or so called block write) to the specified register address in the AD5933.

```
unsigned char TWI_block_write(unsigned char reg_location,\\
unsigned char byte_number, unsigned char *TWI_data){
  int i;
  TWI_set_memloc(reg_location);
  Send_start();
  TWI_send_adr(SLA_write);
  TWI_send_byte(0xA0);
  TWI_send_byte(byte_number);
  for (i=0; i<byte_number; i++){
    TWI_send_byte(*(TWI_data+i));
  }
  TWI_stop();
  return 1;
}
```

An int variable which is to be used as a counter in the for-loop is first declared. Then I use the function TWI_set_memloc to set the starting address for the block write. The next two lines are already explained. Then I write the hex value 0xA0 to the AD5933, this hex value is the AD5933 command for block write so that the AD5933 will initiate the block write operation. Next I send the number of bytes I am going to transmit. Then I start a for-loop that goes through all the bytes to be sent. I then send the byte and use the counter i in the for-loop that goes to all the bytes to be sent to increment the pointer. At last I use TWI_stop() to send stop condition, and return 1 when operation is complete.

The next function is for receiving a byte from the AD5933.

```
unsigned char TWI_byte_read(unsigned reg_addr){
  TWI_set_memloc(reg_addr);
  Send_start();
  TWI_send_adr(SLA_read);
  TWCR=Trans;
  TWI_wait();
  return TWDR;
}
```

SLA_read is defined in the code to the hex value 0x1B. First I set the register location I want to read from by using TWI_set_memloc, then I transmit a start condition on the bus by calling the function Send_start(). Then I transmit the address of the AD5933 and with the last bit high to indicate that this is a reading operation. Then I enable transmission by clearing TWINT in TWCR register, and wait until TWINT is cleared as this indicate that the transmission is complete. The value received is now stored in TWDR so I therefore return the value of TWDR.

The next function is for receiving several bytes (block read) from the AD5933.

```
unsigned char TWI_block_read(unsigned char reg_add,\\
unsigned char byte_number, unsigned char *TWI_data){
  int i;
  TWI_set_memloc(reg_addr);
  Send_start();
  TWI_send_adr(SLA_write);
  TWI_send_byte(0xA0);
  TWI_send_byte(byte_number);
  TWI_init();
  TWI_send_byte(SLA_read);
  for(i=0; i<byte_number; i++){
    *(TWI_data +i)=TWDR;
    TWI_wait();
    TWCR|=(1<<TWEA);
  }
  TWCR=(0<<TWEA);
  TWI_wait();
  TWI_stop();
  return *TWI_data;
}
```

This function is pretty much the same as the TWI_block_write function except some key points that are different. The first one is on the 8th line where I issue a repeated start condition, which is required in the block read to set the read bit high. Then I again send the address but with the last bit high to initialize the reading operation. I then enter a for-loop where I store the received data in a pointer which is incremented with the counter i and wait until the TWINT bit is cleared and then an ACK is sent to the AD5933 device after each received byte. When the for-loop is completed a NACK is also sent to the AD5933 to signalize the last byte has been received. When the TWINT bit is cleared (NACK transmitted) and a stop condition is transmitted to the bus.

main.c

I have by now explained all the functions that I used in the main function. It remains to explain the main function. Instead of posting the whole main function I will rather point out some key points that are important. Comments are also given in the code. The first part of the code to be explained is the part where communication between the microcontroller and PC is started.

```
com:
val=USART_CharReceive();
if((i=strncmp(val, "Init",4))==0){
  USART_CharTransmit("AD5933");
}
else{
  goto com;
}
```

I start by defining a label with the name com. Next I use USART_CharReceive to receive a string from the PC. I then have an if-test to compare if the received string is equal to 'Init', if it is then the string 'AD5933' is transmitted back to the PC. If the string is not equal the code it goes back to where the label com is defined and repeat until the correct string is received from the PC.

The next part handles receiving and setting the start parameters of the AD5933.

```
startfreq=atoi(USART_CharReceive());
i=startfreq*32.0023195;
*data=0x000000ff & (i>>16);*(data+1)=0x000000ff & (i>>8);
*(data+2)=0x000000ff & i;
TWI_block_write(startfreq_reg, 3,data);

freqinc=atoi(USART_CharReceive());
i=freqinc*32.0023195;
*data=0x000000ff & (i>>16); *(data+1)=0x000000ff & (i>>8);
*(data+2)=0x000000ff & i;
TWI_block_write(freqinc_reg, 3, data);

numbinc=atoi(USART_CharReceive());
*data=0x000000ff & (numbinc>>8); *(data+1)=0x000000ff & numbinc;
TWI_block_write(incsteps_reg, 2, data);
```

```
TWI_byte_write(NumSet_high, 0x00);
TWI_byte_write(NumSet_low, 0x32);

val=USART_CharReceive();
if((i=strncmp(val, "V1",2))==0){
  j=1;
}
else if((i=strncmp(val, "V2",2))==0){
  j=2;
}
else if((i=strncmp(val, "V3",2))==0){
  j=3;
}
else if((i=strncmp(val, "V4",2))==0){
  j=4;
}
```

First the desired start frequency for the frequency sweep is read as a char string from the PC; it is then converted to an integer and stored in the int variable startfreq. This start frequency must be transformed to a start frequency code that the AD5933 will understand, this is done as shown in eq. 4.8 (this equation is from page 13 in AD5933 datasheet).

$$Startfreq\ code = \left(\frac{Startfreq}{\left(\frac{MCLK}{4}\right)} \right) * 2^{27} = Startfreq * \frac{4}{MCLK} * 2^{27} \qquad (4.8)$$

Now in my case I use the internal oscillator of the AD5933 which is 16.776MHz, if I set in the value for this oscillator frequency for MCLK in the equation I get the value I need to multiply my start frequency with to get the required start frequency code of the AD5933.

$$Startfreq\ code = Startfreq * \frac{4}{16.776MHz} * 2^{27} = 32.0023195 \qquad (4.9)$$

Therefore I need to multiply the start frequency with this value. Since the start frequency code is stored in three different address locations the value need to be splitted into three hex values which is done on lines 2 and 3. Next I use my block write function to write the three parts of the start frequency to the AD5933.

Then the frequency increment (how many Hz each incrementation increments the excitation frequency with) is received from the PC and undergoes similar treatments as the start frequency. Next the number of increments is read. This value does not need to be scaled, so I only need to split it in two parts that are written to each address in the AD5933 that handles the number of increments.

Next I set the number of settling time cycles, this value determines the number of output excitation cycles that are allowed to pass through the unknown impedance after each start, increment or repeat frequency command. I have not implemented this to be user set, but this could be done. At the moment this value is programmed to be 50 cycles (or $0x32$ in hex format).

In the GUI I have implemented a pull down menu that lets the user choose between different voltage of the excitation signal of the AD5933. So depending on which range the user choose the GUI will transmit either the string 'V1 , 'V2', V3' or 'V4'. Where V1=2.0Vp-p. V2=1.0Vp-p, V3=400mVp-p and V4=200mVp-p. So first the microcontroller read one of these strings. Then I use if-tests and compare the different strings to see which voltage range the user has selected. Depending on which string that match the variable, j is assigned different values which will then later be used in an if-test where the value determines which hex value will be written to the AD5933.

With the AD5933 you can also set the internal gain on the input signal (PGA gain). This can either be 1X or 5X, I have chosen to use 1X in my code since 5X more easily leads to saturation of the ADC in the AD5933. So the least significant bit in the MSB of the controll register is always set to 1 in my code.
I will now jump over many lines of code that I think are pretty well explained by the comments in my code already and instead explain the part where the data is received from the AD5933 and sent to the PC.

```
start:
  while(!(TWI_byte_read(status_reg) & 0x02));
  real_high=TWI_byte_read(real_high_reg);
  real_low=TWI_byte_read(real_low_reg);
  R=hextodec(real_high, real_low);
  itoa(R, s, 10);
  USART_CharTransmit(s);

  im_high=TWI_byte_read(im_high_reg);
  im_low=TWI_byte_read(im_low_reg);
  I=hextodec(im_high, im_low);
  itoa(I, x, 10);
  USART_CharTransmit(x);
  if((TWI_byte_read(status_reg) & 0x04)==0){
    if(j==1){
      TWI_byte_write(control_high_reg, 0x31);
    }
    else if(j==2){
      TWI_byte_write(control_high_reg, 0x37);
    }
    else if(j==3){
      TWI_byte_write(control_high_reg, 0x35);
    }
    else if(j==4){
      TWI_byte_write(control_high_reg, 0x33);
    }
    goto start;
  }
  else{
    TWI_byte_write(control_high_reg, 0xA1);
    goto com;
  }
}
```

First there is a while-loop that goes until the status register of the AD5933 is equal to 0x02, which means that the data in the real and imaginary data registers are valid. So when the data are valid the while-loop is finished and the microcontroller starts reading the the two real data parts. These two values are received as hex values, and to convert them to a decimal number I have to multiply the MSB (most significant byte) with 256 and then add LSB (least significant byte). This is exactly what the function hextodec does. The reason for why I must multiply by 256 is given by the following calculation.

$$\frac{0xFF00}{0xFF} = \frac{65280}{255} = 256 \tag{4.10}$$

So this gives that the MSB is 256 times as large in decimal value as the LSB. After this the values are then converted to ASCII, since I only can transmit char variables. In the conversion I have chosen to represent the value as a string with decimal base. I then transmit the converted value to the PC. The procedure is exactly the same for the imaginary data parts.

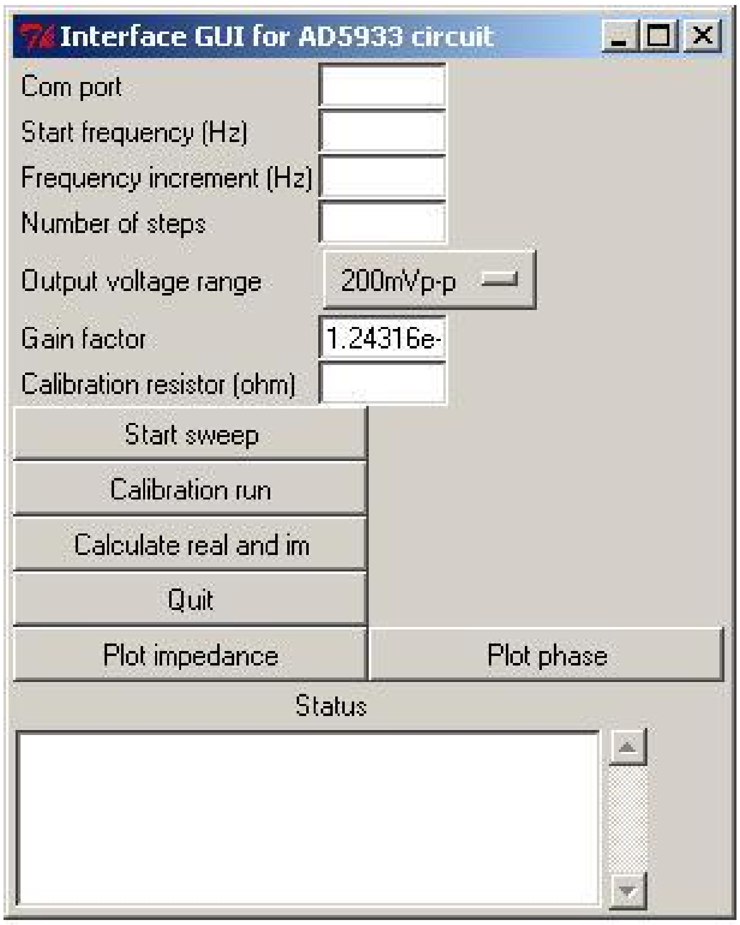
After both the real and imaginary values have been transmitted it is tested if the frequency sweep is complete (if it is complete the status register of AD5933 contains the hex value 0x04). If the sweep is not completed, I send an increment frequency command to the AD5933. Notice that all the if-tests are used to program the correct excitation voltage. If the frequency sweep is completed I program a power-down command, in this mode the V_{in} and V_{out} will be connected internally to ground so there will be no output excitation signal. When this is finished the code goes back to where the label com is defined and is therefore ready for a new run.

4.2.2 Python code for graphical interface

To make the AD5933 based measurement system more user friendly it was found that the best solution would be to make a simple GUI where the user could type in the desired start parameters, then run the sweep and finally get a plot of the result. The only experience I had with GUI's from before was with Python, so therefore I decided to use this as a basis for the new GUI for the AD5933 circuit. First I needed some libraries for the serial port communication with the microcontroller. I found a solution that was released under free software license (so that it can be used commercially and also can be modified), the library is named pySerial (can be downloaded from http://pyserial.sourceforge.net/).

I also wanted to have a plotting feature that let the user see the results after the sweep was completed without having to use time to import the data in Microsoft Excel. I first tried some of Python's inbuilt features on this but didn't find them satisfactory. I then searched the Internet for a library which supported better and easier plotting. I tried some and found one named matplotlib/pylab (can be downloaded from http://matplotlib.sourceforge.net/). This library provides a matlab like environment for plotting in Python.

Figure 4.1 shows the finished interface. So the graphical interface consists of some entry fields where the user can write which com port to connect to, the start frequency, the frequency increment (how many Hz the excitation signal will be incremented by for each step) and the number of increments. Then there is a pull down menu that lets the user choose between the four different excitation voltage ranges (2Vp-p, 1Vp-p, 400mVp-p or 200mVp-p). Then there are two more entry fields, the first lets the user input the gain factor to be used in the calculation of the impedance and the other lets the user type in the value of the calibration resistor. The value of the calibration resistor is used to calculate the gain factor for each frequency. These gain factors are then stored on the hard


Interface GUI for AD5933 circuit

Com port

Start frequency (Hz)

Frequency increment (Hz)

Number of steps

Output voltage range: 200mVp-p

Gain factor: 1.24316e-

Calibration resistor (ohm)

Figure 4.1: The finished graphical interface made with Python

drive as a file named gain_factor.txt. The user can open this file and choose from the different gain factors depending on which range the user is going to use. Normally you use the gain factor for the middle frequency of the range. So if the sweep goes from 1kHz to 10kHz you normally choose the gain factor for 5kHz.

Then there are a set of buttons, first is the button that starts the sweep. This button will call a function that first sends all the start parameters and then receive the result from the microcontroller. The next button is called calibration run and this is used to calibrate the system. This button calls a function that sends the start parameters to the microcontroller and the receive the result. From the result the system phase is calculated and stored to the hard drive in a file named calphase.txt. The gain factor is then calculated using the results and the user supplied value of the calibration resistor. The next button "'Calculate real and im"' is for calculating B, G, theoretical B and theoretical G for measurement data for a resistor/capacitor parallel measurement. This function requires the user to start the program from the command line since it requires the user to supply the value of the resistor and capacitor used. So doing a measurement with R=1kΩ and C=1nF the program would have to be started from the command line by typing:

$$AD5933GUI.py\ 1000\ 0.000000001 \tag{4.11}$$

Then after the sweep is completed one can click on the calculate real and im and a file named realim.txt will contain all the data. This file can for example be imported into Microsoft Excel and be used to make a plot.

The next button is the quit button, this terminates the graphical interface. The next two buttons are the plot impedance and plot phase, these two buttons call two different functions that either plots the impedance or the phase. At the bottom is a scrollable text box that will display some information about the progress of the sweep.

The code is now explained. Instead of posting the whole functions and classes here I rather refer to the code posted in Appendix B, the reason for doing this is that the code is somewhat long and that there are lots of definitions and declaration that doesn't need any further explanation. I will though include some code snippets that I think are important.

class AD5933GUI

This class contains all the functions used in the GUI, the communication with the microcontroller, and calculation and plotting of the results.

Constructor function

The first function in the class is the constructor, this is the function called when a new instance of the class is created. The constructor takes care of declaration of variables and graphical interface parts, such as frames (where the different widgets can be placed), buttons (which when clicked calls different functions), entry fields (where the user can fill in numbers) and so on. It also reads in the system phase from the file calphase.txt, which will then later be used to calculate the phase. I will now show an example of a declaration of a graphical interface widget to explain how these are declared. I will only show one but the others are much the same.

This widget is an entry field for the user to enter the number of the com port that the STK500 board is connected too.

```
self.port=Pmw.EntryField(self.bottom,
  labelpos='w',
  label_text='Com port',
  entry_width=8,
  entry_textvariable=self.com)
```

So I call this widget self.port (self is a reference to the instance), use a function in the PMW megawidget pack called EntryField. The first input to this function is which frame the widget should be in, in my case I have a frame called self.bottom that I am using for this entry field. Next input is the label position and I here choose west (w=west) so that the label will appear on the west of the entry field. Next input specify the label text and here I choose 'Com port'. Next I can select the width of the entry field and I set it to eight. The last input is which variable it is going to use to store the input from the user and I here use a variable called self.com to store the number of com port to be used. self.com is not an ordinary variable but is declared using a function from the TkInter toolpack, StringVar(). It is declared in the following way.

```
self.com=StringVar()
```

To get the stored value one then has to use another function get(), like self.com.get(). And to set the value from the code one has to use a function set(), like self.com.set('1'). To pack the widget in the selected frame you use a function from the TkInter toolpack named pack() is used.

```
self.port.pack(side='top', anchor='w')
```

So first I select to pack it in the direction of top (so that the widget will be under each others). And I choose to anchor it to the west part of the frame. The last part I will explain in the constructor function is the reading of the system phase from the file calphase.txt. The code for this is given under.

```
try:
  ifile=open('calphase.txt', 'r')
  trash=ifile.readline()
  for line in ifile.readlines():
    freq, phase=line.split()
    self.cal['Freq'].append(freq)
    self.cal['Phase'].append(phase)
except:
  print 'error 1'
```

The first thing you will note is the try and except, this functionality in Python will let you first try something and if that doesn't work instead of the program crashing you can have an except state where you either can try something else or print an error message.

realim function

This function calculates B and G and the theoretical values for them for a resistor in parallel with a capacitor circuit. The theoretical values are calculated from the resistor and capacitance value supplied by user from the command line. B and G are calculated from the received measurement values. The results are written to realim.txt.

calibration function

This function is used for calibration of the measurement system. This function calculates the system phase and gain factor and store them in two files on the hard drive.

```
if self.com.get():
self.comport=int(self.com.get())-1
    try:
        self.ser = serial.Serial(self.comport)
    except:
        self.statuslist.insert('end','Could not open specified com port')
else:
    self.statuslist.insert('end','Please specify com port to open')
```

This part tries to open the user specified com port, if not successful it gives the user feedback that it either could not open the specified com port (possible because another device is using it) or, if the user haven't specified the com port, that the user please specify the com port to open.

The next part is the initiate process for the communication between microcontroller and PC.

```
self.ser.write('Init\r')
ID = self.ser.readline()
Pattern="AD5933"
match=re.search(Pattern, str(ID))
if match:
    self.statuslist.insert('end', 'Communication with microcontroller established')
```

```
else:
    self.statuslist.insert('end', 'Error while initializing communication wit
    self.statuslist.insert('end', 'please check specified com port')
```

So first Init is written to the microcontroller via the serial port (the $\backslash r$ is carriage return and marks the end of the string). The line is read from the microcontroller (reads a line terminated with $\backslash n$). Then, if the line contains the word AD5933, the user gets a message that the communication with the microcontroller is established. Otherwise, the user gets an error message and is asked to check the specified com port.

