Installation, quick start, demo overview, and app development

V1.9

```
def analyze_force_curve(distance_nm:np.array,
force_nN:np.array, baseline_window_begin:float,
baseline_window_end:float):
```

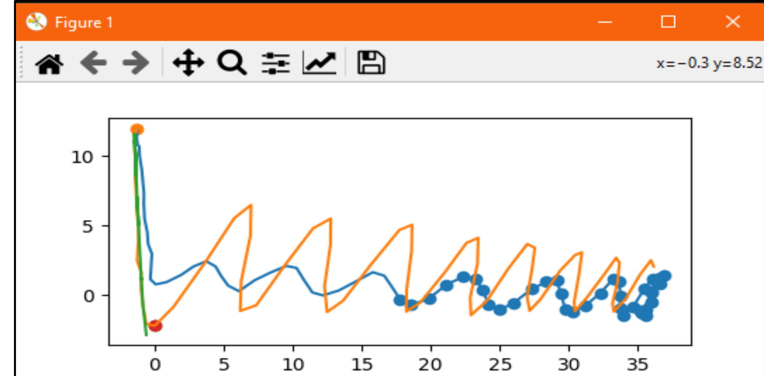
Content

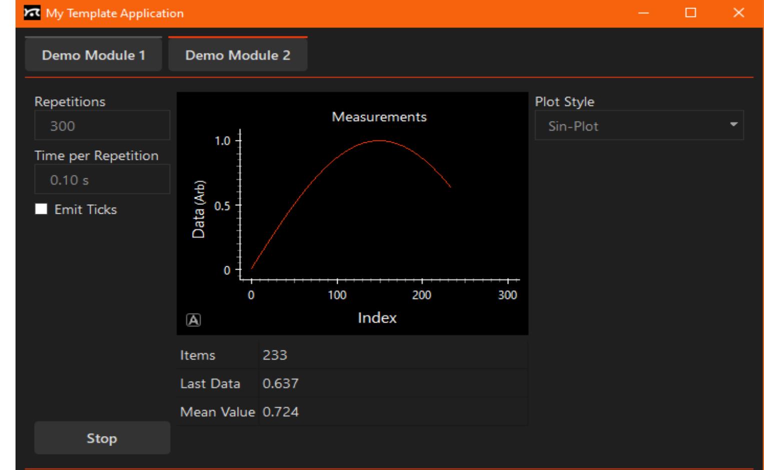
Installing and using the library

Overview of the library functions

Overview of the demo apps

Creating applications with user interfaces





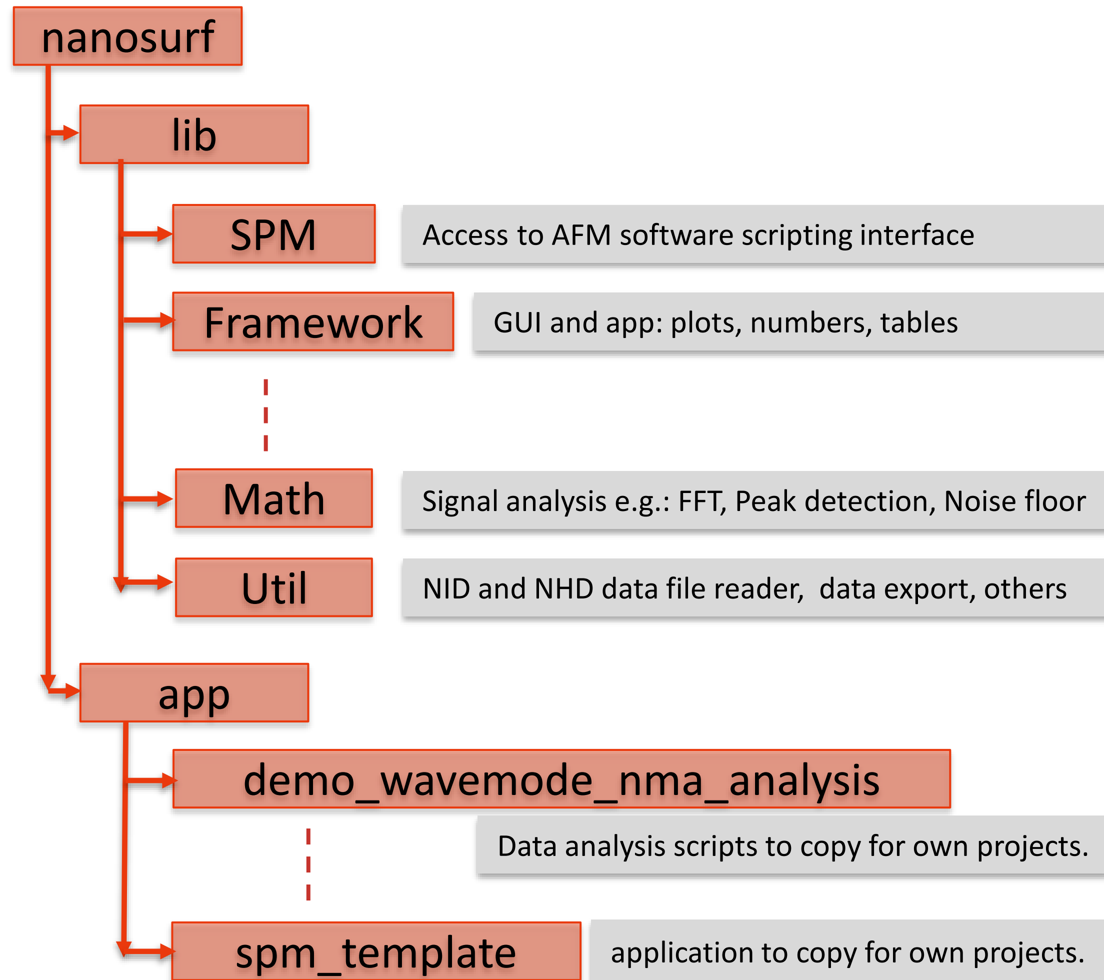
For library version 1.9 and newer

Installing and usage

The Nanosurf library

The library provides packages to access our AFM control software, many functions to analyze data and a full application framework to create apps with graphical user interface.

Many demo applications and templates give you a kick start into own projects.



Installing and prerequisites, Part 1/2

The library can be used with Python v3.9 up to v3.12.

Download it from ‘python.org’. Do not use the python available in Windows Store.

We are using VisualStudioCode as programming environment. But other IDE or even pure command line access is working too.

In VisualStudioCode the extension ‘Python’ should be installed. In addition, we recommend the extension ‘Ruff’ and ‘Code Spell Checker’.

After you have a working python environment, you can install the Nanosurf Library from pipy server. See next slide.

Installing and prerequisites, Part 2/2

After you have a working python environment, you are ready to install the Nanosurf Library.

Open a command shell and type: `pip install nanosurf`

If you want to upgrade to the latest version of the Nanosurf Library, type:

```
pip install Nanosurf --upgrade
```

The library and its dependencies are downloaded automatically.

Depending on your python installation, the library and the demo applications are installed in the following folder (replace the number 312 with your version):

Either here: *%appdata%\Python\Python312\site-packages\nanosurf*

or here: *%programfiles%\Python312\Lib\site-packages\nanosurf*

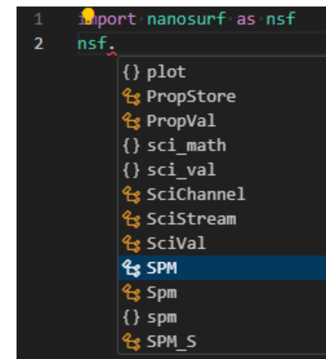
