poise

Jonathan Yong

Nov 14, 2020

CONTENTS

POISE (*Parameter Optimisation by Iterative Spectral Evaluation*) is a Python package for on-the-fly optimisation of NMR parameters in Bruker's TopSpin software.

<citation should be put here>

The code is licensed under the GNU General Public License v3 and can be found on GitHub. The latest HTML version of the documentation can always be found at https://foroozandehgroup.github.io/nmrpoise. Likewise, the latest PDF version is always at https://foroozandehgroup.github.io/nmrpoise/poise.pdf.

In here you will find guides on setting POISE up and using it in routine NMR applications. This guide can largely be read in sequence. However, depending on your level of interaction with the software, you may not need to read all of it. For example, if somebody else has already set up POISE for you, you can probably skip to *Running an optimisation*.

CHAPTER

ONE

INSTALLATION

POISE comprises a *frontend* script, which is accessible from within TopSpin itself, as well as a *backend* script, which has to be run using Python 3 (i.e. not TopSpin's native Python interpreter).

This means that Python 3 must be installed separately first. In particular, POISE requires a minimum version of **Python 3.6.** For Windows, your best bet is to download an installer from the Python website. For Unix machines, we suggest using a package manager to do so (such as Homebrew for macOS or `apt`/`yum` & their equivalents on Linux), although the installers are also fine.

Once that's done, you can install POISE using `pip`:

```
pip install nmrpoise
```

The package requirements are `numpy`, `scipy`, `pandas`, and `Py-BOBYQA`; these will be automatically downloaded if necessary.

`pip` tries to take care of installing the scripts to your TopSpin directory. To do so, it checks for TopSpin installations in standard directories (`/opt` on Unix and `C:\Bruker` on Windows). If `pip` exits without errors, this should have succeeded; you can test it by typing `poise -h` into TopSpin's command-line, which should spawn a popup. If that is the case, congratulations — you can move on to the next chapter, *Setting up a Routine*.

1.1 Troubleshooting

The installation can occasionally fail if TopSpin is installed to a non-standard location. To solve this issue, you can specify the TopSpin installation directory as an environment variable `TSDIR` before installing POISE. The way to do this depends on what operating system (and shell) you use.

On Windows PowerShell, run the following command:

```
$env:TSDIR = "C:\Bruker\TopSpinX.Y.Z\exp\stan\nmr"
```

replacing the part in quotes with your actual TopSpin installation directory (it must point to the `exp/stan/nmr` folder).

On old-school Windows `cmd`, use:

```
set TSDIR="C:\Bruker\TopSpinX.Y.Z\exp\stan\nmr"
```

On Unix systems, use:

```
export TSDIR="/opt/topspinX.Y.Z/exp/stan/nmr"
```

(Unless you're using `csh` or the like, in which case you use `setenv`, although you probably didn't need to be told that!)

After running the appropriate command for your operating system, `pip install nmrpoise` should be able to detect the `TSDIR` environment variable and install the scripts accordingly.

1.2 From source

If you obtained the source code (e.g. from `git clone` or a GitHub release) and want to install from there, simply `cd` into the top-level `nmrpoise` directory and run:

```
pip install .
```

or equivalently:

```
python setup.py install
```

The installation to the TopSpin directory is subject to the same considerations as above.

CHAPTER

TWO

SETTING UP A ROUTINE

If you're coming here from the *Installation*, you should make sure that POISE has been installed correctly. A simple check is to type in `poise -l` into the TopSpin command line: if it shows a text box, then you should be good to go.

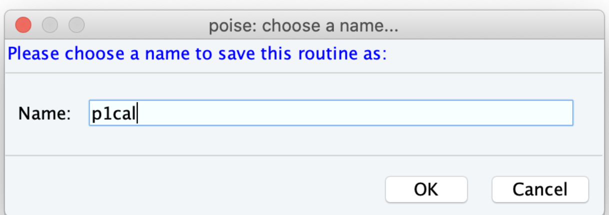
Each optimisation in POISE is controlled by a **routine**, which contains all information necessary for an optimisation. The ingredients of a routine are:

- A name
- The parameters to be optimised
- Lower bounds, upper bounds, initial values, and tolerances for each parameter
- A cost function which determines the 'badness' of a spectrum
- *(Optional)* The name of an AU programme for acquisition and processing

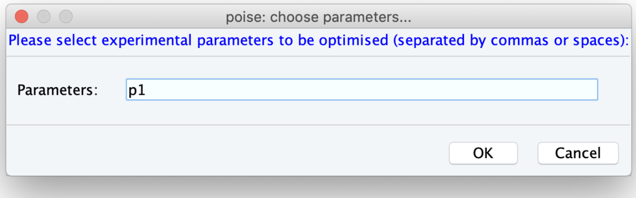
We will now walk through how to set a routine up, elaborating on each of these ingredients as we come to them. To get this process started, type `poise` into the TopSpin command line.

Note: POISE has a number of command-line options. If you're interested in finding out more about these, `poise -h` will give you a short summary of each of them, and *Frontend options* has additional info.