The next part is the transmission of the start parameters.

```
self.statuslist.insert('end', '\textsl{Setting startfreq}')
self.ser.write(self.startfreq.get()+'\r')

self.statuslist.insert('end', 'Setting frequency increment')
self.ser.write(self.numbinc.get()+'\r')

self.statuslist.insert('end', 'Setting number of frequency increments')
self.number_increments=int(self.numbsteps.get())+1
self.ser.write(str(self.number_increments)+'\r')

self.statuslist.insert('end', 'Setting excitation voltage')
if self.volt.get()=='2.0Vp-p':
    self.ser.write('V1'+'\r')
elif self.volt.get()=='1.0Vp-p':
    self.ser.write('V2'+'\r')
elif self.volt.get()=='400mVp-p':
    self.ser.write('V3'+'\r')
elif self.volt.get()=='200mVp-p':
    self.ser.write('V4'+'\r')
```

First the start frequency and the frequency increment is transmitted. To get the correct number of increments I found out that I had to add 1 to the number of increments, so this is done before the transmission. Then the code specifying the voltage range is transmitted.

The next part is for receiving the results from the microcontroller and calculating and storing the gain factor and system phase.

```
for i in range(0, int(self.number_increments)):
    self.savedata(i)
```

```
ofile=open('calphase.txt', 'w')
ofile.write('Freq Phase\n')
ofile2=open('gain_factor.txt', 'w')
ofile2.write('Freq Gain factor\n')
for k in range(0, int(self.number_increments)):
    print k
    str1='%d  %g\n' %(self.data['Freq'][k], self.data['Phase'][k])
    gainfac=1/(float(self.calresistor.get()))
    gainfac=gainfac/(float(self.data['Mag'][k]))
    str2='%d  %g\n' %(self.data['Freq'][k], gainfac)
    ofile.write(str1)
    ofile2.write(str2)
ofile.close()
ofile2.close()
```

First the program enters a for-loop that lasts as long as the number of incrementation, each time calling the sampling function savedata with the variable i. Here i is used as a counter, and is used in the savedata function to calculate the current frequency from the start frequency and the frequency incrementation. Next two files are opened for writing, one is for the system phase and the other for the gain factor. The function loops through all the results and calculates system phase and gain factor and write them to file. After the loop is completed the two files are closed.

sample function

This is the function that is called when a measurement sweep is done. It is essentially the same as the calibration function but with some differences with the treatment of the received data.

```
for i in range(0, int(self.number_increments)):
    self.savedata(i, 1)

ofile=open('out.txt', 'w')
ofile.write('Freq Phase Impedance\n')
for k in range(0, len(self.data['Freq'])):
    str1='%d  %g  %d\n' %(self.data['Freq'][k], self.data['Phase2'][k], self.data['
    ofile.write(str1)
ofile.close()
```

The first difference is when the sample function calls the savedata function, the sample function also gives an extra variable 1, with this the savedata function will also calculate the impedance phase by subtracting the system phase from the measured phase. File out.txt is open for writing and the program loops trough all the results and writes them to file. After the loop is finished the file is closed.

plot_imp function

This function plot the measured impedance as a function of frequency.

```
def plot_imp(self):
    pylab.plot([self.data['Freq']],[self.data['Impedance']], 'ro')
    pylab.xlabel('Frequency (Hz)')
    pylab.ylabel('Impedance (Ohm)')
    pylab.savefig('impedance.png')
    pylab.show()
```

This plots the impedance against the frequency and gives labels for the x and y axises. It also saves the plot to impedance.png and shows the plot in a pop up window.

plot_phase function

This function plots the phase as a function of frequency.

```
def plot_phase(self):
    pylab.plot([self.data['Freq']],[self.data['Phase2']], 'ro')
    pylab.xlabel('Frequency (Hz)')
    pylab.ylabel('Phase (degree)')
    pylab.savefig('phase.png')
    pylab.show()
```

This plots the corrected phase against the frequency and gives labels to the x and y axises. It then shows the plot in a pop up window and saves the plot to phase.png.

savedata function

This function receives the real and imaginary part from the microcontroller and calculates the impedance and phase from these.

```
def savedata(self, count, sel=0):
    i=0
    real=self.ser.readline()
    self.data['Real'].append(int(real))

    im=self.ser.readline()
    self.data['Im'].append(int(im))

    freq=int(self.startfreq.get())+int(self.numbinc.get())*count
    self.data['Freq'].append(int(freq))

    magn=math.sqrt(float(real)**2 +float(im)**2)
    self.data['Mag'].append(magn)

    self.data['Impedance'].append(float(1/(magn*float(self.gainfactor.get()))))


    phase2=math.atan((float(im)/float(real)))*57.2957795
    self.data['Phase'].append(phase2)

    if sel!=0:
        while 1:
            if freq>int(self.cal['Freq'][self.k]):
                self.k=self.k+1
            #else if the freq is the same then the phase is calculated
            elif freq==int(self.cal['Freq'][self.k]):
                phase=(phase2)-float(self.cal['Phase'][self.k])
                if phase>30:
                    phase=phase-180
                    break
        self.data['Phase2'].append(phase)
```

This first receives real and imaginary parts and stores them. Then the current frequency is calculated using the start frequency plus the frequency increment multiplied with the count. To calculate the final impedance it is first necessary to calculate the magnitude from the real and imaginary parts. After the magnitude is calculated the impedance is calculated using the magnitude and the user provided gain factor. From the real and imaginary part the phase is then calculated and transformed to degree by multiplying the value with 57.2957. If it was the calibration function that called the savedata function this phase will be the system phase, if it was called by the sample function this phase is used to calculate the phase by subtracting the system phase. If the variable sel is not equal to zero (this only happens when the sample function has called the function) the program enters an eternal while-loop. To substract the correct system phase the program has to find the system phase for the same frequency. This is done by a simple incrementation, if the current frequency is higher than the calibration frequency a counter is incremented so that the current frequency will be compared to a higher frequency the next time. If the current frequency is equal to the calibration frequency the phase is calculated. Because some measurements may span over two different quadrants I need to subtract 180 degrees from the positive phases to get all the results represented in the 4th quadrant. After that the the final phase is stored.

Chapter 5

Full specification and operational guidelines

5.1 Final system specification

The final specification the the AD5933 based bioimpedance measurement system is listed in table 5.1.

Parameter	Value
Power supply V_{DD}	2.7-5.5V
Impedance range	0.1kΩ-10MΩ
Excitation frequency	3kHz-100kHz[1]
Excitation frequency resolution	0.1Hz
Excitation voltage (p-p), selectable from graphical interface	2V, 1V, 400mV, 200mV
Reference frequency oscillator	16.776MHz[2]
Specified system accuracy	0.5%[3]
Temperature range for operation	-40^o-$+125^{o}$[4]
Measurement and calibration ranges (selectable by choice of R_{FB})[5]:	
Range 1	0.1-1kΩ
Range 2	1-10kΩ
Range 3	10-100kΩ
Range 4	100-1000kΩ
Range 5	1-2MΩ
Range 6	9-10MΩ
Receive stage:	
Gain	1x or 5x[6]
ADC resolution	12 bits
ADC sampling rate	1 MSPS
System requirements of operation: a) operation:	
V_{DD}=3.3V	I_{DD}=10-15mA
V_{DD}=5.5V	I_{DD}=17-25mA
b) standby:	
V_{DD}=3.3V	I_{DD}=11mA
V_{DD}=5.5V	I_{DD}=16ma
c) power down:	I_{DD}=1-8μ

Table 5.1: Final specification for the AD5933 based measurement system

[1]Range for use of 16.776MHz reference oscillator.

[2]Lower frequency oscillators will give lower excitation frequency.

[3]The specified accuracy depends strongly on calibration. Accurate measurements require calibration close to the value of the device tested. Which impedance range that is used also plays an important role.

[4]For operation over this range temperature compensation is required.

[5]Described in operational procedures.

[6]Could have implemented selection of gain from graphical interface, but gain of 5 will often cause saturation so I sat the gain factor to 1x in the microcontroller code

5.2 Operational guidelines

With the developed software package the measurement system is fairly easy to learn and use, especially because the developed graphical interface. This section provides guidelines for setup of the measurement system and use of the graphical interface.

The connection is really simple. First gnd is connected to the pins DGND, AGND1 and AGND2. Then the supply voltage (2.7V-5.5V, a 7805 voltage regulator which outputs 5V was used) is connected to the pins DVDD, AVDD1 and AVDD2. The SCL pin is connected to pin 0 on PORTD on the STK500 (with an ATmega16 microcontroller fitted) and the SDA pin is connected to pin 1 on PORTD on the STK500. The I^2C bus lines also need pull up resistors so a 10kΩ resistor must be connected between SCL and the 5V supply voltage. The same goes for the SDA line. The STK500 get supply voltage from a separate wall adapter. Therefore necessary to connect ground from the STK500 to ground on the AD5933's breadboard, so that they get the same reference. It is also important to remember to use the spare RS232 port on the STK500 board to connect to PC.

Figure 5.1 shows a simple schematic of the system (be aware that the pull up resistors are not included in this figure). Figure 4.1 shows a picture of the running graphical interface.

So when you start the graphical interface I have already set some standard start parameters (these can be edited easily in the code or entered manually in the interface each time). After deciding on the start parameters to use calibration is performed. A value of the calibration resistor must be entered in order to calculate the gain factor. Then a measurement sweep can be started. After the sweep, the program will calculate the system phase and store it to file calphase.txt and calculate gain factor and store to file gain_factor.txt. Then a restart of the program is required so that the new system phase can be reloaded. At that time the user must select the gain factor from the gain_factor.txt file and either edit it into the code or enter it in the gain factor entry field manually each time. When the program has started again one can click start sweep and the start parameters will first be transmitted to the microcontroller and then to the AD5933. After the sweep is complete one can click on plot impedance and plot phase to plot the impedance and phase. Data will also be stored in a file called out.txt that will contain the frequency, impedance and phase. This file can for example be imported in to Excel.

Figure 5.2 shows a picture of the system.

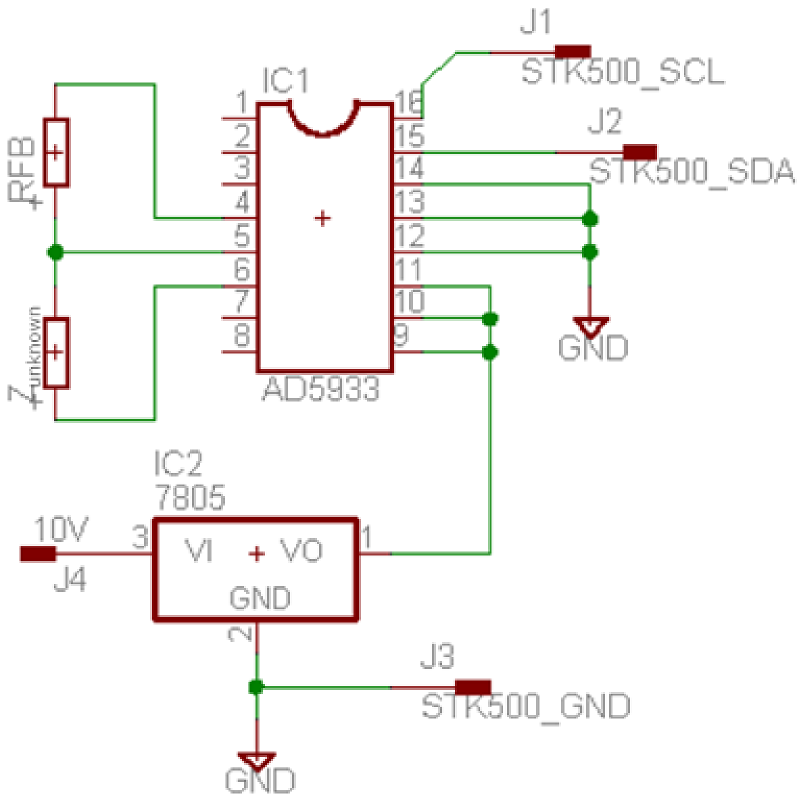


Figure 5.1: A simple schematic for the system (without the pull up resistor on SCL and SDA)

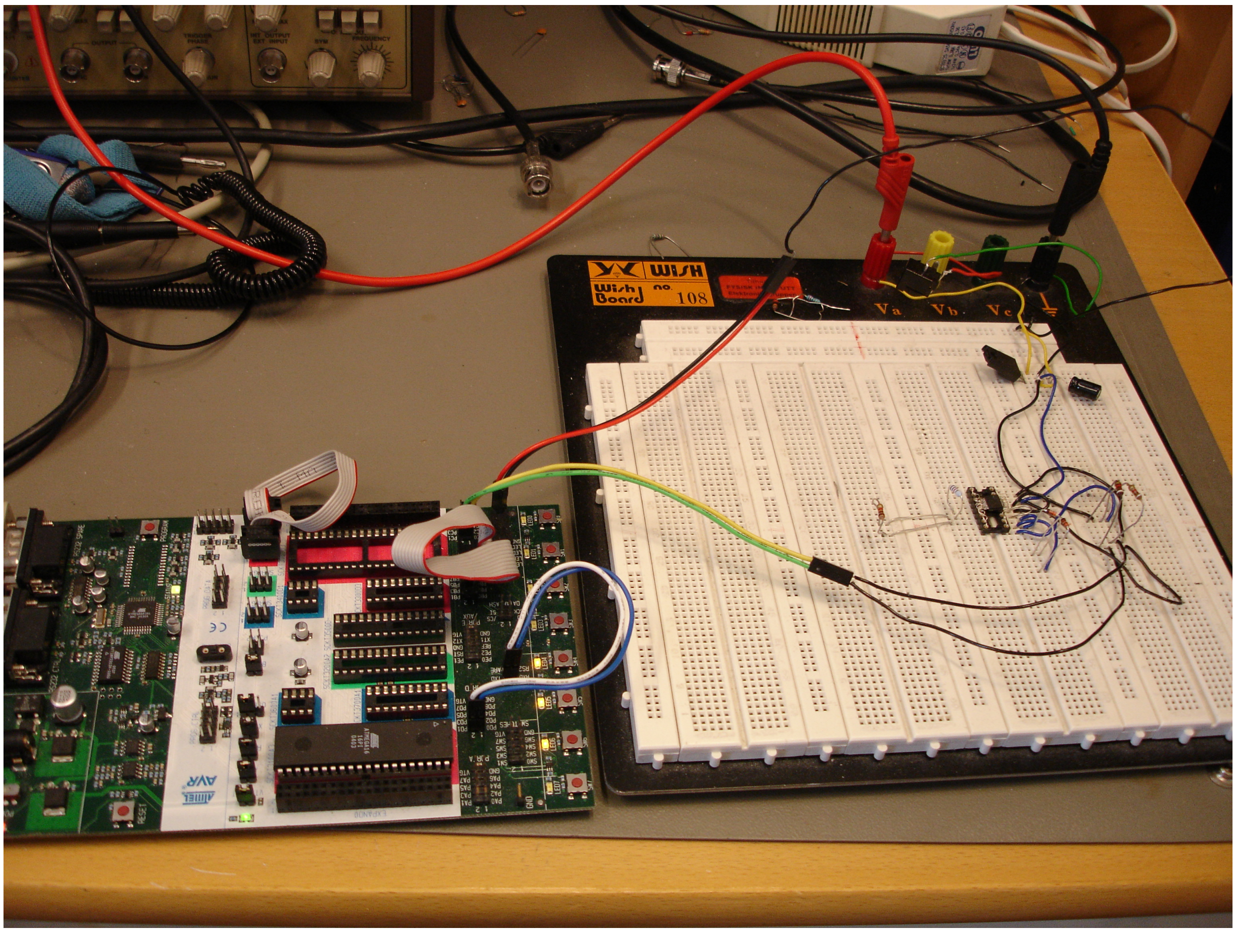


Figure 5.2: Picture of the system with the STK500 and the breadboard with the AD5933

Chapter 6

Measurement and evaluation

6.1 Calibration of the system

The calibration of the AD5933 based measurement system is done in two steps. First you need to calculate the gain factor used to calculate the impedance from the real and imaginary data parts received from AD5933. The gain factor is calculated using the equation in eq. 6.1.

$$Gain\ Factor = \frac{1}{\frac{R_{cal}}{Mag}} \tag{6.1}$$

Where R_{cal} is the resistor with a known value that is used for the calibration of the system, R_{cal} should also have the same value as R_{FB} during the calibration run. Mag is the magnitude calculated from the real and imaginary values received from the AD5933. Mag is calculated by:

$$Mag = \sqrt{Real^2 + Im^2} \tag{6.2}$$

In the AD5933 datasheet the eq. 6.3 is given to calculate the phase.

$$Phase = tan^{-1}\left(\frac{Im}{Real}\right) \tag{6.3}$$

There is very little information about the phase calculations in the AD5933 datasheet, it only says that phase can easily be calculated using eq. 6.3 (this was updated in a new version from May 2008). Only in the data sheet for the evaluation board there is some mention of that the phase measured by the AD5933 takes into account the phase introduced through the entire signal path (see page 12 of AD5933 evaluation board datasheet). To get the phase introduced only by the components between V_{in} and V_{out} all the

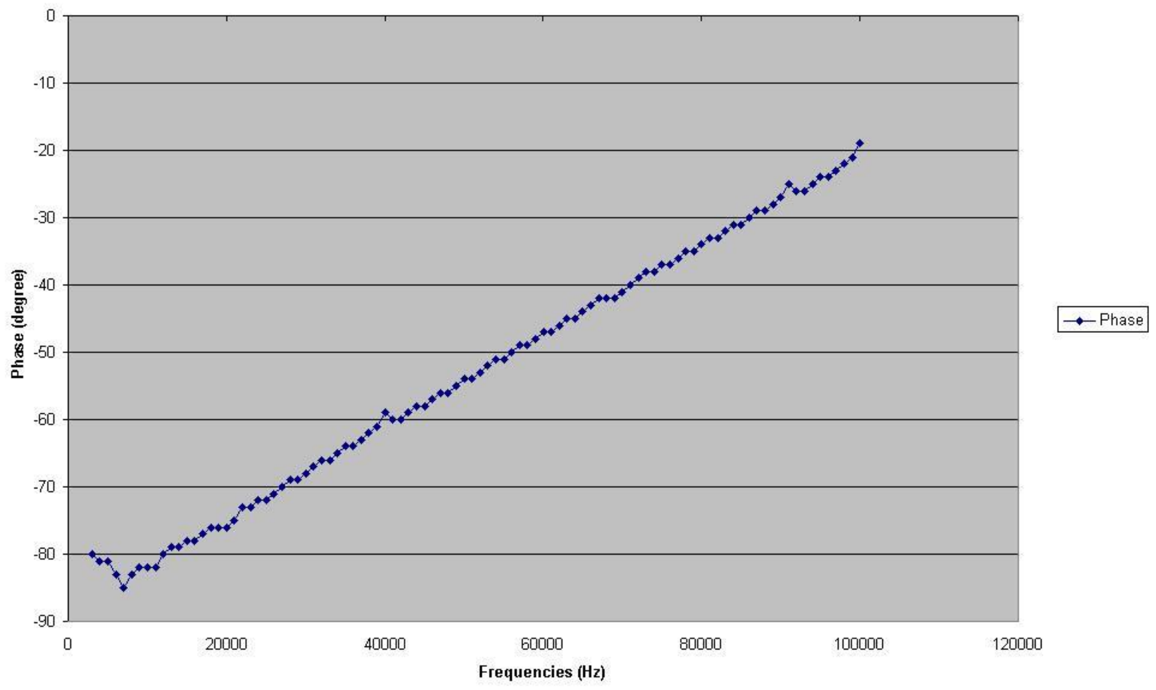


Figure 6.1: The system phase stored from calibration on a resistor with value 1.1kΩ in range 2

phase data for each frequency under a calibration with a resistor have to be saved. Then doing a measurement the phase from the calibration has to be subtracted from the phase that is being measured for each frequency. The phase is then given by calculating:

$$\theta_{impedance} = (\theta_{unknown} - \theta_{system}) \tag{6.4}$$

where θ_{system} is the phase of the system with a calibration resistor between V_{in} and V_{out}. $\theta_{unknown}$ is the total phase of the system with an unknown impedance between V_{in} and V_{out}. $\theta_{impedance}$ is the phase due to this impedance. The system phase received during calibration is stored on the PC as a file named calphase.txt, every time the graphical interface starts it reads the whole file and uses that value to calculate $\theta_{impedance}$. Figure 6.1 shows the system phase as a function of frequency for measurements in range 2 (1kΩ-10kΩ) for a resistor with value 1100Ω. When a measurement is made the stored calibration phase (system phase) is subtracted from the measured phase for corresponding frequency points. Figure 6.2 shows the resulting phase after measurement of a resistor of the same value as the calibration resistor. Ideally this phase should have been zero. There are

