fit2x Documentation

Thomas-Otavio Peulen

Sep 14, 2020

CONTENTS

CHAPTER

ONE

GENERAL DESCRIPTION

fits2x is a collection of fit models that use maximum likelihood estimators for polarization and time-resolved fluorescence decays (see [MCH+01]).

fit2x implement burst integrated fluorescence lifetime fits (BIFL) with scatter. The library can be used in conjuncture with tttrlib to analyze and process single-molecule FRET (smFRET) experiments and confocal laser scanning microscopy (CLSM) data. The fit2x shared libary can be used from LabView and is wrapped by SWIG (Simplified Wrapper and Interface Generator) for common scripting languages as Python as main target language.

Design goals * Low memory footprint (keep objective large datasets, e.g. FLIM in memory). * Platform independent C/C++ library with interfaces for scripting libraries

Capabilities * Polarization and time-resolved analysis * Stable analysis results with minimum photon counts * Robust thoroughly tested maximum likelihood estimators

CHAPTER

TWO

DOCUMENTATION

Documentation for the latest releases is available here,. Previous releases are available through the read-the-docs page page,.

CHAPTER

THREE

CONTENTS

3.1 Installation

`fit2x` can either be installed from prebuilt binaries or from the source code. For most users it is recommended to install `fit2x` using the prebuilt binaries. Below the installation using prebuilt binaries and the prerequisites to compile `fit2x` are briefly outlined.

3.1.1 Prebuilt binaries

It is recommended to install `fit2x` for Python environments via `conda` by.

```
conda install -c tpeulen fit2x
```

Alternatively, `fit2x` can be installed via `pip`.

```
pip install fit2x
```

3.1.2 Compliation

`fit2x` can be compiled and installed using the source code provided in the git repository.

```
git clone --recursive https://github.com/tpeulen/fit2x.git
cd fit2x
sudo python setup.py install
```

To compile `fit2x` a set of prerequisites need to be fulfilled:

1. An installed compiler.
2. The HDF5 library with C/C++ include files.
3. A recent 64bit Python installation with include files.
4. cmake
5. SWIG

To debug and analyze the ouput of `fit2x` in more detail it can be useful to build the library in Debug mode that outputs additional information to the standard standard logging stream. fit2x can be build in debugging mode by.

```
python setup.py build_ext --debug
```

Windows

On windows `fit2x` is best compiled with the Visual Studio 2017. For compilation the Visual Studion Community edition is sufficient. In addition to Visual Studio the libraries and the include files as listed above need to be installed. The prebuilt binaries are compiled on Windows 10 with using 64bit anaconda Python environments miniconda using the conda build recipe that is provided with the source code in the `conda-recipe` folder.

macOS

For MacOS the prebuilt binaries are compiled on MacOS 10.13 with the Apple clang compiler using a anaconda distribution and the provided `conda-recipe`.

Linux

The Linux prebuilt binaries are compiled on Ubuntu 18.04 in an anaconda distribution and the provided `conda-recipe`.

Conda

A conda recipe is provided in the folder 'conda-recipe' to build the fit2x library with the conda build environment.

To build the library download the recipe, install the conda build package and use the provided recipe to build the library.

```
conda install conda-build
conda build conda-recipe
```

3.2 Fits

3.2.1 General

The fits covered by fit2x (e.g. fit23, fit24, and fit25) all have their own model function, target (objective) function, fit function, and parameter correction function. Briefly, the purpose of these functions is as follows.

Function	Description
Model function	Computes the model for a set of parameters
Target/objective function	Computes ob and returns a score for data
Fit function	Optimizes the parameters to the data
Correction function	Assures reasonable range of parameters

The model function computes for a set of input parameters the corresponding realization of the model (the model fluorescence decay curve(s)). The target function (more often called objective function) computes the disagreement of the data and the model function for a given set of model parameters. The fitting function optimizes a set of selected input parameters to the data by minimizing the disagreement of the data and the model. Finally, the estimates of the model parameters are corrected by a correction function. These functions have names that relate to the name of the model, e.g., *target23* corresponds to *fit23*. For computationally efficiency, the functions are hardcoded in C.

The models of the different fits differ in their parameters. The parameters of the fits are described in detail below. The target functions explicitly consider the counting noise and provide an estimate of the likelihood of the parameters. The initial values of the parameters are specified by the user. Next, the likelihood of the target is maximized by

the Limited-Memory Broyden-Fletcher-Goldfarb-Shanno Algorithm (L-BFGS). L-BFGS is an iterative method for solving unconstrained nonlinear optimization problems. Gradients for the L-BFGS algorithm are approximated by linear interpolation to handle the differentiable components, i.e., the model parameters that are free. The parameters of a model ca be either specified as free or as fixed. Only free parameters are optimized.

3.2.2 Python

For a use in python the `fit2x` module exposes a set of C functions that can be used to (1) compute model and (2) target/objective function and (3) optimize model parameters to experimental data. Besides the exposed functions the fit models are accessible via a simplified object-based interface that reduces the number of lines of code that need to be written for analyzing fluorescence decay histograms. The code blocks that are used below to illustrate the fit2x functionality are extracts from the tests located in the test folder of the fit2x repository. The test can be used as a more detailed refernce on how to use fit2x.

Model functions can be computed using the `modelf2x` function of the `fit2x` module. Here, the `x` represents a particular model function. The use of a `fit2x` model function is for the model function 23 (`modelf23`) below. model function

```

irf_np, time_axis = model_irf(
    n_channels=n_channels,
    period=period,
    irf_position_p=irf_position_p,
    irf_position_s=irf_position_s,
    irf_width=irf_width
)
dt = time_axis[1] - time_axis[0]
```

To compute a model, first a set of model parameters and a set of corrections need to be specified. All input parameters for `modelf23` are numpy arrays. In addition to the model parameters and the corrections `modelf23` requires an instrument response function (irf) and a background pattern. The model functions operate on numpy arrays and modify the numpy array for a model in-place. This means, that the output of the function is written to the input numpy-array of the model. In the example above the output is written to the array `model`.

To compute the value of a target for a realization of model parameters `fit2x` provides the functions `targetf2x`. The inputs of a target function (here `tarferf23`) are an numpy array containing the model parameter values and a structure that contains the corrections and all other necessary data to compute the value of the objective function (for fit23 i.e. data, irf, backgound, time resolution).

```
            [1.55512239e-06, 1.28478411e-06, 1.84127094e-03, 6.97704393e-01
                , 5.26292095e+00, 7.46022080e+00, 5.86203555e+00, 4.37710024e+00
                , 3.27767338e+00, 2.46136154e+00, 1.85332949e+00, 1.39904465e+00
                , 1.05863172e+00, 8.02832214e-01, 6.10103966e-01, 4.64532309e-01
                , 3.54320908e-01, 2.70697733e-01, 2.07119266e-01, 1.58689654e-01
                , 1.21735290e-01, 9.34921381e-02, 7.18751573e-02, 5.53076815e-02
                , 4.25947583e-02, 3.28288183e-02, 2.53192074e-02, 1.95393890e-02
                , 1.50872754e-02, 1.16553438e-02, 9.00806647e-03, 6.96482554e-03
                , 1.32357360e-06, 1.00072013e-05, 2.67318412e-02, 1.32397314e+00
```