The routine we will set up now is one for the calibration of the 360° pulse width. The first ingredient we need to provide is a **name**. I've used `p1cal` for this, but of course you can choose anything you prefer:



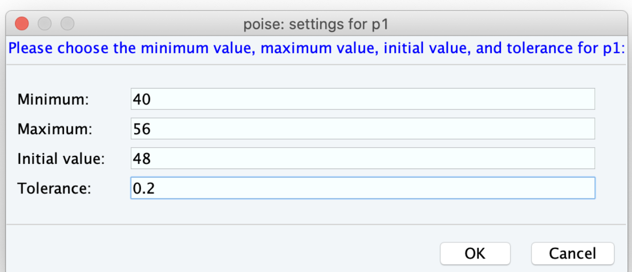
After clicking OK, you will be prompted to provide the **parameters** that are being optimised. Here we are just optimising one parameter, `p1`.



Note: Only parameters that take on float values can be optimised (pulses `p`, delays `d`, constants `cnst...`) Integer values, like `td` or loop counters `l`, will not work.

At this stage, you will be prompted to enter the **bounds**, **initial value**, and **tolerances**. The lower and upper bounds simply reflect a range within which the optimum can reasonably be assumed to lie within, and the initial value should be your best guess at where the optimum is.

On the spectrometer we're currently using, the Prosol value for a 90° pulse is 12 µs, so we'll go ahead and set the initial guess for the 360° pulse to be 48 µs. *(If your Prosol value differs, you should adjust these values accordingly.)* The lower and initial bounds can be 40 and 56 µs respectively, corresponding to a 90° pulse of between 10 and 14 µs. The tolerance, on the other hand, roughly reflects the degree of accuracy that you want in the answer. Here we've used a value of 0.2 µs.



Choosing tolerances can be tricky sometimes. Too large a tolerance can lead to inaccurate answers (as the optimisation converges before it's really found the minimum); and too small a tolerance is meaningless, as often the resulting spectra are barely different. Generally, it's a good idea to choose the smallest value where going in either direction will give you an appreciable difference in the spectrum. However, it doesn't have to be *too* precise: as long as you aren't off by an order of magnitude POISE will still work reasonably well.

Note: The default TopSpin units for pulse lengths are microseconds, so the unitless `48` is equal to 48 µs. However, for delays the default units are seconds.

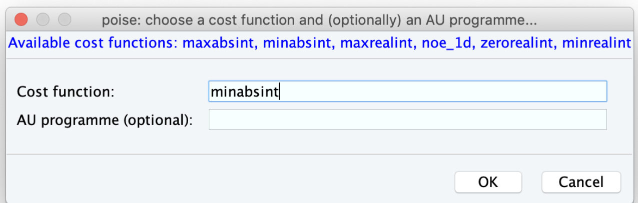
POISE also allows you to specify units using the suffixes 'u', 'm', and 's' for microseconds, milliseconds, and seconds respectively. This is designed to mimic TopSpin's parameter settings, where `30m` means 30 ms (for example). So you can enter `48u` in this screen as well, or indeed `0.048m`.

Finally, we have to choose a **cost function**, as well as (optionally) an **AU programme**.

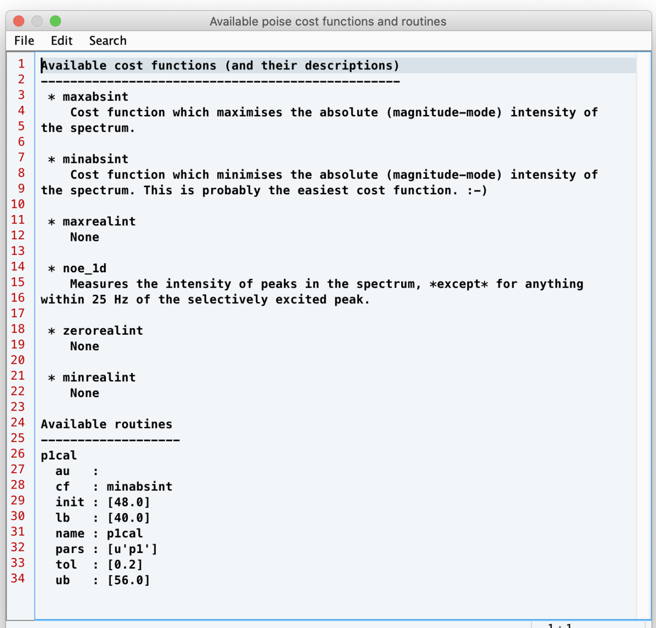
The cost function is a Python function which reads the spectrum and returns a *'cost'*, i.e. how bad the spectrum is. All optimisations in POISE seek to minimise the cost function. In our case, the best value of `p1` is one for which the intensity of a pulse-acquire spectrum (`zg`) is minimised, i.e. magnetisation is returned to the positive z-axis. So, we can conveniently use the intensity of the magnitude-mode spectrum as the cost function. This cost function is also bundled with POISE, and is called `minabsint`. (For those who are familiar with TopSpin's built-in `popt`, this is equivalent to the `MAGMIN` criterion.)

Note: For more information about the built-in cost functions, check out *Builtin cost functions*.

The AU programme controls spectrum acquisition and processing, and can be left blank in this case. All we need to do for this routine is to acquire the spectrum, Fourier transform, then perform phase and baseline correction. For 1D and 2D datasets, if the AU programme option is left blank, POISE will automatically do exactly these steps. Therefore, there is no need to specify an AU programme unless you want to customise this process.



