

AG Fajara

DAMARIS Tutorial

Markus Rosenstihl

November 6, 2010

Contents

1	Python Front End for Experimenters	5
1.1	Basics	5
1.1.1	Configuration	6
1.1.2	Experiment Script	8
1.1.3	Result Script	9
1.2	Accumulation	9
1.3	Phase Cycling	10
1.4	Changing Parameters	11
1.5	Data Handling	13
1.6	Fourier Transform Module	16
1.7	Good Practices	18
1.8	Tuning and Development	18
1.9	Data Processing Outside DAMARIS	19
1.10	List of Commands	19
2	DAMARIS for Developers	25
2.1	The Back End	25
2.2	The DAC driver	27
3	Scripts for Various Experiments	35

1 Python Front End for Experimenters

DAMARIS [3] is the software, established by Achim Gädke, used to control the PFG spectrometer. There are two separate parts: the back end, which controls the hardware and executes the pulse program, and a front end where the pulse program is defined and results are displayed. The programming language used for the scripts in the front end is Python, because it is easy to learn and the programs are easy to maintain. A introduction in Python is out of the scope for this thesis, however, because Python is widely used outside DAMARIS, there are a wealth of excellent tutorials available¹.

1.1 Basics

The basic idea in DAMARIS is the separation of an experiment in three steps:

- Experiment description where the experiment sequence is described
- Back end executes the sequences obtained from the experiment description
- Results from the back end are processed in the result script

The front end is composed of five tabs (Figure 1.1): Experiment, Result, Display, Log and Configuration. This introduction into DAMARIS will begin with a NMR “Hello World” program, i.e. recording a FID. More elaborated techniques like phase cycling and PFG will be discussed in chapter 1.2.

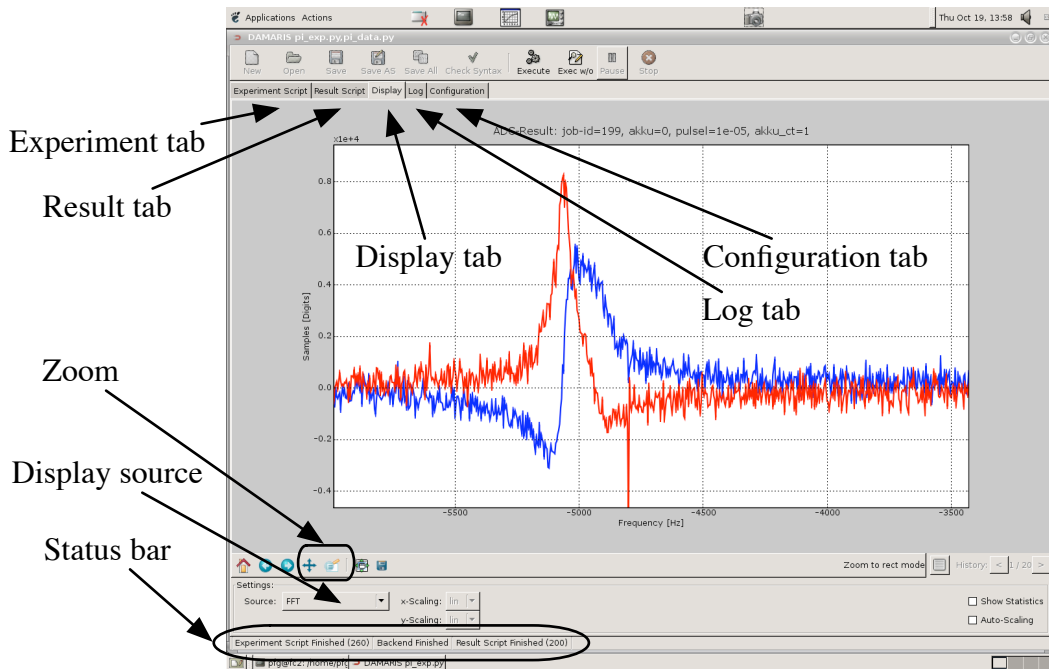


Figure 1.1: The DAMARIS front end

1.1.1 Configuration

The Configuration tab (Figure 1.2) offers a convenient way to setup DAMARIS. Following is a short description of the options:

1. Toggle the start of the back end
2. Path to the back end executable, here **PFGcore.exe**
3. Path to the spool directory, where the job files will be written or picked up
4. Delete the job files after execution
5. Filename of the logfile

¹For a short introduction in Python see:

- <http://www.python.org/doc/current/tut/tut.html>
- <http://www.awaretek.com/tutorials.html#begin>
- http://swaroopch.info/text/Byte_of_Python:Main_Page
- <http://www.ibiblio.org/obp/thinkCSpy>
- <http://www.freenetpages.co.uk/hp/alan.gauld>

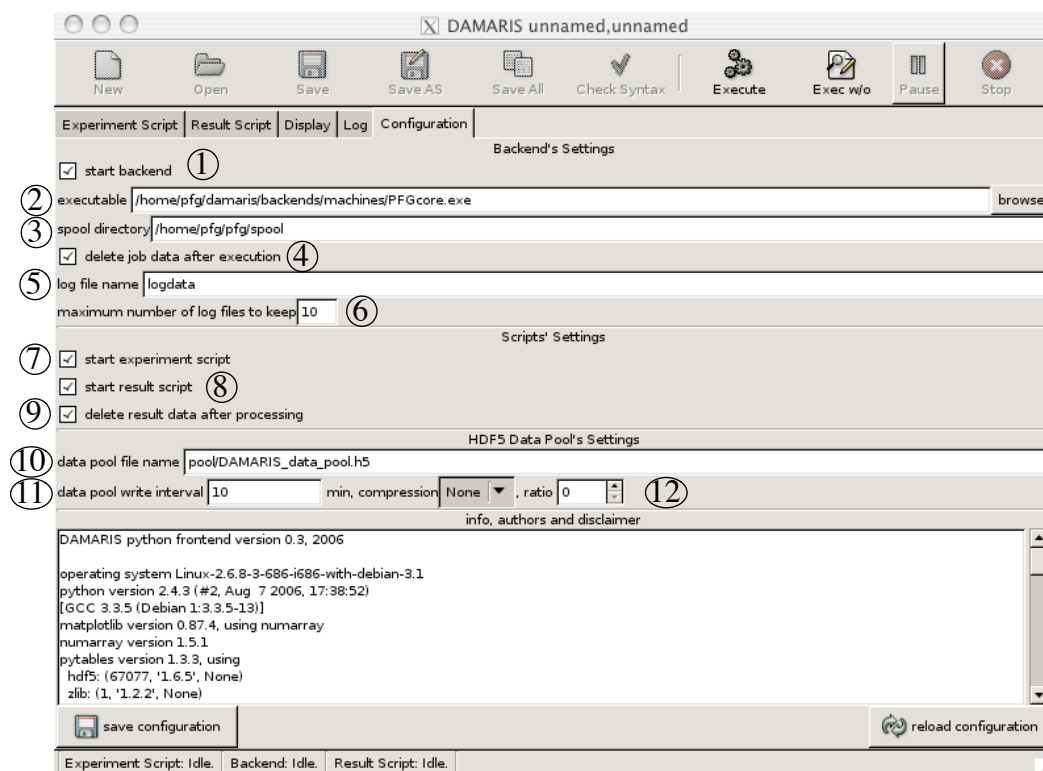


Figure 1.2: Configuration tab

6. Number of logfiles to keep
7. Toggle the start of the experiment script
8. Toggle the start of the result script
9. Toggle the deletion of results after they have been processed by the result script (see 1.8)
10. Filename of the data pool file
11. Write interval after which the data pool is rewritten
12. Compression settings (see page 13)
13. Authors, License and version numbers of used components

1.1.2 Experiment Script

Listing 1.1: Experiment script for recording a FID

```
1
2 def fid_experiment():
3     # create a sequence
4     e=Experiment()
5     e.set_frequency(frequency=300.01e6, phase=0)# set the frequency
6     e.ttl_pulse(length=5e-6, channel=1)          # gate for the RF pulse
7     e.ttl_pulse(length=2e-6, channel=1+2)        # RF pulse
8     e.wait(10e-6)                                # dead time
9     # record 1024 samples at sampling-rate 2 MHz
10    # with +/- 2V range
11    e.record(samples=1024, frequency=2e6, sensitivity=2)
12    return e
13
14 def experiment():
15    yield fid_experiment()
```

On the first line, the basic Experiment module is imported which loads the methods and commands to control an experiment. Next, a function **fid_experiment** is defined which will contain everything necessary to run a single scan. The first line inside the function creates an object **e** in which the states will be written by subsequent application of methods to this experiment object. This sequence of states is then passed to the caller. When the “Execute” (🔌) button is pressed, the experiment script is scanned for a function called **experiment**. Thus, to run **fid_experiment**, the **experiment** function has to contain a call to the **fid_experiment**.

Behind the scenes is the experiment script that writes the state sequence from **fid_experiment**, an XML² file, into the spool directory and the back end is then executed. The back end

²<http://www.w3c.org/XML>

is simply reads these files, translates the contents to pulse programmer code and stores the generated code into the memory of the pulse card. Finally, the pulse program is executed. Then the results are written by the back end into the spool directory, where they are provided to the result script. In the result script, the function **result** is then executed.



1.1.3 Result Script

The function **result** in the result tab defines what should be done with the results. Here is a short example how to display the recorded signal:

Listing 1.2: Result script which is displaying the result

```
1 def result():
2     # loop over the incoming results
3     for timesignal in results:
4         # provide the timesignal to the display
5         data["Timesignal"]=timesignal
```

After pressing the “Execute” (🔌) button to start the experiment, the Display tab is opened. With the drop down menu right to “Source” under the graph window one can choose which data should be displayed. This list is built dynamically by the data dictionary in the result script, where the key denotes the name to be shown and the value contains the data. In this example we overwrite the data in the key (Timesignal) on every scan with the new result. There are now two possibilities to export the figure:

1. Saving the plot as a figure (EPS, PNG, JPEG and more) by pressing the button  in the toolbar below the plot area
2. Saving the data of the plot as CSV file (comma separated values) by pressing the button  on the right side of the toolbar under the plot

1.2 Accumulation

Sometimes the detected signals intensity is low compared to the noise, and can therefore not be seen. A measure for the quality of the recorded signal is the signal-to-noise ratio (SNR) which is defined as the ratio of signal versus the standard deviation of the noise, i.e. noise strength. The SNR is improved by measuring the signal many times and averaging the recorded signals, also called accumulation. In DAMARIS this can be achieved by creating a loop in the experiment script and accumulating the results in the result script.

