TNMR Frappy-NICOS Integration

Version 1.0.0

D. V. Garrad

August 29, 2025

Contents

1 Introduction

This document is meant to outline how one can set up a TNMR-Frappy server, as well as a NICOS-Frappy connection, and some basic information to understand the details "under the hood." Further, it provides limited technical details (i.e., programming reference); the majority of these details should be outlined in the Programming Guide[1] or in-line comments.

Note: this documentation was written throughout the development of the system. Some screenshots will likely be slightly outdated, but the main ideas hold.

[1] https://forge.frm2.tum.de/public/doc/frappy/html/

2 TNMR Interfacing

Tecmag's TNMR software does not have an official API, but it does come shipped with a Component Object Model (COM)[2] interface. This is intended as a means of scripting small experiments inside the TNMR interface, so there are limitations to the interfacing possibilities. Despite this, an API and helper library was written in Python to provide core functionalities, including:

- Pulse sequence generation
- Pulse sequence file generation
- Live parameter alteration
- Data acquisition starting, aborting
- File loading/saving

This API and library are typically hidden from the NICOS user within the Frappy server code, but the passionate user is welcome to contribute[3].

The Frappy server directly connects to an instance of TNMR if one is found, via the COM interface as described above. You will see errors if it cannot find an open instance of the program, and it should be opened at any time, before or after the Frappy server is started, but of course before attempting to take data or execute commands from NICOS.

[2] `https://learn.microsoft.com/en-us/windows/win32/com/component-object-model--com--portal`

[3] `https://github.com/SampleEnvironment/frappy`

3 Connection Scheme

In the most basic NMR experiment using this system, there is only one computer, running TNMR, a Frappy server, and a NICOS server. This is perhaps unrealistic for many institutions, as often the TNMR instance is running on a standalone system, disconnected from external devices such as cryostats and magnets. It is entirely possible, and encouraged, to separate these systems, so the TNMR instance and pulse generation and data acquisition (DAQ) hardware are connected to a dedicated PC. This is the configuration that this system was written for, so the rest of the documentation will assume it.

For notation, the PC running the TNMR instance and connected to the pulse generation and DAQ hardware will be referred to as the “NMR PC“ and the system running the NICOS server will be referred to as the “control interface”.

The basic pipeline for commands the user wishes to execute is as follows:

1. A user enables the TNMR module in NICOS on the control interface, and connects by entering the correct network address
2. A command is sent via NICOS on the control interface
3. The loaded TNMR NICOS module leverages SECoP to send the command to the NMR PC
4. The NMR PC receives this command, starting the necessary processes (i.e., starting an acquisition cycle, changing parameters in TNMR, gathering data, etc.) and sending back polled values through SECoP.
5. The NICOS server receives new values and updates the user display accordingly.

See Fig. 1 for a graphical outline of the connection scheme for an NMR experiment.

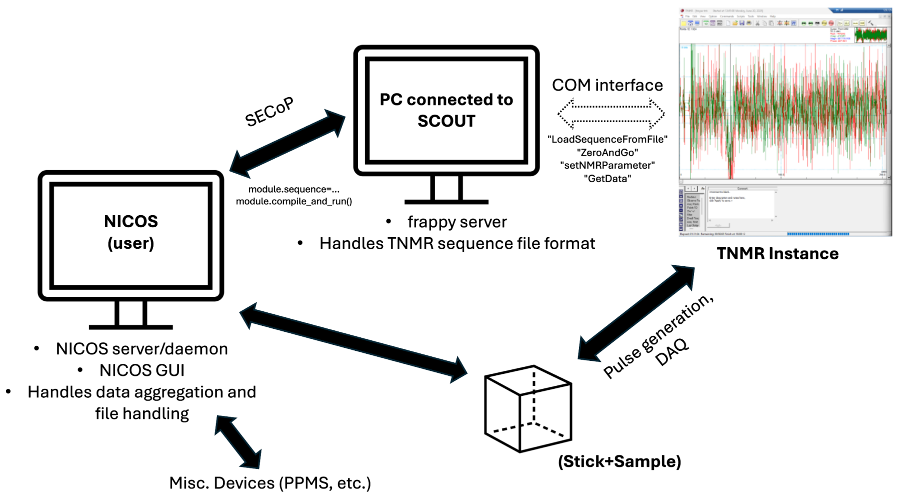


Figure 1: A connection diagram which outlines how each system in a typical NMR experimental setup should interact with each other. An instance of TNMR is open on a PC connected to a transmitter/receiver (in this case, a Tecmag SCOUT), which runs a Frappy server. This Frappy server then connects to a NICOS server via SECoP, and the end user interacts only with NICOS.

4 Setup

4.1 TNMR/Frappy Setup

On the TNMR PC, Frappy should be downloaded from the Git repository:

```
git clone https://github.com/SampleEnvironment/frappy
cd frappy
python -m pip install -r requirements.txt
python -m pip install pywin32 pythoncom
```

This will install Frappy, its requirements, and a Python/COM interface library. This may take some time. Further, after this is complete, it is recommended to navigate to the install location for `pywin32` and run the post-install script, `pywin32_postinstall.py`.