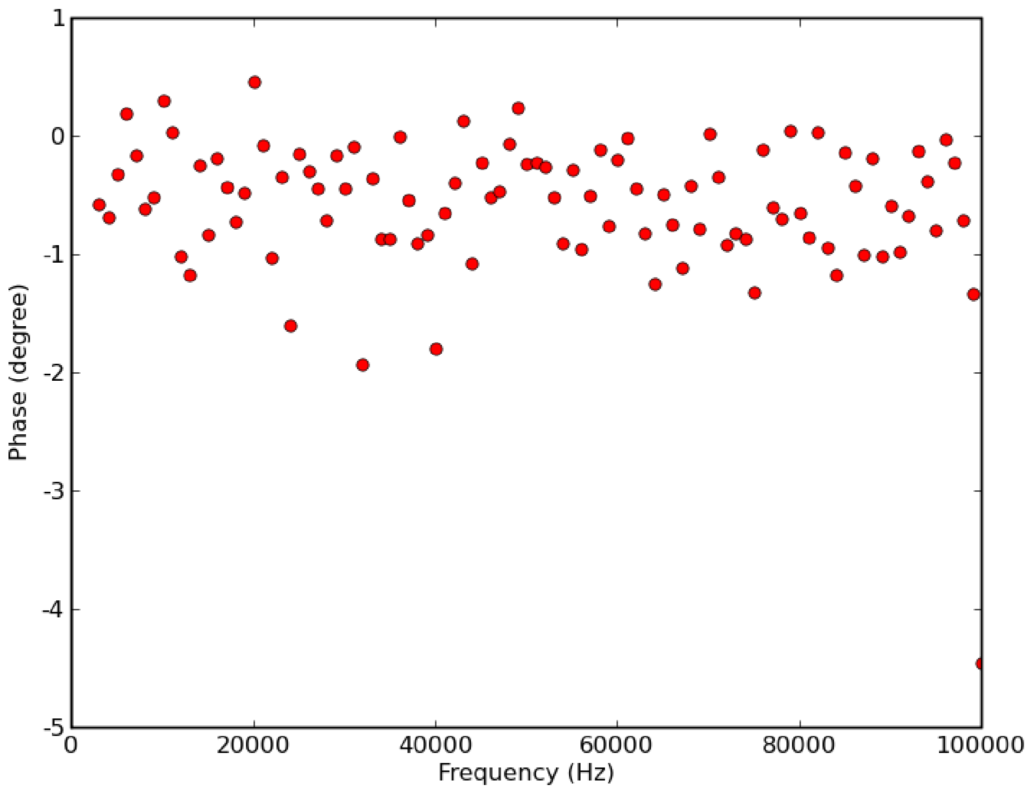


Figure 6.2: Calculated phase for a resistor value of 1.1kΩ using eq. 6.4

however some minor randomly distributed contributions, which can be explained taking into account that the stability of the AD5933's internal reference oscillator operated a 16.776MHz is not perfect. Use of an ultra stable external oscillator could have improved this. The phase plotted in Figure 6.2 is the difference between two phase values, as shown in eq. 6.4. They are both influenced slightly by the instability of the oscillator. We may then have stronger fluctuations than shown in Figure 6.1 that represent one measurement of the system phase.

The calibration has to be rerun if the user changes either of

- Impedance range (changing the R_{FB}).
- The excitation signal voltage.
- The frequency range or incrementation frequency, this is necessary since we need have to have the system phase for all the frequencies

in the frequency range used for measurements.

A routine has been implemented in the GUI allowing the user to obtain the gain factor and the system phase for all the frequencies by only supplying the value of the calibration resistor. So for a run one would first use the automated calibration for the desired frequency range and then do the measurement.

An important thing to be aware of is that the gain factor though not very frequency independent, changes somewhat with frequency so measuring on a pure resistor without any reactance one will see because of this that the impedance seems to increase with the frequency since this is not adjusted hence to the increased gain factor. See fig 6.3 for a plot that shows the gain factor as a function of frequency. This measurement is done with a resistor with value 1.1kΩ. It is seen that the gain factor increases

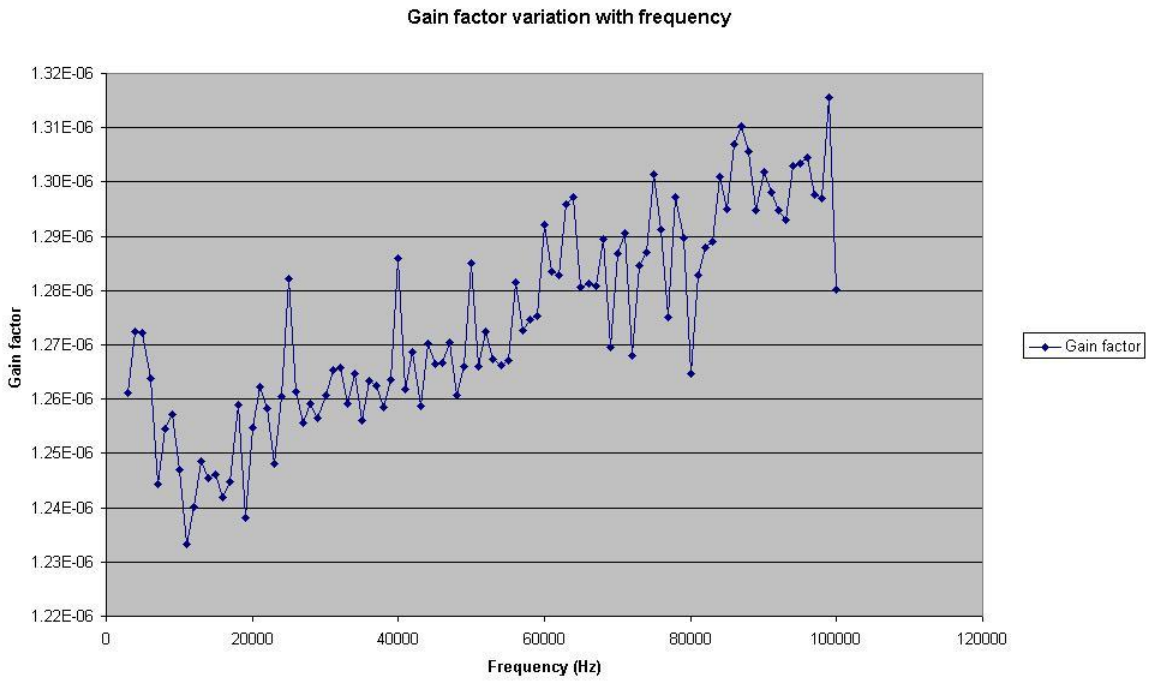


Figure 6.3: Gain factor variation with frequency

about 6,8% over the frequency range. However for most measurements and for narrow frequency bands a 1 point calibration may be sufficient. The gain factor variation is close to a straight line. Thus a two point calibration as recommended in the data sheet, could improve the system.

Range no.	Value(kΩ)	R_{FB}(kΩ)	Cal. resistor (kΩ)
1	0.1-1	0.1	0.1
2	1-10	1	1
3	10-100	10	10
4	100-1000	100	100
5	1000-2000	1000	1000
6	9000-10000	9000	9000

Table 6.1: Measurement ranges for AD5933

Another important issue to be aware of is the gain factor variation with temperature, the typical error variation is 30ppm/^{o}. Figure 6.4 (this is from [2]) shows the variation in the impedance measured due to temperature changes.

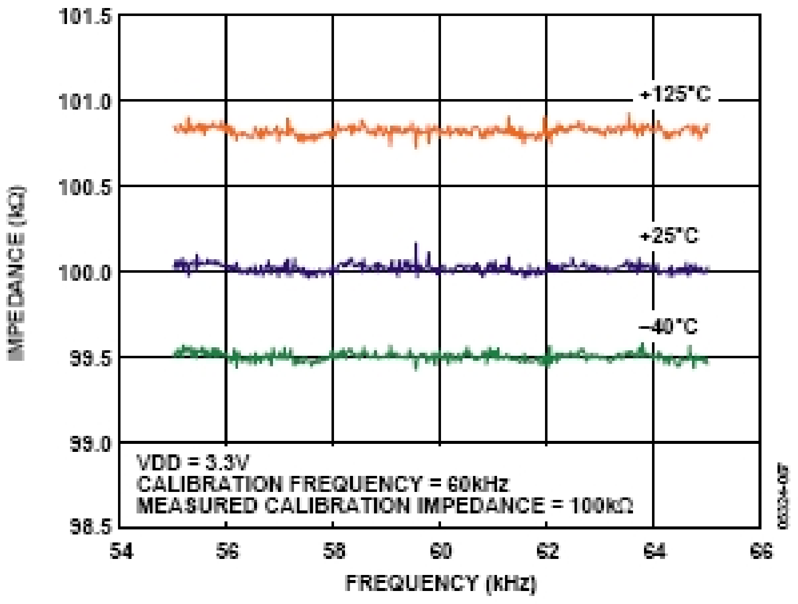


Figure 6.4: Gain factor variation due to temperature changes

The measurement range of the system as specified is 0.1KΩ to 10MΩ divided into six ranges as specified in table 6.1 which also gives the calibration data to be used. It is noted that the ranges 5 and 6 are a bit different from the other ranges. Range 5 goes from 1MΩ to 2MΩ, and range 6 goes from 9MΩ to 10MΩ. There is no range covering 2MΩ to 9MΩ.

Calibration measurements have been performed for all ranges using available and equal valued resistors for the gain setting resistor R$_{FB}$ and for the calibration resistor at the test position.

6.2 Special problems to observe

There are two phenomena that may cause problems during measurement.

- Saturation phenomena giving too high input (outside the linear range) to the ADC.
- Misinterpretation of the phase (θ) information.

It is important that these phenomena, which are not well covered in the AD5933 datasheet are understood and taken into account during the measurements.

6.2.1 Saturation phenomena that may occur during calibration and measurement

The critical issue is the dynamic range of the ADC. This is controlled by the gain through the system as given by equation 6.5.

$$output\ excitation\ voltage * \frac{Gain\ setting\ resistor}{Z_{unknown}} * Gain_{PGA} \tag{6.5}$$

The gain is again controlled by

- The selected voltage for the output excitation.
- The current to voltage gain setting resistor (R_{FB}) in combination with the calibration resistor or during measurement $Z_{unknown}$.
- The PGA gain.

The data sheet uses calibration resistors at the lower edge of each resistor range, meaning that the ratio

$$\frac{Gain\ setting\ resistor}{Z_{unknown}\ (Z_{cal})} = 1 \tag{6.6}$$

With

$$Z_{cal} = R_{FB} \tag{6.7}$$

This means that using the data sheet calibration the device measured will always have a higher impedance value than the calibration resistor, and the gain setting resistor. The gain through the system will be lower and saturation of the ADC will not take place. The data sheet also gives a graphical illustration of how the impedance error typically varies with frequency for measurements on resistors higher than the value of the calibration resistor. Figure 6.5 shows this variation for ranges 1, 2, 3 and 6. For the upper ranges (4-6) the impedance error shows a strong increase

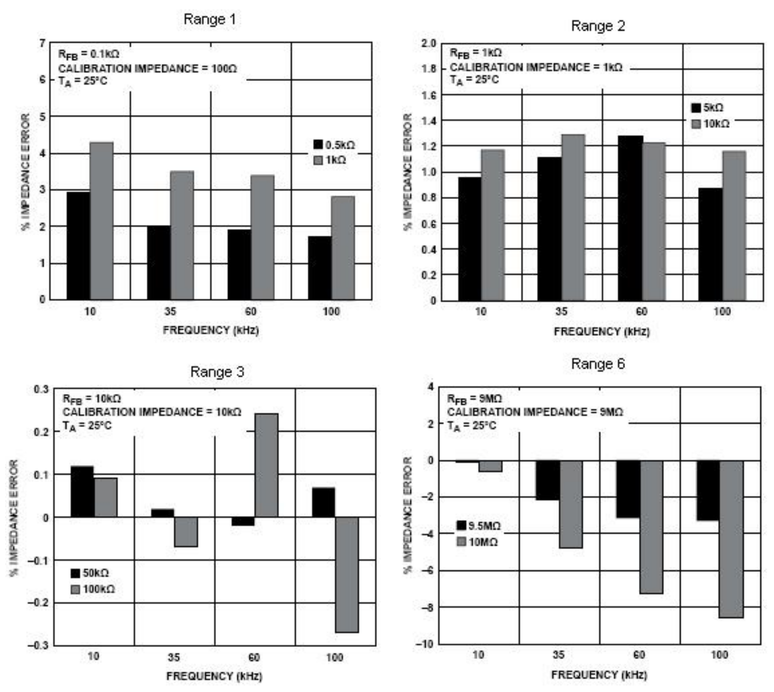


Figure 6.5: The impedance error as function of frequency for ranges 1, 2, 3 and 6 [2]

with frequency, up to -7 and -8% for ranges 5 and 6. The error seems to increase the further away from the calibration value the measurements are done as illustrated by the curves for range 5 and 6 where also the relative deviations from the calibration resistor value are small, meaning that calibration is critical.

For range 1 the error is up to 3% all over the frequency range. Range 2 and 3 seems to give the best results. Range 2 has an error between 1-1.2% while range 3 has an error between about 0.24 to -0.27%. This illustrates that it is critical to have a calibration value close to the value of the resistor to be measured.

For measurements on complex impedances there may be strong variations over the frequency span in both the imaginary part and the magnitude M. This may result in complications, and a frequency run may cover more than one range. Separate calibrations for each range may then be needed.

In the data sheet for the evaluation board for AD5933 [3], Analog Devices state that the gain factor should be calibrated when the largest response signal is present on the ADC, and the signal kept within linear range of interest. This corresponds well with the conditions set in equations 6.6 and 6.7.

It is also stated "The user should choose a calibration impedance which is mid value between the limits of the unknown impedance". Thus the user must know the value of the impedance to be measured relatively well before making a final calibration.

In order to make accurate measurements a two step calibration is needed. The first will give an approximate impedance value and the second calibration with an calibration resistor close to the first measured value. Then the final measurement will give the most accurate value.

For measurements in the low impedance range (0.1-1kΩ) there is another problem. At approximately 500Ω the current drawn may be to high for the transmit side amplifier. There may also be corresponding problems sinking at the receive stage.

The variable resistor R$_{out}$ at the transmit stage will also contribute in the gain setting expression in addition to Z$_{unknown}$ giving the modified expression for the gain through the system

$$output\ excitation * \frac{Gain\ Setting\ Resistor}{Z_{unknown} + R_{out}} * Gain_{PGA} \tag{6.8}$$

R_{out} depends on the output excitation voltage as shown in table 6.2. Taking

V_{out}	R_{out}
2Vp-p	200Ω
1Vp-p	2400Ω
400mVp-p	1000Ω
200mVp-p	600Ω

Table 6.2: R_{out} varying with output excitation voltage

these two restrictions into account it should be possible to calibrate and make measurements also in range 1.

Another possible solution would perhaps be to make body measurements with a fixed resistor of known value in series, so that the measurement will be in range 2 instead of range 1.This was tested and did not work well.

Figure 6.6 and 6.7 show the impedance and phase of a measurement of a R-C parallel combination (R=2.2kΩ, C=3.2nF) and with $R_{FB} = 2.2k\Omega$ where the ADC goes into saturation and one can see some very strange effects on the impedance and phase because of this. On the plots one can see that impedance stops at 1kΩ, it doesn't go any lower, at this point the ADC is saturated and can't measure impedances. To start with the phase is going down, but when the ADC is saturated it for some reason begin to climb upwards. These two plots illustrate the effects saturation have on the measurement results.

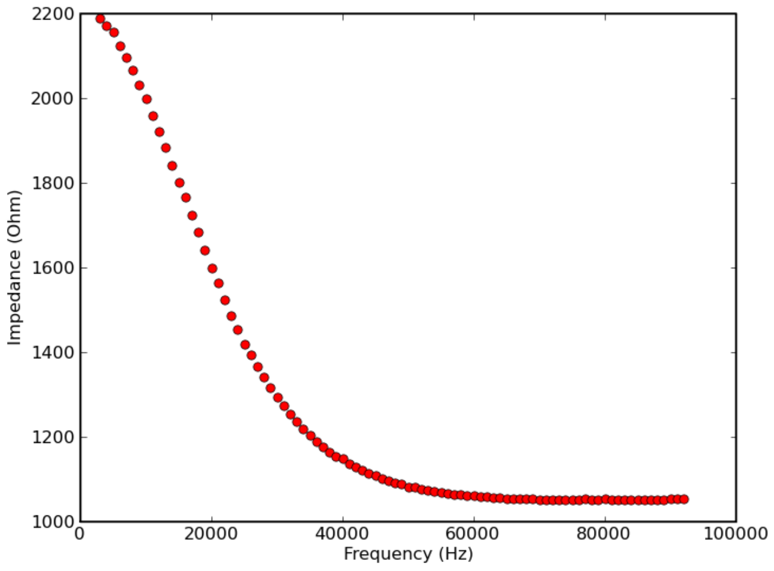


Figure 6.6: Typical saturation effect on the impedance value from parallell RC-networks (R=2.2kΩ, C=3.2nF)

6.2.2 The phase-shift calibrated

The complex output values from the AD5933 is stored in separate register addresses for the real and imaginary parts after each sweep. They are the real and imaginary components of the DFT and not of the impedance

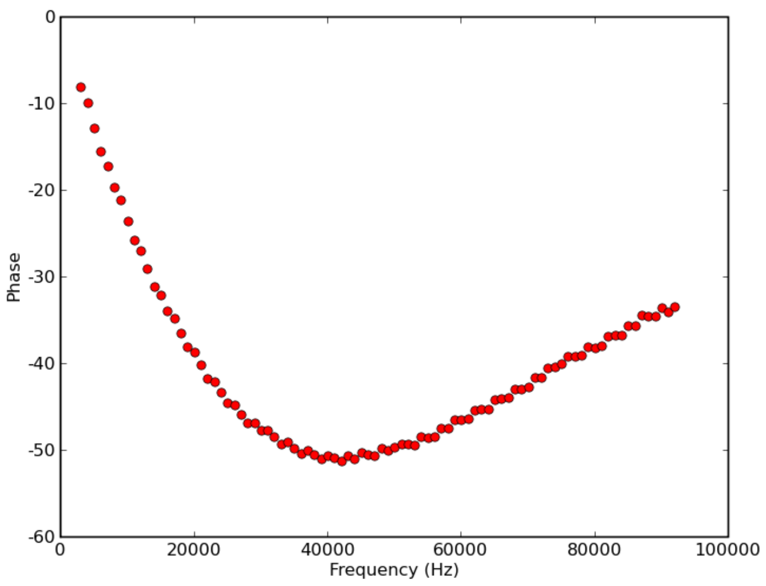


Figure 6.7: Saturation effect on the phase with the same measurement configuration as in Figure 6.6

being measured. The magnitude of the impedance $|Z|$ is calculated by the magnitude of the DFT components.

$$M_{DFT} = \sqrt{R^2 + I^2} \tag{6.9}$$

and the impedance is found as

$$M_Z = \frac{1}{Gain\ Factor * M_{DFT}} \tag{6.10}$$

using the calibration procedure described in section 6.1 The measured phase is given by 6.11.

$$\theta_t = tan^{-1}\left(I/R\right)) = \theta_s + \theta \tag{6.11}$$

This is the total phase from the AD5933 internal system components in the signal path and the externally connected impedance. The impedance components are given by

$$R = Z_{real} = |Z|\cos\left(\theta\right) \tag{6.12}$$

$$X = Z_{imaginary} = |Z|\sin\left(\theta\right) \tag{6.13}$$

where

$$Z = R + jX \tag{6.14}$$

The phase is converted to degrees by

$$\theta = \tan^{-1}(I/R)\,\frac{180^o}{\pi} \tag{6.15}$$

The problem arises in connection with equation 6.11 which returns the correct answer only for one quadrant at a time, so if one want measurement represented in the 4th quadrant, one will need to correct the values, that will be represented in the 1th quadrant. Analog Devices operates with the standard angle taken counter-clock wise from the positive real x-axis, so that all the phases will be represented from 0 degree to 380 degree. For bioimpedance it is normal to use negative phase, and this is the reason for chosing somewhat different approach than Analog Devices. Representing the result measured in 1th quadrant requires subtraction of 180^o from the phase to get the phase represented in 4th quadrant as a negative value. This is also discussed in the new data sheet available from may 2008.