Listing 1.3: Experiment script showing how to create a loop

```
1 import Experiment
2
3 def fid_experiment():
4     e=Experiment.Experiment()
5     e.set_frequency(frequency=300.01e6, phase=0)
6     e.ttl_pulse(length=5e-6, channel=1)      # gate for the RF pulse
```

```
7     e.ttl_pulse(length=2e-6, channel=1+2)    # RF pulse
8     e.wait(10e-6)                            # dead-time
9     # record 1024 samples at sampling-rate 2 MHz
10    # with 2V sensitivity
11    e.record(samples=1024, frequency=2e6, sensitivity=2)
12    return e
13
14    def experiment():
15        accumulations=8
16        for run in xrange(accumulations):
17            yield fid_experiment()
```

Listing 1.4: Result script accumulating the data

```
1    import ADC_Result
2    import Accumulation
3
4    def result():
5        # create accumulation object
6        accu = Accumulation.Accumulation()
7        # loop over the incoming results
8        for timesignal in results:
9            # plot the timesignal
10           data["Timesignal"] = timesignal
11           # add timesignal to the object
12           accu += timesignal
13           # provide the data to display tab
14           data["Accumulation"] = accu
```

1.3 Phase Cycling

Artefacts due to channel imbalance in the receiver, inhomogeneity in the RF field and unwanted signals, for example primary echos, can be suppressed or cancelled out by cycling the phases [1] of the receiver and the RF pulses and then accumulating the signal.

Listing 1.5: Experiment script for phase cycling

```
1
2    def fid_experiment(run):
3        e=Experiment()
4        # set a description, this one is used later in the result script
5        e.set_description("run",run)
6        # run%4 is run modulo(4), and thus goes like 0,1,2,3,0,1,2 ...
7        e.set_frequency(frequency=300.01e6, phase=[0,90,180,270][run%4])
8        e.ttl_pulse(length=5e-6, channel=1)    # gate for the RF pulse
9        e.ttl_pulse(length=2e-6, channel=1+2)  # RF pulse
10       e.wait(10e-6)                            # dead-time
11       # change phase, CYCLOPS phase cycling
12       e.set_phase([0,90,180,270][run%4])
13       # record 1024 samples at sampling-rate 2 MHz
14       # with a +/- 2V range
```

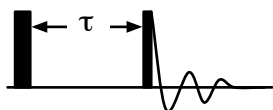
```

15     e.record(samples=1024, frequency=2e6, sensitivity=2)
16     return e
17
18 def experiment():
19     # should be multiple of 4
20     accumulations=8
21     for run in xrange(accumulations):
22         yield fid_experiment(run)

```

New in this script is the **set_description** method and a parameter *run* that is passed to the experiment. The **set_description** method is used to set descriptions and values important to the experimenter or to pass values to the result script. These descriptions are written into the XML job file and stored by the back end into the result file. In this way it is possible to pass values to the result script so the latter can act upon these. In this case the result is either added or subtracted to the accumulation variable.

1.4 Changing Parameters



Now, this script is modified to measure the longitudinal relaxation time T_1 by a so called inversion recovery experiment. Therefore the $\frac{\pi}{2}$ -pulse is preceded by π -pulse both separated by the variable time τ . The FID after the second pulse is recorded.

In order to get a good signal, accumulation and phase cycling are implemented. In the result script, the magnetization will be obtained and, together with the corresponding variable τ , stored in an ASCII file for determination of T_1 .

Listing 1.6: Experiment script illustrating the use of variables

```

1 def fid_experiment(pi, tau, run, accus):
2     e=Experiment()
3     e.wait(9) # repetition time (in s)
4     # set the descriptions, they are used later in the result script
5     e.set_description("run",run)
6     e.set_description("tau",tau)
7     e.set_description("no_accus", accus)
8     # run%4 is run modulo(4), and thus goes like 0,1,2,3,0,1,2 ...
9     e.set_frequency(frequency=300.01e6, phase=[0,180][run%2])
10    e.ttl_pulse(length=5e-6, channel=1) # gate for the RF pulse
11    e.ttl_pulse(length=pi, channel=1+2) # RF pulse
12    e.wait(tau)
13    e.set_phase(phase=[0,0,180,180,90,90,270,270][run%8])
14    e.ttl_pulse(length=5e-6, channel=1) # gate for the RF pulse
15    e.ttl_pulse(length=pi/2, channel=1+2) # RF pulse
16    e.wait(10e-6) # dead-time
17    # change phase, CYCLOPS phase cycling
18    e.set_phase([0,0,180,180,90,90,270,270][run%8])
19    # record 1024 samples at sampling-rate 2 MHz
20    # with a +/- 2V range
21    e.record(samples=1024, frequency=2e6, sensitivity=2)

```

```
22     return e
23
24 def experiment():
25     # should be multiple of 8
26     accumulations=8
27     # here we loop over the tau values starting from 1ms to 10s
28     # in a logarithmic
29     for t in log_range(start=1e-3, stop=10, step_no=20):
30         # this inner loop does the accumulation
31         for r in xrange(accumulations):
32             yield fid_experiment(pi=4e-6, tau=t, run=r, accu=accumulations)
```

Listing 1.7: Result script writing the amplitude and corresponding delay time τ between the pulses to a file

```
1
2 def result():
3     # create accumulation object
4     accu = Accumulation()
5     # loop over the incoming results
6     for timesignal in results:
7         # read in variables
8         tau = float(timesignal.get_description("tau"))
9         run = int(timesignal.get_description("run"))
10        accu = int(timesignal.get_description("no_accu"))-1
11        # provide the single scan to the display tab
12        data["Timesignal"] = timesignal
13        accu += timesignal
14        # provide the accumulated signal to display tab
15        data["Accumulation"] = accu
16        if run == accu:
17            # provide the accumulated signal to display tab;
18            # in this case all accumulated signals will be
19            # available in the data dictionary and thus
20            # saved when the experiment is finished
21            data["Accumulation %i"%run] = accu
22            # open a file in append mode,
23            # file will be created if not existing
24            afile.open('t1.dat','a')
25            # get the maximum on the y channel
26            amplitude = accu.y[0].max()
27            # write tau and amplitude seperated by a <tab> and
28            # ending with a <newline> into the file
29            afile.write("%e\t%e\n"%(tau, amplitude))
30            # close the file
31            afile.close()
32            # create a new accumulation object
33            accu = Accumulation()
```

1.5 Data Handling

To store the result for further data analysis one has several built-in possibilities in DAMARIS. One can store the results as ASCII file or as HDF5³ file. Writing an ASCII file is usable for smaller data-sets and a small number of experiments.

Listing 1.8: Part of a result script for saving data into an ASCII file

```
1 # open file in "w"rite mode
2 nmr_file = open('filename_to_store.dat', 'w')
3 # write data
4 timesignal.write_to_csv(nmr_file)
5 # close the file
6 nmr_file.close()
```

For bigger experiments it is advantageous to store data in a HDF5 file. This is the Hierarchical Data Format developed by the NCSA (National Center for Supercomputer Applications) for storing complex tables and big amounts of data in an effective way. It supports grouping of data, annotation and compression. A very convenient way to access HDF files in Python is the pyTables module, which is also used by DAMARIS internally. Another platform independent way to examine the results is the HDFView Java program provided by NCSA.

After an experiment in DAMARIS has been finished, the results in the data dictionary as well as the experiment and result scripts themselves are written automatically to a HDF file, the data pool file. In the configuration tab (Figure 1.2, 10-12), filename, compression ratio, compression library and write interval can be set up. The write interval is the time after which the data pool file should be periodically rewritten and is useful in case of hardware failure. The results from the ADC are stored as 64-bit floating point values in tables which are compressed/decompressed transparently/on-the-fly. In DAMARIS, three compression libraries are available: zlib, lzo and bz2. For a comparison of these, the pyTables homepage⁴ offers a great wealth of information. HDFView does not support viewing of bzip2 or lzo compressed data. In case the data has been compressed with lzo, the *ptrepack* utility from pyTables can be used to recompress the data with zlib.

The speed-up in post-processing and analysis is already 24 fold in small data-sets with 32768 points per channel and 25 files on a Athlon XP 2600+ CPU with 512 MB RAM running Debian Sarge Linux. Thus, writing HDF5 with compression enabled is highly recommended.

Note that in Listing 1.9 we create a HDF5 file from scratch! Experiment and result script are not saved in the resulting file. The advantage is that one can create arbitrary complex structured files at the cost of having a more complicated result script.

Listing 1.9: Result script for saving data in a HDF5 file

```
1 # import pytables module to
```