That's it — congratulations, you've set up a POISE routine! If you type `poise -l` now, you should now see the `p1cal` routine (or whatever you named it) appear in the text box:



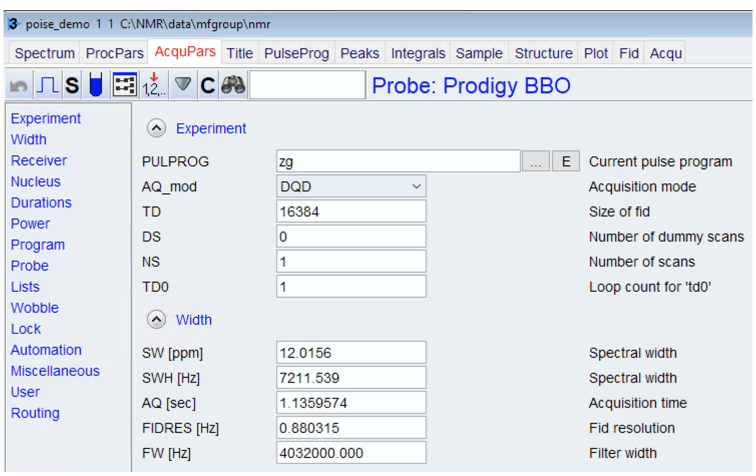
CHAPTER THREE

RUNNING AN OPTIMISATION

Assuming you or someone else has already created a routine (see *Setting up a Routine* if not), this page will show you how to run the optimisation. We'll use the same `p1` calibration routine that we described on that page, but the principles apply equally to all routines.

The first thing to do is to set up the NMR experiment. Use `edc` or `new` to create a new proton pulse-acquire experiment. You should use the pulse programme `zg` (not `zg30` or `zg60`!). Set the other experimental parameters, such as the spectral width `SW`, relaxation delay `d1`, etc. as desired for your compound of interest.

All these steps can in principle be done most easily by loading a parameter set (`rpar`). On Bruker systems, there should already be a builtin `PROTON` parameter set. After loading this parameter set you will have to run `getprosol`, then change the pulse programme to `zg`. Alternatively, there might be a different parameter set that has been set up by a member of the NMR staff for simple proton spectra. As long as you make sure the pulse programme is `zg` the optimisation will work fine.



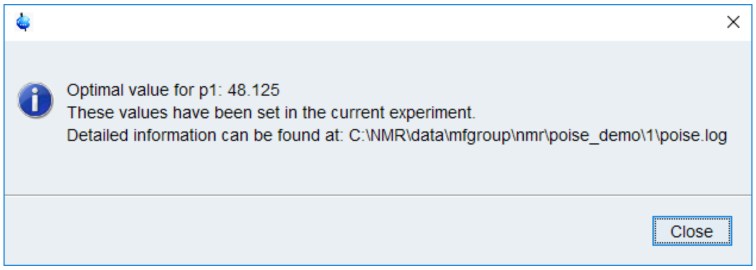
Note: Apart from the pulse programme, basically every other parameter can be set to whatever you like. However, to reduce the overall time taken, it's generally a good idea to try to make each experiment as short as possible. In this case, even with very dilute samples, 1 scan (`NS=1`) will suffice. We find that dummy scans are not needed to obtain accurate results, so it is permissible to set `DS=0` (although see next paragraph for a caveat). You could cut this even further by lowering `TD` to 8192 (for a given `SW`, this translates to a shorter acquisition time `AQ`).

For routine usage we recommend using at least 1 dummy scan. This (`p1cal`) is the only optimisation example in which we have used 0 dummy scans. Skipping dummy scans altogether can lead to inaccuracies in the cost functions (as the system has not reached a steady state).

Lock, shim, and tune as usual (if you haven't already). Once that's done, simply enter into the TopSpin command line:

```
poise p1cal
```

Sit back and watch it run! You should get a result in 1–2 minutes.



The best value(s) will automatically be stored in the corresponding parameter so that any subsequent acquisition will use the optimised parameters. Don't forget that for this particular optimisation, you will have to divide the optimised value by 4 to get back the 90° pulse.

3.1 Parsing the log

If you are interested in analysing data from (possibly multiple) optimisation runs, all information is logged in a `poise.log` file. This log file is stored inside the expno folder (the post-optimisation popup also tells you where it can be found — see above for an example). It can be parsed using a Python 3 script:

```
>>> from nmrpoise import parse_log
>>> # pass it the path to poise.log, or to the directory containing it
>>> parse_log("C:/NMR/data/mfgroup/nmr/poise_demo/1")
  routine  initial param    lb    ub  tol algorithm     costfn    auprog  optimum    fbest  nfev  time
0   p1cal     48.0  [p1]  40.0  56.0  0.2        nm  minabsint  poise_1d   48.125  6.849146e+06    10    77
```

parse_log returns a `pandas.DataFrame` object which contains most of the information in the log file. However, this object does not include details of individual cost function evaluations (even though these are fully logged). If you want to analyse that data, you will have to write your own function!

The full documentation for *parse_log* (which really doesn't say much more than the previous example) is as follows:

`nmrpoise.`**`parse_log`**(*fname='.'*)

Parse a poise.log file.

Parameters

fname [(optional) str or pathlib.Path or int] Path to poise.log file, or the folder containing it (this would be the TopSpin EXPNO folder). If not specified assumes the current working directory. If passed as an int, assumes "./<fname>" (i.e. expno X in the current working directory).

