esrpoise

Jean-Baptiste Verstraete, Jonathan Yong, Mohammadali Foroozandeh

May 05, 2022

Contents

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

In here you will find guides on setting ESR-POISE up and using it to optimise ESR applications. 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 installed and set up some esrpoise scripts for you, you can probably skip to *Run an optimisation*. You should also skip advanced subsections for your first steps with esrpoise.

You can also have a look at the examples scripts present in the package directory (`esrpoise/examples/`).

For more in-depth insight, consult the source code.

Note: The documentation you are currently reading is for version v1.0.0 of POISE. To check your current version of POISE, type `pip show esrpoise` in terminal.

Chapter 1

Table of contents

1.1 Installation

The requirements are:

- **Xepr**. We have tested Xepr versions 2.8b.5
- **Python 3**. ESRPOISE requires a minimum version of **Python 3.6.** (We have tested up to Python 3.8.)

1.1.1 Installing Python 3

This is most easily done by downloading an installer from the official CPython website (https://www.python.org/downloads/), or via your package manager (`apt`, `yum` or similar).

On CentOS 7, you would typically need (creates a python3 command, independant from the already installed Python 2 called with python):

```
yum -y install python3
```

We had to also install tkinter with:

```
yum -y install python3-tkinter
```

A particular issue we faced when setting up POISE was installing Python on CentOS 7 without administrator rights. If you are in this situation, you will have to compile the Python source code, using the following steps:

1. Download the source code for Python 3.7. We used 3.7.11, but anything should work.
2. Download the source code for Tcl/Tk (at least for now, Python needs to be compiled with tkinter for POISE to run successfully).
3. Compile Tcl/Tk, making sure to install it to a user-writable directory (e.g. your home directory).
4. Compile Python, again installing it to your home directory. This step is pretty difficult as there are certain (undocumented) compilation flags which must be passed correctly.

The following series of shell commands (give or take) worked for us, so you could simply run the following:

```
# Download source code
curl -LO https://www.python.org/ftp/python/3.7.11/Python-3.7.11.tgz
curl -LO https://prdownloads.sourceforge.net/tcl/tcl8.6.12-src.tar.gz
curl -LO https://prdownloads.sourceforge.net/tcl/tk8.6.12-src.tar.gz
tar xf Python-3.7.11.tgz
tar xf tcl8.6.12-src.tar.gz
tar xf tk8.6.12-src.tar.gz
# Install tcl/tk
cd tcl8.6.12/unix
./configure --prefix=$HOME/.local/tcltk
make
make install
cd ../../tk8.6.12/unix
./configure --prefix=$HOME/.local/tcltk
make
make install
# Install Python
cd ../../Python-3.7.11/
CPPFLAGS="-I$HOME/.local/tcltk/include" LDFLAGS="-L$HOME/.local/tcltk/lib -Wl,-rpath,$HOME/.
↪local/tcltk/lib -ltcl8.6 -ltk8.6" ./configure --prefix=$HOME/.local/python3 --with-tcltk-
↪includes="-I$HOME/.local/tcltk/include" --with-tcltk-libs="-L$HOME/.local/tcltk/lib"
make
make install
```

You should then have a working installation of Python 3.7 in `~/.local/python3/bin/python`. Note that you should always use this version of Python whenever installing packages: so, it's safer to always use `/path/to/python -m pip install X` rather than just `pip install`. Naturally you can place this Python executable first in your `$PATH` in order to avoid having to type out the full path every time.

One of the tested spectrometer computer also required the installation of `libffi` and `libffi-devel` before being able to `make install`.

If there are any issues, please get in touch via GitHub or email.

1.1.2 Installing esrpoise

Once python is installed, you can install POISE using `pip`:

```
python -m pip install esrpoise
```

(replace `python` with `python3` if necessary)

The following Python packages are required, they should get automatically installed with esrpoise if you do not already have them:

- **numpy**
- **XeprAPI**
- **pybobyqa**

1.1.3 Updating POISE

Simply use:

```
python -m pip install --upgrade esrpoise
```

(again replacing `python` with `python3` if necessary). All other steps (including troubleshooting, if necessary) are the same.

