NanoLib user guide

ETH Zurich
Laboratory of Solid State Physics
Microstructure Research

Danilo A. Zanin - dzanin@phys.ethz.ch

Lorenzo G. De Pietro - depietro@phys.ethz.ch

Quentin Peter - qpeter@stud.phys.ethz.ch

November 9, 2016

NanoLib library allows to open and analyze data generated by the Nanonis SPM Control System™ in MATLAB™. The first version was developed in 2015 by Quentin Peter during its master thesis *Spin Polarized Field Emission STM and Image Processing* in the Solid State Physics Laboratory for Microstruture Research at the ETH Zurich under the supervision of Dr. U. Ramsperger and L.G. De Pietro. Some of the features of this library are still oriented to solve problems related to the master thesis, e.g., *scan_type* field in the header structure (see 3.1). In future versions some of these features may be changed and generalized. Every comment, idea and contribution is welcome.

NanoLib library is divided in the package folders: *+sxm*, *+dat* and *+utility*.
A function in a folder called *+folder* can be called as *folder.function*.

Contents

1 Installation and Use

1.1 Installation

The preferred way to access the NanoLib is to make use of the free version control system Git.

1.1.1 Git

If you are familiar to git, you can directly clone the repository to your workstation. With the command:
`git clone https://github.com/ethz-micro/matlab_nanonis ~/mynanolib`
you will clone the repository to a subdirectory called `mynanonlib` in your actual working directory.

1.1.2 Manual Download

If you do not know git probably the best is to simply download the ZIP file from the homepage. You will find the file here: https://github.com/ethz-micro/matlab_nanonis and click on the green field called "Clone or download", then "Download ZIP".
You can clone the repository to your computer or download the ZIP file and unzip it to a directory of your choice, for example: "~/mynanolib".

1.2 Use

The NanoLib library runs on MATLAB™ 2015a or later. In order to access the library you need to add the path to the directory where you copied the library (e.g., the same provided above, ~/mynanolib). To this end open MATLAB™ and set the folder called `NanoLib` (you should find it in the parent folder of this user guide, e.g., ~/mynanolib/NanoLib) as the MATLAB™ current folder. You may use also the button "Browse for folder" beside the current folder path in the MATLAB™ main window.[1]
At this point you can use NanoLib functions by calling first the "type", second the "operation" and then the specific function. For example, in order to load a SXM image, you can simply write:
`>> sxm.load.loadProcessedSXM(fileName).`

[1] Alternatively, you may type in the MATLAB™ terminal the following command: `>> addpath ~/mynanolib/NanoLib`. An orange warning message appears in the MATLAB™ terminal if the added path does not exist.

1.2.1 Load a simple SXM file

The following instructions describe how to open and load a *file.sxm*.

1. Add the NanoLib path to the MATLAB™ path (if not yet done):
   `>> addpath ~/mynanolib/NanoLib`

2. Define the file name and load it:
   `>> fileName = 'SXM_file.sxm';`
   `>> sxmFile = sxm.load.loadProcessedSxM(fileName);`

3. Define the channel to plot:
   `>> iCh = 1; % Channel number`

4. Plot data:
   `>> figure('Name',sprintf('file:  %s',fileName));`
   `>> sxm.plot.plotFile(sxmFile,iCh);`

1.2.2 Load a simple DAT file

The following instructions describe how to open and load a *file.dat*.

NOTE: While using Nanonis SPM Control System™ it is sometimes useful to create user defined experiments (see sectin 5). The NanoLib library allows you to incorporate user defined functions (see section 3). The first time you load a *file.dat* the NanoLib library will ask you to indicate the path to the NanoLib library and the path for the userNanoLib library (press cancel if you don't have any user defined functions). A file called *datSettings.txt* will be created in the package folder *+dat/+load/* with your local settings.

1. Add the NanoLib path to the MATLAB™ path (if not yet done):
   `>> addpath ~/mynanolib/NanoLib`

2. Define the file name and load it:
   `>> fileName = 'DAT_file.dat';`
   `>> sxmFile = dat.load.loadProcessedDat(fileName);`

3. Define the channel to plot:
   `>> iCh = 1; % Channel number`

4. Plot data:
   `>> figure('Name',sprintf('file:  %s',fileName));`
   `>> dat.plot.plotFile(sxmFile,iCh);`

Other examples can be found in the section 5.