³<http://www.hdfgroup.org/HDF5>

⁴<http://pytables.sourceforge.net>

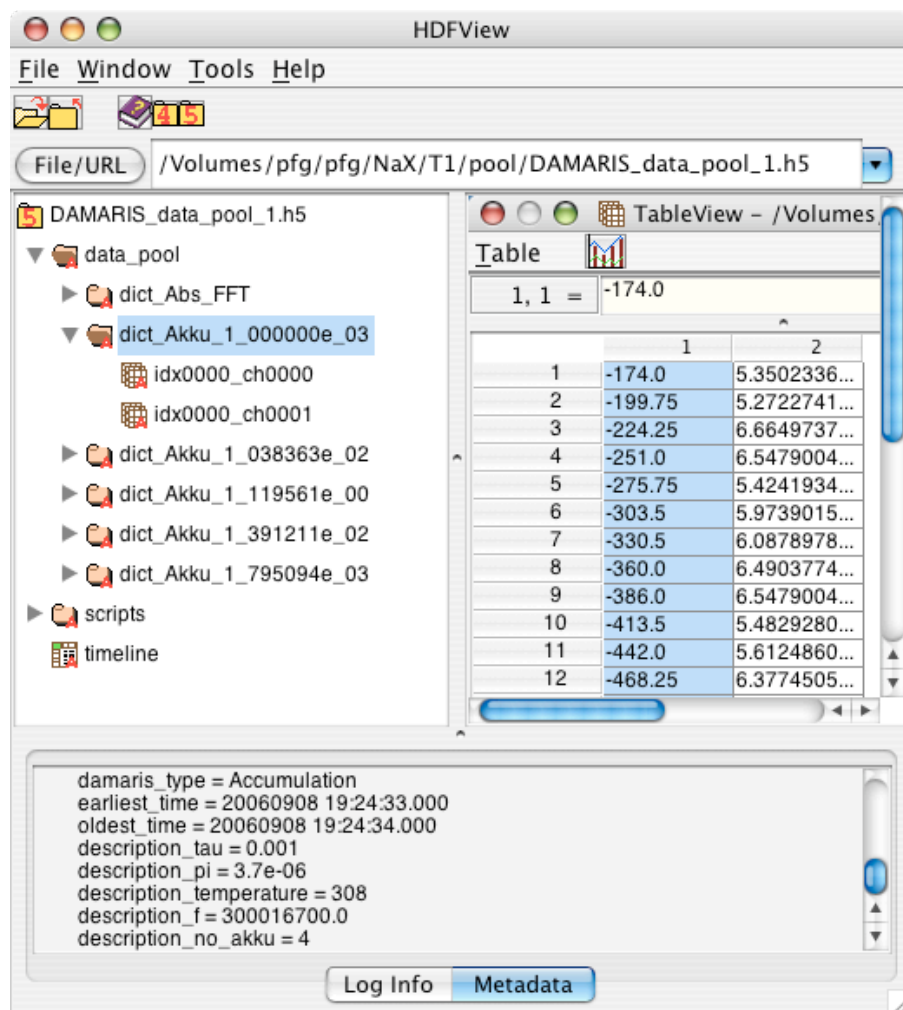


Figure 1.3: A sample data pool file, opened with HDFView. On the bottom of the picture the metadata stored with the table can be seen

```

2 # support the binary HDF file format
3 import tables
4
5 def fid_result():
6     # open file in "w"rite mode to generate a new, empty hdf5 file
7     # this is not absolutely necessary if the file does not already exists
8     nmr_file = tables.openFile('filename_to_store.h5', 'w')
9     # open file in "a"ppend mode
10    nmr_file = tables.openFile('filename_to_store.h5', 'a')
11    # loop over the results
12    for timesignal in results:
13        # plot the timesignal
14        data["Timesignal"]=timesignal
15        # write data
16        timesignal.write_to_hdf(hdffile=nmr_file,
17                               # save it under the root group of the file
18                               where="/",
19                               # name of the group
20                               name="accu",
21                               title="accudata",
22                               # enable lzo compression
23                               level="5",
24                               complib="zlib")
25    # flush the data to the disk
26    nmr_file.flush()
27    # close the file
28    nmr_file.close()
29
30 def result():
31     return fid_result()

```

A very convenient and easier method to save data in HDF is the MeasurementResult class.

```

1 ...
2 def fid():
3     # create a measurement object
4     measurement=MeasurementResult("Magnetization(tau)")
5     for timesignal in results:
6         # extract "tau" from the description dictionary
7         tau = timesignal.get_description('tau')
8         ...
9         # get the magnetization from the timesignal
10        magnetization = sqrt(timesignal.y[0][80:100].mean()*2
11                             + timesignal.y[1][80:100].mean()*2)
12        measurement[tau] += magnetization
13        # provide the magnetization curve
14        data["Magnetization(tau)"] = measurement
15        ...

```

Essentially, a dictionary “measurement” is created with *tau* as the key. For each new key the current magnetization is added. This dictionary is then added to the data pool. The advantage over Listing 1.9 is that the magnetization vs. τ curve can be directly displayed in order to gain an overview of the running experiment, because the “measurement” dictionary is added to the data pool. In this example, the value also stores statistics from the

accumulation. The data dictionary is stored automatically as described before, including the result and experiment scripts, information about the used back end and a timeline of the experiment.

1.6 Fourier Transform Module

A module was written for an easy calculation of spectra from time domain data. The Fourier transformation module “DamarisFFT” is based on the **fft** module of **numpy**, an array module for Python. There are no interactive capabilities such as phase correction incorporated yet. The FFT method can be applied directly to the object (see Listing 1.6). The order in which these methods are applied *does* matter. Here are the available methods and a short explanation of their effects is shown (values in *italic* are default values). The change is *in-place*, which means if the timesignal is needed later on, a copy of it should be made.

clip(start=None,stop=None) Only the data between start and stop is returned. The start and stop parameters can be either in time or frequency domain.

baseline(lastpart=0.1) This method should be applied first as it corrects the baseline of the signal, taking the last **lastpart** part of the signal for determining the correction.

autophase() This will try to automatically phase the spectrum to maximize the real signal. Works only reliably at sufficient high SNR (>20)

fft(samples=None) This method returns the real and the imaginary part of an FFT of the timesignal. Zero-filling of the timesignal can be done via the **points** keyword. If **points** is bigger than the number of data points, the rest will be filled with zeros (zero-filling/zero-paddin

abs_fft(points=None) This method returns an absolute FFT of the timesignal. Zero-filling of the timesignal can be done via the **points** keyword. If **points** is larger than the number of data points, the rest will be filled with zeros (zero-padding). The **zoom** keyword accepts either a tuple with centre frequency and width in Hz. Auto zooming the highest peak can be achieved by setting the centre frequency to *auto*. g).

In order to suppress side-lobes, enhance resolution or improve SNR it is common practice in data processing to apply windowing functions to the data. Several windowing functions are available in the “DaFFT” module. The first two functions, i. e. the exponential and gaussian window function are SNR enhancing windows, while the double exponential window function is resolution enhancing. The TARF window is both, SNR and resolution enhancing[6]. All four are commonly used in NMR spectroscopy. The line broadening factor LB is the rate (in Hz) in which the window function decays. The higher the value,

the faster the function decays, hence the resulting spectrum will be more “smooth” but the peaks will become wider.

exp_window(line_broadening=10) This sets an exponential window over the data, the beginning of the signal is weighted more, where as the tail of the signal gets suppressed. Best results are obtained with line broadening factor set to the peak width:

$$f(x) = \exp(-tp \cdot LB)$$

gauss_window(line_broadening=10) This sets a gaussian window over the data:

$$f(x) = \exp(-(tp \cdot LB)^2)$$

dexp_window(line_broadening=-10, gaussian_multiplier=0.3, show=0) This sets a double-exponential window over the data, which enhances the tail of the FID thus prolonging it and increases the resolution. Note that the line broadening should have a negative value here. The reason is that weighting is supposed to increase from the beginning (exponential part) and then later to decrease again (gaussian part). Another point to mention is that this window is in fact transforming a lorentzian peak in a gaussian peak which has the property of a narrower base:

$$f(x) = \exp(-tp \cdot LB - GM \cdot tp^2)$$

traf_window(line_broadening=10) This sets a TRAF window [6] over the data:

$$f(x) = \frac{\exp(-tp \cdot LB)^2}{[\exp(-tp \cdot LB))^3 + (\exp(-aq_time \cdot LB))^3]}$$

Following standard windowing functions [2] are available additionally. Note that the whole data will be windowed, not only a subset. Furthermore, these windows are not suitable for FID's or similar signals, because they are symmetric around the middle of the data set and zero at the beginning and the end.

- hanning_window()
- hamming_window()
- blackman_window()
- bartlett_window()
- kaiser_window(beta=4, show=0, use_scipy=None)

To use this methods one has to apply consecutively the methods to an *timesignal* or *accumulation* object:

```
1 data["FFT"]=timesignal.baseline(0.4).exp_window(line_broadening=100).abs_fft().clip=(-1e3,1e3)
```

This applies baseline correction using the last 40% of the timesignal for determination of the signal. Then, an exponential window is applied. From this timesignal an FFT is calculated which is further clipped to view to the data between -1 kHz with 1 kHz width.

1.7 Good Practices

- Save often. It can become slow to open a HDF file in append mode and close it again for *each* result, but in case of an error, the file is not completely lost.
- Keep the experiment function simple. Everything necessary for a scan should be contained in a function, the parameters should stay in the **experiment** function. This makes it possible to loop over several parameters easier and, in case of complex experiments, all variables are changed in one place: the call to the function.
- Display if necessary. If the repetition time of subsequent scans is short (several ms) it can slow down the result script notably. The result script will take care of this by not displaying every single shot, but displaying nothing would further speed up the processing of the results.
- Save disk space. If the options “delete results” and “delete jobs” are switched off, the spool directory can get clobbered by several thousand files and in the worst case, one can run out of *inodes*. If it is not possible to delete the files in the directory anymore (rm gives an error “too much arguments”) the following command can help deleting the files anyway:

```
find . -name job\* -exec rm {} \;
```

which will delete all files starting with *job* in the current directory.

- It is a good idea to put a *synchronize()* statement in the experiment script, for example after the 100th accumulation. This will write job files only until the synchronize is requested. Then, the front end will wait writing new job files, until the result script has processed all current results. In big experiments, several ten thousands job of files would be written without this, which can lead to problems with the file system and performance.

```
1 if accu%100 == 0:  
2     synchronize()
```

- Keep an eye on the logs. In case of errors it is highly recommended to check the log file in the log path (as defined in the configuration tab) *as well* as the “Log” tab for error messages.

1.8 Tuning and Development

When new result scripts are developed, it is an advantage if the actual experiment has not to be repeated for every minor change or tweak in the result script. This can become a very tedious task when the sample has a long T_1 relaxation, like water $T_1 \approx 3\text{s}$ [4]. In order to use this technique, the experiment has to be run once with the option “delete

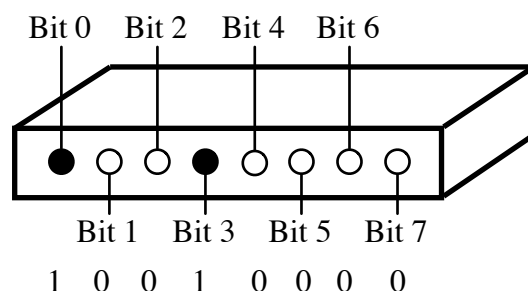


Figure 1.4: Line driver with a decimal value 17 set (or 0x11 hexadecimal). In binary, channel 0 and channel 3 are on

results after processing” turned off. This leads to the result files in the spool directory not being deleted. Starting the experiment again with the option “run back end” switched off, the front end will read the results again like if it would be a new experiment, hence one can test new result scripts quite conveniently. Developing experiment scripts can be done in a similar manner: the dummy back end (to be set in the configuration tab) can be used to test new experiment scripts without the need to have a spectrometer at hand.

1.9 Data Processing Outside DAMARIS

For post-processing of the data, several options are available, depending on the format of the data. Most programs can handle ASCII files (gnuplot, Origin, fitsh, etc.), so this is the more flexible format. HDF can be read by pyTables, Matlab, Octave, R, Mathematica and more⁵.

1.10 List of Commands

Experiment Script Commands

Following is a list of available commands in an experiment script.

ttd_pulse(length, channel=None, value=None) Gives out *either* a TTL pulse with duration **length** on **channel** (counting from zero), *or* multiple channels given by the binary representation of **value** (see Figure 1.4).

Example:

```
1 e.ttd_pulse(length=5e-6, channel=1)
2 e.ttd_pulse(length=3e-6, value=3)
```

⁵<http://hdf.ncsa.uiuc.edu/tools5desc.html>

This example is used to create a RF pulse with a pulse length of $3\ \mu\text{s}$. The first statement sets the gate pulse (channel 0) of the RF amplifier for $5\ \mu\text{s}$, while the second statement combines the gate and RF pulse on channels 0 and 1 ($2^0 + 2^1 = 3$). Hexadecimal representation (number starts with $0x$) is convenient as the numbers are shorter: To set all channels, **value** would be in decimal 16777215 and in hexadecimal $0xffffffff$. One letter in hexadecimal represents four bits.

ttls(length=None, value=None) Same as **ttl_pulse(length, value)**

wait(time) Wait specified **time** in seconds. Minimum time is 90 ns, maximum time is essentially unlimited (years). The limit imposed from the pulse programmer is circumvented by the driver.