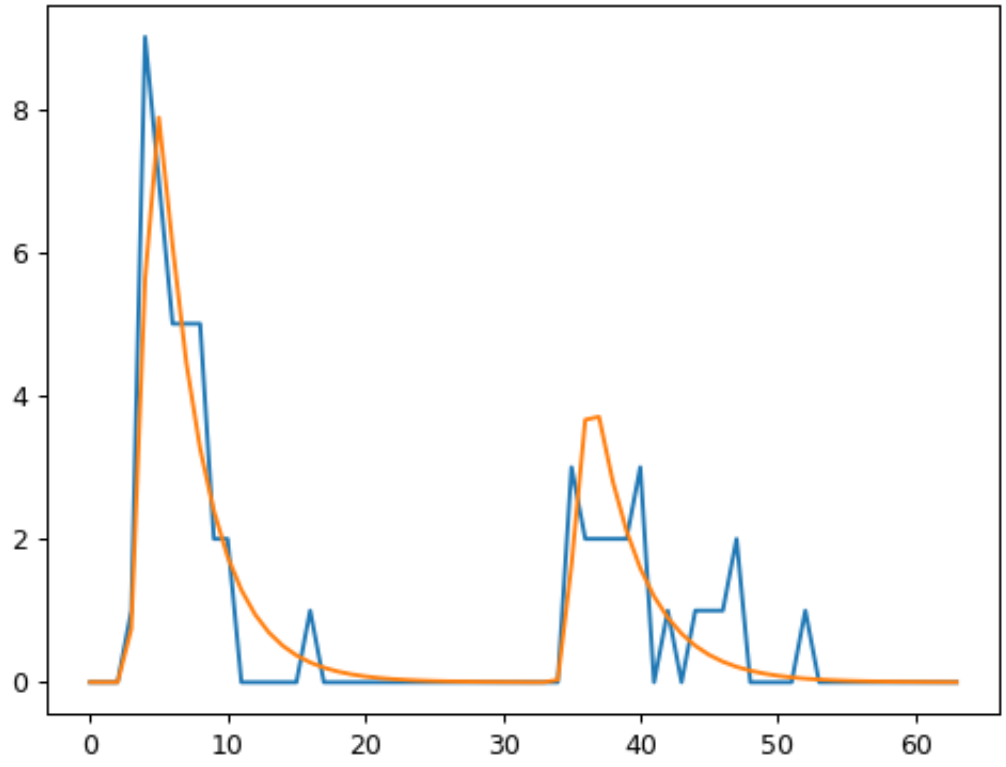
The data needed in addition to the model parameters are passed to the target function using `fit2x.MParam` objects that can be created by the `fit2x.CreateMParam` function from numpy arrays. The return value of a target function is the score of the model parameters

Model parameters can be optimized to the data by fit functions for fit 23 the fit function `fit2x.fit23` is used.

```
        # p.plot(data)
        # p.plot([x for x in m_param.get_data()])
        # p.show()
        self.assertEqual(np.allclose(fit_res, x), True)
```

The fit functions takes like the target function an object of the type `fit2x.MParam` in addition to the initial values, and a list of fixed model parameters as an input. The array containing the initial values of the model parameters are modified in-place buy the fit function.

Alternatively, a simplified python interface can be used to optimize a set of model as illustrated by the source code embedded in the plot below. The simplified interface handles the creation of auxiliary data structures such as `fit2x.MParam`.



In the example shown above, first a fit object of the type `fit2x.Fit23` is created. All necessary data except for the experimental data for a fit is passed to the fit object when it is created. To perform a fit on experimental data for a set for a set of initial values, the fit object is called using the inital values and the data as parameters.

3.3 Description of fits

3.3.1 fit23

fit23 optimizes a single rotational correlation time ρ and a fluorescence lifetime τ to a polarization resolved fluorescence decay considering the fraction of scattered light and instrument response function in the two detection channels for the parallel and perpendicular fluorescence. Fit23 operates on fluorescence decays in the *Jordi-format*.

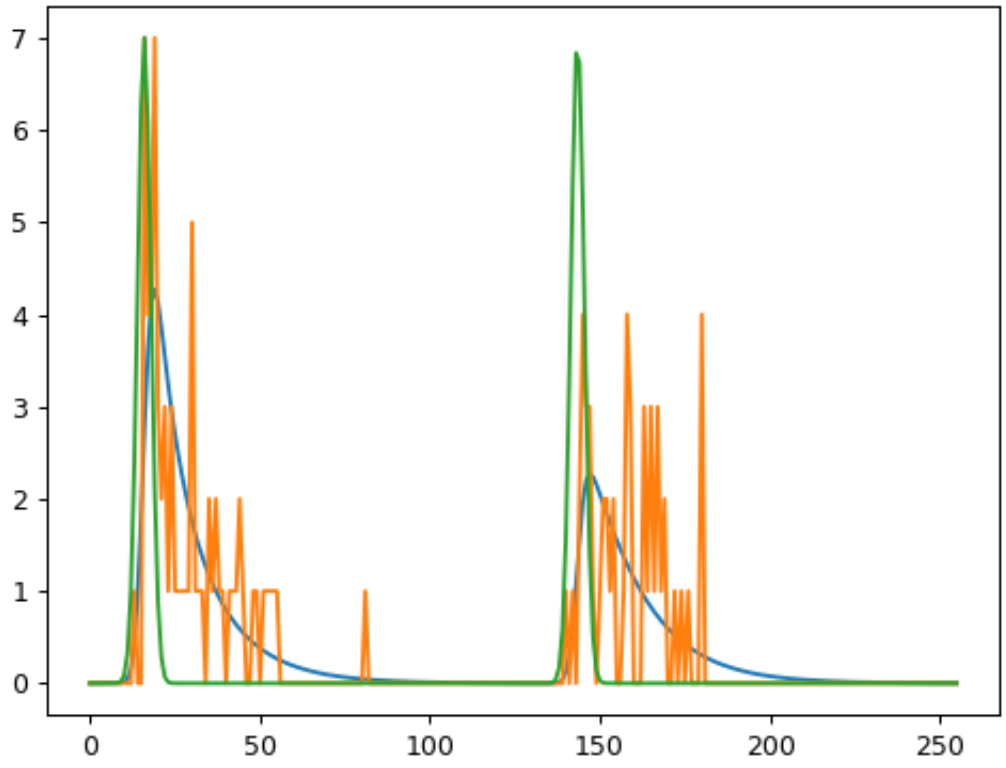


Fig. 1: Simulation and analysis of low photon count data by `fit2x.fit23`.

fit23 is intended to be used for data with very few photons, e.g. for pixel analysis in fluorescence lifetime image microscopy (FLIM) or for single-molecule spectroscopy. The fit implements a maximum likelihood estimimator as previously described [MCH+01]. Briefly, the MLE fit quality parameter 2I* = $-2 \ln L(n, g)$ (where L is the likelihood function, n are the experimental counts, and g is the model function) is minimized. The model function g under magic-angle is given by:

$$g_i = N_g \left[(1-\gamma) \frac{irf_i * \exp(iT/k\tau)+c}{\sum_{i=0}^{k} irf_i * \exp(iT/k\tau)+c} + \gamma \frac{bg_i}{\sum_i^k bg_i} \right]$$

N_e is not a fitting parameter but set to the experimental number of photons N, $*$ is the convolution operation, τ is the fluorescence lifetime, irf is the instrument response function, i is the channel number, bg_i is the background count in the channel i,

The convolution by fit23 is computed recusively and accounts for high repetition rates:

:math:`irf_i ast exp(iT/ktau) = sum_{j=1}^{min(i,l)}irf_jexp(-(i-j)T/ktau) + sum_{j=i+1}^{k}irf_jexp(-(i+k-j)T/ktau)
`

The anisotropy treated as previously described [SVE+99]. The correction factors needed for a correct anisotropy used by `fit2x` are defined in the glossary (*Anisotropy*).