2 The +sxm package folder

Images generated via the scanning interface of the Nanonis SPM Control System™ have extension *file.sxm*. Once loaded, variable, from now on *sxmFile*, is a structure divided in: *i*) **header**, a structure containing all information present in the header of the *file.sxm*, and *ii*) **channels**, an array of channel structures containing data and information about every channel. Both, **header** and **channels** can be called by

sxmFile.header and *sxmfile.channels{#}*

being the number of the channel. When only one channel is loaded one refers to the channel simply by *sxmfile.channels*. More information about the substructure of header and channels is presented below.

2.1 sxmFile: header and channels structures

The functions works with a structure that holds every relevant informations. Header and channels structure have following fields:

header is a structure composed of:

- **scan_file** name of the file
- **rec_date** date of the scan
- **rec_time** time of the scan
- **scan_pixels** [nx;ny], number of pixels
- **scan_range** [rx;ry], range [m]
- **scan_offset** [ox;oy], offset [m]
- **scan_angle** tilt angle of the scan
- **scan_dir** 'up' or 'down'
- **bias** bias voltage [V]
- **scan_type** 'STM', 'SEMPA', 'NFESEM', etc.
- ··· Others informations extracted from the file

channels is an array of channel structures composed of:

- **Direction** 'forward' or 'backward'
- **Unit** 'Z' or whatever the unit is
- **Name** The name of the channel

data A $n \times m$ matrix of processed data

lineMedian A $n \times 1$ matrix of raw line median

lineMean A $n \times 1$ matrix of raw line mean

linePlane A $n \times 1$ matrix of raw line mean linear fit

lineResidualSlope a $1 \times m$ matrix of processed column mean linear fit

lineStd A $n \times 1$ matrix of processed line standard deviation

To access the scan data on a structure named *sxmFile*, one should type, e.g., *sxmFile.header.rec_date*.

2.2 +load

This folder contains everything needed to load and process *.sxm* files.

2.2.1 loadsxm

This file is provided by Nanonis SPM Control System™ and loads a specified channel from a *.sxm* file.

header = loadsxm(fn) loads a file named *fn.sxm* and returns the Header. This function is called by *load.loadProcessedSxM* and **should not be called directly**.

[header, data] = loadsxm(fn, i) reads the channel *i* and returns its *data*.

.sxm files are composed of an ascii header and of single precision binary data. They are separated by 0x1A 0x04 (SUB EOT).

2.2.2 processChannel

channel = processChannel(channel, header) Process the *channel* as described below using the informations form *header*. This function is called by *load.loadProcessedXXX* and should not be called directly.

channel = processChannel(channel,header,corrType) If *corrType* is set to 'Raw', data are only oriented/rotated. If it is set to 'Median', the median is used instead of the mean for lines corrections. If it is set to 'PlaneLineCorrection' a linear fit is used.

The processing orientate and rotate the data so that all the images are comparable. Everything that is removed is saved in the output structure to avoid loosing informations.
The mean value of the measurement under the conditions of each pixel must be extracted from the data. As there is drift and other instabilities, the mean value of the data is generally not a good value. The mean of each line is used instead, as the measurement conditions doesn't change too much during one line. Others possibility include the median or the mean plane. The mean plane along the line is also removed.
For STM, This offset is subtracted. For NFESEM and SEMPA, it is divided, as justified in the thesis.

2.2.3 loadProcessedSxM

Data are processed as followw......... GABRIELE!

file=loadProcessedSxM(fn) loads and process all the channels of *.sxm* file named *fn*. The structure *file* contains all the informations and is used in a large number of other functions.

file=loadProcessedSxM(fn, chn) only loads the channels whose numbers are in the array *chn*.

file=loadProcessedSxM(fn, chnName) searches for all channels (*chn* above) that contains the chnName and loads channels whose numbers are in the array *chn*

file=loadProcessedSxM(fn, corrType) If *corrType* is set to 'MedianCorrection', the median is used instead of the mean for lines corrections. If it set to 'PlaneLineCorrection' a linear fit is used.