Example:

```
1 e.wait(length=2e-3)
```

record(samples, frequency, timelength=None, sensitivity=None) Records data with given number of **samples**, sampling-frequency **frequency** and **sensitivity** in Volts. If **timelength** is given, this state will stay the given time. The maximum sampling-frequency is 20 MHz and the onboard memory can hold 8M samples shared by both channels. Sensitivity can be one of 0.2, 0.5, 1, 2, 5 and 10 V. Not that multiple record statements can be in a single scan (gated sampling) or in a loop.

Example:

```
1 e.record(samples=1024, frequency=2e6, sensitivity=2)
```

Records a signal with 1024 data points and 2 MHz sampling-frequency. The sensitivity will be $\pm 2\ \text{V}$. With a 14bit ADC card this gives a resolution of 0.2 mV.

loop_start(iterations)/loop_end() This creates a loop of given number of **iterations** and has to be closed by **loop_end()**. Commands inside the loop can not change, i.e. the parameters are the same for each loop run. This loop is created on the pulse programmer, thus saving commands.

Example:

```
1 e.loop_start(iterations=10)
2 e.rf_pulse(channel=3, length=2e-6)
3 e.wait(time=2e-6)
4 e.loop_end()
```

This loop will create 10 pulses with $2\ \mu\text{s}$ length and $2\ \mu\text{s}$ apart.

set_frequency(frequency, phase, ttls=0) Sets the **frequency** and **phase** of the frequency source and optionally further channels. The time needed to set the frequency is $2\ \mu\text{s}$.

Example:

```
1 e.set_frequency(frequency=300.01e6, phase=0)
```

The frequency generator will deliver 300.01 MHz.

```
1 e.set_frequency(frequency=300.01e6, phase=0, ttls=3)
```

Same as above, but also sets channel 0 and 1.

set_phase(phase, ttls=0) This command can be used to change the phase of the frequency. The *phase* is given in degree. The *ttls* keyword has the same functionality like in the `set_frequency` command. Setting the phase is done in $0.5\ \mu\text{s}$

Example:

```
1 e.set_phase(phase=90)
2 e.record(samples=1024, frequency=2e6, sensitivity=2)
```

This would set the receiver phase to 90 degree.

set_pts_local() If the frequency source is set to remote control mode, this can be used to set to local mode, thus the frequency of the PTS synthesizer can be changed manually.

set_pfg(length, dac_value, shape=('rec',0), trigger=4, is_seq=0) With this command, pulsed field gradients are generated. In order to create arbitrary ramps or shapes, *is_seq* can be set to 1, leading the DAC to keep the current state until a new strength is applied. The *trigger* keyword allows to set a trigger line with the start of the gradient pulse. Certain gradient shapes are predefined:

- rectangular ('rec')
- sin ('sin')
- \sin^2 ('sin2')

You can select them by giving the *shape* keyword a tuple with shape and resolution in s *Example:*

```
1 e.set_pfg(length=1e-3, dac_value=15040, is_seq=0)
```

This would create a 1 ms long PFG pulse with $1\ \text{Tm}^{-1}$

```
1 for val in lin_range(start=0, stop=pi, step=pi/50):
2     gradient = int( sin(val)**2 * 15040 )
3     e.set_pfg(length=1e-3/50, dac_value=gradient, is_seq=1)
```

This would create a 1 ms long PFG pulse shaped like half period of a \sin^2 with an amplitude of $1\ \text{Tm}^{-1}$.

```
1 e.set_pfg(length=1e-3, dac_value=15040, shape=('sin2', 1e-5))
```

This would create a \sin^2 shaped gradient with 1ms length and 100 interpolation points.

set_description(key, value) Setting a description is accomplished by this command. It creates an entry **key** with value **value** in the description dictionary⁶. In case of the data being stored in a HDF5 file this dictionary is stored as well.

Example:

```
1 e.set_description(key="tau", value=2e-3)
```

Following are convenient functions for use in loops.

lin_range(start, stop, step) This is used for loops. It increases **start** with **step** until **stop** is reached.

Example:

```
1 def experiment():
2     for tp in lin_range(start=5e-7, stop=10e-6, step=5e-7):
3         yield pi_pulse(pulselength=tp)
```

Starting from $0.5 \mu\text{s}$, increase pulse length with $0.5 \mu\text{s}$ until $10 \mu\text{s}$.

log_range(start, stop, stepno) Divide the range **start** to **stop** logarithmically in **stepno** steps.

Example:

```
1 def experiment():
2     for t in log_range(start=5e-3, stop=10, stepno=10):
3         yield inversion_recovery(tau=t)
```

Variate the distance τ of the two pulses in the inversion recovery experiment logarithmically from 5 ms to 10 s. The values would be: ['0.0050', '0.0116', '0.0271', '0.0630', '0.1466', '0.3411', '0.7937', '1.8469', '4.2975', '10.0000']

staggered_range(some_range, size=1) Creates a staggered range from another range, leaving out **size** values in a row and appending the left out later.

Example:

```
1 def experiment():
2     myrange=log_range(start=5e-3, stop=10, stepno=10)
3     for tp in staggered_range(myrange, size=2):
4         yield inversion_recovery(tau=t)
```

This is the same as before, but the values are now staggered: 12345678 will become 12563478. Another possibility for achieving something similar is the *shuffle* function (see Listing 3.1).

interleaved_range(some_range, size=1) Creates an interleaved range from another range, appending every n-th point.

Example:

⁶see `get_description` in result script commands on how to access it

```
1 def experiment():
2     myrange=log_range(start=5e-3, stop=10, stepno=10)
3     for tp in interleaved_range(myrange, size=3):
4         yield inversion_recovery(tau=t)
```

The values are now interleaved: 12345678 will become 14725836. Another possibility for achieving something similar is the *shuffle* function (see Listing 3.1).

combine_ranges(*ranges) With this function it is possible to combine several ranges

Example:

```
1 def experiment():
2     mylinrange=lin_range(start=1e-3, stop=4.9e-3, step=1e-4)
3     mylogrange=log_range(start=5e-3, stop=10, stepno=10)
4     for tp in combine_ranges(mylinrange, mylogrange):
5         yield inversion_recovery(tau=t)
```

Result Script Commands

Following is a list of available commands in a result script. These are methods to be applied to a ADC_Result and Accumulation object. An accumulation object needs to be created before any methods can be applied.

Example:

```
1 akku = Accumulation(error=True)
2 akku += timesignal
```

get_result_by_index(index) If several **record** statements occur in an experiment, the results are saved in an array and can be accessed by this method. *Index* starts with 0.

Example:

```
1 timesignal.get_result_by_index(1)
```

This would return the result of the second record statement.

get_sampling_rate() Returns ADC card sampling rate

uses_statistics() Returns *False* if the result is a single ADC result and *True* if the result is an accumulation with statistics enabled.

write_to_csv(destination=sys.stdout) Writes the result into an ASCII file, destination has to be an open file. The file has to be closed after write_as_csv has been issued otherwise the data is not written to the harddrive.

Example:

```
1 f = open('testfile.dat', 'w')
2 timesignal.write_to_csv(destination=f)
3 f.close()
```

write_to_hdf(hdffile, where, name, title, compress=None) Writes the data to a HDF5 file **hdffile**.

get_job_id

get_description(key) Returns **value** of description **key** as string. Note that in the case of application to an accumulation object, only the common descriptions are stored, leading to a proper set of relevant parameters.

get_xdata() Returns a copy of timedata.

set_xdata(pos, value)

get_ydata(channel) Returns a copy of the data from channel **channel**

set_ydata(channel, pos, value)

get_number_of_channels() Returns the number of ADC channels, usually 2

get_xlabel()/get_ylabel() Returns the x/y-axis description in display register

set_xlabel(label)/set_ylabel(label) Sets the label of the x/y-axis

get_text(index) Returns labels to be plotted (List)

set_text(index, text) Sets labels to be plotted

get_title() Returns the title of the plot

set_title(title) Sets the title of the plot

get_legend() Returns the legend of the plot (dictionary)

set_legend(channel, value) Sets the legend of the plot

get_xmin() Returns minimum of x

get_xmax() Returns maximum of x

get_ymin() Returns minimum of y

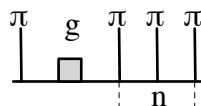
get_ymax() Returns maximum of y

2 DAMARIS for Developers

This chapter is dealing with the internals of DAMARIS. After an introduction of the back end, it is described how the Tecmag DAC20 driver is operating and how it has been implemented into DAMARIS. The complete source code of DAMARIS is available in a subversion¹ source repository for convenient access at <http://www.fkp.physik.tu-darmstadt.de/damaris>.

2.1 The Back End

Starting the back end is causing it to search in the given spool directory for XML job files. These numbered files are containing a sequence composed of states, which themselves contain instructions for the experiment, or sub-sequences (which can contain further states or sub-sequences and can be looped). A state defines the output (24 bits) of the pulse programmer and can be either empty or contain sub-elements (state atoms), which for example could be an instruction to set a channel on the pulse programmer or set the DAC for a PFG current source.



The back end examines this XML job files and traverses the state sequence. Encountering a sub-sequence, this is traversed to its end, where the back end jumps out of the sub-sequence and continues where it entered the sub-sequence. This traversing is called *depth-first*, and causes this nested *state tree* to be processed sequentially in order (Figure 2.1). As an example the pulse sequence π -wait-pfg-wait- π -(wait- π -wait- π)n will be explained in depth. Once this pulse sequence is written in the DAMARIS front end, it results in a XML file (Listing 2.1).