With this problem there will normally be a step in the phase value as illustrated in Figure 6.8 showing the phase for a measurement on a R-C series combination (R=12kΩ, C=3.2nF) for a measurement in range 2. Figure 6.9 shows the corrected phase for the same measurement.

6.3 Verification tests on resistors and capacitors

Test os resistor and capacitors.

6.3.1 Overview

For testing purposes the impedance Z and the phase θ of resistor and capacitors of known value were measured.

Resistors of different value were used for calibration and testing. The AD5933 operates as shown in the system specification in Chapter5.1 in six ranges from 0.1kΩ-10MΩ in total.

6.3.2 Resistors

A resistor of 580Ω was measured in range 1 with the same calibration as for the whole body measurements (R_{FB}=R_{cal}=680Ω). The variations in $|Z|$

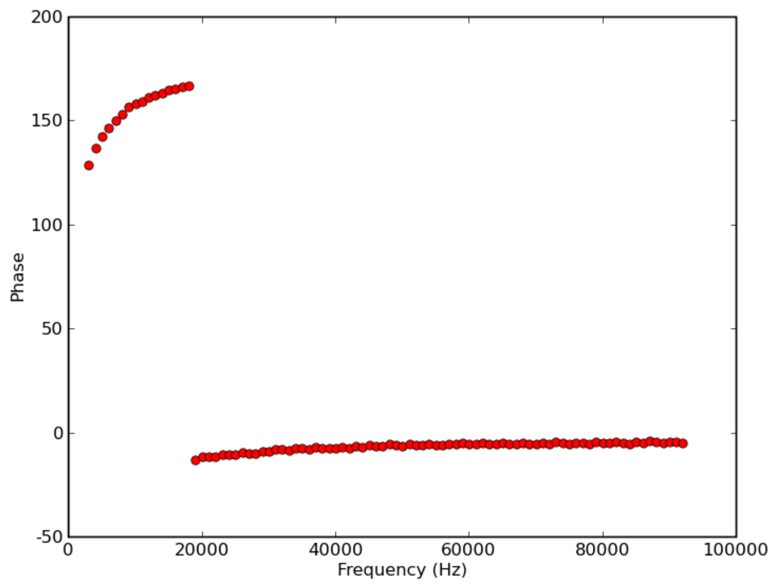


Figure 6.8: Variation of the phase angle with frequency for a RC-series combination measured in impedance range 3 with uncorrected positive phase value

and θ are plotted in Figure 6.10, a) and b) respectively. It is observed that the $|Z|$ value is somewhat high and is increasing with frequency. The phase angle shows a variation of $\pm 0.5^o$ which partly is caused by the lack of stability of the internal reference oscillator.

Resistor tests were also performed going outside the range of calibration. For range 2 (1kΩ-10kΩ) resistors of 1kΩ, 50kΩ, 1MΩ and 10MΩ were used. Figures 6.11 and 6.12 are for measurements performed in range 2 (1k-10kΩ) with calibration using R$_{FB}$=R$_{cal}$=1kΩ.

Figure 6.11 shows resistance and phase values for a 1kΩ resistor. The impedance value measured is correct at the calibration frequency of 50kHz, but there is a variation of a few percent toward the end of the measurement range. This indicated that the calibration procedures covering the full measurement range should be implemented (2 point calibration procedure).

The phase angle shows again a variation of $\pm 0,4^o$ partly attributed to the reference oscillator.

Figure 6.12 shows the corresponding results for a 50kΩ resistor, which is far above the values of range 2. Both measured resistor value and phase

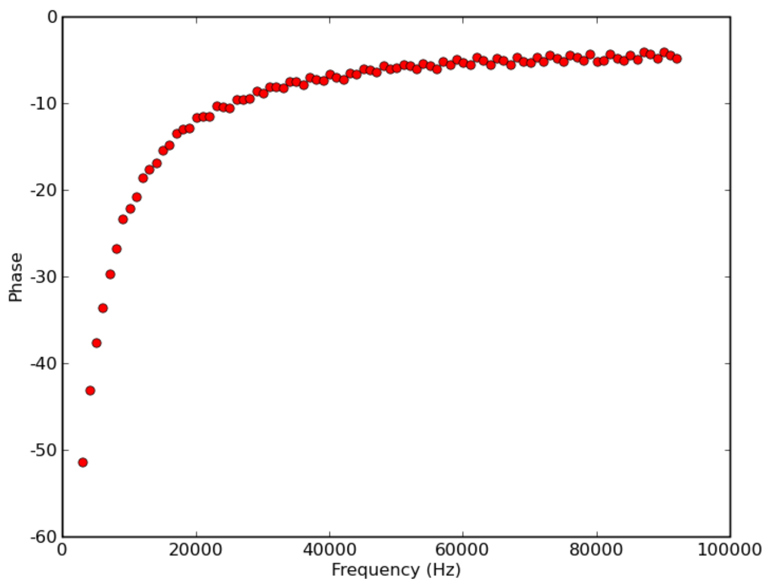


Figure 6.9: Variation of the phase angle with frequency for a RC-series combination measured in impedance range 3 with corrected positive phase value

angle are totally misleading. This illustrates the problem encountered when the ratio R/R$_{FB}$ is very different from 1.

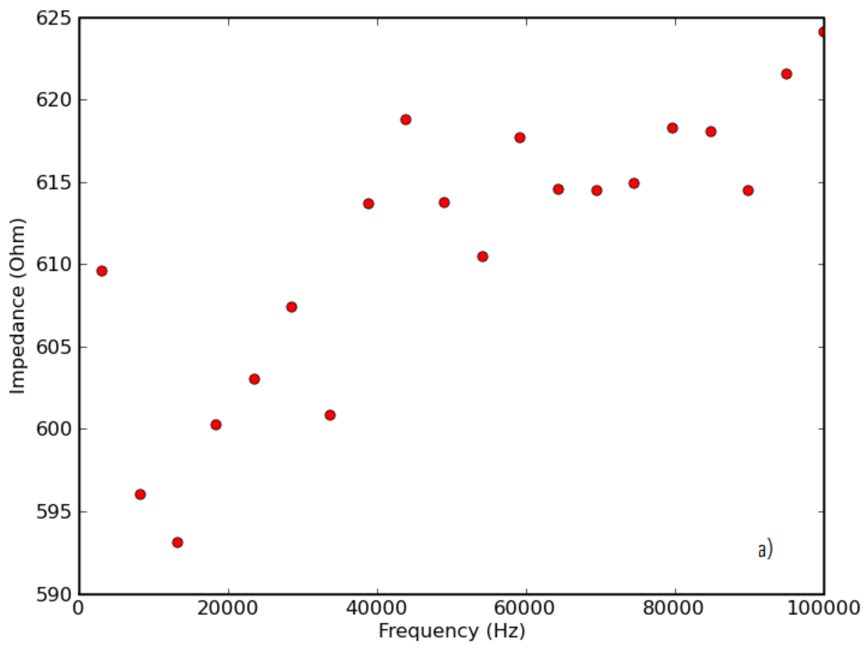
The observed values for 1MΩ and 10MΩ resistors were even worse. Thus, all measurements must be performed in the range of calibration, and the closest possible to the calibration value.

6.3.3 RC-networks

Network with capacitors turned out to be more complicated to test due to the strong frequency dependent variation in both phase and magnitude. For a RC series combination we have:

$$|Z|_s = \frac{\sqrt{1+(\omega RC)^2}}{\omega C} \tag{6.16}$$

$$\theta_s = -\tan^{-1}\left(\frac{1}{\omega RC}\right) \tag{6.17}$$



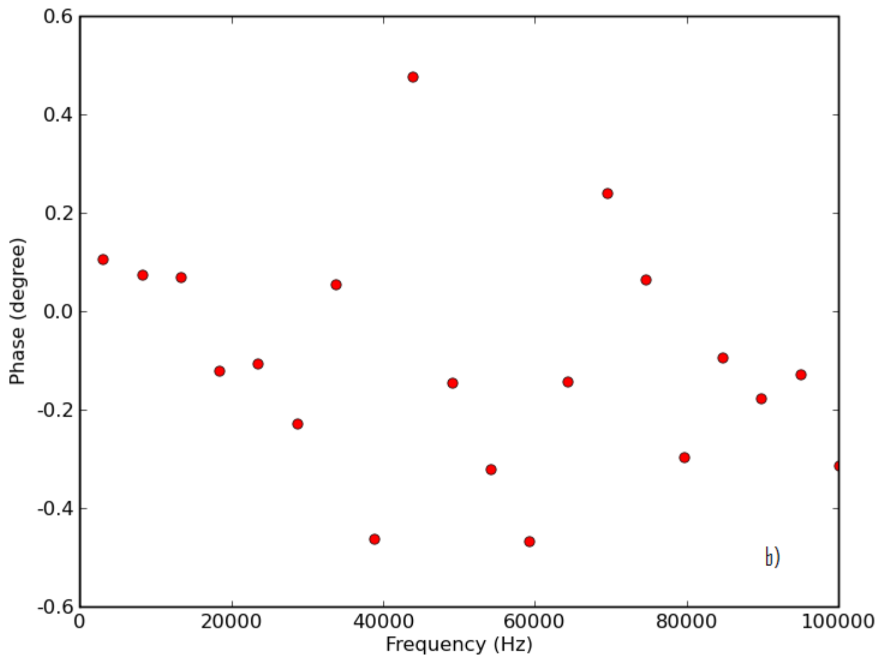
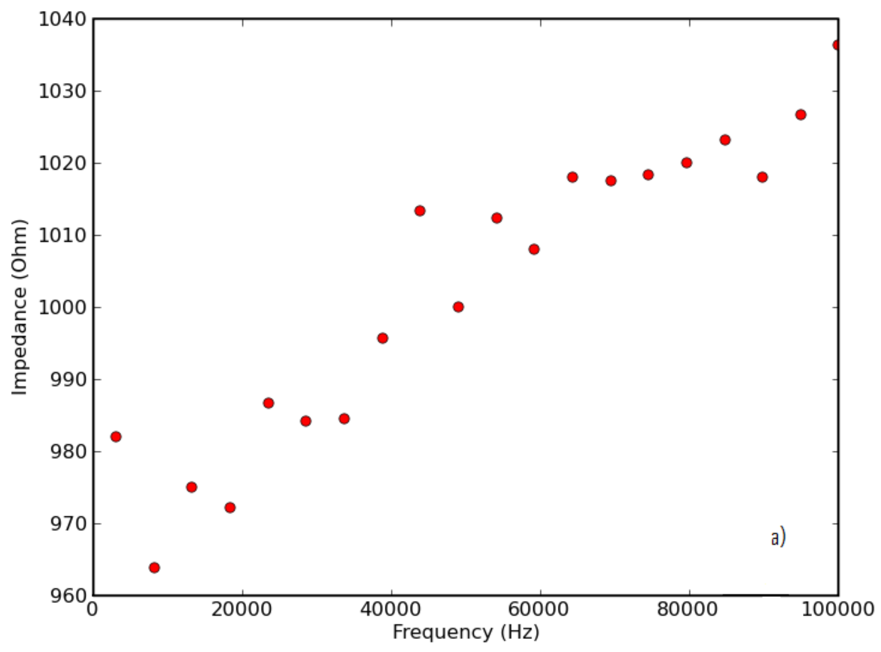


Figure 6.10: Resistance (a) and phase angle (b) as functions of frequency for a 580Ω resistor in range 1



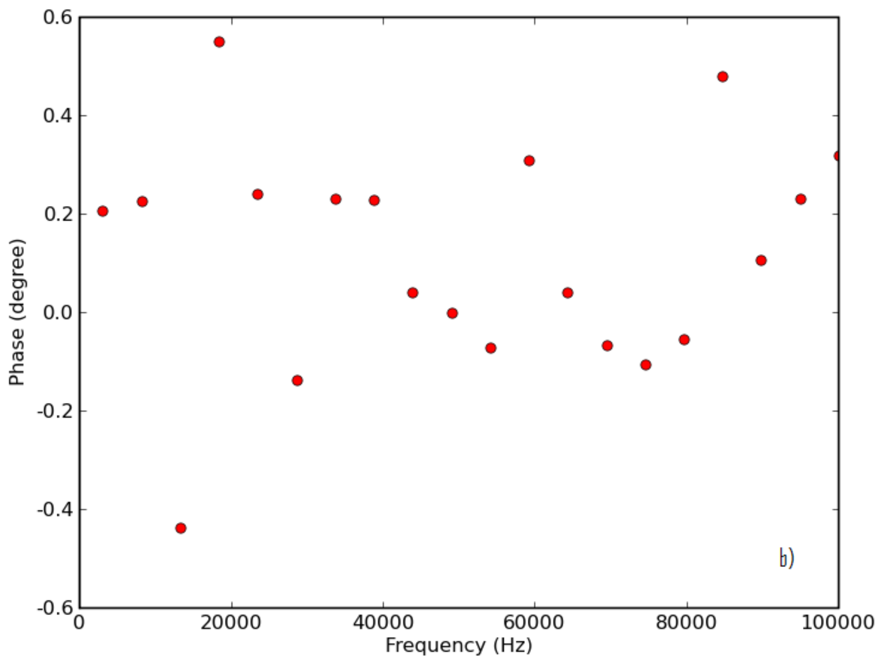
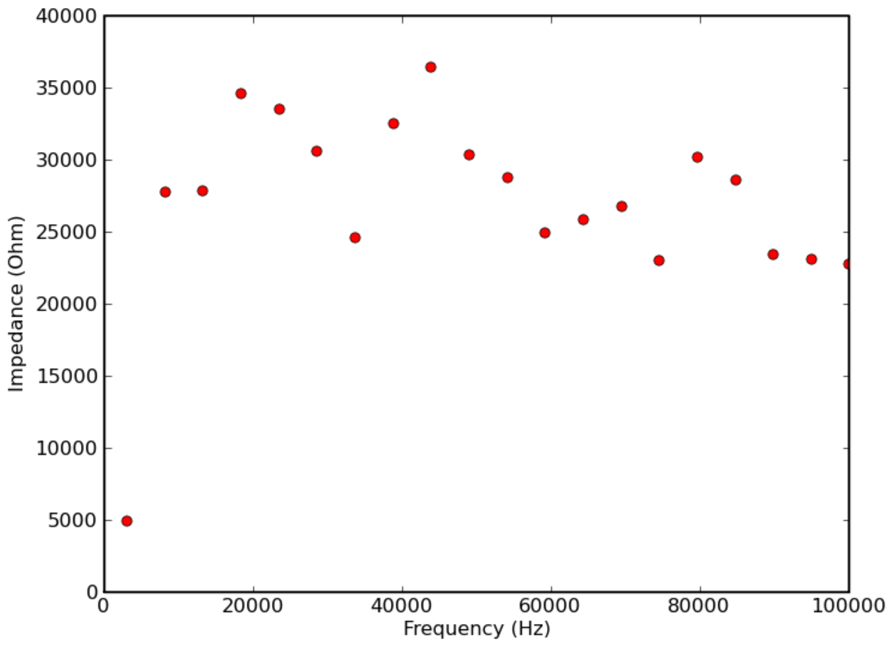


Figure 6.11: Resistance (a) and phase angle (b) for a 1kΩ resistor measured in range 2 with a 1kΩ calibration



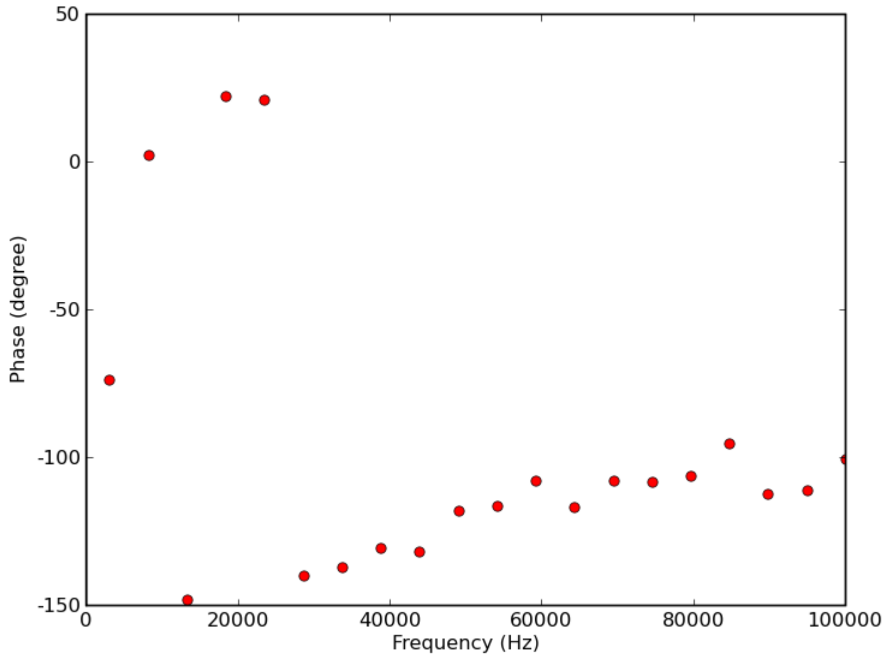


Figure 6.12: Resistance (a) and phase angle (b) for a 50kΩ resistor measured in range 2 with a 1kΩ calibration

For a parallell combination we have:

$$|Z|_p = \frac{R}{\sqrt{1 + (\omega RC)^2}} \tag{6.18}$$

$$\theta_p = -\tan^{-1}(\omega RC) \tag{6.19}$$

This shows that there will be strong variations in $|Z|$ due to variations in frequency, in particular for $|Z|_s$ at low frequencies. However it should be well possible to make measurements in the higher frequency range calibrating at the resistor value.