Before starting an instance of the frappy server, it is important that the user navigates to `frappy/frappy_psi/tnmr/templates` and, if not using the SCOUT, replaces the files `dashboard.txt` and `tmp.tnt` with working dashboard and .TNT files respectively. The values in these do not matter particularly, but a working base configuration otherwise is necessary. This is due to some TNMR dodginess. It is also encouraged that the user adds their setup files to the `setup_files` folder, under the name of their device, to develop a small library of working setups. Who knows, perhaps setup files suitable for your device are there already?

After starting an instance of TNMR (see Tecmag's website for guidance on installation), the Frappy server should be started. In a terminal (potentially the same one as used for the above), navigate to the Frappy base directory and run the following:

```
python bin\frappy-server nmr_scout
```

This will start the Frappy server, with the appropriate configuration file (configuration files are found in `cfg/`. In this case, the configuration used is `nmr_scout_cfg.py`) for TNMR interaction. The user is free to write their own configuration file as well, but one is provided with core functionality for ease of use. They are quite simple to write, and there are plenty of examples to learn from, if one is documentation-averse. Once the server is started, the process of setting up the NMR PC is complete! Keep TNMR and the console used open.

Some additional considerations:

- Windows updates should be disabled, as well as they can be, to minimise interruptions.
- Intel Speed Shift and Intel Speed Step (settings in the BIOS of some Intel-based PCs) should be disabled; these have been known to cause cryptic problems which can cause lost time.
- USB Auto-Suspend and HDD shutoff timers should be disabled to avoid similar problems.
- The `scout` (as created and named by the configuration file) will monitor for many errors ad warnings, but there is no guarantee that it will catch everything. The interface with TNMR is limited, so it is advisable to follow setup carefully to minimise risk.

4.2 NICOS Setup

This section does not provide a tutorial for configuring NICOS in its entirety[4], but rather a basic setup including just a TNMR interface. Further, it assumes that NICOS is already installed and you know where!

There exists a repository[5] with configuration files, helper function definitions, and example scripts for a basic NMR setup NICOS interface. To install these, execute the following to download the scripts from the base NICOS directory (often named `nicos/`, where one would find `bin/`, `data/`, and `nicos_sinq/`):

```
 cd nicos_sinq/
 git clone https://github.com/Davis-Garrad/
         NICOS-TNMR.git
 git clone https://gitea.psi.ch/linse/frappy_sinq.git
 cd ../../ # root directory (outside NICOS)
git clone https://gitea.psi.ch/linse/servicemanager.git
```

Now there are a few tasks to be completed:

- `nicos_sinq/tnmr/nicos.conf` has several values to be updated: cache ports, daemon ports, filepaths, etc.
- `nicos_sinq/tnmr/setups/frappy_tnmr.py` contains the saved data's filename template, as well as the default address to find the TNMR PC at.

Beyond these items, you can also adjust the GUI, filesinks, etc. if desired. This is covered under NICOS documentation, so it will not be covered here, except a listing of options that this package provides:

- NeXus Datasink (`nicos_sinq/tnmr/sinks/HDF5_NEXUS.py`)
- Custom TNMR-purposed commands (`nicos_sinq/tnmr/commands/tnmr_commands.py`). This should almost certainly be included in any setup. See Sec. 6.1 for more detail.
- Demo setups (`nicos_sinq/tnmr/setups`)
- Example scripts (`nicos_sinq/tnmr/example_scripts`)

[4] https://forge.frm2.tum.de/nicos/doc/nicos-stable/installconfig/
[5] https://github.com/Davis-Garrad/NICOS-TNMR

Once everything is configured as one desires, one can start the Frappy server as described in the previous section:

```
python {frappy root directory}/bin/frappy-server {cfg}
```

where `{cfg}` denotes a particular configuration file (as described in the frappy documentation) in `cfg/`. This could be, for example, `psi_nmr_setup` if the filename is `psi_nmr_setup_cfg.py`. Frappy, when starting a server, searches for files in the `cfg/` folder, ending with `_cfg.py`. Once the frappy server is initialised, one should start the NICOS server on a Linux machine (see 7):

```
{NICOS root directory}/bin/start.sh
```

Finally, the user can open a NICOS GUI or CLI and simply connect to the server configured in the previous step. If unsure of the address to connect to, it is helpful to look at the configuration files and the output of the NICOS server start command.

4.3 Installing DNMR

While the NICOS setup provides a framework for generating NeXus files to store NMR data, there does not exist (to the author's knowledge) a suitable analysis software. While the user of this system is encouraged to write any tooling they need as they like, there does exist an easily extensible open-source project called DNMR[6], built specifically for this system and purpose, but open more generally to NMR. This can be found either on `pip`,

```
python -m pip install DNMR
```

which should install all necessary dependencies, or if the user is more concerned with the inner workings of the software and/or wants to modify it (please do!), the source can also be found at its public GitHub repository,

```
git clone https://github.com/Davis-Garrad/DNMR.git
cd DNMR
pip install -e .
```

If you modify the program, please fork the project on GitHub and make your changes accessible to the public. Someone out there is probably looking for exactly what you've added!