The loading is done with *load.loadsxm* and processing with *load.processChannel*.

2.3 +plot

This package contains everything needed to plot the data.

2.3.1 folder2png

folder2png(folderName) finds every *.par* and *.sxm* files in *folderName*, plot all relevant channels and saves the images in a *image* folder.

2.3.2 plotData

[h, range] = plotData(data, name, unit, header) plots the *data* using informations from the *header*. The figure title is deduced from *name* and *unit*. It returns the plot handle *h* and the chosen range *range*.

[h, range] = plotData(data, name, unit, header, varargin) allows to define following options:

- varargin = {'Units', unit} sets axis to 'm', 'nm', 'mum'. Defaults units are meter.
- varargin = {xoffset, yoffset} adds an offset to the plot.*xoffset* and *yoffset* should be in the proper uints.
- varargin = {'HoldPosition'} avoid command `'OuterPosition', [0,0,1,1]` that centers the axis of the image.
- varargin = {'NoTitle'} hide the title of the image.

The range is 2 STD. If the data is STM, only the lines with low std are considered for the range.

2.3.3 plotChannel

[h, range] = plotChannel(channel, header) plots the *channel* using informations from the *header*. It returns the plot handle *h* and the chosen range *range*.

[h, range] = plotChannel(channel, header, varargin) allows to define some plot options (see section 2.3.2).

It calls *plot.plotData* on the channel data.

2.3.4 plotFile

[h, range] = plotFile(file, n) plots the n^{th} channel of *file*. It returns the plot handle *h* and the chosen range *range*.

[h, range] = plotFile(file, n, varargin) allows to define some plot options (see section 2.3.2).

It calls *plot.plotChannel*.

2.3.5 plotHistogram

plotHistogram(data, range) plots an histogram of *data* and draw lines on the limit of *range*. It removes the $.1\%$ most extreme values.

2.3.6 scalebar

hObj = scalebar(xStart,yStart,blength,bunits) plots the scale bar on the current figure. *xStart* is the x position of the scale bar, *yStart* is the y position of the scale bar, *blength* and *bunits* are the length, and resp. the units, of the scale bar.

Remark: *xStart* and *yStart* can be provided as ranges, e.g. *xStart*=`[min(x),max(x)]` and *yStart*=`[min(y),max(y)]`. If this is the case the scale bar will be automatically placed within the 10% of the `[min(x), min(y)]` point.

2.4 +op

This package contains various useful functions.

2.4.1 filterData

[filtered, removed] = filterData(data, pixSize) filters the *data* with Fourier transform. The filtering keeps structures of approximatively *pixSize* pixels. It returns the filtered data *filtered* and the removed noise *removed*.

[filtered, removed] = filterData(data, pixSize, 'plotFFT', zoom) additionally plots the Fourier plane. The optional variable *zoom* has default value 8 and is used to zoom in the Fourier plane.

2.4.2 getOffset

[offset, XC, centerOffset] = getOffset(img1, header1, img2, header2) compares the images matrices *img1* and *img2* using informations from the two *headeri* to find the most probable *offset*. The units of *offset* are from header.scan_range. It correspond to the maximum of the cross correlation matrix *XC*. The corresponding offset relative to the centre of the two images is returned in *centerOffset*.

[offset, XC, centerOffset] = getOffset(img1, header1, img2, header2, 'mask') compares masks instead of images.

The offset is from the origin of the image, which is in a corner. The offset of the center is the centerOffset, but is less convenient to work with.

2.4.3 getRadialFFT

[wavelength, radial_spectrum] =getRadialFFT(data) Computes the *radial spectrum* of the image saved in *data* and the corresponding *wavelength*. The wavelength unit is pixel.

[wavelength, radial_spectrum] =getRadialFFT(data,pixPerUnit) Changes the wavelength unit with the number of pixels per units, *pixPerUnit*.

This function is used to study the radial spectrum of an image computed from the FFT.