General Nanosurf library usage

They library is organized in sub-packages starting at *nanosurf.lib*.

Demo applications are in *nanosurf.app* and documentations in *nanosurf.doc*

To get access to the most common library functionality include the statement `import nanosurf as nsf` at top of a python file.

The code completion functionality in VisualStudioCode will give you hints about the package content when you start writing `nsf.`.



Only when importing rarely used packages or single library elements, use `import nanosurf.lib.x as x` or `from nanosurf.lib.x import y`

Overview of the library functions

Controlling AFM Software: nanosurf.lib.spm

There are two software platforms with different scripting interfaces:

There is the 'SPM' scripting interface which is implemented in our classic AFM/STM Control software (Naio, Easyscan2, C3000, Core, CX software). Access is provided by *nsf.SPM()*. The interface is described in the "*Script Programmers Manual*" which is provided in the SPM Software in "Help" panel.

To access the script commands of the new Studio-Software, use *nsf.Studio()*. The interface is loaded at runtime and a python wrapper is build. Therefore, the code completion functionality of VisualStudioCode can be used.

Connecting with Studio Software

```
import nanosurf as nsf

# connect with Studio application
studio = nsf.Studio()
connected = studio.connect()
if connected:
    wf_approach = studio.spm.workflow.approach
    wf_approach.property.approach_speed.value = 10.0
    wf_approach.start_approach()
```

Connecting with SPM Software

```
import nanosurf as nsf

#connect with classic spm application
spm = nsf.SPM()
connected = spm.is_connected()
if connected:
    obj_approach = spm.application.Approach
    obj_approach.ApproachSpeed = 10.0
    obj_approach.Start()
```

Reading NID and NHF datafiles: nanosurf.lib.util

SPM Control software creates nid-files,

Studio Control Software creates nhf-files.

For both file type there is a class providing read access to these data files to extract measured data.

