NMR-Studies of multi component solids

Author:
Frederik Klama

Supervisor:
Dr. Nigel Clayden

M.Sc. by Research

University of East Anglia
School of Chemistry and Pharmacy

November 2010

Abstract: Particle size determination through relaxation time measurements by Inversion Recovery, Saturation Recovery and Spin-lock measurements using Etravirine and Felodipin as samples. ^{31}P and ^{13}C CP-MAS studies of InP nanoparticles using cross-polarisation dynamics to help with structure determination. TLM diffusion simulation as a method to simulate spin-diffusion systems and comparison of this simulation system to an analytical solution and measured data of such systems. Spin-diffusion Experiments combining spin- and solid-echo with spin-diffusion pulse sequences.

Contents

Acknowledgment

I would like to thank my supervisor *Dr. Nigel J. Clayden* for taking me in as a Master student, for helping me with my thesis and administrative difficulties encountered during this time; my mother *Manuela Klama* for finding the opportunity to do the Master by Research in Norwich and my girlfriend *Sandra Kröger* for encouraging me, helping me through difficult parts of writing this thesis and for proofreading.

I would also like to thank the following people for their various contributions to this thesis and the work it is about: *Simona Chessa*, *Sheng Qi*, *Shu Xu*, *Prof. Dr. Claudia Schmidt*.

Chapter 1

Introduction

As part of this thesis several different experiments were performed for several different groups within the Department of Chemistry and Pharmacy. Different NMR Methods were applied to these tasks.

All NMR Measurements were taken at the University of East Anglia using a Bruker NMR spectrometer with a proton frequency of 300 MHz using a 4 mm double resonance MAS probe. The probes proton channel is capable of frequencies between 289.67 MHz and 309.333 MHz whereas the second channel has three settings each with a different frequency range and intended nucleus. The frequency range of the *low*-setting is 28.813 MHz – 37.733 MHz with ^{15}N as the intended nucleus. The frequency range of the *medium*-setting is 41.270 MHz – 107.493 MHz with ^{13}C as the intended nucleus. Lastly the frequency range of the *high*-setting is 51.58 MHz – 134.76 MHz with ^{31}P as the intended nucleus.

As part of this thesis the author worked on three different projects. The author worked with InP nanoparticles prepared by *Shu Xu* to see if insights into the makeup of the particles could be gained by using NMR Techniques. He also worked with *Sheng Qi* to use NMR to determine the size of crystallites in pharmaceutical samples. The third project discussed in this work is the use of Transmission-Line-Matrix Modeling to simulate spin-diffusion which was done alone with some help by Dr. Clayden.

In the course of this thesis several different NMR Techniques were used. ^{13}C and ^{31}P *CP-MAS* [1], which is actually the combination of two techniques, *Cross Polarization* and *Magic-Angle-Spinning*.

Magic-Angle-Spinning is a technique where the sample is spun at a specific angle to the magnetic field to reduce the problems due to molecular orientation by averaging the orientation of the molecules over time. However spinning a solid sample is usually only possible when it is in the form of a powder since even a minute unbalance in the rotor will lead to tumbling.

Cross-Polarization is used to overcome the low abundance and receptivity of certain nuclei (such as ^{13}C by exiting an abundant and highly receptive nuclei (such as protons) and then transferring the magnetization to the nuclei to be observed. This is done by sending two pulses simultaneously to the frequencies of the two nuclei and setting the amplitude of these pulses so that the Hartmann-

Hahn condition is met. This techniques has the advantage of a shorter delay between experiments, a stronger signal to better magnetization and thus faster experiment times. However since the amount of magnetization transferred is proportional to the distance to the excited nucleus, peaks from nuclei that are further away from the excited nuclei are weaker. This effect can be used to estimate the average distance of nuclei to the protons in the sample for example.

The echo pulse sequences *Spin-Echo* and *Solid-Echo* [1] were used to reduce the effects of dipole-dipole as well as quadrupole couplings and to be able to record a whole FID if part of it has been cut off by the receiver dead time.

Inversion- [2] and *Saturation-Recovery* were used to measure T_1 times and *Spin-Lock* experiments [2] were used to measure $T_{1\rho}$-times. Both the T_1- and $T_{1\rho}$-times were used to determine crystallite sizes in pharmaceutical samples.

Spin-Diffusion experiments were conducted to try and measure the diffusion of magnetization from one polymer phase into another. However either because the sample behaves slightly different in a $300\,MHz$-Field than in a $200\,MHz$-Field, or because the sample was degraded when the measurements were made, no quantifiable spin-diffusion could be observed by the author.

Several ^{13}C and ^{31}P cross-polarization MAS-NMR spectra, as well as regular ^{31}P (with and without decoupling) were recorded in order to help determine the structure of several samples of InP nanoparticles.

T_1 and $T_{1\rho}$ proton measurements were made of the preprepared hot melt extrusions of Felodipin and the polymer Eudragit. The measurements were used to get some idea of the phases present in the hot melt extrusions.

As part of the spin-diffusion simulation a model usually used to describe diffusion of electrical charge in resistor-capacitor networks is used to simulate spin-diffusion [3]. To do this, a simulation software was written by the author to facilitate the simulation of spin diffusion using the *Transmission-Line-Matrix-* or in short *TLM-Model*.

It was tried (although not very successfully) to get some spin-diffusion data using polymer samples, provided by Dr. Clayden, to be able to compare the simulations to real life data. However the separation of the signal of the polymer into two signals made this task very difficult and although it can be concluded from the spectra that some form of spin diffusion took place, it was not possible to prepare the spectra for numerical analysis. Some old spin-diffusion data was provided by Dr. Clayden to compare the simulations against.

Since only data from one spin-diffusion experiment was available for comparison, the simulations were compared to an analytical solution[4] of a spin diffusion system.

Chapter 2

NMR Methods

2.1 Measuring methods for T_1 and $T_{1\rho}$

2.1.1 Spin relaxation and domain size

Spin-lattice relaxation time

The nuclei of the sample are held within a lattice structure, and are in constant vibrational and rotational motion. The complex magnetic field cause by this thermal motion is called the *lattice field*. When the lattice fields, of two nuclei in different energy states, interact, the energy is distributed among them. This effect causes the energy, which was gained from the RF pulse, to be dissipated as increased rotation and vibration increasing the temperature of the sample. The T_1 time is a measure of how fast this process occurs.

The longitudal component of the magnetization vector recovers towards the equilibrium state as a result of the dissipation of the magnetic energy. T_1 describes the time it takes for the FID signal to recover to $1 - e^{-1}$ ($\approx 63.21\%$) of its maximum value. The longitudal component of the magnetization vector is described in equation 2.1.

$$M_z(t) = M_{z,\mathrm{eq}}(1 - e^{-\frac{t}{T_1}}) \tag{2.1}$$

The value of T_1 is dependent on the gyromagnetic ratio γ of the nucleus in question and the mobility of the lattice. As the mobility of the lattice increases, the value of T_1 decreases[1]. The high values of T_1 sometimes encountered in solid-state NMR can be a problem, due to the long time it takes for the sample to return to the equilibrium magnetization.

Whereas in liquids cross magnetization can lead to nuclear Overhauser enhancements (NOEs) or magnetization transfers, this plays hardly any role in solids since it masked by proton spin-diffusion which is a much more efficient energy transfer method among protons in solids[5].

In heterogeneous systems one can use the relaxation times of the different phases to estimate their size[6][7][8]. If one assumes that the process of exchange

[1] Except for highly mobile samples where the T_1 time can actually increase with mobility

does not occur by matter diffusion, (a valid assumption in solid state) but only by spin diffusion, the following equation is valid[8]:

$$2\sqrt{2}\frac{\mathcal{A}^2}{\pi^2\mathcal{D}}|\Delta\gamma| > 1 \tag{2.2}$$

$\mathcal{A}$ represents the smallest dimension over which diffusion takes place, $\mathcal{D}$ represents the spin diffusion coefficient and $|\Delta\gamma|$ is calculated using the relaxation rates of the separated phases:

$$\Delta\gamma = \frac{1}{T_A} - \frac{1}{T_B} \tag{2.3}$$

Spin-spin relaxation time

The spin-spin relaxation time, or T_2, is a measure of how fast the magnitude of the transversal part M_{xy} of the magnetization vector decays. It is the time it takes for M_{xy} to reach $\frac{1}{e}$ ($\approx 36.79\%$) of its initial value after flipping into the transversal plane. Equation 2.4 describes the relaxation of M_{xy} over time. When dealing with liquids or solids with some small internal molecular motions, we use the following equation to describe T_2

$$M_{xy}(t) = M_{xy}(0)e^{-\frac{t}{T_2}} \tag{2.4}$$

T_2 is generally faster than T_1 relaxation. It corresponds to the decoherence of the transverse nuclear spin magnetization. Since the local magnetic field is not constant throughout the sample, the instantaneous precession frequency of the spins differs slightly. Thus the initial phase coherence of the nuclear spins is lost and eventually the phases shift so much that there is no more net xy-magnetization.

This effect is reduced by high mobility, due to the fact that the environment of each single nucleus changes quickly as it moves randomly through the sample. Thus the nuclei have a very similar average precession rate over time. In the case of a solid, like used for this work, the nuclei are unable to move and thus dephase much more quickly. The fact that the precession frequency is already different for differently oriented crystallites makes transverse relaxation a much bigger limiting factor in solid-state NMR than it is in liquid-state NMR. To overcome these problems, which are typical for solid-state NMR, refocusing echo pulse sequences can be used. This is discussed in more detail in chapter 2.2.1.

Equation 2.4 does not apply in such crystalline solids. In a solid the relaxation is not defined by a single exponential process T_2. Various functional forms can be used to describe the lineshape and hence the T_2 time. In solids the following equations are most commonly used:

$$\text{Gaussian function:} \qquad M(t) = e^{-\frac{1}{2}\tau^2 t^2} \tag{2.5}$$

$$\text{Abragam function:} \qquad M(t) = \frac{\sin(\theta t)}{\theta t}\, e^{-\frac{1}{2}\tau^2 t^2} \tag{2.6}$$

$$\text{Pakes doublet:} \qquad M(t) = \cos(\theta t)\, e^{-\frac{1}{2}\tau^2 t^2} \tag{2.7}$$

It is obvious that although the decay itself is not exponential anymore, it can still be defined using a single time constant. These equations are used in chapter 5 to describe the FIDs generated by the spin-diffusion experiments.

Effects of field inhomogeneities

In an idealized system the field B_0 is perfectly homogeneous. In reality however the magnetic field is never perfect. By shimming the magnet it is possible to compensate for inhomogeneities to a certain degree, but it is never possible to get a perfectly homogeneous field all over the sample. This effect adds to the fluctuations in the local field for the spins resulting in even stronger dephasing. The relaxation time taking this into account as well is T_2^* and is usually significantly larger than T_2. The relation between T_2^* and T_2 is described in equation 2.8.

$$\frac{1}{T_2^*} = \frac{1}{T_2} + \frac{1}{T_{\text{inhom}}} = \frac{1}{T_2} + \gamma \Delta B_0 \tag{2.8}$$

MAS compensates for this partially by physically rotating the whole sample and thus averaging the magnetic field inhomogeneities over space. Since the local environment of the nuclei is not changed though, the T_2 values in liquid-state NMR are still higher.

Spin Temperature

In a crystal without any net motion, there is a tight coupling between the nuclear spins. This means that the whole system of spins has to be taken into account. To do this, one usually considers the spin temperature T_S which is defined as:

$$\frac{p_+}{p_-} = \exp\left(+\frac{\gamma \hbar H}{2kT_S}\right) \text{ [9]} \tag{2.9}$$

The relationship between spin temperature and magnetization is easily described as

$$M_Z = \frac{CH}{T_S} \tag{2.10}$$

$$M_Z = N\gamma\hbar \sum_m p_m m \text{ [9]} \tag{2.11}$$

in which we sum over all energy levels, which are denoted by m.

When introducing an RF-pulse to this system, we can see that after a 90°-pulse, the spin temperature is infinite. We can even reach negative temperatures when we apply a 180°-pulse. Note however that a negative spin-temperature is actually *hotter* than an infinite spin-temperature.

One can define T_1 in terms of the time it takes for the spin-temperature to cool down to the lattice temperature:

$$\frac{\mathrm{d}M_Z}{\mathrm{d}t} = \frac{M_0 - M_Z}{T_1} \text{ [9]} \tag{2.12}$$

where M_0 denotes the spin-temperature at equilibrium.

However directly after an RF-pulse, the system has transverse magnetization, which are incompatible with the description of the spin system by a spin-temperature. While these transverse components of the magnetization exist, this description of spin-temperature does not properly describe the state the system is in.

The lifetime of this transverse magnetization is loosely defined at T_2.

Relaxation in the rotating frame

Whereas T_1 and T_2 are relaxations in the laboratory frame when only the field B_0 is present. There is another relaxation when considering the field B_1 which is present during the RF-pulse.

The spin-lattice relaxation time in the rotating frame, or $T_{1\rho}$, is the time the longitudal magnetization relative to the B_1 field takes to relax. Small values of $T_{1\rho}$ are very problematic, since $T_{1\rho}$ always has to be larger than the length of the pulse. Otherwise the pulse is ineffective, because the magnetization has decayed before it is turned into the xy-plane.

2.1.2 Inversion Recovery

There are actually quite a few methods to measure T_1. One of the simplest to understand is the inversion recovery method [2, ch 12.1]. Here the time it takes to recover from a full spin inversion is measured. To do this a 180°-pulse is applied, and after an increasing delay a 90°-pulse is used to acquire the signal. A visual representation of the pulse program can be seen in figure 2.1.

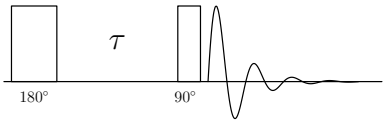


Figure 2.1: Inversion Recovery Pulse Sequence

To estimate T_1 from the data generated by this kind of experiment, it is necessary to make several measurements with different delays. The amplitude of the signal is a function of the delay τ.

$$a(\tau) = \frac{1}{2}\mathbb{B}\left(1 - 2e^{-\frac{\tau}{T_1}}\right) \tag{2.13}$$

Since all we are interested in is T_1, it is not absolutely necessary to know the factor $\mathbb{B}$ to determine T_1. Because only the amplitude of the signal is described by equation 2.13 it is dimensionless.

$$\mathbb{B} = \frac{\hbar\gamma B^0}{k_B T} \tag{2.14}$$

For a regular NMR experiment, $\mathbb{B}$ is a constant factor called the Boltzmann factor that is described by equation 2.14.

The acquired signal is fitted using equation 2.13, which then yields T_1. It should be noted that equation 2.15 can be used to quickly estimate the value of T_1 for simple systems.

$$\tau \approx T_1 \ln 2 \quad \text{for} \quad S = 0 \tag{2.15}$$

This can be useful to get a quick estimate of T_1 using a calculator. It is inherently inaccurate since it only uses a single data point and thus does not benefit of averaging out signal noise like a fit using several data points does.

The only problem with the inversion-recovery method is that a sufficiently long recycle delay has to be chosen to make sure that the sample is fully recovered before the next step of the experiment. This may lead to very long experiment times when the value of T_1 is expected to be large, or unusable data when the recycle delay chosen is too small.

2.1.3 Saturation Recovery

Another, quicker, method to determine T_1-times is *saturation-recovery*. In this sequence multiple 90° pulses are applied with decreasing delays between the pulses to dephase and saturate the spins. A pulse program for such an experiment might look like the example in Figure 2.2. Nevertheless care must be taken since a too regular pulse sequence often does not properly saturate the spins.

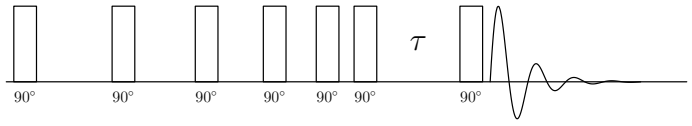


Figure 2.2: Alternate gradient free Saturation Recovery Pulse Sequence

To calculate the T_1-value from data recorded by saturation recovery, a slightly different function must be used to fit the data:

$$S = A \cdot e^{-\frac{\tau}{T_1}} \tag{2.16}$$

Where the factor A represents the maximum amplitude.

2.1.4 $T_{1\rho}$ Spin-lock

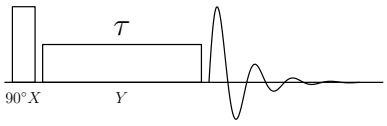


Figure 2.3: $T_{1\rho}$ Pulse Sequence

To determine the $T_{1\rho}$ [10], a pulse sequence in which the spins are spin-locked for a variable time (τ), is used. The spin-lock is achieved using a low power pulse phase shifted by 90° against the exciting pulse. So when the spins are excited by

a pulse along the x-plane (as in Figure 2.3), a low power pulse along the y-plane is used to spin-lock them.

It is also possible to use a spin-echo pulse sequence to determine $T_{1\rho}$ [2, ch 12.2]. However all measurements of $T_{1\rho}$ in this work were done using the spin-lock method [2, ch 12.3].

To fit this data to determine $T_{1\rho}$, a function similar to Equation 2.16 is used:

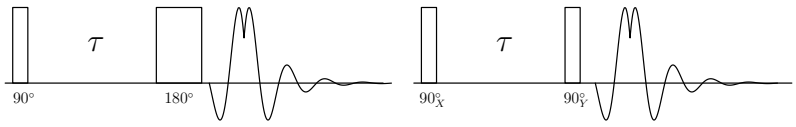
$$S = A \cdot e^{-\frac{\tau}{T_{1\rho}}} \tag{2.17}$$

2.2 Echo and Spin Diffusion Pulse Sequences

2.2.1 Echo Pulse Sequences

In solid state NMR broad signals are encountered quite often. Since the spins of such broad signals dephase very quickly, there is only a short time for observation. In these cases, the dead time of the receiver can become a problem. However an echo pulse sequence can be used to rephase the spins to overcome this problem.

The dead time is necessary, since the power of a pulse is approximately 10^9 to 10^{12} times stronger than the signal being observed. The pulse, or the residual signal from the pulse ringing down in the coil, could easily destroy the sensitive receiver circuits. Usually the receiver circuit is only put onto the line a short time (about $1\mu s$) after the last pulse.



(a) Spin Echo (b) Solid Echo

Figure 2.4: Echo sequences

The solution to the problem of short FID, due to broad spectral lines, is to use an echo pulse sequence [1, pp110–113]. One such echo pulse sequence is the spin-echo (or hahn-echo) which consists of a simple 90° pulse followed by a 180° pulse (see Figure 2.4-a). The two pulses are separated by a time τ[2]. After another time τ coming after the average time of the 180° pulse, the normal FID starts. But in the time between a mirror image of the normal FID can be seen.

The spin-echo sequence is most useful where spectral line broadening is due to chemical shift anisotropy or heteronuclear dipole-dipole coupling. Another echo sequence, called the solid-echo sequence (or quadrupole-echo) differs only in the length of the second pulse. The pulse length is shorter, so that it is a 90° pulse instead a 180° pulse (see Figure 2.4-b). Here the phase of the second pulse, which has to be perpendicular to the first, is more important than in the spin-echo sequence. The solid-echo sequence is most useful where spectral line broadening is due to quadrupole coupling or homonuclear dipole-dipole coupling.

[2]Notice that the middle of the pulse, not the edges, are used as a time reference. This is the average time of the pulse

2.2.2 Spin Diffusion Pulse Sequence

An ideal spin diffusion sequence would just excite the spins of one phase of the sample. However this is usually not possible. Therefore a way to excite just one of the phases is needed. In this work, the difference in relaxation time of the two phases is exploited to select one of the phases.

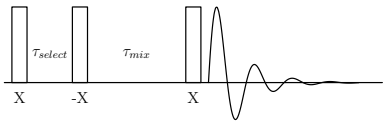


Figure 2.5: Simple Spin Diffusion Pulse Sequence

In the real world it is not possible to directly excite one phase of the sample while leaving the other untouched. However, one of the phases usually decays significantly faster than the other. With a little timing it is possible to have one of the phases decayed close to equilibrium, while the other still has significant magnetization. The pulse sequence to do this is called the Goldman-Shen pulse sequence [11].

The aim of the experiment was do characterize the size of the domains by measuring the spin diffusion [12]. To do this a spin diffusion pulse sequence (as depicted in Figure 2.5) was used. After initial magnetization, a selection delay τ_{select}, which allows the phase with the shorter T_1-time to decay, the magnetization is stored along the z-axis using a 90° pulse (refer to Figure 2.5). After a mixing time τ_{mix}, during which spin diffusion is allowed to occur, the magnetization is brought back into the detection plane by another 90° pulse.

The experiment is repeated several times with increasing τ_{mix}. The short lived phase should reappear with increasing strength as τ_{mix} increases.

2.2.3 Pulse Sequence for Spin Diffusion with Spin-Echo

When dealing with solid samples, the spin diffusion pulse sequence can be improved by introducing a spin-echo at the end of the sequence (see Figure 2.6). The added pulse refocuses the magnetization and creates an echo of the original FID which was partially obscured by the receiver dead-time, thus enabling the operator to record the full FID. Using this technique the amplitudes, calculated by the fitting of the decay functions to the data, will be much more accurate.

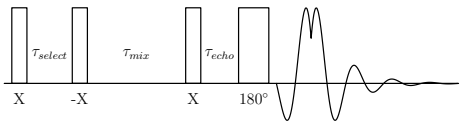


Figure 2.6: Spin diffusion pulse sequence with spin-echo

2.3 High resolution Solid-Sate NMR

Compared to NMR with liquids, solid-state NMR is faced with many problems that don't even exist when dealing with liquid samples. Most of these problems arise from the fact that most samples are analyzed in a powdered form. The crystallites that make up the powder are oriented randomly over space. Since chemical shift, dipole-dipole coupling and quadrupole coupling are dependent on crystallite orientation, a typical powder spectrum is the result. A powder spectrum usually consists of very weak and very broad signals. Since the signals are usually so broad that they overlap, powder spectra are usually very difficult or just about impossible to interpret.

This is why several methods have to be used to overcome these problems.

2.3.1 Magic Angle Spinning

Magic Angle Spinning (or MAS) is a method in which the powdered sample is spun at high speed on an axis with an angle of $\sim 54.74°$ to the main magnetic field. It reduces the effects described earlier, since the orientation of the crystallites are effectively averaged if the sample is spun fast enough.

In liquid phase the molecular motion, which is very fast in an NMR time scale, effectively averages the molecular orientation over time. Thus every nucleus is equivalent to one in the same position in every molecule in the sample and therefore resonates at the same frequency.

The dipolar coupling in a strong magnetic field depends on the angle of the inter-nuclear vector to the magnetic field as described in formula 2.18.

$$D \propto 3\cos^2\theta - 1 \tag{2.18}$$

This means that D will become zero when $\cos^2\theta = \frac{1}{3}$, which occurs exactly at a value of $\theta = \cos^{-1}\frac{1}{\sqrt{3}} \simeq 54.74°$.

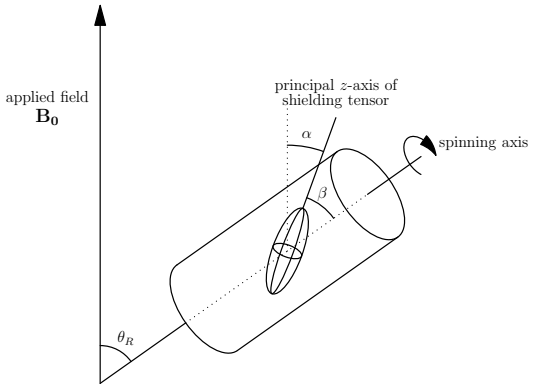


Figure 2.7: MAS coordinate space [1, p61]

Referring to figure 2.7 we see that, for the shielding tensor at an angle β to the rotor axis, being rotated at an angle θ_R relative to a magnetic field B_0, the angle of the shielding tensor relative to the magnetic field (α) will vary between $\theta_R - \beta$

and $\theta_R + \beta$. Over time this averages as θ_R. This means that when the rotation speed is fast relative to the time scales of the experiment, the broadening effect of dipolar coupling is effectively eliminated.

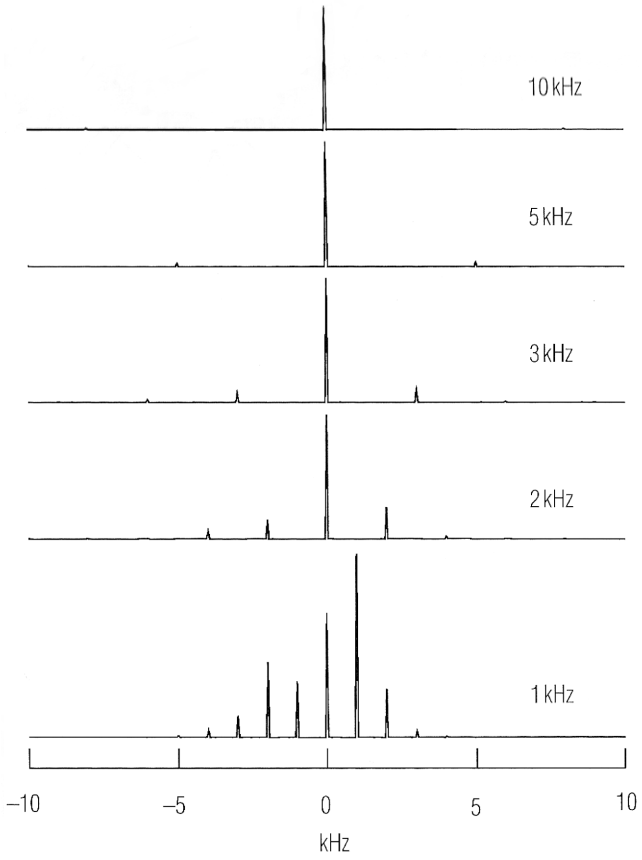


Figure 2.8: Rotational side-bands at different rotation speeds [1, p63]

However usually it is not possible to spin fast enough for a complete averaging of the shielding tensors. The result are spinning side bands. Spinning side bands can occur at $f_{peak} \pm n \cdot f_{rotation}$, and are generally weaker the higher the rotation frequency of the rotor is. Examples of spinning side bands at different rotational frequencies can be seen in figure 2.8. One has to remember that the strongest signal is not necessarily the main signal, but may be a spinning side band. The only sure way to discriminate between the signal and its spinning side bands in a simple MAS spectrum is to compare different spectra recorded with different rotational speeds. The peaks that appear in both spectra are the real signals, while the spinning side bands will have shifted.

The TOSS (*To*tal *S*uppression of *Sidebands*) [1, pp67–72] can be used to suppress the sidebands. A series of precisely timed 180° pulses are applied prior to acquisition. These 180° pulses effectively randomizes the phases of the sideband magnetization from the different crystallite orientations removing or at least reducing the amplitude of the spinning-sidebands.

A very elegant method of sideband removal is the method known as 2D-PASS (*P*hase *A*djusted *S*pinning *S*idebands) [1, pp143–145]. After initial magnetization

via a 90°-pulse, a set of five 180°-pulses is used to select the order of the spinning-sideband. The set of five 180°-pulses is changed in such a way that for each iteration into the second dimension a different order of side bands is selected. The result is a set of spectra each recording a different sideband order. The spectrum of the order 0 is the spectrum with the sidebands completely removed.

2.3.2 Cross polarization

When studying rare nuclei such as ^{13}C, the low natural abundance of the isotope results in two problems. The most obvious is that only a fraction of all carbon nuclei can be exited, since most of the carbon nuclei are ^{12}C, resulting in a weak signal. Secondly the relaxation time of a nucleus is inversely proportional to its concentration in the sample, resulting in long relaxation times. Both of these factors combined mean that many acquisitions are needed and the time between acquisitions has to be quite long. Thus acquiring a ^{13}C-Spectrum can take several hours or days when done by by direct excitation.

The solution to this problem is cross polarization (often abbreviated with CP). In a CP-Experiment one uses a highly abundant nucleus in the sample to excite the target nuclei. To do this the abundant nuclei (protons are often used) are exited directly using a 90°-pulse. Then a low power pulse is sent on the proton- and the carbon-channel. The power of the two pulses is selected in such a way that the transition energy between the two energy bands is the same for both nuclei (Hartmann-Hahn condition). This causes the energy to be transferred from the excited abundant nuclei to the target nuclei.

A typical pulse program to achieve this is shown in diagram 2.9. Notice that the phase of contact of the pulse is perpendicular to the initial excitation pulse. The proton decoupling during acquisition is not essential but is often added to remove the influence of proton coupling.

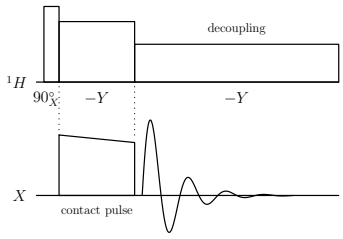


Figure 2.9: Cross polarization pulse program

At the start of the contact pulse (left hand side in diagram 2.10) the proton spins are excited and the carbon spins (marked by an X) are in equilibrium. The proton spins behave as expected and decay towards equilibrium. But since the Hartmann-Hahn condition is met, the energy is transferred onto the carbon spins which are excited.

The magnetic field applied during the contact time, $B_1(^1H)$ and $B_1(X)$ splits the spins into two energy levels (α^* and β^*) parallel to the B_1-fields. The amplitude of the two fields is selected in such a way that $\omega_1(^1H) = \omega_1(X)$. This

is called the Hartmann-Hahn condition and can be described by the following formula.

$$\gamma_H B_1(^1H) = \gamma_X B_1(X) \tag{2.19}$$

Since the values of γ_H and γ_X are known, it is possible to calculate the relation $\frac{B_1(^1H)}{B_1(X)}$. It is usually not possible to directly set the effective field strength due to signal loss in the cables and due to imperfect tuning. Thus the attenuation of the amplifiers for the two channels is usually set to the calculated ratio and adjusted until a signal maximum is reached. For carbon the sample used to adjust the signal attenuations to meet the Hartmann-Hahn condition is usually Adamantane. This highly symmetric molecule only has two different carbon and hydrogen environments and the molecules are able to rotate even in solid-state, thus even MAS is not needed.

At the beginning of the pulse sequence the ^{1}H-spins are rotated into the y-plane by the 90°-x-pulse. They are then held there by the $B_1(^1H)$-field. Since the B_0-field is much stronger than the $B_1(^1H)$-field, the magnetization slowly decays towards equilibrium. This means that the spins move from the α_H^* to the β_H^* energy state to equalize the population in both bands.

Since the Hartmann-Hahn condition is met, the energy gap between the α^* and β^* bands is the same for both nuclei. There is a dipolar coupling between the ^{1}H- and the X-nuclei which takes the usual form of the heteronuclear interaction:

$$\hat{H}_{HX} = -\sum_i d_i(3\cos^2\theta_i - 1)\hat{I}_{iz}^H \hat{S}_z^C. \tag{2.20}$$

Here d_i represents the dipolar coupling constant for the interaction between the ^{1}H- and X-spins. The operator is not affected by transformations in the doubly rotating coordinate system, since it only contains z-components. Therefore the dipole-dipole interaction can not change the overall energy of the system. This is because the energy of the system is defined by the energy levels which are split by the B_1 fields which lie in the x-y-plane.

Since energy as well as angular momentum are conserved, X-spins move from β_X^* to α_X^* while the ^{1}H-spins decay towards equilibrium from α_H^* to β_H^*. The end result of this process is the excitation of the X-spins into the x-y-plane.

A ramp on the X-pulse during contact time is often used to reduce the sensitivity of the pulse program against a slightly off Hartmann-Hahn condition. Thus the signal intensity of the spectrum is less susceptible to slight errors in the Hartmann-Hahn condition over the course of the experiment.

Since CP-experiments are sensitive to deviations of the rotation angle from the magic angle, it is recommended to adjust the magic angle using a KBr sample prior to an experiment.

To set the magic angle one measures ^{79}Br spectra of a KBr sample [1, ch 2.2.4]. If the acquisition frequency is set to the bromine resonance of the KBr, small peaks on top of the normal simple decaying FID signal. In the frequency domain spinning side-bands can be seen. To adjust the magic angle to the optimal setting, one tries to maximize the amount of peaks seen in the FID.

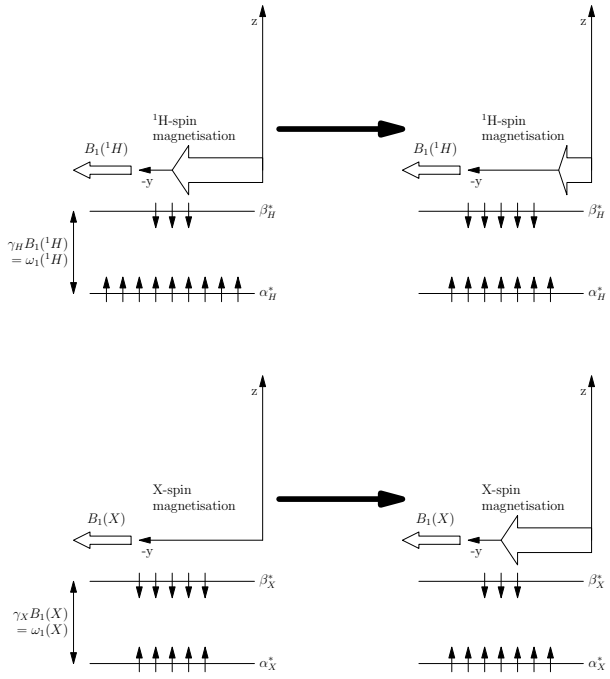


Figure 2.10: Spins during cross polarization [1, p99]

Chapter 3

InP Nanoparticles

3.1 Introduction

InP nanoparticles have physical and chemical properties highly dependent on their surface capping. We study the environment of the phosphorus nuclei using ^{31}P-NMR to determine the amount of surface defects.

^{31}P spectra using a simple 90° pulse are used, both with and without high power proton decoupling. ^{31}P cross polarization spectra using protons as the abundant nuclei in the organic coating of the particles were recorded. ^{13}C-CP spectra were also recorded.

The goal was to study the effect of zinc and different ligands on the physical properties of the InP nanoparticles.

The following chemicals were used in particles synthesis: Octadecene, Stearic acid, Hexadecylamine, Tris(trimethylsilyl)phosphine, zinc undecylenate, indium chloride, indium acetate, zinc diethyldithiocarbamate and cyclohexylisothiocyanate.

A common problem when synthesizing InP nanoparticles (especially those prepared in an indium rich environment), is that there are many dangling indium bonds on the surface of the particles dominating the optical properties. Synthesis in a phosphorus rich environment reduces these effects, however the size distribution of the particles gets broader. The zinc, which replaces excess indium atoms, reduces these surface defects nearly completely.

Please refer to the paper [13] written as part of this work on InP nanoparticles.

3.2 Samples

The InP particles were provided and prepared by Shu Xu. They were prepared in a wet chemical process with organic stabilizers used for the coordinating environment for crystal growth. Organic capping ligands consisting of stearic acid and hexadecylamine coat the particles, providing a proton rich shell which is used in the cross polarization experiments.

Some of the samples had zinc compounds added during the synthesis, with the aim of creating a shell of ZnS around a InP particle core.

The samples were analyzed using UV/Vis spectrometry, photoluminescence

spectrometry, mass spectrometry, Fourier-transform infrared spectroscopy and energy dispersive X-ray spectroscopy as well as ^{13}C and ^{31}P NMR spectrometry. Only the NMR-spectra were made by the author of this thesis.

The samples were dissolved in chloroform for the optical measurements and then dried in an oven before the NMR measurements. Since the amount of the samples was very small, the samples were mixed with zinc oxide before being placed in a rotor for MAS measurements. The rotor was prepacked with zinc oxide and after the sample was filled in, it was capped with more zinc oxide. This was done to make sure that the actual sample is concentrated in the center of the rotor, where the probe is most sensitive. zinc oxide was chosen, because neither natural zinc nor Oxygen isotopes interfere with ^{13}C-, ^{31}P- or proton-NMR. zinc oxide powder is also insoluble in most solvents, making extraction of the nanoparticles possible if needed. It also has the added benefit of being very safe to handle, since it is not toxic or poses any other hazard.

The tiny amounts of the samples that were available made it necessary to fill most of the rotor with zinc oxide powder to get the samples to spin. Since only about 5 – 30% of the rotor was filled with the sample, signal acquisition was made quite difficult due to the small amount of sample. For the carbon spectra this difficulty was made worse by the low natural abundance of ^{13}C. Cross-polarization as well as many acquisitions, often making an experiment last for 12 – 60 hours, made it possible to get a reasonable signal to noise ratio.

A detailed description of the synthesis and preperation of the InP-nanoparticles used in this chapter is detailed in the paper written by Shu Xu[13].

3.3 Results

3.3.1 Surface passivation with zinc carboxylates

The zinc carboxylates were chosen for several useful properties. Their long chains to support nucleation and growth reaction, a stable valence state so that they have weak oxidizing and reducing ability and their solubility in the solvents used and their low toxicity. A low affinity for lattice doping in InP and a low melting point, below the crystal growth temperature and weak reactivity with phosphorus, making a reaction with the phosphorus precursors difficult, especially under the conditions used in the synthesis of InP.

The addition of the zinc carboxylates significantly reduced the amount of defects measured spectrographically. Additionally a shift at the blue end of the spectrum was observed with increasing zinc concentration, indicating that the zinc carboxylates are preventing the reaction between the InP surfaces and free monomers in the solution. Thus high concentrations of zinc carboxylates result in stable capping layers and prevent crystal growth, while low concentrations only give incomplete surface capping of the particles surfaces but enable faster particle growth.

The best ratio of zinc compared to the indium concentration was dependent on the solvent used in the synthesis and can be 1:1 or 2:1.

3.3.2 Fatty acid concentration and its influence on the particles

The effect of the concentration of stearic acid was also investigated. The stearic acid was very effective at fostering nucleation, and caused rapid crystal growth for several seconds at the beginning of the process. Later in the synthesis the stearic acid served as a capping agent, slowing crystal growth.

The stearic acid acts as a protic agent, thus accelerating the release of H_3P, which causes the nucleation burst. However, since indium as well as phosphorus are sensitive to oxidizing agents, the fatty acids will react with the InP and oxidize the InP nanocrystals to amorphous In_2O_3 particles over time. Thus the concentration must be limited, so that all excess stearic acid can be consumed by the excess trimethylindium and $(TMS)_3P$.

3.3.3 Fatty amine concentration and its influence on the particles

Since hexadecylamine is less reactive than zinc carboxylate and stearic acid, and only weakly reducing, it can be added over a wider concentration range. Once the particles are formed, it is very difficult to reduce them with hexadecylamine. Hexadecylamine slowed crystal growth considerably though. It also lead to difficulties in the growing of ZnS shells on the amine capped InP surfaces, due to its higher coordination with indium. Thus the best concentration is the minimum required to give a soluble indium complex. A molar concentration corresponding to the amount of indium and zinc was shown to be optimal.

3.4 Spectra

3.4.1 ^{31}P spectra of InP Nanoparticles without added zinc

The main features in the ^{31}P spectra of the InP nanoparticles are the wide peak between -100ppm and -300ppm and the three peaks at about 10ppm, 30ppm and 50ppm.

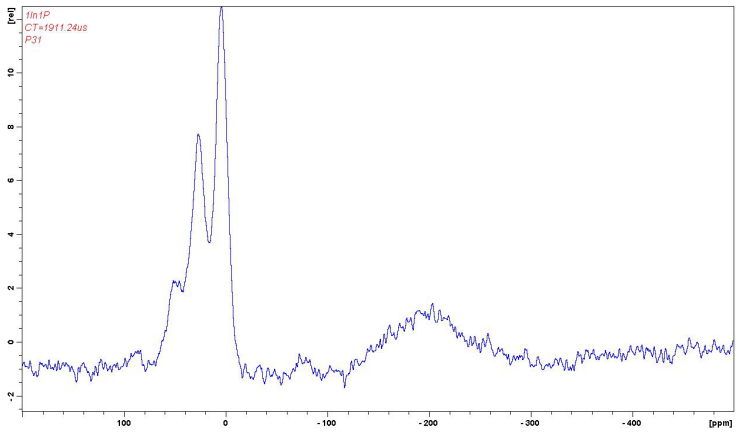


Figure 3.1: ^{31}P CP-spectrum of InP nanoparticles with an In:P ratio of 1:1
Acquisition parameters in table F.1

The wide peak centering at about 200ppm can be seen best in the non-CP spectra (Figures 3.3 and 3.4). They are much weaker in the spectra were cross polarization was used (Figures 3.1 and 3.2) because this resonance is due to the phosphorus in the nanoparticle itself. Since there are no protons in the particle itself, but only on the surface as the shell of the particle, the resonance for the CP-spectrum only shows the phosphorus close to the shell which is able to receive the magnetization from the protons.

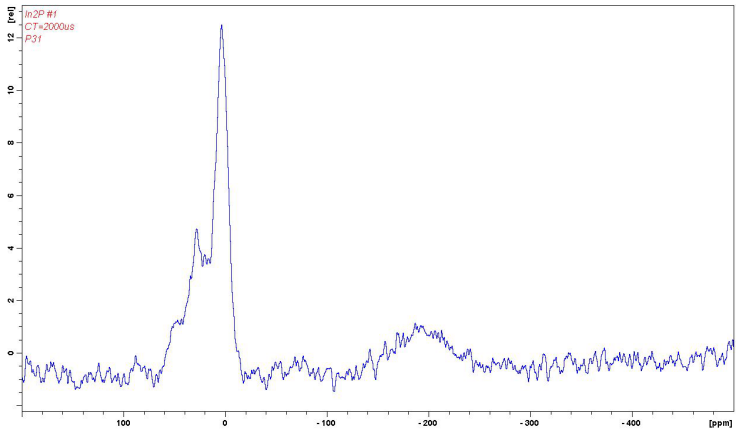


Figure 3.2: ^{31}P CP-spectrum of InP nanoparticles with an In:P ratio of 1:2
Acquisition parameters in table F.2

There are several possible explanations for the resonances at 10ppm, 30ppm and 50ppm. One possible explanation is that they are due to phosphorus on the surface of the particles and that the 10ppm resonance it a phosphorus with one dangling bond, the 30ppm resonance a phosphorus with two dangling bonds and the 50ppm resonance one with three dangling bonds. This would explain that there is a significantly stronger 10ppm signal than 30ppm signal.

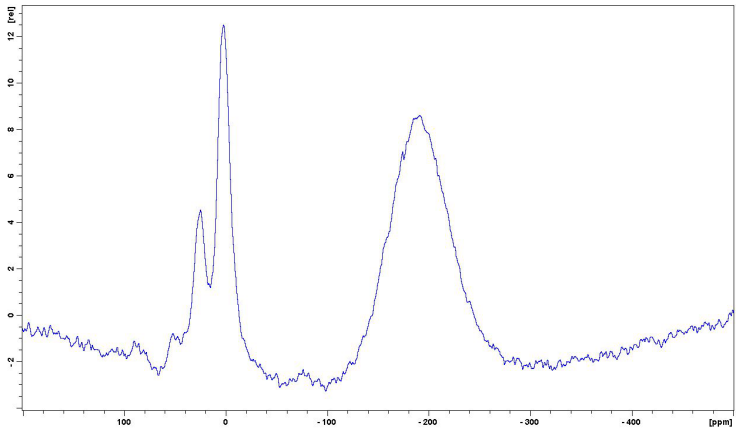


Figure 3.3: ^{31}P high-power proton-decoupled spectrum of InP nanoparticles with an In:P ratio of 1:1
Acquisition parameters in table F.3

If the resonances at 10ppm, 30ppm and 50ppm are from the surface of the par-

ticles, the phosphorus would be bonded to indium. Since indium is a quadrupolar nucleus with spin-$\frac{9}{2}$ it could lead to a widening of the phosphorus signal into a powder pattern. It is difficult to tell if this is the case, since there are two to three different signals in close vicinity. So the shape observed could be from two to three different regular signals partially overlapping, or from two powder patterns, possibly with a third signal mixed in.

It should be noted that the resonance at approximately 50ppm is weak or absent in phosphorus rich environments.

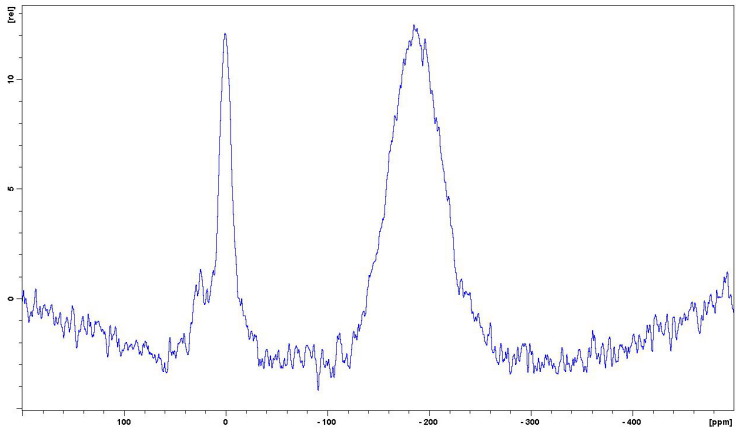


Figure 3.4: ^{31}P high-power proton-decoupled spectrum of InP nanoparticles with an In:P ratio of 1:2
Acquisition parameters in table F.4

To get some more data several CP-experiments with different contact times were made to get an idea of how far the different phosphorus resonances were from the protons in the shell.