3.3.2 fit24

Fit24 optimizes is a bi-exponential model function with two fluorescence lifetimes τ_1, τ_2, and amplitude of the second lifetime a_2, the fraction scattered light γ, and a constant offset to experimental data. Fit24 does not describe anisotropy. The decays passed to the fit in the Jordi format. The two decays in the Jordi array are both treated by the same model function and convoluted with the corresponding *instrument response function*. The model function is

where Δt is the time per micro time channel, i is the micro time channel number, a_2 is the fraction of the second species. The model function is corrected for the fraction of background and a constant offset.

Where, c is a constant offset, B the background pattern, M the model function and γ the fraction of scattered light.

3.3.3 fit25

Selects the lifetime out of a set of 4 fixed lifetimes that best describes the data. Works with polarization resolved Jordi stacks, computes rotational correlation time by the anisotropy. This function selects out of a set of 4 lifetimes tau the lifetime that fits best the data.

3.3.4 fit26

Pattern fit. Determines the fraction f of two mixed patterns.

(No convolution of patterns, area of pattern is normalized by fit)

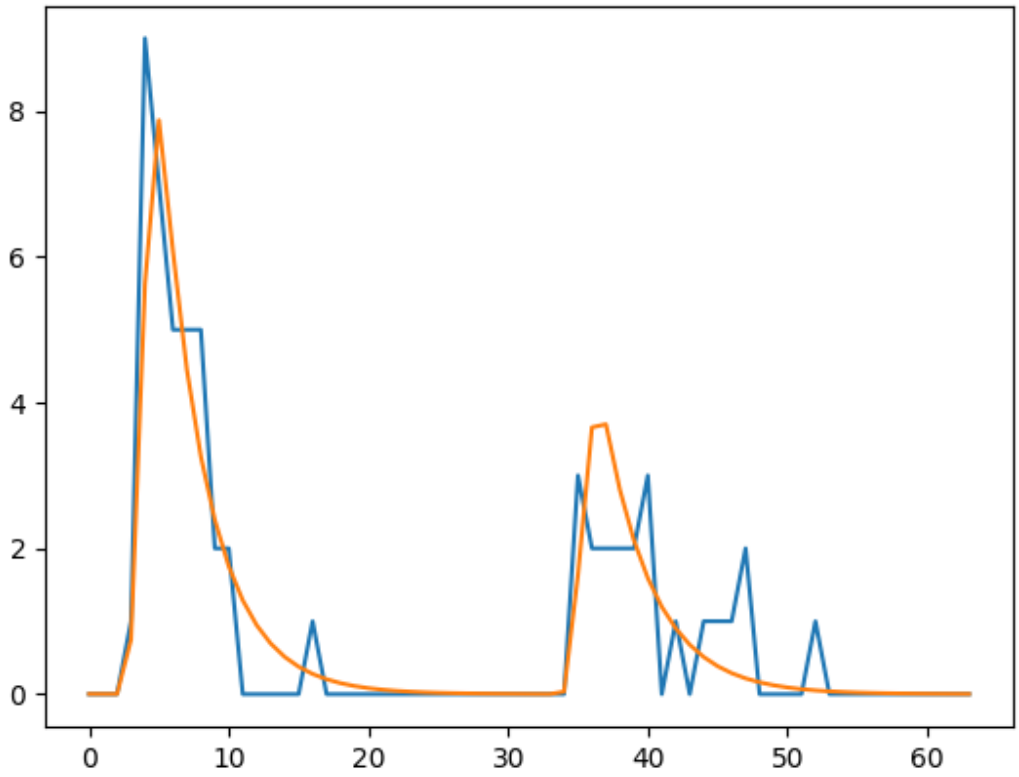
3.4 Applications

3.4.1 Fluorescence decay

Examples

Fluorescence decay applications.

MLE mini example - fit23




```
import fit2x
import numpy as np
import pylab as p

irf = np.array(
    [0, 0, 0, 260, 1582, 155, 0, 0, 0, 0,
     0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
     0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
     0, 0, 0, 0, 22, 1074, 830, 10, 0, 0,
     0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
     0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
     0, 0, 0, 0], dtype=np.float64
)

data = np.array(
    [
        0, 0, 0, 1, 9, 7, 5, 5, 5, 2, 2, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 3, 2, 2, 2, 2, 3, 0, 1, 0,
        1, 1, 1, 2, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
    ]
)
```

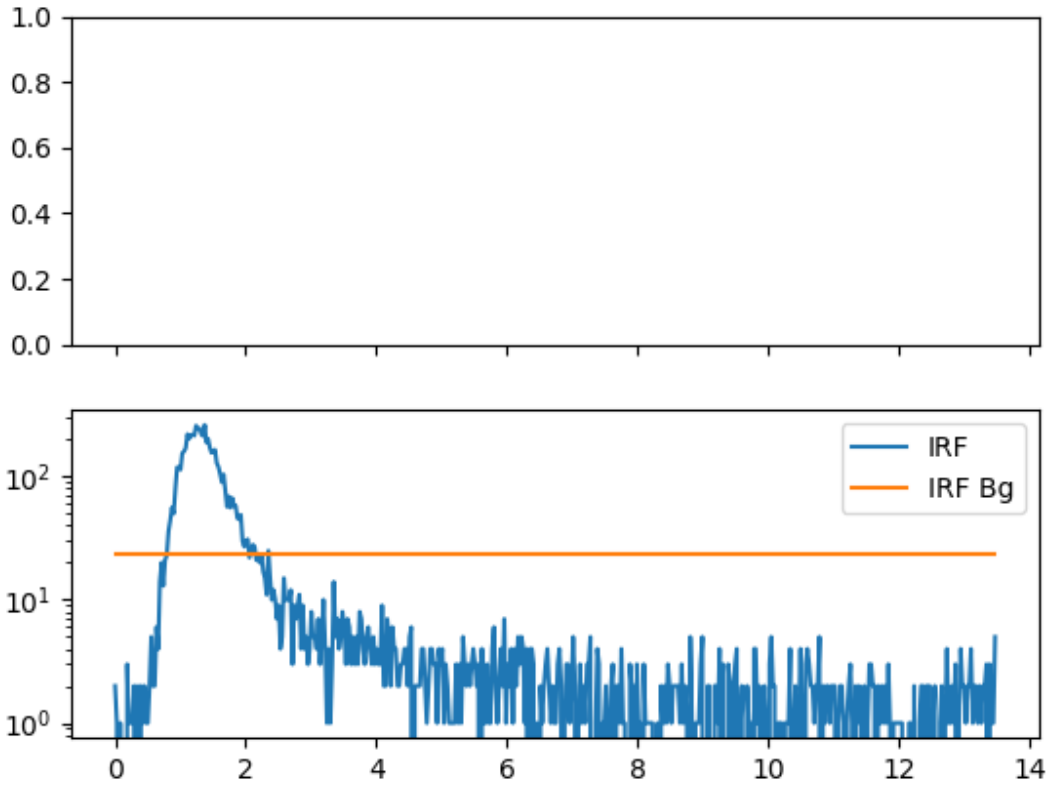
(continues on next page)