[6]https://pypi.org/project/DNMR/

Once either of these versions has been installed, the program itself can be started with the command,

```
python -m DNMR
```

or, if installed from pip, simply

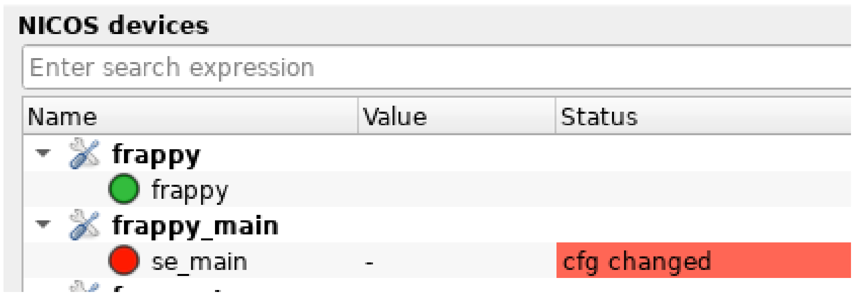
```
dnmr
```

Note: On Windows, `pip` *will additionally install a GUI-only command,* `dnmr-gui`.

5 Getting Started in NICOS

This section does not cover basic setup. The purpose of this section is to get the user started collecting and saving data in a running NICOS server, with an available frappy server to connect to.

The first step in using a frappy device is connecting to it. The NICOS server may automatically connect to it, in which case the user would see some sort of "ready" status. If this is not the case, then the status will contain information on some sort of connection error:



NICOS devices

Enter search expression

Name	Value	Status
frappy		
frappy		
frappy_main		
se_main	-	cfg changed

Figure 2: A failed frappy-NICOS connection on the `se_main` device.

In most cases, if configured correctly, this can be fixed by reconfiguring the device's `uri` parameter, after which NICOS will automatically try to reconnect. This value can be found in the frappy server configuration, as well as the frappy server command output.

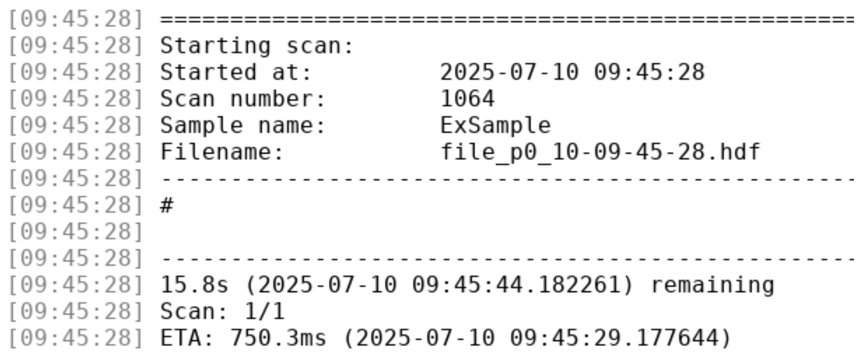
Now that the user has a frappy-TNMR interface configured and running in NICOS, they may want to take some data. This is simple enough, and example scripts for typical experiments are provided in `{nicos}/nicos_sinq/tnmr/example_scripts`. It is recommended that the user familiarises themselves with the the T_1 and frequency scan scripts, in particular, as they are most informative.

After one of these scripts is run, status updates will be appended to the NICOS log, as in Fig. 3 and a file will be generated containing all the data.

The data file can be loaded into DNMR directly from the command line:

```
python -m DNMR {filename(s)}
```

or via the in-app dialogue. Multiple files passed to the CLI or selected in a single loading operation in the GUI will be combined and treated as though



```
[09:45:28] ==================================================
[09:45:28] Starting scan:
[09:45:28] Started at:         2025-07-10 09:45:28
[09:45:28] Scan number:        1064
[09:45:28] Sample name:        ExSample
[09:45:28] Filename:           file_p0_10-09-45-28.hdf
[09:45:28] --------------------------------------------------
[09:45:28] #
[09:45:28]
[09:45:28] --------------------------------------------------
[09:45:28] 15.8s (2025-07-10 09:45:44.182261) remaining
[09:45:28] Scan: 1/1
[09:45:28] ETA: 750.3ms (2025-07-10 09:45:29.177644)
```

Figure 3: What the start of a scan looks like in NICOS. Note the scan number, sample name, and filename. These are functionalities of NICOS. The ETA and completion time are both a function of the TNMR-Frappy-NICOS system; see Sec. 6 for more detail.

the data were taken in one scan. In this fashion, field scans over several domains can be combined to view in tandem.

In this example, a T_1 scan is analysed: First, the data is loaded in and rephased. Each datapoint can be viewed, rephased, and the peak to analyse can be identified separately. For the purpose of a simple T_1 scan, however, it makes the most sense to just use the auto-rephasing feature. This is illustrated in Fig. 4.