The result was that all three resonances had a peak roughly at the same contact time, only the wide resonance between -100ppm and -300ppm had a peak at much higher contact times, conclusive with the previous interpretation that this resonance is due to the InP near the surface of the nanoparticle.

3.4.2 ^{31}P spectra of InP Nanoparticles with added zinc

The main resonances in the spectra of the InP nanoparticles with added zinc are basically the same. Only the CP-spectra are shown here, because except for the stronger wide particle resonance and the higher signal to noise ratio due to the higher recycling delay, the high-power proton-decoupled spectra are essentially the same.

One feature that can be noticed most prominently in Figure 3.5 are the two small peaks at approximately 95ppm and -90ppm. However these are not additional peaks but spinning sidebands of the 10ppm peak. This has been confirmed by changing the rotation frequency upon which these two peaks shifted too.

They can also be observed in Figures 3.1 and 3.7 although much less prominent.

Nothing significant was observed when comparing the ^{31}P-spectra of InP nanoparticles with or without added zinc.

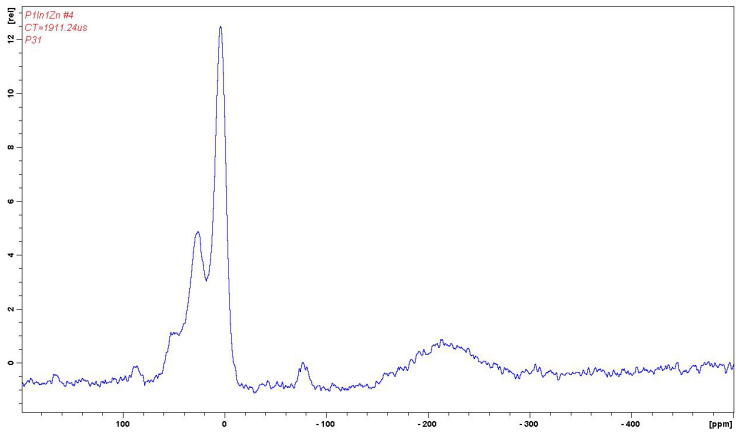


Figure 3.5: ^{31}P CP-spectrum of InP nanoparticles with added zinc undecylenate with an In:P:Zn ratio of 1:1:1
Acquisition parameters in table F.5

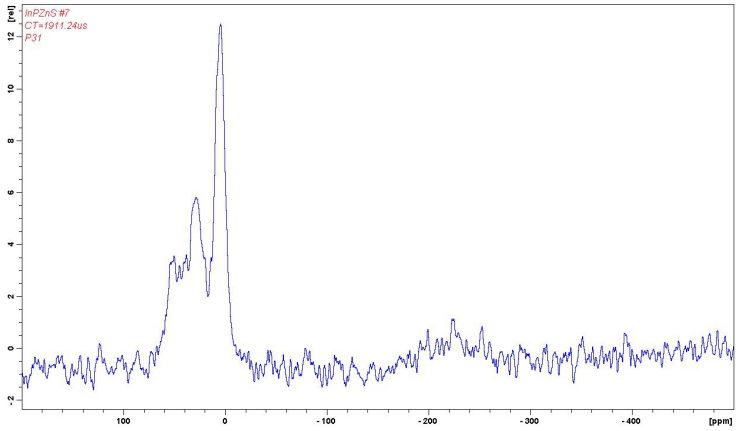


Figure 3.6: ^{31}P CP-spectrum of InP nanoparticles with added zinc diethyldithiocarbamate with an In:P:Zn ratio of 1:1:1
Acquisition parameters in table F.6

The low signal to noise ratio in Figure 3.7 is due to the extremely small amount of sample that was available. It was partially compensated by increasing the amount of acquisitions.

3.4.3 ^{31}P spectra of ZnP particles

The spectrum of the ZnP particles, which were synthesized without any indium present show the same resonances at approximately 10ppm and 30ppm. A third resonance at 50ppm might also be present, but is not strong enough to rise significantly out of the background noise.

One spinning sideband at approximately -90ppm is clearly visible while the one at approximately 95ppm can barely be seen.

The theory that the peaks between 10ppm and 60ppm might be due to interaction with the quadrupole nucleus indium can be neither confirmed nor denied, since when taking into account equation 3.4 and plugging in the known values

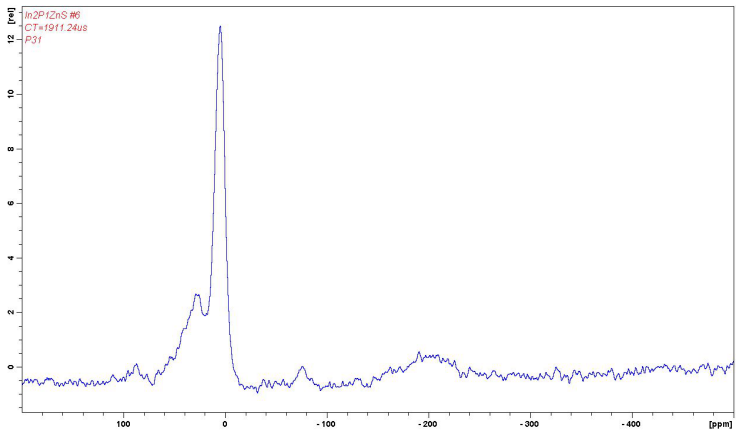


Figure 3.7: ^{31}P CP-spectrum of InP nanoparticles with added zinc diethyldithiocarbamate with an In:P:Zn ratio of 2:1:1
Acquisition parameters in table F.7

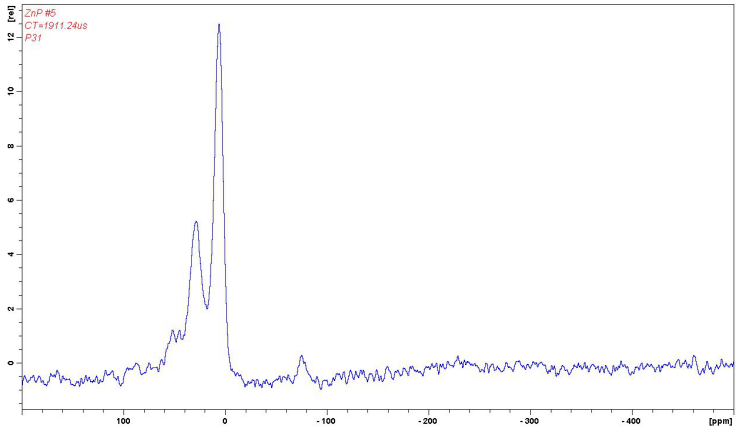


Figure 3.8: ^{31}P CP-spectrum of ZnP particles
Acquisition parameters in table F.8

for the quadrupole moment and larmor frequency of indium and zinc, the resulting values are of similar magnitude. This would mean that the line widening due to the interaction with zinc or indium would have approximately the same magnitude as well.

3.4.4 Spectra of InPZn nanoparticles with fatty amine

Only one sample prepared using the fatty amine synthesis method was investigated by NMR. This sample is a InP nanoparticle with added zinc with a In:P:Zn ratio of 1:1:1.

In the ^{31}P-spectrum (Figure 3.9) of this sample one difference to the other ^{31}P-spectra of InP nanoparticles can be seen. The three resonances that occurred at about 10ppm, 30ppm and 50ppm in the other spectra have shifted to the right and now occur at about -10ppm, 18ppm and 40ppm.

This could be due to the possible different oxidation states of the phosphorus waste generated during the reaction, or due to the different environment in the

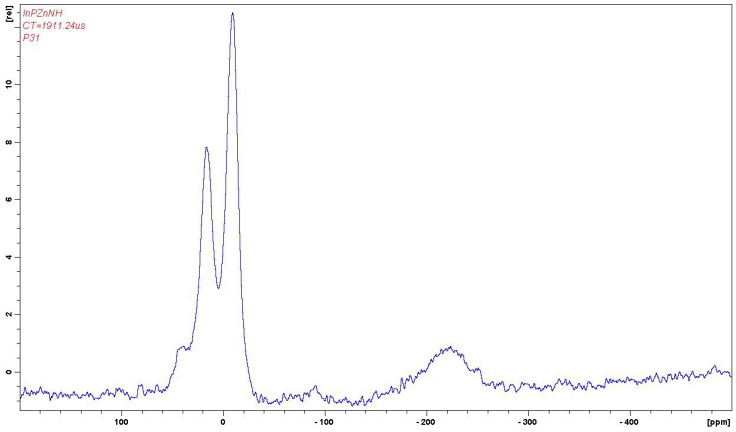


Figure 3.9: ^{31}P CP-spectrum of InP particles with added zinc synthesized using fatty amine

shell around the particles shifting the whole peak if the resonances are from the surface of the particles.

3.5 Conclusion

The wide signal between -100ppm and -300ppm, which is only observed in the InP containing samples and which is weaker in the CP-experiments is easily explained as a resonance resulting from the InP nanoparticle itself. In the CP-experiments only the outer layer of the actual InP particle is excited by energy transfer from the surrounding organic coating.

The three signals at approximately 10ppm, 30ppm and 50ppm remain more difficult to explain with any certainty. It is highly probable that these resonances are caused by phosphates. But whether these phosphates are from surface oxidization of the InP particle, or from reaction byproducts can not be assessed from the available data.

If the three resonances are from surface oxidation of the InP particle, they would probably represent different degrees of oxidation of surface phosphorus. This theory could be tested by synthesizing an indium phosphate complex and measuring its chemical shift.

Another possibility would be that some phosphorus environments, bonded to zinc, are in that region. This can not be the only explanation though, since the signals also occur in the zinc free samples. In theory it might be possible to investigate this using NMR.

3.5.1 Phosphorus-zinc bonding and its implication on ^{31}P-NMR

Since when a quadrupolar nucleus such a ^{67}Zn (spin-$\frac{5}{2}$ [14]) is dipolar coupled with a spin $\frac{1}{2}$ nucleus, the Hamiltonian in the rotating frame is:

$$\hat{H}^*(t) = (\omega_{0,S} - \omega_{0,I})\hat{S}_z + \hat{H}^*_{IS}(t) + \hat{H}^*_Q(t) \tag{3.1}$$

In this formula (formula 3.1 [1, p253, (5.26)]) which describes the Hamiltonian

of a spin-$\frac{1}{2}$ nucleus (I) dipolar coupled with a quadrupolar nucleus (S). It can clearly be seen that there is a quadrupolar term in there ($\hat{H}^*_Q(t)$).

$$\hat{H}^{(0)}_{IS} = \Lambda^{IS}\hat{T}^{IS} \tag{3.2}$$

For the first order Hamiltonian (indicated by $^{(0)}$) the whole thing reduces to the pure dipolar coupling (see equation 3.2 [1, p254, (5.28)]). However the second order Hamiltonian has a clear quadrupolar term as seen in equation 3.3 [1, p254, (5.30)].

$$\bar{H}^{(1)} = -\frac{i\pi}{2\omega_{0,I}} \sum_{q=-2}^{+2} \Lambda^{Q}_{2-q}\Lambda^{dd}_{2q} \left[\hat{T}^{Q}_{2q}, \hat{T}^{dd}_{2-q}\right] \tag{3.3}$$

The products of $\Lambda^{Q}_{2-q}\Lambda^{dd}_{2q}$ form equation 3.3 can also be expressed as a linear combination of new spherical tensors of rank 4, 2, and 0 and order zero.

The rotating frame Hamiltonian to the second order for a $\frac{1}{2}$-spin nucleus dipolar coupled with a quadrupolar nucleus is thus dependent on the spatial orientation. It is expressed by spatial tensors of rank 0, 2 and 4 which have a magnitude of the order of

$$d \cdot \frac{\frac{e^2qQ}{4I(2I-1)}}{\omega_{0,I}}. \tag{3.4}$$

Here d represents the dipolar coupling constant for the dipolar coupling between spins I and S. Even if the dipolar coupling is small, if the quadrupole coupling is large this term is quite non-negligible.

Thus there would be powder patterns with widths of the order described by equation 3.4.

However nice the theory is, there are many problems with this in reality. First there is the natural abundance for the two isotopes. ^{67}Zn has an abundance of only 4.1% [14], meaning that even though ^{31}P has an abundance of 100% [15] only 4.1% of the bonds between phosphorus and zinc would actually be affected by this, even though the nuclear quadrupole moment of relatively large with $150 \pm 15\,mb$ ($1.50.15 \times 10^{-}29\,\mathrm{m}^2$). Thus it is quite unlikely that any significant line broadening into a powder pattern in the ^{31}P-spectrum would be observed.

If a sample, enriched in ^{67}Zn, would be prepared, one might see such a powder pattern in the ^{31}P-spectrum. However while the presence of such powder patterns in the ^{31}P-spectrum would prove bonding to a quadrupolar nucleus, the lack of powder pattern would not prove lack of phosphorus-zinc bonding.

Even if such a powder pattern was present, it would more probably come from ^{115}In which is a $\frac{9}{2}$-spin nucleus with 95.7% abundance [16] and an even higher quadrupole moment of $770 \pm 8\,mb$ [16]. Even ^{113}In, which makes up the rest, is a $\frac{9}{2}$-spin nucleus with a similar quadrupole moment.

Chapter 4

Pharmaceuticals

Low resolution solid-state NMR can be useful in understanding the physical state a sample is in. Since different phases inside a sample redistribute magnetization at different rates, it is possible to get some insight into these phases by measuring T_1 and $T_{1\rho}$.

The two drugs that were used in these experiments, Etravirine and Felodipin, both can not be absorbed well by the human body when they are in a crystalline state.

Felodipin was mixed with a methacrylate copolymer and hot melt extrudates were made. Solid-state relaxation measurements were used to evaluate different ratios of Felodipin and copolymer for their crystallinity.

Etravirine needs to be mixed with hydroxypropyl methylcellulose for it to be absorbed by the human body. Here too crystallites of Etravirine would decrease the availability of the drug for the human body, so the different samples were investigated using relaxation measurements.

4.1 Etravirine

4.1.1 Samples

The aim was to find the proportion of amorphous and crystalline TMC inside several samples. Pure crystalline TMC-125 (the name used during the experiments for Etravirine) and specially milled, supposedly amorphous, TMC-125 were provided as reference.

Etravirine is an anti-viral drug used in the treatment of HIV. It is a non-nucleotide reverse transcriptase inhibitor and is marketed by Tibotec, a subsidiary of Johnson & Johnson.

Proton and ^{13}C-MAS spectra were recorded as well as T_1 and $T_{1\rho}$ measurements were conducted on pure Etravirine, hydroxypropyl-methylcellulose and mixtures of these two.

4.1.2 Results

The samples provided, containing the Etravirine, were numbered. T_1, $T_{1\rho}$ and CP measurements were done for Experiments 9719 – 9721, as well as HPMC and crystalline and amorphous Etravirine.

Since the data from the proton T_1 and $T_{1\rho}$ measurements often had more than one T_1 or $T_{1\rho}$ component, a nonlinear regression was utilized to find the T_1 and $T_{1\rho}$ values for the different components.

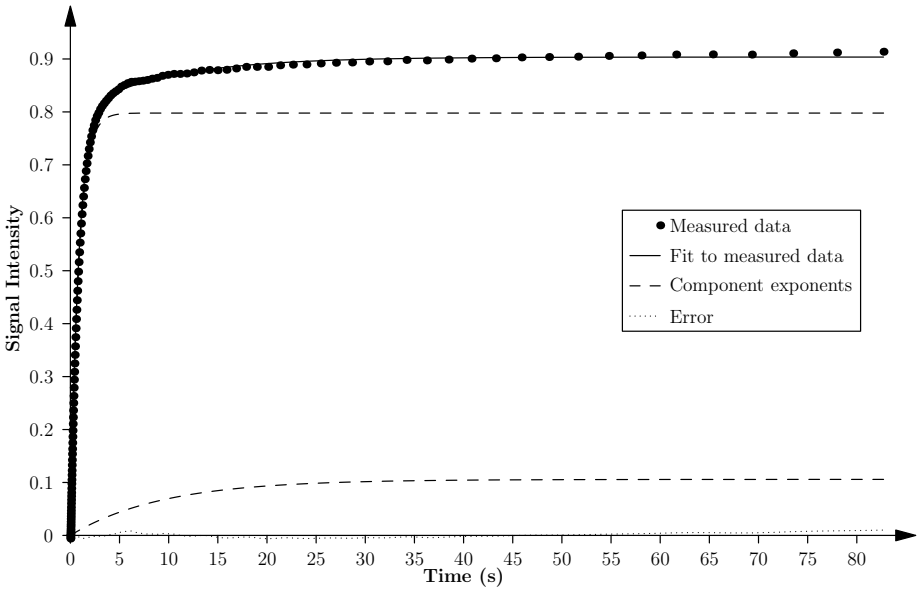


Figure 4.1: T_1 data for *Exp9719* with fit and component fit

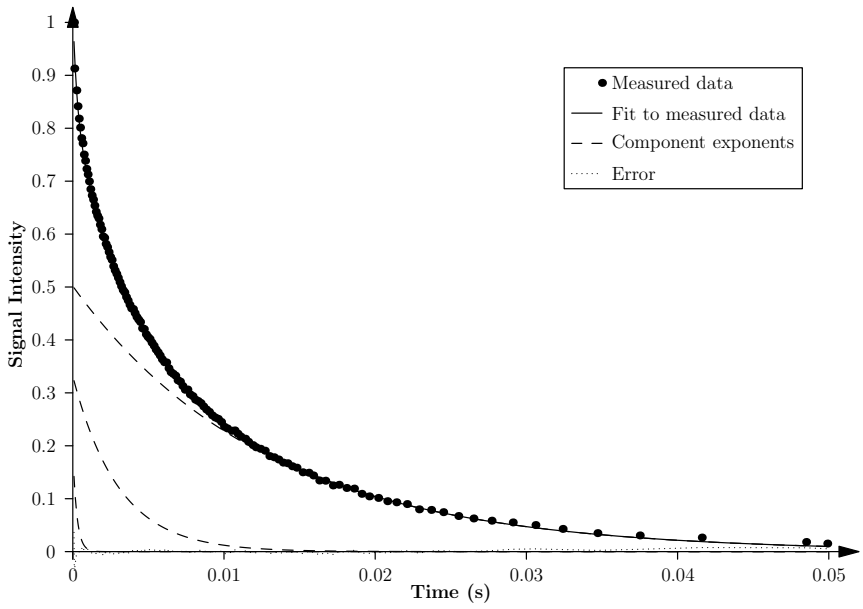


Figure 4.2: $T_{1\rho}$ data for *Exp9719* with fit and component fit

In graphs 4.1 and 4.2 an example for a multicomponent T_1 and $T_{1\rho}$ fit for the sample *Exp9719* can be seen respectively. Note that the measured data-points

are represented by dots, the fit by a solid line, the separate components of the fit by a dashed line, and the error of the fit by a dotted line which can be seen at the bottom near the axis.

		9719	9720	9721	HPMC	TMC_{am}	TMC_{cry}
Amp.		0.88	1.00	0.78	1.00	0.35	0.49
		0.12		0.22		0.65	0.51
Time Const.	$\times 10^{-1} s$	8.27	11.8	10.9	8.56	6.05	6.77
		9.37		19.2		2.28	18.1

Table 4.1: Fitting results for Etravirine saturation recovery experiments

When fitting curves to this data we expect to see one exponential decay for each of the phases present in the sample. This would give us the T_1 and $T_{1\rho}$ times for these phases from the time constants of the decay functions and the amount of each phase from the amplitude of each phase. However since the might be spin-diffusion occurring between the phases in such a sample, it is possible that the amplitudes of the curves do not provide phase composition information about the sample.

It can clearly be seen that there are two components for each the amorphous (milled) and crystalline Etravirine samples. The longer of the two components behaves as expected and is significantly longer with 18.1 s for the crystalline sample compared with the milled sample with 2.28 s T_1 time (see table 4.1).

The fact that there are actually two components for both the crystalline and amorphous samples can lead to two conclusions. Either the samples are not pure and actually a mixture of amorphous and crystalline Etravirine, or Etravirine exists in two or more polymorphs which are remarkably different in their dynamics. CP-MAS spectra with long and short recycle times appear very similar, suggesting that the two components are indeed chemically the same material.

It can also be deduced from the long 19.2 s long T_1 component in sample *Exp9721* that there is clearly crystalline Etravirine present in the sample. Whereas the 9.37 s T_1 component in sample *Exp9719* is clearly longer than in the milled sample but shorter than the crystalline sample. This could indicate that small crystallites have formed in the sample.

The single component in sample *Exp9720* could indicate that the T_1 times of both components are not different enough to differentiate between them. Since the short component in the other samples, that is probably due to the HPMC, is in the range of 0.8 – 1.1 s, and the milled Etravirines long component has a T_1 of 2.28 s, it is very probable that it is just not possible to differentiate between two T_1 components that only differ by a factor of 2. This would also suggest that this sample has the least crystalline Etravirine, or at least the smallest crystallites.

When looking at table 4.2, unfortunately the picture is not so clear. The difference in $T_{1\rho}$ times between the crystalline and amorphous (milled) sample is very clear, but other than that the data does not speak as clearly as the T_1 data does.

		9719	9720	9721	HPMC	TMC_{am}	TMC_{cry}
		0.19	0.12	0.08	0.15	0.06	0.06
Amp.		0.32	0.70	0.29	0.62	0.26	0.28
		0.49	0.18	0.63	0.23	0.68	0.66
	$\times 10^{-4} s$	2.69	14.4	13.2	12.0	8.04	8.28
Time Const.	$\times 10^{-3} s$	3.02	7.01	6.77	6.39	5.48	5.88
	$\times 10^{-2} s$	1.27	6.77	1.81	1.74	8.54	31.4

Table 4.2: Fitting results for Etravirine $T_{1\rho}$ spin-lock experiments

However the fact that the 7.01 ms component of sample *Exp9720* makes up 70% of the signal hints at the fact that there is indeed very little crystalline Etravirine present in that sample. Nonetheless this sample has the longest component of the three samples.

Since the the length scales, calculated using equation 2.2, for $T_{1\rho}$ are in the low nanometre scale instead of the hundreds of nanometre scale for T_1, it is likely that some of the components observed in the $T_{1\rho}$ data might actually be due to water. Due to the fact that the selective ^{13}C experiments result in the same spectra when all but the longest $T_{1\rho}$- component is filtered out, it is very likely that theses shorter $T_{1\rho}$- components are due to something not containing any carbon at all. The most likely candidate would be water.

Since the Felodipin samples were provided as brittle rods and it was not possible to get the rods to spin, even when using an inert material as filler, regular MAS-spectra were only made for the Etravirine samples.

4.2 Felodipin

4.2.1 Samples

Felodipin is a calcium channel blocker intended as a high blood pressure medication. The samples were short rods of a mixture of Felodipin and the methacrylate copolymer Eudragit prepared using hot melt extrusion (HME).

Since the samples could not be ground up, due to the oxygen sensitivity of the Felodipin, a rod of a diameter slightly smaller than the rotor was selected and broken of, so that it would fit into a rotor. The unbalance of the rotor was not a problem, since the T_1 and $T_{1\rho}$ experiments do not require any rotation.

The samples were also analyzed (although not by me) using scanning electron microscopy and differential scanning calorimetry. Heat capacity measurements were also made.

The purpose of all these measurements was to find the miscibility of the Felodipin with the polymer. Where the T_1 and $T_{1\rho}$ measurements allowed for an estimation of the size of the phase domains within the sample as well as the approximate proportions of (protons in) the phases.

4.2.2 Results

One of the first things that can be noticed when looking at the T_1 fitting results (Table 4.3), is that there is only one component for the 10% and the 20% samples with a time constant of about 0.6 s. Whereas the 30%, 50% and 70% samples have an additional component with a time constant of about 1 s. Since crystalline substances are known to have a long T_1, it is reasonable to assume that the component with the ≈ 1 s relaxation time is due to phase separated crystalline Felodipin.

It can also be noted that the percentage of the slow decaying component increases with drug loading.

		10:90	**20:80**	**30:70**	**50:50**	**70:30**
Amp.		1.00	1.00	0.67	0.49	0.43
				0.33	0.51	0.57
Time Const.	s	0.598	0.60	0.549	0.543	0.575
	s			0.994	0.953	1.060

Table 4.3: Felodipin T_1 fitting results

In the heat capacity measurements a clear melting of a crystalline phase can be detected in the 70% sample only. This suggests that the size of the crystals in the 30% and 50% samples is significantly smaller than in the 70% sample, leading to fast dissolution of the sub-micron crystals during heating.

Although the 10% and 20% samples appear as only one component for the T_1 measurements, they have two components for the $T_{1\rho}$ measurements (Table 4.4). This suggests that there is some phase separation even in the 10% and 20% samples. The most likely reason why these phases were not detected in the T_1 measurements is the difference in time scales of T_1 and $T_{1\rho}$.

		10:90	**20:80**	**30:70**	**50:50**	**70:30**
Amp.		0.16	0.15	0.33	0.25	0.24
		0.84	0.85	0.67	0.75	0.76
Time Const.	$\times 10^{-3} s$	1.05	1.16	2.62	1.28	2.70
	$\times 10^{-2} s$	1.34	1.30	1.32	1.19	1.96

Table 4.4: Felodipin $T_{1\rho}$ fitting results

For a heterogeneous system the relaxation times can be used to estimate the dimension of the separated phases. Since we are dealing with a solid it can safely be assumed that the exchange in these time-scales is not by matter diffusion but spin diffusion. Therefore the following relationship[8] is valid:

$$2\sqrt{2}\frac{\mathcal{A}^2}{\pi^2 \mathcal{D}}|\Delta\gamma| > 1 \tag{4.1}$$

Here $\mathcal{A}$ is the smallest dimension (between the separated domains) over which diffusion takes place. $\mathcal{D}$ is the spin-diffusion coefficient which in polymeric sys-

tems has a typical value of 10^{-16} m^2s^{-1}[17, p52–78] and $|\Delta\gamma|$ is calculated from the relaxation rates of the separated phases as follows:

$$\Delta\gamma = \frac{1}{T_A} - \frac{1}{T_B} \tag{4.2}$$

where $T_{1,A}$ represents the longer of the two relaxation times and $T_{1,B}$ the shorter.

Plugging the numbers of the fitting results into formula 4.1 gives a diffusive path length between the neighboring domains in the T_1 experiments of no smaller than ≈ 22 nm, and of no smaller than ≈ 5.6 nm for the $T_{1\rho}$ experiments.

These numbers suggest that the mixing of the drug and polymer in the 10% and 20% samples is between 5.6 nm and 22 nm. However it should be noted that these dimensions are not an accurate size measurement but only an estimation.

4.2.3 Spectra

Testing T_1- and $T_{1\rho}$-selective CP experiments using a mixture of Adamantane and Glycine

The main interest of the spectra was to determine if the different phases seen in the T_1 and $T_{1\rho}$ experiments had different compositions. Different line widths would also reflect the amount of disorder in the different phases. To prove that it was possible and to see how well the methods of pre-saturating the sample to exclude signals from phases with low T_1, and spin-locking the sample prior to acquisition to exclude signals with low $T_{1\rho}$, worked a mixture of Adamantane and Glycine was selected.

Adamantane has a short T_1 but a long $T_{1\rho}$ compared to Glycine. The fact that these two components are readily available, stable and both give nice and easy to acquire ^{13}C spectra made them the ideal sample to test this method.

The regular ^{13}C-CP-MAS-spectrum of the mixture can be seen in Figure 4.3. The two signals of Adamantane at 28.6ppm and 38.0ppm can clearly be seen in this spectrum. The two signals originating from Glycine at 44.2ppm and 173.3ppm are also easily identified.

For both of the selective experiments, the selection was done through the proton channel. This means that the protons were pre-saturated or that there was an additional spin-lock, between the 90°-pulse and the contact phase, introduced on the proton channel to the regular CP experiment. This results in a cross-polarization selectively in either Glycine or Adamantane.

When comparing this to the pre-saturated spectrum (Figure 4.4), where the sample was saturated and left standing for 0.2s prior to each acquisition to remove the Adamantane signal (which has a measured T_1 of 0.669s), it can clearly be seen that although the Adamantane signals were not completely removed, their intensity is clearly reduced. All the while there is no effect on the intensity of the Glycine signal.

The spectrum in Figure 4.5 was acquired by adding a 100ms spin-lock in between the 90° pulse and the contact between the proton and carbon. This

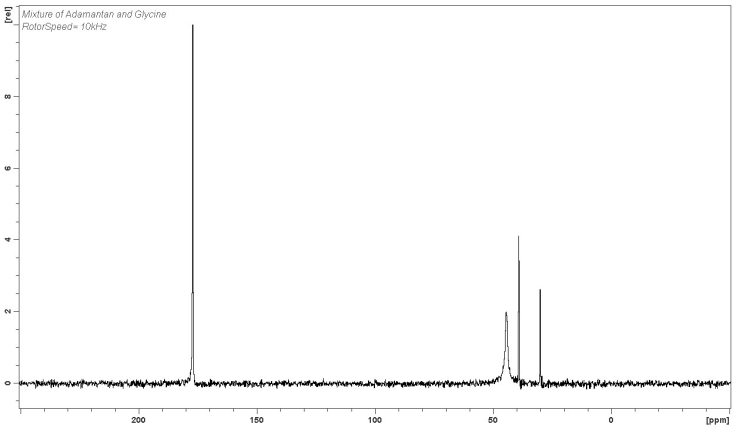


Figure 4.3: ^{13}C CP-MAS-spectrum of a mixture of Adamantane and Glycine
Acquisition parameters in table F.10

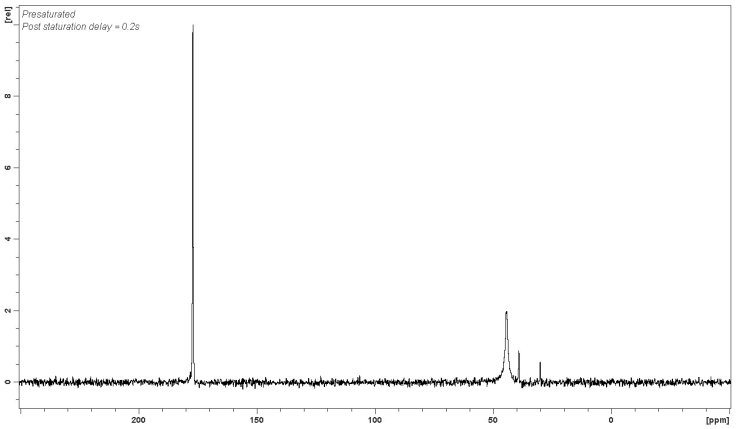


Figure 4.4: pre-saturated ^{13}C-spectrum of a mixture of Adamantane and Glycine with a 0.2s delay between pre-saturation and acquisition
Acquisition parameters in table F.11

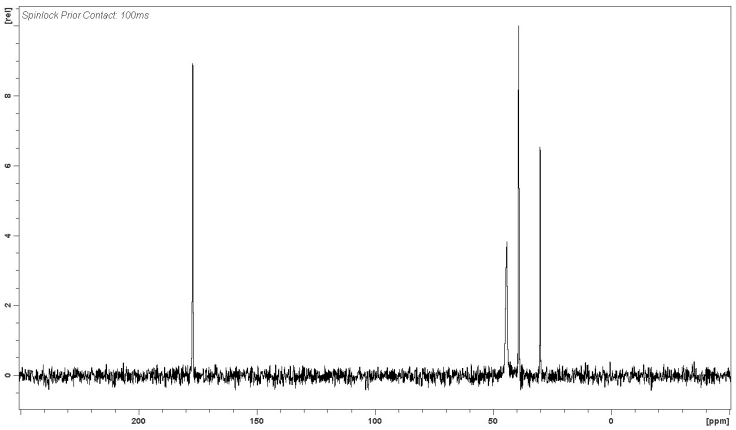


Figure 4.5: ^{13}C-spectrum, with 100ms spin-lock prior acquisition, of a mixture of Adamantane and Glycine
Acquisition parameters in table F.12

should remove any signals with a $T_{1\rho}$ significantly lower than this value but retain any higher than it. It can easily be seen that even though the signal to noise ratio is clearly lowered by this method, the Adamantane signal now has significantly more intensity compared to the Glycine signal. This method was not successful at removing the Glycine signal, but it successfully reduced its intensity resulting in a stronger Adamantane signal. The relative intensity scale can be slightly misleading, since more acquisitions were made for the spin-locked spectrum than the other two to compensate for reduced signal to noise.

T_1- and $T_{1\rho}$-selective CP experiments of Etravirine samples

The T_1- and $T_{1\rho}$-selective CP experiments that were tested using a Adamantane-Glycine mixture were done on the samples *Exp9719* and *Exp9720*.

A regular CP-MAS-spectrum as well as a pre-saturated CP-MAS-spectrum and a CP-MAS-spectrum with added spin-lock was made for the samples *Exp9719* and *Exp9720* as well as a sample of pure Etravirine. A CP-MAS-spectrum for HPMC was made for comparison.

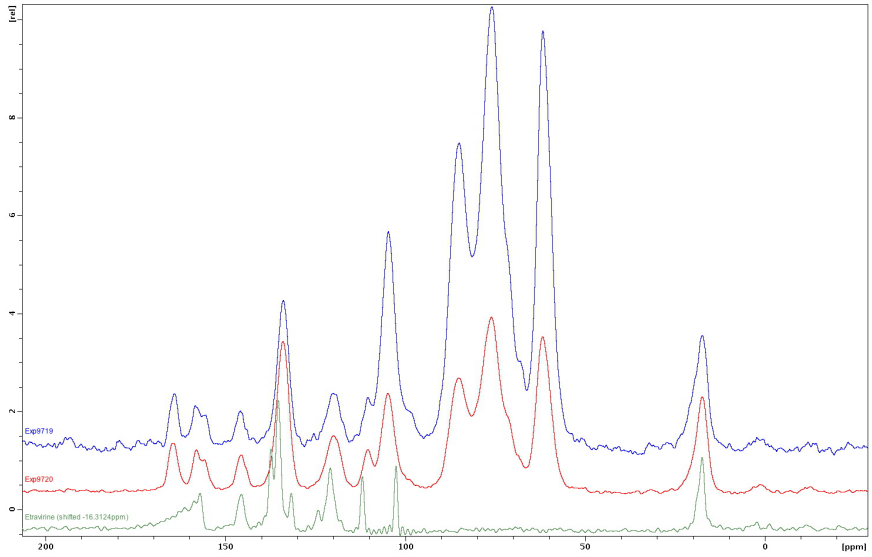


Figure 4.6: ^{13}C-CP-MAS spectrum of samples *Exp9720*, *9720* and crystalline Etravirine
Acquisition data in table F.13

When comparing the regular CP-MAS-spectra of *Exp9719* (Figure 4.6) and *Exp9720* (Figure 4.6) and taking into account the signals from the Etravirine sample (Figure 4.6) and the HPMC (Figure 4.7) one notices that the signals in the two mixtures are broader and less detailed. This can be an effect of the intimate but random mixture of the two compounds. Since the mixture is not ordered on a molecular level, each molecule of Etravirine or HPMC can encounter a slightly different environment broadening the signals.

When comparing just the two mixtures *Exp9719*, which is a 1:3 mixture of Etravirine and HPMC, and *Exp9720*, which is a 1:1 mixture, it can clearly be seen that this different Etravirine content is results in different intensities of the peaks between 110ppm and 170ppm as well as the peak at 20ppm.

When comparing this with the Etravirine spectrum (Figure 4.6) where most peaks occur between 110ppm and 180ppm and an additional peak at about 35ppm this fits quite well with the exception of the 35ppm peak which is shifted upfield by 15ppm. This could be due to interactions with HPMC near the corresponding carbon nucleus.

Comparing the regular CP-MAS-spectrum of *Exp9719* (Figure 4.6) with the pre-saturated CP-MAS-spectrum (Figure 4.8) or the CP-MAS-spectrum with spin-lock (Figure 4.9), one notices that they are nearly identical.

The same can be noticed when comparing the regular CP-MAS-spectrum of *Exp9720* (Figure 4.6) with the corresponding pre-saturated (Figure 4.10) and spin-locked (Figure 4.11) version.

This leads to the conclusion that the different phases observed through the T_1 and $T_{1\rho}$ measurements have the same composition chemically. The phases can only have a slight difference in composition, because any significant difference in composition between the phases would be readily noticeable by an increase or decrease of the corresponding peaks in either the spin-locked or pre-saturated spectra compared to the regular CP-MAS-spectra.

The difference in the phases is therefore most probably physical in nature, with the proportions of Etravirine and HPMC being very similar if not the same in these different phases.

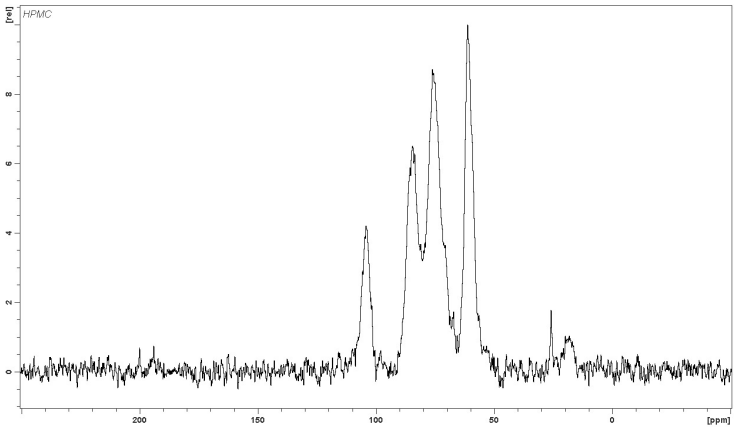


Figure 4.7: ^{13}C-CP-MAS spectrum of HPMC
Acquisition data in table F.14

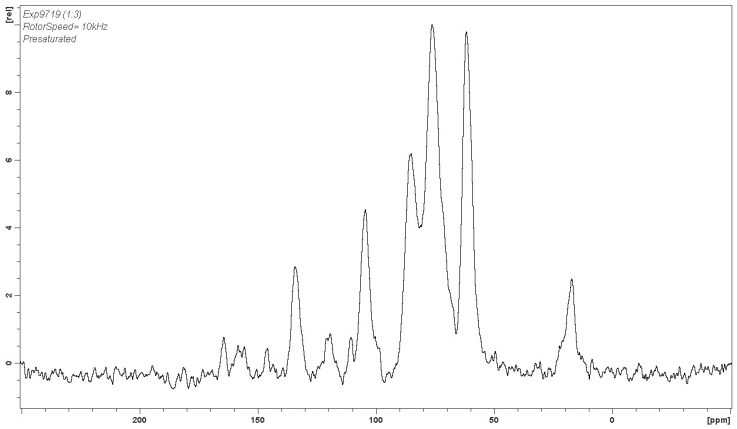


Figure 4.8: ^{13}C-CP-MAS spectrum of sample *Exp9719* with a 2s delay between pre-saturation and regular CP-acquisition
Acquisition data in table F.15

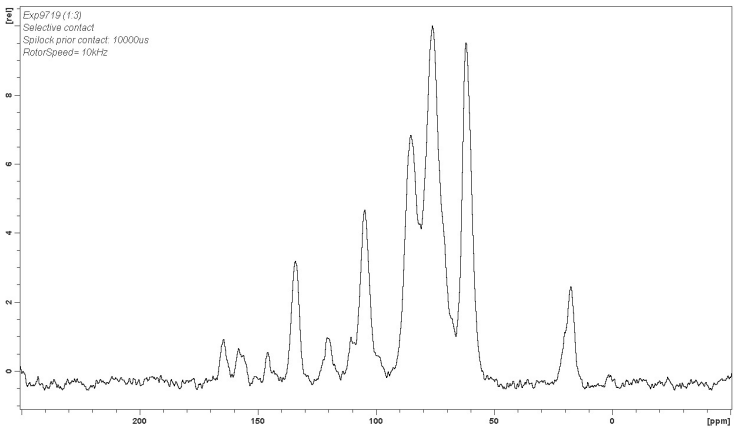


Figure 4.9: ^{13}C-CP-MAS spectrum of sample *Exp9719* with a 10ms spin-lock between the proton pulse and proton-carbon contact
Acquisition parameters in table F.16

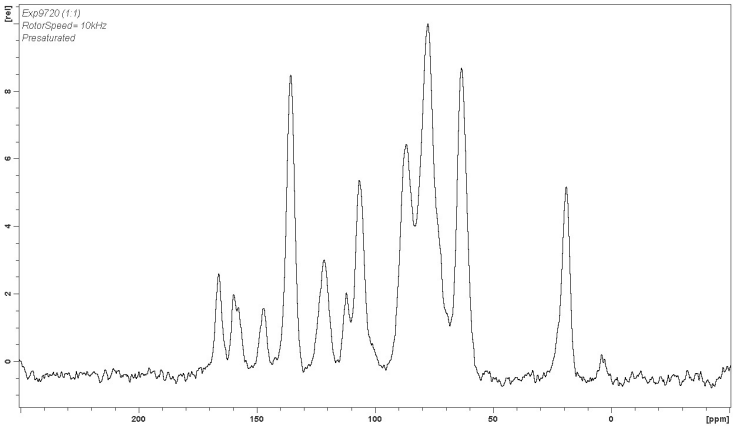


Figure 4.10: ^{13}C-CP-MAS spectrum of sample *Exp9720* with a 2s delay between pre-saturation and regular CP-acquisition
Acquisition parameters in table F.17

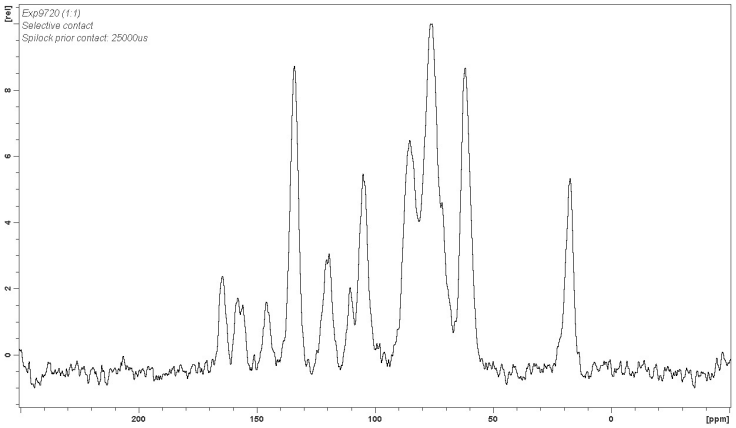


Figure 4.11: ^{13}C-CP-MAS spectrum of sample *Exp9720* with a 25ms spin-lock between the proton pulse and proton-carbon contact
Acquisition parameters in table F.18

Chapter 5

TLM Model

5.1 Introduction

Diffusion occurs in many different systems in physics. The one we are interested in, spin diffusion, behaves similar to many other types of diffusion. When trying to simulate spin diffusion, different diffusion coefficients for different phases within the material as well as the possibility for different concentrations of active nuclei have to be taken into account.

When looking for a system that could be used to simulate this, attention fell to the field of electronics. In a grid of capacitors connected by resistors, the charge will diffuse through the grid. The useful thing for these kind of systems is that there is a model available to describe the behavior of these systems. The system is conveniently quantized, since each cell consists of one capacitor connected to its neighbors through resistors, it is easy to calculate the charge of each capacitor in such a system over time.

The TLM Model [3] can be nicely applied to spin diffusion and is quite easy to implement as a computer program. The resistance between the cells can be used to model different diffusion coefficients between the different materials, while the capacitance could be used to model different active nucleus concentrations, and thus different capacities for energy stored in excited spins, in the different phases.

It should be noted that the TLM Model itself was taken from the literature [3] while its application to spin diffusion and the programming of the software to do the simulations to simulate spin diffusion using this model are the work of the author.

5.2 Theory

A program to simulate Spin Diffusion was written, using an algorithm which originally comes from the field of Electronics. The algorithm is designed to calculate the behavior of a network of resistors with capacitors to ground at each junction.

One of two possible systems can be used, the only significant difference between them is the measurement point. Either the measurement is undertaken across the plates of the capacitors, or between the *center* of the resistors and ground. The

former is called a link-resistor TLM node, while the latter is called link-line TLM node.

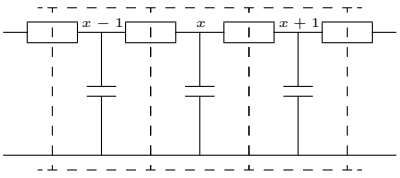


Figure 5.1: A network of link-line nodes [3]

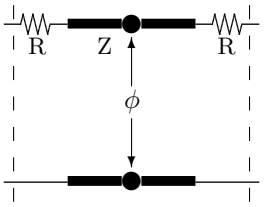


Figure 5.2: A single link-line node [3]

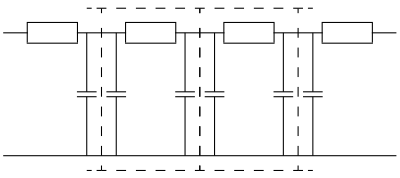


Figure 5.3: A network of link-resistor nodes [3]

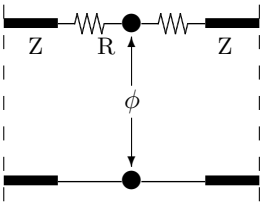


Figure 5.4: A single link-resistor node [3]

A network of link-line nodes (Figure 5.1) consists of several separate link-line nodes (Figure 5.2). Each node is connected over the resistor to its neighbor, while ϕ is measured over the capacitance.