Listing 2.1: "XML job containing a loop"

```
1 <?xml version="1.0" encoding="ISO-8859-1"?>
2 <experiment no="0">
3 <state time="2e-6">
4   <analogout id="0" f="300.01e6" phase="0"/>
5 </state>
6 <state time="5e-6"><ttlout value="0x1"/></state>
7 <state time="4e-6"><ttlout value="0x3"/></state>
8 <state time="5e-3"/>
9 <state time="1e-3">
10  <analogout id="1" dac_value=" 15040"/>
```

¹<http://svnbook.red-bean.com>

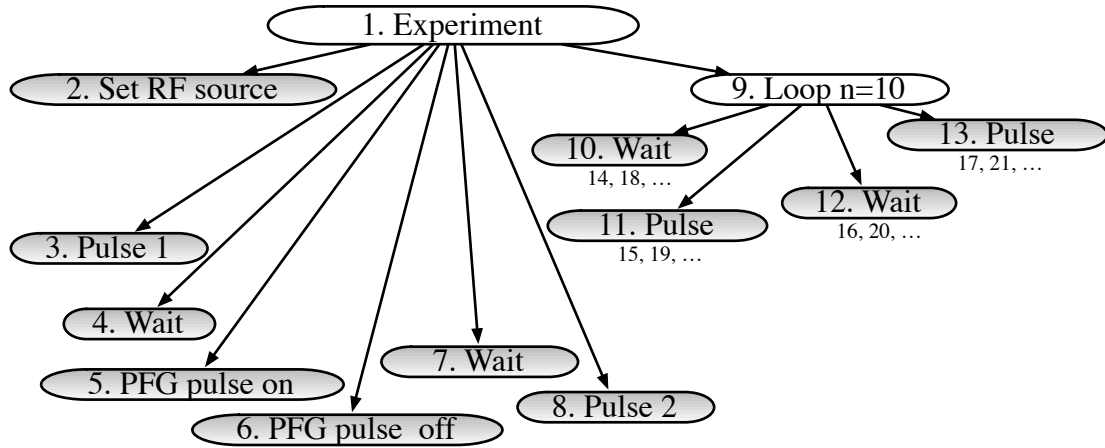


Figure 2.1: The XML file (Listing 2.1) represented as a state tree. Elements are numbered in the order of the traversing. The loop is only traversed once, despite that the elements inside the loop will be executed repeatedly in the experiment.

```

11 </state>
12 <state time="3.8e-6">
13   <analogout id="1" dac_value="0"/>
14 </state>
15 <state time="5e-3"/>
16 <state time="5e-6"><ttlout value="0x1"/></state>
17 <state time="4e-6"><ttlout value="0x3"/></state>
18 <sequent repeat="10">
19   <state time="5e-3"/>
20   <state time="5e-6"><ttlout value="0x1"/></state>
21   <state time="4e-6"><ttlout value="0x3"/></state>
22   <state time="5e-3"/>
23   <state time="5e-6"><ttlout value="0x1"/></state>
24   <state time="4e-6"><ttlout value="0x3"/></state>
25 </sequent>
26 <state time="0.0206848">
27   <analogin s="1024" f="50000" sensitivity="5.0"/>
28 </state>
29 </experiment>

```

Each driver in the back end (PTS synthesizer, PFG, etc.) is traversing this tree, searching for elements it is responsible for. The driver then translates these elements in pure states for the pulse programmer (Figure 2.2), i.e. TTL signals. Then, the next driver is traversing the tree, translating the elements it is responsible for. This is going on until all elements from the tree are translated into states, which are then written in machine code to the pulse programmer and executed. Any result obtained by the back end is written to the corresponding XML result file (job filename + .result).

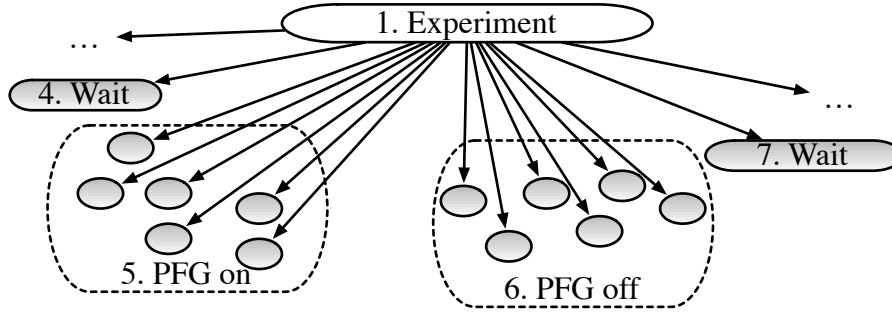


Figure 2.2: Part of the state tree where the element of the state dealing with PFG was translated

2.2 The DAC driver

If in a state an *analogout* element with suitable ID is encountered, the DAC driver extracts the integer value for the gradient strength. The integer value is translated to binary representation (Listing 2.2, lines 133-141), and the single bits are transferred to the DAC (Listing 2.2, lines 160-190). Then, the time of the state is shortened by the time needed to transfer the complete data to the DAC. Every pulse necessary for transferring the data is created by the pulse programmer. Transferring the data to the DAC is achieved by three lines, notably LE (latch enable), CLK (clock signal) and DATA (data line). Reading in a DAC word goes as follows: LE stays high, a data bit is read in with the falling edge of the clock signal, i.e. CLK high to CLK low. After the last (20th) bit is recorded, LE is going low thus signaling the DAC that the word is complete.

Listing 2.2: Main logic of the DAC driver

```

1  if (PFG_aout != NULL) { // Begin state modifications
2      // Check the length of the state
3      if (this_state->length < TIMING*41.0)
4          throw pfg_exception( "time is too short to save DAC information");
5      else {
6          // Copy of original state
7          state* register_state = new state(*this_state);
8          ttlout* register_ttls = new ttlout();
9          register_ttls->id = 0;
10         register_state->length = TIMING;
11         register_state->push_back(register_ttls);
12         // Return error if dac_value is out of bounds
13         if (PFG_aout->dac_value > (pow(2.0, int(DAC_BIT_DEPTH-1))-1) )
14             throw pfg_exception("dac_value too high");
15         if ( abs(PFG_aout->dac_value) > pow(2.0, int(DAC_BIT_DEPTH-1)) )
16             throw pfg_exception("dac_value too low");
17         // Now, create the bit pattern
18         vector<int> dac_word;
19         for (int j = 0; j < DAC_BIT_DEPTH ; j++) {
20             int bit = PFG_aout->dac_value & 1;

```

```
21     dac_word.push_back(bit);
22     PFG_aout->dac_value >>= 1;
23 }
24 // Reverse the bit pattern
25 reverse(dac_word.begin(), dac_word.end());
26 /*
27     Datasheet AD1862:
28     - Bit is read with rising edge of the CLK
29     - Word is read with falling edge of LE
30     The opto-coupler in the DAC20 are inverting the signal!
31     CLK is here inverted, the rest not. This does not affect functionality
32     because the inverse of a 2s complement is then just negative - 1:
33     Example:
34         5 bits:
35         15 --> 01111 -->inverting--> 10000 = -16
36         1 --> 00001 -->inverting--> 11110 = -2
37         0 --> 00000 -->inverting--> 11111 = -1
38         -1 --> 11111 -->inverting--> 00000 = 0
39         -2 --> 11110 -->inverting--> 00001 = 1
40         -16 --> 10000 -->inverting--> 01111 = 15
41     Latch enable is going high after 41st bit.
42 */
43
44 // Transfer the bit pattern
45 for (int i = 0; i < DAC_BIT_DEPTH; i++) {
46     // Two states = one clock cycle to read in bit
47     // State 1
48     register_ttls->ttls=(1<<DATA_BIT)*dac_word[i]+(1<<CLK_BIT)+(1<<LE_BIT);
49     the_sequence.insert(the_state,register_state->copy_new());
50     // State 2
51     register_ttls->ttls = (1<<DATA_BIT)*dac_word[i]+(1<<LE_BIT);
52     the_sequence.insert(the_state,register_state->copy_new());
53     if (i == (DAC_BIT_DEPTH-1)) {
54         // Last bit => LE -> 0, prepare DAC to read the word in
55         register_ttls->ttls = 0;
56         the_sequence.insert(the_state,register_state->copy_new());
57     }
58 }
59
60 /*
61     Shorten the remaining state
62     and add LE high to this state.
63     The word is read in.
64 */
65 ttlout* ttls=new ttlout();
66 // 42nd pulse
67 this_state->length -= TIMING*41;
68 // Here is the word read in
69 ttls->ttls = 1 << LE_BIT;
70 this_state->push_front(ttls);
71 delete register_state;
72 delete PFG_aout;
73
```

```
74     }  
75 }
```

This procedure was first tested and further refined in Python. Embedding the driver properly in the DAMARIS back end made it necessary to translate the test program to C++ [5] code. The following files of the back end needed to be modified or created:

- drivers/pfggen.h
- drivers/Tecmag-DAC20/DAC20.cpp
- drivers/Tecmag-DAC20/DAC20.h

These three files are the main part of the driver. They contain the main logic for the DAC serial line.

- machines/hardware.cpp
- machines/hardware.h
- machines/PFGcore.cpp

Finally, these last files contain the spectrometer setup information, i.e. which frequency synthesizer, ADC card, pulse programmer or temperature controller is used. The resulting PFGcore.exe is the back end which is executed by the front end and has to be specified in the configuration tab.

Listing 2.3: PFGcore.cpp which defines the spectrometer hardware

```
1  #include "machines/hardware.h"  
2  #include "core/core.h"  
3  #include "drivers/Tecmag-DAC20/DAC20.h"  
4  #include "drivers/PTS-Synthesizer/PTS.h"  
5  #include "drivers/Spectrum-MI40xxSeries/Spectrum-MI40xxSeries.h"  
6  #include "drivers/SpinCore-PulseBlaster24Bit/SpinCore-PulseBlaster24Bit.h"  
7  
8  /*  
9      line 0 for gate  
10     line 1 for pulse  
11     line 22 for trigger  
12     line 3 free  
13  */  
14  class PFG_hardware: public hardware {  
15  
16  SpinCorePulseBlaster24Bit* my_pulseblaster;  
17  SpectrumMI40xxSeries* my_adc;  
18  
19  public:  
20  PFG_hardware(){  
21      ttlout trigger;  
22      trigger.id=0;
```

```

23     trigger.ttls=0x400000; /* line 22 */
24     my_adc=new SpectrumMI40xxSeries(trigger);
25     my_pulseblaster=new SpinCorePulseBlaster24Bit(0,1e8,0x800000);
26     PTS* my_pts=new PTS_latched(0); // ID of PTS_analogue is 0
27     the_fg=my_pts;
28     the_pg=my_pulseblaster;
29     the_adc=my_adc;
30     PFG* my_pfg=new PFG(1); // ID of PFG DAC is 1
31     the_gradientpg=my_pfg;
32 }
33
34 result* PFG_hardware::experiment(const state& exp) {
35     result* r=NULL;
36     for(size_t tries=0; r=NULL && core::term_signal==0 && tries<102; ++tries) {
37         state* work_copy=exp.copy_flat();
38         if (work_copy==NULL)
39             return new error_result(1,"could create work copy of experiment sequence");
40         try {
41             if (the_fg!=NULL)
42                 the_fg->set_frequency(*work_copy);
43             if (the_adc!=NULL)
44                 the_adc->set_daq(*work_copy);
45             if (the_gradientpg!=NULL)
46                 the_gradientpg->set_dac(*work_copy);
47             // the pulse generator is necessary
48             // synchronizing with sample clock
49             my_pulseblaster->run_pulse_program_w_sync(*work_copy,
50                 my_adc->get_sample_clock_frequency());
51             // wait for pulse generator
52             the_pg->wait_till_end();
53             // after that, the result must be available
54             if (the_adc!=NULL)
55                 r=the_adc->get_samples();
56             else
57                 r=new adc_result(1,0,NULL);
58             }
59             // catching errors
60             catch (frequ_exception e) {
61                 r=new error_result(1,"frequ_exception: "+e);
62             }
63             catch (ADC_exception e) {
64                 if (e!="ran into timeout!" || tries>=100)
65                     r=new error_result(1,"ADC_exception: "+e);
66             }
67             catch (pulse_exception p) {
68                 r=new error_result(1,"pulse_exception: "+p);
69             }
70             delete work_copy;
71             if (core::quit_signal!=0) break;
72         }
73         // finnish hardware access
74         return r;
75     }