One can also, if using a particular filter which doesn't already, apply a window to the Fourier transform of the data. This will be used in integrating the data for the T_1 datapoints. Regardless of whether the user needs to do this or not, the Fourier transform can be found in the "FT" tab, as shown in Fig. 5.

After the data is made ready from the previous steps, it can be viewed in a custom-made tab, built specifically for T_1 scan data. Fig. **??** shows what this might look like for some particularly pleasant data, as well as a fit which was made after clicking the "Fit" button.

Now the user has, in following this example, taken, inspected, and analysed data. This concludes the "Getting Started" section.

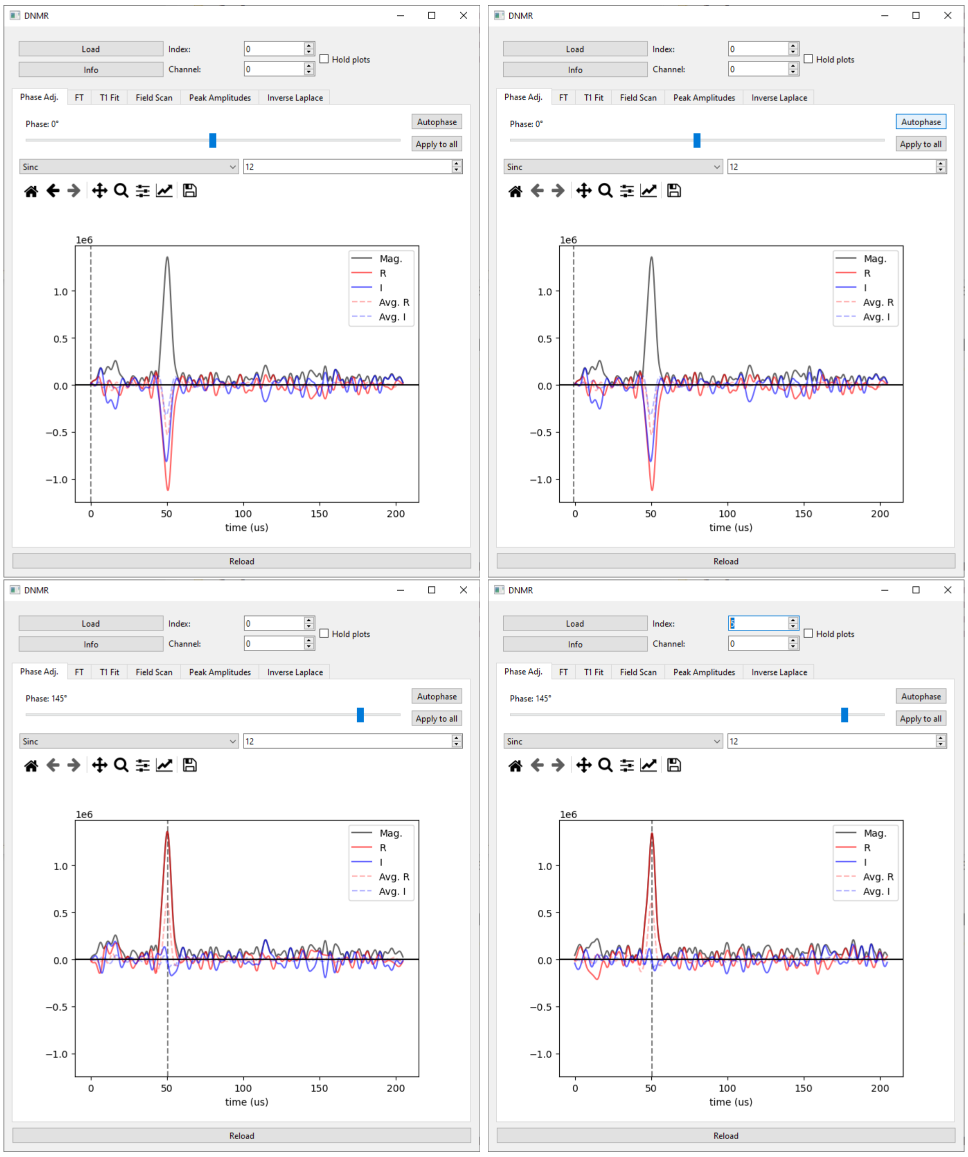


Figure 4: In a T_1 analysis, the first step is rephasing the data. DNMR has an inbuilt auto-rephasing functionality, activated by clicking the button labelled "Autophase." The next step is to scroll through the different scan points (with their own "index") and verify that the data and rephasing looks reasonable.