In a network of link-resistor nodes (Figure 5.3) on the other hand, the nodes (Figure 5.4) are connected over the capacitor and ϕ is measured over the resistance.

TLM-nodes and spin diffusion

When keeping in mind we are actually modeling spin diffusion, it is clear that the capacitor would represent the nuclear spins themselves and their capacity to store energy by being excited to a higher magnetic energy level. The resistors would represent the spin-spin coupling, allowing energy to be transferred from one spin to the other.

5.2.1 The TLM algorithm for a one dimensional system

Link-Line Node

When a voltage impulse is entering a link-line node, it encounters a discontinuity, $Z_T = R + R + Z$. At this discontinuity part of the impulse is reflected back and only a fraction is transmitted. The reflection coefficient is

$$\begin{aligned} \rho = & \frac{Z_T - Z}{Z_T + Z} \\ = & \frac{R}{R+Z}. \end{aligned} \tag{5.1}$$

Therefore the transmission coefficient is described by

$$\tau = \frac{Z}{R+Z}. \tag{5.2}$$

Consider two Incident ($\mathcal{I}$) pulses (${}^{\mathcal{I}}_{\mathbf{L}}V_k(x)$ and ${}^{\mathcal{I}}_{\mathbf{R}}V_k(x)$) are approaching the resistors at the center of node x from **L**eft and **R**ight respectively. The voltage at the measurement point at the center of the node is therefore

$$\begin{aligned} \phi_k(x) = & \frac{2\,{}^{\mathcal{I}}_{\mathbf{L}}V_k(x)\,(R+Z)}{2R+2Z} + \frac{2\,{}^{\mathcal{I}}_{\mathbf{R}}V_k(x)\,(R+Z)}{2R+2Z} && (5.3)\\ = & \quad {}^{\mathcal{I}}_{\mathbf{L}}V_k(x) + {}^{\mathcal{I}}_{\mathbf{R}}V_k(x). && (5.4) \end{aligned}$$

The Scattering ($\mathcal{S}$) (reflection and transmission) due to these incident pulses is described by

$$\begin{aligned} {}^{\mathcal{S}}_{\mathbf{L}}V_k(x) = & \quad \rho\,{}^{\mathcal{I}}_{\mathbf{L}}V_k(x) + \tau\,{}^{\mathcal{I}}_{\mathbf{R}}V_k(x) \\ {}^{\mathcal{S}}_{\mathbf{R}}V_k(x) = & \quad \tau\,{}^{\mathcal{I}}_{\mathbf{L}}V_k(x) + \rho\,{}^{\mathcal{I}}_{\mathbf{R}}V_k(x) \end{aligned} \qquad (5.5)$$

or

$${}^{\mathcal{S}}_{k}\begin{pmatrix} V_L(x) \\ V_R(x) \end{pmatrix} = \begin{pmatrix} \rho & \tau \\ \tau & \rho \end{pmatrix} {}^{\mathcal{I}}_{k}\begin{pmatrix} V_L(x) \\ V_R(x) \end{pmatrix}. \qquad (5.6)$$

Each scattered pulse now takes half a time unit to travel to the boundaries of the nodes, and after another half time unit they become incident pulses at the adjacent nodes:

$$\begin{aligned} {}^{\mathcal{I}}_{\mathbf{L}}V_{k+1}(x) = & \quad {}^{\mathcal{S}}_{\mathbf{R}}V_k(x-1) \\ {}^{\mathcal{I}}_{\mathbf{R}}V_{k+1}(x) = & \quad {}^{\mathcal{S}}_{\mathbf{L}}V_k(x+1) \end{aligned} \qquad (5.7)$$

Repeating the steps (5.3), (5.5) and (5.7) for each *unit* of time (Δt) now constitutes the algorithm.

Link-Resistor Node

For a Link-Resistor Node the algorithm calculates the potentials at the interface between the nodes. This is simply the sum of the left and right going pulses from the nodes at $x-1$, x and $x+1$. The pulse at $x-1$ traveling left and the pulse at $x+1$ traveling right are not relevant to the node at x, therefore we are left with ${}^{\mathcal{S}}_{\mathbf{R}}V_k(x-1)$, ${}^{\mathcal{S}}_{\mathbf{L}}V_k(x)$, ${}^{\mathcal{S}}_{\mathbf{R}}V_k(x)$ and ${}^{\mathcal{S}}_{\mathbf{L}}V_k(x+1)$. These pulses travel for a time $\frac{\Delta t}{2}$ before they are scattered at the resistors, they then become incident on x from left and right:

$$\begin{aligned} {}^{\mathcal{I}}_{\mathbf{L}}V_{k+1}(x) = & \quad \rho\,{}^{\mathcal{S}}_{\mathbf{L}}V_k(x) + \tau\,{}^{\mathcal{S}}_{\mathbf{R}}V_k(x-1) \\ {}^{\mathcal{I}}_{\mathbf{R}}V_{k+1}(x) = & \quad \rho\,{}^{\mathcal{S}}_{\mathbf{R}}V_k(x) + \tau\,{}^{\mathcal{S}}_{\mathbf{L}}V_k(x+1) \end{aligned} \qquad (5.8)$$

The pulses sum to give the potential

$$\phi(x)_{k+1} = {}^{\mathcal{I}}_{\mathbf{L}}V_{k+1}(x) + {}^{\mathcal{I}}_{\mathbf{R}}V_{k+1}(x). \qquad (5.9)$$

Once the pulses continue one should redesignate them for the next iteration:

$$\begin{aligned} {}^{S}_{\mathbf{R}}V_{k+1}(x) &= {}^{J}_{\mathbf{L}}V_{k+1}(x) \\ {}^{S}_{\mathbf{L}}V_{k+1}(x) &= {}^{J}_{\mathbf{R}}V_{k+1}(x) \end{aligned} \tag{5.10}$$

The complete algorithm for the Link-Resistor model consists of the three sets of equations (5.8), (5.9) and (5.10).

5.2.2 Boundaries

Traditionally boundaries are placed at the interface between two nodes.

Insulating Boundary

An insulating boundary will reflect all pulses. This can easily be modeled by setting $\rho = 1$ for the pulses that would otherwise travel out of the boundaries of the simulation.

This can also be used to exploit symmetries in the system. Any planes of symmetry (σ) can help reduce the area to be simulated considerably by only simulating a fraction of the system and replacing the planes of symmetry with insulating boundaries.

When relating this to spin diffusion, an insulating boundary can be used to model a symmetric system. Since all pulses are reflected back, a system with a mirror symmetry could be halved and an insulating boundary placed on the mirror plane. Thus large uniform systems can be simulated by simulating only a single unit cell of the system, since all unit cells should behave exactly the same.

Perfect Heat-Sink Boundary

A perfect heat-sink boundary is a boundary that will act as a perfect energy sink. This boundary has to be modeled slightly different for link-line and link-resistor models.

In the link-line model, the pulse will be half way along a transmission line when it sees a termination $Z_T = 0$. The reflection coefficient is thus $\rho = -1$.

For a normal node in the the link-resistor model the load impedance a pulse sees when it reaches the end-of-line is $R + R + Z$. A short circuit condition is described in such a way that the short is located immediately outside the node. Therefore the line terminating impedance is $Z_T = R$, which then gives a reflection coefficient of

$$\rho = \frac{R - Z}{R + Z}. \tag{5.11}$$

In a spin diffusion system a heat-sink boundary would occur at the edges of the sample. However even a small amount of sample would be so huge on the scale at which spin diffusion occurs, that it usually does not influence the sample significantly.

Constant temperature boundaries

In the link-line model the transmission line touches the boundary which is held at a constant value (V_C). This can easily be modeled by assuming there is a *ghost* node outside the boundary which has a source and a transmission line.

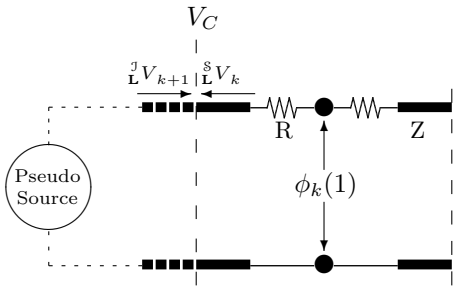


Figure 5.5: Constant temperature boundary showing the ghost node for a link-line system [3]

This leads to a constant potential at the boundary. The sum of the pulse incident from node 1 at the new time step and the pulse scattered from node 1 the previous time step is always constant. Since $\overset{\mathrm{S}}{\mathbf{L}}V_k(1)$ is known, $\overset{\mathrm{J}}{\mathbf{L}}V_{k+1}(1)$ can be calculated using the following equation:

$$\overset{\mathrm{J}}{\mathbf{L}}V_{k+1}(1) + \overset{\mathrm{S}}{\mathbf{L}}V_k(1) = V_C \tag{5.12}$$

With a link-resistor model the situation at the boundary is quite different. Here a resistor touches the boundary. For the node touching this boundary one has to consider two separate things. Firstly the input from the source which is now placed directly at the boundary, and secondly the history of the pulse which is scattered from node 1 and now approaches the boundary.

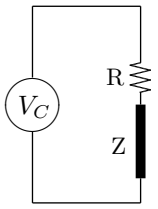


Figure 5.6: The network as seen from the source [3]

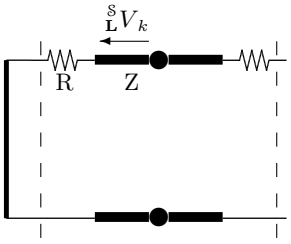


Figure 5.7: The situation for the pulse scattered towards the boundary [3]

The pulse scattered left sees a short-circuit, while the source sees a resistor in series with an impedance. The incidence from the left for node 1 can therefore be calculated as the sum of these two contributions:

$$\begin{aligned} \overset{\mathrm{J}}{\mathbf{L}}V_{k+1}(1) &= \tfrac{Z}{R+Z}\,V_C + \tfrac{R-Z}{R+Z}\,\overset{\mathrm{S}}{\mathbf{L}}V_k(1) \\ &= \rho\,V_C + (\rho-\tau)\,\overset{\mathrm{S}}{\mathbf{L}}V_k(1) \end{aligned} \tag{5.13}$$

For spin diffusion, a constant temperature boundary could theoretically be used to model a case where a phase, surrounding the simulated system, is cur-

rently being subjected to a long low power RF pulse tuned to its resonant frequency.

5.2.3 Inputs

Single shot injection

A single shot injection will ultimately lead to a Gaussian distribution of energy. It is basically a voltage of current source which is switched across a node during the first iteration of the simulation. This injected signal sees a junction with equal impedance to all directions. Thus the current divides equally in all directions.

When considering a one dimensional system where 100 *units* of energy are injected, the initial conditions are: ${}^{\mathcal{J}}_{\mathbf{L}}V_{k=0} = 50$ and ${}^{\mathcal{J}}_{\mathbf{R}}V_{k=0} = 50$.

For a link-line model this kind of injection reveals a curiosity. With a link-line model the propagation of such a single shot injection on only one node will result in singularities, so that for $k = 1$ the values of $\phi(x-1)$ and $\phi(x+1)$ will have a value > 0 while $\phi(x)$ will be zero. The pulses scattered from $\phi(x-1)$ and $\phi(x+1)$ are then scattered so that at $k = 2$ the values for $\phi(x-2)$, $\phi(x)$ and $\phi(x+2)$ will have values > 0 but $\phi(x-1)$ and $\phi(x+1)$ will be zero.

These singularities, which are obviously unphysical, will propagate in such a way that each node will be undefined at every other time-step. These singularities are not observed with the link-resistor model, and can be avoided by moving the excitation point to the boundary between two nodes. An injection of V_I between two nodes (x and $x+1$) can be realized as follows:

$$\begin{aligned} {}^{\mathcal{J}}_{\mathbf{R}}V_{k=0}(x) &= \tfrac{V_I}{2} \\ {}^{\mathcal{J}}_{\mathbf{R}}V_{k=0}(x+1) &= \tfrac{V_I}{2} \end{aligned} \tag{5.14}$$

A single shot injection would be the ideal case for an NMR pulse as one would like it for a perfect spin diffusion experiment. A pulse, infinitely short, which excites only the nuclei of one of the phases. Sadly in reality such a perfect pulse does not exist. However it is useful since it allows for an uncomplicated modeling of just the spin diffusion process.

Multiple injections into bulk material

Energy sources can be realized that inject a constant (or even time variable) energy into the bulk material. It is possible to realize this injection of energy at just about any point in every step of the iteration.

The most convenient point of adding a pulse (I_{EX}) is often immediately after the incidence step. Equation 5.3 or Equation 5.9 (depending on which model is used) is expanded like:

$$\begin{aligned}\phi_k(x) = & \frac{\left(\frac{2\,{}^{\mathfrak{I}}_{\mathbf{L}}V_k(x)}{R+Z}\right)+\left(\frac{2\,{}^{\mathfrak{I}}_{\mathbf{R}}V_k(x)}{R+Z}\right)+I_{EX}}{\left(\frac{2}{R+Z}\right)} \\ = & \quad {}^{\mathfrak{I}}_{\mathbf{L}}V_k(x)+{}^{\mathfrak{I}}_{\mathbf{R}}V_k(x)+\tfrac{2I_{EX}}{R+Z}\end{aligned} \tag{5.15}$$

for the link-line model and

$$\phi_{k+1}(x) = {}^{\mathfrak{I}}_{\mathbf{L}}V^{k+1}(x)+{}^{\mathfrak{I}}_{\mathbf{R}}V_{k+1}(x)+\frac{2I_{EX}}{R+Z} \tag{5.16}$$

for the link-resistor model.

Since this will only change the displayed values of $\phi_{k+1}(x)$, the scattered pulses also need to take the injected energy into consideration. For the link-line model we modify Equation 5.5 in incorporate V_{EX}:

$$\begin{aligned}{}^{\mathfrak{S}}_{\mathbf{L}}V_k(x) = & \quad \rho\,{}^{\mathfrak{I}}_{\mathbf{L}}V_k(x)+\tau\,{}^{\mathfrak{I}}_{\mathbf{R}}V_k(x)+\tfrac{V_{EX}}{2} \\ {}^{\mathfrak{S}}_{\mathbf{R}}V_k(x) = & \quad \tau\,{}^{\mathfrak{I}}_{\mathbf{L}}V_k(x)+\rho\,{}^{\mathfrak{I}}_{\mathbf{R}}V_k(x)+\tfrac{V_{EX}}{2}\end{aligned} \tag{5.17}$$

For the link-resistor model we modify Equation 5.10:

$$\begin{aligned}{}^{\mathfrak{S}}_{\mathbf{R}}V_{k+1}(x) = & \quad {}^{\mathfrak{I}}_{\mathbf{L}}V_{k+1}(x)+\tfrac{V_{EX}}{2} \\ {}^{\mathfrak{S}}_{\mathbf{L}}V_{k+1}(x) = & \quad {}^{\mathfrak{I}}_{\mathbf{R}}V_{k+1}(x)+\tfrac{V_{EX}}{2}\end{aligned} \tag{5.18}$$

This would be a much more realistic modeling of a phase selective RF-pulse, since an RF-pulse as used in solid state NMR is of significant duration that it often can not be reduced to an infinitely short pulse.

5.2.4 Going to the second and third dimension

Taking the TLM-algorithm to the second dimension is not very complicated. The only thing one really has to mind is managing the sheer amount of data and connections.

Two- and three-dimensional link-line nodes

The pulse in Figure 5.8 coming from the north sees an impedance consisting of one resistor in series with three parallel impedances (R + Z). The reflection coefficient for this arrangement is therefore:

$$\begin{aligned}\rho = & \frac{3R+R+Z-3Z}{3R+R+Z+3Z} \\ = & \frac{R-\frac{1}{2}Z}{R+Z}\end{aligned} \tag{5.19}$$

The transmitted component (τ) in the other three directions is therefore $3\tau = 1-\rho$.

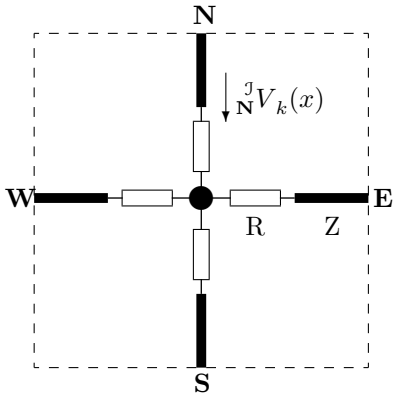


Figure 5.8: A two dimensional link-line node with an incident pulse from the north [3]

Since we have to account pulses from every direction, the potential is thus calculated as:

$$\begin{aligned} \phi_k(x,y) &= \frac{\left(\frac{2\,{}^{\mathfrak{I}}_{N}V_k(x,y)}{R+Z} + \frac{2\,{}^{\mathfrak{I}}_{E}V_k(x,y)}{R+Z} + \frac{2\,{}^{\mathfrak{I}}_{S}V_k(x,y)}{R+Z} + \frac{2\,{}^{\mathfrak{I}}_{W}V_k(x,y)}{R+Z}\right)}{\left(\frac{4}{R+Z}\right)} \\ &= \frac{{}^{\mathfrak{I}}_{\mathbf{N}}V_k(x,y) + {}^{\mathfrak{I}}_{\mathbf{E}}V_k(x,y) + {}^{\mathfrak{I}}_{\mathbf{S}}V_k(x,y) + {}^{\mathfrak{I}}_{\mathbf{W}}V_k(x,y)}{2} \end{aligned} \tag{5.20}$$

The pulses scattered are calculated:

$$\begin{pmatrix} {}^{\mathfrak{S}}V_{\mathbf{N}} \\ {}^{\mathfrak{S}}V_{\mathbf{E}} \\ {}^{\mathfrak{S}}V_{\mathbf{S}} \\ {}^{\mathfrak{S}}V_{\mathbf{W}} \end{pmatrix}_k = \begin{pmatrix} \rho & \tau & \tau & \tau \\ \tau & \rho & \tau & \tau \\ \tau & \tau & \rho & \tau \\ \tau & \tau & \tau & \rho \end{pmatrix} \begin{pmatrix} {}^{\mathfrak{I}}V_{\mathbf{N}} \\ {}^{\mathfrak{I}}V_{\mathbf{E}} \\ {}^{\mathfrak{I}}V_{\mathbf{S}} \\ {}^{\mathfrak{I}}V_{\mathbf{W}} \end{pmatrix}_k \tag{5.21}$$

The connection process is also just like in two dimensions, just for four variables instead of two:

$$\begin{aligned} {}^{\mathfrak{I}}_{\mathbf{N}}V_{k+1}(x,y) &= {}^{\mathfrak{S}}_{\mathbf{S}}V_k(x,y+1) \\ {}^{\mathfrak{I}}_{\mathbf{E}}V_{k+1}(x,y) &= {}^{\mathfrak{S}}_{\mathbf{W}}V_k(x-1,y) \\ {}^{\mathfrak{I}}_{\mathbf{S}}V_{k+1}(x,y) &= {}^{\mathfrak{S}}_{\mathbf{N}}V_k(x,y-1) \\ {}^{\mathfrak{I}}_{\mathbf{W}}V_{k+1}(x,y) &= {}^{\mathfrak{S}}_{\mathbf{E}}V_k(x+1,y) \end{aligned} \tag{5.22}$$

Four now the directional identifiers N, E, S and W have been used to make the maths easier to understand. It is of great benefit to use direction numbers instead for higher order models, since then the mathematical formulae can be expressed in a much more condensed fashion.

The nodal voltage in the three dimensional link-line system can thus be expressed in a very simple expression which is nonetheless equivalent to equation 5.20:

$$\phi_k(x,y,z) = \frac{1}{3}\sum_{j=1}^{6} {}^{\mathrm{J}}_{j}V_k(x,y,z) \tag{5.23}$$

The reflection coefficient can be derived similar to equation 5.19:

$$\rho = \frac{3R-2Z}{3R+3Z} \quad \text{with} \quad 5\tau = 1-\rho \tag{5.24}$$

The scattering process described in equation 5.21 can also easily be extended:

$$\begin{pmatrix} {}^{\mathrm{S}}V_1 \\ {}^{\mathrm{S}}V_2 \\ {}^{\mathrm{S}}V_3 \\ {}^{\mathrm{S}}V_4 \\ {}^{\mathrm{S}}V_5 \\ {}^{\mathrm{S}}V_6 \end{pmatrix}_k = \begin{pmatrix} \rho & \tau & \tau & \tau & \tau & \tau \\ \tau & \rho & \tau & \tau & \tau & \tau \\ \tau & \tau & \rho & \tau & \tau & \tau \\ \tau & \tau & \tau & \rho & \tau & \tau \\ \tau & \tau & \tau & \tau & \rho & \tau \\ \tau & \tau & \tau & \tau & \tau & \rho \end{pmatrix} \begin{pmatrix} {}^{\mathrm{J}}V_1 \\ {}^{\mathrm{J}}V_2 \\ {}^{\mathrm{J}}V_3 \\ {}^{\mathrm{J}}V_4 \\ {}^{\mathrm{J}}V_5 \\ {}^{\mathrm{J}}V_6 \end{pmatrix}_k \tag{5.25}$$

The connection process is the point where direction is important again. It does not matter how they are assigned, as long it is consistent. In this document 1 and 2 are assigned to the x-axis, 3 and 4 to the y-axis and 5 and 6 to the z-axis. The lower number is in the negative and the higher towards the positive direction. Thus we get:

$$\begin{aligned}
{}^{\mathrm{J}}_{1}V_{k+1}(x,y,z) &= {}^{\mathrm{S}}_{2}V_k(x-1,y,z) \\
{}^{\mathrm{J}}_{2}V_{k+1}(x,y,z) &= {}^{\mathrm{S}}_{1}V_k(x+1,y,z) \\
{}^{\mathrm{J}}_{3}V_{k+1}(x,y,z) &= {}^{\mathrm{S}}_{4}V_k(x,y-1,z) \\
{}^{\mathrm{J}}_{4}V_{k+1}(x,y,z) &= {}^{\mathrm{S}}_{3}V_k(x,y+1,z) \\
{}^{\mathrm{J}}_{5}V_{k+1}(x,y,z) &= {}^{\mathrm{S}}_{6}V_k(x,y,z-1) \\
{}^{\mathrm{J}}_{6}V_{k+1}(x,y,z) &= {}^{\mathrm{S}}_{5}V_k(x,y,z+1)
\end{aligned} \tag{5.26}$$

Two- and three-dimensional link-resistor nodes

The scattering in a two dimensional link-resistor node can be easily described as:

$$\begin{pmatrix} {}^{\mathrm{S}}V_1 \\ {}^{\mathrm{S}}V_2 \\ {}^{\mathrm{S}}V_3 \\ {}^{\mathrm{S}}V_4 \end{pmatrix}_{k+1} = \frac{1}{2}\begin{pmatrix} -1 & 1 & 1 & 1 \\ 1 & -1 & 1 & 1 \\ 1 & 1 & -1 & 1 \\ 1 & 1 & 1 & -1 \end{pmatrix} \begin{pmatrix} {}^{\mathrm{J}}V_1 \\ {}^{\mathrm{J}}V_2 \\ {}^{\mathrm{J}}V_3 \\ {}^{\mathrm{J}}V_4 \end{pmatrix}_{k+1} \tag{5.27}$$

This equation can easily be extended to three dimensions:

$$\begin{pmatrix} {}^{\mathrm{S}}V_1 \\ {}^{\mathrm{S}}V_2 \\ {}^{\mathrm{S}}V_3 \\ {}^{\mathrm{S}}V_4 \\ {}^{\mathrm{S}}V_5 \\ {}^{\mathrm{S}}V_6 \end{pmatrix}_{k+1} = \frac{1}{3} \begin{pmatrix} -2 & 1 & 1 & 1 & 1 & 1 \\ 1 & -2 & 1 & 1 & 1 & 1 \\ 1 & 1 & -2 & 1 & 1 & 1 \\ 1 & 1 & 1 & -2 & 1 & 1 \\ 1 & 1 & 1 & 1 & -2 & 1 \\ 1 & 1 & 1 & 1 & 1 & -2 \end{pmatrix} \begin{pmatrix} {}^{\mathrm{J}}V_1 \\ {}^{\mathrm{J}}V_2 \\ {}^{\mathrm{J}}V_3 \\ {}^{\mathrm{J}}V_4 \\ {}^{\mathrm{J}}V_5 \\ {}^{\mathrm{J}}V_6 \end{pmatrix}_{k+1} \tag{5.28}$$

The linking for the two dimensional link-resistor system is described by the following set of equations:

$$\begin{aligned} {}^{\mathrm{J}}_{1}V_{k+1}(x,y) &= \rho\, {}^{\mathrm{S}}_{1}V_k(x,y) + \tau\, {}^{\mathrm{S}}_{2}V_k(x-1,y) \\ {}^{\mathrm{J}}_{2}V_{k+1}(x,y) &= \rho\, {}^{\mathrm{S}}_{2}V_k(x,y) + \tau\, {}^{\mathrm{S}}_{1}V_k(x+1,y) \\ {}^{\mathrm{J}}_{3}V_{k+1}(x,y) &= \rho\, {}^{\mathrm{S}}_{3}V_k(x,y) + \tau\, {}^{\mathrm{S}}_{4}V_k(x,y-1) \\ {}^{\mathrm{J}}_{4}V_{k+1}(x,y) &= \rho\, {}^{\mathrm{S}}_{4}V_k(x,y) + \tau\, {}^{\mathrm{S}}_{3}V_k(x,y+1) \end{aligned} \tag{5.29}$$

In the previous and the three dimensional linking equation (Equation 5.31, ρ and τ are defined as follows:

$$\begin{aligned} \rho &= \tfrac{R}{R+Z} \\ \tau &= \tfrac{Z}{R+Z} \end{aligned} \tag{5.30}$$

The three dimensional linking equation is very similar to the two dimensional version (Equation 5.29):

$$\begin{aligned} {}^{\mathrm{J}}_{1}V_{k+1}(x,y,z) &= \rho\, {}^{\mathrm{S}}_{1}V_k(x,y,z) + \tau\, {}^{\mathrm{S}}_{2}V_k(x-1,y,z) \\ {}^{\mathrm{J}}_{2}V_{k+1}(x,y,z) &= \rho\, {}^{\mathrm{S}}_{2}V_k(x,y,z) + \tau\, {}^{\mathrm{S}}_{1}V_k(x+1,y,z) \\ {}^{\mathrm{J}}_{3}V_{k+1}(x,y,z) &= \rho\, {}^{\mathrm{S}}_{3}V_k(x,y,z) + \tau\, {}^{\mathrm{S}}_{4}V_k(x,y-1,z) \\ {}^{\mathrm{J}}_{4}V_{k+1}(x,y,z) &= \rho\, {}^{\mathrm{S}}_{4}V_k(x,y,z) + \tau\, {}^{\mathrm{S}}_{3}V_k(x,y+1,z) \\ {}^{\mathrm{J}}_{5}V_{k+1}(x,y,z) &= \rho\, {}^{\mathrm{S}}_{5}V_k(x,y,z) + \tau\, {}^{\mathrm{S}}_{6}V_k(x,y,z-1) \\ {}^{\mathrm{J}}_{6}V_{k+1}(x,y,z) &= \rho\, {}^{\mathrm{S}}_{6}V_k(x,y,z) + \tau\, {}^{\mathrm{S}}_{5}V_k(x,y,z+1) \end{aligned} \tag{5.31}$$

The two- and three-dimensional equations for the nodal potential are thus:

$$\phi_{k+1}(x,y) = \frac{1}{2}\sum_{j=1}^{4} {}^{\mathrm{J}}_{j}V_{k+1}(x,y) \tag{5.32}$$

and

$$\phi_{k+1}(x,y,z) = \frac{1}{3}\sum_{j=1}^{6} {}^{\mathrm{J}}_{j}V_{k+1}(x,y,z) \tag{5.33}$$

5.3 Effects of different Parameters on simulated output

5.3.1 Dimensionality

The dimensionality has a profound impact on the shape of the generated curve. Since a different formula is used to describe each of the possible dimensionalities of a spin diffusion system, the shape of the curve is characteristic. The simulated system also follows this nicely (see Figure 5.9). Due to the fact that the limiting signal intensity is determined by the ratio of the two phases, and the fact that an even sided shape was simulated, it was not possible to make the limiting signal intensity the same for the three dimensionalities, due to the fact that only integers may be chosen as the sizes of the system and the maximum system size is limited by the computers memory.

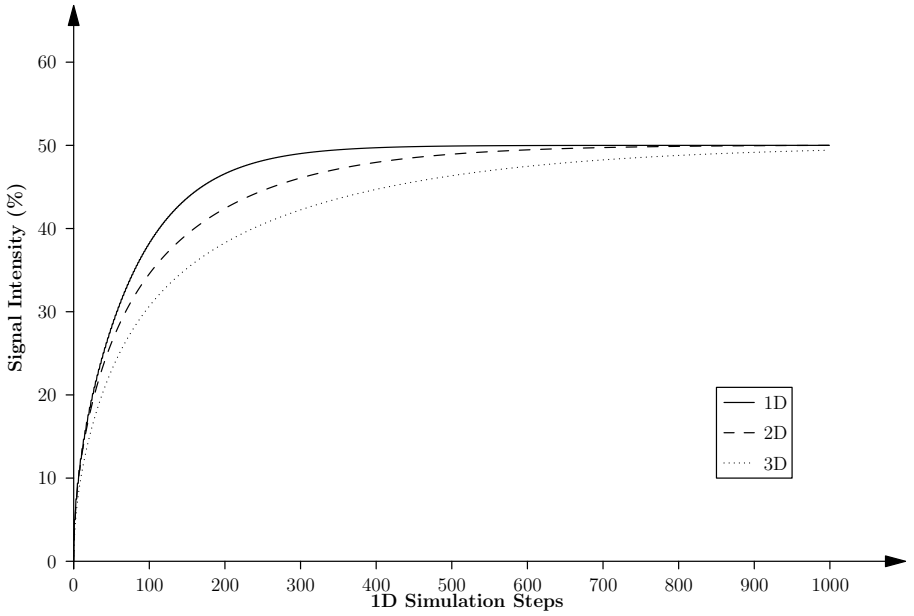


Figure 5.9: Effect of different dimensionality on simulated diffusion behavior. Parameters in Table E.1.

In Figure 5.9 one can see the energy of the spins of the material the energy is diffusing into.

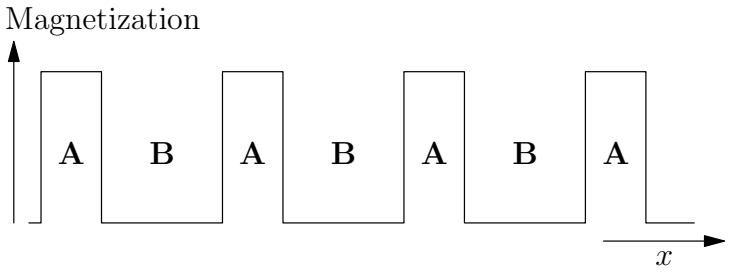


Figure 5.10: Diagram showing an example of a 1D lattice

The material is sitting in one corner/edge of the simulated system. All simu-

lation parameters used to generate this graph are listed in Appendix E.

5.3.2 Different Box Sizes

Using different box sizes when simulating will result in different behavior. To date only squares and cubes have been simulated as region **B**. A summary for 1D systems can be found in Figure 5.11.

The volume percentage of region **A** nuclei has the most obvious effect on the maximum amplitude of the simulated signal. But it also has an effect on the time it takes to reach that maximum, which occurs when the system is close to equilibrium.

The size of the *Box*, which contains the region **B** cells we are interested in, is a convenient size in terms of the simulation software. However, the percentage of the whole simulated "volume" is different depending on the dimensionality of the simulated system. The relation between region **B** partial volume $V_{\mathbf{B}}$, side length of the simulated system l_{Total} (if all sides are the same length) and the side length of the *Box* l_{Box} (if all sides are the same length) for an n-dimensional system is:

$$V_{\mathbf{B}} = \frac{l_{\text{Total}}^n}{l_{\text{Box}}^n}. \tag{5.34}$$

The volume percentage for different box sizes for 1D, 2D and 3D systems in a simulated system of the size 50 can be seen in Table 5.1.

The graphs for 2D and 3D systems can be found in Figure 5.12 and Figure 5.13 respectively.

Box Size	1D	2D	3D
5	10%	1%	0.1%
10	20%	4%	0.8%
15	30%	9%	2.7%
25	50%	25%	12.5%
35	70%	49%	34.3%
40	80%	64%	51.2%
45	90%	81%	72.9%

Table 5.1: Box sizes and volume percentages for different dimensionalities.

It can clearly be seen that when the volume of the region **B** phase decreases, the time needed for the diffusion increases. This is because the resistance value in region **A** is set to 100 whereas the resistance value of the region **B** phase is set to 10. A high resistance will lead to a slower equilibration within the region resulting in greater inhomogeneity, whereas a lower resistance will result in quicker equilibration within that region. This means that since region **A** has a high resistance, and region **B** has a low resistance, the magnetization from the edge of region **A** is quickly siphoned of and spread over region **B**. The slow diffusion within region **A** to the edge now becomes the significant factor on the time needed to reach total equilibrium.

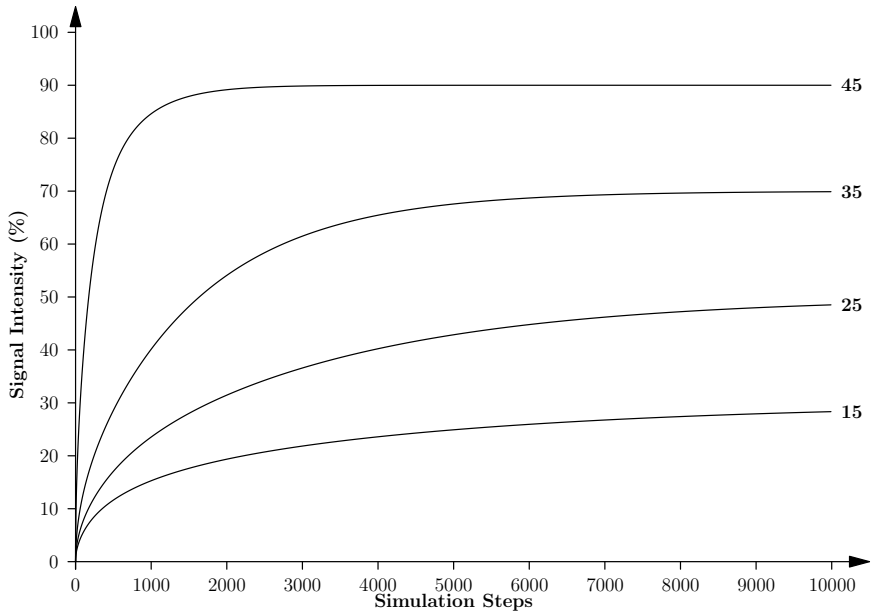


Figure 5.11: Effect of different box sizes on diffusion behavior in single dimensional systems. Parameters in Table E.2

5.3.3 Differences between Link-Line and Link-Resistor simulations

Even though simulation using link-line- or link-resistor nodes is in its implementation quite different, the results are remarkably similar if enough steps are simulated.

In terms of spin diffusion, a link-line node is centered on the nucleus and the magnetization stored within it. A link-resistor node is centered on the dipole-dipole interaction and the the magnetization contained within the field between the nuclei around it. In my humble opinion a link-line node is a more intuitive representation of a spin diffusion system, because in the NMR context the elevated magnetization of the nuclei is being measured, not the magnetic field between them.

5.3.4 Simulation of different geometric shapes

Circle vs. Square

In this set of simulations a 200×200 sized two dimensional system was simulated. The capacitance was set to 100 for every cell, and the resistance to 100 for the bulk material (region **A**) and 10 for the material in region **B**. The square was placed in one corner and had a size of 141 filling approximately 49.7% of the simulated system.

The aim was a filling of approximately 50%, that would mean that the size of each side would have to be 141.421. But since only integer values are allowed as box sizes, a value of 141 was chosen since it was closest.

For the distance from the origin is calculated by Pythagoras (equation 5.35)

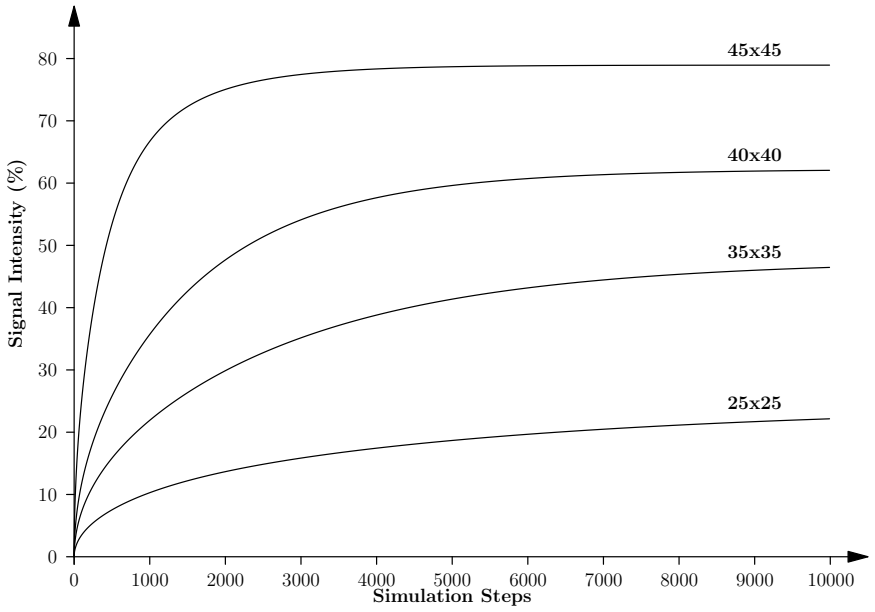


Figure 5.12: Effect of different box sizes on diffusion behavior in two dimensional systems. Parameters in Table E.3

$$d = \sqrt{x^2 + y^2} \tag{5.35}$$

and any cell which has a distance smaller or equal to a set radius is included in the region **B** material, each other cell is bulk material. The value of the radius has been chosen so that the area covered is equal to that of the square. To do this equation 5.36 was used.

$$r = \sqrt{\frac{4x^2}{\pi}} \tag{5.36}$$

The size of the simulated system was chosen to be quite large, since then the approximation of the circle in the square geometry of the simulated system is closer to a real sphere than in a smaller system.

The results show that there is little difference between the square and circular phase geometries. It can however be seen that there is actually a small difference. The difference between the two curves tends towards zero as the two systems tend towards equilibrium.

In the square system the diffusion is slightly faster at first. This is consistent, since the corners of the square reach further into the bulk material allowing a little more energy to diffuse into it at first. Over time however this difference gets smaller, because the systems would be the same at equilibrium.

Sphere vs. Cube

These simulations were done using a $100 \times 100 \times 100$ system. This size was chosen as a compromise between high simulation resolution and the amount of time needed to run a set of simulations. The capacitance was again 100 for the

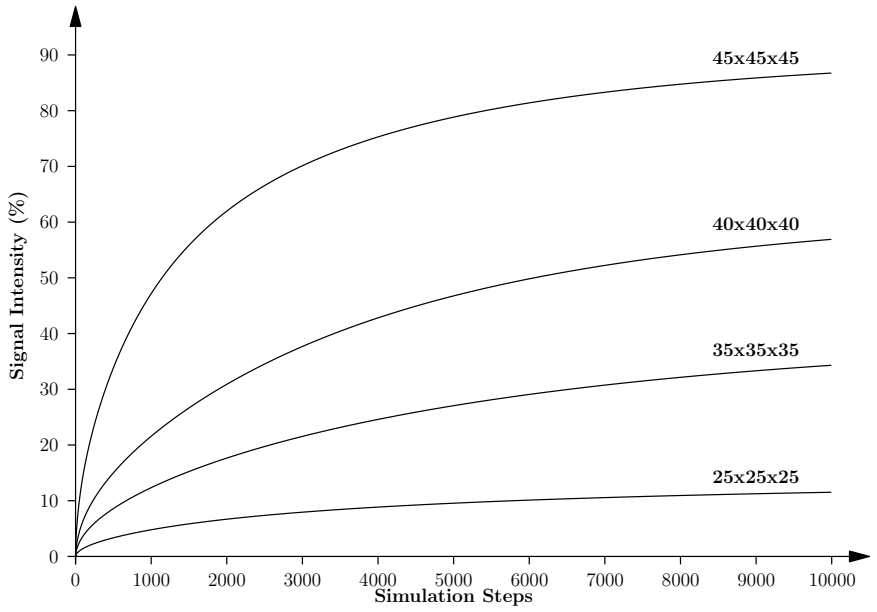


Figure 5.13: Effect of different box sizes on diffusion behavior in three dimensional systems. Parameters in Table E.4

whole system and the resistance was set to 100 for the bulk material and 10 for region **B**.

The size of an edge of the cube was chosen as 79, since it is closest to the ≈ 79.370 that would be needed for a 50% filling. The radius of the sphere was set (using equation 5.37)so that two systems have the same volume and was set to 98.015.

$$r = \sqrt[3]{\frac{6x^3}{\pi}} \tag{5.37}$$

The results as seen in figure 5.15 are slightly more spectacular than for the circle and square system, since the difference of the two curves is a bit more pronounced. But other than the slower diffusion and the slightly different base curve shape, the results are very similar to the circle and square system.

5.3.5 Comparing the Simulation Results with Analytical Solutions

The Analytical Solution

When solving spin diffusion analytically, one needs to consider the diffusion equation for z-magnetization $M(\mathbf{r}, t_m)$:

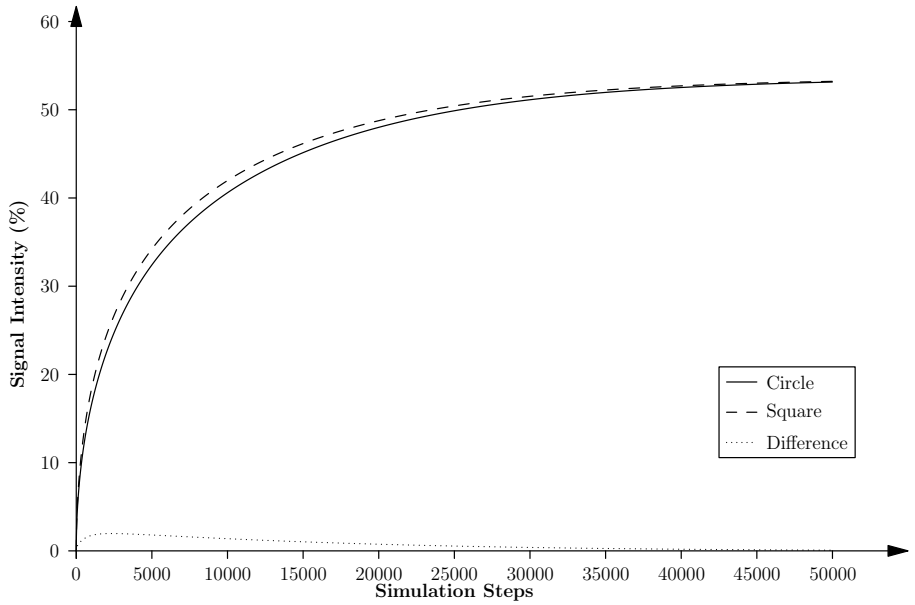


Figure 5.14: Effect of different geometries in two dimensional systems. Parameters in Table E.5

$$\begin{aligned}\frac{\partial M(\mathbf{r},t_m)}{\partial t_m} &= \nabla\cdot\{D(\mathbf{r})\nabla M(\mathbf{r},t_m)\}\\ &= \frac{\partial}{\partial x}\left\{D(\mathbf{r})\frac{\partial}{\partial x}M(\mathbf{r},t_m)\right\}\\ &+ \frac{\partial}{\partial y}\left\{D(\mathbf{r})\frac{\partial}{\partial y}M(\mathbf{r},t_m)\right\}\\ &+ \frac{\partial}{\partial z}\left\{D(\mathbf{r})\frac{\partial}{\partial z}M(\mathbf{r},t_m)\right\}\end{aligned} \tag{5.38}$$

The aim is to solve this equation for certain initial conditions (as in [4, ch 13.3.2]). The initial conditions used for demonstration purposes are a constant diffusivity $D(\mathbf{r}) = D$ and a simple periodic lamellar morphology. Since only the direction perpendicular to the lamellae is relevant, the result in an initial magnetization as depicted in Figure 5.16 and only a one-dimensional equation that needs to be solved.