Returns

log_df [pandas.DataFrame] Dataframe with rows corresponding to optimisations which successfully terminated. The time taken is given in seconds.

3.2 Errors

POISE tries its best to exit gracefully from errors, and often you won't need to care about any of them. However, if something *does* go wrong during an optimisation, errors will be logged to the two files `poise_err_frontend.log` and `poise_err_backend.log`, depending on which script runs into an error. These files reside in the same folder as `poise.log`, i.e. the "expno folder". We welcome bug reports — please submit an issue on GitHub or drop us an email (addresses can be found on the paper).

CHAPTER

FOUR

FRONTEND OPTIONS

The frontend script (i.e. the script which you run in TopSpin) has a variety of flags which control its behaviour. These are fully described here. You can also access short descriptions of these by typing `poise -h` in TopSpin.

4.1 General options

`--create NAME PARAM MIN MAX INIT TOL CF AU`

Create a new named routine. This can only handle one-parameter routines; if you want a multiple-parameter routine, please use the GUI. The AU programme must be specified, it can't be left blank like in the GUI. Also, short forms such as `4u` for 4 µs are not allowed here.

`--delete DELETE`

Delete a named routine. For example, run `poise --delete p1cal` to delete a routine named `p1cal`.

If you want to edit a routine, you can just re-create a new routine with the same name: the old one will be overwritten.

`-h, --help`

Show a help message then exit.

`--install`

For ultra-quick installation of p1 or DOSY optimisation scripts (see *POISE User Guide*). Should not be otherwise used.

`--kill`

Kill POISE backends that may still be running.

Running `poise --kill` should be the first course of action if you find unusual behaviour after terminating a POISE optimisation (e.g. being unable to delete a log file as it is still in use). If this does not work, then you may need to manually kill the Python processes: for example, on Windows PowerShell, run:

```
Stop-Process -name python
```

or on Unix systems:

```
killall -9 python
```

(replace `python` with `python3` as appropriate for your system).

`-l, --list`

List all available routines and their parameters, then exit.

4.2 Options for running optimisations

`-a ALG, --algorithm ALG`

Use the algorithm ALG for the optimisation. ALG can be one of `nm` (for Nelder–Mead), `mds` (for multidirectional search), or `bobyqa` (for Py-BOBYQA). The default is `nm`.

`--maxfev MAXFEV`

Maximum function evaluations to allow (i.e. maximum number of spectra to acquire during the optimisation run). If the optimisation reaches the limit, it will terminate, reporting the best value so far as the 'optimum'.

This is useful for enforcing an upper limit on the time taken to perform an optimisation. Since by far the majority of the time is spent on acquiring the NMR spectra, `MAXFEV` evaluations will simply take roughly `MAXFEV * t` time to run (where `t` is the time taken for one spectrum — you can find this out using TopSpin's `expt` command).

If you don't want to have a limit on function evaluations, just don't use this flag, or pass the value of 0. Technically, there is always a hard limit on the number of function evaluations (which is 500 times the number of parameters being optimised). However, it is probably almost impossible to run into that hard limit.

`-q, --quiet`

Don't display the final popup at the end of the optimisation informing the user that the optimisation is done. This is mostly a matter of taste, as the final popup does not block any subsequent commands from being executed.

`-s, --separate`

Use a separate expno for each function evaluation. Note that if POISE runs into an expno which already exists, it will terminate with an error!

CHAPTER

FIVE

POISE UNDER AUTOMATION

The command-line interface that POISE offers (see *Frontend options*) allows us to wrap POISE within a larger script. Here we present some examples of how this can be accomplished: after this, the extension to automation is largely straightforward. For example, if POISE is used inside an AU programme, then set the the `AUNM` parameter in TopSpin appropriately.

5.1 Basics

To incorporate POISE in an AU programme, you can use the syntax:

```
XCMD("sendgui xpy poise <routine_name> [options]")
```

inside the AU script.

(Optional: To suppress POISE's final popup (telling the user that the optimum has been found), you can add the `-q` flag in the options. The popup won't stop TopSpin from running whatever it was going to run, though, so it's completely safe to show the message.)

Alternatively, you can wrap POISE within a Python script, which is arguably easier to write. The corresponding syntax for running an optimisation would be:

```
XCMD("xpy poise <routine_name> [options]")
```

As you can see, it is the same except that `sendgui` isn't needed.

5.2 A simple(ish) example

Here's an example of how the `p1` optimisation (shown in *Setting up a Routine*) can be incorporated into an AU script (download from here). This AU script performs a very similar task to the existing `pulsecal` script: it finds the best value of `p1` and plugs it back into the current experiment. However, as we wrote in the paper, it tends to provide a much more accurate result. In practice, we've already used it many times to calibrate `p1` before running other experiments.

```
// Use EXPNO 99999 in the current folder for optimisation.
GETCURDATA
int old_expno = expno;
DATASET(name, 99999, procno, disk, user)
// Set some key parameters. Notice that these lines can be substantially cut
// if an appropriate parameter set is set up beforehand.
RPAR("PROTON", "all")
GETPROSOL
```

(continues on next page)