```

```

76
77 virtual ~PFG_hardware() {
78     if (the_adc!=NULL) delete the_adc;
79     if (the_pg!=NULL) delete the_pg;
80     if (the_gradientpg!=NULL) delete the_gradientpg;
81     if (the_fg!=NULL) delete the_fg;
82 }
83
84 };
85
86 // PFG core class is subclassed from the main core class
87 // this assigns the proper hardware to a core, resulting
88 // in a a backend
89 class PFG_core: public core {
90     std::string the_name;
91 public:
92     PFG_core(const core_config& conf): core(conf) {
93         the_hardware=new PFG_hardware();
94         the_name="PFG core";
95     }
96     virtual const std::string& core_name() const {
97         return the_name;
98     }
99 };
100
101 int main(int argc, const char** argv) {
102     int return_result=0;
103     try {
104         core_config my_conf(argv, argc);
105         // setup input and output
106         PFG_core my_core(my_conf);
107         // start core application
108         my_core.run();
109     }
110     // error checking
111     catch(ADC_exception ae) {
112         fprintf(stderr,"adc: %s\n",ae.c_str());
113         return_result=1;
114     }
115     catch(core_exception ce) {
116         fprintf(stderr,"core: %s\n",ce.c_str());
117         return_result=1;
118     }
119     catch(pulse_exception pe) {
120         fprintf(stderr,"pulse: %s\n",pe.c_str());
121         return_result=1;
122     }
123     catch(pfg_exception pfge) {
124         fprintf(stderr,"pfg: %s\n",pfge.c_str());
125         return_result=1;
126     }
127
128     return return_result;

```

129 }

In the front end the following file needed to be modified to be able to use the PFG conveniently:

- Experiment.py

This file contains the functions to create a pulse sequence. The command **set_pfg**, which adds the proper XML element to the state tree, has been added.

```

1 ...
2
3 def set_pfg(self, I_out=None, dac_value=None, length=None, is_seq=0):
4     """This sets the value for the PFG, it also sets it back automatically.
5     If you don't wish to do so (i.e. line shapes) set is_seq=1"""
6     if I_out != None:
7         print "I_out is deprecated"
8     if I_out == None and dac_value == None:
9         dac_value=0
10    if I_out != None and dac_value == None:
11        dac_value=dac.conv(I_out)
12    if I_out == None and dac_value != None:
13        dac_value=dac_value
14    if I_out !=None and dac_value != None:
15        dac_value = 0
16        print "WARNING: You can't set both, I_out and dac_value! dac_value set to 0"
17    if length == None:
18        # minimum length
19        length=42*9e-8
20    self.state_list.append('<state time="%s">
21                            <analogout id="1" dac_value="%i"/>
22                            </state>\n' %(repr(length), dac_value))
23    if is_seq == 0:
24        # Set the DAC back to zero if this is not part of a sequence
25        self.state_list.append('<state time="%s">
26                                <analogout id="1" dac_value="0"/>
27                                </state>\n' %(repr(42*9e-8)) )
28 ...

```

The conversion factor is 50 AV^{-1} . The DAC receives the data from the pulse card after the signals are brought into rectangular shape by the cable driver. The bipolar signal is digitally encoded in 2s-complement, i.e. one bit represents the sign of the integer, thus the range of integer values starts from -2^{19} to $2^{19} - 1$. The DAC outputs a voltage according to the DAC word until the DAC receives another value. This behavior is important to the experimenter because if one wants to set a value only for a certain time it is necessary to set the value back to zero. In the DAMARIS front end this is done automatically with **set_pfg(dac_value, length)** whereas in the back end it is done at the beginning of each experiment to protect the hardware and to set the DAC into an defined state. Arbitrary shaping of gradient pulses is also possible by using the command **set_pfg(dac_value, length, isseq=1)** which prevents setting the DAC to zero after the given length. Through

concatenating subsequently `set_pfg(dac_value, length, iseq=1)` commands one can create almost any possible shape and/or background gradients. Shaping the gradients was not the focus of this work but the general function was tested using an oscilloscope at the current monitor of the PFG current source with the DAC creating a sine wave using several resolutions (Figure 2.3).

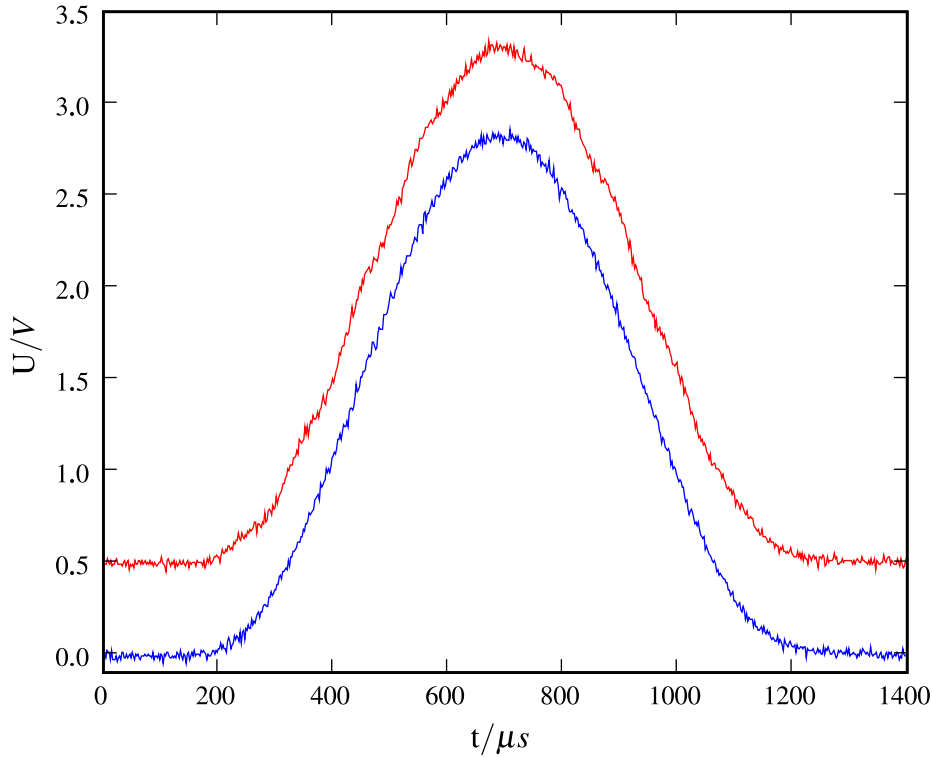


Figure 2.3: A \sin^2 PFG pulse with 1 ms length. The upper one with $100 \mu\text{s}$, the lower one with $3.78 \mu\text{s}$ resolution

Theoretical resolution is given by approximately $280\text{A}/2^{19}=0.000534\text{ A}$. Transferring in a 20bit word needs 21 cycles where each clock cycle is 180 ns long. This leads to a time resolution of $3.78 \mu\text{s}$. Note that each DAC setting needs 42 instructions which can lead to problems regarding the memory of the PulseBlaster in the case of gradient pulse shaping. In Figure 2.3, the lower gradient, the maximum time resolution of $3.78 \mu\text{s}$ was used, there are 264 steps, each of them with 42 states leading to more than 11.000 instructions for this pulsed field gradient alone.

An improvement would be the use of loops for repeating patterns in the DAC word. The easiest approach would be counting consecutively ones and zeros and write a loop if a one or a zero is repeated, i.e. so called *run-length compression*. Even better would be searching for general patterns and use the best (in respect to memory, the smallest)

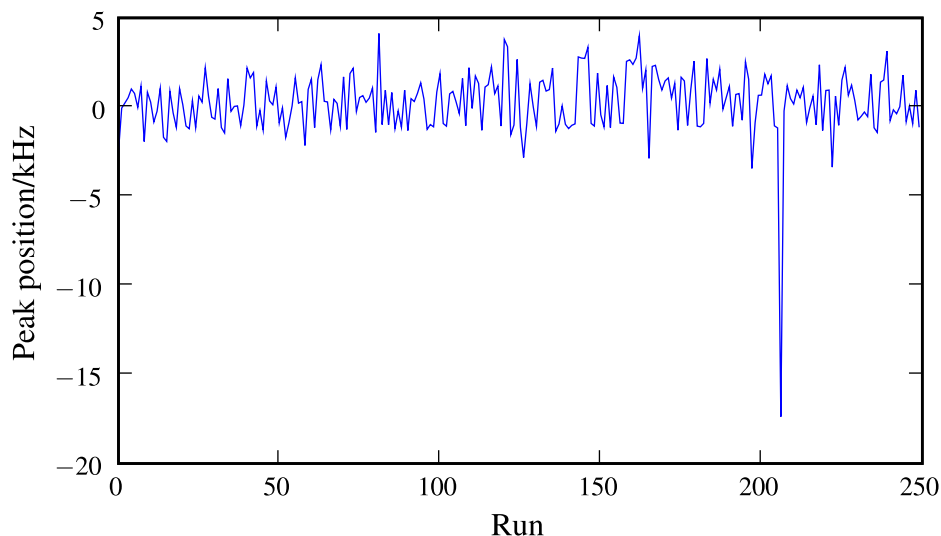


Figure 2.4: The DAC is controlling the B_0 field and the peak position of the FFT signal has been recorded. For each signal the DAC is newly set to the same value. The time delay between each scan is 50 s

instruction set. Another way to save instructions is to save the shape as a subprogram on the pulse programmers memory and recall it when it is needed. This possibility is not yet implement in the pulse programmer driver.

Stability of the DAC has been tested by measuring the shifts of resonance frequency in a Field Cycling spectrometer where the DAC controlled the magnetic field (Figure 2.4). The DAC was very stable and the standard deviation of the frequency shift was about 3 kHz and no drifts were visible except for one outlier which is good compared to the usual 6 kHz² with the standard field cycling DAC. Further stability measurements using an Agilent 3440 A high resolution multimeter have been also preformed. These measurements lead to the conclusion that the DAC is sufficiently stable and temperature drifts do not occur.