The boxes in the periodic array of boxes described in Figure 5.16 have a width d_A and height M_0. This array can be described by the superposition of spatial δ-functions. The initial magnetization of a single lamella of width d_A and centered around $x = 0$ can thus be described with:

$$M_{n=0}(x, t_m = 0) = M_0 \int_{-1/2\,d_A}^{1/2\,d_A} \mathrm{d}\tilde{x}\ \delta(\tilde{x} - x) \tag{5.39}$$

which changes over time into

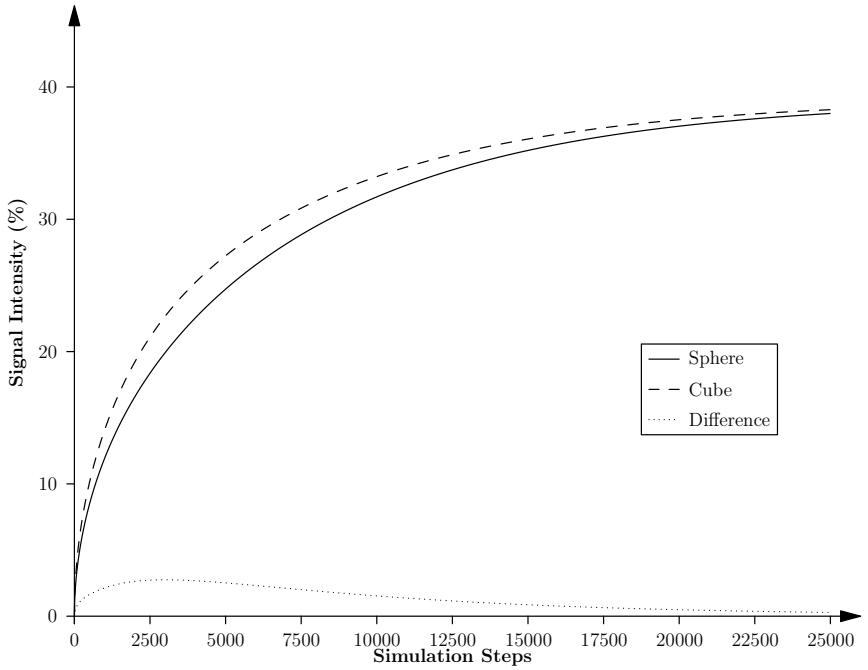


Figure 5.15: Effect of different geometries in three dimensional systems. Parameters in Table E.6

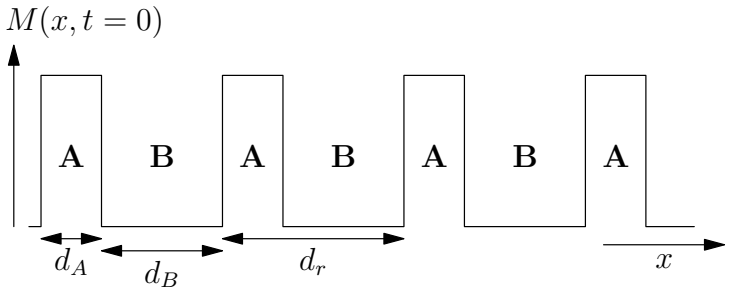


Figure 5.16: The initial magnetization for a lamellar morphology [4]

$$M_{n=0}(x, t_m) = \frac{M_0}{\sqrt{4\pi D t_m}} \int_{-1/2\, d_A}^{1/2\, d_A} \mathrm{d}\tilde{x}\; e^{-\frac{(\tilde{x}-x)^2}{4Dt_m}}. \tag{5.40}$$

For the n^{th} lamella, centered on the lamella and the integration limits as $nd_r - \frac{1}{2}d_A$ and $nd_r + \frac{1}{2}d_A$ the integral gives:

$$M_n(x, t_m) = \frac{M_0}{2}\left\{\operatorname{erfc}\left(\frac{nd_r - 1/2\, d_A - x}{\sqrt{4Dt_m}}\right) - \operatorname{erfc}\left(\frac{nd_r + 1/2\, d_A - x}{\sqrt{4Dt_m}}\right)\right\} \tag{5.41}$$

$\operatorname{erfc}(x)$ is the complement of the error function $\operatorname{erf}(x)$ as defined in Table 5.2. The error function is the integral of the Gaussian function.

Since the magnetization behaves the same for each unit cell of size d_r, it is sufficient to consider only a single one. For region **B** we chose the region from $\frac{1}{2}d_A$ to $\frac{1}{2}d_A + d_B$:

$$\operatorname{erf}(x) := \frac{2}{\sqrt{\pi}} \int_0^x e^{-\tilde{x}^2}\, \mathrm{d}\tilde{x} \qquad \operatorname{erf}(x) = \begin{cases} -1 & : \quad x = -\infty \\ 0 & : \quad x = 0 \\ +1 & : \quad x = \infty \end{cases}$$

$$\operatorname{erf}(-x) = -\operatorname{erf}(x)$$

$$\operatorname{erfc}(x) := 1 - \operatorname{erf}(x) \qquad \operatorname{erfc}(x) = \begin{cases} 2 & : \quad x = -\infty \\ 1 & : \quad x = 0 \\ 0 & : \quad x = \infty \end{cases}$$

$$= \frac{2}{\sqrt{\pi}} \int_x^\infty e^{-\tilde{x}^2}\, \mathrm{d}\tilde{x} \qquad \operatorname{erfc}(-x) = 1 + \operatorname{erf}(x) = 2 - \operatorname{erfc}(x)$$

$$\operatorname{ierfc}(x) := \int_x^\infty \operatorname{erfc}(\tilde{x})\, \mathrm{d}\tilde{x} \qquad \operatorname{ierfc}(x) = \begin{cases} \approx -2x & : \quad x < -3 \\ \frac{1}{\sqrt{\pi}} & : \quad x = 0 \\ 0 & : \quad x = \infty \end{cases}$$

$$= \frac{1}{\sqrt{\pi}} e^{-x^2} - x \operatorname{erfc}(x)$$

Table 5.2: Definition of the error function $\operatorname{erf}(x)$ and related functions

$$\begin{aligned} I_B(t_m) &= \int_{1/2\, d_A + d_B}^{1/2\, d_A} \sum_{n=-N}^{N} M_n(x, t_m)\ \mathrm{d}x \\ &= \frac{M_0}{2} \sum_{n=1}^{N} \sqrt{4Dt_m} \left\{ -\operatorname{ierfc}\left(\frac{nd_r - d_A}{\sqrt{4Dt_m}}\right) + \operatorname{ierfc}\left(\frac{nd_r}{\sqrt{4Dt_m}}\right) \right. \\ &\quad \left. + \operatorname{ierfc}\left(\frac{(n-1)d_r}{\sqrt{4Dt_m}}\right) - \operatorname{ierfc}\left(\frac{nd_r - d_B}{\sqrt{4Dt_m}}\right) \right\}. \end{aligned} \tag{5.42}$$

In this equation we exploit the symmetry about $x = 1/2\, d_r$ to restrict the sum to $n \geq 1$.

For most practical purposes only the first terms of the near infinite sum (5.42) contribute to any real significance. Each term of the sum represents the contribution of a lamella a distance nd_r away from the detection region. Terms with large n only become relevant after a very long time has passed.

Terms with $n > n_c$ can be approximated as a semi-infinite region with an initial magnetization density of $M_0 \frac{d_A}{d_r}$ separated from the detection region by a distance $x_c = n_c d_r$. When using the solution $\operatorname{erfc}(\frac{x}{\sqrt{4Dt_m}})$ for a semi-infinite source we get:

$$
\begin{aligned}
I_{B,n>n_c} &\simeq M_0 \frac{d_A}{d_r} \int_{x_c}^{x_c+d_B} \mathrm{d}x \ \mathrm{erfc}\left(\frac{x}{\sqrt{4Dt_m}}\right) \\
&= M_0 \frac{d_A}{d_r} \sqrt{4Dt_m} \left\{ \mathrm{ierfc}\left(\frac{x_c}{\sqrt{4Dt_m}}\right) - \mathrm{ierfc}\left(\frac{x_c+d_B}{\sqrt{4Dt_m}}\right) \right\}.
\end{aligned} \tag{5.43}
$$

Using the terms $n = 0, \ldots, 4$ from (5.42) and the correction term (5.44) for a $n_c = 4$ good results are apparently obtained for arbitrary t_m according to [4, p419].

In the higher dimensional systems that are now introduced, this correctional term is left out for simplicity sake. If N is chosen high enough it is not absolutely necessary if very high values of t_m are not of interest.

From this point onward the mathematics differs slightly from the formulae presented in *Multidimensional Solid-State NMR and Polymers* [4]. This is because the attempt to plot the curves using the formulae in the book resulted in regular curves when setting the variables to the ones used in the text, but something very different when trying to change some of these parameters.

It was decided by the author that solving the equations himself was a better way to acquire the formulae for an analytical solution to spin diffusion. The result is still quite close to what is presented in the book and is based quite substantially on it.

However it is not practical to transfer the bounds used for the integral ($\frac{1}{2}d_A \Rightarrow \frac{1}{2}d_A + d_B$) to higher dimensions, thus we now subtract an integral over area d_B from an integral over area d_r:

$$
\begin{aligned}
I(t_m) &= M_0 \left(\int_{-1/2\,d_r}^{+1/2\,d_r} M(x,t_m)\,dx - \int_{-1/2\,d_A}^{+1/2\,d_A} M(x,t_m)\,dx \right) \\
&= \frac{M_0}{d_r}\sqrt{4Dt_m} \sum_{n=-N}^{N} \Bigg[\Bigg\{ \mathrm{ierfc}\left(\frac{nd_r - 1/2\,d_A - 1/2\,d_r}{\sqrt{4Dt_m}}\right) \\
&\qquad - \mathrm{ierfc}\left(\frac{nd_r - 1/2\,d_A + 1/2\,d_r}{\sqrt{4Dt_m}}\right) - \mathrm{ierfc}\left(\frac{nd_r + 1/2\,d_A - 1/2\,d_r}{\sqrt{4Dt_m}}\right) \\
&\qquad + \mathrm{ierfc}\left(\frac{nd_r + 1/2\,d_A + 1/2\,d_r}{\sqrt{4Dt_m}}\right) \Bigg\} \\
&\quad - \Bigg\{ \mathrm{ierfc}\left(\frac{nd_r - d_A}{\sqrt{4Dt_m}}\right) - 2\,\mathrm{ierfc}\left(\frac{nd_r}{\sqrt{4Dt_m}}\right) \\
&\qquad + \mathrm{ierfc}\left(\frac{nd_r + d_A}{\sqrt{4Dt_m}}\right) \Bigg\} \Bigg]
\end{aligned} \tag{5.44}
$$

To take this into the second dimension and subsequently into the third, we will just write the integral. The magnetic field for a ϵ-dimensional system would be:

$$M(\mathbf{r}, t_m) = \frac{1}{d_r} \sum_{n=-N}^{N} \left\{ \operatorname{erfc}\left(\frac{n d_r - 1/2\, d_A - r}{\sqrt{4 D t_m}} \right) - \operatorname{erfc}\left(\frac{n d_r + 1/2\, d_A - r}{\sqrt{4 D t_m}} \right) \right\}. \tag{5.45}$$

To extend this equation in ϵ dimensions all we have to do is raising it to the power of ϵ like this:

$$I(t_m) = M_0 \left(\int_{-1/2\, d_r}^{+1/2\, d_r} M(x, t_m)\ \mathrm{d}x \right)^{\epsilon} - M_0 \left(\int_{-1/2\, d_A}^{+1/2\, d_A} M(x, t_m)\ \mathrm{d}x \right)^{\epsilon} \tag{5.46}$$

This is now the generic analytical solution and for $\lim N \to \infty$ this solution is true even at arbitrarily high t_m. However on the time-scales used in the simulated cases in this work, $N = 4$ is quite sufficient.

However keep in mind that equation 5.46 was derived by the author after finding that there must be an error in the equations 13.21 and 13.22 from [4, p419].

A different approach to the analytical solution

Cheung et al. describe a different approach to the analytical solution in their paper [18]. Instead of using an infinite space filled with regular lamellae, they use a space with length L and without any magnetization transfer outside the system.

However instead of regularly spaced lamellae, they assume that the spacing between the lamellae is random and follows a Poisson distribution (eq. 5.47).

$$P(b) = 1/\bar{b}\ e^{-b/\bar{b}} \tag{5.47}$$

With such a distribution of lamellae they get the following equation to describe the magnetization in area **A**:

$$\varphi(t) = \exp\left(\frac{Dt}{\bar{b}^2} \right) \operatorname{erfc}\left(\sqrt{\frac{Dt}{\bar{b}^2}} \right) \tag{5.48}$$

To describe area **B**, they simply use $I_B(t) = 1 - \varphi(t)$. The system can be extended to higher dimensional cases just as easily by multiplying several one dimensional equations yielding

$$I_{B,\epsilon} = 1 - \varphi_x(t)\varphi_y(t)\varphi_z(t) \tag{5.49}$$

for the three dimensional case, with

$$\varphi_\alpha(t) = \exp\left(\frac{Dt}{\bar{b_\alpha}^2} \right) \operatorname{erfc}\left(\sqrt{\frac{Dt}{\bar{b_\alpha}^2}} \right) \quad \text{for} \quad \alpha = x, y, z. \tag{5.50}$$

However the solution acquired using this equation is not very comparable to the solutions offered by equation 5.46 or the simulations. This is mainly due to

the random spacing of the lamellae. If one really wanted to use the analytical solution described by equation 5.46 to model infinite lamellae spaced randomly following a Poisson distribution, one would have to do something like this:

$$\int_0^\infty P(d_B)I(t_m)\,\mathrm{d}d_B \tag{5.51}$$

Plotting both analytical solutions we can clearly see the profound difference between the two with $d_A = d_B = 1$, $D = .1$, $M_0 = 1$ and $b = \frac{1}{3}$ and scaling elation 5.50 by multiplying it by $\frac{3}{4}$ to make sure that both equations reach the same value at $\lim t \to \infty$, we get figure 5.17. Showing clearly the difference between the two analytical solutions.

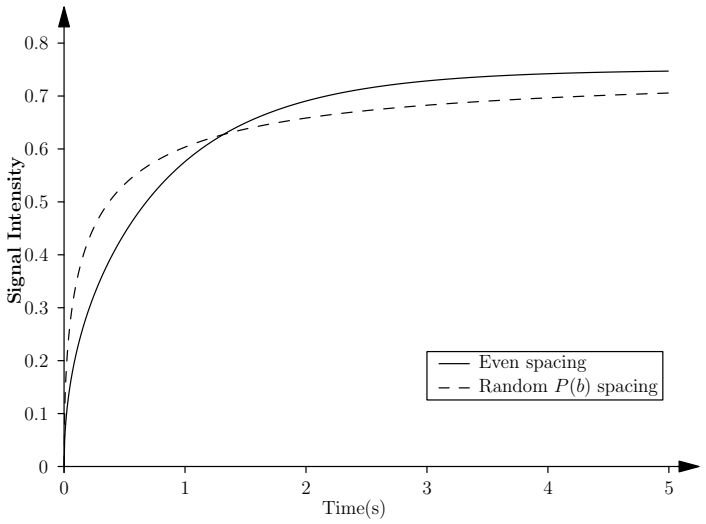


Figure 5.17: Comparison of the two analytical models

Calibration of the software using the analytical solution

The software was calibrated using the formulae from the analytical solution. The relation of the parameters was found to be:

$$Cd_r^2 = \mathcal{R}PD\Delta t \tag{5.52}$$

with $C = 2.5 \times 10^{-3}$. Where d_r represents the size of one simulated cell in meters, not the size of the simulation area. Since the walls of the simulated area are reflecting, the size of the simulated area is often not equal to d_r. $\mathcal{R}$ represents the ratio $\frac{R}{Z}$ of the two simulations parameters R and Z; P represents the number of points used to represent d_r; Δt represents the duration in seconds for one simulation step and D represents the diffusion coefficient in $\mathrm{m^2 s^{-1}}$.

Equation 5.52 is able to predict one of the unknowns d_r, Δt or R if the following conditions are met:

1. The simulated area has the same diffusion coefficient over its whole area
2. The dimensions of the simulated area (d_r) are equal

3. The dimensions of area **A** are equal

in all other cases it may only be useful as a rough guideline. Since the analytical solution used is limited by these constraints, it was not possible to formulate a calibration function for any arbitrary system.

To archive optimal simulation conditions the fraction $\frac{R}{Z}$ should be neither too big nor too small. If it is too small $\lesssim 0.1$, the diffusion happens too quickly pulses are running back and forth between the boundaries resulting in a low frequency oscillation in the simulated output. If it is too big $\gtrsim 1$, a high frequency oscillation which occurs at the interface can be found in the simulated output.

Comparing the Analytical Solution with the Simulations

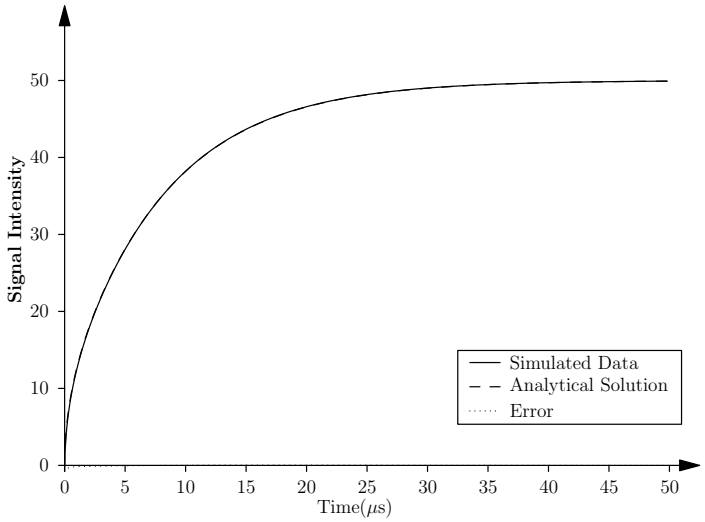


Figure 5.18: Comparison of a 1D analytical solution for spin diffusion with the simulation. Simulation parameters in Table E.7, analytical parameters in Table E.8

Comparing the 1D analytical solution with a link-line TLM-simulation yields a very high similarity in the curves. As seen in Figures 5.18, 5.19 and 5.20 the simulated curves and the analytical solutions fit so well that they nearly overlap completely.

This proves that the analytical solution in Formula 5.46 describes the simulated data nicely. Due to the lack of measured data it is not possible to be entirely sure that this describes real spin diffusion systems nicely, it does however hint towards this since two different ways to obtain this data agree very nicely.

5.4 Experimental

As a first step of the spin diffusion experiment a solid-echo spectrum is made to evaluate the decay-time of the two different signals. This is done to choose a selection-delay which makes it possible to have one phase magnetized, while the magnetization of the other phase has already decayed.

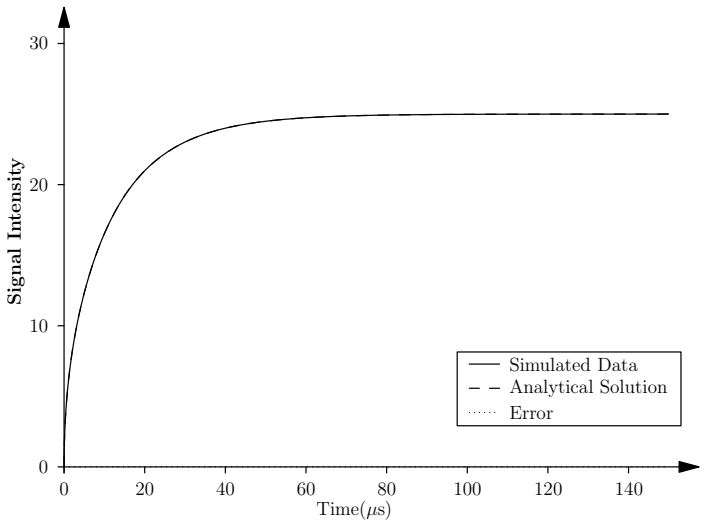


Figure 5.19: Comparison of a 2D analytical solution for spin diffusion with the simulation Parameters in Table E.9, analytical parameters in Table E.10

Since the two phases of the sample have different decay rates, it is possible to selectively have one of them decay before applying a pulse which turns the magnetization back into the z-plane.

After a short delay (the *diffusion time*), which is increased slightly for each iteration of the experiment, the magnetization is brought back into the observable plane, and the amplitudes of the two components are compared by fitting a function to the resulting FID.

This experimental procedure ideally requires that the spectrum of the sample only has one symmetric peak, and that the two phases are both present in this one peak, at the same frequency. The receiver is then tuned to the exact frequency of this signal, so that the FID resembles an exponential decay curve.

The amplitudes of the two components are then plotted against diffusion time.

5.4.1 Extracting the magnetization components from the FID

The following formula describes a basic (amorphous) free induction decay for the solid-echo assuming a standard Gaussian decay:

$$\sigma = e^{-\frac{\tau^2 t^2}{2}} \tag{5.53}$$

Where τ is the decay constant and t is the passage of time.

When fitting using this function we have two unknowns. The amplitude of the signal and the decay constant. These two unknowns have been designated A and B respectively.

$$\sigma = Ae^{-\frac{B^2 \cdot t^2}{2}} \tag{5.54}$$

When dealing with the phases, the resulting signal is the sum of two exponential decays, each with a different decay are constant. Such a sum of two signals would be described as

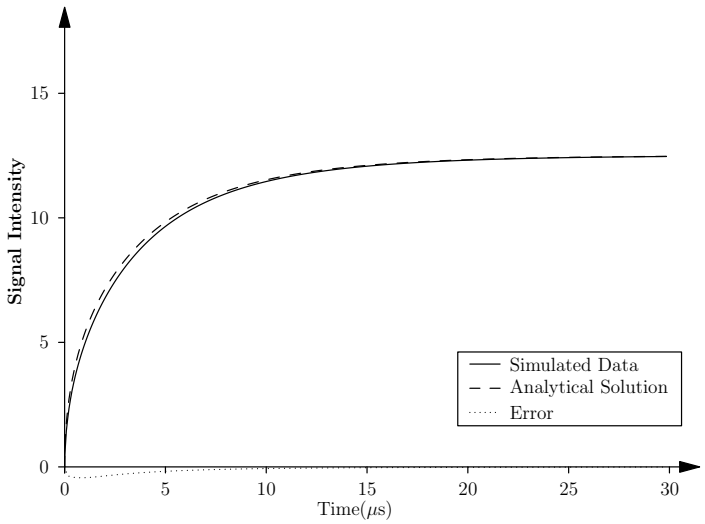


Figure 5.20: Comparison of a 3D analytical solution for spin diffusion with the simulation Parameters in Table E.11, analytical parameters in Table E.12

$$\sigma = A_1 e^{-\frac{B_1^2 \cdot t^2}{2}} + A_2 e^{-\frac{B_2^2 \cdot t^2}{2}}. \tag{5.55}$$

When the signal is normalized against the maximum amplitude, $A_1 + A_2 = 1$ holds true at the point of maximum amplitude. This can only hold true if the beginning of the signal is very close to the maximum of the echo. Assuming this is the case, then the previously four dimensional problem can be reduced to a three dimensional problem:

$$\sigma = A e^{-\frac{B_1^2 \cdot t^2}{2}} + (1 - A) e^{-\frac{B_2^2 \cdot t^2}{2}} \tag{5.56}$$

5.4.2 Experimental complications with the real-life samples

In the real-life samples the spectrum was not a single symmetric peak. This resulted in a free induction decay which had a small frequency component. Some crystalline phases phases display such a behavior. The decay of such crystalline phases can be described with one of two formulae [19]:

$$\sigma = \quad \omega e^{-\frac{\tau^2 t^2}{2}} \tag{5.57}$$

$$\omega \in \quad \left[\frac{\sin(\theta t)}{\theta t}, \cos(\theta t)\right] \tag{5.58}$$

with the first is called the Abragam function[9] and the second the Pakes doublet. Thus with one amorphous and one crystalline component, the equation to be fitted would be

$$\psi = A_1 e^{-\frac{B_1^2 \cdot \tau^2}{2}} + A_2 \frac{\sin(C\tau)}{C\tau} e^{-\frac{B_2^2 \cdot \tau^2}{2}} \tag{5.59}$$

or

$$\psi = A_1 e^{-\frac{B_1^2 \cdot \tau^2}{2}} + A_2 \cos(C\tau) e^{-\frac{B_2^2 \cdot \tau^2}{2}}. \tag{5.60}$$

Both are five dimensional problems, which are more difficult to fit. Especially considering that the phase of the frequency component ω is only correct if τ is placed exactly on the maximum of the echo signal. If this is not the case, τ must be offset to compensate this phase difference, or an additional phase parameter must be introduced. Both measures would increase the problems dimensionality to six dimensions.

This does not necessarily have to be the case for the sample though. There could be two different signals occurring naturally within the sample. With a lower B_0 field these two signals would not be resolved and appear as one single signal and thus appear as a single signal.

Due to the small changes that need to be made to large variables (due to the huge impact on the sum of squares), there is a problem concerning the computer hardware. Modern computers are quite limited when it comes to high precision mathematics. This problem is further discussed in Appendix A.

Also nonlinear fitting over a multidimensional surface with many local minima is a very difficult problem if one is seeking for a global minimum. The algorithm is very likely to find one of the local minima and home in on it. So finding a global minimum is more a case of luck in finding the right starting values that lead to a slope going to the global minimum.

Additionally to this the phase of the receiver is also a factor. If the phase of the receiver is not optimal, the FID would have to be rephased in order to make the equations fit again.

5.4.3 Plotting the amplitudes against diffusion time

The amplitudes resulting from the FIDs are plotted against diffusion time to give a plot of the diffusion of magnetization from one phase to another. The exact shape of the resulting curve is characteristic of the dimensionality of the phases in the sample.

Phase dimensionality

The phase dimensionality depends on how the phases are aligned to one another. In a one dimensional sample the phases are flat planes going all through the bulk of the sample. Different phases are only encountered if one moves perpendicular to the planes. When moving in the other two dimensions the phase will always stay the same. Please refer to Figure 5.21 as an example.

A one dimensional sample does not necessarily need to repeat the phases like in the example, instead it could be two slaps joined through one plane or one phase sandwiched between others. The exact arrangement of the phases can have subtle effects, but the basic lineshape is the same for all one dimensional samples.

For a two dimensional system, phases change for movement in two out of three dimensions, but stay the same for movement in the third. A prominent example of this are rods of material inside another. Refer to Figure 5.22 as an example.

The rods could be square, hexagonal, triangular or round. As long as they only vary in two dimensions.

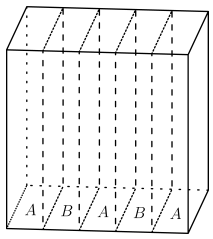


Figure 5.21: A one dimensional sample

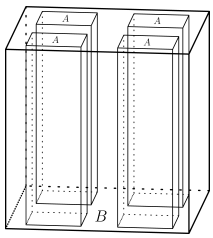


Figure 5.22: A two dimensional sample

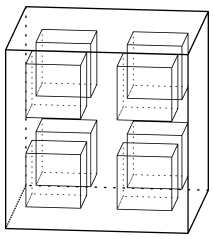


Figure 5.23: A three dimensional sample

A three dimensional sample is one in which the phases can change in any direction. This could be tiny grains inside a medium, a three dimensional grid of rods or just about any other distribution of phases throughout the sample. Refer to Figure 5.23 as an example.

One thing that might be counterintuitive is that large structures are not taken into account. For example a large sphere coated with a material of different phase would be a one dimensional system. This is because locally it is not distinguishable from a one dimensional system. Much like the fact that the earth might seem to be flat to the casual observer.

The real world can be more complicated

In the samples that were examined there was a complicated phase behavior, for the spin diffusion experiments, which made extracting amplitude data out of the FIDs extremely difficult or altogether impossible. This phase behavior had most effect on the off resonance peak, but also changed the lineshape in the spectra significantly.

Due to this anomalous behavior, very probably also due to the fact that the spectra of the samples had multiple peaks, proper analysis of the spin diffusion was made nearly impossible. There are some indications that there was indeed some spin diffusion going on, however this is not enough to get any kind of numerical data.

5.5 Comparison between provided Spin Diffusion data and simulations

Due to the complications in acquiring spin diffusion data, previous data was provided by Dr. Clayden. It was then attempted to overlay 2D TLM-Simulation data over the measured data-points.

In figure 5.24 the dots and crosses represent the measured data. They are regular and spin-echo spin diffusion datapoints respectively. The curves are extracts from a simulated 2D TLM-System. Both curves are scaled exactly the same, the dashed curve is just shifted to the right along the x-axis.

It can clearly be seen that the regular spin diffusion datapoints fit quite well onto the simulated curve. The spin-echo spin diffusion datapoints are significantly more entropic and as such difficult to fit, but the shifted curve fits reasonably

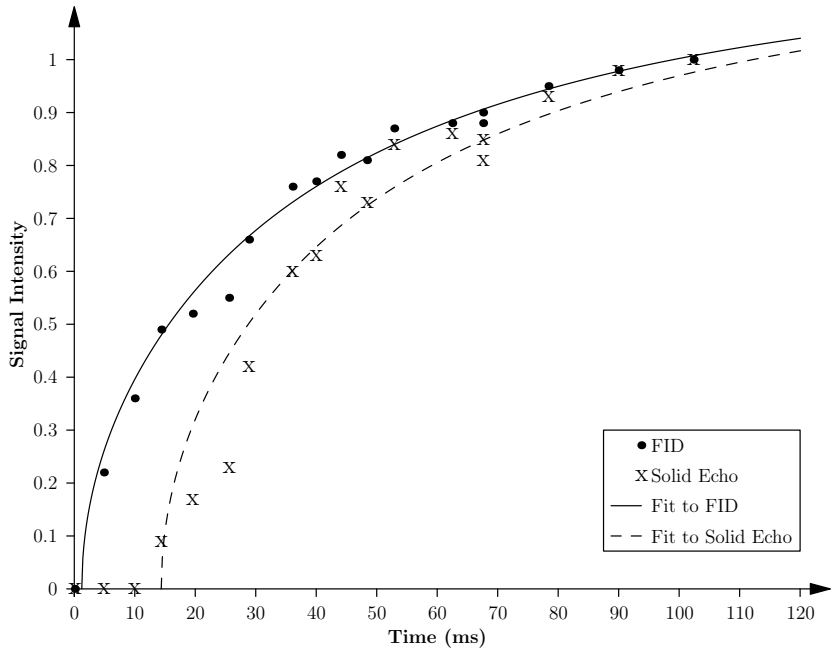


Figure 5.24: Spin Diffusion Data with overlaid TLM-Simulation data Parameters in Table E.13

well.

Since only 2D data is available, only the 2D mode of the TLM Simulation can be compared to real data. Unfortunately there was not enough time and too many problems to acquire enough data for a more in depth comparison.

5.6 Possible further studies

In my opinion it would be interesting to simulate if short rods or disks in a 3D sample. It would be interesting to see if the curve of simulated rods resembles a 2D sample, a 3D sample or something in between. With increasing ratio of rod length to rod radius the result should resemble the result of a 2D sample more and more.

A similar effect should occur for disks. With very extensive thin disks, the result should resemble a 1D sample more and more.

Simulating different shapes might be interesting too, for example comparing a square or cube to a circle or sphere.

Appendix A

Bibliography

[1] Melinda J. Duer. *Solid-State NMR Spectroscopy*. Wiley, 2004. ISBN 978-1-4051-0914-7

[2] Malcom H. Levitt. *Spin Dynamics*. Wiley, 2nd edition, 2009. ISBN 0-321-17385-6

[3] Donald de Cogan and Anne de Cogan. *Applied Numerical Modelling Engineers*. Oxford University Press, 1997. ISBN 978-01-9856-437-9

[4] K. Schmidt-Rohr and H. W. Spiess. *Multidimensional Solid-State NMR and Polymers*. Academic Pr Inc, 1994

[5] Klaus Schmidt-Rohr and Wolfgang Spiess. *Mutidimensional Solid-State NMR and Polymers*. Academic Press, 1994. ISBN 0-12-626630-1

[6] Sheng Qi, Peter Belton, Kathrin Nollenberger, Nigel Claden, Mike Reading, and Duncan Q. M. Craig. *Characterisation and Prediction of Phase Separation in Hot-Melt Extruded Solid Dispersions: A Thermal, Microscopic and NMR Relaxometry Study*. **Pharmaceutical Research**, 27(9):1869–1883, June 2010

[7] Chunli Gao, Mats Standing, Nikolaus Wellner, et al. *Plasticization of a Protein-Based Film by Glycerol: A Spectroscopic, Mechanical, and Thermal Study*. **J. Agric. Food Chem.**, 54:4611–4616, 2006

[8] Peter S. Belton and B. P. Hills. *The effects of diffusive exchange in heterogeneous systems on N.M.R. line shapes and relaxation processes*. **Molecular Physics**, 61(4):999–1018, 1987

[9] A. Abragam. *The Principles of Nuclear Magnetism*. Clarendon Press, 1961

[10] T. E. Bull. **Prog. NMR Spctrosc.**, 24:377–410, 1992

[11] M. Goldman and L. Shen. **Phys. Rev.**, 144:321, 1966

[12] J Clauss, K Schmidtrohr, and HW Spiess. *Determination of Domain Sizes in Heterogeneous Polymers by Solid-State NMR*. **Acta Polymerica**, 44(1):1–17, February 1993. ISSN 0323-7648

[13] Shu Xu, Frederik Klama, Henrietta Ueckermann, Jurian Hoogewerff, Nigel Clayden, and Thomas Nann. *Optical and Surface Characterisation of Capping Ligands in the Preparation of InP and InP/ZnS Quantum Dots.* **Sci. Adv. Mater.**, 1:125–137, 2009

[14] *WebElements Periodic Table of the Elements — Zinc — NMR data*, June 2010
URL: `http://www.webelements.com/zinc/nmr.html`

[15] *WebElements Periodic Table of the Elements — Phosphorus — NMR data*, June 2010
URL: `http://www.webelements.com/phosphorus/nmr.html`

[16] *WebElements Periodic Table of the Elements — Indium — NMR data*, June 2010
URL: `http://www.webelements.com/indium/nmr.html`

[17] V. J. McBrierty and K. J. Packer. *Nuclear magnetic resonance in solid polymers.* Cambridge University Press, 1993

[18] T. T. P. Cheung and B. C. Gerstein. *1H nuclear magnetic resonance studies of domain structures in polymers.* **J. Appl. Phys.**, 52(9):5517–5528, September 1981

[19] V.D. Fedotov and H. Schneider. *Structure and Dynamics of Bulk Polymers by NMR-Methods.* Spinger-Verlag Berlin, 1989

[20] Nigel J. Clayden. *Lammelar Thickness of Crystallizable Ethene Runs in Ethene-propene Copolymers by Solid State NMR.* **Journal of Polymer Science: Part B: Polymer Physics**, 32:2321–2327, 1994. ISSN 0887-6266

[21] Nigel J. Clayden and Gerard Smyth. *Investigation of Structural Inhomogeneities in Soft-Block Modified Methacrylate Resins.* **Magnetic Resonance in Chemistry**, 33:710–716, 1995. ISSN 0749-1581

[22] Q Chen and K Schmidt-Rohr. *Measurement of the local H-1 spin-diffusion coefficient in polymers.* **Solid State Nuclear Magnetic Resonance**, 29(1-3):142–152, February 2006. ISSN 0926-2040

[23] B. Meurer and G. Weill. *Measurement of spin diffusion coefficients in glassy polymers: Failure of a simple scaling law.* **Macromolecular Chemistry and Physics**, 209(2):212–219, 2008

[24] *Non-linear least squares - Wikipedia, the free encyclopedia*, June 2010
URL: `http://en.wikipedia.org/wiki/Non-linear_least_squares`

[25] *Gaussian Elimination - Wikipedia, the free encyclopedia*, June 2010
URL: `http://en.wikipedia.org/wiki/Gaussian_elimination`

[26] *Simplex Algorithm - Wikipedia, the free encyclopedia*, June 2010
URL: `http://en.wikipedia.org/wiki/Simplex_Algorithm`

[27] *Simplex - Wikipedia, the free encyclopedia*, June 2010
URL: `http://en.wikipedia.org/wiki/Simplex`

[28] *Regression Analysis - Wikipedia, the free encyclopedia*, June 2010
URL: `http://en.wikipedia.org/wiki/Regression_Analysis`

[29] *Arbitrary-precision arithmetic - Wikipedia, the free encyclopedia*, June 2010
URL: `http://en.wikipedia.org/wiki/Arbitrary_precision`

Appendix B

Data Fitting

B.1 Least Squares Method

B.1.1 Least Squares

Since the systems measured often consist of multiple phases, one has multiple T_1 and $T_{1\rho}$ components, which all add up in the measured data.

Since the parameters are not all linear, the parameters cannot be estimated using linear regression. Instead an iterative, non-linear approach has to be used.

A non-linear regression [24] model with n parameters and m squared residuals can be written as

$$y_i = f(x_i, \beta) + Z_i \tag{B.1}$$

where β is a vector consisting of the parameters, x is the independent variable and Z_i represents the statistical error.

To measure how well a set of parameters fits the data, the least-squares method is used. Here the squared sum of the residuals is reduced to improve the fit of the parameters.

$$S = \sum_{i=1}^{i=m} r_i^2 \tag{B.2}$$

$$r_i = (y_i - f(x_i, \beta)) \tag{B.3}$$

When the gradient of S is zero, a local minimum of S has been found. Since the model contains j parameters, there are j different gradients for S represented by the following set of equations:

$$\frac{\partial S}{\partial \beta_j} = 2 \sum_i r_i \frac{\partial r_i}{\partial \beta_j} \tag{B.4}$$

Since $\frac{\partial r_i}{\partial \beta_j}$ depends on both the independent variable and the parameters, there is no closed solution for this. Instead a set of initial values has to be chosen for the parameters and the parameters are then refined iteratively.

$$\beta^{\nu+1} = \beta^{\nu} + \Delta\beta \tag{B.5}$$

At each step of the iteration the model is linearized by approximation to a first-order Taylor series expansion about β^{ν}.

$$f(x_i,\beta) \approx f(x_i,\beta^{\nu-1}) + \sum_j \frac{\partial f(x_i,\beta^{\nu-1})}{\partial \beta_j}(\beta_j^{\nu-1} - \beta_j) = f(x_i,\beta^{\nu-1}) + \sum_j \mathbf{J}_{ij}\Delta\beta_j \tag{B.6}$$

Since the Jacobian J contains both the independent variable and the parameters, it changes with each iteration. In terms of the linearized model, the Jacobian can be written as

$$\frac{\partial r_i}{\partial \beta_j} = -\mathbf{J} \tag{B.7}$$

the residuals can thus be written as

$$r_i = \Delta y_i - \sum_{j=1}^{j=n} \mathbf{J}_{ij}\Delta\beta_j \tag{B.8}$$

$$\Delta y_i = y_i - f(x_i,\beta^{\nu}). \tag{B.9}$$

Substituting into the gradient equations, one gets the following set of equations

$$-2\sum_{i=1}^{i=m} \mathbf{J}_{ij}\left(\Delta y_i - \sum_j \mathbf{J}_{ij}\Delta\beta_j\right) = 0 \tag{B.10}$$

on rearrangement one gets a set of n linear equations, the *normal equations*

$$\sum_{i=1}^{i=m}\sum_{k=1}^{k=n} \mathbf{J}_{ij}\mathbf{J}_{ik}\Delta\beta_k = \sum_i \mathbf{J}_{ij}\Delta y_i \quad \text{where} \quad j = \langle 1, n\rangle \tag{B.11}$$

for j between 1 and n.

In matrix notation this can be written as

$$(\mathbf{J^T J})\Delta\beta = \mathbf{J^T}\Delta y \tag{B.12}$$

B.1.2 Gaussian Elimination

The linearized equation system described in equation B.12 can be solved to get a shift vector β which points in the direction of the local minimum.

One possible method of solving this equation system is *Gaussian elimination* [25].

This method is used to solve equation systems in the form

$$\mathbf{A}x = b. \tag{B.13}$$

When comparing this to our linearized equation system, we find that

$$\mathbf{A} = \mathbf{J^T J} \tag{B.14}$$

$$x = \Delta\beta \tag{B.15}$$

$$b = \mathbf{J^T}\Delta y. \tag{B.16}$$

For solving in a computer one usually deals with the *augmented matrix*:

$$[\mathbf{A} \mid b] \tag{B.17}$$

The Gaussian elimination is now performed in two steps, first the Matrix is put into row echelon form. The following Octave function can be used for that:

```
function A = echelon(A)
  i = 1;
  j = 1;
  [m,n] = size(A);
  while(i<=m \&\& j<=n)
    maxi = i;
    k = i + 1;
    while(k<=m)
      if(abs(A(k,j)) > abs(A(maxi,j))
        maxi = k;
      endif
      k = k+1;
    endwhile
    if(A(maxi,j) != 0)
      # Swap rows i and maxi
      a = A(maxi,:);
      A(maxi,:) = A(i,:);
      A(i,:) = a;
      A(i,:) = A(i,:)/A(i,j);
      u = i + 1;
      while(u<=m)
        A(u,:) = A(u,:) - A(i,:)*A(u,j);
      endwhile
      i=i+1;
    endif
    j=j+1;
  endwhile
endfunction

```

The resulting matrix may now be solved using back-substitution.

```
function A = backSubstitue(A)
  [m,n] = size(A);
  i = n-1;
  while(i>1)
    A(i,:) = A(i,:) * (1/A(i,i));
    j = i - 1;
    while(j>=1)
      if(A(j,i)!=0)
```

```
9           A(j,:) = A(j,:) - A(i,:) * (1/A(j,i));
10        endif
11        j = j - 1;
12      endwhile
13      i = i - 1;
14    endwhile
15    A(1,:) = A(1,:) * (1/A(1,1));
16  endfunction
```

After these two steps, $\mathbf{A} = \mathbf{I}$ and therefore $b = x$ so the last column of our augmented matrix (or b) is the solution to the linear equation system, and therefore our new shift vector.

B.1.3 Shift-cutting

Since the linearized equation system (equation B.12) is only a local approximation of the real system, it is possible that when divergence occurs, the fit of $f(x, \beta^\nu + \Delta\beta)$ is actually worse than the fit of $f(x, \beta^\nu)$. To overcome this problem, the magnitude of the step can be reduced by introducing a cutting parameter f:

$$\beta^{\nu+1} = \beta^\nu + f\Delta\beta \tag{B.18}$$

This parameter is usually set to 1, and halved until the fit of $f(x, \beta^\nu + f\Delta\beta)$ is better than the fit of $f(x, \beta^\nu)$.

B.1.4 Marquardt parameter

When the shift-vector is far from the "ideal" direction, shift-cutting becomes quite ineffective, since the fraction f is then required to be very small to avoid divergence. The Marquardt parameter is introduced to allow the shift vector to be rotated towards the steepest descent. To achieve this, the normal equation is modified to give

$$(\mathbf{J}^T\mathbf{J} + \lambda\mathbf{I})\Delta\beta = \mathbf{J}^T\Delta y. \tag{B.19}$$

Here λ is the Marquardt parameter. When $\lambda = 0$, the new normal equation is equivalent to the original normal equation (Equation B.12). When λ is increased though, the direction of the shift-vector is changed towards the steepest descent, while the length of the vector is reduced, due to the $1/\lambda$ factor in

$$\lim_{\lambda\to\infty} \Delta\beta = \frac{1}{\lambda}\mathbf{J}^T\Delta y. \tag{B.20}$$

If the new iteration is not an improvement over the last one, the value of λ needs to be increased. It can also be reduced, if possible. When reducing the value, it is save to set it to zero, once

$$\lambda < \frac{1}{\text{trace}(\mathbf{J}^T\mathbf{J})^{-1}}. \tag{B.21}$$

B.2 Simplex Algorithm

Another algorithm that can be used to move towards a solution of the normal equations (Equation B.12), is the simplex algorithm [26].

Here $n+1$ sets of least squares with different Δb are calculated in such a way, that the angles between the vertices of the current point, and the current point in parameter space are the same. For a two dimensional parameter space, one would get an equilateral triangle when connecting all the points.

The point with the best least squares fit is carried on to the next iteration, and using it as the new origin, another set of points is calculated.

When the origin is the best fit, the magnitudes of the shift vectors are reduced, until one point, which is not the origin, is a better fit.

B.2.1 The Simplex

The simplex is a set of geometric structures, one for each dimensionality, which have the property that a minimum amount of vertices are needed to define a volume (for 3 or more dimensions) or an area (for 2 dimensions). The simplex [27] for a 2-dimensional environment is the equilateral triangle, and for a 3-dimensional environment it is the tetrahedron. Thus a simplex always has $n+1$ vertices, where n is the dimensionality of the simplex. A single point can be considered the 0-dimensional simplex, while the 1-dimensional simplex is a simple line.

B.2.2 How to construct the simplex for the Simplex Algorithm

The simplex used in a simplex algorithm needs to have the current point at its exact center. This is needed so that the whole parameter space is covered evenly by the simplex.