1.1.4 Installing 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 `esrpoise` directory and run:

```
python -m pip install .
```

Add `` -e`` to be able to edit the code. Equivalently you can run:

```
python -m setup.py install
```

Replace `install` instead with `develop` to edit the code.

(use `python3` if necessary)

1.1.5 Installing without Internet

If the computer you are using does not have an Internet connection, then you will need to:

1. Download the POISE source code from GitHub: `git clone https://github.com/foroozandehgroup/esrpoise` and copy it over to the target computer.
2. Install Python by downloading the installer from a different computer and copying it over.
3. On the target computer, install the POISE package locally by navigating to the `esrpoise` directory you copied over and doing `python -m pip install .` (note the full stop at the end).

1.2 Set up an optimisation

To set up an optimisation, you can to create a small python script which:

- loads an instance Xepr of the XeprAPI,
- calls the function optimise from esrpoise.

1.2.1 with standard parameters

When calling `optimise()`, you should to at least pass:

- the Xepr instance,
- your parameters names, initial values, lower bounds and upper bounds as lists,
- a cost function, which can be imported from the pre-existing ones (cf. *costfunctions.py*),
- the maximum number of function evaluations (not technically mandatory but set to `0` by default).

Only a handful of Xepr parameters can be accessed through a simple optimisation (cf. *Built-in parameters*).

Example of standard parameter optimisation script:

```
from esrpoise import xepr_link
from esrpoise import optimise
from esrpoise.costfunctions import maxabsint_echo

# load Xepr instance
xepr = xepr_link.load_xepr()

# fine adjustment of centre field for bisnitroxide sample at X-band
xbest, fbest, message = optimise(xepr,
                                 pars=['CenterField'],
                                 init=[3450],
                                 lb=[3445],
                                 ub=[3455],
                                 tol=[0.1],
                                 cost_function=maxabsint_echo,
                                 maxfev=20)
```

Note that `xepr`, `pars`, `init`, `lb`, `ub`, `tol` need to be input in this order, all other parameters are keywords arguments whose order does not matter.

> **Warning:** Make sure to chose an appropriate tolerance for your instrumentation. For example, delays and pulse duration need to be multiple of 2ns on our instrument. We therefore must choose a tolerance which is a multiple of 2ns when optimising those parameters.

1.2.2 with .def file parameters

Optimise parameters from your .def file by indicating their names. In addition to the standard parameter requiremer

- the path of your .exp file,
- the path of your .def file.

Example (also note the use of several parameters in lists and the optimiser explicit choice):

```
# .exp and .def files location
location = '/home/xuser/xeprFiles/Data/'
exp_f = location + '2pflip.exp'
def_f = location + '2pflip.def'
```

(continues on next page)

(continued from previous page)

```
# optimisation of pulse length and amplitude
xbest, fbest, message = optimise(xepr,
                                 pars=["p0", "Attenuation"],
                                 init=[8, 5],
                                 lb=[2, 0],
                                 ub=[36, 10],
                                 tol=[2, 0.5],
                                 cost_function=maxabsint_echo,
                                 exp_file=exp_f,
                                 def_file=def_f,
                                 optimiser="bobyqa",
                                 maxfev=20)
```

1.2.3 with user-defined parameters (advanced)

To optimise your own parameters, you should define a callback function, used to set up your parameters. The simplest is to define it in your optimisation script (after the imports and before your script code):

```
def my_callback(my_callback_pars_dict, *mycallback_args):
    """
    Parameters
    ----------
    pars_dict: dictionary
        dictionary with the name of the parameters to optimise as key and
        their value as value
    *mycallback_args
        other possible arguments for callback function
    """
    # user operations and parameters modifications
```

Indicate that your parameter is user-defined by including '&' at the start of its name:

```
optimise(Xepr, pars=[... , '&_', ...], ...,
                callback=my_callback, callback_args=my_callback_args)
```

You can pass arguments (as a tuple) to `callback` by using `callback_pars`.