For a parallell combination $|Z|_p$ decreases with frequency. A calibration with R is risky because if the resulting resistance become too low the ADC will be saturated.

It should however be possible to use the value at the upper frequency for calibration depending on the value of the RC components, but the parallell circuit has been difficult to handle.

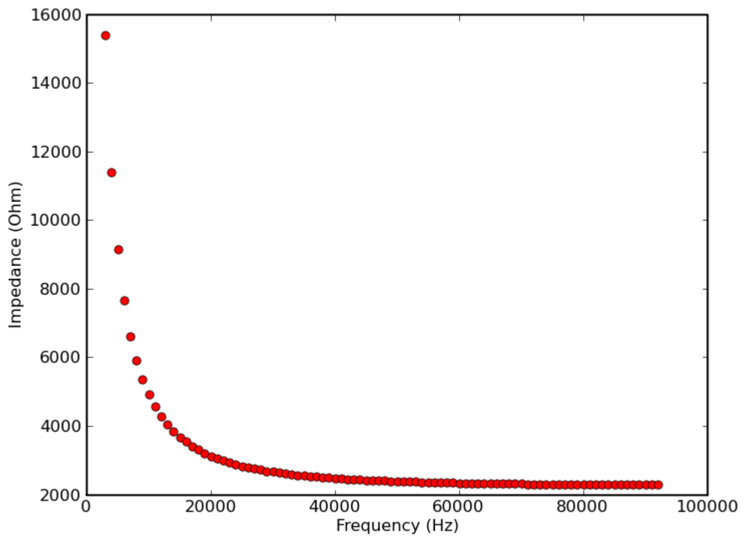
Figure 6.13 shows magnitude and phase for a 2.2kΩ resistor in series with a 3.2nF capacitor, calibration was performed using a 2.2kΩ resistor.

Fortunately, doing whole body measurements the phase values are only minus a few degrees, thus problem of this type are not bad, and suitable calibration values covering the frequency range can be found.

6.4 Verification testing on living tissue

For measurements on the body the contacts must be good. It was decided make whole body measurements between right hand and right foot. To best possible eliminate the impedance from the skin it was decided to use big electrodes consisting of two buckets containing a solution of salt dissolved in water. A concentration of 32g salt per liter water was used. Each basket had a hole drilled in the side of the basket, and the connection to the salt-water solution was established by gluing ECG electrodes over the holes in the side of the two baskets. The electrodes were then connected to the breadboard and to the AD5933 with cables with banana contacts at each end. The measurement set-up is shown in the picture in figure 6.14 (note that picture is only a illustration, and in the picture the left foot and hand is used but in my measurements

For the first trial measurement the instrument was calibrated for range 2, when the measurement were done we saw the measured results was in



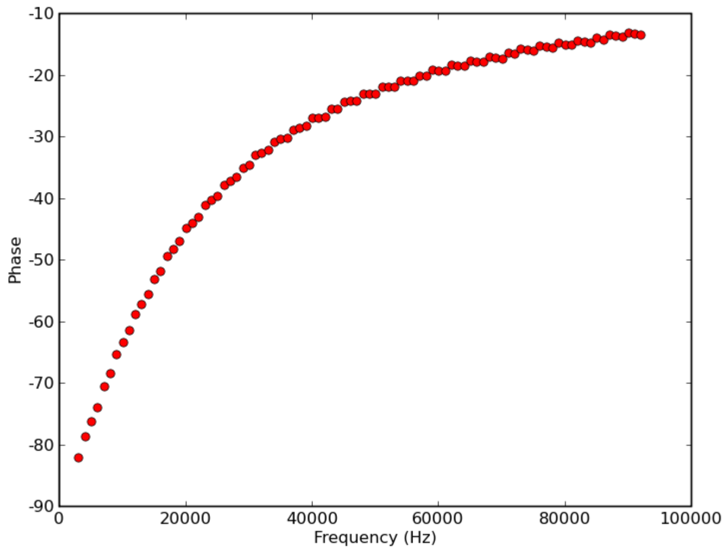


Figure 6.13: Magnitude and phase angle for a series resistor capacitance circuit, R=2.2kΩ, C=3.2nF

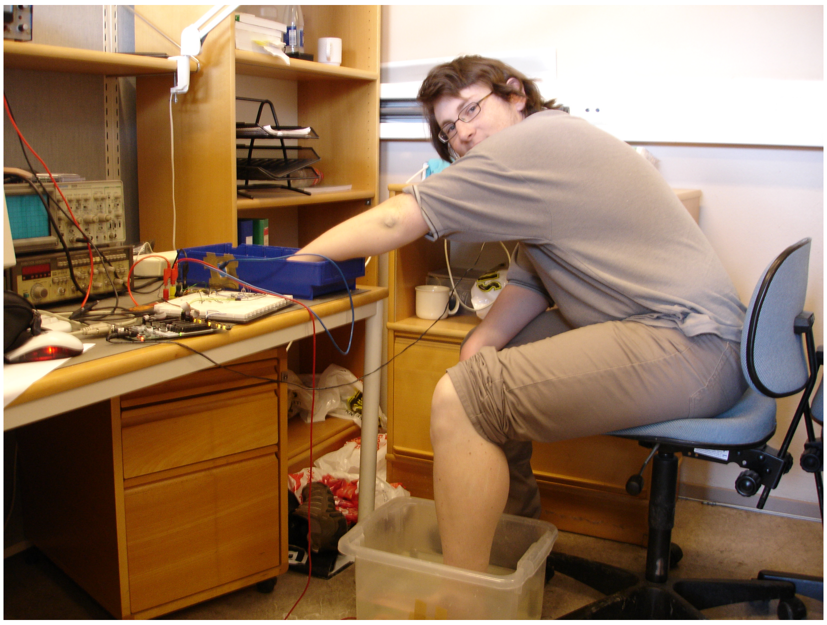


Figure 6.14: Illustration of the measurement setup used under measurements

range 1 and not 2, there there was a need for recalibration for range 1 to be sure to avoid saturation.

One measurement made in range 1 is shown in figures 6.15 and 6.16.

The measurements were done with 19 incrementations, in total at 20 frequency points.

6.5 Evaluation of the results

6.5.1 Analysis of the body measurements results

Using a frequency scanning system for impedance measurements results in a large number of data. For the body measurements impedance values are obtained for 11 persons at 20 frequencies. Analysis of such an amount of data and larger, required use of statistical methods. Two methods have been used analysing the data collected from body measurements.

The measured data are given in Appendix C.

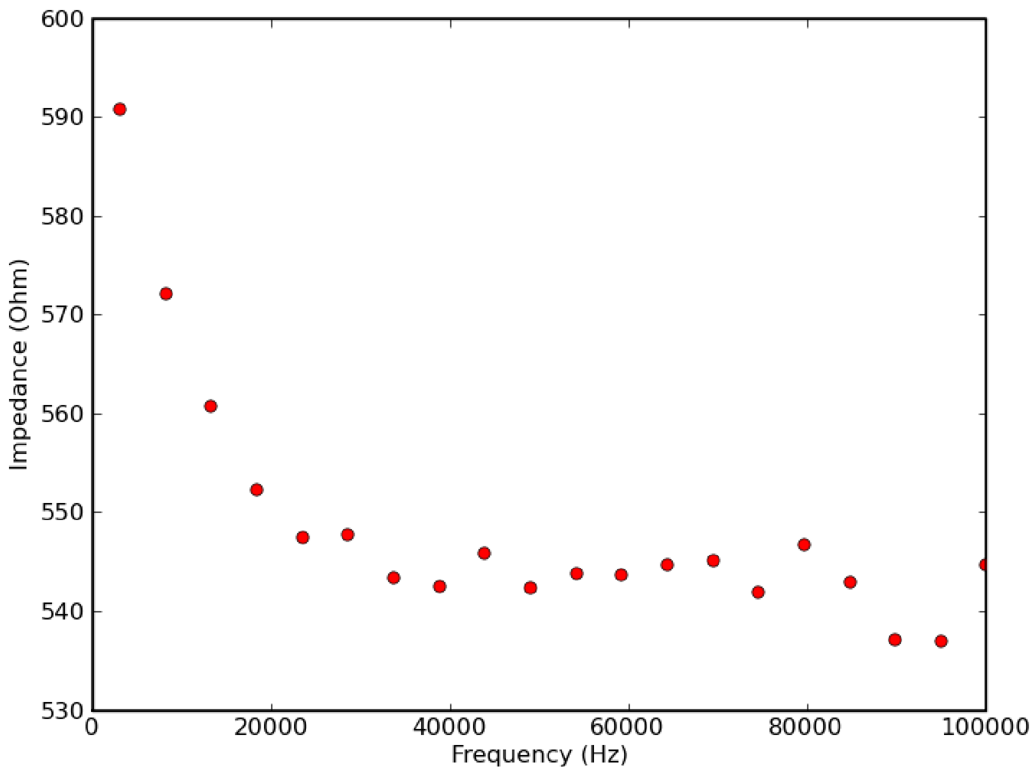


Figure 6.15: Impedance as function of frequency for a whole body measurement

The standard deviation approach

The standard deviation σ which is a measure of the dispersion of the values obtained has been calculated for each of the frequency points where measurements are made using the Microsoft Excel software. The average values of impedance and phase are plotted as functions of frequency and the curve also gives the standard derivation $\pm\sigma$. For the variable x the average is given by

$$\overline{x} = \frac{1}{n}\sum_{i=1}^{n} x_i \tag{6.20}$$

where n is the number of frequencies used and x_i represents either the impedance or the phase angle. The standard derivation calculated is given

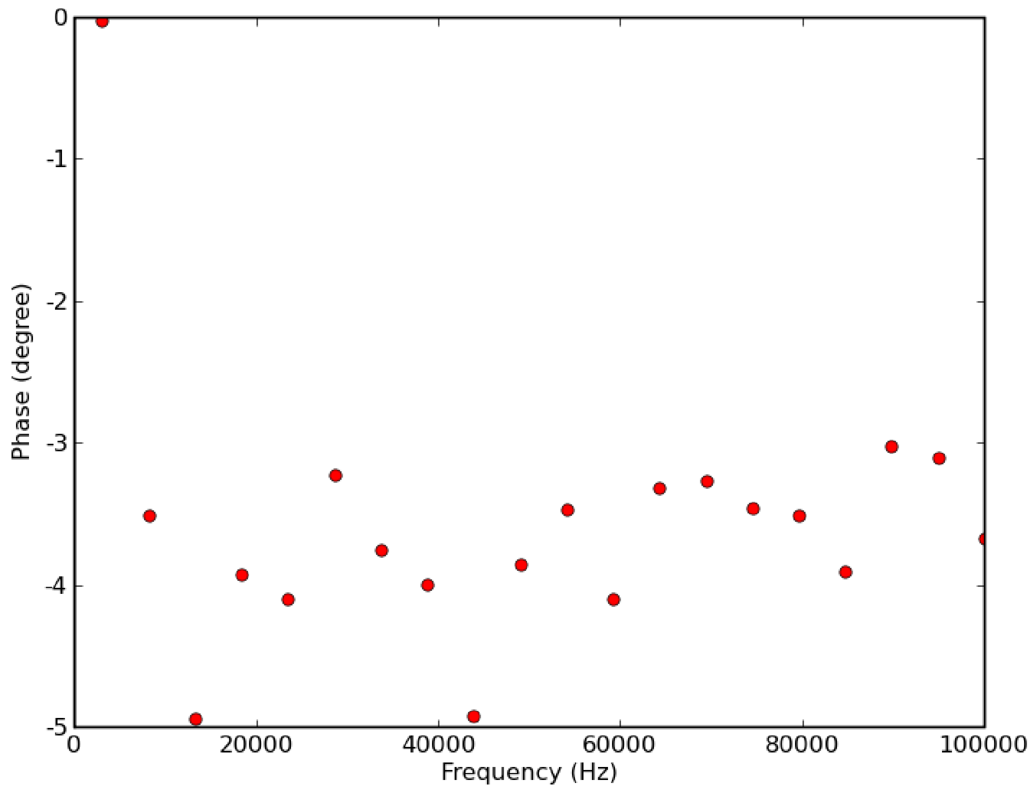


Figure 6.16: Phase as function of frequency for a whole body measurement

by

$$\sigma = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \overline{x})^2} \tag{6.21}$$

The results of this analysis are shown in figures 6.17 and 6.18. The resulting impedance curve shows a relatively constant average value down to about 30KHz, then it increases slightly (about 50Ω from 30kHz down to 3kHz). This is caused by the reduced intracellular conductivity due to the blocking effect of the capacitances at the cell membranes at low frequencies. Making an impedance body measurement of a person of about the same age as the rest of the group used, the resistance value is expected to be in the same range which is

- relatively constant from 100kHz down to 30kHz, and then it increases.

- show a pronounced increase in value of about 8-10% from 20-30kHz down to 3kHz.
- give a value within the $\pm\sigma$ limits (66% should be within this limit).

Marked deviations from these values should create concern if there is no obvious explanation.

The phase angle θ curve in 6.18 shows a slight decrease in the average θ from -2.5^o to -4^o over the frequency span. For the last measurement point there is an increase in the value. This however should most probably be ignored since the measurement at 3kHz lies outside the lower operational frequency limit of 5kHz using a 16.776MHz oscillator. There is another close to periodic variation in the data which is not fully explainable, but a possible effect relating to the heart rhythm has been considered. Again the curve seems to represent a good reference for measurements.

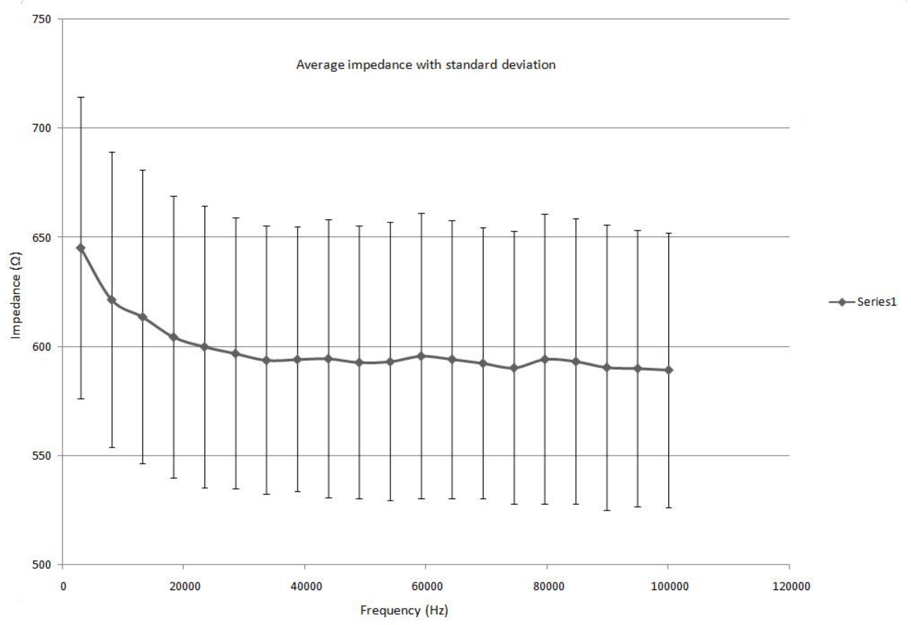


Figure 6.17: Plot of the average impedance for all the 11 measurements, shown with the standard deviation

Principal Components Analysis

Another method for analysis of multivariate data like this is the Principal Components Analysis. The general idea of this analysis is to find a structure in the data and to reduce the dimensionality of the data

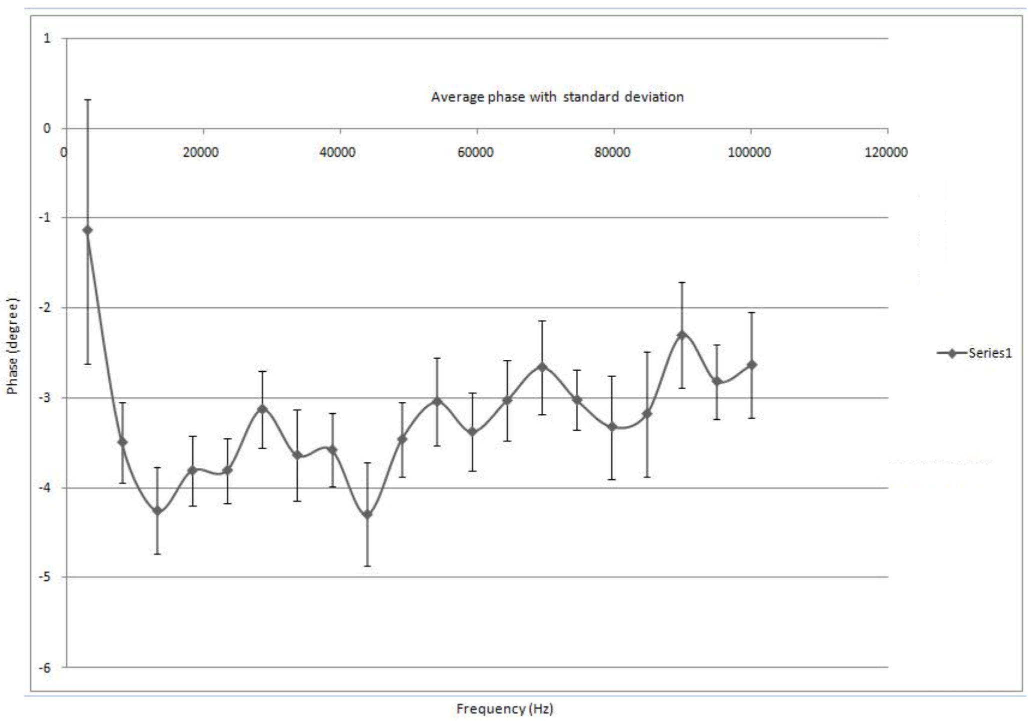


Figure 6.18: Plot of the average phase for all the 11 measurements, shown with the standard deviation

set and identify patterns in the data [21]. The process again starts taking the average values of the measured data and then forming the different covariance elements, and the matrix of the covariance elements. Obtaining the eigenvector with the highest eigenvalue is then the principal component of the data set. Eigenvectors are then ordered by value, which again means order of significance. If the less important are left out, the final data set will have a lower dimension than the original. There are several math programs available that perform this analysis. Using The Unscrambler software from Camo and using the Body Mass Index ($\frac{weight}{height^2}$) for the measured Y the curve in Figure 6.19 was obtained as the best fit. The predicted Y should be the closest possible to the measured.

The offset is the point where the regression line crosses the y-axis. The R^2 is a statistical measure of how well a regression line approximates the real data points, an R^2 of 1 represents a perfect fit. The obtained value of 0,76 is not bad, as is also observed from the plots.

RMSEC is the Root Mean Square Error of Calibration. It is expressed as average modeling error in the same units as the original response values.