We need to construct $i = D + 1$ amount of vertices, each being a $j = D$ component vector. The coordinates of such a simplex can be represented as a i-by-j matrix. When constructing the $n+1$ coordinates of a n-dimensional simplex a few basic rules must be observed:

1. All vectors must must be unit-vectors i.e. have a magnitude of 1
2. The dot product between any pair of vectors must be $-1/n$
3. For the n^{th} vector all but the first n numbers are 0

As an example we will now construct a 5-dimensional simplex. All vectors will be represented as row-vectors in a 6x5 matrix. The first vector is easy to choose, when making all but the first number 0, we get a 1 as the fist number and observing rule 3 we get:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ ? & ? & 0 & 0 & 0 \\ ? & ? & ? & 0 & 0 \\ ? & ? & ? & ? & 0 \\ ? & ? & ? & ? & ? \\ ? & ? & ? & ? & ? \end{bmatrix} \tag{B.22}$$

We then set the first coordinate of each vector to $-1/n$. This is done, since we know the second coordinate of the second vector is irrelevant, since it is multiplied by 0 when taking the dot-product with the first vector. Thus to give $-1/n$ for a dot product with the first vector, each first coordinate must be set to $-1/n$ like thus:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ -\frac{1}{5} & ? & 0 & 0 & 0 \\ -\frac{1}{5} & ? & ? & 0 & 0 \\ -\frac{1}{5} & ? & ? & ? & 0 \\ -\frac{1}{5} & ? & ? & ? & ? \\ -\frac{1}{5} & ? & ? & ? & ? \end{bmatrix} \tag{B.23}$$

When using rule 1 we can now calculate the second coordinate for the second vector using Pythagoras (any of the two solutions is fine) and using rule 2 we calculate the second coordinate of the other four vectors:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ -\frac{1}{5} & \frac{2}{5\sqrt{6}} & 0 & 0 & 0 \\ -\frac{1}{5} & -\frac{1}{10\sqrt{6}} & ? & 0 & 0 \\ -\frac{1}{5} & -\frac{1}{10\sqrt{6}} & ? & ? & 0 \\ -\frac{1}{5} & -\frac{1}{10\sqrt{6}} & ? & ? & ? \\ -\frac{1}{5} & -\frac{1}{10\sqrt{6}} & ? & ? & ? \end{bmatrix} \tag{B.24}$$

Now it is easy to calculate the third coordinate of all vectors:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ -\frac{1}{5} & \frac{2}{5\sqrt{6}} & 0 & 0 & 0 \\ -\frac{1}{5} & -\frac{1}{10\sqrt{6}} & \frac{3}{10\sqrt{10}} & 0 & 0 \\ -\frac{1}{5} & -\frac{1}{10\sqrt{6}} & -\frac{1}{10\sqrt{10}} & ? & 0 \\ -\frac{1}{5} & -\frac{1}{10\sqrt{6}} & -\frac{1}{10\sqrt{10}} & ? & ? \\ -\frac{1}{5} & -\frac{1}{10\sqrt{6}} & -\frac{1}{10\sqrt{10}} & ? & ? \end{bmatrix} \tag{B.25}$$

And the fourth coordinate:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ -\frac{1}{5} & \frac{2}{5\sqrt{6}} & 0 & 0 & 0 \\ -\frac{1}{5} & -\frac{1}{10\sqrt{6}} & \frac{3}{10\sqrt{10}} & 0 & 0 \\ -\frac{1}{5} & -\frac{1}{10\sqrt{6}} & -\frac{1}{10\sqrt{10}} & \frac{2}{5\sqrt{5}} & 0 \\ -\frac{1}{5} & -\frac{1}{10\sqrt{6}} & -\frac{1}{10\sqrt{10}} & -\frac{1}{5\sqrt{5}} & ? \\ -\frac{1}{5} & -\frac{1}{10\sqrt{6}} & -\frac{1}{10\sqrt{10}} & -\frac{1}{5\sqrt{5}} & ? \end{bmatrix} \tag{B.26}$$

Now the last two numbers are easily solvable, the two possible solutions make the two last numbers:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ -\frac{1}{5} & \frac{2}{5\sqrt{6}} & 0 & 0 & 0 \\ -\frac{1}{5} & -\frac{1}{10\sqrt{6}} & \frac{3}{10\sqrt{10}} & 0 & 0 \\ -\frac{1}{5} & -\frac{1}{10\sqrt{6}} & -\frac{1}{10\sqrt{10}} & \frac{2}{5\sqrt{5}} & 0 \\ -\frac{1}{5} & -\frac{1}{10\sqrt{6}} & -\frac{1}{10\sqrt{10}} & -\frac{1}{5\sqrt{5}} & \frac{1}{5\sqrt{3}\sqrt{5}} \\ -\frac{1}{5} & -\frac{1}{10\sqrt{6}} & -\frac{1}{10\sqrt{10}} & -\frac{1}{5\sqrt{5}} & -\frac{1}{5\sqrt{3}\sqrt{5}} \end{bmatrix} \tag{B.27}$$

This method works to construct the cartesian coordinates of any n-dimensional simplex with its center of gravity at the origin.

B.3 Regression Analysis

When one has several possible models, with a different amount of parameters for each, one needs a way to determine which model is best used to approximate the data. The mathematical method of determining how well a model describes the data is called regression analysis [28].

Considering two models, a simple one, with only a few degrees of freedom, and a complicated one, with many degrees of freedom, which fits the data only slightly better, then the simple model is to be preferred.

The measure used to evaluate the complexity of a model, is the number of dimensions (or *degrees of freedom*) of the model, and how this relates to the degrees of freedom of the data. The total degrees of freedom of the dataset is equal to $m-1$, while the degrees of freedom of the model is equal to the amount of parameters n.

To evaluate how well a model explains the data, one needs to have a measure for this. Generally a dimensionless model, the average value of the datapoints ($\bar{Y}$), is used as a reference. The total variation from this reference, which is called *total sum of squares* (SSY), is defined as

$$SSY = \sum_{i=1}^{n} (Y_i - \bar{Y})^2. \tag{B.28}$$

The *residual sum of squares* (SSE), which is a measure of how much the data varies from the model ($\hat{Y}$) is defined as

$$SSE = \sum_{i=1}^{n} (Y_i - \hat{Y})^2. \tag{B.29}$$

The amount of variation which is explained by the model is now apparent as

$$\sum_{i=1}^{n} (\bar{Y} - \hat{Y})^2 = SSY - SSE. \tag{B.30}$$

These values by themselves are useful to evaluate a single dataset, but they are not very useful to compare different datasets to another. To do this, one needs values which are normalized. One thing one can do, is to calculate the fraction of variation explained by the model:

$$r^2 = \frac{\sum_{n=1}^{n} (\bar{Y} - \hat{Y})^2}{\sum_{n=1}^{n} (Y_i - \bar{Y})^2} = \frac{SSY - SSE}{SSY} \tag{B.31}$$

Another value, which is useful to compare two different datasets, is the *F-value*. This value compares the amount of variance explained by the model to the residual variance:

$$F = \frac{\sum_{n=1}^{n} (\bar{Y} - \hat{Y})^2}{\sum_{n=1}^{n} (Y_i - \hat{Y})^2} \tag{B.32}$$

Appendix C

Software

C.1 Software capabilities and limitations

C.1.1 Choice of programming languages

The computer program used to simulate and the one to do the curve fitting for the spin diffusion systems, were written in the programming language *C*. Although a first version of the computer program to do the curve fitting was written in *octave*, a programming language very similar and compatible to *Mathematica* but open source, the final program was implemented in *C* because the program would run very slowly in *octave* and use up considerable amounts of memory.

The choice fell on *C* as a programming language since it is possible to write very fast and optimized software in it. The result was much faster software (approximately six times faster).

C.1.2 Curve Fitting

For curve fitting, a function implementing the simplex algorithm was implemented in *octave*. This function could then be used in octave like any other function. One would manually read in the input data into a matrix using internal functions of *octave* and then start the Simplex function on that data, feeding it a model function and some initial parameters.

C.1.3 TLM Simulator

The TLM Simulator is capable of simulating diffusion systems with a dimensionality of 1 to 3. It is possible to set the R and Z values as well as the starting values of every node. The amount of nodes simulated that can be simulated and the amount of steps are only limited by the memory of the machine used.

Hardware and Software requirements

The TLM Simulator, as listed in Appendix G.2 on Page 116, is written for a UNIX system with a 64bit processor. The memory requirements for the data

Dimension	Size	Memory requirements (kB)
1	10	157.8
1	100	162.7
1	1000	211.9
1	10000	704.1
2	10	165.8
2	100	1'016.6
2	1000	86'094.7
2	3500	1'052'891.6
2	10000	8'593'907.2
3	10	274.4
3	100	117'344.7
3	200	937'657.2
3	208	1'054'717.2
3	250	1'831'211.9
3	1000	117'187'657.3

Table C.1: Memory usage for different sizes and dimensionalities for 10'000 steps

depends on the dimensionality of the system simulated and the size of the edges of the simulated system.

$$\text{Mem} = x^d(32d + 24) + 16s + 1030 \tag{C.1}$$

Where x represents the size of an edge, d represents the dimensionality and s represents the amount of steps simulated. Thus the memory requirement (for $10'000$ steps) is shown for different dimensions and simulation sizes in table C.1.

It is easily seen that while the amount of memory needed for 1D systems is quite small higher dimensional systems get big very fast. When simulating 2D systems or even 3D systems, the amount of memory needed scales unfavorably for large systems. Where with a 2D model a size of 3500 is needed to reach a memory usage of over 1 GiB, with a 3D model a size of only 208 is needed.

C.1.4 TLM helper tools

For the TLM helper tools a fairly recent version of *Perl* is needed (Perl v5.8.8 was used), as well as the the *Getopt::Long* and *Parallel::ForkManager* Perl modules.

graphGen.pl also needs a version of *gnuplot* capable of producing scalable vector graphics and texdraw output.

C.2 Software usage

C.2.1 TLM Simulator

confGen.pl

The *TLM-Sim* program itself is quite specific in how it is supposed to be used. It needs a special configuration file, which can be constructed by the helper tool *confGen.pl*. *confGen.pl* works like a regular Unix command line program and generates the text of the configuration file on *STDOUT*, it is therefore recommendable to redirect the output into a file. The options for *confGen.pl* are summarized in Table C.2 on page 90.

A typical call of *confGen.pl* looks something like this:

```
$ ./confGen.pl --size=50:50:50 --steps=10000 --LL \
  --box=0:24:0:24:0:24 --R=100:1000 --Z=100:10 --initVal=100:0 \
  "--areas=0:24:0:24:0:24" --T1=250 > tlm.conf
```

and creates a file *tlm.conf* defining the configuration for a $50 \times 50 \times 50$ simulation, running for 10000 steps, using the Link-Line node model, including a $25 \times 25 \times 25$ box in one corner in which the reflection coefficient $\rho = 1000$ and the transmission coefficient $\tau = 10$. The bulk material has the coefficients $\rho = 100$ and $\tau = 100$ and initial value of 100. The values of the nodes will decay with a half-life of 250 steps.

table2conf.pl

The TLM helper tool *table2conf.pl* was written to allow a large number of simulations to be started in bulk using only one command. The tool will create several configuration files in a directory tree contained within a previously set directory.

Then it will start *TLM-Sim* within for each of these configuration files and put the output into a file called *OUTPUT* in the same directory.

The configuration of *table2conf.pl* is done within the source code of that file by editing some variables and lists, defining parameters similar to the ones in *confGen.pl*. In fact *table2conf.pl* calls *confGen.pl* using every possible combination of the values contained within these lists.

It is also necessary to set the variables *$confGen*, *$TLM_Sim* and *$basePath* to the locations of *confGen.pl*, the compiled *TLM Simulator* and an empty directory, into which the configuration files and the results are written, respectively.

Option	Description
`--size=`*`Xsize[,Ysize[,Zsize]]`*	Specify the size of the simulation, Ysize and Zsize are optional, Xsize is not. The setting of Ysize implies a 2D (or 3D) model and the setting of Zsize implies a 3D model.
`--steps=`*`Number`*	The number of steps the simulation is supposed to run.
`--LL | --LR`	Either `--LL` or `--LR` should be set, but not both! `--LL` sets the simulation to Link-Line mode and `--LR` sets it to Link-Resistor mode.
`--box=`X_1`:`X_2`[:`Y_1`:`Y_2`[:`Z_1`:`Z_2`]]`	Define the size of the box which is observed. Changes in this variable will only change what is observed as *Box* or *Bulk*, not define the different environment within that box.
`--R=`R_{bulk}`:`R_1`[:`R_2`[,...]]`	Define the reflection coefficient for the bulk material as well as the separate areas defined with `--areas`.
`--Z=`R_{bulk}`:`R_1`[:`R_2`[,...]]`	Define the transmission coefficient for the bulk material as well as the separate areas defined with `--areas`.
`--initVal=`V_{bulk}`:`V_1`[:`V_2`[,...]]`	Defines the initial values of the nodes in the bulk material as well as the different areas defined with `--areas`.
`--areas=`X_1`:`X_2`[:`Y_1`:`Y_2`[:`Z_1`:`Z_2`]][,...]`	Define the different areas, for each area there has to be an entry in `--R`, `--Z` and `--initVal`. The rest of the simulation will be defined as bulk.
`--T1=[Half-Life]`	Switch on T_1 simulation with given half life.
`-v | --verbose`	Switch on output of nodal values for each step.

Table C.2: Options for the TLM helper tool *genConf.pl*

Appendix D

Floating Point Accuracy Problems

D.1 Introduction

In many parts of this work, computer software was used to do curve fitting to a set of modeled data or simulation of spin diffusion data.

For the simulation this is not a significant issue, because for values that were used in the calculations a tiny percentage change in a number did not have a significant impact. For some of the fitting tasks, especially when trying to fit the complex FID from the spin diffusion measurements (5.4.2, which also contained a wave component, this can be a significant issue.

Especially when calculating the square-sum of the fitted data, to determine how well it fits with the data, there can be a huge difference between the values to be added. Since even a tiny improvement in fitting can indicate a path to a minimum and a large number of datapoints was used, the sum of all those errors can indeed become significant.

While running these fits, debugging statements were added to the software to extract the numbers the software was actually working on. In some cases, a significant proportion of the numbers were close or below ϵ. ϵ is a value denoting the smallest number that can be added to 1 for the machine to actually calculate a sum > 1.

$$\begin{aligned} x &= 1 + y \\ x &= \begin{cases} > 1 & : \quad y \geq \epsilon \\ 1 & : \quad y < \epsilon \end{cases} \end{aligned} \tag{D.1}$$

Thus, if this happens often enough, and it easily can when using many datapoints, there can be a significant amount of datapoints not considered within the square-sum. Another point where this can be of concern is, when the amount the parameters are varied falls close or below *epsilon* with significant differences in the square sum. This would mean that even though a variation of this param-

eter would result in an improvement, the machine can not actually change the parameter anymore.

This could be circumvented by carefully adjusting the formula and introducing a new base term and only calculating the offset to this base term. But this would increase the number calculations needed drastically since the changes needed to make this possible without introducing new points where rounding errors could ruin the result would bloat the formula significantly. It might not always be possible to do this for every parameter.

D.2 Floating Point and Precision

When representing numbers in a computer, one has the choice between two different types of arithmetic (*floating-* and *fixed-point*), with advantages and disadvantages each.

Fixed point numbers are essentially integers with a fixed scaling factor, which is not stored as part of the number. Thus we can represent the number 1.23 as $\frac{123}{100}$. With floating point numbers the scaling factor is part of the number itself.

The scaling factor is usually binary or decimal. A binary scaling factor is a power of two, while a decimal scaling factor is a power of ten. Most commonly a binary scaling factor is used, because rescaling can be implemented using fast bit shifts. Binary fixed-point can represent fractional powers of two exactly, while decimal fixed-point can only represent fractional powers of ten exactly.

In *IEEE-754*[1] floating point numbers a binary scaling factor is usually used.[2] There are three different floating point formats defined:

Name	Precision (bin)	Precision (dec)	Epsilon	Max Exponent (2^x)
binary32[3]	23+1 bits	7.225	1.192093×10^{-7}	127
binary64[4]	52+1 bits	15.95	2.220446×10^{-16}	1023
binary128[5]	112+1 bits	34.02	1.925930×10^{-34}	16383

Since only the fractional part of the significant is stored, and the most significant bit which would be equal to 1 is assumed to be *on*, the binary precision is always one digit bigger than the amount of bits used in storage. The decimal precision is calculated using the following formula:

$$P_{dec} = \log_{10}(2^{P_{bin}}) \tag{D.2}$$

where P_{dec} is the decimal precision and P_{bin} is the binary precision. The value epsilon is defined as the difference between 1 and the next biggest representable number. It can be calculated as follows:

$$\epsilon = 2^{P_{bin}-1} \tag{D.3}$$

Since the *binary128* format is quite new (it was added in the 2008 version of IEEE-754), most programming libraries still only use *binary32* and *binary64*. Therefore one can not easily

[1] The standard most commonly used by current FPUs

[2] The standard does define decimal floating point formats, but the binary floating point formats are more commonly used

[3] Also called single-precision

[4] Also called double-precision

[5] Also called quad-precision

develop software using the *binary128* format, since no or only very few programming libraries exist that support this format.

D.3 Rounding and Accuracy Problems

Since the precision of the number formats is limited, rounding must occur for every number which can not be exactly represented. Such as π, but also 0.1 and 0.01. This rounding introduces a small error, which can grow to be quite significant as more and more mathematical operations are carried out with the results of each previous equation.

For example, when calculating something simple like 0.1^2 using *binary32* floating point numbers:

Since 0.1 can not be represented directly it is rounded to the nearest number:

0.100000001490116119384765625 exactly.

Squaring this number gives

0.010000000298023226097399174250313080847263336181640625 exactly.

Squaring it using a single precision FPU gives (after rounding):

0.010000000707805156707763671875 exactly.

But the number closest to the actual result of 0.1^2 is

0.009999999776482582092285156250 exactly.

This shifting rounds the number and thus reduces the accuracy.

```
  e=5;  s=1.234567    (123456.7)
+ e=3;  s=9.481957    (9481.957)

  e=5;  s=1.234567
 [e=5;  s=0.09481957]
+ e=5;  s=0.094820    (after shifting)
--------------------
= e=5;  s=1.329387
```

If the difference of the two numbers is greater than the significance of the number format used, the number with smaller magnitude is effectively dropped.

```
  e=5;  s=1.234567    (123456.7)
+ e=-3; s=9.481957    (0.009481957)

  e=5;  s=1.234567
 [e=5;  s=0.00000009481957]
+ e=5;  s=0.000000
--------------------
= e=5;  s=1.234567
```

When calculating the squared sum of the residuals, this can be quite significant. When the differential is large, the square is even larger, but when the differential is small the square will become even smaller. This means that in some conditions a significant amount of datapoints is not taken into consideration anymore, since each one is too small after shifting to change the sum.

A loss of significance occurs when two numbers, which are close to one another, are subtracted. The closer two numbers are, the less accurate the calculated difference between them is.

```
  e=5;  s=1.234571
- e=5;  s=1.234567
--------------------
= e=5;  s=0.000004
  e=-1; s=4.000000   (after rounding and normalisation)
```

A nice example is the calculation of π using Archimedes approximation by calculating the perimeter of polygons inscribing and circumscribing a circle. The following iterative model starts with hexagons and successively doubles the number of sides:

$$t_0 = \frac{1}{\sqrt{3}} \tag{D.4}$$

$$t_{i+1} = \frac{\sqrt{t_i^2+1}-1}{t_i} \quad \text{Original iterative step} \tag{D.5}$$

$$t_{i+1} = \frac{t_i}{\sqrt{t_i^2+1}+1} \quad \text{Alternate iterative step} \tag{D.6}$$

$$\pi \approx 6 \times 2^i \times t_i \tag{D.7}$$

Both the original and alternative iterative steps are mathematically equivalent, but when used for computing the result they are obviously very different. In the original iterative step, 1 is subtracted from a number extremely close to 1, which leads to a very significant cancellation error

The following table shows the calculating with the original and the alternative iterative step using IEEE *double precision* arithmetic:

i	$6 \times 2^i \times t_i$ original	$6 \times 2^i \times t_i$ alternative
0	3.4641016151377543863	3.4641016151377543863
1	3.2153903091734710173	3.2153903091734723496
2	3.1596599420974940120	3.1506599420975006733
3	3.1460862151314012979	3.1460862151314352708
4	3.1427145996453136334	3.1427145996453689225
5	3.1418730499801259536	3.1418730499798241950
6	3.1416627470548084133	3.1416627470568494473
7	3.1416101765997805905	3.1416101766046906629
8	3.1415970343230776862	3.1415970343215275928
9	3.1415937488171150615	3.1415937487713536668
10	3.1415929278733740748	3.1415929273850979885
11	3.1415927256228504127	3.1415927220386148377
12	3.1415926717412858693	3.1415926707019992125
13	3.1415926189011456060	3.1415926578678454728
14	3.1415926717412858693	3.1415926546593073709
15	3.1415919358822321783	3.1415926538571730119
16	3.1415926717412858693	3.1415926536566394222
17	3.1415810075796233302	3.1415926536065061913
18	3.1415926717412858693	3.1415926535939728836
19	3.1414061547378810956	3.1415926535908393901
20	3.1405434924008406305	3.1415926535900560168
21	3.1400068646912273617	3.1415926535898608396
22	3.1349453756585929919	3.1415926535898122118
23	3.1400068646912273617	3.1415926535897995552
24	3.2245152435345525443	3.1415926535897968907
25		3.1415926535897962246
26		3.1415926535897962246
27		3.1415926535897962246
28		3.1415926535897962246
	$\pi =$	3.1415926535897932384626433832...

D.4 Arbitrary-Precision Arithmetic

One solution to overcome this kind of problem is arbitrary-precision arithmetic [29]. Arbitrary-precision arithmetic is a method to do calculations with any selectable precision. The exact precision is only limited by the amount of memory available. However there are several downsides. One is that special programming-libraries or programming languages are needed to implement arbitrary-precision arithmetic. The other is that these calculations are generally considerably slower than normal calculations using the floating point unit. With increasing precision not only the memory requirement, but also the time needed for each calculation step increases.

Since normal calculations on a computer are implemented in hardware, they are quite fast. Most computers (except maybe a few specially designed for this task) however do not have any integrated hardware solution to arbitrary-precision arithmetic, so this has to be implemented

entirely in software.

This implementation in software brings with it a high flexibility though. Numbers can be stored in fixed point or floating point format with any preselected precision. However when introducing division, a simple fraction can make perfect precision using fixed- or floating point numbers impossible.

An example could be a simple fraction such as $\frac{4}{7}$. Since it has an infinitely repeating sequence of digits, it has to be truncated at some point. Usually, with arbitrary-precision systems, the programmer has to set a variable defining the maximum precision of the calculations. This is first to make sure that the computer does not spend hours calculating a simple fraction to the millionth digit, and second to to limit the amount of memory large collections of such numbers would take up.

Some pieces of software take the mathematical approach and represent these rational numbers as the fractions themselves instead of fixed- or floating-point numbers. Unfortunately mathematics with rational numbers can get quite unwieldy, as shown in this example:

$$\frac{1}{99} - \frac{1}{100} = \quad \frac{1}{9900} \tag{D.8}$$

$$\frac{1}{9900} + \frac{1}{101} = \quad \frac{10001}{999900} \tag{D.9}$$

The library implementing this kind of precision would have to be aware of a whole multitude of mathematical rules. It would have to be able to simplify any mathematical representation of a number to a shortest possible formula. Usually this kind of complexity is only available in computer algebra software.

While software like this exists, it is quite unsuited to calculate vast amounts of data, since the drawback of such precision is an even higher requirement for CPU cycles and memory. Usually, owing the complexity of programming a piece of software able to solve mathematical formulae, these kind of programs come with a high price tag.

Since neither the funds to purchase the kind of software, that would be needed to solve the kind of highly complicated and data intensive problems involved in this work, nor the processing power which would be needed to compute these kinds of problems to a satisfactory precision, were available, this approach was unfeasible. At the time, at which it became apparent, that such measures would be necessary to solve some of the problems discussed in this work, it was no longer possible to reprogram all the software involved implementing arbitrary-precision arithmetic.

D.5 Relevance to this work

These problems are relevant to this work in two different cases. Firstly when using nonlinear regression, the square sum can be scewed due to rounding errors if there are a lot of points.

Most importantly however is the TLM Simulator. Since it is an iterative software, the results from the previous step are used to calculate the next. Even though there might not be a specific point that is more vulnerable to rounding errors, the simple fact that a minute rounding error near the beginning of the simulation will propagate through all subsequent steps can make this a significant factor, especially if a high number of steps are simulated.

Appendix E

Simulation Parameters

Figure 5.9 (p57)

This 2D and 3D curves have been scaled to make them better comparable. The difference in system size is so that for each so that for each of them about 50% of the simulated nodes were used for area **A** and the other 50% were used for area **B**.

Dimensionality	**1D**	**2D**	**3D**
Model	LL	LL	LL
Size	20	90x90	24x24x24
Steps	1500	25000	15000
Initial Val (Area A)	0	0	0
Initial Val (Area B)	100	100	100
Z (Area A & B)	100	100	100
R (Area A)	100	100	100
R (Area B)	100	100	100
Area A Coordinate	9	69,69	18,18,18
Factor x-axis	1	25	3
Factor y-axis	0	+1.05‰	+7.73‰

Table E.1: Parameters for Figure 5.9

Figure 5.11 (p59)

Except for the area **A** coordinate these graphs share the parameters listed in the table:

Dimensionality	1D
Model	LL
Size	50
Steps	10000
Initial Val (Area A)	0
Initial Val (Area B)	100
Z (Area A & B)	100
R (Area A)	100
R (Area B)	1000

Table E.2: Parameters for Figure 5.11

Figure 5.12 (p60)

Except for the area **A** coordinate these graphs share the parameters listed in the table:

Dimensionality	2D
Model	LL
Size	50
Steps	10000
Initial Val (Area A)	0
Initial Val (Area B)	100
Z (Area A & B)	100
R (Area A)	100
R (Area B)	1000

Table E.3: Parameters for Figure 5.12

Figure 5.13 (p61)

Except for the area **A** coordinate these graphs share the parameters listed in the table:

Dimensionality	3D
Model	LL
Size	50
Steps	10000
Initial Val (Area A)	0
Initial Val (Area B)	100
Z (Area A & B)	100
R (Area A)	100
R (Area B)	1000

Table E.4: Parameters for Figure 5.13

Figure 5.14 (p62)

The radius was chosen so that the area of area **A** of both simulations is exactly the same.

Shape	**Circle**	**Square**
Dimensionality	2D	2D
Model	LL	LL
Size	200	200
Steps	50000	50000
Initial Val (Area A)	0	0
Initial Val (Area B)	100	100
Z (Area A & B)	100	100
R (Area A)	100	100
R (Area B)	10	10
Area A Coordinate	—	141
Radius	159.101	—

Table E.5: Parameters for Figure 5.14

Figure 5.15 (p63)

The radius was chosen so that the volume of area **A** of both simulations is exactly the same.

Shape	**Sphere**	**Cube**
Dimensionality	3D	3D
Model	LL	LL
Size	100	100
Steps	25000	25000
Initial Val (Area A)	0	0
Initial Val (Area B)	100	100
Z (Area A & B)	100	100
R (Area A)	100	100
R (Area B)	10	10
Area A Coordinate	—	79
Radius	98.015	—

Table E.6: Parameters for Figure 5.15

Figure 5.18 (p68)

Dimensionality	1D
Model	LL
Size	20
Steps	1000
Initial Val (Area A)	0
Initial Val (Area B)	100
Z (Area A & B)	100
R (Area A)	100
R (Area B)	100
Area A Coordinate	9

Table E.7: Simulation parameters for Figure 5.18

Dimensionality	1D
M_0	100
d_A	10 nm
d_B	10 nm
D	$1.25\,\mu\mathrm{m}^{-2}$
Δt	20 ns

Table E.8: Parameters for analytical solution of Figure 5.18

Figure 5.19 (p69)

Dimensionality	2D
Model	LL
Size	20
Steps	2500
Initial Val (Area A)	0
Initial Val (Area B)	100
Z (Area A & B)	100
R (Area A)	100
R (Area B)	100
Area A Coordinate	9

Table E.9: Simulation parameters for Figure 5.19

Dimensionality	2D
M_0	100
d_A	6.4 nm
d_B	6.4 nm
D	$0.273\,\mu\text{m}^{-2}$
Δt	15 ns

Table E.10: Parameters for analytical solution of Figure 5.19

Figure 5.20 (p70)

Dimensionality	3D
Model	LL
Size	20
Steps	10000
Initial Val (Area A)	0
Initial Val (Area B)	100
Z (Area A & B)	100
R (Area A)	100
R (Area B)	100
Area A Coordinate	9

Table E.11: Simulation parameters for Figure 5.20

Dimensionality	3D
M_0	100
d_A	6.4 nm
d_B	6.4 nm
D	0.683 μm^{-2}
Δt	15 ns

Table E.12: Parameters for analytical solution of Figure 5.20

Figure 5.24 (p73)

Dimensionality	2D
Model	LL
Size	90x90
Steps	25000
Initial Val (Area A)	0
Initial Val (Area B)	100
Z (Area A & B)	100
R (Area A)	100
R (Area B)	69,69
Area A Coordinate	9
Δt	17.5 μs
y-scale	0.161

Table E.13: Simulation parameters for Figure 5.24

Appendix F

Acquisition Data

Figure 3.1 (p27)

Nucleus	^{31}P
Frequency	121.474851 MHz
CP-Nucleus	^{1}H
CP-Frequency	300.133400 MHz
Pulse Program	cp
TD	3618
NS	2048
SW	250 kHz
RG	64
90° Pulse	4.1 μs
Conctact time	1911.24 μs
Relaxation time	1 s

Table F.1: Acquisition parameters for Figure 3.1

Figure 3.2 (p28)

Nucleus	^{31}P
Frequency	121.474851 MHz
CP-Nucleus	^{1}H
CP-Frequency	300.133400 MHz
Pulse Program	cp
TD	3618
NS	2048
SW	250 kHz
RG	64
90° Pulse	4.1 μs
Conctact time	2000 μs
Relaxation time	1 s

Table F.2: Acquisition parameters for Figure 3.2

Figure 3.3 (p28)

Nucleus	^{31}P
Frequency	121.474851 MHz
Pulse Program	hpdec
TD	3618
NS	1024
SW	250 kHz
RG	64
90° Pulse	3.0 μs
Relaxation time	60 s

Table F.3: Acquisition parameters for Figure 3.3

Figure 3.4 (p29)

Nucleus	^{31}P
Frequency	121.474851 MHz
Pulse Program	hpdec
TD	3618
NS	128
SW	250 kHz
RG	64
90° Pulse	3.0 μs
Relaxation time	60 s

Table F.4: Acquisition parameters for Figure 3.4

Figure 3.5 (p30)

Nucleus	^{31}P
Frequency	121.474851 MHz
CP-Nucleus	^{1}H
CP-Frequency	300.133400 MHz
Pulse Program	cp
TD	3618
NS	512
SW	250 kHz
RG	64
90° Pulse	4.1 μs
Conctact time	1911.24 μs
Relaxation time	1 s

Table F.5: Acquisition parameters for Figure 3.5

Figure 3.6 (p30)

Nucleus	^{31}P
Frequency	121.474851 MHz
CP-Nucleus	^{1}H
CP-Frequency	300.133400 MHz
Pulse Program	cp
TD	3618
NS	2048
SW	250 kHz
RG	64
90° Pulse	4.1 μs
Conctact time	1911.24 μs
Relaxation time	1 s

Table F.6: Acquisition parameters for Figure 3.6

Figure 3.7 (p31)

Nucleus	^{31}P
Frequency	121.474851 MHz
CP-Nucleus	^{1}H
CP-Frequency	300.133400 MHz
Pulse Program	cp
TD	3618
NS	512
SW	250 kHz
RG	64
90° Pulse	4.1 μs
Conctact time	1911.24 μs
Relaxation time	1 s

Table F.7: Acquisition parameters for Figure 3.7

Figure 3.8 (p31)

Nucleus	^{31}P
Frequency	121.474851 MHz
CP-Nucleus	^{1}H
CP-Frequency	300.133400 MHz
Pulse Program	cp
TD	3618
NS	512
SW	250 kHz
RG	64
90° Pulse	4.1 μs
Conctact time	1911.24 μs
Relaxation time	1 s

Table F.8: Acquisition parameters for Figure 3.8

Figure 3.9 (p32)

Nucleus	^{31}P
Frequency	121.474851 MHz
CP-Nucleus	^{1}H
CP-Frequency	300.133400 MHz
Pulse Program	cp
TD	3618
NS	2048
SW	250 kHz
RG	64
90° Pulse	4.1 μs
Conctact time	1911.24 μs
Relaxation time	1 s

Table F.9: Acquisition parameters for Figure 3.9

Figure 4.3 (p41)

Nucleus	^{13}C
Frequency	75.4752958 MHz
CP-Nucleus	^{1}H
CP-Frequency	300.133400 MHz
Pulse Program	cp
TD	2048
NS	32
SW	22.727273 kHz
RG	912
90° Pulse	3.34 μs
Conctact time	2000 μs
Relaxation time	5 s

Table F.10: Acquisition parameters for Figure 4.3

Figure 4.4 (p41)

Nucleus	^{13}C
Frequency	75.4752958 MHz
CP-Nucleus	^{1}H
CP-Frequency	300.133400 MHz
Pulse Program	sat_cp
TD	2048
NS	64
SW	22.727273 kHz
RG	912
90° Pulse	3.88 μs
Conctact time	2000 μs
Relaxation time	15 s
Presaturation delay	200 ms

Table F.11: Acquisition parameters for Figure 4.4

Figure 4.5 (p41)

Nucleus	^{13}C
Frequency	75.4752958 MHz
CP-Nucleus	^{1}H
CP-Frequency	300.133400 MHz
Pulse Program	cp_selective
TD	2048
NS	64
SW	22.727273 kHz
RG	912
90° Pulse	3.34 μs
Conctact time	2000 μs
Relaxation time	15 s
Spinlock time	100 ms

Table F.12: Acquisition parameters for Figure 4.5

Figure 4.6 (p42)

	Etravirine	Exp0719	Exp9720
Nucleus	^{13}C	^{13}C	^{13}C
Frequency	75.4752958 MHz	75.4752958 MHz	75.4752958 MHz
CP-Nucleus	^{1}H		
CP-Frequency	300.131500 MHz	300.131500 MHz	300.131500 MHz
Pulse Program	cp	cp	cp
TD	1024	2048	2048
NS	256	256	2048
SW	50 kHz	22.727273 kHz	22.727273 kHz
RG	912	912	912
90° Pulse	3.34 μs	3.34 μs	3.34 μs
Conctact time	2000 μs	2000 μs	2000 μs
Relaxation time	15 s	15 s	5 s

Table F.13: Acquisition parameters for Figure 4.6

Figure 4.7 (p44)

Nucleus	^{13}C
Frequency	75.4752958 MHz
CP-Nucleus	^{1}H
CP-Frequency	300.131500 MHz
Pulse Program	cp
TD	2048
NS	384
SW	22.727273 kHz
RG	912
90° Pulse	3.34 μs
Conctact time	2000 μs
Relaxation time	3 s

Table F.14: Acquisition parameters for Figure 4.7

Figure 4.8 (p44)

Nucleus	^{13}C
Frequency	75.4752958 MHz
CP-Nucleus	^{1}H
CP-Frequency	300.131500 MHz
Pulse Program	sat_cp
TD	2048
NS	512
SW	22.727273 kHz
RG	912
90° Pulse	3.88 μs
Conctact time	2000 μs
Relaxation time	15 s
Presaturation delay	2 s

Table F.15: Acquisition parameters for Figure 4.8

Figure 4.9 (p44)

Nucleus	^{13}C
Frequency	75.4752958 MHz
CP-Nucleus	^{1}H
CP-Frequency	300.131500 MHz
Pulse Program	cp_selective
TD	2048
NS	2233
SW	22.727273 kHz
RG	912
90° Pulse	3.34 μs
Conctact time	2000 μs
Relaxation time	5 s
Spinlock time	10 ms

Table F.16: Acquisition parameters for Figure 4.9

Figure 4.10 (p45)

Nucleus	^{13}C
Frequency	75.4752958 MHz
CP-Nucleus	^{1}H
CP-Frequency	300.131500 MHz
Pulse Program	sat_cp
TD	2048
NS	512
SW	22.727273 kHz
RG	912
90° Pulse	3.88 μs
Conctact time	2000 μs
Relaxation time	10 s
Presaturation delay	2 s

Table F.17: Acquisition parameters for Figure 4.10

Figure 4.11 (p45)

Nucleus	^{13}C
Frequency	75.4752958 MHz
CP-Nucleus	^{1}H
CP-Frequency	300.131500 MHz
Pulse Program	cp_selective
TD	2048
NS	2233
SW	22.727273 kHz
RG	912
90° Pulse	3.34 μs
Conctact time	2000 μs
Relaxation time	5 s
Spinlock time	25 ms

Table F.18: Acquisition parameters for Figure 4.11

Appendix G

Source Code

G.1 Line Fitting

G.1.1 OptSimp.m

```
1  function [p, f_out, resid] = OptSimp(A, F, p_in, dp_in, n_max, acc, dispNum)
2    N = 0;
3    n_sum = 0;
4    p = p_in;
5    s45 = sin(pi/4);
6    x = A(:,1); data = A(:,2);
7    n=0;
8    maxDiff = 9.9e99;
9    dim = size(p)(1,1);
10   simpDim = dim+1;
11   dp = dp_in;
12   dp_diff_norm = 1; p_diff_norm = 1;
13   f   = feval(F, x, p);
14   res = sum((f - data).^2);
15   P        = zeros(dim, dim+1);
16   DP      = zeros(dim, dim+1);
17   simplex = zeros(dim, dim+1);
18   for i = 1:simpDim
19     for j = 1:dim
20       if (j == 1 && i == 1)
21         simplex(j,i) = -1;
22       elseif (j == i)
23         simplex(j,i) = -s45;
24       elseif (j > i)
25         simplex(j,i) = 0;
26       else
27         simplex(j,i) = s45;
28       endif
29     endfor
30   endfor
31
32
33   while(n<n_max && (maxDiff>acc))
34     simRes = [];
35     p_prev  = p;
36     dp_prev = dp;
37     for i = 1:simpDim
38       P(:,i)  = p;
39       DP(:,i) = dp;
```

```
40      endfor
41      P = P+(simplex.*DP);
42      for i = 1:simpDim
43        f = feval(F, x, P(:,i));
44        simRes(i,:) = sum((f - data).^2);
45      endfor
46      [minSim, minP] = min(simRes);
47
48      if (minSim < res)
49        p = p + (simplex(:,minP).*dp);
50        dp = dp * 1.01;
51        res = minSim;
52        n++;
53      else
54        dp = dp * 0.5;
55        n++;
56      endif
57      maxDiff = max(dp./p);
58    endwhile
59
60    f_out = feval(F, x, p);
61
62    p
63    n
64
65    disp("-- Degrees of Freedom --");
66    Regression = size(p)(1,1)
67    Residual = size(data)(1,1) - Regression - 1
68    Total = size(data)(1,1) - 1
69
70    disp("-- Variance (SSE & SSY) --");
71    sse = sum((f - data).^2);
72    ssy = sum((data - (sum(data)/size(data)(1,1))).^2);
73    Regression = ssy - sse
74    Residual = sse
75    Total = ssy
76
77    disp("-- Mean Squares --");
78    Regression = (ssy - sse) / size(p)(1,1)
79    Residual = sse / (size(data)(1,1) - size(p)(1,1) - 1)
80
81    F_tmp = F;
82    disp("---")
83    F = Regression / Residual
84    r_squared = (ssy-sse)/ssy
85    F = F_tmp;
86
87    resid = f_out - data;
88
89
90    if(dispNum < size(data)(1,1))
91      for i = 1:dispNum
92        Px(i)     = x(i);
93        Pdata(i)  = data(i);
94        Pf_out(i) = f_out(i);
95        Presid(i) = resid(i);
96      endfor
97    else
```

```
98        Px=x;
99        Pdata=data;
100       Pf_out=f_out;
101       Presid=resid;
102     endif
103
104     plot(Px, Pdata, "-;Data;", Px, Pf_out, "-;Fit;", Px, Presid, "-;Residuals;"\
105          ,Px,zeros(size(Pdata)(1,1),1));
106
107   endfunction
```

G.1.2 SpinDiff.m

```
1   A = load("fid-1-i");
2   [A_max, A_max_pos] = max(A(:,2));
3   [A_size, dummy]    = size(A);
4   i = A_max_pos;
5   j = 1;
6
7   while(i<=A_size)
8     B(j,:) = [j, A(i,2)];
9     i++;
10    j++;
11  endwhile
12
13
14  function y = f1(x,p)
15    if(p(4)>=0)
16      X = x+p(4);
17    else
18      X = x-p(4);
19    endif
20    pX = p(3)*X;
21    s = sin(pX) ./ pX;
22    y = p(1) * s .* exp(- ( p(2).^2 * X.^2 ) / 2 );
23  endfunction
24
25  function y = f2(x,p)
26    if(p(4)>=0)
27      X = x+p(4);
28    else
29      X = x-p(4);
30    endif
31    c = cos(p(3)*X);
32    y = p(1) * c .* exp(- ( p(2).^2 * X.^2 ) / 2 );
33  endfunction
34
35  function y = f3(x,p)
36    if(p(3)>=0)
37      X = x+p(3);
38    else
39      X = x-p(3);
40    endif
41    y = p(1) * exp(- ( p(2).^2 * X.^2 ) / 2 );
42  endfunction
43
44  function y = F13(x,p)
45    p1 = [p(1); p(2); p(3); p(6)];
46    p2 = [p(4); p(5); p(6)];
```

```
47    y = f1(x, p1) + f3(x, p2);
48  endfunction
49
50  function y = F23(x,p)  # 6 Pars
51    p1 = [p(1); p(2); p(3); p(6)];
52    p2 = [p(4); p(5); p(6)];
53    y = f2(x, p1) + f3(x, p2);
54  endfunction
55
56  function y = F33(x,p) # 5 Pars
57    p1 = [p(1); p(2); p(5)];
58    p2 = [p(3); p(4); p(5)];
59    y = f3(x, p1) + f3(x, p2);
60  endfunction
```

G.2 TLM-Simulator

The source code of the TLM-Simulator is also available at: `http://www.fklama.de/academic/TLM-Simulator.tar.bz2`