The offset of the PFG amplifier was adjusted via the offset potentiometer on the DAC board. There is also an offset adjustment possible at the PFG amplifier directly. For adjustment, the signal of water in a sample tube with a 0.5 mm teflon stripe perpendicular to the magnetic field B_0 was recorded. First, the PFG power supply has been switched off and the sample was shimmed with the room temperature shims until the longest decay time T_2^* of the FID has been achieved. After this, the PFG was switched on and the offset has been adjusted to cause least distortion in the spectrum.

²Value in the laboratoy book

3 Scripts for Various Experiments

Listing 3.1: Experiment script for measuring diffusion in NaX with 13 interval pulse sequence

```
1 import Experiment
2 from pylab import arange, log, exp, polyval, load, concatenate, array
3 from random import shuffle
4 import pickle
5
6 def gradient(t_pfg, grad):
7     """
8     Calculate dac_value
9     """
10    dac_value = (grad)/6.649e-5
11    return int(dac_value)
12
13 def rf(exp, length):
14     """
15     Short form for making a RF pulse
16     """
17    exp.ttl_pulse(5e-6, value=1)    # gate 2**0
18    exp.ttl_pulse(length, value=3)  # gate+rf 2**0+2**1
19
20 def pfg_exp(frequency,      # Frequency
21            f0,              # Resonance frequency
22            pi,              # Pi pulse length
23            pi_h,            # Pi/2 pulse length
24            grad_G,          # Gradient stength
25            grad_F,          # Gradient strength
26            t_pfg,           # Gradient pulse length
27            read_gradient,    # Read gradient strength
28            delta1,          # Time before gradient
29            delta2,          # Time after gradient
30            akku,            # Current accumulation
31            Delta,           # Time Delta between RF pulses 3 and 4
32            no_aku,          # Total number of accumulation
33            t_corr,          # Correction time length of the read gradient in prep interval
34            sample_temp,     # Sample temperature
35            pfg_rise_time,    # Rise time of the gradients, needed to center the gradients
36            t1,              # T1 longitudinal relaxation time
37            t2,              # T2 transversal relaxation time
38            use_cor,         # Flag if correction should be used or not
39            the_index,        # (next two not used)
40            t_pfg_index):
41     """
```

3 Scripts for Various Experiments

```
42 The PFG pulse sequence
43 """
44
45 e=Experiment.Experiment()
46 e.wait(7*t1) # Repetition time
47 # Descriptions
48 e.set_description("type","13interval")
49 e.set_description("no_akku", no_akku)
50 e.set_description("pi",pi)
51 e.set_description("f",frequency)
52 e.set_description("f0",f0)
53 e.set_description("t_pfg",t_pfg)
54 e.set_description("Akku", akku)
55 e.set_description("tau",delta1+delta2+t_pfg)
56 e.set_description('delta1', delta1)
57 e.set_description('delta2', delta2)
58 e.set_description("Delta", Delta)
59 e.set_description("SampleTemp", sample_temp)
60 e.set_description("G", grad_G)
61 e.set_description("F", grad_F)
62 e.set_description("t1",t1)
63 e.set_description("index",the_index)
64 e.set_description("t_pfg_index",t_pfg_index)
65 e.set_description("t2",t2)
66 e.set_description("use_cor",use_cor)
67
68 # Phase cycling from J. Phys. Chem. B, 105 (25), 5922 -5927, 2001. 10.1021
69 phases = {'ph1': [270,270,90,90,0,0,180,180,90,90,270,270,180,180,90,90][akku%16],
70          'ph2': [270,270,90,90,0,0,180,180,90,90,270,270,180,180,90,90][akku%16],
71          'ph3': [270,270,90,90,0,0,180,180,90,90,270,270,180,180,90,90][akku%16],
72          'ph4': [270,270,90,90,0,0,180,180,90,90,270,270,180,180,90,90][akku%16],
73          'ph5': [0,180,90,270,90,270,180,0,180,0,270,90,270,90,0,180][akku%16],
74          'rec_ph': [0,0,0,0,90,90,90,90,180,180,180,180,270,270,270,270][akku%16]}
75
76 # 1. pi/2
77 ph1 = phases['ph1']
78 # 1. pi
79 ph2 = phases['ph2']
80 # 2. pi/2
81 ph3 = phases['ph3']
82 # 3. pi/2
83 ph4 = phases['ph4']
84 # 2. pi
85 ph5 = phases['ph5']
86 # Receiver
87 rec_ph = phases['rec_ph']
88
89 # Create the dac_values for the four gradients
90
91 pos_G = gradient(t_pfg,grad_G)
92 pos_F = gradient(t_pfg,grad_F)
93 neg_G = -pos_G
94 neg_F = -pos_F
```

```

95
96 # This is necessary for the result script
97 # to identify a zero gradient signal
98 if grad_G == 0.0:
99     e.set_description("dac_value",0)
100 else:
101     e.set_description("dac_value",pos_G)
102
103 # Create zero gradient and read_gradient dac_values
104
105 zero_grad = gradient(2e-3, 0.0)
106
107 if (read_gradient != None):          # Did we set a read gradient ?
108     if use_cor:                      # Should we do correction ?
109         key = (pos_G, t_pfg_index, the_index) # Generate key for the dictionary
110         try:
111             CORR_file = open('CORR')
112             corr_data = pickle.load(CORR_file) # Load correction data dictionary
113             CORR_file.close()
114             # t_corr is decremented by the correction time found in dictionary
115             t_corr -= corr_data[key]
116         except:
117             keys_tr = str(key[0]) + "_" + str(key[1]) + "_" + str(key[2])
118             print "No t_corr found: %s"%keys_tr
119     else:
120         pass
121 if t_corr < 0:
122     # If t_corr becomes negative, reverse the sign of the correction gradient
123     t_corr = abs(t_corr)
124     small_const_g = gradient(2e-3, -read_gradient)
125 else:
126     neg_t_corr = False
127     small_const_g = gradient(2e-3, read_gradient)
128
129     if abs(t_corr) < 3.78e-6: # Shortest possible pfg length
130         t_corr = 3.78e-6
131
132
133     if abs(t_corr) > delta2:
134         # If t_corr bigger than the time between PFG and RF pulse print warning
135         print "t_corr too long, shortening it ..."
136         t_corr = delta2
137     if grad_G == 0.0:
138         # Set no correction gradient at all if gradient is set to 0 T/m
139         small_const_g = None
140
141 else:
142     # Without read gradient don't set any
143     small_const_g = None
144
145
146 ## Preparation part
147

```

3 Scripts for Various Experiments

```
148 e.set_pfg(dac_value=zero_grad,is_seq=1) # Zero gradient
149 e.set_frequency(frequency, ph1) # Needs 2 microseconds
150 ##### Pulse 1
151 rf(e,pi_h)
152 e.wait(delta1 - pfg_rise_time)
153 ## F
154 e.set_pfg(dac_value=pos_F, length=t_pfg, is_seq=1)
155 e.set_pfg(dac_value=zero_grad, is_seq=1) # Set to zero
156 e.wait(delta2 - 2e-6 + pfg_rise_time)
157 e.set_phase(ph2)
158 ## Pulse 2
159 rf(e,pi)
160 e.wait(delta1 - pfg_rise_time)
161 ## G
162 e.set_pfg(dac_value=pos_G,length=t_pfg, is_seq=1)
163 if small_const_g:
164     # The small read gradient
165     e.set_pfg(dac_value=small_const_g, length=t_corr, is_seq=1)
166     # Now go back to zero
167     e.set_pfg(dac_value=zero_grad, is_seq=1)
168     e.wait(delta2 - 2e-6 + pfg_rise_time - t_corr)
169 else:
170     e.set_pfg(dac_value=zero_grad, is_seq=1) # Set to zero
171     e.wait(delta2 - 2e-6 + pfg_rise_time)
172
173 ## Evolution part
174
175 e.set_phase(ph3)
176 ##### Pulse 3
177 rf(e,pi_h)
178 e.wait(Delta-2e-6) # Evolution time
179
180 ## Refocussing part
181
182 e.set_phase(ph4)
183 ##### Pulse 4
184 rf(e,pi_h)
185 e.wait(delta1 - pfg_rise_time)
186 ## -G
187 e.set_pfg(dac_value=neg_G, length=t_pfg, is_seq=1)
188 # now go back to zero
189 e.set_pfg(dac_value=zero_grad, is_seq=1)
190 e.wait(delta2 - 2e-6 + pfg_rise_time)
191 e.set_phase(ph5)
192 ## Pulse 5
193 rf(e,pi)
194 e.wait(delta1 - pfg_rise_time)
195 ## -F
196 e.set_pfg(dac_value=neg_F,length=t_pfg)
197 e.set_pfg(dac_value=zero_grad, is_seq=1)
198 e.wait(delta2 - 2e-6 + pfg_rise_time)
199
200
```

```

201     ## Recording part
202     # Adjust the receiver phase to get a real signal on the real channel
203     e.set_phase(rec_ph-45)
204     if small_const_g:
205         # The small read_gradient while recording
206         e.set_pfg(dac_value=small_const_g, is_seq=1)
207     e.record(4*1024, 1e6, sensitivity=2)
208     e.set_pfg(dac_value=zero_grad, is_seq=0)
209     # Synchronize Experiment script with the Result script
210     if akku+1 == no_aku:
211         synchronize()
212     return e
213
214
215
216 def experiment():
217     """
218     Conduct the experiment
219     """
220     # Gradients to measure
221     gradient_list = arange(0, 18, 1)
222     # Gradient lengths to be measured
223     t_pfg_list = arange(0.2e-3, 1.6e-3, 0.2e-3)
224     # Time before the gradient pulses
225     delta1 = 0.5e-3
226     # How many values for the evolution time
227     points = 25
228
229     no_aku = 2      # Number of accumulations for the data
230     no_aku_corr = 6 # Number of accumulations for corrected signal
231
232     # calculate approximate duration
233     duration = len(gradient_list)*len(t_pfg_list)*points*(no_aku
234                                                         + no_aku_corr)*7*load('t1')[0]/3600.0
235     print "Duration will be approx. ", duration, "hours"
236     if duration > 48:
237         print "Warning: load nitrogen"
238
239     # Start value for the evolution time
240     start = 5.5e-3 #3.5e-3
241     # End value for the evolution time
242     end = 160e-3
243
244     # Logarithmic distribution of the evolution times
245     tp_list = [exp(i) for i in arange(log(start), log(end), log(end/start)/points)]
246
247     # Create the list of parameters, each entry in the list has 3 values:
248     # gradient strength, Delta (tp) and gradient length
249     #
250     zero_grad_list = [(grad, tp, t_pfg) for grad in [0]
251                      for tp in tp_list
252                      for t_pfg in t_pfg_list]
253