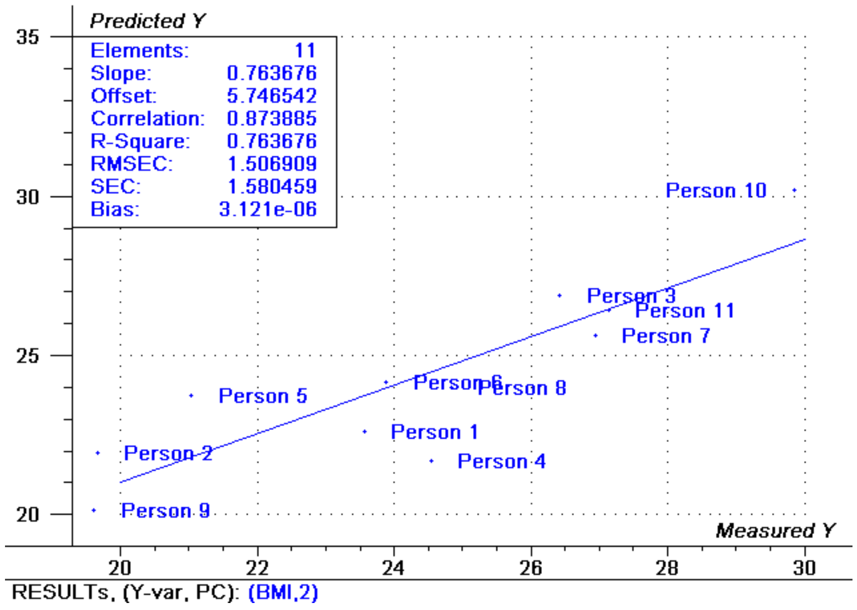


Figure 6.19: Predicted versus measured Y-values

The correlation between measurements and predictions is as high as 0.874. The slope of the regression line is 0.7636. Figure 6.20 shows the scores plotted the system of PC1 and PC2. The score plot shows similarities and differences among the samples, allowing for pattern investigation and identification of outliers. It is observed that the score of PC1 are much higher than the scores for PC2, and that measurements 8, 9 and 10 could be considered outliers. The lower values of PC2 relative to PC1, may however imply that sample 9 is the only outlier candidate. Looking at the first plot it is observed that the predicted value for 10 is somewhat high, but the deviation from the regression line is less than for some others. Figure 6.21 shows the residual validation variance used to find an optimal number of principal components. It is an expression for the modeling or for the prediction error: Residuals in general detects lack of fit in the model. The residuals are deviations between observed data values and the model approximation of those values. It is observed that the values are close to constant beyond Principal Component 2. Figure 6.22 shows the regression coefficient for all 20 phase angles and the impedances for 20 frequencies, f on this figure means phase. The regression coefficient when the regression line is linear, is the constant that represent the rate of change of one variable as a function of changes in the others. Because the phase is mostly zero, the phase does have a big role in the determination of the BMI. This is because the BMI is calculated by

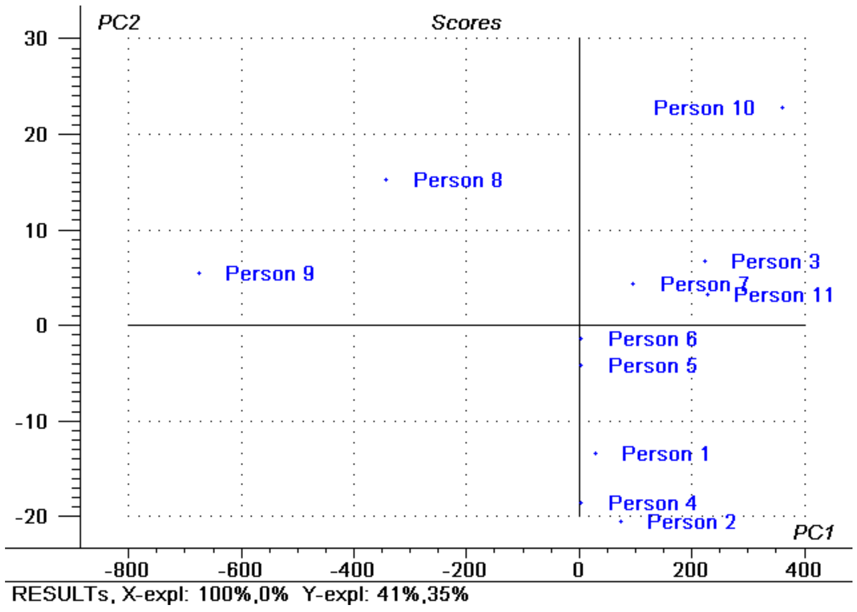


Figure 6.20: Scores for the individuals subjects plotted in the PC1/PC2 coordinate system

multiplying each component with its coefficient and then adding all of the contributions together.

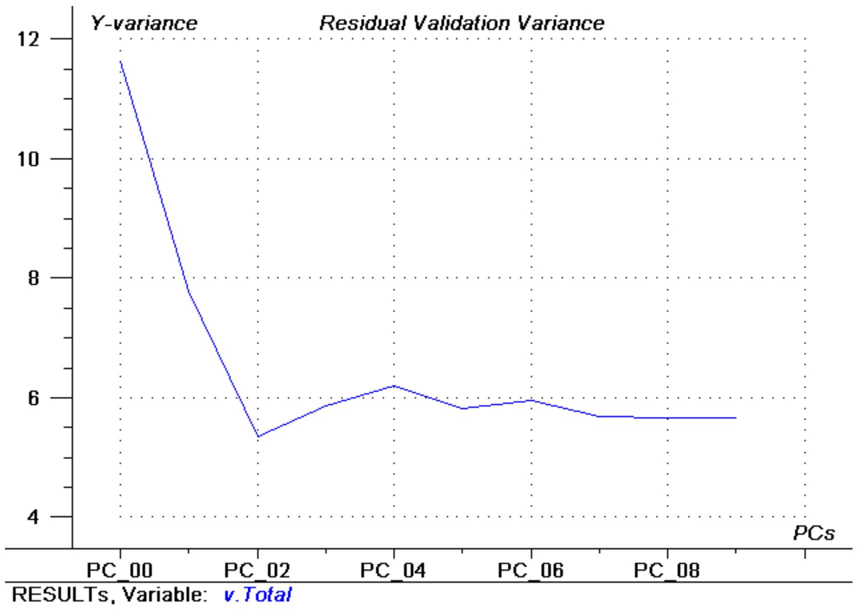


Figure 6.21: Residual validation variance as function of Principal Component number

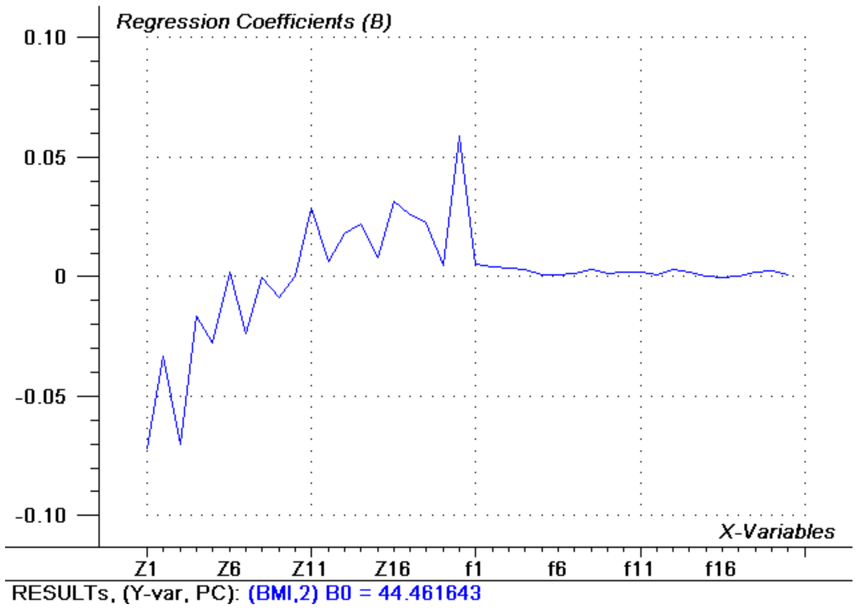


Figure 6.22: Regression coefficients for the impedance and phase

Chapter 7

Possible improvements of the system

The basic principles of the system are very interesting, even though the functionality of the prototype was not the best. But learning about problems have also given some ideas on possible improvements, using both hardware and software.

7.1 External reference oscillator

An external very stable oscillator would improve stability and reduce the variations observed in the measured θ values. An external unit with two oscillators one operating at 16.776MHz and the other at 2MHz would make it possible to measure down to 300Hz in two frequency measurement ranges, 5-100kHz (as now) and 100Hz-5kHz.

7.2 Improved calibration system

A network with one set of calibration/R_{FB} resistors, one set for each measurement range would simplify the calibration procedure, which should be in three steps, the first to identify the measurement range, the second to function as a reference and check for the range, and as a reference for the initial measurement. The third would required externally connected calibration and feed back resistors of value close to the device under test.

7.3 Program for calculating the input level to the ADC

A program for calculating input level to the ADC could be made and function as a controll of the critical level.

7.4 New device under development by Analog Devices

Analog Devices has realized that the AD5933 is not well suited for bioimpedance measurements and has in an email told that they are currently working on a modified IC. Hopefully most of the developed software can be used in that module too. The succeeder of AD5933/5934 is planned for release in 2009 sometime. It may also be able to operate at higher frequencies than the AD5933, possibly up to 200kHz. That would be an improvement.

7.5 Systems improvement

The prototype as it is, can be used for single frequency measurements, multifrequency measurements and spectroscopy, but the frequency band is somewhat narrow. If that could be increased to 200-400kHz the applicability would increase.

It could also be possible to build a cheap scanner making measurements using a number of contacts. That would require a interface unit with two contacts in and many out, where the two in contacts could be switched between the different contacts. For a fixed excitation contact a swept measurement should be made on the other contacts. The data would have to be taken from the impedance converter and stored on a PC for each run, but that could be managed by adding some features to the existing software code.

Chapter 8

Conclusion

A prototype of a new system for bioimpedance measurements has been developed and body measurements performed. The system designed and implemented is based on use of the integrated circuit AD5933. Operational procedures have been developed and implemented in operational software for PC operation via a microcontroller. The prototype which operates in the frequency range 5 - 100 kHz is well suited for single frequency measurements, multifrequency measuremnts and spectroscopic measurements.

Verification testing performed using fixed components revealed that the calibration process is very critical due to a saturation effect occuring in the ADC in the receiver part which was very sensitive to gain variations and signal level in the signal loop. Though this represents a problem, the limitations of the circuit have been understood and described. Safe measurement critera have been established. Performing calibration close to the value of the device under test, resulted in repeatable and reliable measurement.

Whole body measurements have been performed and the results have been statistically analysed using both a Standard Deviation software tool and a software tool for Principal Components Analysis. They both gave results that can be used for identification of "normal" measurement values and also possible outliers.

Even though there were some limitations in device performance, the goals set have all been met. The principles employed are promising allowing for collection of large amounts of data that can be treated statistically for information retrival. With some hardware improvements a very attractive low cost system could be made something that could also be of interest for other areas like food control and plant measurements.

Bibliography

[1] S. Grimnes and Ø. G. Martinsen, *Bioimpedance and Bioelectricity Basics*, Academic Press, 2000.

[2] Analog Devices, *AD5933 datasheet, updated may 2008*, attainable from:
http://www.analog.com/UploadedFiles/Data_Sheets/AD5933.pdf

[3] Analog Devices, *AD5933 evaluation board datasheet*, attainable from
http://www.analog.com/UploadedFiles/Associated_Docs/537700023EVAL_AD5933EB.pdf

[4] K. S. Cole and R. H. Cole, *Dispersion and absorption in dielectrics–I, alternating current characteristics.*, J. Chem Phys 1941, vol. 9, pp. 341-951.

[5] E. C. Hoffer et al., *Correlation of whole-body impedance with total body water volume*, Journal of Applied Physiology, vol. 27, No. 4, pp. 531-534, October 1969.

[6] H. Fricke, *Theory of electrolytic polarisation*, Philosophical Magazine, vol. 14, pp. 310-318, 1932.

[7] U. G. Kyle et al., *Bioelectric impedance analysis-part I: review of principles and methods.*, Clinical Nutrition 23, pp 1226-1243, 2004.

[8] S. Grimnes and Ø. G. Martinsen, *Cole Electrical Impedance Model - A Critique and a alternative*, IEEE Transactions on Biomedical Engineering Vol. 52, No. 1, January 2005.

[9] W. J. Hannanet et al., *Evaluation of multi frequency analysis for the assessment of extracellular and total body water in surgical patients.*, Clin. Sci, vol. 86 pp. 479-485, 1994.

[10] K. J. Ellis, *Human body composition. In Vivio methods*, Physiol. Rev. 80, pg 649-680, April 2000

[11] Z-M Wang et al., *The five level model: A new approach to organizing body composition research*, Americal J. of Clinical Nutrition: vol 56, pp 19-28, 1992.

[12] Y. Zou and Z. Guo, *A review of electrical impedance techniques for breast cancer detection*, Medical Engineering & Physics (Elsevier) 25, pp 79-90, 2003.

[13] B. Dewberry, *A Review of Electrical Impedance Spectroscopy Methods for Parametric Estimation of Physiologic Fluids Volumes*, NASA/TM-200-21200,

[14] A. Stojadinovic et al., *Electrical Impedance Scanning of Breast Cancer in Young Women: Preliminary Results of a Multicenter Perspective Clinical Trial*, J of Clinical Oconology, Vol 23, No 12, pp 2703-2715, april 2005.

[15] R. Y. Halter et al., *Electrical Impedance Spectroscopy of Prostatic Tissues.*, ICEBI 2007, IFMBE Proceedings 17, pp 126-129, 2007.

[16] S. Hong et al., *Integrated Data Collection in Electrical Impedance Tomography*, ICEBI 2007, IFMBE Proceedings 17, pp 348-357, 2007.

[17] Y. He et al., *Preliminary study of a Cole-Cole model curve fitting method for Electrical Impedance Tomography*, ICEBI 2007, IFMBE Proceedings 17, pp 436-439, 2007.

[18] D. C. Barber, *Electrical Impedance Tomography*, The Biomedical Engineering Handbook, Second Edition, Chapter 68, CRC Press LLC, 2000

[19] ATmel, *ATmega16 datasheet*, attainable from:
http://www.atmel.com/dyn/resources/prod_documents/doc2466.pdf

[20] Atmel, *STK500 User Guide* attainable from:
http://www.atmel.com/dyn/resources/prod_documents/doc1925.pdf

[21] L. I. Smith, *A tutorial in Principal Components Analysis*, 2002, attainable from:

http://csenet.ontago.ac/nz/cosc453/student_turtorials/principal components

[22] P. Åberg, *Skin Cancer as seen by Electrical Impedance*, PhD thesis, Karolinska Instituttet Stocholm, ISBN 91-7140103, 2004.

Appendix A

Microcontroller code

A.1 file:main.c

```
#include <avr/io.h>
#include "myusart.h"
#include "mytwi.h"
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>

/*AD5933 registers adresses*/
#define real_high_reg 0x94
#define real_low_reg 0x95
#define im_high_reg 0x96
#define im_low_reg 0x97
#define status_reg 0x8F
#define control_high_reg 0x80
#define control_low_reg 0x81
#define NumSet_high 0x8A
#define NumSet_low 0x8B
#define startfreq_reg 0x82
#define freqinc_reg 0x85
#define incsteps_reg 0x88

#define AD5933CLK 16776000

//Function for converting the two hex values to a decimal value
```

```
unsigned long int hextodec(unsigned char data_high,\\
unsigned char data_low)
{
    unsigned long int data, temp;
    data=(unsigned long int)data_high*256+data_low;
    return data;
}

int main (void)
{
    #Declaration of variables
    unsigned char* data;
    char s[6];
    char x[6];
    int R, I,j;
    unsigned long int i, kl;
    char* val;
    unsigned char real_high, real_low;
    unsigned char im_low, im_high;
    unsigned int startfreq, numbinc,freqinc;
    DDRB=0xFF;

    /*Initializing USART*/
    USART_init();
    /*Initializing TWI*/
    i=TWI_init();



    /*Allocating memory for pointer which is
    used to store data to be written to the AD5933*/
    data = (unsigned char*) malloc(10*sizeof(unsigned char));
    /*Allocating memory for pointer which
    is used to store data from usart*/
    val=(char*) malloc(10*sizeof(char));
    //Setting up connection with the PC interface
com:
    val=USART_CharReceive();
    if((i=strncmp(val, "Init",4))==0){
        USART_CharTransmit("AD5933");
    }
```

```
else{
    goto com;
}


//Setting startfreq routine
startfreq=atoi(USART_CharReceive());
i=startfreq*32.0023195;
*data=0x000000ff & (i>>16); *(data+1)=0x000000ff & (i>>8);
*(data+2)=0x000000ff & i;
TWI_block_write(startfreq_reg, 3,data);

//Setting frequency increment
freqinc=atoi(USART_CharReceive());
i=freqinc*32.0023195;
*data=0x000000ff & (i>>16); *(data+1)=0x000000ff & (i>>8);
*(data+2)=0x000000ff & i;
TWI_block_write(freqinc_reg, 3, data);

//Setting number of increments
numbinc=atoi(USART_CharReceive());
*data=0x000000ff & (numbinc>>8); *(data+1)=0x000000ff & numbinc;
TWI_block_write(incsteps_reg, 2, data);

/*User input on number of settlings from USART*/
/*Could have been made user settable
the same way as the output excitation voltage*/
TWI_byte_write(NumSet_high, 0x00);
TWI_byte_write(NumSet_low, 0x32);

//receiving desired voltrange from PC
val=USART_CharReceive();
if((i=strncmp(val, "V1",2))==0){
    j=1;
}
else if((i=strncmp(val, "V2",2))==0){
    j=2;
}
else if((i=strncmp(val, "V3",2))==0){
    j=3;
}
```

```
else if((i=strncmp(val, "V4",2))==0){
    j=4;
}

/*PLacing AD5933 in standby mode, see manual p. 20-21*/
if(j==1){
    PORTB=0x1F;
    TWI_byte_write(control_high_reg, 0xb1);
}
else if(j==2){
    PORTB=0x2F;
    TWI_byte_write(control_high_reg, 0xb7);
}
else if(j==3){
    PORTB=0x4F;
    TWI_byte_write(control_high_reg, 0xb5);
}
else if(j==4){
    PORTB=0x8F;
    TWI_byte_write(control_high_reg, 0xb3);
}

//Initialize with start frequency:
if(j==1){
    TWI_byte_write(control_high_reg, 0x11);
}
else if(j==2){
    TWI_byte_write(control_high_reg, 0x17);
}
else if(j==3){
    TWI_byte_write(control_high_reg, 0x15);
}
else if(j==4){
    TWI_byte_write(control_high_reg, 0x13);
}


//Some settling time
for(i=0;i==100;i++);

//Start sample routine
```

```
    while(1){
        val=USART_CharReceive();
        if((i=strncmp(val, "StSample",8))==0){
            /*Programming start frequency sweep
            and voltage range and PGA gain*/
            //TWI_byte_write(control_high_reg, 0x25);
            if(j==1){
                TWI_byte_write(control_high_reg, 0x21);
            }
            else if(j==2){
                TWI_byte_write(control_high_reg, 0x27);
            }
            else if(j==3){
                TWI_byte_write(control_high_reg, 0x25);
            }
            else if(j==4){
                TWI_byte_write(control_high_reg, 0x23);
            }
            break;
        }
    }
start:
    //Waits until the real and imaginary data in the AD5933 is valid
    while(!(TWI_byte_read(status_reg) & 0x02));

    //Reads the two hex values from the real register
    real_high=TWI_byte_read(real_high_reg);
    real_low=TWI_byte_read(real_low_reg);
    //Converting the real value to decimal
    R=hextodec(real_high, real_low);
    //Converts to aascii for transmission to computer
    itoa(R, s, 10);
    USART_CharTransmit(s);

    im_high=TWI_byte_read(im_high_reg);
    im_low=TWI_byte_read(im_low_reg);
    I=hextodec(im_high, im_low);
    itoa(I, x, 10);
    USART_CharTransmit(x);

    //Test if the sweep is complete, if not complete program increment frequency
    if((TWI_byte_read(status_reg) & 0x04)==0){
        for(kl=0; kl==10000; kl++);

        if(j==1){
            TWI_byte_write(control_high_reg, 0x31);
        }
        else if(j==2){
            TWI_byte_write(control_high_reg, 0x37);
        }
        else if(j==3){
            TWI_byte_write(control_high_reg, 0x35);
        }
        else if(j==4){
            TWI_byte_write(control_high_reg, 0x33);
        }
        goto start;
    }
    //If complete programming power down mode
    else{
        TWI_byte_write(control_high_reg, 0xA1);
        goto com;
    }
}
```