G.2.1 TLM-Sim.c

```
1   /*
2    *        TLM-Simulator v1.0
3    *
4    *  Author: Frederik Klama
5    *  Copyright 2010 Frederik Klama
6    *
7    *   This program is free software: you can redistribute it and/or modify
8    *  it under the terms of the GNU General Public License as published by
9    *  the Free Software Foundation, either version 3 of the License, or
10   *  (at your option) any later version.
11   *
12   *  This program is distributed in the hope that it will be useful,
13   *  but WITHOUT ANY WARRANTY; without even the implied warranty of
14   *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
15   *  GNU General Public License for more details.
16   *
17   *  You should have received a copy of the GNU General Public License
18   *  along with this program.  If not, see <http://www.gnu.org/licenses/>.
19   *
20   *
21   *  Description:
22   *  This program reads all needed parameters from the configuration file
23   *  'tlm.conf' in the directory it is called from.
24   *  The configuration file can be generated using 'confGen.pl'
25   *  The program then does a Transmission-Line-Matrix simulation and
26   *  outputs the values of the main box and the bulk for each iteration
27   *  to STDOUT.
28   *  It is recommended to redirect the output into a file.
29   *  Depending on the verbose-value, an additional file 'map.txt' is
30   *  generated and contains all potentials from each iteration. It is
31   *  intended for debugging and demonstration purposes.
32   */
33
34  #include <stdio.h>
35  #include <string.h>
```

```
36  #include <stdlib.h>
37  #include <math.h>
38  #include <unistd.h>
39  #include <errno.h>
40
41  #include "common.h"
42  #include "parser.h"
43  #include "dataStruct.h"
44  #include "output.h"
45  #include "worker.h"
46  #include "fillBoxMag.h"
47
48  // The configuration file is hardcoded here.
49  // If you would like to use a different filename,
50  // just change this constant.
51  #define CONFIG_FILE "tlm.conf"
52
53  // Hardcoded ln(2)
54  #define LN2 0.6931471806
55
56  inline char hexChar(int);
57  inline void updateFN(char*, long long);
58  inline void errOut(char*);
59  inline void die(char*);
60
61  int
62  main()
63  {
64    struct confStruct c;
65
66    FILE* configFile;
67    FILE* verbOutFile;
68
69    unsigned long n;
70
71    double* dataArray;
72    double* Vi;
73    double* Vs;
74    double* BoxIntensity;
75    double* BulkIntensity;
76    double* initPot;
77    double* R;
78    double* Z;
79
80    double sumBox, sumBulk;
81
82    int dummy;
83
84    char filename[16];
85
86    unsigned long i;
87
88  #ifdef DEBUG
89    char arrFile[16];
90    FILE* arrOutFile;
91  #endif
92
93    // Read Config
```

```
94    fprintf(stderr, "Reading config:"); fflush(stderr);
95    configFile = fopen(CONFIG_FILE, "r");
96    parseConfig(&c, configFile);
97    fclose(configFile);
98    fprintf(stderr, " done\n"); fflush(stderr);
99
100   #ifdef DEBUG
101     printf("dim     = %d\nXsize   = %d\n", c.dim, c.Xsize);
102     printf("Ysize   = %d\nZsize   = %d\n", c.Ysize, c.Zsize);
103     printf("Box0    = %d\nBox1    = %d\n", c.Box0, c.Box1);
104     printf("Box2    = %d\nBox3    = %d\n", c.Box2, c.Box3);
105     printf("Box4    = %d\nBox5    = %d\n", c.Box4, c.Box5);
106     printf("steps   = %d\nT1      = %d\n", c.steps, c.T1);
107     printf("verbose = %d\nboundary= %d\n", c.verbose, c.boundary);
108     printf("\nx = %d\ny = %d\nz = %d\n", c.x, c.y, c.z);
109     printf("\n==== Initialising data structure.\n");
110     fflush(stdout);
111   #endif
112
113     // Initialise data structures
114     dataArray = initDataArray(c);
115     Vi = getVi(c, dataArray);
116     Vs = getVs(c, dataArray);
117     BoxIntensity  = (double*) malloc(sizeof(double) * c.steps);
118     for(i=0;i<c.steps;i++)
119       *(BoxIntensity+i) = 0.0;
120     BulkIntensity = (double*) malloc(sizeof(double) * c.steps);
121     for(i=0;i<c.steps;i++)
122       *(BulkIntensity+i) = 0.0;
123     fprintf(stderr, "Initialised datastructures.\n"); fflush(stderr);
124
125   #if DETAILED_3D_MAP == 1
126     if(c.dim==3)
127       strcpy(filename, "map00000000.txt");
128     else
129   #endif
130       strcpy(filename, "map.txt");
131   #ifdef DEBUG
132     strcpy(arrFile, "arr00000000.txt");
133   #endif
134
135     // Generate initial magnetization
136     fillBoxMag(c, Vi);
137
138       if(c.verbose>1)
139     {
140       verbOutFile = fopen(filename, "w");
141       if(verbOutFile == NULL)
142         die("Could not open file for writing.\n");
143     }
144
145     fprintf(stderr, "Starting Simulation.\n"); fflush(stderr);
146
147     // Main Iteration Loop
148     for(n=0; n<c.steps; n++)
149     {
150       // Print Progress
151       if(n%10 == 0 && c.verbose>0)
```

```
152       {
153         fprintf(stderr, "\n%lu/%lu", n, c.steps);
154         if(n%100==0)
155           fflush(stderr);
156       }
157
158   #if DETAILED_3D_MAP == 1
159       if(c.dim==3 && c.verbose>1)
160       {
161         updateFN(filename, n);
162         verbOutFile = fopen(filename, "w");
163         if(verbOutFile == NULL)
164           die("Could not open file for writing.\n");
165       }
166   #endif
167
168       if(c.verbose>1)
169         errOut(" +");
170
171       // Calculate Phi values and save intesities
172       calcSums(c, n, Vi, &sumBox, &sumBulk, BoxIntensity, BulkIntensity);
173
174       if(c.verbose>1)
175         errOut(">");
176
177       // Generate debugging output
178       if(c.verbose>1)
179         outputDetailedData(c, verbOutFile, n, Vi, sumBox, sumBulk);
180   #if DETAILED_3D_MAP == 1
181       if(c.dim==3 && c.verbose>1)
182       {
183         dummy = fclose(verbOutFile);
184         if(dummy != 0)
185           die("Could not close file.\n");
186       }
187   #endif
188
189       if(c.verbose>1)
190         errOut("S");
191
192       /*
193        * The actual work is done here
194        * c.model==0  => Link Line
195        * c.model==1  => Link Resistor
196        */
197       if(c.model==1) {
198         LRscatter(c, Vi, Vs);
199         if(c.verbose>1)
200           errOut("C");
201         LRconnect(c, Vi, Vs);
202       } else {
203         LLscatter(c, Vi, Vs);
204         if(c.verbose>1)
205           errOut("C");
206         LLconnect(c, Vi, Vs);
207       }
208
209       // T1 decay if set
```

```
210       if(c.T1>0)
211       {
212         if(c.verbose>1)
213           errOut("T");
214         T1decay(c, Vi, (LN2 / (double)c.T1));
215       }
216   #ifdef DEBUG
217       fclose(arrOutFile);
218   #endif
219
220       if(c.verbose>1)
221         errOut("=");
222
223     }
224
225     // Print intesity list
226     outputGraphData(c, BoxIntensity, BulkIntensity);
227   #if DETAILED_3D_MAP == 1
228     if(c.dim<3 && c.verbose>1)
229   #else
230     if(c.verbose>1)
231   #endif
232       fclose(verbOutFile);
233
234     // Clean up and free memory
235     free(BoxIntensity);
236     free(BulkIntensity);
237     destroyDataArray(dataArray);
238
239   }
240
241   inline void updateFN(char* filename, long long n)
242   {
243     // filename = "map00000000.txt";
244     //             0123456789a
245
246     filename[0x0a] = hexChar( n & 0x0000000f );
247     filename[0x09] = hexChar((n & 0x000000f0) >> 4);
248     filename[0x08] = hexChar((n & 0x00000f00) >> 8);
249     filename[0x07] = hexChar((n & 0x0000f000) >> 12);
250     filename[0x06] = hexChar((n & 0x000f0000) >> 16);
251     filename[0x05] = hexChar((n & 0x00f00000) >> 20);
252     filename[0x04] = hexChar((n & 0x0f000000) >> 24);
253     filename[0x03] = hexChar((n & 0xf0000000) >> 28);
254   }
255
256   inline char hexChar(int i)
257   {
258     if(i<10 && i>=0)
259       return (char) i + 0x30;
260     else if(i<16)
261       return (char) i-10 + 0x61;
262     else
263       return 'x';
264   }
265
266   inline void errOut(char* text)
267   {
```

```
268     fprintf(stderr, text);
269     fflush(stderr);
270   }
271
272   inline void die(char* text)
273   {
274     fprintf(stderr, text);
275     fprintf(stderr, "Error: %i\n", errno);
276     fflush(stderr);
277     exit(1);
278   }
279
280   // vim:set ts=2 sw=2:
```

G.2.2 common.h

```
1   /*
2    *  File: common.h
3    *
4    *  Author: Frederik Klama
5    *  Copyright 2010 Frederik Klama
6    *
7    *  This file is part of TLM-Simulator.
8    *
9    *  TLM-Simulator is free software: you can redistribute it and/or modify
10   *  it under the terms of the GNU General Public License as published by
11   *  the Free Software Foundation, either version 3 of the License, or
12   *  (at your option) any later version.
13   *
14   *  TLM-Simulator is distributed in the hope that it will be useful,
15   *  but WITHOUT ANY WARRANTY; without even the implied warranty of
16   *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
17   *  GNU General Public License for more details.
18   *
19   *  You should have received a copy of the GNU General Public License
20   *  along with TLM-Simulator.  If not, see <http://www.gnu.org/licenses/>.
21   *
22   */
23  #define DETAILED_3D_MAP 0
24
25  struct confStruct {
26    /*
27     * This structure is universially used to pass the
28     * simulation parameters around.
29     */
30    short dim;
31    short Xsize;
32    short Ysize;
33    short Zsize;
34    unsigned long steps;
35    unsigned long T1;
36    char verbose;
37    char model;
38    char boundary;
39    long x;
40    long y;
41    long z;
42    long Y;
43    long Z;
```

```
44    short Box0;
45    short Box1;
46    short Box2;
47    short Box3;
48    short Box4;
49    short Box5;
50    double round;
51    double* initPot;
52    double* pR;
53    double* pZ;
54  };
55
56  /*    4   2
57   *     \  ^
58   *      \ |
59   *       \|
60   * 0 <----+----> 1
61   *        |\
62   *        | \
63   *        V  \
64   *        3   5
65   */
66
67
68  // vim:set ts=2 sw=2:
```

G.2.3 StringTools.h

```
1   /*
2    *  File: StringTools.h
3    *
4    *  Author: Frederik Klama
5    *  Copyright 2010 Frederik Klama
6    *
7    *  This file is part of TLM-Simulator.
8    *
9    *  TLM-Simulator is free software: you can redistribute it and/or modify
10   *  it under the terms of the GNU General Public License as published by
11   *  the Free Software Foundation, either version 3 of the License, or
12   *  (at your option) any later version.
13   *
14   *  TLM-Simulator is distributed in the hope that it will be useful,
15   *  but WITHOUT ANY WARRANTY; without even the implied warranty of
16   *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
17   *  GNU General Public License for more details.
18   *
19   *  You should have received a copy of the GNU General Public License
20   *  along with TLM-Simulator.  If not, see <http://www.gnu.org/licenses/>.
21   *
22   */
23
24  char*
25  clipStr(
26    char*,
27    long
28  );
29
30  short
31  splitEqual(
```

```
32     char *,
33      char **,
34      char **
35   );
36
37   // vim:set ts=2 sw=2:
```

G.2.4 StringTools.c

```
1    /*
2     *  File: StringTools.c
3     *
4     *  Author: Frederik Klama
5     *  Copyright 2010 Frederik Klama
6     *
7     *  This file is part of TLM-Simulator.
8     *
9     *  TLM-Simulator is free software: you can redistribute it and/or modify
10    *  it under the terms of the GNU General Public License as published by
11    *  the Free Software Foundation, either version 3 of the License, or
12    *  (at your option) any later version.
13    *
14    *  TLM-Simulator is distributed in the hope that it will be useful,
15    *  but WITHOUT ANY WARRANTY; without even the implied warranty of
16    *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
17    *  GNU General Public License for more details.
18    *
19    *  You should have received a copy of the GNU General Public License
20    *  along with TLM-Simulator.  If not, see <http://www.gnu.org/licenses/>.
21    *
22    */
23
24   #include <stdlib.h>
25   #include <string.h>
26
27   char *
28   clipStr(
29       char * in,
30       long length
31       )
32   {
33     /*
34      * This function removes any whitespaces from the beginning and end
35      * of a string
36      */
37     char   *  in_p = in;
38     char  * out;
39     char  * out_p;
40     char  *  end_p = in + length;
41     long    i;
42     long    l;
43     while((*(in_p) == ' ' || *(in_p) == '\t' || *(in_p) == '\n' ||
44           *(in_p) == '\r') && (in_p-in) < length && *in_p != '\0')
45       in_p++;
46     while((*(end_p) == ' ' || *(end_p) == '\t' || *(end_p) == '\n' ||
47           *(end_p) == '\r') && end_p>=in)
48       end_p--;
49     l = end_p - in_p + 1;
50     out = (char *) malloc(sizeof(char) * (l+1));
```

```
51    out_p = out;
52    for(i=0;i<l;i++)
53    {
54      *(out_p+i) = *(in_p+i);
55    }
56    *(out_p+l) = '\0';
57    return out;
58  }
59
60  short
61  splitEqual(
62      char * in,
63      char ** par,
64       char ** data
65      )
66  {
67    /*
68     * This function takes a string and splits it into two substrings.
69     * One before the equals sign and another after.
70     * It uses clipStr to remove whitespaces from the beginning and
71     * end of the substrings.
72     */
73    char     * in_p;
74    char     * eq_p;
75    long      eq_pos;
76    eq_p = in_p = in;
77    while(*(eq_p) != '\0' && *(eq_p) != '=')
78      eq_p++;
79    if(*(eq_p) == '\0')
80      return 0;
81    eq_pos = eq_p - in;
82    *par = clipStr(in, eq_pos-1);
83    *data = clipStr(eq_p+1,strlen(eq_p+1));
84    return 1;
85  }
86
87  // vim:set ts=2 sw=2:
```

G.2.5 dataStruct.h

```
1   /*
2    *  File: dataStruct.h
3    *
4    *  Author: Frederik Klama
5    *  Copyright 2010 Frederik Klama
6    *
7    *  This file is part of TLM-Simulator.
8    *
9    *  TLM-Simulator is free software: you can redistribute it and/or modify
10   *  it under the terms of the GNU General Public License as published by
11   *  the Free Software Foundation, either version 3 of the License, or
12   *  (at your option) any later version.
13   *
14   *  TLM-Simulator is distributed in the hope that it will be useful,
15   *  but WITHOUT ANY WARRANTY; without even the implied warranty of
16   *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
17   *  GNU General Public License for more details.
18   *
19   *  You should have received a copy of the GNU General Public License
```

```
20   *  along with TLM-Simulator.  If not, see <http://www.gnu.org/licenses/>.
21   *
22   */
23
24  double*
25  initDataArray(
26    struct confStruct
27  );
28
29  double*
30  getVi(
31    struct confStruct,
32    double*
33  );
34
35  double*
36  getVs(
37    struct confStruct,
38    double*
39  );
40
41  void
42  destroyDataArray(
43    double*
44  );
45
```

G.2.6 dataStruct.c

```
1   /*
2    *  File: dataStruct.c
3    *
4    *  Author: Frederik Klama
5    *  Copyright 2010 Frederik Klama
6    *
7    *  This file is part of TLM-Simulator.
8    *
9    *  TLM-Simulator is free software: you can redistribute it and/or modify
10   *  it under the terms of the GNU General Public License as published by
11   *  the Free Software Foundation, either version 3 of the License, or
12   *  (at your option) any later version.
13   *
14   *  TLM-Simulator is distributed in the hope that it will be useful,
15   *  but WITHOUT ANY WARRANTY; without even the implied warranty of
16   *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
17   *  GNU General Public License for more details.
18   *
19   *  You should have received a copy of the GNU General Public License
20   *  along with TLM-Simulator.  If not, see <http://www.gnu.org/licenses/>.
21   *
22   */
23  #include <stdio.h>
24  #include <math.h>
25  #include <stdlib.h>
26
27  #include "common.h"
28  #include "dataStruct.h"
29
30
```

```
31
32  double*
33  initDataArray(
34      struct confStruct c
35      )
36  {
37    /*
38     * This function allocates the memory for and
39     * initializes the main data array
40     */
41    double* out;
42    long long size = 1;
43
44    // Determine size of data array
45    switch(c.dim)
46    {
47      case 3: size *= c.Zsize;
48      case 2: size *= c.Ysize;
49      case 1: size *= c.Xsize * (4 * c.dim);
50    }
51
52    // Allocate just a little more, to avoid segfaults
53    size += 100;
54  #ifdef DEBUG
55    printf("size           = %d\n", size);
56    printf("sizeof(double) = %d\n", sizeof(double));
57    printf("Mem usage      = %d\n", size * sizeof(double));
58  #endif
59
60    // Actually allocate the memory
61    out = (double*) malloc(sizeof(double) * size);
62
63    // Write every value to 0.0 to initialize array
64    long long i;
65    for(i=0; i<size; i++)
66      *(out) = 0.0;
67
68    // Return pointer to array
69    return out;
70  }
71
72  /*
73   * The next two functions split the large data array
74   * into two parts. One for incident pulses and another
75   * for scattering pulses
76   * They are given the pointer to main data array as
77   * input and return a pointer to be used for data
78   */
79
80  double*
81  getVi(
82      struct confStruct c,
83      double* in
84      )
85  {
86    // Incident pulses are stored in the first half
87  #ifdef DEBUG
88    printf("Vi         = 0x%x\n", in);
```

```
89  #endif
90    return in;
91  }
92
93  double*
94  getVs(
95      struct confStruct c,
96      double* in
97      )
98  {
99    // Scattering pulses in the second half
100   double* out;
101   long long offset = 1;
102
103   // Determine system size again
104   switch(c.dim)
105   {
106     case 3: offset *= c.Zsize;
107     case 2: offset *= c.Ysize;
108     case 1: offset *= c.Xsize * (2 * c.dim);
109   }
110
111   // Add a little bit of buffer space
112   offset+=50;
113
114   // Actually set the pointer for Vs by ofsetting
115   // the pointer out relative to in
116   out = in + offset;
117
118 #ifdef DEBUG
119   printf("Vs         = 0x%x\n", out);
120   printf("offset     = %d\n", offset);
121   printf("offset * 8 = 0x%x\n", offset*8);
122 #endif
123   return (double*) out;
124 }
125
126
127 void
128 destroyDataArray(
129     double* in
130     )
131 {
132   // Simply frees the memory
133   free(in);
134 }
135
136 // vim:set ts=2 sw=2:
```

G.2.7 fillBoxMag.h

```
1  /*
2   *  File: fillBoxMag.h
3   *
4   *  Author: Frederik Klama
5   *  Copyright 2010 Frederik Klama
6   *
7   *  This file is part of TLM-Simulator.
8   *
```

```
 9    *  TLM-Simulator is free software: you can redistribute it and/or modify
10    *  it under the terms of the GNU General Public License as published by
11    *  the Free Software Foundation, either version 3 of the License, or
12    *  (at your option) any later version.
13    *
14    *  TLM-Simulator is distributed in the hope that it will be useful,
15    *  but WITHOUT ANY WARRANTY; without even the implied warranty of
16    *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
17    *  GNU General Public License for more details.
18    *
19    *  You should have received a copy of the GNU General Public License
20    *  along with TLM-Simulator.  If not, see <http://www.gnu.org/licenses/>.
21    *
22    */
23
24   void
25   fillBoxMag(
26     struct confStruct,
27     double*
28   );
29
30   // vim:set ts=2 sw=2:
```

G.2.8 fillBoxMag.c

```
 1   /*
 2    *  File: fillBoxMag.c
 3    *
 4    *  Author: Frederik Klama
 5    *  Copyright 2010 Frederik Klama
 6    *
 7    *  This file is part of TLM-Simulator.
 8    *
 9    *  TLM-Simulator is free software: you can redistribute it and/or modify
10    *  it under the terms of the GNU General Public License as published by
11    *  the Free Software Foundation, either version 3 of the License, or
12    *  (at your option) any later version.
13    *
14    *  TLM-Simulator is distributed in the hope that it will be useful,
15    *  but WITHOUT ANY WARRANTY; without even the implied warranty of
16    *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
17    *  GNU General Public License for more details.
18    *
19    *  You should have received a copy of the GNU General Public License
20    *  along with TLM-Simulator.  If not, see <http://www.gnu.org/licenses/>.
21    *
22    */
23
24   #include <stdio.h>
25
26   #include "common.h"
27   #include "fillBoxMag.h"
28
29   void
30   fillBoxMag(
31       struct confStruct c,
32       double* Vi
33       )
34   {
```

```
35      /*
36       * This function generates the initial magnetization
37       * by setting the incidence pulses.
38       */
39
40      long x = c.x;
41      long y = c.y;
42      long z = c.z;
43      double *iP;
44
45      iP = c.initPot;
46
47      switch(c.dim)
48      {
49      case 1:;
50        {
51          register long long i;
52          for(i=0;i<c.Xsize;i++)
53          {
54            *(Vi + i * x)     = *(iP+i);
55            *(Vi + i * x + 1)  = *(iP+i);
56          }
57        }
58        break;
59      case 2:;
60        {
61          register long long i;
62          for(i=0;i<(c.Xsize * c.Ysize);i++)
63          {
64            register double *ptr;
65            ptr = Vi + i*x;
66            *ptr = *(iP+i);       ++ptr;
67            *ptr = *(iP+i);       ++ptr;
68            *ptr = *(iP+i); ++ptr;
69            *ptr = *(iP+i);
70
71          }
72        }
73        break;
74      case 3:;
75        {
76          long i;
77          long j;
78          long k;
79          double *ptr;
80          for(k=0;k<(c.Zsize);k++)
81          {
82            for(j=0;j<(c.Ysize);j++)
83            {
84              for(i=0;i<(c.Xsize);i++)
85              {
86                ptr = Vi + i*x + j*y + k*z;
87                *ptr = *(iP+i); ++ptr;
88                *ptr = *(iP+i); ++ptr;
89                *ptr = *(iP+i); ++ptr;
90                *ptr = *(iP+i); ++ptr;
91                *ptr = *(iP+i); ++ptr;
92                *ptr = *(iP+i);
```

```
93              }
94            }
95          }
96
97        break;
98        }
99      }
100   }
101
102
103
104   // vim:set ts=2 sw=2:
```

G.2.9 output.h

```
1   /*
2    *  File: output.h
3    *
4    *  Author: Frederik Klama
5    *  Copyright 2010 Frederik Klama
6    *
7    *  This file is part of TLM-Simulator.
8    *
9    *  TLM-Simulator is free software: you can redistribute it and/or modify
10   *  it under the terms of the GNU General Public License as published by
11   *  the Free Software Foundation, either version 3 of the License, or
12   *  (at your option) any later version.
13   *
14   *  TLM-Simulator is distributed in the hope that it will be useful,
15   *  but WITHOUT ANY WARRANTY; without even the implied warranty of
16   *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
17   *  GNU General Public License for more details.
18   *
19   *  You should have received a copy of the GNU General Public License
20   *  along with TLM-Simulator.  If not, see <http://www.gnu.org/licenses/>.
21   *
22   */
23
24  void
25  outputDetailedData(
26      struct confStruct,
27       FILE*,
28      long long,
29      double*,
30      double,
31      double
32  );
33
34  void
35  outputGraphData(
36      struct confStruct,
37      double*,
38      double*
39  );
40
41  void
42  outputFullDataArray(
43      struct confStruct,
44       FILE*,
```

```
45      long long,
46      double*,
47      double*
48  );
49
50  // vim:set ts=2 sw=2:
```

G.2.10 output.c

```
1   /*
2    *  File: output.c
3    *
4    *  Author: Frederik Klama
5    *  Copyright 2010 Frederik Klama
6    *
7    *  This file is part of TLM-Simulator.
8    *
9    *  TLM-Simulator is free software: you can redistribute it and/or modify
10   *  it under the terms of the GNU General Public License as published by
11   *  the Free Software Foundation, either version 3 of the License, or
12   *  (at your option) any later version.
13   *
14   *  TLM-Simulator is distributed in the hope that it will be useful,
15   *  but WITHOUT ANY WARRANTY; without even the implied warranty of
16   *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
17   *  GNU General Public License for more details.
18   *
19   *  You should have received a copy of the GNU General Public License
20   *  along with TLM-Simulator.  If not, see <http://www.gnu.org/licenses/>.
21   *
22   */
23
24  #include <stdio.h>
25  #include <stdlib.h>
26  #include <string.h>
27
28  #include "common.h"
29  #include "output.h"
30
31
32  double normedTotal;
33
34  void
35  outputDetailedData(
36      struct confStruct c,
37       FILE* out,
38      long long n,
39      double* Vi,
40       double sumBox,
41      double sumBulk
42      )
43  {
44    /*
45     * This function prints the debugging output if debugging is
46     * switched on.
47     */
48    long x = c.x;
49    long y = c.y;
50    long z = c.z;
```

```
51
52      long i;
53      long j;
54      long k;
55      switch(c.dim)
56      {
57      case 3:;
58        #if DETAILED_3D_MAP == 1
59        {
60          ////////////////////
61          // print results //
62          ////////////////////
63          //printf("i=%8d\n   ",i);
64
65          // layer loop
66          for(k=0; k<c.Zsize; k++)
67          {
68            fprintf(out, "===== layer %d =====\n\n", k);
69
70            fprintf(out, "   ");
71
72            // print column numbers
73
74            for(i=0; i<c.Xsize; i++)
75            {
76              if((i==c.Box0 || i==c.Box1+1) && k>=c.Box4 && k<=c.Box5)
77                fprintf(out, "   ");
78              fprintf(out, "%8d", i);
79            }
80
81            fprintf(out, "\n");
82
83            // print horizontal box lines
84            for(j=0; j<c.Ysize; j++)
85            {
86              if((j==c.Box2 || j==c.Box3+1) && k>=c.Box4 && k<=c.Box5)
87              {
88                fprintf(out, "   ");
89                for(i=0; i<c.Xsize; i++)
90                {
91                  if(i<c.Box0 || i>c.Box1)
92                    fprintf(out, "        ");
93                  else
94                  {
95                    if(i==c.Box0) fprintf(out, " +-");
96                    fprintf(out, "--------");
97                    if(i==c.Box1) fprintf(out, "-+ ");
98                  }
99                }
100               fprintf(out, "\n");
101             }
102
103             fprintf(out, "%3d", j); // print row number
104             for(i=0; i<c.Xsize; i++)
105             {
106               if((i==c.Box0 || i==c.Box1+1) && k>=c.Box4 && k<=c.Box5)
107               {
108                 if(j>=c.Box2 && j<=c.Box3)
```

```
109                 fprintf(out, " | "); // print vertical box lines
110               else
111                 fprintf(out, "   ");
112             }
113             double v = *(Vi+(i*x)+(j*y)+(k*z))   +\
114                        *(Vi+(i*x)+(j*y)+(k*z)+1) +\
115                        *(Vi+(i*x)+(j*y)+(k*z)+2) +\
116                        *(Vi+(i*x)+(j*y)+(k*z)+3) +\
117                        *(Vi+(i*x)+(j*y)+(k*z)+4) +\
118                        *(Vi+(i*x)+(j*y)+(k*z)+5);
119             char valueString[140] = "";
120             sprintf(valueString, "%8.3e", v);
121             fprintf(out, "%s", valueString);
122           }
123           fprintf(out, "\n");
124         }
125         fprintf(out, "\n\n");
126       }
127
128       if(c.verbose>2)
129       {
130         // summation
131         long long  countBox  = (c.Box1 - c.Box0 + 1) * \
132                                (c.Box3 - c.Box2 + 1) * \
133                                (c.Box5 - c.Box4 + 1);
134         long long countBulk = (c.Xsize * c.Ysize * c.Zsize) - countBox;
135         double total = (sumBulk+sumBox)/(countBulk+countBox);
136         if(n==0) normedTotal=total;
137         fprintf(out, "\n\n");
138         fprintf(out, "+====================================+\n");
139         fprintf(out, "H             S U M S              H\n");
140         fprintf(out, "+====================================+\n");
141         fprintf(out, "| Bulk:Box     = %8.3f :%8.3f |\n",\
142             sumBulk/countBulk, \
143             sumBox /countBox );
144         fprintf(out, "| Total        = %8.3f           |\n",
145             total );
146         fprintf(out, "| Normed Total = %8.6f           |\n",\
147             total/normedTotal );
148         fprintf(out, "+------------------------------------+\n");
149       }
150     }
151     #else
152     fprintf(out, "+---------------------------------------+\n");
153     fprintf(out, "|               n =%7li             |\n",n);
154     fprintf(out, "+---------------------------------------+\n");
155     for(k=0; k<c.Zsize; k++)
156     {
157       fprintf(out, "**** n=%li, z=%li ****\n", n, k);
158       for(j=0; j<c.Ysize; j++)
159       {
160         fprintf(out, "**** n=%li, z=%li, y=%li ****\n", n, k, j);
161
162         for(i=0; i<c.Xsize; i++)
163         {
164           double v;
165           if(!(i%10))
166             fprintf(out, "%4li-%4li:", i, (i+9));
```

```
167             v  = *(Vi+(i*c.x)+(j*c.y)+(k*c.z));
168             v += *(Vi+(i*c.x)+(j*c.y)+(k*c.z)+1);
169             v += *(Vi+(i*c.x)+(j*c.y)+(k*c.z)+2);
170             v += *(Vi+(i*c.x)+(j*c.y)+(k*c.z)+3);
171             v += *(Vi+(i*c.x)+(j*c.y)+(k*c.z)+4);
172             v += *(Vi+(i*c.x)+(j*c.y)+(k*c.z)+5);
173             fprintf(out, "%8.3lf ", v);
174             if((i%10)==9)
175               fprintf(out, "\n");
176           }
177         }
178       }
179       #endif
180
181       break;
182
183     case 2:;
184       {
185         fprintf(out, "------{ n=%d }-----\n", n);
186
187         fprintf(out, "   ");
188
189         // print column numbers
190         for(i=0; i<c.Xsize; i++)
191         {
192           if(i==c.Box0 || i==c.Box1+1) fprintf(out, "   ");
193           fprintf(out, "%8d", i);
194         }
195
196         fprintf(out, "\n");
197
198         // print horizontal box lines
199         for(j=0; j<c.Ysize; j++)
200         {
201           if(j==c.Box2 || j==c.Box3+1)
202           {
203             fprintf(out, "   ");
204             for(i=0; i<c.Xsize; i++)
205             {
206               if(i<c.Box0 || i>c.Box1)
207                 fprintf(out, "        ");
208               else
209               {
210                 if(i==c.Box0) fprintf(out, " +-");
211                 fprintf(out, "--------");
212                 if(i==c.Box1) fprintf(out, "-+ ");
213               }
214             }
215             fprintf(out, "\n");
216           }
217
218           fprintf(out, "%3d", j); // print row number
219           for(i=0; i<c.Xsize; i++)
220           {
221             double v;
222             if(i==c.Box0 || i==c.Box1+1)
223             {
224               if(j>=c.Box2 && j<=c.Box3)
```

```
225                 fprintf(out, " | "); // print vertical box lines
226               else
227                 fprintf(out, "   ");
228             }
229             v = *(Vi+(i*x)+(j*y))   +
230                 *(Vi+(i*x)+(j*y)+1) +
231                 *(Vi+(i*x)+(j*y)+2) +
232                 *(Vi+(i*x)+(j*y)+3);
233             fprintf(out, "%8.3f", v);
234           }
235           fprintf(out, "\n");
236         }
237         fprintf(out, "\n\n");
238       }
239       break;
240
241     case 1:;
242       {
243         if(n % 25 == 0)
244         {
245           fprintf(out, "\n          ");
246           for(i=0; i<c.Xsize; i++)
247           {
248             if(i==c.Box0 || i==c.Box1+1)
249               fprintf(out, "| ");
250             fprintf(out, "%8d", i);
251           }
252           fprintf(out, "\n");
253         }
254         fprintf(out, "%8d: ", n);
255         for(i=0; i<c.Xsize; i++)
256         {
257           if(i==c.Box0 || i==c.Box1+1)
258             fprintf(out, "| ");
259           fprintf(out, "%8.3f", (*(Vi+(i*x))+ *(Vi+(i*x)+1)));
260         }
261         fprintf(out, "\n");
262         break;
263       }
264     }
265   }
266
267   void
268   outputGraphData(
269       struct confStruct c,
270        double* data1In,
271        double* data2In
272       )
273   {
274     /*
275      * This function prints the magnetization amounts for each iteration.
276      * The main output of the simulator.
277      */
278     unsigned long i;
279     double *d1 = data1In;
280     double *d2 = data2In;
281     for(i=0;i<c.steps;i++)
282     {
```

```
283       printf("%.6f\t%.6f\n", *d1, *d2);
284       ++d1; ++d2;
285     }
286   }
287
288   void
289   outputFullDataArray(
290       struct confStruct c,
291       FILE* out,
292       long long n,
293       double* Vi,
294       double* Vs
295       )
296   {
297     /*
298      * This function was used to output the full data array
299      * It is currently not used anywhere, but was essential
300      * during the inital debugging of the code.
301      */
302     long i, j, k, l;
303     long x = c.x;
304     long y = c.y;
305     long z = c.z;
306     for(k=0;k<c.Zsize;k++)
307       for(j=0;j<c.Ysize;j++)
308         for(i=0;i<c.Xsize;i++)
309         {
310           long long off = i*x+j*y+k*z;
311           fprintf(out, "i=%6d  \t", i);
312           fprintf(out, "j=%6d  \t", j);
313           fprintf(out, "k=%6d\n", k);
314           fprintf(out, "n | Vi       | Vs\n");
315           for(l=0;l<6;l++)
316           {
317             fprintf(out, "%1d", l);
318             fprintf(out, " | %8.4f | ", *(Vi+off+l));
319             fprintf(out, "%8.4f", *(Vs+off+l));
320             if(k>0 && j>0 && i>0 && i<c.Xsize-1 &&
321                 j<c.Ysize-1 && k<c.Zsize-1)
322             {
323               switch(l)
324               {
325                 case 0: fprintf(out, " | %8.4f\n", *(Vs+off-x+1));
326                         break;
327                 case 1:  fprintf(out, " | %8.4f\n", *(Vs+off+x));
328                         break;
329                 case 2: fprintf(out, " | %8.4f\n", *(Vs+off-y+3));
330                         break;
331                 case 3:  fprintf(out, " | %8.4f\n", *(Vs+off+y+2));
332                         break;
333                 case 4:  fprintf(out, " | %8.4f\n", *(Vs+off-z+5));
334                         break;
335                 case 5: fprintf(out, " | %8.4f\n", *(Vs+off+z+4));
336               }
337             }
338             else
339               fprintf(out, "\n");
340           }
```

```
341           fprintf(out, "====================\n");
342           fflush(out);
343         }
344   }
345
346
347   // vim:set ts=2 sw=2:
```

G.2.11 parser.h

```
1   /*
2    *  File: parser.h
3    *
4    *  Author: Frederik Klama
5    *  Copyright 2010 Frederik Klama
6    *
7    *  This file is part of TLM-Simulator.
8    *
9    *  TLM-Simulator is free software: you can redistribute it and/or modify
10   *  it under the terms of the GNU General Public License as published by
11   *  the Free Software Foundation, either version 3 of the License, or
12   *  (at your option) any later version.
13   *
14   *  TLM-Simulator is distributed in the hope that it will be useful,
15   *  but WITHOUT ANY WARRANTY; without even the implied warranty of
16   *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
17   *  GNU General Public License for more details.
18   *
19   *  You should have received a copy of the GNU General Public License
20   *  along with TLM-Simulator.  If not, see <http://www.gnu.org/licenses/>.
21   *
22   */
23
24  void
25  parseConfig(
26    struct confStruct*,
27    FILE*
28  );
29
30
31  // vim:set ts=2 sw=2:
```

G.2.12 parser.c

```
1   /*
2    *  File: parser.c
3    *
4    *  Author: Frederik Klama
5    *  Copyright 2010 Frederik Klama
6    *
7    *  This file is part of TLM-Simulator.
8    *
9    *  TLM-Simulator is free software: you can redistribute it and/or modify
10   *  it under the terms of the GNU General Public License as published by
11   *  the Free Software Foundation, either version 3 of the License, or
12   *  (at your option) any later version.
13   *
14   *  TLM-Simulator is distributed in the hope that it will be useful,
15   *  but WITHOUT ANY WARRANTY; without even the implied warranty of
```

```
16    *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
17    *  GNU General Public License for more details.
18    *
19    *  You should have received a copy of the GNU General Public License
20    *  along with TLM-Simulator.  If not, see <http://www.gnu.org/licenses/>.
21    *
22    */
23
24   #include <stdio.h>
25   #include <string.h>
26   #include <math.h>
27   #include <stdlib.h>
28
29   #include "common.h"
30   #include "parser.h"
31   #include "StringTools.h"
32
33   #define FLUSH fflush(stdout)
34
35   void
36   parseConfig(
37       struct confStruct* out,
38        FILE* confFile
39       )
40   {
41     /*
42      * This function parses the configuration file and sets
43      * the values in c.
44      */
45     char       line[1024];
46     char*      ptr;
47     char*      numPtr;
48     char*      par;
49     char*      data;
50     char      pos;
51     char      numC[80];
52     double*    iP;
53     double*    initPot;
54     double*    R;
55      double*    R_orig;
56     double*    Z;
57      double*    Z_orig;
58     long      i;
59
60     // Defaults
61     out->dim        = 1;
62     out->Xsize      = 4;
63     out->Ysize      = 0;
64     out->Zsize      = 0;
65     out->steps      = 10;
66     out->T1          = 0;
67     out->model      = 0;
68     out->verbose    = 2;
69     out->boundary    = 1;
70     out->x           = 0;
71     out->y           = 0;
72     out->z          = 0;
73     out->Y          = 0;
```

```
74      out->Z          = 0;
75      out->Box0        = 0;
76      out->Box1        = 0;
77      out->Box2        = 0;
78      out->Box3        = 0;
79      out->Box4        = 0;
80      out->Box5        = 0;
81      out->round      = 0.0;
82
83      /*
84       * Fetch one line at the time with a buffer of 1024 until we
85       * encounter a line with only "{Begin Data}" on it.
86       */
87      while(fgets(line, 1024, confFile) && strcmp(line, "{Begin Data}\n"))
88      {
89        /*
90         * lines starting with '#' are ignored, others are split
91         * at the equals sign.
92         */
93        if(*line != '#' && splitEqual(line, &par, &data))
94        {
95          /*
96           * We then compare the part before the equals sign
97           * i.e. the parameter to the values we are looking
98           * for and parse the value and set the corresponding
99           * parameter.
100          */
101         if(!strcmp(par, "dim"))
102           sscanf(data, "%d", &out->dim);
103         if(!strcmp(par, "Xsize"))
104           sscanf(data, "%d", &out->Xsize);
105         if(!strcmp(par, "Ysize"))
106           sscanf(data, "%d", &out->Ysize);
107         if(!strcmp(par, "Zsize"))
108           sscanf(data, "%d", &out->Zsize);
109         if(!strcmp(par, "steps"))
110           sscanf(data, "%u", &out->steps);
111         if(!strcmp(par, "T1"))
112           sscanf(data, "%u", &out->T1);
113         if(!strcmp(par, "verbose"))
114           sscanf(data, "%d", &out->verbose);
115         if(!strcmp(par, "Box0"))
116           sscanf(data, "%d", &out->Box0);
117         if(!strcmp(par, "Box1"))
118           sscanf(data, "%d", &out->Box1);
119         if(!strcmp(par, "Box2"))
120           sscanf(data, "%d", &out->Box2);
121         if(!strcmp(par, "Box3"))
122           sscanf(data, "%d", &out->Box3);
123         if(!strcmp(par, "Box4"))
124           sscanf(data, "%d", &out->Box4);
125         if(!strcmp(par, "Box5"))
126           sscanf(data, "%d", &out->Box5);
127         if(!strcmp(par, "round"))
128           sscanf(data, "%lf", &out->round);
129         if(!strcmp(par, "model"))
130         {
131           if( !strcmp(data, "ll") || \
```

```
132               !strcmp(data, "LL"))
133             out->model = 0;
134           if( !strcmp(data, "lr") || \
135               !strcmp(data, "LR"))
136             out->model = 1;
137         }
138       }
139     }
140
141     // Initialise Shift Constants
142     out->x = 2 * out->dim;
143     if(out->dim>1)
144     {
145       out->y = out->Xsize * (2 * out->dim);
146       out->Y = out->Xsize;
147     }
148     if(out->dim>2)
149     {
150       out->z = out->Xsize * out->Ysize * (2 * out->dim);
151       out->Z = out->Xsize * out->Ysize;
152     }
153
154     // Allocate initPot, R and Z
155     if(out->dim==3)
156     {
157       initPot = (double*) malloc(
158           sizeof(double) * (out->Xsize * out->Ysize * out->Zsize)+10
159           );
160       R_orig  = (double*) malloc(
161           sizeof(double) * (out->Xsize * out->Ysize * out->Zsize)+10
162           );
163       Z_orig  = (double*) malloc(
164           sizeof(double) * (out->Xsize * out->Ysize * out->Zsize)+10
165           );
166       for(i=0;i<(out->Xsize * out->Ysize * out->Zsize)+10;i++)
167       {
168         *(initPot+i) = 0.0;
169         *(R_orig+i)  = 0.0;
170         *(Z_orig+i)  = 0.0;
171       }
172     }
173     else if(out->dim==2)
174     {
175       initPot = (double*) malloc(
176           sizeof(double) * (out->Xsize * out->Ysize)+10
177           );
178       R_orig  = (double*) malloc(
179           sizeof(double) * (out->Xsize * out->Ysize)+10
180           );
181       Z_orig  = (double*) malloc(
182           sizeof(double) * (out->Xsize * out->Ysize)+10
183           );
184       for(i=0;i<(out->Xsize * out->Ysize)+10;i++)
185       {
186         *(initPot+i) = 0.0;
187         *(R_orig+i)  = 0.0;
188         *(Z_orig+i)  = 0.0;
189       }
```

```
190     }
191     else // out->dim==1
192     {
193       initPot = (double*) malloc(sizeof(double) * (out->Xsize)+10);
194       R_orig  = (double*) malloc(sizeof(double) * (out->Xsize)+10);
195       Z_orig  = (double*) malloc(sizeof(double) * (out->Xsize)+10);
196       for(i=0;i<out->Xsize+10;i++)
197       {
198         *(initPot+i) = 0.0;
199         *(R_orig+i)  = 0.0;
200         *(Z_orig+i)  = 0.0;
201       }
202     }
203
204     iP = initPot;
205     R = R_orig;
206     Z = Z_orig;
207
208
209     /*
210      * Here the values for R, Z and the potential are parsed
211      * for each node.
212      */
213     while(fgets(line, 1024, confFile))
214     {
215       if(*line != '#' && splitEqual(line, &par, &data))
216       {
217         if(!strcmp(par, "P"))
218           sscanf(data, "%le", iP++);
219         if(!strcmp(par, "R"))
220           sscanf(data, "%le", R++);
221         if(!strcmp(par, "Z"))
222           sscanf(data, "%le", Z++);
223       }
224     }
225
226     out->initPot = initPot;
227     out->pR = R_orig;
228     out->pZ = Z_orig;
229   }
230
231   // vim:set ts=2 sw=2:
```

G.2.13 worker.h

```
1   /*
2    *  File: worker.h
3    *
4    *  Author: Frederik Klama
5    *  Copyright 2010 Frederik Klama
6    *
7    *  This file is part of TLM-Simulator.
8    *
9    *  TLM-Simulator is free software: you can redistribute it and/or modify
10   *  it under the terms of the GNU General Public License as published by
11   *  the Free Software Foundation, either version 3 of the License, or
12   *  (at your option) any later version.
13   *
14   *  TLM-Simulator is distributed in the hope that it will be useful,
```

```
15   *  but WITHOUT ANY WARRANTY; without even the implied warranty of
16   *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
17   *  GNU General Public License for more details.
18   *
19   *  You should have received a copy of the GNU General Public License
20   *  along with TLM-Simulator.  If not, see <http://www.gnu.org/licenses/>.
21   *
22   */
23
24  void
25  LLconnect(
26      struct confStruct,
27      double*,
28      double*
29  );
30
31  void
32  LLscatter(
33      struct confStruct,
34      double*,
35      double*
36  );
37
38  void
39  LRscatter(
40      struct confStruct,
41      double*,
42      double*
43  );
44
45  void
46  LRconnect(
47      struct confStruct,
48      double*,
49      double*
50  );
51
52  void
53  T1decay(
54      struct confStruct,
55      double*,
56      double
57  );
58
59  void
60  calcSums(
61      struct confStruct,
62      long long,
63      double*,
64      double*,
65      double*,
66      double*,
67      double*
68  );
69
70  /*
71   *    4   2
72   *     \  ^
```

```
73   *      \ |
74   *       \|
75   * 0 <----+----> 1
76   *        |\
77   *        | \
78   *        V  \
79   *        3   5
80   */
81
82
83  // vim:set ts=2 sw=2:
```