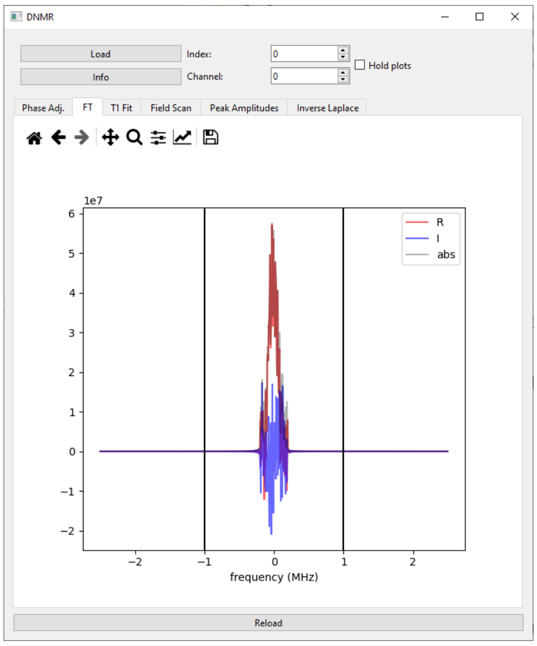
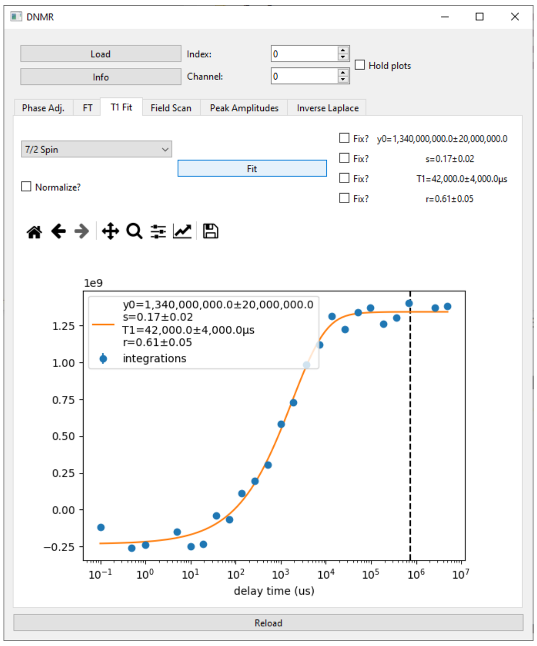


Figure 5: The Fourier transform of the data can be viewed in the “FT” tab.



Fix? y0=1,340,000,000.0±20,000,000.0
Fix? s=0.17±0.02
Fix? T1=42,000.0±4,000.0μs
Fix? r=0.61±0.05

y0=1,340,000,000.0±20,000,000.0
s=0.17±0.02
T1=42,000.0±4,000.0μs
r=0.61±0.05

Figure 6: The “T_1 Fit” tab is custom-made to perform fits on T_1 data, so this is the logical next tab to visit. After pressing the “Fit” button, we can then parse the result.

6 Basic Programming Reference

In this section, commands and functions will be detailed together, but colour-coded separately. Commands (accessible to NICOS users) will be coloured **blue**, parameters (also accessible to NICOS users) will be coloured **teal**, and developer functions (accessible only behind the scenes) will be coloured **grey**. Section titles are the filenames in which one can find the discussed functionality, without extensions.

6.1 NICOS

User Commands

`generate_pulse(pw, ph, dt, pc)`

Generates a dictionary representing a pulse in a pulse sequence. Parameters are pulse width (us), pulse height (a.u., % total power for the SCOUT), delay time (us), phase cycle (str, e.g., "0 1 2 3"). These dictionaries can be modified, and have elements `pulse_width`, `pulse_height`, `delay_time`, and `phase_cycle`.

`generate_sequences(base_sequence: list, pulse_indices: list, var_name: str, vals: list)`

Creates a list of pulse sequences from a single pulse sequence, assigning a variable in a collection of pulses to a value in the given list. For instance, if a user wanted to scan the first and second delay times over the range 5-10us in 2us steps, they could write `generate_sequences(base_sequence, [0,1], 'delay_time', [5,6,8,10])` to generate the appropriate list. In effect, `base_sequence` is copied for each entry in `vals`, and the value of `var_name` in each of the pulses indicated by `pulse_indices` are changed to one of the values in `vals`.

`log_durations(s, e, N)`

Generates a list of durations between `s` and `e` (inclusive), logarithmically spaced.

`estimate_sequence_length_from_device(dev, seq)`

Returns the estimated time, in seconds, of a single pulse sequence acquisition. Uses the parameters in `dev`, a TNMR device loaded into NICOS.

`estimate_sequence_length(params, seq)`

Returns the estimated time, in seconds, of a single pulse sequence acquisition. Uses the parameters in `params`, a dictionary with keys like those in a TNMR device. Requires that `params` contains at least `acquisition_time`, `pre_acquisition_time`, `post_acquisition_time`, and `num_acqs`.

`estimate_scan_length_from_device(dev, scan_seq)`

Returns the estimated time, in seconds, of a list of pulse sequence acquisitions. Uses the parameters in `dev`, a TNMR device loaded into NICOS.