See the two examples on the right

```
nid_file = nsf.util.nid_reader.NIDFileReader()
ok = nid_file.read(r"example_date.nid")
if ok:
    fwd_segment = nid_file.data.image.forward
    topo_index = nid_file.param['Dim2Name'].index("Z-Axis")
    topo_data = fwd_segment["Z-Axis"]
    topo_unit = nid_file.data_param.image_dim_info["X"]['units'][topo_index]

    # do some calculations on data
    mean_val = np.mean(topo_data)
    rms_val = np.sqrt(np.sum(np.square(topo_data-mean_val)))/len(topo_data)
    print(f"Roughness of topography is {rms_val:.3e} {topo_unit}")
```

```
nhf_file = nsf.util.nhf_reader.NHFFileReader(verbose=False)
ok = nhf_file.read(r"example_date.nhf")
if ok:
    nhf_file.pretty_print_structure() # show content of file

    fwd_segment = nhf_file.measurement["Image 1"].segment["Forward"]
    topo_channel = fwd_segment.read_channel("Topography")
    topo_data = topo_channel.dataset
    topo_unit = topo_channel.unit

    # do some calculations on data
    mean_val = np.mean(topo_data)
    rms_val = np.sqrt(np.sum(np.square(topo_data-mean_val)))/len(topo_data)
    print(f"Roughness of {topo_channel.name} is {rms_val:.3e} {topo_unit}")
```

Export data to Gwyddion: nanosurf.lib.util

Datasets read from files or calculated by analysis functions can be exported into Gwyddion file format.

A minimal example is shown by the following code:

```
from nanosurf.lib.util import gwy_export

    # prerequisites: data are first read from nhf-file as done in example in the nhf_reader slide

    image_points_per_line = measurement.attribute['image_points_per_line']
    image_number_of_line  = measurement.attribute['image_number_of_lines']
    image_size_x = measurement.attribute['image_size_x']
    image_size_y = measurement.attribute['image_size_y’]

    # Transform lists into image matrix
    gwy_image = np.flipud(np.reshape(np.array(topo_data), (image_number_of_line, image_points_per_line)))

    done = gwy_export.savedata_gwy("my_file.gwy", size_info=gwy_export.GwySizeInfo(
                                x_range=image_size_x,
                                y_range=image_size_y,
                                unit_xy="m"
                            ),
                            data_sets=[gwy_image],
                            data_labels=[topo_channel.name],
                            data_units=[topo_channel.unit])
```

A more sophisticated usage of the export functionality can be found in this example script:
nanosurf\app\demo_wavemode_nma_analysis\demo_spec_analysis_contact_mech_mod.py

Export plots to PNG file: nanosurf.lib.util

Data plotted by graphical UI applications using ‘pyqtgraph’ can be saved to to images in the png file format.

‘Pyqtgraph’ is a plot package for applications based on the QT graphical user interface framework.

This is used in our GUI applications and demo apps. Read on in the chapter of “GUI application programming” for more details.

See an example below shows a section from the *nanosurf\app\app_demo_scanning_and_lib_usage* demo

```
from nanosurf.lib.util import dataexport

    # code extraction from the demo “nanosurf\app\app_demo_scanning_and_lib_usage\qui.py” for “Save Image” button event

    def save_color_map(self):
        destination_file = nsf.util.fileutil.ask_save_file("Please provide filename", suffix_mask="*.png")
        if destination_file:
            dataexport.saveplot_png(destination_file, self.colormap_plot.plot)
```

More file utilities: nanosurf.lib.util.fileutil

In many scripts one must create files and folders or ask the user for files or folders with a dialog.

The file utilities sub-package provides routines to accomplish these tasks.

```
from nanosurf.lib.util import fileutil

fileutil.create_folder(file_path: pathlib.Path) -> bool:
    """ Make sure the folder exists. If needed, it  creates the intermediate directories starting from root. """

fileutil.create_unique_filename(base_name: str, folder: pathlib.Path, suffix:str, …) -> pathlib.Path:
    ""“ Create a unique filename.  The new file name has the structure of 'base_name_timestamp_index.suffix’ """

fileutil.create_unique_folder(base_name: str, folder: pathlib.Path, add_timestamp: bool = True, separator: str = '_'):
    ""“ Create a unique folder.  The folder name has the structure of 'base_name_timestamp_index’ """


fileutil.ask_folder(title:str = None, start_dir:pathlib.Path = None) -> pathlib.Path:
    """ Prompt user to select a folder. The function present the user a dialog to select a folder or create a new one. """

fileutil.ask_open_file(title:str = None, start_dir:pathlib.Path = None, suffix_mask:str = None) -> pathlib.Path:
    """ Prompt user to select a file. Prensents the user a dialog to select an existing. ""“

fileutil.ask_save_file(title:str = None, target_dir:pathlib.Path = None, default_file:pathlib.Path = None, suffix_mask:str =
None) -> pathlib.Path:
    """ Prompt user to define a file to save to. Presents the user a dialog to select an existing file or to define one. """
```

Values with 'unit' and 'name': nanosurf.lib.datatypes

Processing data and especially plotting or displaying data values in real-life scripts typically requires plotting correct values and units of the data as well as the signal name associated to the data (e.g.: print 2.45e-9 as "Topography rms = 2.45nm").