In the following example, we modify a shape parameter with the module `mrpypulse` (bandwidth `bw` of an HS1 pulse)

```
import os
from esrpoise import xepr_link
from esrpoise import optimise
from esrpoise.costfunctions import maxabsint_echo
from mrpypulse import pulse


def shape_bw(callback_pars_dict, shp_nb):

    # getting  bw value from the callback parameters to be optimised
    bw = callback_pars_dict["&bw"]
```

(continues on next page)

(continued from previous page)

```
    # create hyperbolic sechant shape bw value
    p = pulse.Parametrized(bw=bw, tp=80e-9, Q=5, tres=0.625e-9,
                           delta_f=-65e6, AM="tanh", FM="sech")

    p.xepr_file(shp_nb)    # create shape file

    # shape path
    path = os.path.join(os.getcwd(), str(shp_nb) + '.shp')

    xepr_link.load_shp(xepr, path)  # send shape to Xepr

    return None


xepr = xepr_link.load_xepr()

#  HS pulse bandwidth optimisation
xbest, fbest, message = optimise(xepr,
                                 pars=['&bw'],
                                 init=[80e6],
                                 lb=[30e6],
                                 ub=[120e6],
                                 tol=[1e6],
                                 cost_function=maxabsint_echo,
                                 maxfev=120,
                                 nfactor=5,
                                 callback=shape_bw,
                                 callback_args=(7770,))
# NB: '(7770,)' is equivalent to 'tuple([7770])'
```

Note that we first have to import the modules, then defining the function before writing the actual optimisation script.

1.2.4 with user-defined cost function (advanced)

You can define your own cost function and pass it to the function optimise. Your cost function should treat the data from one of your experiment run to return a single number which will be minimised by the optimiser.

We can for example conduct an optmisation on the spectrum with a zero-filling operation (data is here a simple time-domain FID):

```
import numpy
from esrpoise import xepr_link
from esrpoise import optimise

def maxabsint(data):
    """
    Maximises the absolute (magnitude-mode) intensity of the spectrum.
    """
    zero_filling = 4*length(data.0.real)
    spectrum = np.fft.fft(data.0.real + 1j * data.0.imag, n=zero_filling)
```

(continues on next page)

(continued from previous page)

```
    return -np.sum(np.abs(spectrum(data)))


# load Xepr instance
xepr = xepr_link.load_xepr()

# fine adjustment of center field for bisnitroxide sample at X-band
xbest, fbest, message = optimise(xepr,
                                 pars=['CenterField'],
                                 init=[3450],
                                 lb=[3445],
                                 ub=[3455],
                                 tol=[0.1],
                                 cost_function=maxabsint,
                                 maxfev=20)
```

1.2.5 Setup Tips (advanced)

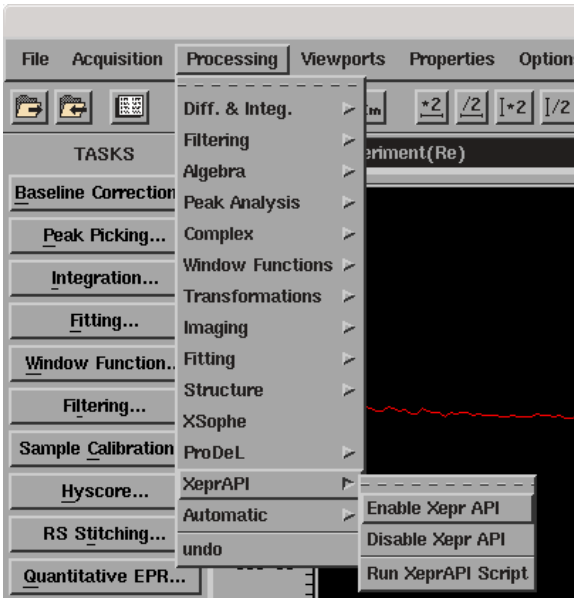
- Put several optimisations in one script.
- Automate your actions by using XeprAPI commands, the functions from *xepr_link.py* and `param_set` from *main.py*
- Reuse the best parameter from the optimiser `xbest`.
- Use `callbak` to add user-specific operation at each iteration. You do not need to indicate user-defined parameters, `callback_pars_dict` is sent back empty if no user-defined parameters are found.
- Use `acquire_esr.calls` in your callback function to access the current number of your iteration.
- Use the parameter `nfactor` of `optimise()` to expand the distance between the first steps of the optimisers, in particular if you have a low tolerance.
- Accelerate you optimisation routine if your .shp, .def and .exp file compile fast enough with `xepr_link.COMPILATION_TIME` (cf. *Compilation*)
- When using a single script with functions, be aware of your variables scope.