```

3 Scripts for Various Experiments

```
254     rest_grad_list = [(grad, tp, t_pfg) for grad in gradient_list
255                        for tp in tp_list
256                        for t_pfg in t_pfg_list]
257     # Randomize the data aquisition, so any drifts are
258     # statistically distributed/obscured
259     shuffle(zero_grad_list)
260     shuffle(rest_grad_list)
261     # but do the zero gradients first
262     parameter_list = concatenate((zero_grad_list, rest_grad_list))
263
264     # Bookmarking several variables
265     t_dict={}
266     for i,t in enumerate(tp_list):
267         t_dict[t]=i
268
269     t_pfg_dict = {}
270     for i,t in enumerate(t_pfg_list):
271         t_pfg_dict[t]=i
272
273     # First measure without correction, then correct
274     for correction in [False, True]:
275         if correction:
276             akkus = no_akku
277             print "with correction"
278         else:
279             # Do more accumulations for corrected signal
280             akkus = no_akku_corr
281             print "determinig correction"
282
283     for tpl in parameter_list:
284         print tpl
285         grad = tpl[0]
286         tp    = tpl[1]
287         t_pfg= tpl[2]
288         d1 = delta1
289         for akku in range(akkus): # Accumulations
290             yield pfg_exp(
291                 pi = 3.8e-6,
292                 pi_h = 1.9e-6,
293                 frequency = load('freq0')[0]+0e4,
294                 f0 = load('freq0')[0],
295                 grad_G = grad,
296                 grad_F = grad/(8.0/(1+1.0/3.0*(t_pfg/(d1*2+t_pfg))**2)-1),
297                 t_pfg = t_pfg,
298                 t_corr = 0.2e-3,
299                 pfg_rise_time= 120e-6,
300                 read_gradient= 0.03,
301                 delta1 = d1, # Time before Ggradient
302                 delta2 = d1, # Time before Ggradient
303                 akku = akku,
304                 Delta = tp, # Evolution time
305                 no_akku = akkus,
306                 sample_temp = 263.3,
```

```
307         t1          = load('t1')[0], # Load T1 from file
308         t2          = load('t2')[0], # Load T2 from file
309         use_cor      = correction,
310         the_index    = t_dict[tp],
311         t_pfg_index  = t_pfg_dict[t_pfg])
```

Listing 3.2: Result script for measuring diffusion in NaX with 13 interval pulse sequence

```
1 import ADC_Result
2 import Accumulation
3 import DaFFT
4 import tables
5 import time
6 from numpy import correlate, sqrt
7 import os,cPickle
8
9 def check_filename(filename):
10     """
11     Convenience function to get numbered filenames
12     """
13     i=0
14     stop = False
15     filename = "%s_%03i.h5"%(filename,i)
16     while not stop:
17         if os.path.isfile(filename):
18             i += 1
19             filename = "%s_%03i.h5"%(filename[:-7],i)
20         else:
21             stop = True
22     return filename
23
24 def result_it():
25     start_time = time.time()
26     dacs = {}
27     tps = {}
28     tpfgs = {}
29
30     var = 100
31     fft = False
32     for timesignal in results:
33         if not isinstance(timesignal, ADC_Result.ADC_Result):
34             print "No result: ",timesignal
35             continue
36
37         # get some information
38         sr = timesignal.get_sampling_rate()
39         al = len(timesignal.x)
40         data["Timesignal"]=timesignal
41         if fft:
42             fft = DaFFT.FFT(timesignal)
43             fftdata = fft.base_corr(0.1).fft()
44             data["FFT"] = fftdata
45
46         ## Get the descriptions
47         dac_value = int(timesignal.get_description("dac_value"))
48         t_pfg = float(timesignal.get_description("t_pfg"))
49         akku_number = int(timesignal.get_description("Akku"))+1
50         akkus = int(timesignal.get_description("no_aku"))
51         tau = float(timesignal.get_description("tau"))
52         Delta = float(timesignal.get_description("Delta"))
```

```

53     use_cor = (timesignal.get_description("use_cor"))
54     index = int(timesignal.get_description("index"))
55     delta1 = float(timesignal.get_description("delta1"))
56     grad_G = float(timesignal.get_description("G"))
57     t_pfg_index = int(timesignal.get_description("t_pfg_index"))
58     t_pfg_int = int(t_pfg*1e4)
59     d1_int = int(delta1*1e4)
60     # easter egg :-)
61     if use_cor == 'False':
62         use_cor = False
63     else:
64         use_cor = True
65
66     # Check if the dac_value has changed
67     try:
68         if not dac_old == dac_value:
69             dac_old = dac_value
70             # this variable is for naming the groups in the HDF file
71             var = 100
72     except:
73         dac_old = dac_value
74
75     # Check for a measurement without gradients
76     if dac_value == 0:
77         zero_grad = True
78     else:
79         zero_grad = False
80     dac_int = dac_value
81     # Conversion of numbers to strings which play nicely with pyTables
82     if dac_value < 0:
83         dac_value = "neg_%s"%(str("%07i"%dac_value)[1:])
84     else:
85         dac_value = "pos_%s"%(str("%07i"%dac_value)[1:])
86
87     if akku_number <= akkus:
88         # Initializing the accumulation variable
89         if akku_number == 1:
90             timesignal_akku = Accumulation.Accumulation(error=True)
91             if fft:
92                 fft_akku = Accumulation.Accumulation(error=True)
93         # accumulate
94         timesignal_akku += timesignal
95         if fft:
96             fft_akku += fftdata
97
98         # Plot current accumulation
99         data["Akku"]=timesignal_akku
100
101         if fft:
102             data["FFT-Akku"]=fft_akku
103         # Accumulation is finished
104         if akku_number == akkus:
105             if use_cor: # correction was applied, hence save the result

```

3 Scripts for Various Experiments

```
106     # Open a file in "a"ppend mode
107     h5file = tables.openFile(fn, mode = "a")
108
109     try:
110         h5file.root.raw_data._g_checkHasChild("g_tpfg%03i_delta%03i_%s"
111                                             %(t_pfg_int,d1_int,dac_value))
112     except:
113         h5file.createGroup("/raw_data", "g_tpfg%03i_delta%03i_%s"
114                             %(t_pfg_int,d1_int,dac_value), 'the data')
115         print "data group %s created ... "%dac_value
116
117     if fft:
118         try:
119             h5file.root.fft_data._g_checkHasChild("g_tpfg%03i_delta%03i_%s"
120                                                    %(t_pfg_int,d1_int,dac_value))
121         except:
122             h5file.createGroup("/fft_data", "g_tpfg%03i_delta%03i_%s"
123                                     %(t_pfg_int,d1_int,dac_value), 'fftDaten')
124             print "fft data group %s created ... "%dac_value
125
126
127     timesignal_akku.write_to_hdf(h5file,
128                                where="/raw_data/g_tpfg%03i_delta%03i_%s"%(t_pfg_int,
129                                                                            d1_int,dac_value),
130                                name="data_%04i_delta%04i"%(index,d1_int),
131                                title="Data", compress=1)
132
133     if fft:
134         fft_akku.write_to_hdf(h5file,
135                               where="/fft_data/g_tpfg%03i_delta%03i_%s"%(t_pfg_int,
136                                                                             d1_int,dac_value),
137                               name="data_%04i_delta%04i"%(index,d1_int),
138                               title="FFTDData", compress=1)
139     h5file.flush()
140     h5file.close()
141
142     # This is the FIRST run, and the zero gradient data is saved in signal_0
143     # and the corection file is created
144     thekey = (dac_int, t_pfg_index, index)
145     if zero_grad and not use_cor:
146         signal_0 = sqrt(timesignal_akku.y[0]**2 + timesignal_akku.y[1]**2)
147         correctionfile = open('CORR','w')
148         t_corr = 0.0
149         dacs[thekey] = t_corr
150         cPickle.dump(dacs,correctionfile)
151         #correctionfile.write("%i\t%e\n"%(dac_int,t_corr))
152         correctionfile.close()
153     # Correction time is calculated for this signal and evolution time value
154     if not zero_grad and not use_cor:
155         corr_signal = correlate(sqrt(timesignal_akku.y[0]**2+
156                                     timesignal_akku.y[1]**2),
157                                signal_0,mode=1)
158         correctionfile = open('CORR','w')
159         t_corr = (timesignal.x-timesignal.x.max())/2.0)[corr_signal.argmax()]
```

```

159             # put it into correctionfile
160             dacs[thekey] = t_corr
161             cPickle.dump(dacs, correctionfile)
162             correctionfile.close()
163             var += 1
164         end_time = time.time()
165         h5file = tables.openFile(fn)
166         datasets=0
167         for group in h5file.root.raw_data:
168             datasets += group._v_nchildren
169         h5file.close()
170         elapsed_time = end_time - start_time
171         ## send email at the end of experiment
172         import smtplib
173         server = smtplib.SMTP('localhost')
174         from_address = 'pfg@fc2.fkp.physik.tu-darmstadt.de'
175         to = 'markusro@element.fkp.physik.tu-darmstadt.de'
176         msg = """From:pfg@fc2\nTo:markusro@element\nsubject:Messung beendet\n
177             \nDie Messung ist beendet
178             \nDauer: %f\n\tDatenpunkte: %i"""%(elapsed_time, datasets)
179         server.sendmail(from_address, to, msg)
180
181
182
183 filename = "data/NaX_file_22092006"
184 fn = check_filename(filename)
185 h5file = tables.openFile(fn, mode = "w", title = "Measurement Data")
186 h5file.createGroup("/", "raw_data", 'raw data')
187 h5file.createGroup("/", "fft_data", 'fft data')
188 # save the Experiment and Result scripts
189 h5file.root.raw_data._v_attrs.__setattr__('exp_script',open('NaX_exp.py').read())
190 h5file.root.raw_data._v_attrs.__setattr__('data_script',open('NaX_data.py').read())
191 h5file.flush()
192 h5file.close()
193
194 result=result_it
195
196 h5file.close()

```


Bibliography

- [1] S. Berger and S. Braun. **200 and More NMR Experiments**. Wiley-VCH, 2004.
- [2] T. Butz. **Fouriertransformation für Fußgänger**. Teubner, 1 edition, 1998.
- [3] A. Gädke, C. Schmitt, H. Stork, and N. Nestle. **DAMARIS – A Flexible and Open Software Platform for NMR Spectrometer Control**. In *8th International Bologna Conference on Magnetic Resonance in Porous Media*, page P72, 2006.
- [4] J. H. Simpson and H. Y. Carr. **Diffusion and Nuclear Spin Relaxation in Water**. *Phys. Rev.*, 111(5):1201–1202, Sep 1958.
- [5] B. Stroustrup. **The C++ Programming Language**. Addison-Wesley, 2000.
- [6] D. D. Traficante and G. A. Nemeth. **A New and Improved Apodization Function for Resolution Enhancement in NMR Spectroscopy**. *Journal of Magnetic Resonance (1969)*, 71(2):237–245, 1987.