(continued from previous page)

```
STOREPAR("PULPROG", "zg")
STOREPAR("NS", 1)
STOREPAR("DS", 0)
STOREPAR("D 1", 1.0)
STOREPAR("RG", 1)
// Run optimisation.
XCMD("sendgui xpy poise p1cal -a bobyqa -q")
// POISE stores the optimised value in p1 after it's done. We can retrieve it
// here. Don't try to get the *status* parameter, since that is not the
// optimised value (it is the value used for the last function evaluation!)
float p1opt;
FETCHPAR("P 1", &p1opt)
p1opt = p1opt/4;
// Move back to old dataset and set p1 to optimised value.
DATASET(name, old_expno, procno, disk, user)
VIEWDATA_SAMEWIN  // not strictly necessary, just re-focuses the original spectrum
STOREPAR("P 1", p1opt)
Proc_err(INFO_OPT, "Optimised value of p1: %.3f", p1opt);
// (Optional) Run acquisition.
// ZG
QUIT
```

Note that all six lines underneath "set some key params" can be collapsed to one line if an appropriate parameter set is set up beforehand.

Here's the Python equivalent of the AU programme above (download from here):

```
# Use EXPNO 99999 in the current folder for optimisation.
old_dataset = CURDATA()
opt_dataset = CURDATA()
opt_dataset[1] = "99999"
# Create new dataset and move to it.
NEWDATASET(opt_dataset, None, "PROTON")
RE(opt_dataset)
# Set some key parameters. Notice that these lines can be cut if an appropriate
# parameter set is set up beforehand (and loaded using NEWDATASET()).
XCMD("getprosol")
PUTPAR("PULPROG", "zg")
PUTPAR("NS", "1")
PUTPAR("DS", "0")
PUTPAR("D 1", "1")
PUTPAR("RG", "1")
# Run optimisation.
XCMD("poise p1cal -a bobyqa -q")
# POISE stores the optimised value in p1 after it's done. We can retrieve it
# here. Don't try to get the *status* parameter, since that is not the
# optimised value (it is the value used for the last function evaluation!)
p1opt = float(GETPAR("P 1"))/4
# Move back to old dataset and set p1 to optimised value.
RE(old_dataset)
PUTPAR("P 1", str(p1opt))
ERRMSG("Optimised value of p1: {:.3f}".format(p1opt))
# A TopSpin quirk: using MSG() will block subsequent commands until user hits
# "OK". You can use ERRMSG(), as is done here. There are other ways around
# this, see MSG_nonmodal() in the POISE frontend script.
# (Optional) Run acquisition.
# ZG()
```

5.3 A helpful trick

POISE itself is a Python programme which calls an AU programme. When incorporating POISE inside another script, it can become *very* difficult to terminate the entire thing as there are several nested loops in which scripts are being run. So, for example, you can kill the top-level script using TopSpin's `kill` command. However, that won't kill POISE itself, and so it will keep acquiring spectra, etc.

(If anybody has a better idea please let us know. As far as we can tell, this is a necessary limitation of the TopSpin ecosystem.)

Anyway, we deal with this using a trick involving the `TI` TopSpin parameter, which can be any arbitrary string. POISE, upon successful function evaluation, will store the value of the cost function in the `TI` parameter. If it doesn't successfully run (for example if a requested routine or cost function is not found, or some other error), then `TI` will be left untouched.

In order to detect when POISE fails from the top-level script, we therefore:

1. Set `TI` to be blank. Please read the note below.

Note: In an AU or Python programme, you have to set `TI` to be a non-empty string that contains only whitespace. For example:

```
PUTPAR("TI", " ")    # this will work
PUTPAR("TI", "")     # this will NOT work!
```

TopSpin mangles empty strings: instead of putting an empty string in, it puts the string `"0"` in. On the other hand, if the string is not empty but contains whitespace, TopSpin automatically trims it to an empty string after it's been put in. I don't know why. The same applies to the `STOREPAR` macro in AU programmes.

---

2. Run POISE.
3. Check if `TI` is an empty string. If it is, then quit unceremoniously with an error message of your choice.

You can see this strategy in action in the first part of the DOSY optimisation script (where we optimise the diffusion delay):

```
msg_nonmodal("dosy_opt: optimising diffusion delay...")
XCMD("poise_1d", wait=WAIT_TILL_DONE)

# Set up optimisation expno.
optimisation_expno = 99999
optimisation_dataset = list(original_2d_dataset)    # make a copy
optimisation_dataset[1] = str(optimisation_expno)
NEWDATASET(optimisation_dataset, None, "dosytemp")
RE(optimisation_dataset)
PUTPAR("PULPROG", pp1d)
PUTPAR("PARMODE", "1D")
PUTPAR("PPARMOD", "1D")
```

You don't actually have to set it to be blank, of course: it can be any sentinel value you like, as long as it cannot be confused with the value of a cost function. For example, you could set `TI` to be the string `"ILoveTopSpin"` before the optimisation, then after that check whether it is still `"ILoveTopSpin"`.

CHAPTER

SIX

BUILTIN COST FUNCTIONS

POISE comes with a few, very basic, builtin cost functions. These largely mirror those that are in TopSpin's native `popt` screen.

Note: Just like in `popt`, it is possible to use the `dpl` command in TopSpin to select a *portion* of the spectrum to be optimised. This stores the left and right region of the currently active view to the parameters `F1P` and `F2P` respectively. This works for all the builtin cost functions except for `noe_1d`.

More generally, any cost function that uses any of the `get1d` or `get2d` functions will respect the bounds placed in `F1P` and `F2P`. See *Custom cost functions* for a more in-depth explanation.