1.3 Run an optimisation

1.3.1 Procedure

In Xepr:

- Set up your experiment as you would normally do.



- **Enable XeprAPI (Processing → XeprAPI → enable XeprAPI).**
- Run your experiment once by clicking on play to load it into XeprAPI (can also be done before enabling XeprAPI).

Run your optimisation script in a terminal with

```
python yourscript.py
```

(replace `python` with `python3` if necessary)

At each iteration, esrpoise:

- adjusts the parameters to be optimised
- runs the experiment
- computes the cost function.

Once `optimise()` is done, you get a set of optimised parameters. These are set up in Xepr but your experiment is not run. You need to run it if you want to have the optimised result loaded in Xepr.

1.3.2 Running tips

- You can interupt the optimisation with `ctrl+C`. It is recommend to do so while a measurement is running.
- When using .exp and .def file, save them beforehand to avoid getting a warning. Otherwise this warning pauses the optimisation and you need to manually get rid of it at each iteration.
- When running your experiment once to load it in XeprAPI, you can interupt it by clicking on play again. This is enough to set it up, sparing the time of a full run.

1.4 Built-in parameters

Built-in parameters can have different constraints depending on your spectrometer. Values documented here are just indications. All parameters currently supported are indicated below.

1.4.1 Bridge - Receiver Unit

- `"VideoGain"` (dB), video gain, indicative minimum tolerances: 3dB, 6dB
- `"Attenuation"` (dB), high power attenuation, indicative minimum tolerance: 0.01dB
- `"SignalPhase"` (~0.129deg), signal phase, indicative minimum tolerance: 1
- `"TMLevel"` (%), transmitter level, 0 to 100

1.4.2 Bridge - MPFU control

- `"BrXPhase"`, +<x> Phase
- `"BrXAmp"`, +<x> Amplitude
- `"BrYPhase"`, +<y> Phase
- `"BrYAmp"`, +<y> Amplitude
- `"BrMinXPhase"`, -<x> Phase
- `"BrMinXAmp"`, -<x> Amplitude
- `"BrMinYPhase"`, -<y> Phase
- `"BrMinYAmp"`, -<y> Amplitude

These are rounded in Xepr to closest 0.049% (approximately, not linear).

1.4.3 FT EPR Parameters

- `"CenterField"`, field position (G), indicative minimum tolerance: 0.05G

1.5 Modules

The esrpoise code is organized into 4 modules:

- `main.py` for the optimisation to take place (set parameters, run the experiment, report results...),
- `xepr_link.py` to handle communication with Xepr,
- `costfunctions.py` which contains standard cost functions,
- `optpoise.py` which contains the necessary for the optimisers (cf. source code for more details).

1.5.1 main.py

Contains the main code required for executing an optimisation.

SPDX-License-Identifier: GPL-3.0-or-later

esrpoise.main.**optimise**(*xepr*, *pars*, *init*, *lb*, *ub*, *tol*, *cost_function*, *exp_file=None*, *def_file=None*, *optimiser='bobyqa'*, *maxfev=0*, *nfactor=10*, *callback=None*, *callback_args=None*)

Run an optimisation.