2.4.4 getRadialNoise

[noise_fit, signal_start, signal_error, noise_coeff] = getRadialNoise(wavelength, radial_average) tries to fit a noise from the data of *getRadialFFT*. *noise_fit* is the detected noise. *signal_start* is the first position where the signal is detected. *signal_error* is the error caused by the discrete nature of the signal on *signal_start*. *noise_coeff* gives the power law coefficients for the first detected noise.

[noise_fit, signal_start, signal_error, noise_coeff] = getRadialNoise(wavelength, radial_average, maxNbrNoise) Limits the number of noises to *maxNbrNoise*. The default value is 10.

2.4.5 getRange

[xrange, yrange] = getRange(header) extract the ranges *xrange*, *yrange* from *header*.

2.4.6 interpHighStd

data = interpHighStd(data) Removes the lines with high STD values and interpolates the missing values.

2.4.7 interpPeaks

data = interpPeaks(data) Removes the data witch are too far from the mean and interpolates the missing values.

2.4.8 nanHighStd

data = nanHighStd(data) is useful for STM measurements. Usually the lines with very high std don't carry informations, and thus if a line has $std > 3median$, it is set to nan.

2.4.9 nanonisMap

colorMap = nanonisMap(nPti) is a color map function that generates a Nanonis color like mapping of *nPti* number of colors. *nPti* is an optional value. If not provided *nPti* = 64 per default.

2.5 +mask

Theses functions are useful to compute threshold mask and apply them.

2.5.1 applyMask

applyMask(mask) apply the boolean mask *mask* to the current figure.

applyMask(mask, color, alpha, xrange, yrange) apply the boolean mask *mask* in the range *xrange*, *yrange* with color *color* and transparency *alpha*.

The ranges are vectors containing a start point and an end point. See MATLAB's *image* documentation.

2.5.2 getMask

[maskUp, maskDown, flatData] = getMask(data, pixSize, prctUp, prctDown) flatten and filter the *data* before computing threshold masks. *flatData* is the flattened and filtered data. *maskUp* marks everithing above *prctUp* and *maskDown* below *prctDown*. The filtering is done using *op.filterData*, to which *pixSize* is passed to keep features of this approximate size.

[maskUp, maskDown, flatData] = getMask(data, pixSize, prctUp, prctDown, 'plotFFT', zoom) Additionally passes *'plotFFT',zoom* to *op.filterData* to visualize the Fourier plane. *zoom* is optional.

The flattening is done using sliding mean.

2.6 +convolve2

This in an improved version of MATLAB's conv2 matrix. It allows a better gestion of boundaries. It was downloaded from MATLAB file exchange. See the license file.

3 The +dat package folder

Besides surface imaging Nanonis SPM Control System™ allows to store data measured by the physical channels. Data from the so called experiments are stored in a *file.dat*. Once loaded, variable, from now on *datFile*, is a structure divided in: *i*) **header**, a structure containing all information present in the header of the *file.dat*, and *ii*) **channels**, an array of channel structures containing data and information about every channel. Both, **header** and **channels** can be called by

datFile.header and *datfile.channels{#}*

being the number of the channel. When only one channel is loaded one refers to the channel simply as *datfile.channels*. More information about the substructure of header and channels is presented below.

3.1 datFile: header and channels tructures

The functions works with a structure that holds every relevant informations. To access the scan data on a structure named *expFile*, one should type *expFile.header.rec_date*. Header and channels structure have following fields:

header is a structure composed of:

- **file** name of the file
- **path** path of the file
- **experiment** experiment name
- **rec_date** date of the scan
- **rec_time** time of the scan
- **points** number of experiment points
- **grid_points** number of experiment repetition
- **channels** list of registered channels
- **list** is a $2 \times n$ list of string, n being the number of lines in the text header. Lines in the *header.list* are of the form {'Key','data'}, e.g., {'rec_date','22.08.2016'}
- $\cdots$ Others informations extracted from the file depending on the specific experiment

channels is an array of channel structures composed of:

Direction 'forward' or 'backward'

Unit 'Z' or whatever the unit is

Name The name of the channel

data A $n \times m$ matrix of processed data, where n is the number of points and m is the loop number (default $m = 1$)

The first channel, i.e. channel(1), is reserved to the sweep_signal.