G.2.14 worker.c

```
1   /*
2    *  File: worker.c
3    *
4    *  Author: Frederik Klama
5    *  Copyright 2010 Frederik Klama
6    *
7    *  This file is part of TLM-Simulator.
8    *
9    *  TLM-Simulator is free software: you can redistribute it and/or modify
10   *  it under the terms of the GNU General Public License as published by
11   *  the Free Software Foundation, either version 3 of the License, or
12   *  (at your option) any later version.
13   *
14   *  TLM-Simulator is distributed in the hope that it will be useful,
15   *  but WITHOUT ANY WARRANTY; without even the implied warranty of
16   *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
17   *  GNU General Public License for more details.
18   *
19   *  You should have received a copy of the GNU General Public License
20   *  along with TLM-Simulator.  If not, see <http://www.gnu.org/licenses/>.
21   *
22   */
23  #include <omp.h>
24  #include <stdio.h>
25  #include <math.h>
26
27  #include "common.h"
28  #include "worker.h"
29
30  #define FLUSH fflush(stdout)
31
32  /*    4   2
33   *     \  ^
34   *      \ |
35   *       \|
36   * 0 <----+----> 1
37   *        |\
38   *        | \
39   *        V  \
40   *        3   5
41   */
42
43
44  /*********************************************************
45   *                   Link Line Model                     *
```

```
46   ********************************************************
47   * Connecting in 1D, 2D and 3D.                         *
48   * Most of the code is for the corners, edges and       *
49   * areas                                                *
50   ********************************************************/
51
52  void
53  LLconnect(
54      struct confStruct c,
55      double* Vi,
56      double* Vs
57      )
58  {
59    long x = c.x;
60    long y = c.y;
61    long z = c.z;
62    long long off = 0;
63    register double *pVi;
64    double *pVs;
65
66    //// Bulk Connect
67    switch(c.dim)
68    {
69    case 1:;
70      {
71        register long i;
72        pVi = Vi+2;
73        pVs = Vs+2;
74        for(i=1; i<(c.Xsize-1); i++)
75        {
76          *(pVi)   = *(pVs-x+1);
77          *(++pVi) = *(pVs+x);
78          ++pVi;
79          pVs += 2;
80        }
81      }
82      break;
83
84    case 2:;
85      {
86        register long i;
87        pVi = Vi+y;
88        pVs = Vs+y;
89        for(i=c.Xsize; i<(c.Xsize*(c.Ysize-1)); i++)
90        {
91          *(pVi)   = *(pVs-x+1);
92          *(++pVi) = *(pVs+x);
93          *(++pVi) = *(pVs-y+3);
94          *(++pVi) = *(pVs+y+2);
95          ++pVi;
96          pVs += 4;
97        }
98      }
99      break;
100
101   case 3:;
102     {
103       long long i;
```

```
104         long long ii;
105         pVi = Vi+z;
106         pVs = Vs+z;
107         #pragma omp parallel private(i,ii)
108         {
109           #pragma omp for
110           for(
111               i=(c.Xsize*c.Ysize);
112               i<(c.Xsize*c.Ysize*c.Zsize-c.Xsize*c.Ysize);
113               i++)
114           {
115             ii = i*6;
116             *(pVi + ii + 0) = *(pVs + ii -x+1);
117             *(pVi + ii + 1) = *(pVs + ii +x);
118             *(pVi + ii + 1) = *(pVs + ii -y+3);
119             *(pVi + ii + 1) = *(pVs + ii +y+2);
120             *(pVi + ii + 1) = *(pVs + ii -z+5);
121             *(pVi + ii + 1) = *(pVs + ii +z+4);
122           }
123         }
124       }
125       break;
126     } // switch
127
128     //// Reflections at the boundaries
129     switch(c.dim)
130     {
131     case 3:;
132       //// Back Frame (z=0)
133       // Top left corner (x=0, y=0)
134       pVi = Vi+4;
135       pVs = Vs+4;
136       *(pVi)   = *(pVs);    // 4 : 4
137       *(++pVi) = *(pVs+z);  // 5 : 4
138
139       // Top right corner (x=max, y=0)
140       off = (c.Xsize-1) * x;
141       pVi += off;
142       pVs += off;
143       *(pVi)   = *(pVs+z);  // 5 : 4
144       *(--pVi)   = *(pVs);  // 4 : 4
145
146       // Bottom left corner (x=0, y=max)
147       off = (c.Ysize-1) * y;
148       pVi = Vi+off+4;
149       pVs = Vs+off+4;
150       *(pVi)   = *(pVs);    // 4 : 4
151       *(++pVi) = *(pVs+z);  // 5 : 4
152
153       // Bottom right corner (x=max, y=max)
154       off = (c.Xsize-1) * x;
155       pVi += off;
156       pVs += off;
157       *(pVi)   = *(pVs+z);  // 5 : 4
158       *(--pVi) = *(pVs);    // 4 : 4
159
160       {
161         register long i;
```

```
162         for(i=1; i<(c.Xsize-1); i++)
163         {
164           // Top border (x=i, y=0)
165           off = i * x;
166           pVi = Vi+off+4;
167           pVs = Vs+off+4;
168           *(pVi)   = *(pVs);    // 4 : 4
169           *(++pVi) = *(pVs+z);  // 5 : 4
170
171           // Bottom border (x=i, y=max)
172           off = (c.Ysize-1) * y;
173           pVi += off;
174           pVs += off;
175           *(pVi)   = *(pVs+z); // 5 : 4
176           *(--pVi) = *(pVs);   // 4 : 4
177         }
178
179         for(i=1; i<(c.Ysize-1); i++)
180         {
181           // Left border (x=0, y=i)
182           off = i * y;
183           pVi = Vi+off+4;
184           pVs = Vs+off+4;
185           *(pVi)   = *(pVs);   // 4 : 4
186           *(++pVi) = *(pVs+z); // 5 : 4
187
188           // Right border (x=max, y=i)
189           off = (c.Xsize-1) * x;
190           pVi += off;
191           pVs += off;
192           *(pVi)   = *(pVs+z); // 5 : 4
193           *(--pVi) = *(pVs);   // 4 : 4
194         }
195       }
196
197       //// Front Frame  (z=max)
198       // Top left corner (x=0, y=0)
199       off = (c.Zsize-1) * z;
200       pVi = Vi+off;
201       pVs = Vs+off;
202       *(pVi)   = *(pVs);              // 0 : 0
203       *(++pVi) = *(pVs+x); pVs += 2;  // 1 : 0
204       *(++pVi) = *(pVs);              // 2 : 2
205       *(++pVi) = *(pVs+y); pVs += 3;  // 3 : 2
206       *(++pVi) = *(pVs-z);            // 4 : 5
207       *(++pVi) = *(pVs);              // 5 : 5
208
209       // Top right corner (x=max, y=0)
210       off = (c.Xsize-1) * x;
211       pVi += off;
212       pVs += off;
213       *(pVi)   = *(pVs);               // 5 : 5
214       *(--pVi) = *(pVs-z); pVs -= 3;  // 4 : 5
215       *(--pVi) = *(pVs+y);            // 3 : 2
216       *(--pVi) = *(pVs);   pVs -= 1;  // 2 : 2
217       *(--pVi) = *(pVs);               // 1 : 1
218       *(--pVi) = *(pVs-x);            // 0 : 1
219
```

```
220       // Bottom left corner (x=0, y=max)
221       off = (c.Ysize-1) * y + (c.Zsize-1) * z;
222       pVi = Vi+off;
223       pVs = Vs+off;
224       *(pVi)   = *(pVs);              // 0 : 0
225       *(++pVi) = *(pVs+x); pVs += 3;  // 1 : 0
226       *(++pVi) = *(pVs-y);             // 2 : 3
227       *(++pVi) = *(pVs);   pVs += 2;  // 3 : 3
228       *(++pVi) = *(pVs-z);             // 4 : 5
229       *(++pVi) = *(pVs);              // 5 : 5
230
231       // Bottom right corner (x=max, y=max)
232       off = (c.Xsize-1) * x;
233       pVi += off;
234       pVs += off;
235       *(pVi)   = *(pVs);               // 5 : 5
236       *(--pVi) = *(pVs-z); pVs -= 2;  // 4 : 5
237       *(--pVi) = *(pVs);               // 3 : 3
238       *(--pVi) = *(pVs-y); pVs -= 2;  // 2 : 3
239       *(--pVi) = *(pVs);               // 1 : 1
240       *(--pVi) = *(pVs-x);             // 0 : 1
241
242       {
243         register long i;
244         for(i=1; i<(c.Xsize-1); i++)
245         {
246           // Top front border (x=i, y=0, z=max)
247           off = i * x + (c.Zsize-1) * z;
248           pVi = Vi+off;
249           pVs = Vs+off;
250           *(pVi)   = *(pVs-x+1);          // 0 : 1
251           *(++pVi) = *(pVs+x); pVs += 2;  // 1 : 0
252           *(++pVi) = *(pVs);               // 2 : 2
253           *(++pVi) = *(pVs+y); pVs += 3;  // 3 : 2
254           *(++pVi) = *(pVs-z);             // 4 : 5
255           *(++pVi) = *(pVs);              // 5 : 5
256
257           // Bottom front border (x=i, y=0, z=max)
258           off = (c.Ysize-1) * y;
259           pVi += off;
260           pVs += off;
261           *(pVi)   = *(pVs);              // 5 : 5
262           *(--pVi) = *(pVs-z); pVs -= 2;  // 4 : 5
263           *(--pVi) = *(pVs);              // 3 : 3
264           *(--pVi) = *(pVs-y); pVs -= 3;  // 2 : 3
265           *(--pVi) = *(pVs+x);            // 1 : 0
266           *(--pVi) = *(pVs-x+1);          // 0 : 1
267         }
268
269         for(i=1; i<(c.Ysize-1); i++)
270         {
271           // Left front border (x=0, y=i, z=max)
272           off = i * y + (c.Zsize-1) * z;
273           pVi = Vi+off;
274           pVs = Vs+off;
275           *(pVi)   = *(pVs);              // 0 : 0
276           *(++pVi) = *(pVs+x); pVs += 2;  // 1 : 0
277           *(++pVi) = *(pVs-y+1);           // 2 : 3
```

```
278           *(++pVi) = *(pVs+y); pVs += 3;  // 3 : 2
279           *(++pVi) = *(pVs-z);             // 4 : 5
280           *(++pVi) = *(pVs);               // 5 : 5
281
282           // Right front border (x=max, y=i, z=max)
283           off = (c.Xsize-1) * x;
284           pVi += off;
285           pVs += off;
286           *(pVi)   = *(pVs);              // 5 : 5
287           *(--pVi) = *(pVs-z);   pVs -=3; // 4 : 5
288           *(--pVi) = *(pVs+y);            // 3 : 2
289           *(--pVi) = *(pVs-y+1); --pVs;    // 2 : 3
290           *(--pVi) = *(pVs);               // 1 : 1
291           *(--pVi) = *(pVs-x);             // 0 : 1
292         }
293
294         for(i=1; i<(c.Zsize-1); i++)
295         {
296           // Top left border (x=0, y=0, z=i)
297           off = i * z;
298           pVi = Vi+off;
299           pVs = Vs+off;
300           *(pVi)   = *(pVs);              // 0 : 0
301           *(++pVi) = *(pVs+x);  pVs += 2;  // 1 : 0
302           *(++pVi) = *(pVs);               // 2 : 2
303           *(++pVi) = *(pVs+y);  pVs += 2; // 3 : 2
304           *(++pVi) = *(pVs-z+1);           // 4 : 5
305           *(++pVi) = *(pVs+z);             // 5 : 4
306
307           // Bottom left border (x=0, y=max, z=i)
308           off = (c.Ysize-1) * y;
309           pVi += off;
310           pVs += off;
311           *(pVi)   = *(pVs+z);            // 5 : 4
312           *(--pVi) = *(pVs-z+1); --pVs;    // 4 : 5
313           *(--pVi) = *(pVs);               // 3 : 3
314           *(--pVi) = *(pVs-y);   pVs-=3;  // 2 : 3
315           *(--pVi) = *(pVs+x);             // 1 : 0
316           *(--pVi) = *(pVs);              // 0 : 0
317
318           // Top right border (x=max, y=0, z=i)
319           off = (c.Xsize-1) * x + i * z;
320           pVi = Vi+off;
321           pVs = Vs+off+1;
322           *(pVi)   = *(pVs-x);             // 0 : 1
323           *(++pVi) = *(pVs);    ++pVs;    // 1 : 1
324           *(++pVi) = *(pVs);               // 2 : 2
325           *(++pVi) = *(pVs+y);  pVs += 2; // 3 : 2
326           *(++pVi) = *(pVs-z+1);           // 4 : 5
327           *(++pVi) = *(pVs+z);             // 5 : 4
328
329           // Bottom right border (x=max, y=max, z=i)
330           off = (c.Ysize-1) * y;
331           pVi += off;
332           pVs += off;
333           *(pVi)   = *(pVs+z);             // 5 : 4
334           *(--pVi) = *(pVs-z+1); --pVs;    // 4 : 5
335           *(--pVi) = *(pVs);               // 3 : 3
```

```
336           *(--pVi) = *(pVs-y);   pVs-=2;  // 2 : 3
337           *(--pVi) = *(pVs);              // 1 : 1
338           *(--pVi) = *(pVs-x);            // 0 : 1
339         }
340       }
341
342       //// Areas
343       {
344         register long i;
345         register long j;
346         for(i=1; i<(c.Xsize-1); i++)
347           for(j=1; j<(c.Ysize-1); j++)
348           {
349             // Back area (x=i, y=j, z=0)
350             off = x*i + y*j;
351             pVi = Vi+off;
352             pVs = Vs+off;
353             *(pVi)   = *(pVs-x+1);        // 0 : 1
354             *(++pVi) = *(pVs+x);  pVs+=2;  // 1 : 0
355             *(++pVi) = *(pVs-y+1);        // 2 : 3
356             *(++pVi) = *(pVs+y);  pVs+=2;  // 3 : 2
357             *(++pVi) = *(pVs);            // 4 : 4
358             *(++pVi) = *(pVs+z);          // 5 : 4
359
360             // Front area (x=i, y=j, z=max)
361             off = z*(c.Zsize-1);
362             pVi += off;
363             pVs += off+1;
364             *(pVi)   = *(pVs);            // 5 : 5
365             *(--pVi) = *(pVs-z);  pVs-=3;  // 4 : 5
366             *(--pVi) = *(pVs+y);          // 3 : 2
367             *(--pVi) = *(pVs-y+1);pVs-=2;  // 2 : 3
368             *(--pVi) = *(pVs+x);          // 1 : 0
369             *(--pVi) = *(pVs-x+1);        // 0 : 1
370           }
371
372         for(i=1; i<(c.Xsize-1); i++)
373           for(j=1; j<(c.Zsize-1); j++)
374           {
375             // Top area (x=i, y=0, z=j)
376             off = x*i + z*j;
377             pVi = Vi+off;
378             pVs = Vs+off+1;
379             *(pVi)   = *(pVs-x);          // 0 : 1
380             *(++pVi) = *(pVs+x-1); pVs+=2;// 1 : 0
381             *(++pVi) = *(pVs);            // 2 : 3
382             *(++pVi) = *(pVs+y);   ++pVs;  // 3 : 3
383             *(++pVi) = *(pVs-z+1);        // 4 : 5
384             *(++pVi) = *(pVs+z);          // 5 : 4
385
386             // Bottom area (x=i, y=max, z=j)
387             off = y * (c.Ysize-1);
388             pVi += off;
389             pVs += off;
390             *(pVi)   = *(pVs+z);          // 5 : 4
391             *(--pVi) = *(pVs-z+1); pVs-=2;// 4 : 5
392             *(--pVi) = *(pVs);            // 3 : 2
393             *(--pVi) = *(pVs-y);   pVs-=2;// 2 : 2
```

```
394             *(--pVi) = *(pVs+x);          // 1 : 0
395             *(--pVi) = *(pVs-x+1);        // 0 : 1
396           }
397
398         for(i=1; i<(c.Ysize-1); i++)
399           for(j=1; j<(c.Zsize-1); j++)
400           {
401             // Left area (x=0, y=i, z=j)
402             off = y*i+z*j;
403             pVi = Vi+off;
404             pVs = Vs+off;
405             *(pVi)   = *(pVs);            // 0 : 0
406             *(++pVi) = *(pVs+x);  pVs+=3;  // 1 : 0
407             *(++pVi) = *(pVs-y);          // 2 : 3
408             *(++pVi) = *(pVs+y-1); ++pVs;  // 3 : 2
409             *(++pVi) = *(pVs-z+1);        // 4 : 5
410             *(++pVi) = *(pVs+z-1);        // 5 : 4
411
412             // Right area (x=max, y=i, z=j)
413             off = x*(c.Xsize-1);
414             pVi += off;
415             pVs += off;
416             *(pVi)   = *(pVs+z);          // 5 : 4
417             *(--pVi) = *(pVs-z+1); --pVs;  // 4 : 5
418             *(--pVi) = *(pVs+y-1);        // 3 : 2
419             *(--pVi) = *(pVs-y);  pVs-=2;  // 2 : 3
420             *(--pVi) = *(pVs);            // 1 : 1
421             *(--pVi) = *(pVs-x);          // 0 : 1
422           }
423
424       }
425
426     case 2:;
427       // Top left corner (x=0, y=0, z=0)
428       pVi = Vi+2;
429       pVs = Vs+2;
430       *(pVi)   = *(pVs);    // 2 : 2
431       *(++pVi) = *(pVs+y);  // 3 : 2
432
433       // Top right corner (x=max, y=0, z=0)
434       off = (c.Xsize-1) * x;
435       pVi += off;
436       pVs += off;
437       *(pVi)   = *(pVs+y);  // 3 : 2
438       *(--pVi) = *(pVs);    // 2 : 2
439
440       // Bottom left corner (x=0, y=max, z=0)
441       off = (c.Ysize-1) * y;
442       pVi = Vi+off;
443       pVs = Vs+off;
444       *(pVi)   = *(pVs);              // 0 : 0
445       *(++pVi) = *(pVs+x); pVs += 3;  // 1 : 0
446       *(++pVi) = *(pVs-y);             // 2 : 3
447       *(++pVi) = *(pVs);              // 3 : 3
448
449       // Bottom right corner (x=max, y=max, z=0)
450       off = (c.Xsize-1) * x;
451       pVi += off;
```

```
452       pVs += off;
453       *(pVi)   = *(pVs);              // 3 : 3
454       *(--pVi) = *(pVs-y); pVs -= 2;  // 2 : 3
455       *(--pVi) = *(pVs);               // 1 : 1
456       *(--pVi) = *(pVs-x);             // 0 : 1
457
458       {
459         register long i;
460         for(i=1; i<(c.Xsize-1); i++)
461         {
462           // Top border (x=i, y=0, z=0)
463           long off = i * x;
464           pVi = Vi+off;
465           pVs = Vs+off;
466           *(pVi)   = *(pVs-x+1);          // 0 : 1
467           *(++pVi) = *(pVs+x); pVs += 2;  // 1 : 0
468           *(++pVi) = *(pVs);              // 2 : 2
469           *(++pVi) = *(pVs+y);            // 3 : 2
470
471           // Bottom border (x=i, y=max, z=0)
472           off = (c.Ysize-1) * y;
473           pVi += off;
474           pVs += off+1;
475           *(pVi)   = *(pVs);              // 3 : 3
476           *(--pVi) = *(pVs-y); pVs -= 3;  // 2 : 3
477           *(--pVi) = *(pVs+x);            // 1 : 0
478           *(--pVi) = *(pVs-x+1);          // 0 : 1
479         }
480
481         for(i=1; i<(c.Ysize-1); i++)
482         {
483           // Left border (x=0, y=i, z=0)
484           off = i * y;
485           pVi = Vi+off;
486           pVs = Vs+off;
487           *(pVi)   = *(pVs);            // 0 : 0
488           *(++pVi) = *(pVs+x); pVs+=2;  // 1 : 0
489           *(++pVi) = *(pVs-y+1);        // 2 : 3
490           *(++pVi) = *(pVs+y);          // 3 : 2
491
492           // Right border (x=max, y=i, z=0)
493           off = (c.Xsize-1) * x;
494           pVi += off;
495           pVs += off;
496           *(pVi)   = *(pVs+y);          // 3 : 2
497           *(--pVi) = *(pVs-y+1);        // 2 : 3
498           *(--pVi) = *(pVs);     --pVs; // 1 : 1
499           *(--pVi) = *(pVs-x);          // 0 : 1
500         }
501       }
502
503     case 1:;
504       // Left boundary reflection (x=0, y=0, z=0)
505       pVi = Vi;
506       pVs = Vs;
507       *pVi     = *pVs;      // 0 : 0
508       *(++pVi) = *(pVs+x);  // 1 : 0
509
```

```
510       // Right boundary reflection (x=max, y=0, z=0)
511       off = (c.Xsize-1) * x;
512       pVi += off-1;
513       pVs += off;
514       *(pVi)   = *(pVs-x+1);    // 1 : 1
515       *(++pVi) = *(pVs);  // 0 : 1
516     } // switch
517
518   }
519
520
521   /*******************************************************
522    *                   Link Line Model                   *
523    *******************************************************
524    * Scattering in 1D, 2D and 3D.                        *
525    *******************************************************/
526
527   void
528   LLscatter(
529       struct confStruct c,
530       double* Vi,
531       double* Vs
532       )
533   {
534     double refl;
535     double trans;
536
537     long x = c.x;
538     long y = c.y;
539     long z = c.z;
540
541     double* R = c.pR;
542     double* Z = c.pZ;
543
544     switch(c.dim)
545     {
546     case 1:;
547       {
548         register long  i;
549         register double *pVs;
550         double *pVi;
551         double *pR;
552         double *pZ;
553
554         pVs = Vs;
555         pVi = Vi;
556         pR  = R;
557         pZ  = Z;
558
559         for(i=0; i<c.Xsize; i++)
560         {
561           refl  = *pR/(*pR + *pZ);
562           trans = 1-refl;
563
564           *(pVs)   = *(pVi)   * refl +\
565                      *(pVi+1) * trans;
566           *(++pVs) = *(pVi)   * trans +\
567                      *(pVi+1) * refl;
```

```
568           ++pVs;
569           ++pR; ++pZ;
570           pVi += 2;
571         }
572       }
573       break;
574
575     case 2:;
576       {
577         long i;
578         long j;
579         double *pVs;
580         double *pVi;
581         double *pR;
582         double *pZ;
583
584         pVi = Vi;
585         pVs = Vs;
586         pR  = R;
587         pZ  = Z;
588
589         for(i=0; i<c.Ysize; i++)
590           for(j=0; j<c.Xsize; j++)
591           {
592             refl  = (*pR - (*pZ/2))/(*pR + *pZ);
593             trans  = (1-refl)/3;
594             ++pR; ++pZ;
595
596             *(pVs)   = *(pVi)   * refl  +\
597                        *(pVi+1) * trans +\
598                        *(pVi+2) * trans +\
599                        *(pVi+3) * trans;
600             *(++pVs) = *(pVi)   * trans +\
601                        *(pVi+1) * refl  +\
602                        *(pVi+2) * trans +\
603                        *(pVi+3) * trans;
604             *(++pVs) = *(pVi)   * trans +\
605                        *(pVi+1) * trans +\
606                        *(pVi+2) * refl  +\
607                        *(pVi+3) * trans;
608             *(++pVs) = *(pVi)   * trans +\
609                        *(pVi+1) * trans +\
610                        *(pVi+2) * trans +\
611                        *(pVi+3) * refl;
612             ++pVs;
613             pVi+=4;
614           }
615       }
616       break;
617
618     case 3:;
619       {
620         long i;
621         long j;
622         long k;
623         double *pVs;
624         double *pVi;
625         double *pR;
```

```
626         double *pZ;
627
628         pVi = Vi;
629         pVs = Vs;
630         pR  = R;
631         pZ  = Z;
632
633         for(i=0; i<c.Zsize; i++)
634           for(j=0; j<c.Ysize; j++)
635             for(k=0; k<c.Xsize; k++)
636             {
637               refl  = (*pR - (2/3) * *pZ)/(*pR + *pZ);
638               trans = (1 - refl)/5;
639               ++pR; ++pZ;
640
641               *(pVs)   = *(pVi)   * refl  +\
642                          *(pVi+1) * trans +\
643                          *(pVi+2) * trans +\
644                          *(pVi+3) * trans +\
645                          *(pVi+4) * trans +\
646                          *(pVi+5) * trans;
647               *(++pVs) = *(pVi)   * trans +\
648                          *(pVi+1) * refl  +\
649                          *(pVi+2) * trans +\
650                          *(pVi+3) * trans +\
651                          *(pVi+4) * trans +\
652                          *(pVi+5) * trans;
653               *(++pVs) = *(pVi)   * trans +\
654                          *(pVi+1) * trans +\
655                          *(pVi+2) * refl  +\
656                          *(pVi+3) * trans +\
657                          *(pVi+4) * trans +\
658                          *(pVi+5) * trans;
659               *(++pVs) = *(pVi)   * trans +\
660                          *(pVi+1) * trans +\
661                          *(pVi+2) * trans +\
662                          *(pVi+3) * refl  +\
663                          *(pVi+4) * trans +\
664                          *(pVi+5) * trans;
665               *(++pVs) = *(pVi)   * trans +\
666                          *(pVi+1) * trans +\
667                           *(pVi+2) * trans +\
668                          *(pVi+3) * trans +\
669                          *(pVi+4) * refl  +\
670                          *(pVi+5) * trans;
671               *(++pVs) = *(pVi)   * trans +\
672                          *(pVi+1) * trans +\
673                          *(pVi+2) * trans +\
674                          *(pVi+3) * trans +\
675                          *(pVi+4) * trans +\
676                          *(pVi+5) * refl;
677               ++pVs;
678               pVi += 6;
679             }
680       }
681       break;
682     } // switch
683   }
```

```
684
685
686  /*********************************************************
687   *                 Link Resistor Model                   *
688   *********************************************************
689   * Scattering in 1D, 2D and 3D.                          *
690   * Most of the code is for the corners, edges and        *
691   * areas                                                 *
692   *********************************************************/
693
694  void
695  LRscatter(
696      struct confStruct c,
697       double* Vi,
698       double* Vs
699      )
700  {
701    register long long  i;
702    register double      *pVs = Vs;
703    double *pVi = Vi;
704    switch(c.dim)
705    {
706      case 1:  for(i=0;i<c.Xsize;i++) {
707                 *pVs     = *(pVi+1);
708                 *(++pVs) = *pVi;
709                 ++pVs;
710                 pVi += 2;
711               }
712               break;
713      case 2: for(i=0;i<(c.Xsize*c.Zsize);i++) {
714                 double A;
715                 double B;
716
717                 A  = *(pVi)   + *(pVi+1);
718                 B = *(pVi+2) + *(pVi+3);
719
720                 *pVs     = (*(pVi+1) + B - *(pVi)  )/2;
721                 *(++pVs) = (*(pVi)   + B - *(pVi+1))/2;
722                 *(++pVs) = (A + *(pVi+3) - *(pVi+2))/2;
723                 *(++pVs) = (A + *(pVi+2) - *(pVi+3))/2;
724                 ++pVs;
725                 pVi += 4;
726               }
727               break;
728      case 3: for(i=0;i<(c.Xsize*c.Ysize*c.Zsize);i++) {
729                 double A;
730                 double B;
731                 double C;
732
733                 A = *(pVi) + *(pVi+1);
734                 B = *(pVi+2) + *(pVi+3);
735                 C = *(pVi+4) + *(pVi+5);
736
737                 {
738                   double tmp1;
739                   double tmp2;
740                   tmp1 = A;
741                   A = B + C;
```

```
742                   tmp2 = B;
743                   B = tmp1 + C;
744                   C = tmp1 + tmp2;
745                 }
746
747                 *pVs     = (*(pVi+1) + A - *(pVi)  )/2;
748                 *(++pVs) = (*(pVi)   + A - *(pVi+1))/2;
749                 *(++pVs) = (*(pVi+3) + B - *(pVi+2))/2;
750                 *(++pVs) = (*(pVi+2) + B - *(pVi+3))/2;
751                 *(++pVs) = (*(pVi+5) + C - *(pVi+4))/2;
752                 *(++pVs) = (*(pVi+4) + C - *(pVi+5))/2;
753                 ++pVs;
754                 pVi += 6;
755               }
756     } // switch
757   }
758
759
760   /*********************************************************
761    *                 Link Resistor Model                   *
762    *********************************************************
763    * Connecting in 1D, 2D and 3D.                          *
764    * Most of the code is for the corners, edges and        *
765    * areas                                                 *
766    *********************************************************/
767
768   void
769   LRconnect(
770       struct confStruct c,
771        double* Vi,
772       double* Vs
773       )
774   {
775     double* Rin = c.pR;
776     double* Zin = c.pZ;
777
778     long i, j, k;
779
780     long long offset;
781     long long Offset;
782
783     long x = c.x;
784     long y = c.y;
785     long z = c.z;
786     long Y = c.Y;
787     long Z = c.Z;
788
789     double refl;
790     double trans;
791     double *pVi;
792     double *pVs;
793     double *pR;
794     double *pZ;
795
796     //// Bulk connect
797     {
798       long long a;
799       long long A;
```

```
800       register double *pVi;
801       short Xmax = c.Xsize-1;
802       short Ymax = c.Ysize-1;
803       short Zmax = c.Zsize-1;
804
805       switch(c.dim)
806       {
807         case 1: pVi = Vi  + 2;
808                 pVs = Vs  + 2;
809                 pR   = Rin + 1;
810                 pZ  = Zin + 1;
811                 for(i=1;i<(c.Xsize-1);i++) {
812                   refl  = *pR/(*pR + *pZ);
813                   trans = 1-refl;
814                   ++pR; ++pZ;
815
816                   *pVi     = refl  * *pVs + \
817                              trans * *(++pVs-x);
818
819                   *(++pVi) = trans * *(pVs+x-1) + \
820                              refl  * *(pVs);
821                   ++pVi; ++pVs;
822                 }
823                 break;
824
825           /*    4   2
826            *     \  ^
827            *      \ |
828            *       \|
829            * 0 <----+----> 1
830            *        |\
831            *        | \
832            *        V  \
833            *        3   5
834            */
835
836         case 2:  for(i=1;i<(c.Ysize-1);i++)
837                 {
838                   a   = i*y;
839                   A    = i*Y;
840                   pVi = Vi  + a + x;
841                   pVs = Vs  + a + x;
842                   pR  = Rin  + A + 1;
843                   pZ  = Zin  + A + 1;
844
845                   for(j=1;j<(c.Xsize-1);j++)
846                   {
847                     refl  = *pR/(*pR + *pZ);
848                     trans  = 1-refl;
849
850                     *pVi     = refl  * *(pVs)   + trans * *(pVs-x+1);
851                     *(++pVi) = refl * *(pVs+1) + trans * *(pVs+x);
852                     *(++pVi) = refl * *(pVs+2) + trans * *(pVs-y+3);
853                     *(++pVi) = refl * *(pVs+3) + trans * *(pVs+y+2);
854                     ++pVi; pVs += 4;
855                   }
856                 }
857                 break;
```

```
858         case 3: for(i=1;i<(c.Zsize-1);i++)
859                   for(j=1;j<(c.Ysize-1);j++)
860                   {
861                     a   = i*z + j*y;
862                     A    = i*Z + j*Y;
863                     pVi = Vi  + a + x;
864                     pVs = Vs  + a + x;
865                     pR  = Rin  + A + 1;
866                     pZ  = Zin  + A + 1;
867
868                     for(k=1;k<(c.Xsize-1);k++)
869                     {
870                       refl  = *pR/(*pR + *pZ);
871                       trans  = 1-refl;
872
873                       *pVi     = refl * *(pVs)   + trans * *(pVs-x+1);
874                       *(++pVi) = refl * *(pVs+1) + trans * *(pVs+x);
875                       *(++pVi) = refl * *(pVs+2) + trans * *(pVs-y+3);
876                       *(++pVi) = refl * *(pVs+3) + trans * *(pVs+y+2);
877                       *(++pVi) = refl * *(pVs+4) + trans * *(pVs-z+5);
878                       *(++pVi) = refl * *(pVs+5) + trans * *(pVs+z+4);
879                       ++pVi; pVs += 6;
880                     }
881                   }
882       } // switch
883     }
884           /*    4   2
885            *     \  ^
886            *      \ |
887            *       \|
888            * 0 <----+----> 1
889            *        |\
890            *        | \
891            *        V  \
892            *        3   5
893            */
894
895     // Corners, Edges and Areas
896     {
897       long xmax;
898       long ymax;
899       long zmax;
900       long Xmax;
901       long Ymax;
902       long Zmax;
903
904       Xmax = c.Xsize-1;
905       Ymax = c.Ysize-1 * c.Xsize;
906       Zmax = c.Zsize-1 * c.Ysize * c.Xsize;
907       xmax = Xmax * c.dim * 2;
908       ymax = Ymax * c.dim * 2;
909       zmax = Zmax * c.dim * 2;
910
911       switch(c.dim)
912       {
913         case 3: ;
914           /*****************************
915            *         Corners-3D         *
```

```
916           ****************************/
917          // Top-Left-Back Corner
918          pVi = Vi;
919          pVs = Vs;
920          pR  = Rin;
921          pZ  = Zin;
922
923          refl  = *pR/(*pR + *pZ);
924          trans  = 1-refl;
925
926          *(pVi)   = *(pVs);
927          *(pVi+1) = refl * *(pVs+1) + trans * *(pVs+x);
928          *(pVi+2) = *(pVs+2);
929          *(pVi+3) = refl * *(pVs+3) + trans * *(pVs+y+2);
930          *(pVi+4) = *(pVs+4);
931          *(pVi+5) = refl * *(pVs+5) + trans * *(pVs+z+4);
932
933          // Top-Right-Back Corner
934          pVi = Vi  + xmax;
935          pVs = Vs  + xmax;
936          pR  = Rin + Xmax;
937          pZ  = Zin + Xmax;
938
939          refl  = *pR/(*pR + *pZ);
940          trans  = 1-refl;
941
942          *(pVi)   = refl * *(pVs)   + trans * *(pVs-x+1);
943          *(pVi+1) = *(pVs+1);
944          *(pVi+2) = *(pVs+2);
945          *(pVi+3) = refl * *(pVs+3) + trans * *(pVs+y+2);
946          *(pVi+4) = *(pVs+4);
947          *(pVi+5) = refl * *(pVs+5) + trans * *(pVs+z+4);
948
949          // Bottom-Left-Back Corner
950          pVi = Vi + ymax;
951          pVs = Vs + ymax;
952          pR  = Rin  + Ymax;
953          pZ  = Zin  + Ymax;
954
955          refl  = *pR/(*pR + *pZ);
956          trans  = 1-refl;
957
958          *(pVi)   = *(pVs);
959          *(pVi+1) = refl * *(pVs+1) + trans * *(pVs+x);
960          *(pVi+2) = refl * *(pVs+2) + trans * *(pVs-y+3);
961          *(pVi+3) = *(pVs+3);
962          *(pVi+4) = *(pVs+4);
963          *(pVi+5) = refl * *(pVs+5) + trans * *(pVs+z+4);
964
965          // Bottom-Right-Back Corner
966          pVi = Vi + xmax + ymax;
967          pVs = Vs + xmax + ymax;
968          pR  = Rin  + Xmax + Ymax;
969          pZ  = Zin  + Xmax + Ymax;
970
971          refl  = *pR/(*pR + *pZ);
972          trans  = 1-refl;
973
```

```
974         *(pVi)   = refl * *(pVs)   + trans * *(pVs-x+1);
975         *(pVi+1) = *(pVs+1);
976         *(pVi+2) = refl * *(pVs+2) + trans * *(pVs-y+3);
977         *(pVi+3) = *(pVs+3);
978         *(pVi+4) = *(pVs+4);
979         *(pVi+5) = refl * *(pVs+5) + trans * *(pVs+z+4);
980
981         // Top-Left-Front Corner
982         pVi = Vi + zmax;
983         pVs = Vs + zmax;
984         pR  = Rin  + Zmax;
985         pZ  = Zin  + Zmax;
986
987         refl  = *pR/(*pR + *pZ);
988         trans  = 1-refl;
989
990         *(pVi)   = *(pVs);
991         *(pVi+1) = refl * *(pVs+1) + trans * *(pVs+x);
992         *(pVi+2) = *(pVs+2);
993         *(pVi+3) = refl * *(pVs+3) + trans * *(pVs+y+2);
994         *(pVi+4) = refl * *(pVs+4) + trans * *(pVs-z+5);
995         *(pVi+5) = *(pVs+5);
996
997         // Top-Right-Front Corner
998         pVi = Vi + xmax + zmax;
999         pVs = Vs + xmax + zmax;
1000        pR  = Rin  + Xmax + Zmax;
1001        pZ  = Zin  + Xmax + Zmax;
1002
1003        refl  = *pR/(*pR + *pZ);
1004        trans  = 1-refl;
1005
1006        *(pVi)   = refl * *(pVs)   + trans * *(pVs-x+1);
1007        *(pVi+1) = *(pVs+1);
1008        *(pVi+2) = *(pVs+2);
1009        *(pVi+3) = refl * *(pVs+3) + trans * *(pVs+y+2);
1010        *(pVi+4) = refl * *(pVs+4) + trans * *(pVs-z+5);
1011        *(pVi+5) = *(pVs+5);
1012
1013        // Bottom-Left-Front Corner
1014        pVi = Vi + ymax + zmax;
1015        pVs = Vs + ymax + zmax;
1016        pR  = Rin  + Ymax + Zmax;
1017        pZ  = Zin  + Ymax + Zmax;
1018
1019        refl  = *pR/(*pR + *pZ);
1020        trans  = 1-refl;
1021
1022        *(pVi)   = *(pVs);
1023        *(pVi+1) = refl * *(pVs+1) + trans * *(pVs+x);
1024        *(pVi+2) = refl * *(pVs+2) + trans * *(pVs-y+3);
1025        *(pVi+3) = *(pVs+3);
1026        *(pVi+4) = refl * *(pVs+4) + trans * *(pVs-z+5);
1027        *(pVi+5) = *(pVs+5);
1028
1029        // Bottom-Right-Front Corner
1030        pVi = Vi + xmax + ymax + zmax;
1031        pVs = Vs + xmax + ymax + zmax;
```

```
1032          pR  = Rin  + Xmax + Ymax + Zmax;
1033          pZ  = Zin  + Xmax + Ymax + Zmax;
1034
1035          refl  = *pR/(*pR + *pZ);
1036          trans  = 1-refl;
1037
1038          *(pVi)   = refl * *(pVs)   + trans * *(pVs-x+1);
1039          *(pVi+1) = *(pVs+1);
1040          *(pVi+2) = refl * *(pVs+2) + trans * *(pVs-y+3);
1041          *(pVi+3) = *(pVs+3);
1042          *(pVi+4) = refl * *(pVs+4) + trans * *(pVs-z+5);
1043          *(pVi+5) = *(pVs+5);
1044
1045          /****************************
1046           *         Edges-3D         *
1047           ****************************/
1048          /*    4   2
1049           *     \  ^
1050           *      \ |
1051           *       \|
1052           * 0 <----+----> 1
1053           *        |\
1054           *        | \
1055           *        V  \
1056           *        3   5
1057           */
1058
1059          // Top-Back Edge
1060          for(i=1;i<(c.Xsize-1);i++)
1061          {
1062            offset  = i*x;
1063            Offset  = i;
1064            pVi      = Vi + offset;
1065            pVs      = Vs + offset;
1066            pR      = Rin  + Offset;
1067            pZ      = Zin  + Offset;
1068
1069            refl  = *pR/(*pR + *pZ);
1070            trans  = 1-refl;
1071
1072            *(pVi)   = refl * *(pVs)   + trans * *(pVs-x+1);
1073            *(pVi+1) = refl * *(pVs+1) + trans * *(pVs+x);
1074            *(pVi+2) = *(pVs+2);
1075            *(pVi+3) = refl * *(pVs+3) + trans * *(pVs+y+3);
1076            *(pVi+4) = *(pVs+4);
1077            *(pVi+5) = refl * *(pVs+5) + trans * *(pVs+z+4);
1078
1079          }
1080
1081          // Bottom-Back Edge
1082          for(i=1;i<(c.Xsize-1);i++)
1083          {
1084            offset  = ymax + i*x;
1085            Offset  = Ymax + i;
1086            pVi      = Vi + offset;
1087            pVs      = Vs + offset;
1088            pR      = Rin  + Offset;
1089            pZ      = Zin  + Offset;
```

```
1090
1091            refl  = *pR/(*pR + *pZ);
1092            trans  = 1-refl;
1093
1094            *(pVi)   = refl * *(pVs)   + trans * *(pVs-x+1);
1095            *(pVi+1) = refl * *(pVs+1) + trans * *(pVs+x);
1096            *(pVi+2) = refl * *(pVs+2) + trans * *(pVs-y+3);
1097            *(pVi+3) = *(pVs+3);
1098            *(pVi+4) = *(pVs+4);
1099            *(pVi+5) = refl * *(pVs+5) + trans * *(pVs+z+4);
1100          }
1101
1102          // Left-Back Edge
1103          for(i=1;i<(c.Ysize-1);i++)
1104          {
1105            offset  = i*y;
1106            Offset  = i*Y;
1107            pVi      = Vi + offset;
1108            pVs      = Vs + offset;
1109            pR      = Rin  + Offset;
1110            pZ      = Zin  + Offset;
1111
1112            refl  = *pR/(*pR + *pZ);
1113            trans  = 1-refl;
1114
1115            *(pVi)   = *(pVs);
1116            *(pVi+1) = refl * *(pVs+1) + trans * *(pVs+x);
1117            *(pVi+2) = refl * *(pVs+2) + trans * *(pVs-y+3);
1118            *(pVi+3) = refl * *(pVs+3) + trans * *(pVs+y+2);
1119            *(pVi+4) = *(pVs+4);
1120            *(pVi+5) = refl * *(pVs+5) + trans * *(pVs+z+4);
1121          }
1122
1123          // Right-Back Edge
1124          for(i=1;i<(c.Ysize-1);i++)
1125          {
1126            offset  = xmax + i*y;
1127            Offset  = Xmax + i*Y;
1128            pVi      = Vi + offset;
1129            pVs      = Vs + offset;
1130            pR      = Rin  + Offset;
1131            pZ      = Zin  + Offset;
1132
1133            refl  = *pR/(*pR + *pZ);
1134            trans  = 1-refl;
1135
1136            *(pVi)   = refl * *(pVs)   + trans * *(pVs-x+1);
1137            *(pVi+1) = *(pVs+1);
1138            *(pVi+2) = refl * *(pVs+2) + trans * *(pVs-y+3);
1139            *(pVi+3) = refl * *(pVs+3) + trans * *(pVs+y+2);
1140            *(pVi+4) = *(pVs+4);
1141            *(pVi+5) = refl * *(pVs+5) + trans * *(pVs+z+4);
1142          }
1143
1144          // Top-Front Edge
1145          for(i=1;i<(c.Xsize-1);i++)
1146          {
1147            offset  = i*x + zmax;
```

```
1148            Offset  = i   + Zmax;
1149            pVi      = Vi + offset;
1150            pVs      = Vs + offset;
1151            pR      = Rin  + Offset;
1152            pZ      = Zin  + Offset;
1153
1154            refl  = *pR/(*pR + *pZ);
1155            trans  = 1-refl;
1156
1157            *(pVi)   = refl * *(pVs)   + trans * *(pVs-x+1);
1158            *(pVi+1) = refl * *(pVs+1) + trans * *(pVs+x);
1159            *(pVi+2) = *(pVs+2);
1160            *(pVi+3) = refl * *(pVs+3) + trans * *(pVs+y+3);
1161            *(pVi+4) = refl * *(pVs+4) + trans * *(pVs-z+5);
1162            *(pVi+5) = *(pVs+5);
1163
1164          }
1165
1166          // Bottom-Front Edge
1167          for(i=1;i<(c.Xsize-1);i++)
1168          {
1169            offset  = ymax + i*x + zmax;
1170            Offset  = Ymax + i + Zmax;
1171            pVi      = Vi + offset;
1172            pVs      = Vs + offset;
1173            pR      = Rin  + Offset;
1174            pZ      = Zin  + Offset;
1175
1176            refl  = *pR/(*pR + *pZ);
1177            trans  = 1-refl;
1178
1179            *(pVi)   = refl * *(pVs)   + trans * *(pVs-x+1);
1180            *(pVi+1) = refl * *(pVs+1) + trans * *(pVs+x);
1181            *(pVi+2) = refl * *(pVs+2) + trans * *(pVs-y+3);
1182            *(pVi+3) = *(pVs+3);
1183            *(pVi+4) = refl * *(pVs+4) + trans * *(pVs-z+5);
1184            *(pVi+5) = *(pVs+5);
1185          }
1186
1187          // Left-Front Edge
1188          for(i=1;i<(c.Ysize-1);i++)
1189          {
1190            offset  = i*y + zmax;
1191            Offset  = i*Y + Zmax;
1192            pVi      = Vi + offset;
1193            pVs      = Vs + offset;
1194            pR      = Rin  + Offset;
1195            pZ      = Zin  + Offset;
1196
1197            refl  = *pR/(*pR + *pZ);
1198            trans  = 1-refl;
1199
1200            *(pVi)   = *(pVs);
1201            *(pVi+1) = refl * *(pVs+1) + trans * *(pVs+x);
1202            *(pVi+2) = refl * *(pVs+2) + trans * *(pVs-y+3);
1203            *(pVi+3) = refl * *(pVs+3) + trans * *(pVs+y+2);
1204            *(pVi+4) = refl * *(pVs+4) + trans * *(pVs-z+5);
1205            *(pVi+5) = *(pVs+5);
```

```
1206          }
1207
1208          // Right-Front Edge
1209          for(i=1;i<(c.Ysize-1);i++)
1210          {
1211            offset  = xmax + i*y + zmax;
1212            Offset  = Xmax + i*Y + Zmax;
1213            pVi      = Vi + offset;
1214            pVs      = Vs + offset;
1215            pR      = Rin  + Offset;
1216            pZ      = Zin  + Offset;
1217
1218            refl  = *pR/(*pR + *pZ);
1219            trans  = 1-refl;
1220
1221            *(pVi)   = refl * *(pVs)   + trans * *(pVs-x+1);
1222            *(pVi+1) = *(pVs+1);
1223            *(pVi+2) = refl * *(pVs+2) + trans * *(pVs-y+3);
1224            *(pVi+3) = refl * *(pVs+3) + trans * *(pVs+y+2);
1225            *(pVi+4) = refl * *(pVs+4) + trans * *(pVs-z+5);
1226            *(pVi+5) = *(pVs+5);
1227          }
1228
1229          /*    4   2
1230           *     \  ^
1231           *      \ |
1232           *       \|
1233           * 0 <----+----> 1
1234           *        |\
1235           *        | \
1236           *        V  \
1237           *        3   5
1238           */
1239
1240          // Left-Top Edge
1241          for(i=1;i<(c.Zsize-1);i++)
1242          {
1243            offset  = i*z;
1244            Offset  = i*Z;
1245            pVi      = Vi + offset;
1246            pVs      = Vs + offset;
1247            pR      = Rin  + Offset;
1248            pZ      = Zin  + Offset;
1249
1250            refl  = *pR/(*pR + *pZ);
1251            trans  = 1-refl;
1252
1253            *(pVi)   = *(pVs);
1254            *(pVi+1) = refl * *(pVs+1) + trans * *(pVs+x);
1255            *(pVi+2) = *(pVs+2);
1256            *(pVi+3) = refl * *(pVs+3) + trans * *(pVs+y+2);
1257            *(pVi+4) = refl * *(pVs+4) + trans * *(pVs-z+5);
1258            *(pVi+5) = refl * *(pVs+5) + trans * *(pVs+z+4);
1259          }
1260
1261          // Right-Top Edge
1262          for(i=1;i<(c.Zsize-1);i++)
1263          {
```

```
1264            offset  = i*z + xmax;
1265            Offset  = i*Z + Xmax;
1266            pVi      = Vi + offset;
1267            pVs      = Vs + offset;
1268            pR      = Rin  + Offset;
1269            pZ      = Zin  + Offset;
1270
1271            refl  = *pR/(*pR + *pZ);
1272            trans  = 1-refl;
1273
1274            *(pVi)   = refl * *(pVs)   + trans * *(pVs-x+1);
1275            *(pVi+1) = *(pVs+1);
1276            *(pVi+2) = *(pVs+2);
1277            *(pVi+3) = refl * *(pVs+3) + trans * *(pVs+y+2);
1278            *(pVi+4) = refl * *(pVs+4) + trans * *(pVs-z+5);
1279            *(pVi+5) = refl * *(pVs+5) + trans * *(pVs+z+4);
1280          }
1281
1282          // Left-Bottom Edge
1283          for(i=1;i<(c.Zsize-1);i++)
1284          {
1285            offset  = i*z + ymax;
1286            Offset  = i*Z + Ymax;
1287            pVi      = Vi + offset;
1288            pVs      = Vs + offset;
1289            pR      = Rin  + Offset;
1290            pZ      = Zin  + Offset;
1291
1292            refl  = *pR/(*pR + *pZ);
1293            trans  = 1-refl;
1294
1295            *(pVi)   = *(pVs);
1296            *(pVi+1) = refl * *(pVs+1) + trans * *(pVs+x);
1297            *(pVi+2) = refl * *(pVs+2) + trans * *(pVs-y+3);
1298            *(pVi+3) = *(pVs+3);
1299            *(pVi+4) = refl * *(pVs+4) + trans * *(pVs-z+5);
1300            *(pVi+5) = refl * *(pVs+5) + trans * *(pVs+z+4);
1301          }
1302
1303          // Right-Bottom Edge
1304          for(i=1;i<(c.Zsize-1);i++)
1305          {
1306            offset  = i*z + xmax + ymax;
1307            Offset  = i*Z + Xmax + Ymax;
1308            pVi      = Vi + offset;
1309            pVs      = Vs + offset;
1310            pR      = Rin  + Offset;
1311            pZ      = Zin  + Offset;
1312
1313            refl  = *pR/(*pR + *pZ);
1314            trans  = 1-refl;
1315
1316            *(pVi)   = refl * *(pVs)   + trans * *(pVs-x+1);
1317            *(pVi+1) = *(pVs+1);
1318            *(pVi+2) = refl * *(pVs+2) + trans * *(pVs-y+3);
1319            *(pVi+3) = *(pVs+3);
1320            *(pVi+4) = refl * *(pVs+4) + trans * *(pVs-z+5);
1321            *(pVi+5) = refl * *(pVs+5) + trans * *(pVs+z+4);
```

```
1322          }
1323
1324          /*****************************
1325           *          Areas-3D          *
1326           *****************************/
1327
1328          // Back Area
1329          for(i=1;i<(c.Xsize-1);i++)
1330            for(j=1;j<(c.Ysize-1);i++)
1331            {
1332              offset  = i*x + j*y;
1333              Offset  = i   + j*Y;
1334              pVi      = Vi + offset;
1335              pVs      = Vs + offset;
1336              pR      = Rin  + Offset;
1337              pZ      = Zin  + Offset;
1338
1339              refl  = *pR/(*pR + *pZ);
1340              trans = 1-refl;
1341
1342              *(pVi)   = refl * *(pVs)   + trans * *(pVs-x+1);
1343              *(pVi+1) = refl * *(pVs+1) + trans * *(pVs+x);
1344              *(pVi+2) = refl * *(pVs+2) + trans * *(pVs-y+3);
1345              *(pVi+3) = refl * *(pVs+3) + trans * *(pVs+y+2);
1346              *(pVi+4) = *(pVs+4);
1347              *(pVi+5) = refl * *(pVs+5) + trans * *(pVs+z+4);
1348            }
1349
1350          // Front Area
1351          for(i=1;i<(c.Xsize-1);i++)
1352            for(j=1;j<(c.Ysize-1);i++)
1353            {
1354              offset  = i*x + j*y + zmax;
1355              Offset  = i + j*Y + Zmax;
1356              pVi      = Vi + offset;
1357              pVs      = Vs + offset;
1358              pR      = Rin  + Offset;
1359              pZ      = Zin  + Offset;
1360
1361              refl  = *pR/(*pR + *pZ);
1362              trans = 1-refl;
1363
1364              *(pVi)   = refl * *(pVs)   + trans * *(pVs-x+1);
1365              *(pVi+1) = refl * *(pVs+1) + trans * *(pVs+x);
1366              *(pVi+2) = refl * *(pVs+2) + trans * *(pVs-y+3);
1367              *(pVi+3) = refl * *(pVs+3) + trans * *(pVs+y+2);
1368              *(pVi+4) = refl * *(pVs+4) + trans * *(pVs-z+5);
1369              *(pVi+5) = *(pVs+5);
1370            }
1371
1372          // Left Area
1373          for(i=1;i<(c.Zsize-1);i++)
1374            for(j=1;j<(c.Ysize-1);i++)
1375            {
1376              offset  = i*z + j*y;
1377              Offset  = i*Z + j*Y;
1378              pVi      = Vi + offset;
1379              pVs      = Vs + offset;
```

```
1380              pR      = Rin  + Offset;
1381              pZ      = Zin  + Offset;
1382
1383              refl  = *pR/(*pR + *pZ);
1384              trans = 1-refl;
1385
1386              *(pVi)   = *(pVs);
1387              *(pVi+1) = refl * *(pVs+1) + trans * *(pVs+x);
1388              *(pVi+2) = refl * *(pVs+2) + trans * *(pVs-y+3);
1389              *(pVi+3) = refl * *(pVs+3) + trans * *(pVs+y+2);
1390              *(pVi+4) = refl * *(pVs+4) + trans * *(pVs-z+5);
1391              *(pVi+5) = refl * *(pVs+5) + trans * *(pVs+z+4);
1392            }
1393
1394          // Right Area
1395          for(i=1;i<(c.Zsize-1);i++)
1396            for(j=1;j<(c.Ysize-1);i++)
1397            {
1398              offset  = i*z + j*y + xmax;
1399              Offset  = i*Z + j*Y + Xmax;
1400              pVi      = Vi + offset;
1401              pVs      = Vs + offset;
1402              pR      = Rin  + Offset;
1403              pZ      = Zin  + Offset;
1404
1405              refl  = *pR/(*pR + *pZ);
1406              trans = 1-refl;
1407
1408              *(pVi)   = refl * *(pVs)   + trans * *(pVs-x+1);
1409              *(pVi+1) = *(pVs+1);
1410              *(pVi+2) = refl * *(pVs+2) + trans * *(pVs-y+3);
1411              *(pVi+3) = refl * *(pVs+3) + trans * *(pVs+y+2);
1412              *(pVi+4) = refl * *(pVs+4) + trans * *(pVs-z+5);
1413              *(pVi+5) = refl * *(pVs+5) + trans * *(pVs+z+4);
1414            }
1415
1416          // Top Area
1417          for(i=1;i<(c.Xsize-1);i++)
1418            for(j=1;j<(c.Zsize-1);i++)
1419            {
1420              offset  = i*x + j*z;
1421              Offset  = i   + j*Z;
1422              pVi      = Vi + offset;
1423              pVs      = Vs + offset;
1424              pR      = Rin  + Offset;
1425              pZ      = Zin  + Offset;
1426
1427              refl  = *pR/(*pR + *pZ);
1428              trans = 1-refl;
1429
1430              *(pVi)   = refl * *(pVs)   + trans * *(pVs-x+1);
1431              *(pVi+1) = refl * *(pVs+1) + trans * *(pVs+x);
1432              *(pVi+2) = *(pVs+2);
1433              *(pVi+3) = refl * *(pVs+3) + trans * *(pVs+y+2);
1434              *(pVi+4) = refl * *(pVs+4) + trans * *(pVs-z+5);
1435              *(pVi+5) = refl * *(pVs+5) + trans * *(pVs+z+4);
1436            }
1437
```

```
1438          for(i=1;i<(c.Xsize-1);i++)
1439            for(j=1;j<(c.Zsize-1);i++)
1440            {
1441              offset  = i*x + j*z;
1442              Offset  = i + j*Z;
1443              pVi      = Vi + offset;
1444              pVs      = Vs + offset;
1445              pR      = Rin  + Offset;
1446              pZ      = Zin  + Offset;
1447
1448              refl  = *pR/(*pR + *pZ);
1449              trans = 1-refl;
1450
1451              *(pVi)   = refl * *(pVs)   + trans * *(pVs-x+1);
1452              *(pVi+1) = refl * *(pVs+1) + trans * *(pVs+x);
1453              *(pVi+2) = refl * *(pVs+2) + trans * *(pVs-y+3);
1454              *(pVi+3) = *(pVs+3);
1455              *(pVi+4) = refl * *(pVs+4) + trans * *(pVs-z+5);
1456              *(pVi+5) = refl * *(pVs+5) + trans * *(pVs+z+4);
1457            }
1458
1459
1460          break;
1461        case 2:  ;
1462          /*****************************
1463           *         Corners-2D         *
1464           *****************************/
1465
1466          // Top-Left Corner
1467          pVi = Vi;
1468          pVs = Vs;
1469          pR  = Rin;
1470          pZ  = Zin;
1471
1472          refl  = *pR/(*pR + *pZ);
1473          trans  = 1-refl;
1474
1475          *(pVi)   = *(pVs);
1476          *(pVi+1) = refl * *(pVs+1) + trans * *(pVs+x);
1477          *(pVi+2) = *(pVs+2);
1478          *(pVi+3) = refl * *(pVs+3) + trans * *(pVs+y+2);
1479
1480          // Top-Right Corner
1481          pVi = Vi + xmax;
1482          pVs = Vs + xmax;
1483          pR  = Rin  + Xmax;
1484          pZ  = Zin  + Xmax;
1485
1486          refl  = *pR/(*pR + *pZ);
1487          trans  = 1-refl;
1488
1489          *(pVi)   = refl * *(pVs)   + trans * *(pVs-x+1);
1490          *(pVi+1) = *(pVs+1);
1491          *(pVi+2) = *(pVs+2);
1492          *(pVi+3) = refl * *(pVs+3) + trans * *(pVs+y+2);
1493
1494          // Bottom-Left Corner
1495          pVi = Vi + ymax;
```

```
1496          pVs = Vs + ymax;
1497          pR  = Rin  + Ymax;
1498          pZ  = Zin  + Ymax;
1499
1500          refl  = *pR/(*pR + *pZ);
1501          trans  = 1-refl;
1502
1503          *(pVi)   = *(pVs);
1504          *(pVi+1) = refl * *(pVs+1) + trans * *(pVs+x);
1505          *(pVi+2) = refl * *(pVs+2) + trans * *(pVs-y+3);
1506          *(pVi+3) = *(pVs+3);
1507
1508          // Bottom-Right Corner
1509          pVi = Vi + xmax + ymax;
1510          pVs = Vs + xmax + ymax;
1511          pR  = Rin  + Xmax + Ymax;
1512          pZ  = Zin  + Xmax + Ymax;
1513
1514          refl  = *pR/(*pR + *pZ);
1515          trans  = 1-refl;
1516
1517          *(pVi)   = refl * *(pVs)   + trans * *(pVs-x+1);
1518          *(pVi+1) = *(pVs+1);
1519          *(pVi+2) = refl * *(pVs+2) + trans * *(pVs-y+3);
1520          *(pVi+3) = *(pVs+3);
1521
1522          /****************************
1523           *         Edges-2D         *
1524           ****************************/
1525
1526          // Top-Edge
1527          for(i=1;i<(c.Xsize-1);i++)
1528          {
1529            offset  = i*x;
1530            Offset  = i;
1531            pVi      = Vi + offset;
1532            pVs      = Vs + offset;
1533            pR      = Rin  + Offset;
1534            pZ      = Zin  + Offset;
1535
1536            refl  = *pR/(*pR + *pZ);
1537            trans  = 1-refl;
1538
1539            *(pVi)   = refl * *(pVs)   + trans * *(pVs-x+1);
1540            *(pVi+1) = refl * *(pVs+1) + trans * *(pVs+x);
1541            *(pVi+2) = *(pVs+2);
1542            *(pVi+3) = refl * *(pVs+3) + trans * *(pVs-y+3);
1543          }
1544
1545          // Bottom-Edge
1546          for(i=1;i<(c.Xsize-1);i++)
1547          {
1548            offset  = ymax + i*x;
1549            Offset  = Ymax + i;
1550            pVi      = Vi + offset;
1551            pVs      = Vs + offset;
1552            pR      = Rin  + Offset;
1553            pZ      = Zin  + Offset;
```

```
1554
1555            refl  = *pR/(*pR + *pZ);
1556            trans  = 1-refl;
1557
1558            *(pVi)   = refl * *(pVs)   + trans * *(pVs-x+1);
1559            *(pVi+1) = refl * *(pVs+1) + trans * *(pVs+x);
1560            *(pVi+2) = refl * *(pVs+2) + trans * *(pVs-y+3);
1561            *(pVi+3) = *(pVs+3);
1562          }
1563
1564          // Left-Edge
1565          for(i=1;i<(c.Ysize-1);i++)
1566          {
1567            offset  = i*y;
1568            Offset  = i*Y;
1569            pVi      = Vi + offset;
1570            pVs      = Vs + offset;
1571            pR      = Rin  + Offset;
1572            pZ      = Zin  + Offset;
1573
1574            refl  = *pR/(*pR + *pZ);
1575            trans  = 1-refl;
1576
1577            *(pVi)   = *(pVs);
1578            *(pVi+1) = refl * *(pVs+1) + trans * *(pVs+x);
1579            *(pVi+2) = refl * *(pVs+2) + trans * *(pVs-y+3);
1580            *(pVi+3) = refl * *(pVs+3) + trans * *(pVs+y+2);
1581          }
1582
1583          // Right Edge
1584          for(i=1;i<(c.Ysize-1);i++)
1585          {
1586            offset  = xmax + i*y;
1587            Offset  = Xmax + i*Y;
1588            pVi      = Vi + offset;
1589            pVs      = Vs + offset;
1590            pR      = Rin  + Offset;
1591            pZ      = Zin  + Offset;
1592
1593            refl  = *pR/(*pR + *pZ);
1594            trans  = 1-refl;
1595
1596            *(pVi)   = refl * *(pVs)   + trans * *(pVs-x+1);
1597            *(pVi+1) = *(pVs+1);
1598            *(pVi+2) = refl * *(pVs+2) + trans * *(pVs-y+3);
1599            *(pVi+3) = refl * *(pVs+3) + trans * *(pVs+y+2);
1600          }
1601          break;
1602        case 1: ;
1603          // Left Corner
1604          pVi = Vi;
1605          pVs = Vs;
1606          pR  = Rin;
1607          pZ  = Zin;
1608
1609          refl  = *pR/(*pR + *pZ);
1610          trans = 1-refl;
1611
```

```
1612          *(pVi)   = *(pVs);
1613          *(pVi+1) = refl * *(pVs+1) + trans * *(pVs+x);
1614
1615          // Right Corner
1616          pVi = Vi + xmax;
1617          pVs = Vs + xmax;
1618          pR  = Rin  + Xmax;
1619          pZ  = Zin  + Xmax;
1620
1621          refl  = *pR/(*pR + *pZ);
1622          trans = 1-refl;
1623
1624          *(pVi)   = refl * *(pVs)   + trans * *(pVs-x+1);
1625          *(pVi+1) = *(pVs+1);
1626          break;
1627
1628          /*    4   2
1629           *     \  ^
1630           *      \ |
1631           *       \|
1632           * 0 <----+----> 1
1633           *        |\
1634           *        | \
1635           *        V  \
1636           *        3   5
1637           */
1638
1639      } // switch
1640    }
1641  }
1642
1643  void
1644  T1decay(
1645      struct confStruct c,
1646       double* Vi,
1647      double decayConst
1648      )
1649  {
1650    long long i;
1651    long long maxSize=1;
1652
1653    switch(c.dim)
1654    {
1655      case 3:
1656        maxSize *= c.Zsize;
1657      case 2:
1658        maxSize *= c.Ysize;
1659      case 1:
1660        maxSize *= c.Xsize;
1661    }
1662
1663    maxSize *= c.dim*2;
1664
1665    for(i=0;i<maxSize;i++)
1666      *(Vi+i) -= decayConst * *(Vi+i);
1667  }
1668
1669  /******************
```

```
1670    * Calculate \phi *
1671    ******************/
1672   void
1673   calcSums(
1674       struct confStruct c,
1675       long long n,
1676       double* Vi,
1677       double* Box,
1678        double* Bulk,
1679        double* Intensity,
1680        double* BulkInten
1681       )
1682   {
1683     long BoxSize = 1;
1684     long BulkSize = 1;
1685
1686     *Box = 0;
1687     *Bulk = 0;
1688
1689
1690     if(c.round == 0.0)
1691     {
1692       switch(c.dim)
1693       {
1694         case 3:  BoxSize *= c.Box5 - c.Box4 + 1;
1695                  BulkSize *= c.Zsize;
1696         case 2:  BoxSize *= c.Box3 - c.Box2 + 1;
1697                  BulkSize *= c.Ysize;
1698         case 1:  BoxSize *= c.Box1 - c.Box0 + 1;
1699                  BulkSize *= c.Xsize;
1700       }
1701
1702       BulkSize -= BoxSize;
1703
1704     //  printf("===\nBoxSize=%ld\n===\n", BoxSize);
1705
1706       switch(c.dim)
1707       {
1708       case 1:;
1709         {
1710           register long i;
1711           for(i=0;i<c.Xsize;i++)
1712           {
1713             if(i>=c.Box0 && i<=c.Box1)
1714               *Box  += *(Vi+(c.x * i)) + *(Vi+(c.x * i)+1);
1715             else
1716               *Bulk += *(Vi+(c.x * i)) + *(Vi+(c.x * i)+1);
1717           }
1718         }
1719         break;
1720
1721       case 2:;
1722         {
1723           long i, j;
1724           for(i=0;i<c.Ysize;i++)
1725             for(j=0;j<c.Xsize;j++)
1726             {
1727               register long off = j * c.x + i * c.y;
```

```
1728              if(i>=c.Box2 && i<=c.Box3 && j>=c.Box0 && j<=c.Box1)
1729                *Box  += *(Vi+off) + *(Vi+off+1) + *(Vi+off+2) + *(Vi+off+3);
1730              else
1731                *Bulk += *(Vi+off) + *(Vi+off+1) + *(Vi+off+2) + *(Vi+off+3);
1732            }
1733        }
1734        break;
1735      case 3:;
1736        {
1737          long i, j, k;
1738          for(i=0;i<c.Zsize;i++)
1739            for(j=0;j<c.Ysize;j++)
1740              for(k=0;k<c.Xsize;k++)
1741              {
1742                double *ptr = Vi + k * c.x + j * c.y + i * c.z;
1743
1744    //            printf("\nn=%ld\n", n);
1745    //            printf(" Pre: Box=%10.6lf    Bulk=%10.6lf\n", *Box, *Bulk);
1746    //            printf(" Pre: Box=> %p    Bulk=> %p\n", Box, Bulk);
1747
1748                if(i>=c.Box4 && i<=c.Box5 && j>=c.Box2 && \
1749                   j<=c.Box3 && k>=c.Box0 && k<=c.Box1)
1750                *Box  += *(ptr)   + *(ptr+1) + *(ptr+2) +\
1751                         *(ptr+3) + *(ptr+4) + *(ptr+5);
1752                else
1753                *Bulk += *(ptr)   + *(ptr+1) + *(ptr+2) +\
1754                         *(ptr+3) + *(ptr+4) + *(ptr+5);
1755    //            printf("Post: Box=%10.6lf    Bulk=%10.6lf\n", *Box, *Bulk);
1756    //            printf("Post: Box=> %p    Bulk=> %p\n", Box, Bulk);
1757              }
1758          break;
1759        }
1760      }
1761
1762      *(Intensity+n) = 100 * *Box  / (*Box + *Bulk);
1763      *(BulkInten+n) = 100 * *Bulk / (*Box + *Bulk);
1764    //  printf("Intesnsity = %lf\n", *(Intensity+n));
1765    }
1766    else     // round != 0.0
1767    {
1768      double distance;
1769      long i, j, k;
1770      BulkSize = 0;
1771      BoxSize = 0;
1772
1773      switch(c.dim)
1774      {
1775        case 3:
1776          for(k=0;k<c.Zsize;k++)
1777            for(j=0;j<c.Ysize;j++)
1778              for(i=0;i<c.Xsize;i++)
1779              {
1780                double x, y, z;
1781                x = (double) i;
1782                y = (double) j;
1783                z = (double) k;
1784                double *ptr = Vi + i * c.x + j * c.y + k * c.z;
1785                distance = sqrt(pow(x,2)+pow(y,2)+pow(z,2));
```

```
1786                if(distance <= c.round)
1787                {
1788                  *Box += *(ptr);
1789                  *Box += *(ptr+1);
1790                  *Box += *(ptr+2);
1791                  *Box += *(ptr+3);
1792                  *Box += *(ptr+4);
1793                  *Box += *(ptr+5);
1794                  ++BoxSize;
1795                }
1796                else
1797                {
1798                  *Bulk += *(ptr);
1799                  *Bulk += *(ptr+1);
1800                  *Bulk += *(ptr+2);
1801                  *Bulk += *(ptr+3);
1802                  *Bulk += *(ptr+4);
1803                  *Bulk += *(ptr+5);
1804                  ++BulkSize;
1805                }
1806              }
1807
1808          break;
1809        case 2:
1810          for(j=0;j<c.Ysize;j++)
1811            for(i=0;i<c.Xsize;i++)
1812            {
1813              double x, y;
1814              x = (double) i;
1815              y = (double) j;
1816              distance = sqrt(pow(x,2)+pow(y,2));
1817              double *ptr = Vi + i * c.x + j * c.y;
1818              if(distance <= c.round)
1819              {
1820                *Box += *(ptr);
1821                *Box += *(ptr+1);
1822                *Box += *(ptr+2);
1823                *Box += *(ptr+3);
1824                ++BoxSize;
1825              }
1826              else
1827              {
1828                *Bulk += *(ptr);
1829                *Bulk += *(ptr+1);
1830                *Bulk += *(ptr+2);
1831                *Bulk += *(ptr+3);
1832                ++BulkSize;
1833              }
1834            }
1835          break;
1836      }
1837      *(Intensity+n) = 100 * *Box  / (*Box + *Bulk);
1838      *(BulkInten+n) = 100 * *Bulk / (*Box + *Bulk);
1839    }
1840  } // calcSums
1841
1842  // vim:set ts=2 sw=2:
```