(continued from previous page)

```
settings = {
    'dt': 0.5079365079365079,
    'g_factor': 1.0,
    'l1': 0.1,
    'l2': 0.2,
    'convolution_stop': 31,
    'irf': irf,
    'period': 16.0,
    'background': np.zeros_like(irf)
}

fit23 = fit2x.Fit23(**settings)

tau, gamma, r0, rho = 2.2, 0.01, 0.38, 1.22
x0 = np.array([tau, gamma, r0, rho])
fixed = np.array([0, 1, 1, 0])
r = fit23(
    data=data,
    initial_values=x0,
    fixed=fixed
)

p.plot(fit23.data, label='data')
p.plot(fit23.model, label='model')
p.show()
```

Total running time of the script: (0 minutes 0.159 seconds)

Fluorescence decay analysis - 3




```
import pylab as p
import scipy.optimize
import scipy.stats
import numpy as np
import tttrlib


def objective_function(
        x: np.ndarray,
        x_min: int,
        x_max: int,
        irf: np.array,
        max_irf_fraction: float = 0.1,
        verbose: bool = False
):
    max_irf = np.max(irf) * max_irf_fraction
    w = np.copy(irf[x_min:x_max])
    w[w < 1] = 1
    w[np.where(irf > max_irf)[0]] = 1
    chi2 = (((irf[x_min:x_max] - x[0])/w)**2).sum(axis=0)
    return chi2
```

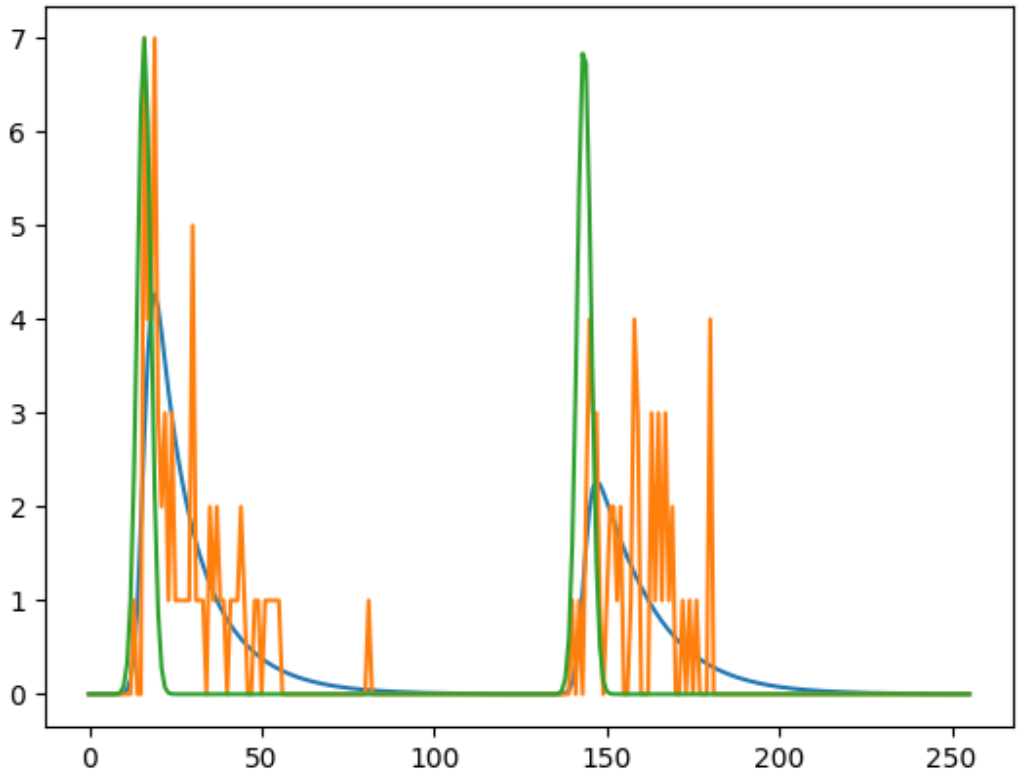
(continues on next page)

(continued from previous page)

```
# Read TTTR
spc132_filename = '../../test/data/bh_spc132_sm_dna/m000.spc'
data = tttrlib.TTTR(spc132_filename, 'SPC-130')
data_green = data[data.get_selection_by_channel([0, 8])]
# the macro time clock in ms
macro_time_resolution = data.header.macro_time_resolution / 1e6

# Make histograms
n_bins = 512  # number of bins in histogram
x_min, x_max = 1, 512  # fit range
dt = data.header.micro_time_resolution * 4096 / n_bins  # time resolution
time_axis = np.arange(0, n_bins) * dt

# IRF
# select background / scatter, maximum 7 photons in 6 ms
time_window_irf = 6.0
n_ph_max_irf = 7
irf, _ = np.histogram(
    data_green[
        data_green.get_selection_by_count_rate(
            time_window=time_window_irf,
            n_ph_max=n_ph_max_irf
        )
    ].micro_times, bins=n_bins
)

x0 = np.array([4])
fit = scipy.optimize.minimize(
    objective_function, x0,
    args=(x_min, x_max, irf),
    method='BFGS'
)
irf_bg = fit.x


fig, ax = p.subplots(nrows=2, ncols=1, sharex=True, sharey=False)
ax[1].semilogy(time_axis, irf, label="IRF")
ax[1].semilogy(
    time_axis[x_min:x_max],
    np.ones_like(time_axis)[x_min:x_max] * irf_bg, label="IRF Bg"
)
ax[1].legend()
p.show()
```

Total running time of the script: (0 minutes 0.768 seconds)

MLE - Fit23




```
import fit2x
import numpy as np
import scipy.stats
import pylab as p


def model_irf(
        n_channels: int = 256,
        period: float = 32,
        irf_position_p: float = 2.0,
        irf_position_s: float = 18.0,
        irf_width: float = 0.25
):
    time_axis = np.linspace(0, period, n_channels * 2)
    irf_np = scipy.stats.norm.pdf(time_axis, loc=irf_position_p, scale=irf_width) + \
             scipy.stats.norm.pdf(time_axis, loc=irf_position_s, scale=irf_width)
    return irf_np, time_axis


# setup some parameters
n_channels = 128
n_corrections = 5
```

(continues on next page)