3.2 +load

This folder contains everything needed to load and process *.dat* files. **NOTE: While using Nanonis SPM Control System™ it is sometimes useful to create user defined experiments (see section 5.5). The NanoLib library allows you to incorporate user defined functions. The first time you load a *file.dat* the NanoLib library will ask you to indicate the path to the NanoLib library and the path for the userNanoLib library (press cancel if you don't have any user defined functions). A file called *datSettings.txt* will be created in the package folder *+dat/+load/* with the local settings.**

3.2.1 setSettings

[nanoLib,userNanoLib]=setSettings() sets the local path for the nanoLib library. This function is automatically called the first time a *file.dat* is loaded. It create a file called *datSettings.txt* with the local settings.

3.2.2 loaddat

This file is provided by Nanonis SPM Control System™ and loads a specified channel from a *file.dat*.

[header,data,channels]=loadDat(fn) loads a file named *fn.dat* and returns the *header*, the *data* and the *channels* list. This function is called by *load.loadProcessedDat* and **should not be called directly**.

3.2.3 experiment_XXX

files.dat are all characterized by a unique **experiment_name**, that is saved in the first line of every *.dat* file. In the follow we refer to those *files.dat* simply as *experiments*. Different *experiments* have different headers and data characteristics. Every *experiment* have a specific function called *experiment_XXX*, *XXX* being the name of the experiment. *experiment_XXX* are called automatically by the *loadProcessedDat* function as listed below.

experiment_name = experiment_XXX('get experiment') returns the **experiment_name**.

header = experiment_XXX('process header',header) process the *header* of the *experiment*. The *header* variable is result of the function.

[header,channels] = experiment_XXX('process data',header,data) stores data into the *channels* structure described above. Where needed some additional processing are applied to the data. Header's information are adjusted accordingly.

Further *experiments* can be implemented by simply defining a function called *experiment_newExperiment*. New *experiment* functions **must** have the same structure described above and should be saved in a *+load* package folder (see section 5.5).

3.2.4 getAllExperiments

experiment_list = getAllExperiments() returns a $2 \times n$ list, where n is the number of the function *experiment_XXX*. In the first column is listed the unique name of the experiment saved in the *+load* package folder. In the second column compare the correspondent function, i.e., *experiment_XXX*.

This function is used by the function *loadProcessedDat* when loading different *experiments*.

3.2.5 loadProcessedDat

loadProcessedDat loads a *file.dat* calling the function *loaddat*. And process the *header* and the *data* according to the type of experiment by calling – automatically – the corresponding *experiment_XXX*.

file=loadProcessedDat() ask for a *fileName.dat* and load it.

file=loadProcessedDat(fileName) load the file named *fileName.dat*.

file=loadProcessedDat(fileName,pathName) load the file named *fileName.dat* at a given *pathName*.

3.3 +plotDat

This package contains everything needed to plot the data.

3.3.1 plotData

hObject = plotData(data, name, unit, sweep_channel,varargin) plots the *data* using according to the *sweep_channel*. The figure title is deduced from *name* and *unit*. It returns the plot handle *hObject*. *varargin* are the standard plot options. Additional options are provided.

- varargin = {'xOffset', NUMBER } shifts the x axis by the given offset
- varargin = {'hideLabels'} leaves all extra labels out.

3.3.2 plotChannel

hObject = plotChannel(channel,sweep_channel,varargin) plots the *channel* using informations from the *sweep_channel*. It returns the plot handle *hObject*. It calls *plot.plotData* on the channel data, *varargin* are therefore the same as *plot.plotData* 3.3.1.

3.3.3 plotFile

hObject = plotFile(file,channel_numbers) plots the n^{th} channel of *file*. *channel_numbers* may be a $n \times 1$ array. It returns the plot handle *h*.

hObject = plotFile(file,channel_numbers,run_numbers) plots the n^{th} repetition of the provided *channel_numbers* . Whenever an *experiment* has more loops, repetitions are characterized by a *run_numbers*.

It calls *plot.plotData* on the channel data.