6.1 minabsint

Seeks to minimise the intensity of the magnitude-mode spectrum. The intensity is measured by integration of the entire spectral region, i.e. summation of every point.

Note that this is different from the `MAGMIN` criterion in `popt`, which (from what the Bruker documentation suggests) seeks to minimise the highest point in the magnitude-mode spectrum.

6.2 maxabsint

Seeks to maximise the intensity of the magnitude-mode spectrum.

6.3 minrealint

Seeks to minimise the intensity of the real spectrum (this is probably equivalent to `INTMIN` in `popt`).

Note that this does *not* behave in the same way as `minabsint`. Because the real spectrum can have negative peaks, this essentially tries to maximise the intensity of negative peaks.

6.4 maxrealint

Seeks to maximise the intensity of the real spectrum (equivalent to `INTMAX`).

6.5 zerorealint

Seeks to make the intensity of the real spectrum as close as possible to zero (equivalent to `ZERO`).

6.6 noe_1d

Seeks to minimise the intensity of the spectrum, *except* for a region of 50 Hz centred on the parameter `SPOFFS2` (which corresponds to the frequency of the selective pulse).

Since NOE crosspeaks are typically negative (and `apk` typically phases them to be so), this essentially seeks to maximise the intensity of the crosspeaks.

CHAPTER

SEVEN

BUILTIN AU PROGRAMMES

POISE is bundled with two basic AU programmes for 1D and 2D spectra respectively. These are mostly self-explanatory, so the text is just given below. Of course, you can write your own AU programmes to be used with POISE routines.

These are the “default” AU programmes that POISE uses (for 1D and 2D spectra respectively) if an AU programme is not specified with the routine.

7.1 poise_1d

```
ZG
EFP
APBK
QUIT
```

The `APBK` command on older versions of TopSpin falls back to `APK` then `ABS`, so can be safely used.

7.2 poise_2d

```
ZG
XFB
XCMD("apk2d")
ABS2
QUIT
```

CHAPTER

EIGHT

CUSTOM COST FUNCTIONS

All user-defined cost functions are stored inside the file:

```
$TS/exp/stan/nmr/py/user/poise_backend/costfunctions_user.py
```

where `$TS` is your TopSpin installation path. In order to modify or add cost functions, you will need to edit this file (with your favourite text editor or IDE).

The corresponding file containing builtin cost functions is `costfunctions.py`. You *can* edit this file directly: if you add a cost function there, it will work. However, there are two risks with this. Firstly, if you ever reinstall POISE, this file will be reset to the default (whereas `costfunctions_user.py` will not). Secondly, any cost functions defined in `costfunctions_user.py` will shadow (i.e. take priority over) the cost functions defined in `costfunctions.py` if they have the same name.

8.1 The rules for cost functions

Cost functions are defined as a standard Python 3 function which takes no parameters and returns a float (the value of the cost function).

Do write a useful docstring if you can: this docstring will be shown to the user when they type `poise -l` (which lists all available cost functions and routines).

Also, you should *never* print anything inside a cost function directly to `stdout`. That will cause the optimisation to stop. If you want to do some debugging, read on — there's a function for that.

That's it!

Of course, it's quite useless saying that without telling you how to (for example) access the spectrum that's being optimised. The way this is done is by using several variables inside the class `_g`, which is imported from `shared.py`. These global variables provide information about the current optimisation. For example, using `_g.p_spectrum` you can find the path of the real spectrum, then parse the file to get the spectrum as a `numpy.ndarray` (for example).

class `nmrpoise.poise_backend.shared.`**`_g`**

Class to store the "global" variables.

Attributes

optimiser [str from {'nm', 'mds', 'bobyqa'}] The optimiser being used.

routine_id [str] The name of the routine being used.

p_spectrum [`Path`] The path to the procno folder of the spectrum just acquired. (e.g. `/path/to/data/1/pdata/1`)

p_optlog [`Path`] The path to the currently active `poise.log` file.

p_errlog [Path] The path to the currently active `poise_err_backend.log` file.

maxfev [int] The maximum number of function evaluations specified by the user. Can be zero, indicating no limit (beyond the hard limit of 500 times the number of parameters).

p_poise [Path] The path to the `$TS/exp/stan/nmr/py/user/poise_backend` folder.

spec_f1p [float or tuple of float] The `F1P` parameter. For a 1D spectrum this is a float. For a 2D spectrum this is a tuple of floats (`indirect, direct`) corresponding to the values of `F1P` in both spectral dimensions.

spec_f2p [float or tuple of float] The `F2P` parameter.

8.2 Helper methods

If you've looked inside the `costfunctions.py` file, you've probably realised that none of them actually use these variables directly. Instead, we have a bunch of helper methods that use these to get more useful information directly. All of these methods are stored inside `cfhelpers.py` and are already imported.

The ones you are likely to use are the following:

`nmrpoise.poise_backend.cfhelpers.`**`make_p_spec`**`(`*path=None*, *expno=None*, *procno=None*`)`

Constructs a `Path` object corresponding to the procno folder `<path>/<expno>/pdata/<procno>`. If parameters are not passed, they are inherited from the currently active spectrum (_g.p_spectrum).

Thus, for example, `make_p_spec(expno=1, procno=1)` returns a path to the spectrum with EXPNO 1 and PROCNO 1, but with the same name as the currently active spectrum.