A.2 file:myusart.c

```
#include <avr/io.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "myusart.h"

int USART_init(void){
   //Initializing the USART
   // Turn on the transmission and reception :
   UCSRB |= (1 << RXEN) | (1 << TXEN);
   //Use 8-bit character sizes:
   UCSRC |= (1 << URSEL) | (1 << UCSZ0) | (1 << UCSZ1);
   // Load lower 8-bits of the baud rate
   //value into the low byte of the UBRR register
   UBRRL = BAUD_PRESCALE;
   // Load upper 8-bits of the baud rate
   //value into the high byte of the UBRR register
   UBRRH = (BAUD_PRESCALE >> 8);
   return 1;
}

void USART_transmit(char data){
    // Do nothing until UDR is ready for more data to be written to it
    while ((UCSRA & (1 << UDRE)) == 0) {};
    UDR = data;

}

int USART_receive(void){
    char data;
    // Do nothing until data have been recieved and is ready to be read from UDR
    while ((UCSRA & (1 << RXC)) == 0) {};
    return UDR;
}

void USART_CharTransmit(char* data)
{
    int n;
    n=0;
    while (1)
   {
        // Do nothing until UDR is ready for more data to be written to it
        while ((UCSRA & (1 << UDRE)) == 0) {};
        UDR =*(data+n);
        n++;
        if(!(*(data+(n)))){
            break;
        }
   }
   USART_transmit('\n');
   USART_transmit('\r');
}

char* USART_CharReceive(void){
    int i;
    char* data;
    char temp;
    data = (char*) malloc(30*sizeof(char));
    for(i=0; i<30; i++){
        temp =USART_receive();
        //If received data is equal to carriage return stor receiving
        if(temp=='\r'){
            break;
        }
        *(data+i)=temp;
    }
    *(data +i)='\0';
    return data;
}
```

A.3 file:myusart.h

```
//Definisjoner:
#define USART_BAUDRATE 9600
#define BAUD_PRESCALE (((F_CPU / (USART_BAUDRATE * 16UL))) - 1)
#define DEBUG 0

//Funksjonsdefinisjoner:
int USART_init(void);
void USART_transmit(char data);
int USART_receive(void);
void USART_CharTransmit(char* data);
char* USART_CharReceive(void);
```

A.4 file:mytwi.c

```
//Code is written by Bernt Nordbotten as a part of my master thesis

#include <stdint.h>
#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/twi.h>
#include "myusart.h"
#include "mytwi.h"

/*Atmega32 I2C bus status*/
#define START 0xa4
#define Stop 0x94
#define Trans 0x84

/*AD5933*/
#define SLA_read 0x1B
#define SLA_write 0x1A


#define SUCCES 0xff


/*Wait for TWINT flag set*/
void TWI_wait(void){
    int j=0;
    while(!(TWCR &(1<<TWINT))){
        j++;

    }
}

/*Initialize TWI*/
int TWI_init(void){
    TWBR=10; /*Setter SCL frekvensen til ca. 100kHz */
    TWCR=0x04; /*Send start condition*/
    return 1;
}

unsigned char Send_start(void)
```

```
{
    TWCR=START; //Send START
    TWI_wait(); //Wait for TWI interrupt flag to be set
    if((TWSR & 0xF8)!=0x08 || (TWSR & 0xF8)!=0x10)

        return TWSR; //If it failed, return the TWSR value
    return 0xFF; //If succeeded, return SUCCESS
}
/*Send stop condition*/
void TWI_stop(void){
    TWCR=Stop;
}

/*Send address*/
unsigned char TWI_send_adr(unsigned char adr){
    TWI_wait();
    TWDR=adr;
    TWCR=Trans;
    TWI_wait();
    /*If nack received from slave:*/
    if((TWSR & 0xF8)!= 0x18 || (TWSR & 0xF8)!= 0x40){

        return TWSR;
    }
    return SUCCES;
}

/*Send one byte to the bus*/
unsigned char TWI_send_byte(unsigned char data){
    TWI_wait();
    TWDR=data;
    TWCR=Trans;
    TWI_wait();
    if((TWSR & 0xF8) != 0x28){ /*If ack received from slave*/

        return TWSR;
    }
    else{
        return SUCCES;
    }
}
```

```
unsigned char TWI_set_memloc(unsigned char mem_location){
    Send_start();
    TWI_send_adr(SLA_write);
    TWI_send_byte(0xB0); /*Adress pointer see page 26 of AD5933 manual*/
    TWI_send_byte(mem_location); /*Send memory location*/
    return 1; /*Return 1 if succeded*/
}

unsigned char TWI_byte_write(unsigned char reg_addr,\\
unsigned char data){
    Send_start();
    TWI_send_adr(SLA_write);
    TWI_send_byte(reg_addr);
    TWI_send_byte(data);
    TWI_stop();
    return 1; /*Return 1 when succeded*/
}
/*Write several bytes of data*/
/*byte_number=number of data bytes to be sendt*/
unsigned char TWI_block_write(unsigned char reg_location,\\
unsigned char byte_number, unsigned char *TWI_data)
{
    int i;

    TWI_set_memloc(reg_location);
    Send_start();
    TWI_send_adr(SLA_write);
    TWI_send_byte(0xA0); /*Block write command, page 26 AD5933 manual*/
    TWI_send_byte(byte_number);
    for (i=0; i<byte_number; i++){
        TWI_send_byte(*(TWI_data+i));
    }
    TWI_stop();
    return 1;
}

unsigned char TWI_byte_read(unsigned reg_addr){

    TWI_set_memloc(reg_addr);
    Send_start();
    TWI_send_adr(SLA_read);
    TWCR=Trans;
    TWI_wait();
    return TWDR;
}


unsigned char  TWI_block_read(unsigned char reg_addr,\\
unsigned char byte_number, unsigned char *TWI_data)
{
    int i;

    TWI_set_memloc(reg_addr);

    Send_start();
    TWI_send_adr(SLA_write);
    TWI_send_byte(0xA0);
    TWI_send_byte(byte_number);
    TWI_init();
    TWI_send_byte(SLA_read);
    for(i=0; i<byte_number; i++){
        *(TWI_data +i)=TWDR;
        TWI_wait();
        TWCR|=(1<<TWEA); /*Send ACK after each byte*/
    }
    TWCR=(0<<TWEA); /*Send NACK to signalise last byte (end of read)*/
    TWI_wait();
    TWI_stop();
    return *TWI_data;
}
```

A.5 file:mytwi.h

```
/*Prosedures*/
int TWI_init(void);
void TWI_wait_int(void);
void TWI_stop(void);

/*Functions*/
unsigned char TWI_send_adr(unsigned char adr);
unsigned char set_memlocation(unsigned char mem_location);
unsigned char TWI_send_byte(unsigned char data);
unsigned char TWI_byte_write(unsigned char reg_addr, unsigned char data);
unsigned char TWI_block_write(unsigned char reg_location,\\
unsigned char byte_number, unsigned char *TWI_data);
unsigned char TWI_byte_read(unsigned reg_addr);
unsigned char TWI_block_read(unsigned char reg_addr,\\
unsigned char byte_number, unsigned char *TWI_data);
```

Appendix B

Python code for graphical interface

```
#!/usr/bin/env python
#Imports the different libraries that is used.
import os, sys, serial, time, Pmw, re, tkMessageBox, math, pylab
from Tkinter import *


class AD5933GUI:
    #The constructor function:
    def __init__(self,parent):
        #Stores the main frame
        #(alle the other frames will be within in this parent frame)
        self.parent=parent
        self.k=0
        self.l=0
        self.voltranges=['2.0Vp-p', '1.0Vp-p', '400mVp-p', '200mVp-p']

        #Declaration of different TkInter variables that will
        #be used to store user given values from the graphical interface:
        self.com=StringVar()
        self.calresistor=StringVar()
        self.startfreq=StringVar()
        self.numbinc=StringVar()
        self.numbsteps=StringVar()
        self.volt=StringVar()
        self.gainfactor=DoubleVar()
self.com.set('1')
self.startfreq.set('3000')
self.numbinc.set('5105')
self.numbsteps.set('19')

#Set a default value for the gain factor
#and for the excitation voltage
self.gainfactor.set(2.44007*(10**-6))
self.volt.set('200mVp-p')

#Declares two dictionaries with several lists in them.
#self.cal is for storing the system phase that is read
#from calphase.txt. self.data is
#used to store the data for the sweep.
self.cal={'Freq':[], 'Phase':[], 'Gain_Fac':[], 'Gain':[]}
self.data={'Real':[] , 'Im':[], 'Freq':[], 'Mag':[],\\
'Impedance':[], 'Phase':[], 'Phase2':[]}

#Set a name for the graphical interface
#that will appear on top part of the interface
self.parent.title('Interface GUI for AD5933 circuit')

#Declared some different frames
#wich will be putted different widgets in.
self.left=Frame(parent)
self.bottom=Frame(parent)
self.plot=Frame(parent)
self.status=Frame(parent)


#Packs the frames. The frames is packed from top to bottom.
#So that self.left frame will appear on top and self.status
#will appear on the bottom.
self.left.pack(side='top', anchor='w')
self.bottom.pack(side='top', anchor='w')
self.plot.pack(side='top', anchor='w')
self.status.pack(side='top', anchor='w')

self.number_increments=0

#Declares some different widgets that will be used in
```

```
        #the interface for user input.
        #EntryField=Entry field where the user can input numbers or text.
        #Button=Each button are linked to different functions in the code,
#clicking a button on the interface will call that’s button function.
        #OptionMenu=OptionMeny will present the user for a scroll down meny
#with some predefined options that the user can choose between.
        #ScrolledListBox=A list box where text can be displayed.
        self.port=Pmw.EntryField(self.bottom,
            labelpos='w',
            label_text='Com port',
            entry_width=8,
            entry_textvariable=self.com)
        self.setstartfreq=Pmw.EntryField(self.bottom,
            labelpos='w',
            label_text='Start frequency (Hz)',
            entry_width=8,
            entry_textvariable=self.startfreq)
        self.setnumbinc=Pmw.EntryField(self.bottom,
            labelpos='w',
            label_text='Frequency increment (Hz)',
            entry_width=8,
            entry_textvariable=self.numbinc)
        self.setnumbsteps=Pmw.EntryField(self.bottom,
            labelpos='w',
            label_text='Number of steps',
            entry_width=8,
            entry_textvariable=self.numbsteps)
        self.setgainfactor=Pmw.EntryField(self.bottom,
            labelpos='w',
            label_text='Gain factor',
            entry_width=8,
            entry_textvariable=self.gainfactor)
        self.calres=Pmw.EntryField(self.bottom,
            labelpos='w',
            label_text='Calibration resistor (ohm)',
            entry_width=8,
            entry_textvariable=self.calresistor)
        self.calc=Button(self.bottom,text='Calculate real and im', width=23,\\
        command=self.realim)
        self.setvoltrange=Pmw.OptionMenu(self.bottom,
               labelpos='w',  # n, nw, ne, e, and so on
```

```
                label_text='Output voltage range',
                items=self.voltranges,
                menubutton_textvariable=self.volt,
                menubutton_width=8)

        self.quit=Button(self.bottom, text=' Quit ', width=23,\\
        command=self.quit)
        self.calrun=Button(self.bottom, text='Calibration run',\\
        width=23, command=self.calibration)
        self.sample=Button(self.bottom, text='Start sweep', width=23,\\
        command=self.sample)
        self.statuslist=Pmw.ScrolledListBox(self.status,
            vscrollmode='static', hscrollmode='dynamic',
            listbox_width=40, listbox_height=5,
            label_text='Status',
            labelpos='n')
        self.plot_p=Button(self.plot, text='Plot phase', width=23,\\
        command=self.plot_phase)
        self.plot_impedance=Button(self.plot, text='Plot impedance', \\
        width=23, command=self.plot_imp)


        #This will align the labels for the widgets, this can be necesary
        #because thelabels have different lengths.
        widgets=(self.calres, self.port,self.calc, self.setstartfreq,\\
self.setnumbinc, self.setnumbsteps, self.setvoltrange, self.setgainfactor)
        Pmw.alignlabels(widgets)

        #After the labels are aligned the widgets are packed
        #in the different frames.
        self.port.pack(side='top', anchor='w')
        self.setstartfreq.pack(side='top', anchor='w')
        self.setnumbinc.pack(side='top', anchor='w')
        self.setnumbsteps.pack(side='top', anchor='w')
        self.setvoltrange.pack(side='top', anchor='w')
        self.setgainfactor.pack(side='top', anchor='w')
        self.calres.pack(side='top', anchor='w')
        self.sample.pack(side='top', anchor='w')
        self.calrun.pack(side='top', anchor='w')
        self.calc.pack(side='top', anchor='w')
        self.quit.pack(side='top', anchor='w')
        self.plot_impedance.pack(side='left', anchor='w')
        self.plot_p.pack(side='left', anchor='w')
        self.statuslist.pack(side='top',expand=1,fill='both');

        #This will open the file named calphase.txt for reading
        #Then read in every line and store them,
        #except for the first line wich only
        #is information to the user on wich column\\
        #is frequency and wich is the system phase.
        #If this doesn't work (for example the file don't exsist)
        #the code will go to the except state and \\
        #print error 1 on the screen.
        try:
            ifile=open('calphase.txt', 'r')
            ifile2=open('gain_factor.txt', 'r')
            trash=ifile.readline()
            trash=ifile2.readline()
            for line in ifile.readlines():
                freq, phase=line.split()
                self.cal['Freq'].append(freq)
                self.cal['Phase'].append(phase)
            for line in ifile2.readlines():
                freq, gain=line.split()
                self.cal['Gain_Fac'].append(gain)
        except:
            print 'error 1'




    #This function is for calculating G and B (and the theoretical values)
#for an parallell conection between a resistor and capacitor.
    def realim(self):
        #Reads the resistor and capacitance value from the command line
        res=float(sys.argv[1])
        cap=float(sys.argv[2])

        #Open file realim.txt for writing
        ofile2=open('realim.txt', 'w')

        ofile2.write('Resistor=%gohm\nCapacitor=%gF\n' %(res, cap))
        ofile2.write('\nFreq  G  Gteo  B  Bteo\n')
        #Does the calculation and write to file
        for i in range(0, len(self.data['Freq'])-1):
            f=float(self.data['Freq'][i])
            Z=float(self.data['Impedance'][i])
            theta=float(self.data['Phase2'][i])
            R=float(Z*math.cos(theta/57.2957795))
            X=float(Z*math.sin(theta/57.2957795))

            G=R/float((R**2)+(X**2))
            B=-X/((R**2)+(X**2))*1.56
            Gteo=1/res
            Bteo=2*3.14*f*cap
            ofile2.write('%g  %g  %g  %g  %g\n' %(f, G, Gteo, B, Bteo))

    #Function for calibration routine.
#Saves the system phase to calphase.txt and calculates the gain
#factor for every frequency and stores to gain_factor.txt
    def calibration(self):
        self.data={'Real':[] , 'Im':[], 'Freq':[], 'Mag':[],\\
        'Impedance':[], 'Phase':[], 'Phase2':[]}
        #If the com port is given by the
        #user then try to open the com port,
#if the com port for example already is used by another software
        #then display a message to user explaining that
        #it could not open specified com port.
        #If the com port is not defined the display a message to user
#telling the user to specify the com port to use.
        if self.com.get():
            self.comport=int(self.com.get())-1
            try:
#Open the specified com port
                self.ser = serial.Serial(self.comport)
            except:
                self.statuslist.insert('end',\\
                'Could not open specified com port')
        else:
            self.statuslist.insert('end',\\
            'Please specify com port to open')

        self.ser.flush()
        #Send 'Init\r' to microcontroller via serial port.
        #This is the initializing keyword for the microcontroller.
        self.ser.write('Init\r')

        #Reads from the serial port, if the received data is 'AD5933'
        #then it means the microcontroller received the Init,
        #and it has sendt AD5933 as an ack of recieved init.
        #Is using regex, but could just have checked if equal
        ID = self.ser.readline()
        Pattern="AD5933"
        match=re.search(Pattern, str(ID))
        if match:
            self.statuslist.insert('end',\\
            'Communication with microcontroller established')
        else:
            self.statuslist.insert('end',\\
            'Error while initializing communication with microcontroller,')
            self.statuslist.insert('end',\\
            'please check specified com port')

        #Sending the start frequency to the microcontroller
#and giving the user feedback that the start frequency is beeing set.
        self.statuslist.insert('end', 'Setting startfreq')
        self.ser.write(self.startfreq.get()+'\r')

        #Sending frequency increment,
        #and giving feedback to the user of the progress.
        self.statuslist.insert('end', 'Setting frequency increment')
        self.ser.write(self.numbinc.get()+'\r')

        #Sending number of frequency incrementations,
        #and giving the user feedback of the progress.
        #Had to add 1 to the number of increments
        #to get the correct number of increments.
        self.statuslist.insert('end', 'Setting number of frequency increments')
        self.number_increments=int(self.numbsteps.get())+1
        self.ser.write(str(self.number_increments)+'\r')

        #If test for sending the correct desired
        #voltage range to microcontroller,
        #and giving feedback to user of the progress
        self.statuslist.insert('end', 'Setting excitation voltage')
        if self.volt.get()=='2.0Vp-p':
            self.ser.write('V1'+'\r')
        elif self.volt.get()=='1.0Vp-p':
            self.ser.write('V2'+'\r')
        elif self.volt.get()=='400mVp-p':
            self.ser.write('V3'+'\r')
        elif self.volt.get()=='200mVp-p':
            self.ser.write('V4'+'\r')

        #Sending the command for starting sweep to microcontroller
#and giving feedback to user about progress.
        self.ser.write('StSample\r')
        self.statuslist.insert('end', 'Starting sampling')

        count=0
        i=0
        #For loop that for each time calls
        #the sampling function self.savedata
        #with the number of increments
        #(this is used to calculate the current frequency).
        for i in range(0, int(self.number_increments)):
            self.savedata(i)

        #Open files calphase.txt and gain_factor.txt for writing.
        #Writes the frequency and system phase to calphase.txt
#and frequency and gain factor to gain_factor.txt.
        ofile=open('calphase.txt', 'w')
        ofile.write('Freq Phase\n')
        ofile2=open('gain_factor.txt', 'w')
        ofile2.write('Freq Gain factor\n')
        for k in range(0, int(self.number_increments)):
            print k
            str1='%d  %g\n' %(self.data['Freq'][k], self.data['Phase'][k])
            gainfac=1/(float(self.calresistor.get()))
            gainfac=gainfac/(float(self.data['Mag'][k]))
            str2='%d  %g\n' %(self.data['Freq'][k], gainfac)
            ofile.write(str1)
            ofile2.write(str2)
        ofile.close()
        ofile2.close()
```

```
#Function used for a measurement sweep
#The starting part is the same as the calibration function,
#so see on those comment.
def sample(self):
    self.data={'Real':[] , 'Im':[], 'Freq':[], 'Mag':[],\\
    'Impedance':[], 'Phase':[], 'Phase2':[]}
    if self.com.get():
        self.comport=int(self.com.get())-1
        try:
            self.ser = serial.Serial(self.comport)
            #self.statuslist.insert('end', '%s is open at %s baudrate'\\
            %(ser.portstr,ser.baudrate))

        except:
            self.statuslist.insert('end',\\
            'Could not open specified com port')
    else:
        self.statuslist.insert('end','Please specify com port to open')
    self.ser.flush()
    self.ser.write('Init\r')
    self.ser.flushInput()
    self.ser.flush()

    ID = self.ser.readline()
    Pattern="AD5933"
    match=re.search(Pattern, str(ID))

    if match:
        self.statuslist.insert('end', \\
        'Communication with microcontroller established')
    else:
        self.statuslist.insert('end',\\
        'Error while initializing communication with microcontroller,')
        self.statuslist.insert('end',\\
        'please check specified com port')

    self.statuslist.insert('end', 'Setting startfreq')
    self.ser.write(self.startfreq.get()+'\r')
self.statuslist.insert('end', 'Setting frequency increment')
self.ser.write(self.numbinc.get()+'\r')

self.statuslist.insert('end', 'Setting number of frequency increments
self.number_increments=int(self.numbsteps.get())+1

self.ser.write(str(self.number_increments)+'\r')

self.statuslist.insert('end', 'Setting excitation voltage')
if self.volt.get()=='2.0Vp-p':
    self.ser.write('V1'+'\r')
elif self.volt.get()=='1.0Vp-p':
    self.ser.write('V2'+'\r')
elif self.volt.get()=='400mVp-p':
    self.ser.write('V3'+'\r')
elif self.volt.get()=='200mVp-p':
    self.ser.write('V4'+'\r')

self.ser.write('StSample\r')
self.statuslist.insert('end', 'Starting sampling')

count=0
i=0

#Calling the sampling function,
#here I also sendt with an parameter 1
#this paramterer tells the sampling
#function to calulate the impedance phase.
for i in range(0, int(self.number_increments)):
    self.savedata(i, 1)

#Open file out.txt for writing
ofile=open('out.txt', 'w')
ofile.write('Freq Phase Impedance\n')
#Loops through all the results and write the frequency,
#phase and impedance to file.
for k in range(0, len(self.data['Freq'])):
    str1='%d  %g  %d\n' %(self.data['Freq'][k],\\
    #self.data['Phase2'][k], self.data['Impedance'][k])
    ofile.write(str1)
```