G.2.15 Makefile

```
1  #  File: Makefile
2  #
3  #  Author: Frederik Klama
4  #  Copyright 2010 Frederik Klama
5  #
6  #  This file is part of TLM-Simulator.
7  #
8  #  TLM-Simulator is free software: you can redistribute it and/or modify
9  #  it under the terms of the GNU General Public License as published by
10 #  the Free Software Foundation, either version 3 of the License, or
11 #  (at your option) any later version.
12 #
13 #  TLM-Simulator is distributed in the hope that it will be useful,
14 #  but WITHOUT ANY WARRANTY; without even the implied warranty of
15 #  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
16 #  GNU General Public License for more details.
17 #
18 #  You should have received a copy of the GNU General Public License
19 #  along with TLM-Simulator.  If not, see <http://www.gnu.org/licenses/>.
20
21 CFLAGS := -W
22
23 objects = dataStruct.o fillBoxMag.o output.o worker.o StringTools.o
24
25 default: all
26
27 verb-debug: CFLAGS := $(CFLAGS) -g -DDEBUG
28 verb-debug: all
29
30 debug: CFLAGS := $(CFLAGS) -g
31 debug: all
32
33 all: TLM-Simulator
34
35 clean:
36   rm -f TLM-Simulator
37   rm -f *.o
38
39 $(objects): %.o: %.c common.h
40   gcc -c $(CFLAGS) $< -o $@
41
42 parser.o: parser.c StringTools.h StringTools.o
43   gcc -c $(CFLAGS) $< -o $@
44
45 TLM-Sim.o: TLM-Sim.c StringTools.h dataStruct.h fillBoxMag.h output.h\
46          parser.h worker.h
47   gcc -c $(CFLAGS) $< -o $@
48
49 TLM-Simulator: TLM-Sim.o parser.o $(objects)
50   gcc $(CFLAGS) $^ -o $@
```

G.3 TLM helper tools

G.3.1 table2conf.pl

```
1  #!/usr/bin/perl
```

```
2  # vim:set ts=2 sw=2:
3  #
4  #  File: table2conf.pl
5  #
6  #  Author: Frederik Klama
7  #  Copyright 2010 Frederik Klama
8  #
9  #  This file is part of TLM-Simulator.
10 #
11 #  TLM-Simulator is free software: you can redistribute it and/or modify
12 #  it under the terms of the GNU General Public License as published by
13 #  the Free Software Foundation, either version 3 of the License, or
14 #  (at your option) any later version.
15 #
16 #  TLM-Simulator is distributed in the hope that it will be useful,
17 #  but WITHOUT ANY WARRANTY; without even the implied warranty of
18 #  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
19 #  GNU General Public License for more details.
20 #
21 #  You should have received a copy of the GNU General Public License
22 #  along with TLM-Simulator.  If not, see <http://www.gnu.org/licenses/>.
23
24 #
25 #  This program is intended for bulk simulations over a large parameter
26 #  space. The parameters are defines as arrays with one or more entries
27 #  and the program will automatically generate a directory structure.
28 #  It will then generate a configuration file for each possible permu-
29 #  tation of the parameters and put it into the fitting directory.
30 #  Lastly it will run the simulation software for each of these permu-
31 #  tations and even allows for paralell processing by running a pre-
32 #  defined number of simulator processes.
33 #  Lastly it will generate graphs, as scalable vector graphic and in
34 #  TeX-format, for each simulation using gnuplot.
35 #
36
37 use strict;
38 use Parallel::ForkManager;
39
40 # The maximum number of simulation processes to run at the same time
41 my $numProcs = 2;
42
43 # Directory where TLM-Simulator is installed
44 my $BIN_DIR  = "/Users/fklama/Documents/Uni/UEA/";
45    $BIN_DIR .= "FrederikNMR/TLM/TLM-Simulator";
46
47 # You should not need to modify the next two lines
48 my $confGen  = "$BIN_DIR/confGen.pl";
49 my $TLM_Sim  = "$BIN_DIR/TLM-Simulator";
50
51 # The directory in which the directory structure is built and
52 # into which the results are stored
53 my $basePath = "$BIN_DIR/runHere";
54
55 # Defining the parameter space
56 # Each of the parameters (@dim, @size, @boxSize, @R1, @R2, @Z, @T1 and
57 # @graphSize) must be set to either a single or multiple values
58 # Take care, the amount of simulations run is equal to the product
59 # of the number of items in each array (except @graphSize which only
```

```
60  # causes one plot be generated for each of these sizes
61  my @dim      = (1,2,3);
62  my @size     = (50);
63  my @boxSize  = (5,10,15,25,35,40,45);
64  my @R1       = (80,85,90,95,100,105,110);
65  my @R2       = (100);
66  my @Z        = (100);
67  my @T1       = (0, 100, 250, 500, 1000);
68  my @graphSize = (300, 500, 1000, 2000, 3000, 5000, 7000, 10000);
69  my $path;
70
71  # Some single parameters
72  my $initValBulk = 0;
73  my $initValBox  = 100;
74  my $LL_LR       = 1; # 0 = both, 1 = LL, 2 = LR
75  my $verbose     = 0;
76  my $steps       = 10000;
77
78  # Variable declarations
79  my $size;
80  my $box;
81  my $R1;
82  my $R2;
83  my $R;
84  my $Z;
85  my $T1;
86  my $arg;
87  my $oArg;
88  my $dim;
89  my @dirList;
90  my @dispList;
91  my @typeList;
92  my @runList;
93  my @argList;
94  my @mkConfList;
95
96  # Subroutine to write the configuration files
97  sub writeConf {
98    unless($LL_LR == 2) { # LL
99      system("mkdir $path/LL");
100     push @mkConfList, "$confGen $arg --LL > $path/LL/tlm.conf";
101   }
102   unless($LL_LR == 1) { # LR
103     system("mkdir $path/LR");
104     push @mkConfList, "$confGen $arg --LR > $path/LR/tlm.conf";
105   }
106 }
107
108 foreach(@dim) { # 1D,2D,3D
109   $dim = $_;
110
111   # Create Directory for dimensionality
112   mkdir "$basePath/$dim"."D";
113
114   foreach(@size) { # System size
115     my $s = $_;
116
117     # Generate Size String
```

```
118       $size  = "$s";
119       $size .= ":$s" if($dim>1);
120       $size .= ":$s" if($dim>2);
121
122       # Create Directory for Size
123       system("mkdir $basePath/$dim"."D/S$size");
124
125       foreach(@boxSize) { # Box Sizes
126         my $b = $_;
127
128         # Generate Box String
129         $b -= 1; # Start counting at 0 thus -1
130         $box  = "0:$b";
131         $box .= ":0:$b" if($dim>1);
132         $box .= ":0:$b" if($dim>2);
133
134         # Create Directory for Box
135         system("mkdir $basePath/$dim"."D/S$size/B$box");
136
137         foreach(@R1) { # R Bulk
138           $R1 = $_;
139           system("mkdir $basePath/$dim"."D/S$size/B$box/R1_$R1");
140           foreach(@R2) { # R Box
141             $R2 = $_;
142             $R = "$R1:$R2";
143             system("mkdir $basePath/$dim"."D/S$size/B$box/R1_$R1/R2_$R2");
144
145             foreach(@Z) { # Z
146               $Z = "$_:$_";
147               $path = "$basePath/$dim"."D/S$size/B$box/R1_$R1/R2_$R2/Z$Z";
148               system("mkdir $path");
149               my $Path = $path;
150
151               foreach(@T1) { # T1
152                 $T1 = $_;
153                 $path = $Path."/T1_$T1";
154                 system("mkdir $path");
155
156                 # Generating directory list
157                 push @dirList, "$path/LL" unless($LL_LR == 2);
158                 push @dirList, "$path/LR" unless($LL_LR == 1);
159
160                 # Generating the arguments for confGen.pl
161                 $arg  = "--T1=$T1 --size=$size ";
162                 $arg .= "--steps=$steps --initVal=$initValBulk:$initValBox ";
163                 $arg .= "\"--box=$box\" --R=$R --Z=$Z ";
164                 $arg .= "\"--areas=0";
165                 $arg .= ":$b";
166                 $arg .= ":0" if($dim>1);
167                 $arg .= ":$b" if($dim>1);
168                 $arg .= ":0" if($dim>2);
169                 $arg .= ":$b" if($dim>2);
170                 $arg .= "\" ";
171                 $arg .= " -v=$verbose" if($verbose);
172                 $oArg  = "$dim"."D  Size=$s  Box=".($b+1)." ";
173                 $oArg .= "  R=$R  Z=$Z  T1=$T1";
174                 writeConf();
175                 push @argList, "$oArg  LL\n" unless($LL_LR == 2);
```

```
176                 push @argList, "$oArg  LR\n" unless($LL_LR == 1);
177               }
178             }
179           }
180         }
181       }
182     }
183   }
184
185   # Defining variables for tracking configuration generation
186   my $conf = 0;
187   my $confMax = @mkConfList;
188
189   # Iterate through the list of to be generated configurations
190   while(@mkConfList) {
191
192     # Take first item out of array
193     my $run = shift @mkConfList;
194     # Increase conf-count
195     $conf++;
196     push @typeList, 0; # Type 0 is configuration
197     push @dispList, "Creating config $conf/$confMax\n";
198     push @runList, $run;
199   }
200
201   # Defining variables for tracking simulations
202   my $procMax = @runList;
203   my $proc = 0;
204
205   # Iterate through directories
206   while(@dirList) {
207     $proc++;
208     # Setting string to be displayed
209     $oArg = shift @argList;
210     push @dispList, "Starting Process $proc/$procMax:\n$oArg\n\n";
211
212     push @typeList, 1; # Type 1 is simulation
213
214     # Pushing the directories onto the runList to be forked
215     my $p = shift @dirList;
216     push @runList, $p;
217   }
218
219   my $i = -1;
220   my $manager = new Parallel::ForkManager( $numProcs );
221
222   # This loop will spawn processes so that there are always $numProcs
223   # processes running if possible
224   while($i <= @runList) {
225     $i++;
226
227     # After this we are running multi-threaded
228     $manager->start and next;
229
230     # Print the text which was set earlier
231     print $dispList[$i];
232
233     # Set $p to the parameter for the current task
```

```
234     my $p = $runList[$i];
235
236     if($typeList[$i] == 0) { # Config
237       # $p is a command line that will generate the configuration file
238       # run it
239       system($p);
240     } else { # $typeList[$i] == 1   i.e. simulation
241       # $p is a directory in which TLM-Simulator is to be started
242       # Change to the directory
243       chdir $p;
244
245       # Start simulation
246       system("$TLM_Sim > OUTPUT");
247
248       # Call gnuplot and in write mode (i.e. write to gnuplots STDIN)
249       open GNUPLOT, "| gnuplot";
250
251       # Generate one graph per @graphSize
252       foreach my $size (@graphSize) {
253         print GNUPLOT "set terminal svg size 800,600 dynamic\n";
254         print GNUPLOT "set output \"".$size.".svg\"\n";
255         print GNUPLOT "plot [0:".$size."] \'OUTPUT\' using 1 title \"Box\", ";
256         print GNUPLOT "\'OUTPUT\' using 2 title \"Bulk\"\n";
257         print GNUPLOT "set terminal texdraw\n";
258         print GNUPLOT "set output \"".$size.".tex\"\n";
259         print GNUPLOT "plot [0:".$size."] \'OUTPUT\' using 1 title \"Box\", ";
260         print GNUPLOT "\'OUTPUT\' using 2 title \"Bulk\"\n";
261       }
262
263       # Close connection to gnuplot
264       close GNUPLOT;
265     }
266     $manager->finish;
267   }
```

G.3.2 confGen.pl

```
1   #!/usr/bin/perl
2   #
3   #  File: confGen.pl
4   #
5   #  Author: Frederik Klama
6   #  Copyright 2010 Frederik Klama
7   #
8   #  This file is part of TLM-Simulator.
9   #
10  #  TLM-Simulator is free software: you can redistribute it and/or modify
11  #  it under the terms of the GNU General Public License as published by
12  #  the Free Software Foundation, either version 3 of the License, or
13  #  (at your option) any later version.
14  #
15  #  TLM-Simulator is distributed in the hope that it will be useful,
16  #  but WITHOUT ANY WARRANTY; without even the implied warranty of
17  #  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
18  #  GNU General Public License for more details.
19  #
20  #  You should have received a copy of the GNU General Public License
21  #  along with TLM-Simulator.  If not, see <http://www.gnu.org/licenses/>.
22
```

```
23  #
24  #  This program is called with several command line options and outputs
25  #  a configuration file for TLM-Simulator on STDOUT.
26  #  It is recommended to redirect the output to 'tlm.conf'
27  #
28
29  use strict;
30
31  use Getopt::Long;
32
33  my $dim;
34  my $sizeStr;
35  my $Xsize = 0;
36  my $Ysize = 0;
37  my $Zsize = 0;
38  my $size;
39  my $helpSW;
40  my $steps = 1000;
41  my $LL_SW;
42  my $LR_SW;
43  my $model = "LL";
44  my $verboseSW;
45  my $BoxStr;
46  my @Box;
47  my $R_Str;
48  my $Z_Str;
49  my $initValStr;
50  my $areasStr;
51  my @R_vals;
52  my @Z_vals;
53  my @initVal;
54  my @areas;
55  my @R;
56  my @Z;
57  my @iV;
58  my $R;
59  my $Z;
60  my $iV;
61  my $i;
62  my $j;
63  my $k;
64  my $T1;
65  my $round;
66
67
68  my $result = GetOptions (
69    "help|h"    => \$helpSW,
70    "size=s"    => \$sizeStr,
71    "steps=i"    => \$steps,
72    "LL"        => \$LL_SW,
73    "LR"        => \$LR_SW,
74    "verbose|v=i"  => \$verboseSW,
75    "box=s"      => \$BoxStr,
76    "R=s"        => \$R_Str,
77    "Z=s"        => \$Z_Str,
78    "initVal=s"  => \$initValStr,
79    "areas=s"    => \$areasStr,
80    "T1=i"      => \$T1,
```

```
81    "round=f"    => \$round
82  );
83  # Can't have both a box and a round area. If they are both
84  # set, abort with error message
85  die("Can not define box and round!") if($BoxStr && $round);
86
87  # Split the $sizeStr into the x,y,z-components
88  ($Xsize, $Ysize, $Zsize) = split(/:/, $sizeStr);
89
90  # Determine dimensionality by seeing which dimensions are set
91  $dim = 3;
92  $dim = 2 if($Zsize==0);
93  $dim = 1 if($Ysize==0);
94
95  # Calculate total size
96  $size  = $Xsize;
97  $size *= $Ysize if($dim>1);
98  $size *= $Zsize if($dim>2);
99
100 # Split R,Z and inital magnetization values
101 # First value is Bulk, the others are the areas
102 @R_vals    = split(/:/, $R_Str);
103 @Z_vals    = split(/:/, $Z_Str);
104 @initVal  = split(/:/, $initValStr);
105
106 # Link-Line xor Link-Resistor  i.e. not both
107 die("Can not set --LL and --LR at the same time!") if($LL_SW && $LR_SW);
108
109 # Default for $model is "LL"
110 $model = "LR" if($LR_SW);
111
112
113 # Essentially two different programs from here on
114 if($round) {
115
116   # Round makes no sense with a 1D system
117   if($dim<2 || $dim>3) {
118     die("--round only makes sense for 2D or 3D systems!") ;
119   }
120
121   # Check if it is within the simulated system
122   if($Xsize<$round || $Ysize<$round || ($Zsize<$round && $dim==3)) {
123     die("--round can not be bigger than simulated system!");
124   }
125
126   # Very small numbers make no sense
127   if($round<2) {
128     die("--round has to be a positive number bigger than 2!");
129   }
130
131   # Print main part of the configuration file
132   print "dim = $dim\n";
133   print "Xsize = $Xsize\n";
134   print "Ysize = $Ysize\n" if($dim>1);
135   print "Zsize = $Zsize\n" if($dim>2);
136   print "steps = $steps\n";
137   print "verbose = ";
138   print $verboseSW."\n";
```

```
139     print "round = $round\n";
140     print "T1 = $T1\n" if($T1>0);
141     print "model = $model\n\n";
142
143     # After the following line the potential as well as the values for Z and R
144     # are defined
145     print "{Begin Data}\n";
146
147
148     if($dim==2) {
149       for($j=0;$j<$Ysize;$j++) {
150         for($i=0;$i<$Xsize;$i++) {
151           # Calculate the distance from the origin (0,0)
152           my $dist = sqrt($i**2 + $j**2);
153
154           print "# x=$i  y=$j  distance=$dist\n";
155           if($dist > $round) {
156             # We are outside of the circle
157             print "P = ".($initVal[0])/($dim*2)."\n";
158             print "R = ".$R_vals[0]."\n";
159             print "Z = ".$Z_vals[0]."\n";
160           } else {
161             # We are inside the circle
162             print "P = ".($initVal[1])/($dim*2)."\n";
163             print "R = ".$R_vals[1]."\n";
164             print "Z = ".$Z_vals[1]."\n";
165           }
166         }
167       }
168     } else { # $dim==3
169       for($k=0;$k<$Zsize;$k++) {
170         for($j=0;$j<$Ysize;$j++) {
171           for($i=0;$i<$Xsize;$i++) {
172             # Calculate the distance from the origin (0,0)
173             my $dist = sqrt($i**2 + $j**2 + $k**2);
174
175             print "# x=$i  y=$j  z=$k  distance=$dist\n";
176             if($dist > $round) {
177               # We are outside of the circle
178               print "P = ".($initVal[0])/($dim*2)."\n";
179               print "R = ".$R_vals[0]."\n";
180               print "Z = ".$Z_vals[0]."\n";
181             } else {
182               # We are inside of the circle
183               print "P = ".($initVal[1])/($dim*2)."\n";
184               print "R = ".$R_vals[1]."\n";
185               print "Z = ".$Z_vals[1]."\n";
186             }
187           }
188         }
189       }
190     }
191
192
193   } # if($round)
194   else
195   { # Box instead of round
196
```

```
197     # Split up the single coordinates of the box
198     @Box       = split(/:/, $BoxStr);
199
200     # Split up the different areas (if there are more than one)
201     @areas    = split(/,/, $areasStr);
202
203     # One coorinate pair times dimension
204     die("Box has to have two entrys per dimension!") if(@Box < $dim*2);
205
206     # Die if any dimension is smaller than 2
207     if($Xsize<2 || ($dim>1 && $Ysize<2) || ($dim>2 && $Zsize<2)) {
208       die("Dimensions have to be bigger than 1!");
209     }
210
211     # Getting the bulk values out of the arrays
212     $R  = shift @R_vals;
213     $Z  = shift @Z_vals;
214     $iV = (shift @initVal) / ($dim*2);
215
216     # Simply set the whole system to the bulk values
217     for($i=0; $i<$size; $i++)
218     {
219       $Z[$i]  = $Z;
220       $R[$i]  = $R;
221       $iV[$i] = $iV;
222     }
223
224     # Now set anything that differs from the bulk values
225     foreach my $area (@areas)
226     {
227       $R  = shift @R_vals;
228       $Z  = shift @Z_vals;
229       $iV = (shift @initVal) / ($dim*2);
230
231       # Split up the coordinates of this area
232       my @coords = split(/:/, $area);
233
234       # One coordinate pair time dimension
235       die("Define boxes with two corners please!") if(@coords != $dim*2);
236
237       # Split the coordinates into one array each
238       my @min;
239       my @max;
240       for(my $i=0;$i<@coords;$i++) {
241         push @min, $coords[$i] if($i % 2);
242         push @max, $coords[$i] unless($i % 2);
243       }
244
245       # Each set of coordinates should be the size of the dimensionality
246       if(@min != $dim || @max != $dim)
247       { die("The coordinates of the corners have to be ".$dim."-dimensional!");  }
248
249       # Sort the coordinates to get one pair closest and one furthest
250       # from the origin (0,0)
251       my $tmp;
252       for($i=0; $i<$dim; $i++) {
253         if($min[$i] > $max[$i]) {
254           $tmp = $min[$i];
```

```
255           $min[$i] = $max[$i];
256           $max[$i] = $tmp;
257         }
258       }
259
260       # Write the values for this area into the array
261       if($dim == 3)
262       {
263         for($k=$min[2]; $k<=$max[2]; $k++) {
264           for($j=$min[1]; $j<=$max[1]; $j++) {
265             for($i=$min[0]; $i<=$max[0]; $i++) {
266               my $offset = ($k * $Ysize + $j) * $Xsize + $i;
267               $Z[$offset]  = $Z;
268               $R[$offset]  = $R;
269               $iV[$offset] = $iV;
270             }
271           }
272         }
273       }
274       elsif($dim == 2)
275       {
276         for($j=$min[1]; $j<=$max[1]; $j++) {
277           for($i=$min[0]; $i<=$max[0]; $i++) {
278             my $offset = $j * $Xsize + $i;
279             $Z[$offset]  = $Z;
280             $R[$offset]  = $R;
281             $iV[$offset] = $iV;
282           }
283         }
284       }
285       else # $dim == 1
286       {
287         for($i=$min[0]; $i<=$max[0]; $i++) {
288           $Z[$i]  = $Z;
289           $R[$i]  = $R;
290           $iV[$i] = $iV;
291         }
292       }
293     }
294
295     # Printing main part of the configuration file
296     print "dim = $dim\n";
297     print "Xsize = $Xsize\n";
298     print "Ysize = $Ysize\n" if($dim>1);
299     print "Zsize = $Zsize\n" if($dim>2);
300     print "steps = $steps\n";
301     print "verbose = ";
302     print $verboseSW."\n";
303     print "Box0 = ".$Box[0]."\n";
304     print "Box1 = ".$Box[1]."\n";
305     print "Box2 = ".$Box[2]."\n" if($dim>1);
306     print "Box3 = ".$Box[3]."\n" if($dim>1);
307     print "Box4 = ".$Box[4]."\n" if($dim>2);
308     print "Box5 = ".$Box[5]."\n" if($dim>2);
309     print "T1 = $T1\n" if($T1>0);
310     print "model = $model\n\n";
311
312     print "{Begin Data}\n";
```

```
313
314     # Now printing values for potential, R and Z
315     if($dim == 3) {
316       for($k=0; $k<$Zsize; $k++) {
317         for($j=0; $j<$Ysize; $j++) {
318           for($i=0; $i<$Xsize; $i++) {
319               my $offset = ($k * $Ysize + $j) * $Xsize + $i;
320               print "# x=$i  y=$j  z=$k  offset=$offset\n";
321               print "P = ".$iV[$offset]."\n";
322               print "R = ".$R[$offset]."\n";
323               print "Z = ".$Z[$offset]."\n";
324           }
325         }
326       }
327     }
328     elsif($dim == 2) {
329       for($j=0; $j<$Ysize; $j++) {
330         for($i=0; $i<$Xsize; $i++) {
331           my $offset = $j * $Xsize + $i;
332           print "# x=$i  y=$j  offset=$offset\n";
333           print "P = ".$iV[$offset]."\n";
334           print "R = ".$R[$offset]."\n";
335           print "Z = ".$Z[$offset]."\n";
336         }
337       }
338     }
339     else # $dim == 1
340     {
341       for($i=0; $i<$Xsize; $i++) {
342         print "# x=$i\n";
343         print "P = ".$iV[$i]."\n";
344         print "R = ".$R[$i]."\n";
345         print "Z = ".$Z[$i]."\n";
346       }
347
348     }
349   }
350
351   # vim:set ts=2 sw=2:
```