Parameters

path [str or Path, optional] Path to the main folder of the spectrum (one level above the expno folders).

expno [int, optional]

procno [int, optional]

Returns

p_spec [Path] Path pointing to the requested spectrum.

`nmrpoise.poise_backend.cfhelpers.`**`get1d_fid`**`(`*p_spec=None*`)`

Returns the FID as a `ndarray`.

Note that this does *not* modify the "group delay" at the beginning of the FID.

Also, Bruker spectrometers record real and imaginary points in a sequential fashion. Therefore, each imaginary point in the ndarray is actually measured `DW` *after* the corresponding real point. When Fourier transforming, this can be accounted for by using fftshift().

Parameters

p_spec [Path, optional] Path to the procno folder of interest. (The FID is taken from the expno folder two levels up.) Leave blank to use the currently active spectrum (i.e. _g.p_spectrum).

Returns

ndarray Complex-valued array containing the FID.

nmrpoise.poise_backend.cfhelpers.**get1d_real**(*bounds=''*, *p_spec=None*)

Return the real spectrum as a `ndarray`. This function accounts for TopSpin's `NC_PROC` variable, scaling the spectrum intensity accordingly.

Note that this function only works for 1D spectra. It does *not* work for 1D projections of 2D spectra. If you want to work with projections, you can use *get2d_rr* to get the full 2D spectrum, then manipulate it using numpy functions as appropriate. Examples can be found in the docs.

The *bounds* parameter may be specified in the following formats:

- between 5 and 8 ppm: `bounds="5..8"` OR `bounds=(5, 8)`
- greater than 9.3 ppm: `bounds="9.3.."` OR `bounds=(9.3, None)`
- less than -2 ppm: `bounds="..-2"` OR `bounds=(None, -2)`

Parameters

bounds [str or tuple, optional] String or tuple describing the region of interest. See above for examples. If no bounds are provided, uses the `F1P` and `F2P` processing parameters, which can be specified via `dpl`. If these are not specified, defaults to the whole spectrum.

p_spec [`Path`, optional] Path to the procno folder of interest. Leave blank to use the currently active spectrum (i.e. _g.p_spectrum).

Returns

`ndarray` Array containing the spectrum or the desired section of it (if bounds were specified).

nmrpoise.poise_backend.cfhelpers.**get1d_imag**(*bounds=''*, *p_spec=None*)

Same as *get1d_real*, except that it reads the imaginary spectrum.

nmrpoise.poise_backend.cfhelpers.**get2d_rr**(*f1_bounds=''*, *f2_bounds=''*, *p_spec=None*)

Return the real part of the 2D spectrum (the "RR" quadrant) as a 2D `ndarray`. This function takes into account the `NC_PROC` value in TopSpin's processing parameters.

The *f1_bounds* and *f2_bounds* parameters may be specified in the following formats:

- between 5 and 8 ppm: `f1_bounds="5..8"` OR `f1_bounds=(5, 8)`
- greater than 9.3 ppm: `f1_bounds="9.3.."` OR `f1_bounds=(9.3, None)`
- less than -2 ppm: `f1_bounds="..-2"` OR `f1_bounds=(None, -2)`

Parameters

f1_bounds [str or tuple, optional] String or tuple describing the indirect-dimension region of interest. See above for examples. If no bounds are provided, uses the `1 F1P` and `1 F2P` processing parameters, which can be specified via `dpl`. If these are not specified, defaults to the whole spectrum.

f2_bounds [str or tuple, optional] String or tuple describing the direct-dimension region of interest. See above for examples. If no bounds are provided, uses the `2 F1P` and `2 F2P` processing parameters, which can be specified via `dpl`. If these are not specified, defaults to the whole spectrum.

p_spec [`Path`, optional] Path to the procno folder of interest. Leave blank to use the currently active spectrum (i.e. _g.p_spectrum).

Returns

ndarray 2D array containing the spectrum or the desired section of it (if *f1_bounds* or *f2_bounds* were specified).

`nmrpoise.poise_backend.cfhelpers.`**`get2d_ri`**(*f1_bounds=''*, *f2_bounds=''*, *p_spec=None*)

Same as `get2d_rr`, except that it reads the '2ri' file.

`nmrpoise.poise_backend.cfhelpers.`**`get2d_ir`**(*f1_bounds=''*, *f2_bounds=''*, *p_spec=None*)

Same as `get2d_rr`, except that it reads the '2ir' file.

`nmrpoise.poise_backend.cfhelpers.`**`get2d_ii`**(*f1_bounds=''*, *f2_bounds=''*, *p_spec=None*)

Same as `get2d_rr`, except that it reads the '2ii' file.

`nmrpoise.poise_backend.cfhelpers.`**`getpar`**(*par*, *p_spec=None*)

Obtains the value of a numeric (acquisition or processing) parameter. Non-numeric parameters (i.e. strings) are not currently accessible! Works for both 1D and 2D spectra (see return type below), but nothing higher.

Parameters

par [str] Name of the parameter.

p_spec [`Path`, optional] Path to the procno folder of interest. Leave blank to use the currently active spectrum (i.e. _g.p_spectrum).

Returns

float or ndarray Value(s) of the requested parameter. None if the given parameter was not found.

For parameters that exist for both dimensions of 2D spectra, getpar() returns an ndarray consisting of (f1_value, f2_value). Otherwise (for 1D spectra, or for 2D parameters which only apply to the direct dimension), getpar() returns a float.