Parameters

xepr [instance of XeprAPI.Xepr] The instantiated Xepr object.

pars [list of str] Parameter names. Parameters starting with the character & are considered user parameters which needs to be modified with the callback function.

init: list of float Initial value for each parameter.

lb [list of float] Lower bounds for each parameter.

ub [list of float] Upper bounds for each parameter.

tol [list of float] Optimisation tolerances for each parameter.

cost_function [function] A function which takes the data object and returns a float.

exp_file [str, default None] Experiment file (.exp) path to be used for the experiment in Xepr. Required to modify parameters in .def file.

def_file [str, default None] Definition file (.exp) path to be used for the experiment in Xepr. Required to modify parameters in .def file.

optimiser [str from {“nm”, “mds”, “bobyqa”, “brute”}, default “nm”] Optimisation algorithm to use. The options correspond to Nelder-Mead, multidimensional search, BOBYQA, and brute-force search respectively.

maxfev [int, default 0] Maximum number of spectra to acquire during the optimisation. The default of ‘0’ sets this to 500 times the number of parameters.

nfactor [int, default 10] Initial search region relative to tols

callback [function, default None] User defined function called when setting up parameters.

callback_args [tuple, default None] Arguments for callback function

Returns

xbest [numpy.ndarray] Numpy array of best values found. First element corresponds to the first parameter optimised, etc.

fbest [float] Value of the cost function at x = xbest.

message [str] A message indicating why the optimisation terminated.

Notes

To quit the optimisation, simply type 'ctlr+C' in the terminal. It is recommended to do so during an acquisition phase of Xepr to avoid Xepr crashes.

Note once the optimisation is done, the best parameters found are set up in Xepr but the experiment is not run.

esrpoise.main.**acquire_esr**(*x*, *cost_function*, *pars*, *lb*, *ub*, *tol*, *optimiser*, *xepr*, *exp_file=None*, *def_file=None*, *callback=None*, *callback_args=None*)

This is the function which is actually passed to the optimisation function as the "cost function", and is responsible for triggering acquisition in Xepr.

Briefly, this function does the following:

- Returns np.inf immediately if the values are outside the given bounds.
- Otherwise, prints the values to stdout, which triggers acquisition by the frontend.
- Waits for the frontend to pass the message "done" back, indicating that the spectrum has been acquired and processed.
- Calculates the cost function using the user-defined cost_function().
- Performs logging throughout.

Parameters

x [ndarray] Scaled values to be used for spectrum acquisition and cost function evaluation.

pars [list of str] Parameter names. Parameters starting with the character & are considered user parameters which needs to be modified with the callback function.

lb [list of float] Lower bounds for each parameter.

ub [list of float] Upper bounds for each parameter.

tol [list of float] Optimisation tolerances for each parameter.

optimiser [str from {"nm", "mds", "bobyqa", "brute"}, default "nm"] Optimisation algorithm to use. The options correspond to Nelder-Mead, multidimensional search, BOBYQA, and brute-force search respectively.

xepr [instance of XeprAPI.Xepr TODO doc] The instantiated Xepr object.

cost_function [function] A function which takes the data object and returns a float.

exp_file [str, default None] Experiment file (.exp) path to be used for the experiment in Xepr. Required to modify parameters in .def file.

def_file [str, default None] Definition file (.exp) path to be used for the experiment in Xepr. Required to modify parameters in .def file.

callback [function, default None] User defined function called when setting up parameters.

callback_args: tuple, default None Arguments for callback function

Returns

cf_val [float] Value of the cost function.

esrpoise.main.**param_set**(*xepr*, *pars*, *val*, *tol*, *exp_file=None*, *def_file=None*, *callback=None*, *callback_args=None*)

Set a variety of parameters in Xepr.

Parameters

xepr [XeprAPI.Xepr object] The instantiated Xepr object.

pars [list of str] Parameter names. Parameters starting with the character & are considered user parameters which needs to be modified with the callback function.

val [list of floats or ndarray] values of the parameters

tol [list of float] Optimisation tolerances for each parameter.

exp_file [str, default None] Experiment file (.exp) path to be used for the experiment in Xepr. Required to modify parameters in .def file.

def_file [str, default None] Definition file (.exp) path to be used for the experiment in Xepr. Required to modify parameters in .def file.

callback [function, default None] User defined function called when setting up parameters.

callback_args [tuple, default None] Arguments for callback function

Returns

None

esrpoise.main.**round2tol_str**(*values*, *tols*)

Round values to closest multiple of tolerance.