For this, the library defines three data types: *SciVal, SciChannel* and *SciStream*

SciVal – holds a value and a unit in a single class together. It also provides printing and converting from string to number.

SciChannel – holds an array of values (numpy array) with a name and a unit.

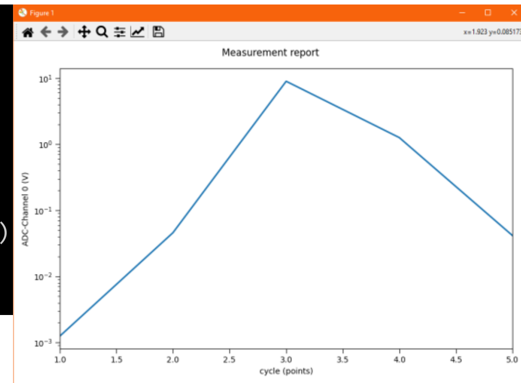
SciStream – holds a reference axis (e.g. timeline or frequency spectrum) and multiple synchronous data channels (e.g. Topography, Deflection, Tip-Current or Amplitude/Phase) each as SciChannel.

Most of the functions in nanosurf.lib.sci_math take or create these datatypes.

But also different export and plot functions handles these datatypes.

```
a = nsf.SciVal(2.45e-9, unit_str="m")
print(a)    >>> 2.450 nm
i = nsf.sci_val.from_str("0.1mm")
print(i)    >>> 100.000 µm


reference = nsf.SciChannel([1,2,3,4,5], unit="points", name="cycle")
measurement = nsf.SciChannel([0.125, 0.4587, 0.895, 1.258, 3.415], unit="V", name="ADC-Channel 0")
data = nsf.SciStream((reference, measurement))
nsf.plot.plot_stream(data, title="Measurement report", log_y=True)
```



Mathematical functions: nanosurf.lib.sci_math

In addition to mathematical libraries like numpy and scipy we provide with this library some functions not found in these libraries but that are useful for signal analysis.

Most of them are working with *SciChannels* and *SciStreams*.

```
nsf.sci_math.calc_fft(data_samples: ch.SciChannel, samplerate: float, window: fft_window_type = fft_window_type.hanning, spectral_density):
    """ calculate amplitude or spectral density spectrum from time data array(s)
nsf.sci_math.calc_compressed_spectrum(spec_data: ss.SciStream, min_dist_factor=1.02, algo:compress_spec_algo) -> ss.SciStream:
    """Reduces number of data points in large data sets by logarithmic compression method


nsf.sci_math.get_total_harmonic_distortion(spec_data: ss.SciStream, channel_index: int, max_number_of_harmonics: int) -> float:
nsf.sci_math.get_noise_floor(spec_data: ss.SciStream, channel_index: int) -> sci_val.SciVal:


nsf.sci_math.find_highest_peak(stream: ss.SciStream, channel_index: int) -> tuple[bool, float, float]:
nsf.sci_math.remove_peaks_in_spectrum(spec_data: ss.SciStream, channel_index:int, frq_peaks:list[float) -> ss.SciStream:


nsf.sci_math.calc_signal_integral_of_frq_band(spec_data: ss.SciStream, start_frq: float, end_frq:float, spectral_density=False) -> float:
    """ calculates the area of data in a specified frequency band
```

Example usage:

```
measurement = nsf.SciChannel([0.00125, 0.04587, 8.95, 1.258, 0.0415], unit="V", name="ADC-Channel 0")
spec = nsf.sci_math.calc_fft(measurement, samplerate=5000, spectral_density=True)
noise_floor = nsf.sci_math.get_noise_floor(spec)
print(f"Noise floor: {noise_floor}")

>>> Noise floor: 122.261 mV\sqrt(Hz)
```

Hardware device drivers: nanosurf.lib.devices

Device drivers for different Nanosurf accessories can be found here

accessory_interface: Class for communicating to devices connected to the Accessory Interface

device_tip_access_addon: Driver class to access all functionality of the "TipAccess Addon" for DriveAFM

device_tip_current_addon: Driver class to access all functionality of the "TipCurrent Addon" for DriveAFM

i2c: Chip drivers for many chips used in our devices. Attention, this functionality is only for experts and can potentially cause harm to the system, if used incorrectly.