(continued from previous page)

```
n_photons = 120
irf_position_p = 2.0
irf_position_s = 18.0
irf_width = 0.25
period, g, l1, l2, conv_stop = 32, 1.0, 0.1, 0.1, n_channels // 2 - 1
tau, gamma, r0, rho = 2.0, 0.01, 0.38, 1.2
np.random.seed(0)

irf_np, time_axis = model_irf(
    n_channels=n_channels,
    period=period,
    irf_position_p=irf_position_p,
    irf_position_s=irf_position_s,
    irf_width=irf_width
)
dt = time_axis[1] - time_axis[0]
conv_stop = min(len(time_axis), conv_stop)
param = np.array([tau, gamma, r0, rho])
corrections = np.array([period, g, l1, l2, conv_stop])

# compute a model function that is later used as "data"
model = np.zeros_like(time_axis)
bg = np.zeros_like(time_axis)
fit2x.modelf23(param, irf_np, bg, dt, corrections, model)
# add poisson noise to model and use as data
data = np.random.poisson(model * n_photons)

# create MParam structure that contains all parameters for fitting
m_param = fit2x.CreateMParam(
    irf=irf_np,
    background=bg,
    data=data.astype(np.int32),
    corrections=corrections,
    dt=dt
)

tau, gamma, r0, rho = 4., 0.01, 0.38, 1.5
bifl_scatter = -1
p_2s = 0
x = np.zeros(8, dtype=np.float64)
x[:6] = [tau, gamma, r0, rho, bifl_scatter, p_2s]

# test fitting
fixed = np.array([0, 1, 1, 1], dtype=np.int16)
chi2 = fit2x.fit23(x, fixed, m_param)

m = np.array([m for m in m_param.get_model()])
p.plot(m)
p.plot(data)
p.plot(irf_np / max(irf_np) * max(data))
p.show()
```

Total running time of the script: (0 minutes 0.133 seconds)

Fluorescence decay analysis - 2

```
import pylab as p
import scipy.optimize
import numpy as np
import tttrlib
import fit2x


def objective_function_chi2(
        x: np.ndarray,
        decay_object: fit2x.Decay,
        x_min:int = 20,
        x_max:int = 150
):
    time_shift, scatter, lifetime_spectrum = x[0], x[1], x[2:]
    scatter = abs(scatter)
    lifetime_spectrum = np.abs(lifetime_spectrum)
    decay_object.lifetime_spectrum = lifetime_spectrum
    decay_object.scatter_fraction = scatter
    decay_object.irf_shift_channels = time_shift
    wres = decay_object.weighted_residuals
    return np.sum(wres[x_min:x_max]**2)


# load file
spc132_filename = '../../test/data/bh_spc132_sm_dna/m000.spc'
data = tttrlib.TTTR(spc132_filename, 'SPC-130')
ch0_indeces = data.get_selection_by_channel([0, 8])
data_ch0 = data[ch0_indeces]

# selection low count rate region
tttr_low_count_rate = data_ch0[
    data_ch0.get_selection_by_count_rate(
        time_window=6.0,
        n_ph_max=7,
        invert=False
    )
]

# Select high count region
tttr_high_count_rate = data_ch0[
    data_ch0.get_selection_by_count_rate(
        time_window=0.005,
        n_ph_max=2,
        invert=True
    )
]
```

Make decay object

```
decay_object = fit2x.Decay(
    tttr_data=tttr_high_count_rate,
    tttr_irf=tttr_low_count_rate,
    tttr_micro_time_coarsening=16,
)
time_axis = decay_object.get_time_axis()

# Define fit range
# The BH card are not linear at the end of the TAC
# Thus, do not the full range
x_min, x_max = 40, 200


# inital parameters for fit
time_shift = 0
scatter = 0.001
mean_tau = tttrlib.TTTR.compute_mean_lifetime(
    tttr_data=tttr_high_count_rate,
    tttr_irf=tttr_low_count_rate
)

lifetime_spectrum = [0.5, 0.8, 0.5, 3.0]
```

Optimize

```
x0 = np.hstack(
    [
        time_shift,
        scatter,
        lifetime_spectrum
    ]
)
fit = scipy.optimize.minimize(
    objective_function_chi2, x0,
    args=(decay_object, x_min, x_max),
)
```

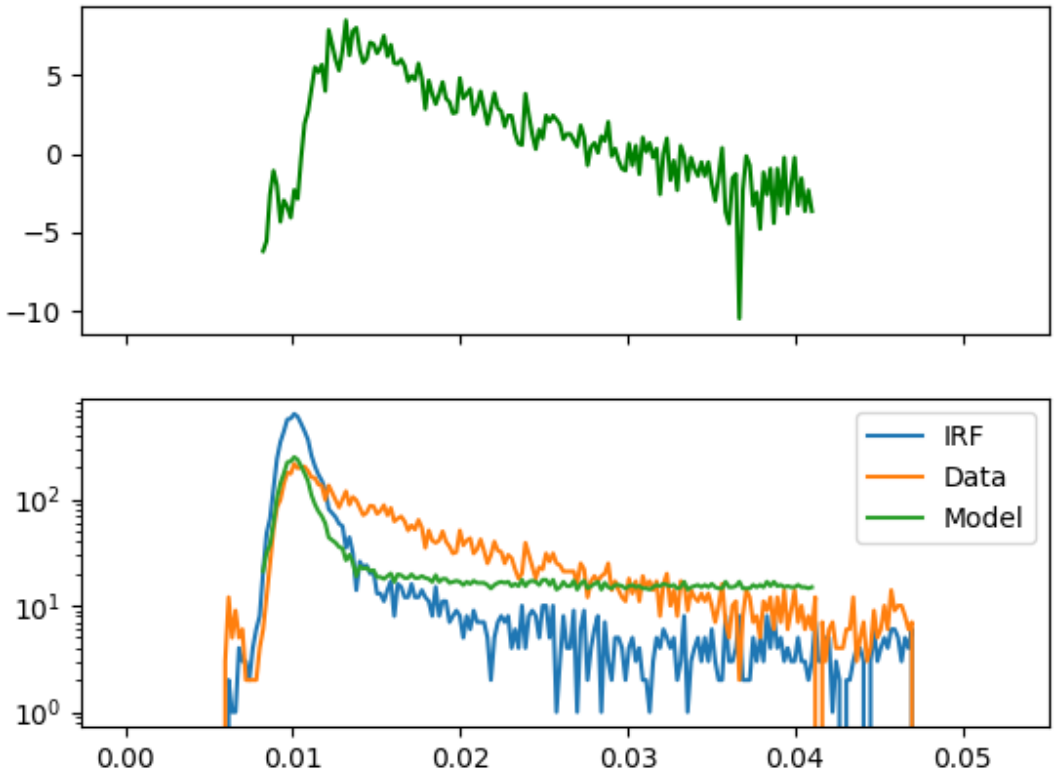
Plot

```
wres = decay_object.get_weighted_residuals()
fig, ax = p.subplots(nrows=2, ncols=1, sharex=True, sharey=False)
ax[1].semilogy(time_axis, decay_object.get_irf(), label="IRF")
ax[1].semilogy(time_axis, decay_object.get_data(), label="Data")
ax[1].semilogy(time_axis[x_min:x_max], decay_object.get_model()[x_min:x_max], label=
↪"Model")
ax[1].legend()
ax[0].plot(
    time_axis[x_min:x_max], wres[x_min:x_max],
    label='w.res.',
    color='green'
```