`nmrpoise.poise_backend.cfhelpers.`**`getndim`**(*p_spec=None*)

Obtains the dimensionality of the spectrum, i.e. the status value of PARMODE. Note that Bruker uses PARMODE=n for (n+1)D spectra, whereas this function simply returns (n+1) (as an int).

Note that we call _get_acqu_par() instead of getpar() to avoid an infinite loop.

Parameters

p_spec [`Path`, optional] Path to the procno folder of interest. Leave blank to use the currently active spectrum (i.e. _g.p_spectrum).

Returns

int Dimensionality of the spectrum.

`nmrpoise.poise_backend.cfhelpers.`**`log`**(*s*)

Prints a string to the poise.log file. If this is called from inside a cost function, the text is printed *before* the cost function is evaluated, so will appear above the corresponding function evaluation.

Parameters

s [object] The object to be printed. Typically a string, but since this is just passed directly to print(), anything with a __str__() method can be used.

Returns

None

8.3 Examples

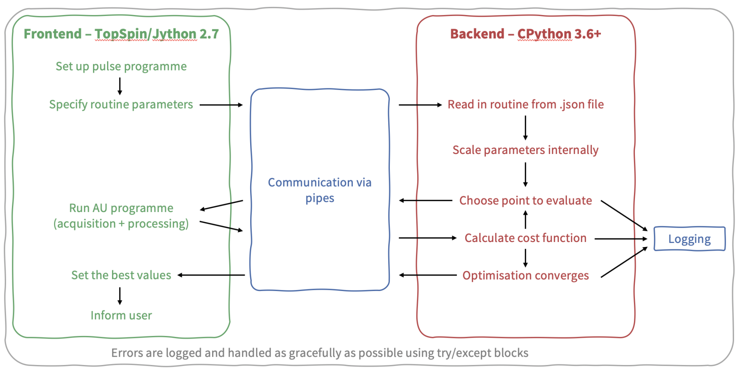
There are a number of cost functions which ship with POISE. These can all be found inside the `costfunctions.py` file referred to above. This file also contains a number of more specialised cost functions, which were used for the examples in the POISE paper. (Instead of opening the file, you can also find the source code on GitHub.)

A number of these are thoroughly commented with detailed explanations; do consider checking these out if you want more guidance on how to write your own cost function.

CHAPTER NINE

DEVELOPER NOTES

POISE does not contain all that much code. However, if you are thinking of modifying it, it helps to have an understanding of the distinction between the *frontend* and the *backend*. This graphic (taken from an old presentation) is a slightly more technical description of the flowchart presented in the POISE paper, and shows which part is responsible for which step. It's a good starting point for understanding how POISE works internally.



Beyond this, we recommend reading the source code of POISE: start at the frontend script (`poise.py`, which is run from inside TopSpin), then go to the backend script (`poise_backend/backend.py`) at the appropriate time (when the frontend launches it as a subprocess). The source code is quite thoroughly commented.

There are two main things to point out. Probably the most important thing worth mentioning is the location of the relevant files. The backend script is always ran from `$TS/py/user/poise_backend` (putting it here allows the frontend to access it much more easily). The entire `$TS/py/user/poise_backend` folder is treated as if it is a Python package, by virtue of some code near the top of `poise_backend.py`:

```
if __name__ == "__main__" and __package__ is None:
    __package__ = "poise_backend"
    sys.path.insert(1, str(Path(__file__).parents[1].resolve()))
    __import__(__package__)
```

This allows relative imports of the other files in the same directory, such as `costfunctions.py`, in which the user-defined cost functions reside.

Note: These backend files *also* reside inside the Python 3 `site-packages` directory, where all packages are installed to. However, these files will *never* be used by POISE. So, there is nothing to be gained by modifying these at all.

The only file inside the `site-packages` directory which has any effect is `nmrpoise/__init__.py`, where the `parse_log()` function is defined. This allows you to (for example) run:

```
>>> from nmrpoise import parse_log
```

Python will look inside the `site-packages` directory, *not* `$TS/py/user`, to find this function.

The second thing is that you should never, ever, do anything with the backend's `stdin` and `stdout`, because these are exclusively reserved for communication with the frontend. So you should never print anything from the backend, since the frontend will just interpret that as an error. This applies to all files inside the `$TS/py/user/poise_backend` directory, including cost functions, which is why custom cost functions should always use POISE's *`log`* function instead of plain old `print`.

Note: If you really just want to do some quick-and-dirty debugging, you *can* actually use this behaviour to your advantage. The frontend will echo any "invalid" message it receives from the backend, so if you print some unexpected text from the backend (on purpose), you should see it pop up as a TopSpin message when you run an optimisation. This is slightly less hassle than printing to a file and opening the file.

9.1 Testing

Tests are carried out using the excellent pytest and tox tools. To run all tests, simply run:

```
tox
```

from anywhere inside the `nmrpoise` directory. This runs tests on Python 3.6, 3.7, and 3.8. If you only have one of these versions, use:

```
tox -e py38    # or py36 or py37
```

To build the Sphinx documentation, use:

```
tox -e docs
```

The HTML documentation will be built in `docs/dirhtml`, and the PDF documentation in `docs/latex` (this assumes you have a working installation of `pdflatex` on your system).