Example usage: change gain of tip current addon module for DriveAFM

```
import nanosurf.lib.devices.device_tip_current_addon as device_tip_current_addon

spm = nsf.SPM()
tip_current_addon = device_tip_current_addon.DriveAFM_Tip_Current_Addon()
if tip_current_addon.connect(spm):
    tip_current_addon.set_gain(device_tip_current_addon.AmplifierGain.Gain_1Meg)
```

Example usage: Access a specific device connected to Accessory Interface

```
import nanosurf.lib.devices.accessory_interface as nsf_ai
import nanosurf.lib.devices.trinamic_motor_controller as tmc

spm = nsf.SPM()
ai = nsf_ai.AccessoryInterface(spm)
if ai.connect():
    # search a specific device by its serial number
    if ai.select_port_with_slave("998-00-00"):
        slaveid = ai.get_slave_device_id()
        print(f" Device found on Port({ai.get_current_port()})")

        motor_chip = tmc.TrinamicMotorController(0x28)
        ai.assign_chip(motor_chip)
```

Where they are, how to start them

How to start a demo application from the library

First, you need to know where the demo applications are. They are in the same place as the library.

Depending on your python installation, the demo applications are in the following folder (replace the python version with your installation):

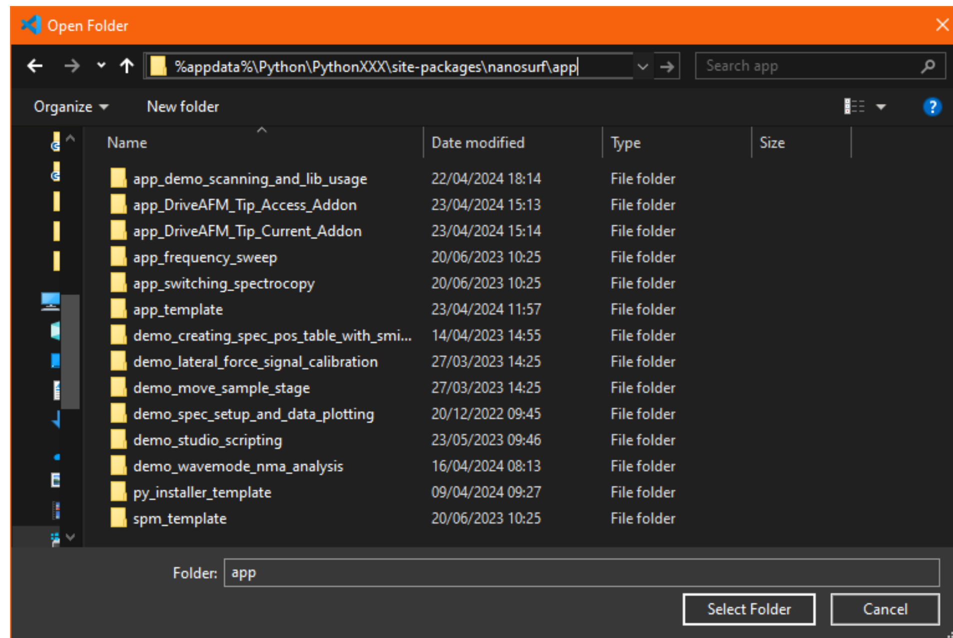
Either here: *%appdata%\Python\Python312\site-packages\nanosurf\app*

or here: *%programfiles%\Python312\Lib\site-packages\nanosurf\app*

Open VisualStudioCode (VSC), select Menu File->Open Folder...

Enter one of these path above, but with your correct python version,

into VSC's file explorer. Select an application of interest, click "Select Folder"



**Important: "Open Folder..." is important to use.
This moves the "current directory" of VSC to the folder where your app is.**

Starting the application by pressing the "F5" key is now possible, and debugging is activated correctly.

Library demo applications – Simple scripts

Some applications in the app folder demonstrate different concepts of accessing library elements, but do not serve any real-life purpose:

```
demo_studio_scripting:                    # Two simple script to show how to connect to Studio and use its functions. Need a running Studio software.
demo_spec_setup_and_data_plotting:        # Two simple scripts showing how to setup a spectroscopy, run it and plot results. All without big GUI
demo_creating_spec_pos_table_with_smiley: # A script showing how to setup a lithography to create indents. All without big GUI
demo_lateral_force_signal_calibration:    # A Jupiter notebook showing how to calibrate a lateral force signal.
demo_move_sample_stage:                   # A script demonstrating the usage of the stage interface of the SPM Software
demo_scanning_and_lib_usage:              # A GUI based demo showing some library functions (sci_math, gui.plot), start imaging, access data and plot
                                            them. Need a running SPM control software.
```

The image on the right shows the GUI that is provided by the *demo_scanning_an_lib_usage* application:

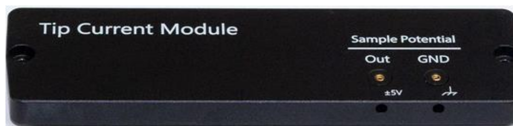
The code of this application demonstrates how to

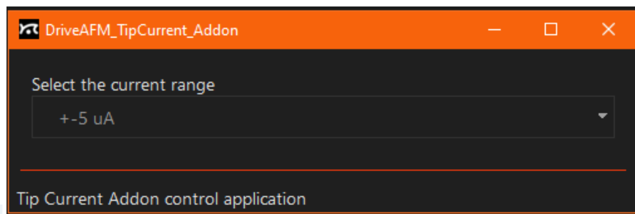
- access SPMfunctions in background tasks,
- create am application GUI and plot data with the nanosurf.lib.framework ,
- save and analyze data with nsf controller.lib.sci_math and nsf.lib.util.export functions

Adding Operating Modes and DriveAFM Add-ons

Nanosurf provides hardware add-on modules for the DriveAFM. These apps below are required to control these add-ons:

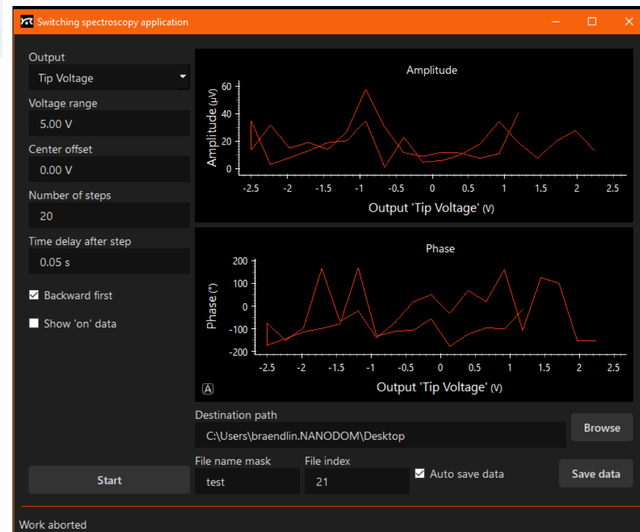
```
app_DriveAFM_Tip_Access_Addon:     # Controlling the TipAccess Addon for DriveAFM scan head
app_DriveAFM_Tip_Current_Addon:    # Controlling the TipCurrent Addon for DriveAFM scan head
```





Through python scripting, the functionality of the DriveAFM can be extended. The set of applications shown below perform some useful measurements.

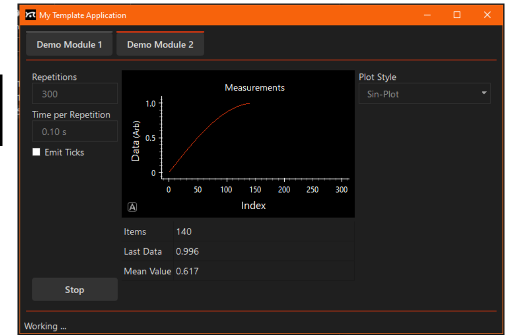
```
app_Switching_Spectroscopy:   # Voltage-Modulated-Spectroscopy for PiezoForce
app_Frequency_Sweep:          # Amplitude versus Frequency sweeps of signal channels
```



Templates to give a kick start

The two following application templates provide you with the core of a QUI based application which can be used to be filled in your specific code.
More details about programming your own GUI based application is described in the next chapter.

```
app_template:          # Template to create a GUI application for general usage.
spm_template:          # Template to create a GUI application and connect to SPM or Studio software
```



If you plan to distribute a python-based application as a windows executable, this template help you creating most of the necessary configuration files and tweaks needed to create one.

Copy the content of the sub folder 'pyinstaller' to your own folder where you have your 'main.py'.
Then adjust some parameters (e.g., application name) in the pyinstaller\resources\pyinstaller_one_file.spec file to your needs. Execute the 'create_one_file_exe.bat' and you get an executable in 'dist' folder.

With complex applications it's sometimes tricky to get all data into the executable.

```
py_installer_template:  # template to create a self executable application based on py_installer package
```

Using the app_themplate. Coding principle, style guide

Introduction

If you plan to create a fully featured application with a nice GUI, you will faces many coding task which are very common to all kinds of application development and are typically time consuming.