(continues on next page)

(continued from previous page)

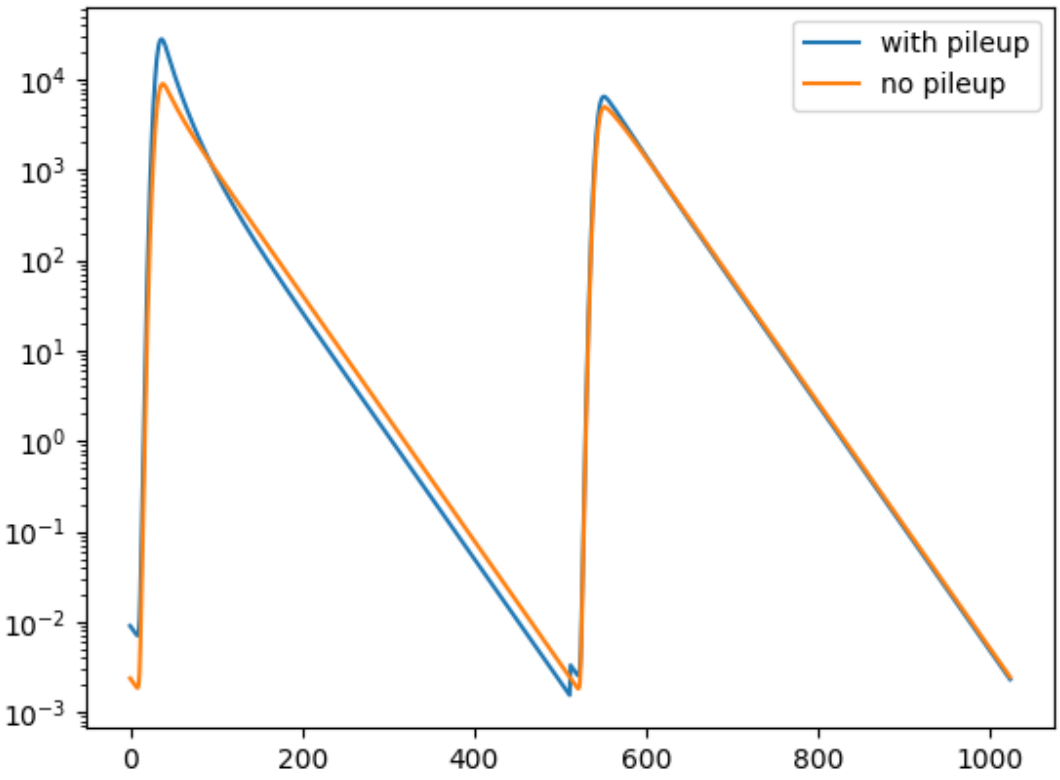
```
)
p.show()
```



Total running time of the script: (0 minutes 0.322 seconds)

Pile-up

This example illustrates the effect of pile up on the shape of fluorescence decay curves and demonstrates how the shape of a model function can be modified to account for pile up.




```
import pylab as p
import numpy as np
import scipy.stats
import fit2x

# Set seed to make results reproducible
np.random.seed(42)

# Define number of channels and excitation period
n_channels = 512
period = 32
time_axis = np.linspace(0, period, n_channels)
conv_stop = n_channels // 2

# compute a irf
irf_width = 0.25
irf_position = 2.0
dt = time_axis[1] - time_axis[0]
irf_p = scipy.stats.norm.pdf(time_axis, loc=irf_position, scale=irf_width)
irf_s = irf_p

"""
In this example first an ideal fluorescence decay is computed for the parallel (p)
and perpendicular (s) channel using fit23.
"""
```

(continues on next page)

(continued from previous page)

```
# Compute model fo a single photon
n_photons = 1
period, g, l1, l2 = 32, 1.0, 0.1, 0.1
tau, gamma, r0, rho = 2.0, 0.01, 0.38, 1.2

model = np.zeros(n_channels * 2)  # For parallel and perpendicualr
bg = np.zeros_like(model)
fit2x.modelf23(
    np.array([tau, gamma, r0, rho]),
    np.hstack([irf_p, irf_s]),
    bg, dt,
    np.array([period, g, l1, l2, conv_stop]),
    model
)

"""
The number the model is scaled to a number of photons typically recorded in a eTCSPC
experiment. In an eTCSPC experiment usually 1e6 - 20e6 photons are recorded.
"""

n_photons = 5.5e5
model *= n_photons

# Add pileup to parallel (p) and perpendicular (s)
model_p = model[:len(model) // 2]
model_p_with_pileup = np.copy(model_p)
model_s = model[len(model) // 2:]
model_s_with_pileup = np.copy(model_s)

"""
The pile up depends on the instrument dead time, the repetition rate used to excite
the sample and the total number of registered photons. Here, to highlight the effect
of pile up an unrealistic combination of the measurement time and the number of␣
↪photons
is used. In modern counting electronics the dead time is often around 100 nano
seconds.

In this example there is no data. Thus, the model without pile up is used as
"data". In a real experiment use the experimental histogram in the data argument.
"""

pile_up_parameter = {
    'repetition_rate': 1./period * 1000,  # Repetition rate is in MHz
    'instrument_dead_time': 120.,  # Instrument dead time nano seconds
    'measurement_time': 0.05,  # Measurement time in seconds
    'pile_up_model': "coates"
}

fit2x.add_pile_up_to_model(
    model=model_p_with_pileup,
    data=model_p,
    **pile_up_parameter
)
fit2x.add_pile_up_to_model(
    model=model_s_with_pileup,
    data=model_s,
```

(continues on next page)

(continued from previous page)

```
    **pile_up_parameter
)

model_ps_pile_up = np.hstack([model_p_with_pileup, model_s_with_pileup])
p.semilogy(model_ps_pile_up, label='with pileup')
p.semilogy(model, label='no pileup')
p.legend()
p.show()
```

Total running time of the script: (0 minutes 0.305 seconds)