Parameters

values [list or ndarray] values to round

tol [list or ndarray] values tolerances

Returns

values_str [] rounded values as a list of strings

1.5.2 xepr_link.py

esrpoise.xepr_link.**load_xepr**()

Initialise the Xepr module and return the Xepr object thus created.

Parameters

None

Returns

Xepr [instance of XeprAPI.Xepr] The instantiated Xepr object, used for communication with Xepr-the-programme.

esrpoise.xepr_link.**load_exp**(*xepr*, *exp_file*)

Load and compile an Xepr experiment file.

Parameters

xepr [XeprAPI.Xepr object] The instantiated Xepr object.

exp_file [str] The Xepr experiment file (the full path with .exp extension).

Returns

None

esrpoise.xepr_link.**load_def**(*xepr*, *def_file*)

Load and compile an Xepr definition file.

Parameters

xepr [XeprAPI.Xepr object] The instantiated Xepr object.

def_file [str] Name of the Xepr definition file (full path with .def extension).

Returns

None

esrpoise.xepr_link.**modif_def**(*xepr*, *def_file*, *var_name*, *var_value*)

Modify definitions by modifying the .def file and reloading it.

Parameters

xepr [XeprAPI.Xepr object] The instantiated Xepr object.

def_file [str] Name of the Xepr definition file (full path with .def extension).

var_name [list of strings] Variable names as named in the .def file.

var_value [list of strings] List of variable values to be input.

Returns

None

esrpoise.xepr_link.**load_shp**(*xepr*, *shp_file*)

Load and compile an Xepr shape file.

Parameters

xepr [XeprAPI.Xepr object] The instantiated Xepr object.

shp_file [str] Shape file name (full path, including .shp extension).

Returns

None

esrpoise.xepr_link.**run2getdata_exp**(*xepr*, *SignalType=None*, *exp_name=None*)

Run an experiment and get the data from it.

Parameters

xepr [XeprAPI.Xepr object] The instantiated Xepr object.

SignalType [str, by default None] To change the signal type (pick from {"TM", "Signal", "RM"})

exp_name [str, by default None] The name of the experiment to run, usually "AWG-Transient" for the Signal or "AWG_1pulseSHP" for the TM.

Returns

data [XeprAPI.Dataset]

The data retrieved from Xepr. Attributes of interest are:

- data.X : the X-abscissa values of the dataset
- data.O : complex-valued numpy.ndarray of the signal
- data.O.real : real part of the signal
- data.O.imag : imaginary part of the signal

esrpoise.xepr_link.**reset_exp**(*xepr*)

Copy the current experiment and use it to replace the current experiment.

Needed to reset the AWG after 114 sequential shape load and run.

Parameters

xepr [XeprAPI.Xepr object] The instantiated Xepr object.

Returns

None

1.5.3 costfunctions.py

Spectrum

esrpoise.costfunctions.**spectrum**(*data*)

Computes a spectrum from Xepr standard time domain data.

Parameters

data [XeprAPI.Dataset] data retrieved from Xepr

Returns

spectrum [ndarray] spectrum computed from time domain data

esrpoise.costfunctions.**minabsint**(*data*)

Minimises the absolute (magnitude-mode) intensity of the spectrum.

esrpoise.costfunctions.**maxabsint**(*data*)

Maximises the absolute (magnitude-mode) intensity of the spectrum.

esrpoise.costfunctions.**minrealint**(*data*)

Minimises the intensity of the real part of the spectrum. If the spectrum has negative peaks this cost function will try to maximise those.

esrpoise.costfunctions.**maxrealint**(*data*)

Maximises the intensity of the real part of the spectrum.

esrpoise.costfunctions.**minimagint**(*data*)

Minimises the intensity of the imaginary part of the spectrum. If the spectrum has negative peaks this cost function will try to maximise those.

esrpoise.costfunctions.**maximagint**(*data*)

Maximises the intensity of the real part of the spectrum.

esrpoise.costfunctions.**zerorealint**(*data*)

Tries to get the intensity of the real spectrum to be as close to zero as possible. This works by summation, so dispersion-mode peaks will not contribute to this cost function (as they add to zero).

esrpoise.costfunctions.**zeroimagint**(*data*)

Tries to get the intensity of the imaginary spectrum to be as close to zero as possible.