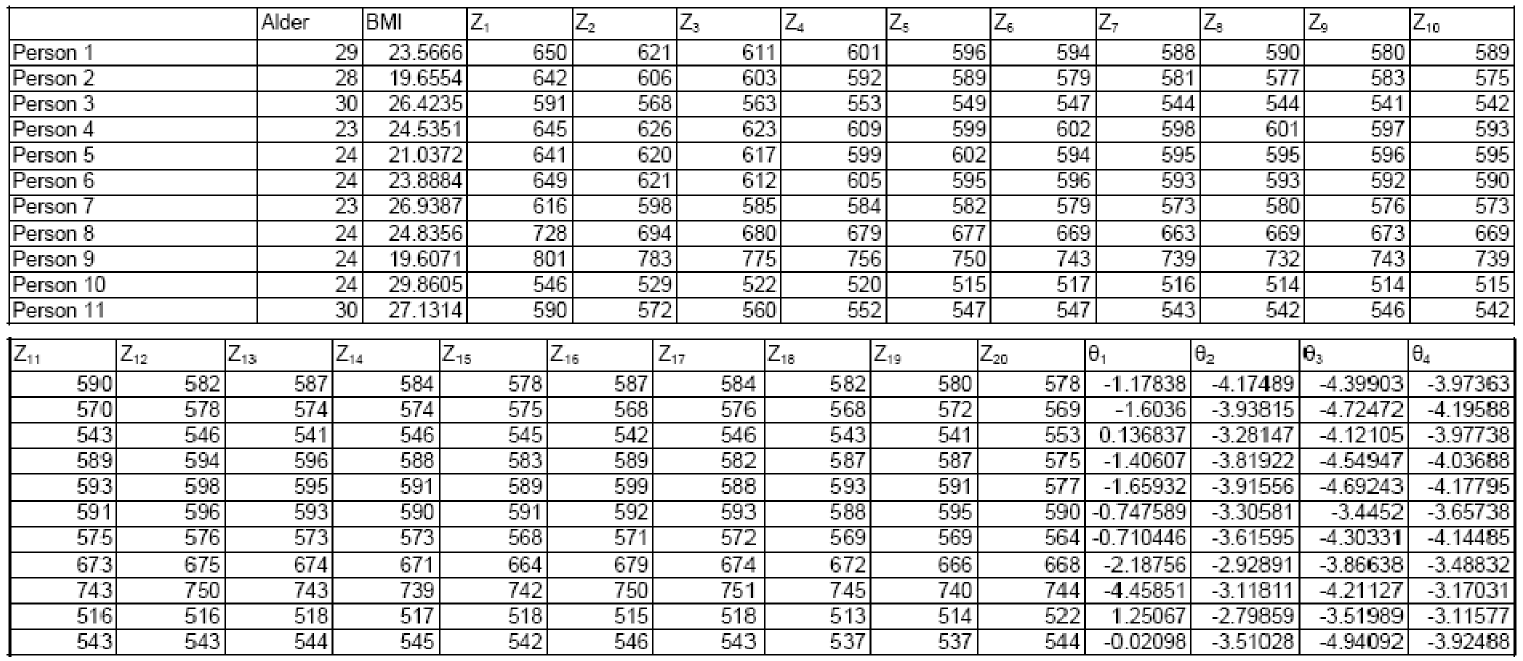
```
        ofile.close()

    #Function for plotting impedance
    def plot_imp(self):
        pylab.plot([self.data['Freq']],[self.data['Impedance']], 'ro')
        pylab.xlabel('Frequency (Hz)')
        pylab.ylabel('Impedance (Ohm)')
        pylab.savefig('impedance.png')
        pylab.show()

    #Function for plotting phase
    def plot_phase(self):
        pylab.plot([self.data['Freq']],[self.data['Phase2']], 'ro')
        pylab.xlabel('Frequency (Hz)')
        pylab.ylabel('Phase (degree)')
        pylab.savefig('phase.png')
        pylab.show()

    #Function for terminating the parent frame (this closes the interface)
    def quit(self):
        self.parent.quit()

    #Sampling function that receives measured data and calulates results.
    def savedata(self, count, sel=0):
        i=0

        #Reads the value that is stored in the real register in the AD5933
        real=self.ser.readline()
        self.data['Real'].append(int(real))

        #Reads the value that is stored in the imaginary
        #register in the AD5933
        im=self.ser.readline()
        self.data['Im'].append(int(im))

        #Calculates the current frequency
        #by adding the increment frequncy
        #multiplied with the number of current
        #incrementation to the start frequency
        freq=int(self.startfreq.get())+int(self.numbinc.get())*count
        self.data['Freq'].append(int(freq))
        #Calculates the magnitude from
        #the received real and imaginary parts.
        magn=math.sqrt(float(real)**2 +float(im)**2)
        self.data['Mag'].append(magn)

        #Calculates and stores the impedance from the
        #magnitude and gain factor.
        self.data['Impedance'].append(\\
        float(1/(magn*float(self.gainfactor.get()))))

        #Calculates the phase and stores it
        phase2=math.atan((float(im)/float(real)))*57.2957795
        self.data['Phase'].append(phase2)

        #Routine for getting the correct
        #system phase for the current frequency.
        if sel!=0:
            while 1:
                #If the current frequency is higher
                #than the frequency for the calibration data
                #the counter is incremented
                if freq>int(self.cal['Freq'][self.k]):
                    self.k=self.k+1
                #else if the freq is the same then
                #the phase is calculated
                elif freq==int(self.cal['Freq'][self.k]):
                    phase=(phase2)-float(self.cal['Phase'][self.k])
                    #Over the whole frequency span the
                    #phase might change the quadrant
                    #and to get the results to the 4th quadrant
                    #I subtract 180 degrees from
                    #the results that are larger than 30.
                    if phase>30:
                        phase=phase-180
                    break
            self.data['Phase2'].append(phase)

#root is main window where the other frames will be placed
root=Tk()
#Initialises the pmw megawidget pack
```

```
Pmw.initialise(root)
#Lager en ny instance A av klassen AD5933GUI
A=AD5933GUI(root)
#Loop for running the graphical interface
root.mainloop()
```

Appendix C

Table from Excel with all the results

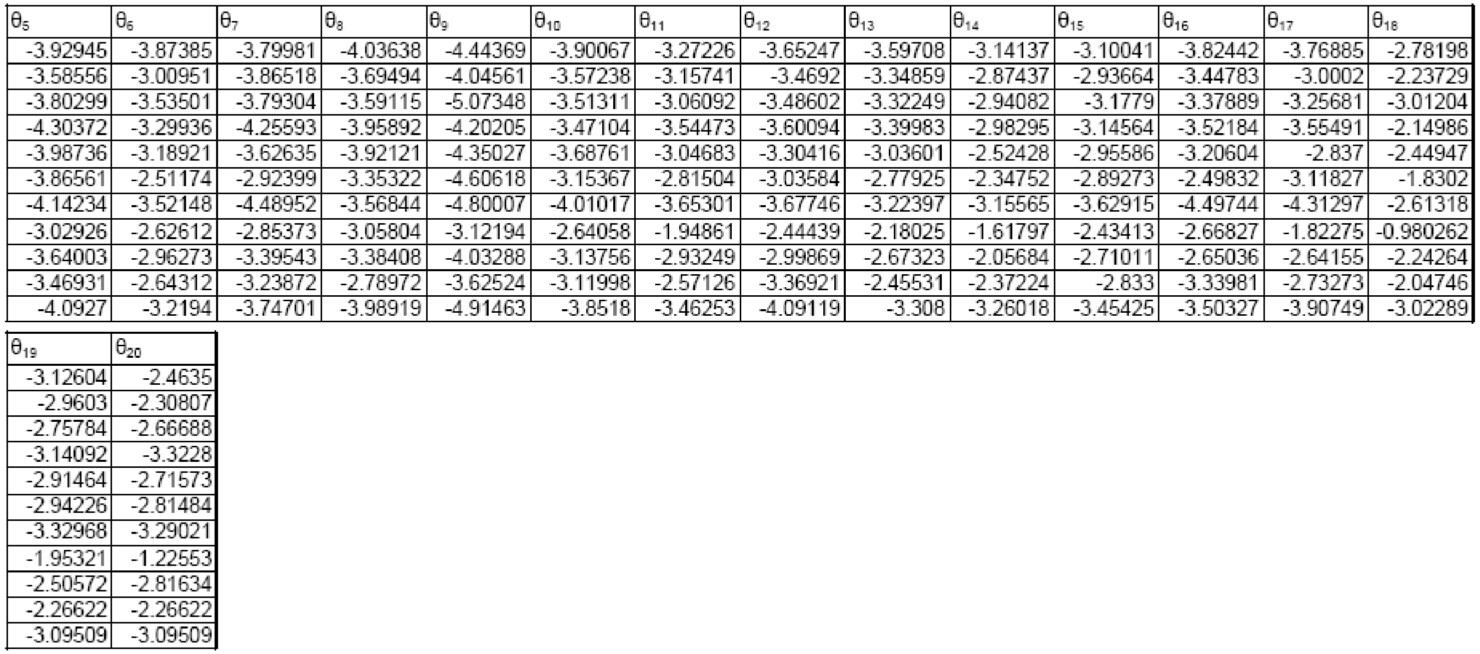
C.1 Table of results



	Alder	BMI	Z_1	Z_2	Z_3	Z_4	Z_5	Z_6	Z_7	Z_8	Z_9	Z_{10}
Person 1	29	23.5666	650	621	611	601	596	594	588	590	580	589
Person 2	28	19.6554	642	606	603	592	589	579	581	577	583	575
Person 3	30	26.4235	591	568	563	553	549	547	544	544	541	542
Person 4	23	24.5351	645	626	623	609	599	602	598	601	597	593
Person 5	24	21.0372	641	620	617	599	602	594	595	595	596	595
Person 6	24	23.8884	649	621	612	605	595	596	593	593	592	590
Person 7	23	26.9387	616	598	585	584	582	579	573	580	576	573
Person 8	24	24.8356	728	694	680	679	677	669	663	669	673	669
Person 9	24	19.6071	801	783	775	756	750	743	739	732	743	739
Person 10	24	29.8605	546	529	522	520	515	517	516	514	514	515
Person 11	30	27.1314	590	572	560	552	547	547	543	542	546	542

Z_{11}	Z_{12}	Z_{13}	Z_{14}	Z_{15}	Z_{16}	Z_{17}	Z_{18}	Z_{19}	Z_{20}	θ_1	θ_2	θ_3	θ_4
590	582	587	584	578	587	584	582	580	578	-1.17838	-4.17489	-4.39903	-3.97363
570	578	574	574	575	568	576	568	572	569	-1.6036	-3.93815	-4.72472	-4.19588
543	546	541	546	545	542	546	543	541	553	0.136837	-3.28147	-4.12105	-3.97738
589	594	596	588	583	589	582	587	587	575	-1.40607	-3.81922	-4.54947	-4.03688
593	598	595	591	589	599	588	593	591	577	-1.65932	-3.91556	-4.69243	-4.17795
591	596	593	590	591	592	593	588	595	590	-0.747589	-3.30581	-3.4452	-3.65738
575	576	573	573	568	571	572	569	569	564	-0.710446	-3.61595	-4.30331	-4.14485
673	675	674	671	664	679	674	672	666	668	-2.18756	-2.92891	-3.86638	-3.48832
743	750	743	739	742	750	751	745	740	744	-4.45851	-3.11811	-4.21127	-3.17031
516	516	518	517	518	515	518	513	514	522	1.25067	-2.79859	-3.51989	-3.11577
543	543	544	545	542	546	543	537	537	544	-0.02098	-3.51028	-4.94092	-3.92488

Figure C.1: Part 1/2 of the table with all the measurements results



θ_5	θ_6	θ_7	θ_8	θ_9	θ_{10}	θ_{11}	θ_{12}	θ_{13}	θ_{14}	θ_{15}	θ_{16}	θ_{17}	θ_{18}
-3.92945	-3.87385	-3.79981	-4.03638	-4.44369	-3.90067	-3.27226	-3.65247	-3.59708	-3.14137	-3.10041	-3.82442	-3.76885	-2.78198
-3.58556	-3.00951	-3.86518	-3.69494	-4.04561	-3.57238	-3.15741	-3.4692	-3.34859	-2.87437	-2.93664	-3.44783	-3.0002	-2.23729
-3.80299	-3.53501	-3.79304	-3.59115	-5.07348	-3.51311	-3.06092	-3.48602	-3.32249	-2.94082	-3.1779	-3.37889	-3.25681	-3.01204
-4.30372	-3.29936	-4.25593	-3.95892	-4.20205	-3.47104	-3.54473	-3.60094	-3.39983	-2.98295	-3.14564	-3.52184	-3.55491	-2.14986
-3.98736	-3.18921	-3.62635	-3.92121	-4.35027	-3.68761	-3.04683	-3.30416	-3.03601	-2.52428	-2.95586	-3.20604	-2.837	-2.44947
-3.86561	-2.51174	-2.92399	-3.35322	-4.60618	-3.15367	-2.81504	-3.03584	-2.77925	-2.34752	-2.89273	-2.49832	-3.11827	-1.8302
-4.14234	-3.52148	-4.48952	-3.56844	-4.80007	-4.01017	-3.65301	-3.67746	-3.22397	-3.15565	-3.62915	-4.49744	-4.31297	-2.61318
-3.02926	-2.62612	-2.85373	-3.05804	-3.12194	-2.64058	-1.94861	-2.44439	-2.18025	-1.61797	-2.43413	-2.66827	-1.82275	-0.980262
-3.64003	-2.96273	-3.39543	-3.38408	-4.03288	-3.13756	-2.93249	-2.99869	-2.67323	-2.05684	-2.71011	-2.65036	-2.64155	-2.24264
-3.46931	-2.64312	-3.23872	-2.78972	-3.62524	-3.11998	-2.57126	-3.36921	-2.45531	-2.37224	-2.833	-3.33981	-2.73273	-2.04746
-4.0927	-3.2194	-3.74701	-3.98919	-4.91463	-3.8518	-3.46253	-4.09119	-3.308	-3.26018	-3.45425	-3.50327	-3.90749	-3.02289

θ_{19}	θ_{20}
-3.12604	-2.4635
-2.9603	-2.30807
-2.75784	-2.66688
-3.14092	-3.3228
-2.91464	-2.71573
-2.94226	-2.81484
-3.32968	-3.29021
-1.95321	-1.22553
-2.50572	-2.81634
-2.26622	-2.26622
-3.09509	-3.09509

Figure C.2: Part 2/2 of the table with all the measurements results

Appendix D

Flow diagram for software

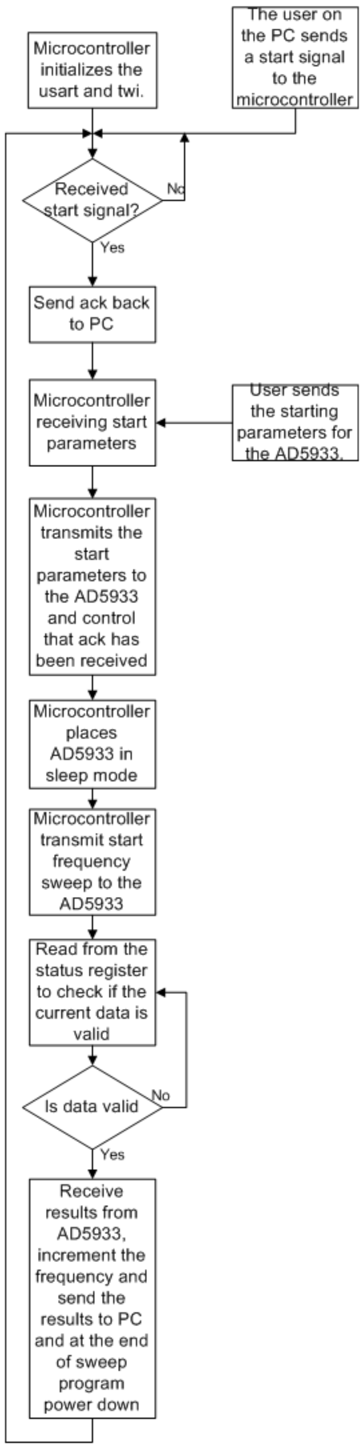

Microcontroller initializes the usart and twi.
The user on the PC sends a start signal to the microcontroller
Received start signal?
No
Yes
Send ack back to PC
Microcontroller receiving start parameters
User sends the starting parameters for the AD5933.
Microcontroller transmits the start parameters to the AD5933 and control that ack has been received
Microcontroller places AD5933 in sleep mode
Microcontroller transmit start frequency sweep to the AD5933
Read from the status register to check if the current data is valid
Is data valid
No
Yes
Receive results from AD5933, increment the frequency and send the results to PC and at the end of sweep program power down

Figure D.1: Flow chart for the developed software