`estimate_scan_length(params, scan_seq)`

Returns the estimated time, in seconds, of a list of pulse sequence acquisitions. Uses the parameters in `params`, a dictionary with keys like those in a TNMR device. Requires that `params` contains at least `acquisition_time`, `pre_acquisition_time`, `post_acquisition_time`, and `num_acqs`

`timestring(seconds)`

Returns a nicely formatted string indicating a length and how far that is into the future.

`print_sequence(seq)`

Pretty-prints a pulse sequence for easy viewing.

`scan_sequence(dev, seq, additional_lambdas={})`

Given a TNMR interface `dev`, performs a compilation, zero-go, and saves data from the acquisition for a single pulse sequence. Every value returned by the lambda functions in `additional_lambdas` (each of the signature `f()`) will be saved as part of the environment, with names given by their keys in the dictionary.

`scan_sequences(dev, sequence_list, additional_lambdas={})`

Given a TNMR interface `dev`, acquires data for a list of pulse sequences and bundles the data together in one file (with separate entries). This is simply the list-form of `scan_sequence`. See the reference info for that function to get a better idea of how this works.

`get_tnmr_params(dev)`

Returns a dictionary of the currently loaded parameters in the given TNMR interface, like acquisition time, observed frequency, and 1D scans.

`update_device_parameters(dev, params: dict)`

Takes a dictionary of keys corresponding to some device parameters (such as those described in the TNMRModule/Frappy section below), and sets their values in the device `dev` to those in `params` `tnmr_scan()`

While this is technically neither a parameter nor a function, it *is* likely of use to the reader. The `tnmr_scan` object is a Python Context Manager, for use in a `with` statement. The top-level `tnmr_scan` object both initialises a file, and dictates which NMR acquisitions are written to said file. Every call to the provided Data Manager's `putValues` and `finishPoint()`, *etc.* deal with the file opened via a line of code that looks like `with tnmr_scan() as data_manager: ....` This includes those included in the `scan_sequence(s)` functions. The shown data_manager variable would then be a NICOS DataManager object; see the NICOS Reference if you wish to use this advanced functionality, or leave it as `with tnmr_scan(): ...` if you plan to only use the `scan_sequence(s)` funcionality described herein. In essence, this class provides a context for NICOS file handling.

6.2 Frappy

6.2.1 TNMR VisualBasic/COM Interface (`tnmr_interface` and `NTNMR`)

It is recommended to check Tecmag's TNMR documentation for detailed notes. This document does not seem to be publicly available, but can likely be sourced from the support.

6.2.2 `TNMRModule`

Commands/Parameters accessible through SECoP

This encapsulates all the functionality that NICOS directly sees.

`compile_and_run(thread=True)`

Compiles and runs the currently-loaded pulse sequence and options. If `thread` is set to `True`, then this function starts a child thread. Otherwise, this function blocks.

`kill()`

This command sends an Abort signal to TNMR, signalling the program to halt an acquisition. This does *not* guarantee that the acquisition will end, though one can check the status of the module to be sure.

`title`

This parameter only determines what the title written into the sequence file is.

`sample`

Open text describing the sample which is written to metadata.

`nucleus`

Open text meant to describe the addressed nucleus/nuclei.

`comments`

This parameter is saved to metadata. It acts as an open text field for any miscellaneous comments.

`(pre,post,-)acquisition_time`

See `sequence_generation`. Pre- and regular acquisition times in μs, post-acquisition time in ms.

`ringdown_time`

See `sequence_generation`. In μs

`acq_phase_cycle`

See `sequence_generation`.

`num_acqs`

This parameter determines the total number of averaged scans for each datapoint. Formerly `num_scans`.

`obs_freq`

The frequency of the receiver. In MHz.

Developer Functions

`tnmr()`

This function creates a new instance of the TNMR API wrapper. To be used upon every access to the TNMR software from code.

`stop()`

Simply provides the core functionality for `kill()`.

`__compile_sequence(lockstatus=False)`

Compiles the sequence loaded in `sequence_data`. This involves the following:
1. creating the sequence table via `seq_gen.get_single_pulse_block` calls;
2. combining them all; saving this sequence where TNMR can see it, and in a format it can read;
3. taking a copy of all the acquisition parameters (so that if they are changed mid-acquisition, no incorrect information is written to files);
4. telling TNMR to read it (i.e., `tnmr().load_sequence()`), reloading the dashboard and parameters in the process;
5. and giving TNMR the correct parameters to populate the new dashboard with.

The temporary sequence file is saved in `root/sequences` and timestamped where `root` is the working directory of the frappy server command. The word "compile" is a bit of a misnomer in its usage here, but signifies that all of a pulse sequence's information is collected and prepared according to some process to ensure both that it works and that it is saved in an accessible and reproducible manner.
If the optional parameter `lockstatus` is set to `True`, the function will only set the status of the device to `BUSY` when it begins, and will not set the status to `PREPARED` when it terminates.

`__zero_go()`

This command simply sends a command to TNMR to begin to acquire data (Zero-and-Go). This is not meant to be called before `__compile_sequence()`, and the consequences of such an action are deemed **undefined behaviour**. Things probably won't break, but they won't do what you want, either!