3.4 +op

To be done

4 +utility

The package folder *+utility* contains generic functions, which can be used when analyzing both *sxmFiles* and *datFiles*

4.0.1 combineChannel

channel=combineChannel(file, name, chn, chw) combined the channels *chn* of the *file* structure with weights *chw* and return a new *channel* with name *name*.

4.0.2 divideByChannel

channel=divideByChannel(file,name,chn_N,chn_D,chw) divide the channel *chn_N* by the channel *chn_D* of the *file* structure with weights *chw* and return a new *channel* with name *name*.

4.0.3 getAlphaColor

outputRGB = getAlphaColor(inputRGB,alpha) returns *alpha%* lighter *inputRGB* color.

outputRGB = getAlphaColor(inputRGB,alpha,'dark') returns *alpha%* darker *inputRGB* color.

4.0.4 getChannel

channelNumber = getChannel(channels,channelNames) returns all channel numbers where *channels.Name* matches the *channelNames*. *channelNames* can be either a single string or a list of strings.

channelNumber = getChannel(channels,channelNames,direction) returns only the channel number where *channels.Direction* matches the *direction*, too.

4.0.5 getColor

[color,colorScale] = getColor(x,xRange) computes the ratio between a value x and the *xRange* (2 by 1 array). It returns the *color* – and the *colorScale* – according to a predefined color map. *Jet* is the default color map.

[color,colorScale] = getColor(x,xRange,mapping) allows to provide e specified *mapping* other than the default, i.e. *jet*. *mapping* should be a color map function, e.g. *hsv*, *parula*, *hot*, *summer*m *autumn*. Note that, since *mapping* is an argument of functions *getColor*, it must be called by function handle "at" – @.

5 Examples

NanoLib library comes with few example showing the basics usage of the library. Some files are also provided in the directory *Files*. Below a list of all examples with a short explanation.

5.1 SxM_Example

example_open_SxM

shows different ways to load a *file.sxm*.

example_process_option

shows different ways to process a file while loading a *file.sxm*.

example_plot_SxM

shows different ways to load a *file.sxm*.

example_get_drift

detect XY-offset between two different *sxmFiles*.

example_mask

generates a mask of a *sxmFile* and apply on the original image.

example_RadialFFT

applies *op.getRadialFFT* and plots some interesting quantities.

5.2 SxM_viewer

The SXM viewer allows to open sxm files and have a fast overview of all channels stored within a SXM image.

Setup

In order to use the SXM viewer you need to set local paths for the NanoLib and the path to directory where images are saved. The first time you run the viewer you will be asked to find paths. Information will be saved in a file *localSettings.txt*

in the same directory where *SXM.m* is with the following informations:
`nanoLib ∼/MATLAB/matlab_nanonis/NanoLib`
`dataPath ∼/Nanonis/Data`

Usage

1. Run SXM viewer main file:
   `>> SXM`
2. Select the process type in the popupmenu (default = Raw)
3. Press open button and search for the directory where measurements are;
4. Press items in the list box in order to let them appear in a new figure;
5. Press on plotted channels to export them on a new figure.

5.3 Dat_Example

example_Dat shows different ways to load some *experiment.dat*.

5.4 Dat_viewer

The DAT viewer allows to open dat files and have a fast overview of all channels stored within a dat file.

Setup

In order to use the DAT viewer you need to set local paths for the NanoLib and the path to directory where images are saved. The first time you run the viewer you will be asked to find paths. Information will be saved in a file *localSettings.txt* in the same directory where *DAT.m* is with the following informations:
`nanoLib ∼/MATLAB/matlab_nanonis/NanoLib`
`dataPath ∼/Nanonis/Data`

Usage

1. Run SXM viewer main file:
   `>> DAT`
2. Press open button and search for the directory where measurements are;
3. Press items in the list box in order to let them appear in a new figure;
4. Press on plotted channels to export them on a new figure.

5. Whenever the button 'hold exported' is active (blue capital letters), all exported channels will be inserted in the same figure.

5.5 NanoLib_micro

Contains user defined examples in order to integrate user defined experiments which are defined as *file.dat*.