Convolution routines

Fast convolution

Overview

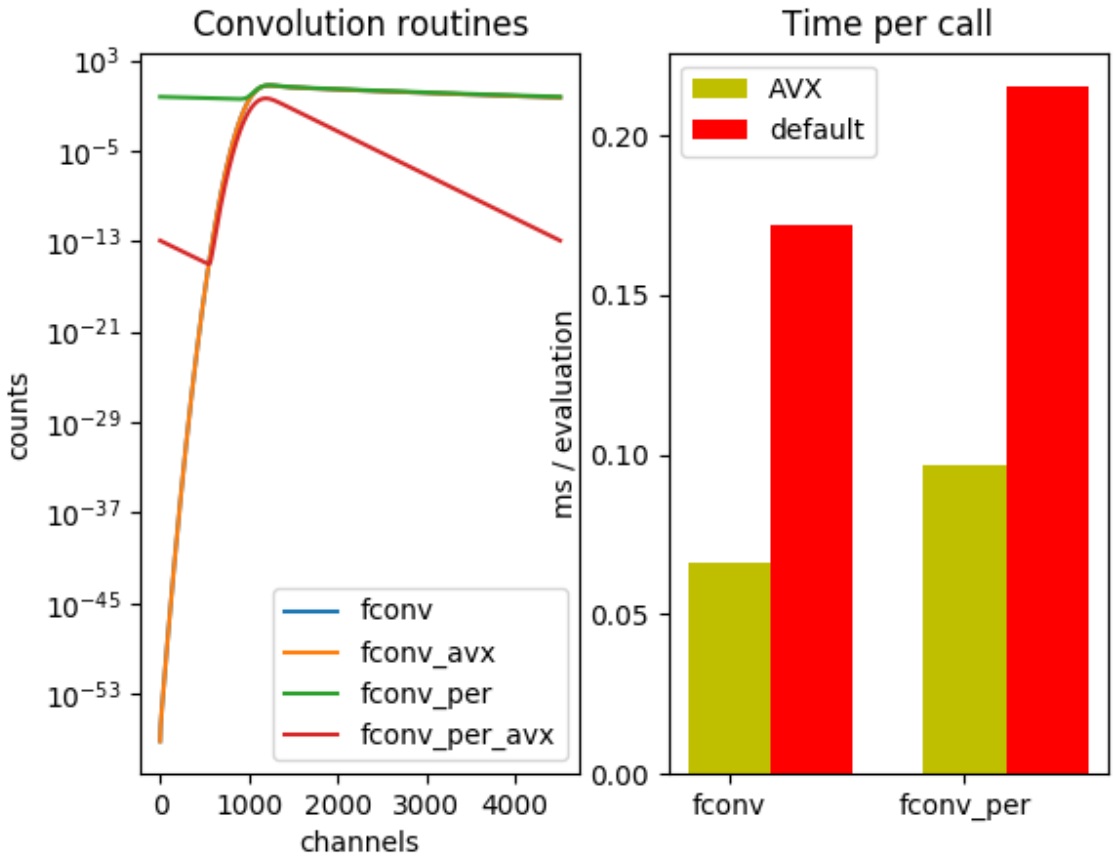
In `fit2x` there are a set of different routines that can be used to compute fluorescence decays. Most fluorescence decays are linear combinations of exponential decays. Convolutions of such fluorescence decays with instrument response functions can be computed using the routines (fconv, fconv_per, fconv_per_cs, etc.). Here, fconv stands for fast convolution.

Advanced vector extension

For faster convolutions `fit2` provides routines that make use of SIMD (Single Instruction Multiple Data). The SIMD routines use of the AVX2 extension (Advanced Vector Extension). For instance, the routines `fconv` and `fconv_per` (periodic convolution) have corresponding SIMD routines named `fconv_avx``and ``fconv_per_avx`. The SIMD routines compute in parallel the decay in cases the decay is compose of more than a single fluorescence lifetime.

The SIMD AVX make use of AVX2 and FMA (Fused Multiply Add). AVX2 and FMA require CPUs with 'modern' instruction sets. Most x86 CPUs that were manufacture since 2012 are supported.

Benchmark




```
from __future__ import annotations
import fit2x
import scipy.stats
import time
import numpy as np
import pylab as p


period = 16
n_channels = 4505
irf_position = 4.0
irf_width = 0.25
time_axis = np.linspace(0, period, n_channels)
irf = scipy.stats.norm.pdf(time_axis, loc=irf_position, scale=irf_width)

dt = time_axis[1]-time_axis[0]
lifetime_spectrum = np.array([1., 10] + ([1., 4, 1., 0.4] * 4))
model = np.zeros_like(irf)
stop = len(irf)
start = 0
n_runs = 50
```

(continues on next page)

(continued from previous page)

```
# benchmark
################
times = list()
times_avx = list()
names = ["fconv", "fconv_per"]
t_start = time.perf_counter()
for _ in range(n_runs):
    fit2x.fconv(fit=model, irf=irf, x=lifetime_spectrum, start=start, stop=stop,␣
↪dt=dt)
ex = time.perf_counter() - t_start
times.append(ex)

t_start = time.perf_counter()
for _ in range(n_runs):
    fit2x.fconv_avx(fit=model, irf=irf, x=lifetime_spectrum, start=start, stop=stop,␣
↪dt=dt)
ex = time.perf_counter() - t_start
times_avx.append(ex)

t_start = time.perf_counter()  # in seconds
for _ in range(n_runs):
    fit2x.fconv_per(fit=model, irf=irf, x=lifetime_spectrum, period=period,␣
↪start=start, stop=stop, dt=dt)
ex = time.perf_counter() - t_start
times.append(ex)

t_start = time.perf_counter()
for _ in range(n_runs):
    fit2x.fconv_per_avx(fit=model, irf=irf, x=lifetime_spectrum, period=period,␣
↪start=start, stop=stop, dt=dt)
ex = time.perf_counter() - t_start
times_avx.append(ex)

times = np.array(times) * 1000.0 / n_runs
times_avx = np.array(times_avx) * 1000.0 / n_runs


# make plots
##################
fig, ax = p.subplots(nrows=1, ncols=2, sharex=False)

ax[0].set_title('Convolution routines')
ax[0].set_ylabel('counts')
ax[0].set_xlabel('channels')

model = np.zeros_like(irf)
fit2x.fconv(fit=model, irf=irf, x=lifetime_spectrum, start=start, stop=stop, dt=dt)
ax[0].semilogy(model, label="fconv")

model_avx = np.zeros_like(irf)
fit2x.fconv_avx(fit=model_avx, irf=irf, x=lifetime_spectrum, start=start, stop=stop,␣
↪dt=dt)
ax[0].semilogy(model, label="fconv_avx")

model = np.zeros_like(irf)
fit2x.fconv_per(fit=model, irf=irf, x=lifetime_spectrum, period=period, start=start,␣
↪stop=stop, dt=dt)
```

(continues on next page)