`__compile_and_run()`

An extremely thin wrapper for first compiling and then Zero-Going.

6.3 `tnmr_interface`

This module exists entirely as an API for Tecmag. Most functions are simple wrappers for the COM interface.

`get_instance()`

This function attempts to create or get an existing instance of a COM interface object, connected to a TNMR instance. It is important that the handle for this COM object is often refreshed (i.e., not stored in an object), as the interface can fail when new threads/processes are created and try to do the same. Treat this as a singleton pattern to avoid headaches and trouble.

`execute_cmd(cmd)`

Sends an arbitrary COM interface command through to TNMR. This is useful for debugging, but should not be accessible to end users, as it can destabilise things if used improperly. Think of it as a really big sledgehammer - no user should ever come close, even if *they* think they need to remove a wall.

`openfile(filepath, active=True)`

Opens a `.TNT` file into TNMR. This can provide direct access to the data and sequence used to get it. Further, it also loads some settings from the .TNT.

`set_activefile()`

Updates the active file and active filepath internally. (Currently unused)

`ZeroGo(lock=True, interval=0.5[s], check_time=10[s])`

If no acquisition is already running, signals to TNMR to Zero-and-Go. If successful in sending this signal, the function then forces a halt until some condition is met (within the function, a constant `CHECK_MODE` is set to either “data” or “thread”. The condition for “data” is that the zeroth data point changes within `check_time`. The condition for “thread” is that the thread started to send the Zero-Go command terminates before `check_time`. Default is “thread”) or `check_time` elapses. If `check_time` does elapse, this is a signal of some fault in TNMR and the function recursively tries again. Once this succeeds, if `lock` is enabled then every `interval`, TNMR is polled to see if the acquisition is complete and control is released once it is.

`acquisition_running()`

Returns a boolean value: whether an acquisition is currently running or not.

`get_data()`

Pulls all the data from TNMR and returns it as a tuple: (reals, imaginaries).

`save_file(filepath=”)`

Saves the current setup as a .TNT file. If `filepath` is left as the default, then the currently active file is overwritten.

`set_nmrparameter(param_name, value)`

Sets a dashboard parameter. It does not matter which page the parameter is on. Eg., `set_nmrparameter("Obs. Freq", "41.5MHz")` has the same effect as manually changing “Obs. Freq” in TNMR to 41.5MHz.

`get_nmrparameter(param_name)`

Returns the value of a dashboard parameter.

`is_nmrparameter(param_name)`

Returns a boolean: whether the provided parameter name is found in the list of valid parameters.

`get_all_nmrparameters()`

Returns a dictionary of all the dashboard parameters, in the form { [page name]: { [parameter name]: [parameter value], ... }, ... }. This may result in duplicates, as parameters can be on multiple pages at once.

`get_page_parameters(page)`

Like `get_all_nmrparameters`, but for a single page.

`load_sequence(filename)`

Possibly destructive to live data. This is more interesting than it sounds. Sequences cannot be loaded directly as TNMR's API is slightly dodgy here. This function, in short, closes the active file (this is the destructive part), loads a template file with some default values, loads a default dashboard, loads a sequence into the template file, saves the template file into an even more temporary file, and finally re-opens that file. These acrobatics are justified in that doing anything else results in TNMR being unhappy.

`load_dashboard(dashboard_fn)`

Loads a dashboard from a dashboard setup file. Despite what the TNMR documentation says, this *does* seem to overwrite the values in the dashboard. Potentially useful, but not used currently.

6.3.1 `sequence_generation`

This file exists to hold helper functions for reading and writing pulse sequence files in Tecmag format. The structure of a pulse sequence is limited to aid in usability and simplicity. Every pulse sequence is composed of a sequence of pulses (if one cannot wrap their mind around this concept, they should be kept far from RF electronics). Each of these pulses is parameterised by 4 values:

- Pulse width; how long the signal remains HIGH for.
- Pulse height; the power output of the signal.
- Delay time; how long to keep the signal LOW for after the pulse.
- Phase cycle; a string, of the form “0 1 2 3 3 2 1 0” which determines the phase cycling sequence. This is dependent on the hardware used. This string must be a series of integers separated by whitespace.

These pulses, in Python, are stored as dictionaries with keys `pulse_width`, `pulse_height`, `delay_time`, and `phase_cycle`. These pulses (dictionaries) can be combined into a pulse sequence (list of dictionaries).

In the `sequence_generation` implementation the dictionaries are slightly more complicated as this is the lowest level of pulse sequence handling; they contain fields for every possibility in TNMR. There exists a helper function, `combine_blocks(l, r)`, which should be used to combine blocks (collections of columns) in Python (TNMR). Further, all pulse sequences must be bookended by an “initial” block and a “final” block, which define phase reset, acquisition, ringdown, etc..

Functionality to create pulse sequences

```
get_single_pulse_block(name, pulse_width, pulse_height, delay_time, phase_cycle=“0”)
```

Creates a single “block” dictionary for a single pulse. All times should be provided in the format “10u”.

```
get_initial_block()
```

Creates a single “block” dictionary to be placed at the start of any pulse sequence.