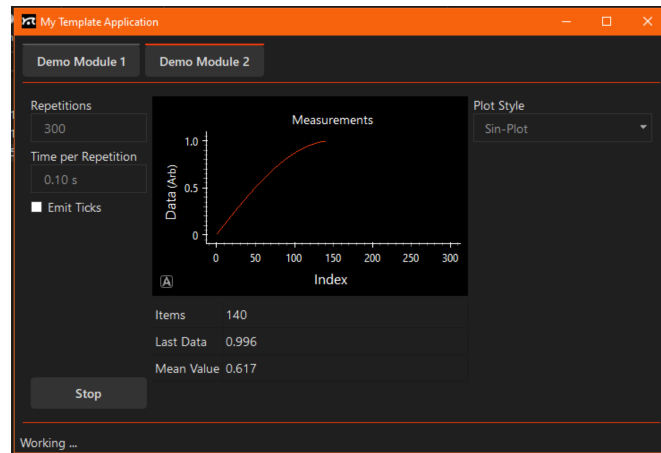
Therefor, many developers start creating simplified version of their application, copy and past old code, just to be faster.

But this leads to hard to maintain source code and many times also to less robust code. Or users do not get the standard behavior of the application that they are used to (e.g., remember last application window position and size, loading/storing last parameter settings, …)

To over come all this problems, we provide with our library an application framework with lots of core code to handle all this.

In addition, we provide multiple GUI-Elements found in control software, e.g. entering scientific number, plotting data, and handling background worker tasks to make the app responsive to user actions (e.g., 'Stop' button click or resizing application while performing a measurement)

Our framework's GUI-Elements are based on "QT" and the python "pyside6" package. Plotting is based on the "pyqtgraph" package. Both packages are being installed when you install our "nanosurf" package.



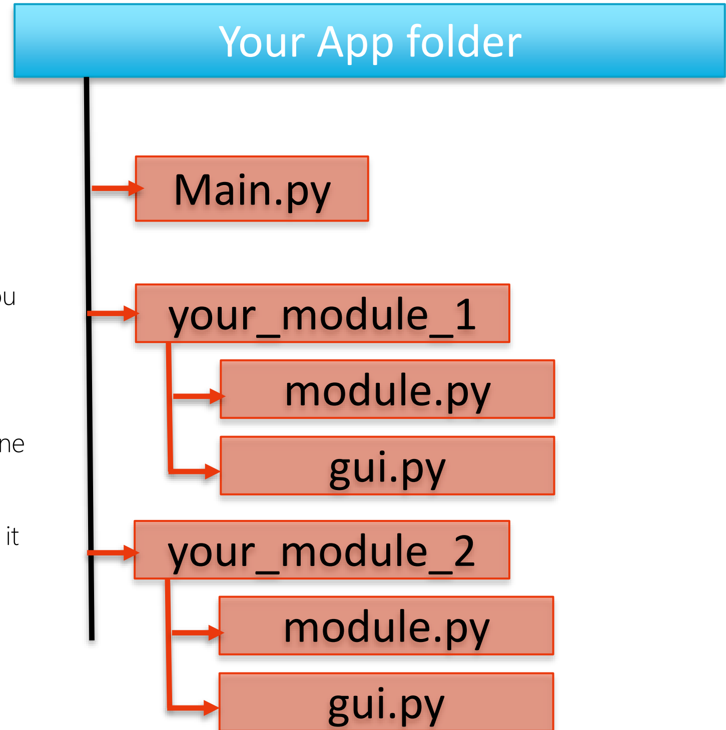
Getting started

To create your own application, use *app_template* or *spm_template* as your starting point:

- Copy the template folder to an own project folder rename it to your project name.
- In your new project, copy the demo_module folder and rename it to your own function.
- Open Visual-Studio-Code and select File->Open Folder. Select the project folder that you just created.
- In *main.py* change *app_name* and *company_name* (about line 15) to your project name.
- Add *import statements* of your new module (line 10) and add it with *self.add_module()* (line 30)
- Use demo_module as a reference how to create your own module. Later you can delete it and remove it from main.py

That's it. Now start coding your functionality in *module.py* and create the gui in *gui.py*

Start your app in debugging mode from any code window inside VSC with F5-Key



Coding Style

To create code that is readable and maintainable by others, programmers should follow coding style guides and documentation.

Luckily, Python has its code style guide well defined in PEP8.

(PEP is a naming convention like we have in Jira with NANO1245 and is available at https://peps.python.org/pep-xxxx/)

We follow this guide and in addition we use the **typing hints** defined in PEP484. Typing hints is something like giving a variable a type information which is not needed by python itself but helps the Visual Studio Code Editor extension 'pylance' and 'ruff' to help programmers with tips and color the code correctly.

As documentation style we follow the numpy library doc style, which is defined here: Numpy doc. Again, Visual Studio Code can read them and help programmers during coding. While hovering the mouse over a function name the editor shows the function description found in your documentation as a popup.