Echo

esrpoise.costfunctions.**minabsint_echo**(*data*)

Minimises the absolute (magnitude-mode) intensity of the echo.

esrpoise.costfunctions.**maxabsint_echo**(*data*)

Maximises the absolute (magnitude-mode) intensity of the echo.

esrpoise.costfunctions.**minrealint_echo**(*data*)

Minimises the maximum intensity of the real part of the echo.

esrpoise.costfunctions.**maxrealint_echo**(*data*)

Maximises the maximum intensity of the real part of the echo.

esrpoise.costfunctions.**minimagint_echo**(*data*)

Minimises the maximum intensity of the imaginary part of the echo.

esrpoise.costfunctions.**maximagint_echo**(*data*)

Maximises the maximum intensity of the real part of the echo.

esrpoise.costfunctions.**zerorealint_echo**(*data*)

Tries to get the intensity of the real part of the echo to be as close to zero as possible.

esrpoise.costfunctions.**zeroimagint_echo**(*data*)

Tries to get the intensity of the imaginary part of the echo to be as close to zero as possible.

esrpoise.costfunctions.**maxrealint_plus_zeroimagint_echo**(*data*)

Tries to get the intensity of the imaginary part of the echo to be as close to zero as possible.

esrpoise.costfunctions.**minabsmax_echo**(*data*)

Minimises the absolute (magnitude-mode) maximum of the echo.

esrpoise.costfunctions.**maxabsmax_echo**(*data*)

Maximises the absolute (magnitude-mode) maximum of the echo.

esrpoise.costfunctions.**minrealmax_echo**(*data*)

Minimises the maximum of the real part of the echo.

esrpoise.costfunctions.**maxrealmax_echo**(*data*)

Maximises the maximum of the real part of the echo.

esrpoise.costfunctions.**minimagmax_echo**(*data*)

Minimises the maximum of the imaginary part of the echo.

esrpoise.costfunctions.**maximagmax_echo**(*data*)

Maximises the maximum of the imaginary part of the echo.

Other

esrpoise.costfunctions.**max_n2p**(*data*)

Maximises n2p parameter for DEER trace. Data should contain the 2 points of interest in position 0 and 1.

1.6 Troubleshoot

1.6.1 Compilation

Bugs can be created if `xepr_link` does not allow enough time for compilation.

Increase the compilation time of the .exp, .def and .shp files (1s by default) with the global variable `COMPILATION_TIME` from `Xepr_link`:

```
xepr_link.COMPILATION_TIME = 2  # (s)

xepr = xepr_link.load_xepr()

# .exp and .def files location
location = '/home/xuser/xeprFiles/Data/'
exp_f = location + '2pflip.exp'
def_f = location + '2pflip.def'

# optimisation of pulse length and amplitude
xbest, fbest, message = optimise(xepr,
                                 pars=["p0", "Attenuation"],
                                 init=[8, 5],
                                 lb=[2, 0],
                                 ub=[36, 10],
                                 tol=[2, 0.5],
                                 cost_function=maxabsint_echo,
                                 exp_file=exp_f,
                                 def_file=def_f,
                                 optimiser="bobyqa",
                                 maxfev=20)

# run experiment with optimal parameters
xepr_link.run2getdata_exp(xepr, "Signal", exp_f)
```

When decreased, `COMPILATION_TIME` can be used to accelerate an optimisation routine if the files are compiling fast enough.

1.6.2 Shape loading

If a bug with shape loading is encountered after a certain number of iterations (typically 114 on older versions of Xepr), it should be solved by reseting Xepr to avoid AWG overloading. Use the following lines in your callback function (requires to import `acquire_esr` from `esrpoise`):

```
if acquire_esr.calls % 114 == 0 and acquire_esr.calls != 0:
    print('reset required')
    xepr_link.reset_exp(Xepr)
```

Python Module Index

e

Index

P

R

S

Z