`get_final_block(ringdown_time, preacquire_time, acquire_time, cooldown_time, acq_phase_cycle="0")`

Creates a single "block" dictionary to be placed at the end of any pulse sequence. Ringdown before acquisition, a delay after ringdown (`preacquire_time`), acquisition time, post-acquisition delay, and the acquisition phase cycle are all configurable. All times should be provided in the format "10u".

`combine_blocks(l, r)`

Combines two "blocks", `l` and `r`, in the order: `l` first, then `r`.

Please review the below code snippet for an example:

```
'''Creates a pulse sequence containing a 5 microsecond pulse,
   followed by a 50 microsecond pause, a 10 microsecond pulse, a
   204.8 microsecond acquisition, and then a 500ms delay.'''
start = get_initial_block()
end = get_final_block('10u', '1u', '204.8u', '500m', '0 2 0 2')
p1 = get_single_pulse_block('p90', '5u', '40', '50u', '0 1 2 3')
p2 = get_single_pulse_block('p180', '10u', '40', '1u', '1 0 3 2') # I
    don't really know how to phase cycle... But who does?

total_pulse = combine_blocks(start, p1)
total_pulse = combine_blocks(total_pulse, p2)
total_pulse = combine_blocks(total_pulse, end)
```

Functionality to handle files

`save_sequence(filename, sequence)`

Saves a sequence to a file in Tecmag's format.

`save_sequence_cfg(filename, sequence)`

Saves a sequence to a file in a human-readable JSON file. Good for testing and logging.

6.3.2 `sequence_fileformat`

This is the best documentation of the TNMR sequence file format that I could generate from inspection and trial and error. It also happens to be the file which provides some helper functions to generate proper file structure. Generally, even a developer for this software can ignore this particular interface, and limit themselves to using the `sequence_generation` interface.

`fm(s, spacing=3)`

Generates the correct string for a value in string format such that TNMR can understand it. Pretty much every value in a TNMR sequence file is formatted as such.

`get_info_header(filename, author, col_names, tuning_number, binary_name="PSEQ1.001.18 BIN")`

Generates a string for the top of the sequence file, which contains information on the filename, author, columns, the number of columns (`tuning_number` for historical reasons), and the binary used to generate the file. While programs using this API will emphatically *not* be named "PSEQ1.001.18 BIN", this is what TNMR expects when loading a file. We always tell TNMR what it wants to hear.

`get_delay_header(col_delays, tuning_number)`

Generates a string for the delay section of the sequence file.

`get_event_header(event_type, vals, tables, table_reg, tuning_number, col_delays)`

Generates a string describing the events section of the sequence file. If tables are used anywhere in the events, they should be registered in the table registry (`table_reg`). This part should come after the delay section.

`get_table_spec(table)`

Generates the string for the table specification in the sequence file. This goes after the event section.

```
generate_default_sequence(col_names, col_delays)
```

Generates a dictionary containing all the necessary information to provide a solid base to make small adjustments to. Default values can be found in the `sequence_fileformat.py` file, under `event_defaults`. The generated dictionary should be passed to `create_sequence_file` after adjustments.

```
create_sequence_file(filename, data, author="NA")
```

Takes a dictionary with all possible information that Tecmag could possibly require, and generates a .TPS Tecmag pulse sequence file from it. The helper function `generate_default_sequence` can be useful here.

7 Drawbacks, Known Issues, and Other Notes

- Only Linux is supported for the NICOS server. This is due to the use of `os.fork`, technically a part of Python's standard library but very much not cross-platform. Since no TNMR program exists for Linux, this means that two computers are necessary for this setup. Fortunately, one of them can also act as "master" for the rest of the setup, and the TNMR PC can be somewhat isolated.
  - A potential solution to this is to run NICOS on Windows Subsystem for Linux (WSL), which allows running Linux live on a Windows instance. This has not been tested.
- If there is some problem in TNMR that requires intervention, the acquisition commands will recursively retry, possibly leading to a stack overflow and therefore a crash of the Frappy server. This, in the author's experience, has never happened in reality but it is quite possible in theory. In the end, the only adverse effect this will have is a disconnection from the NICOS side, which may allow a script to either throw an exception or continue at points it should acquire.
- It seems from debugging that NICOS cannot handle variable-length arrays. This implies that pulse sequences must be given a maximum length. This is handled in-code via creating a large (currently 100 pulses) pulse sequence, and editing/sending to TNMR only the first *N* pulses as required by the use case. 100 pulses is likely to be overkill for 99% of users, but the 1% can either rethink their experiment a little bit, or edit the `TNMR_MAX_PULSES` global in (Frappy root)`/TNMRModule.py`.
- TNMR does not report to the API when precision issues occur. This can effect the user if they desire to set a delay less than 0.1μs, or if they require a high floating point precision. This is not likely to be an issue, as the SCOUT, for example, only has time precision down to 80ns in any case.