Return type

Type hints

Function names are lower case

```
def create_filename(base_name: str, ext: str = '.dat', sep: str = "_") -> str:
    """ Construct a file name based on pattern and current date/time.
        The result will be something like 'my_data_20210613-100543.dat'
        this with base_name 'my_data'

    Parameters
    ----------
    base_name: str
        Mask of the name (e.g., 'my_data')

    ........

    Result
    ------
    str:
        constructed file name
    """
    current_datetime = datetime.now().strftime("%Y%m%d-%H%M%S")
    filename = base_name + sep + current_datetime + ext
    return filename
```

Numpy doc style: Description, Parameters and Result

You get nicely colored code in the editor and code completion works

Separating functionality and GUI

A common mistake it to mangle functionality and GUI-Elements in one source code. This makes code maintenance and increasing functionality hard. A common method to solve this issue is the model/view pattern:

The functionality of a software is written in one part of code: You derive your code by subclassing the *ModuleBase* class of the library and you program all the functionality as members in *module.py*.

Think of creating a device driver where all functionality of a "device" is exposed. So, create all functions needed to do the measurement task or what ever it does exactly in here. No visualization is done here, only data processing and commanding.

Visual components of your app (data entry elements, result visualization like plots or numbers, status messages) are programmed in a class derived from *ModuleScreen* class and placed in *gui.py*

Interaction from module -> gui is done by emitting signals and the gui is listening to such signals by connection to it (Use the Signal/Slot mechanism of Qt).

Interaction from gui -> module is done by calling functions in the module.

In case of parameters there is a "binding" mechanism available which handles the synchronization between value stored in a member variable in the module with the user interface element.

Such "bondable" parameters must be defined as members of type *nsf.datatypes.ProVal*.

GUI-Elements are connected to them by the *bind_gui.connect_to_property()* function in *gui.py/bind_gui_elements()*. By this the GUI-Element gets updated whenever the property is updated by the module and the modules parameter gets informed when the user change a value in the GUI.

So, the module itself connects to the parameter and a function is called whenever the parameter is changed (from GUI or from any other direct change of its value)

The graphic to the right is visualizing this principle.

module.py

```
class MyModul(ModuleBase):
  def __init__(self):
        self.image_size = PropVal(SciVal(2,”m”))
        self.image_size.sig_value_changed.connect(self.size_changed)

  def size_changed(self):
        # do something with a (e.g send to controller)
        self.spm_scan.ImageWidth = self.image_size.value
```

gui.py

```
class MyScreen(ModuleScreen):
   def do_setup_screen(self):
        self.my_edit = nsf_sci_edit.SciSciEdit(“Size”)
        bind_gui.connect_to_property(self.my_edit, module.image_size)
```

Signal/Slot Communication

A function module typically process data or measures data and has new information about its state or data content.

Such state transition could be *file_loaded*, *start_measuring*, *new_data_available*, …

We use the Qt.Signal/Slot mechanism for such communication.

A function module sends out event signals with emit() and other code parts (e.g. GUI Elements) can receive such events and as a consequence a function is called. To do so, a receiver must call connect() of the corresponding signal and provide its call back function.

By this method, the GUI elements can react accordingly to the event (e.g. plot new data, disable parameters during measurement, stop a task, …).

This principle keeps the GUI always reactive to user events and never blocks keyboard and mouse input.

module.py

```
class MyModul(ModuleBase):
        sig_work_started = Signal()
        sig_work_done = Signal()

def do_something(self):
        sig_work_started.emit()
        # do some work
        sig_work_done.emit()
```

gui.py

```
class MyScreen(ModuleScreen):
  def do_setup_screen(self):
        self.module.sig_work_started.connect(self.measurement_started)
        self.module.sig_work_done.connect(self.show_result)

def measurement_started(self):
  # disable some gui elements

def show_result(self):
  # enable gui elemets
  # read result from module and plot result
```

Background Tasks

If a function of a module must perform long lasting processes (e.g., measuring a data stream over 10s) then this task must be executed in a background thread. If not, the GUI would be blocked and changing parameters or pressing a "Stop" button would not be possible.

To simplify setting up such a background task, the framework provides the classes

`nsf.frameworks.qt_app.nsf_thread.NSFBackgroundWorker`

and

`nsf.frameworks.qt_app.nsf_thread.SPMWorker`

Derive a new class (e.g mytask) from it and implement the *do_work()* function. The background task is initialized once with *mytask.start_thread*(). And each time you would like to execute its task call *mytask.start_worker()*

The background task emits *sig_worker_started*, and *sig_worker_finished*

The module or the GUI should connect at least to *sig_worker_finished* to know when the long-lasting task is finished.

worker_task.py

```
class MyTask(nsf.frameworks.qt_app.nsf_thread.SPMWorker):
  def do_work(self):
        # do some work (e.g measuring data)
        sig_new_data.emit()

class MyModul(ModuleBase):
        self.mylongwork = MyTask()

def start_measure_data(self):
        self.mylongwork.start()
```

gui.py

```
class MyScreen(ModuleScreen):
  self.module.mylongwork.sig_started.connect(self. measurement_started)
  self.module.mylongwork.sig_new_data.connect(self.show_result)
```