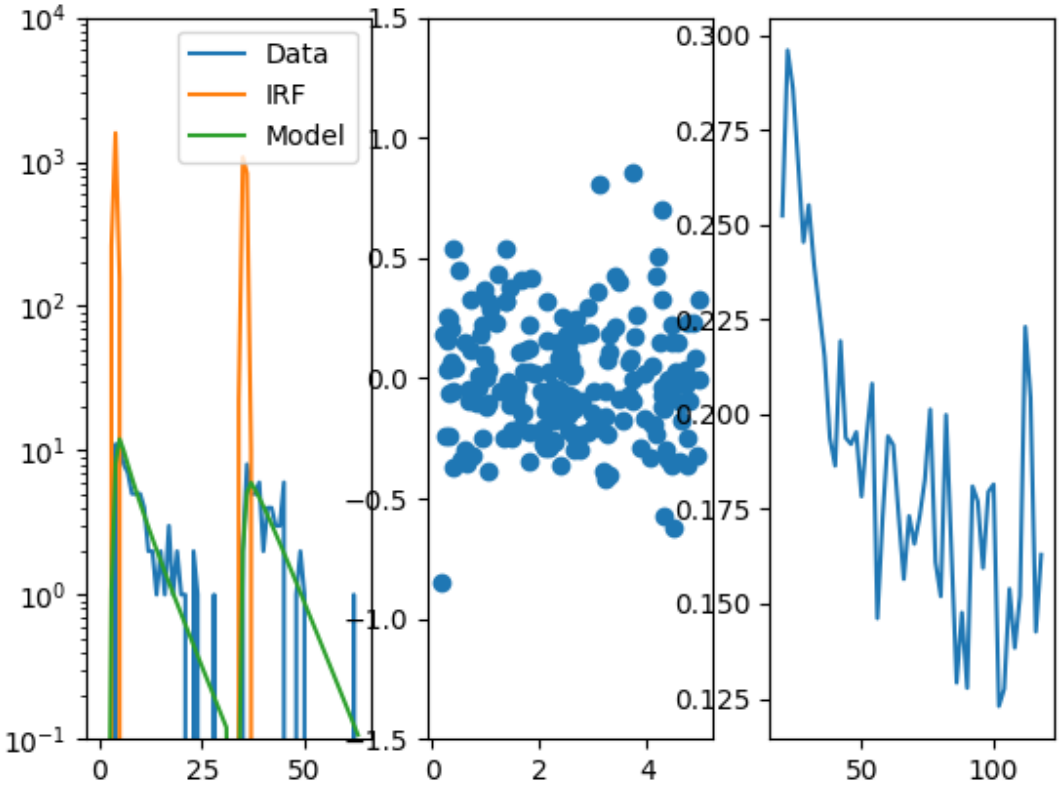
(continued from previous page)

```
ax[0].semilogy(model, label="fconv_per")

model = np.zeros_like(irf)
fit2x.fconv_per_avx(fit=model, irf=irf, x=lifetime_spectrum, period=period,␣
↪start=start, stop=stop, dt=dt)
ax[0].semilogy(model, label="fconv_per_avx")
ax[0].legend()

# Benchmark
ax[1].set_title('Benchmark')
ax[1].set_ylabel('ms / evaluation')

ind = np.arange(len(times))  # the x locations for the groups
ax[1].set_title('Time per call')
ax[1].set_xticks(ind)
ax[1].set_xticklabels(names)

width = 0.35
ax[1].bar(ind, times_avx, width, color='y', label='AVX')
ax[1].bar(ind + width, times, width, color='r', label='default')
ax[1].legend()

p.show()
```

Total running time of the script: (0 minutes 0.301 seconds)

Benchmark MLE - fit23

Generate n_samples random decays in range (tau_min, tau_max) with n_photons photons and fits lifetime. Compares recovered lifetime with fitted lifetime




```
import fit2x
import numpy as np
import pylab as p
np.random.seed(0)

# setup some parameters
tau_min = 0.1
tau_max = 5.0

n_photons_min = 20
n_photons_max = 120
n_photon_step = 2

n_samples = 200

n_channels = 32
irf_position_p = 2.0
irf_position_s = 18.0
irf_width = 0.25
period, g_factor, l1, l2, conv_stop = 32, 1.0, 0.1, 0.1, 31
dt = 0.5079365079365079
np.random.seed(0)

irf_np = np.array([0, 0, 0, 260, 1582, 155, 0, 0, 0, 0,
                   0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

(continues on next page)

(continued from previous page)

```
                    0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                    0, 0, 0, 0, 22, 1074, 830, 10, 0, 0,
                    0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                    0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                    0, 0, 0, 0], dtype=np.float64)

bg = np.zeros_like(irf_np)

p.show()
fit23 = fit2x.Fit23(
    dt=dt,
    irf=irf_np,
    background=bg,
    period=period,
    g_factor=g_factor,
    l1=l1, l2=l2
)

tau, gamma, r0, rho = 2.0, 0.01, 0.38, 1.22
x0 = np.array([tau, gamma, r0, rho])
fixed = np.array([0, 1, 1, 0])


n_photon_dict = dict()
for n_photons in range(n_photons_min, n_photons_max, n_photon_step):
    tau_sim = list()
    tau_recov = list()
    n_photons = int(n_photons)
    for i in range(n_samples):
        tau = np.random.uniform(tau_min, tau_max)
        # n_photons = int(5./tau * n_photon_max)
        param = np.array([tau, gamma, r0, rho])
        corrections = np.array([period, g_factor, l1, l2, conv_stop])
        model = np.zeros_like(irf_np)
        bg = np.zeros_like(irf_np)
        fit2x.modelf23(param, irf_np, bg, dt, corrections, model)
        model *= n_photons / np.sum(model)
        data = np.random.poisson(model)
        r = fit23(
            data=data,
            initial_values=x0,
            fixed=fixed
        )
        # print("tau_sim: %.2f, tau_recov: %s" % (tau, r['x'][0]))
        tau_sim.append(tau)
        tau_recov.append(r['x'][0])
        n_photon_dict[n_photons] = {
                'tau_simulated': np.array(tau_sim),
                'tau_recovered': np.array(tau_recov)
        }

devs = list()
for k in n_photon_dict:
    tau_sim = n_photon_dict[k]['tau_simulated']
    tau_recov = n_photon_dict[k]['tau_recovered']
    dev = (tau_recov - tau_sim) / tau_sim
    devs.append(dev)
```

(continues on next page)

(continued from previous page)

```
fig, ax = p.subplots(nrows=1, ncols=3)
ax[0].semilogy([x for x in fit23._m_param.get_data()], label='Data')
ax[0].semilogy([x for x in fit23._m_param.get_irf()], label='IRF')
ax[0].semilogy([x for x in fit23._m_param.get_model()], label='Model')
ax[0].set_ylim((0.1, 10000))
ax[0].legend()
k = list(n_photon_dict.keys())[0]
tau_sim = n_photon_dict[k]['tau_simulated']
tau_recov = n_photon_dict[k]['tau_recovered']
dev = (tau_recov - tau_sim) / tau_sim
ax[1].plot(tau_sim, dev, 'o', label='#Photons: %s' % k)
ax[1].set_ylim((-1.5, 1.5))
ax[0].legend()
sq_dev = np.array(devs)**2
ax[2].plot(list(n_photon_dict.keys()), np.sqrt(sq_dev.mean(axis=1)))
p.show()
```

Total running time of the script: (0 minutes 2.246 seconds)

Fluorescence decay analysis - 1

Introduction

In time-resolved fluorescence experiments the fluorescence intensity decay of a sample is monitored. To date this is mostly accomplished by repeatedly exciting the sample by a short laser pulse and recording the time between the sample excitation and the detection of photons. In such pulsed experiments the micro time in a `TTTR` object encodes the time between the excitation pulse and the detection of the photon. In analogue counting electronics is mostly operated in an inverse mode where the time between the photon and the subsequent detection pulse is recorded. After multiple photons have been recorded fluorescence decay histograms are computed. The shape of these histograms corresponds to the underlying fluorescence decay, f(t).

In many cases fluorescence decays can be described by linear combinations of exponential decays. The `Decay` class of `tttrlib` computes models for fluorescence decays, f(t), fluorescence intensity decay histograms, h(t), and to scores models against experimental decay histograms. The `Decay` class handles typical artifacts encountered in fluorescence decays such as scattered light, constant backgrounds, convolutions with the instrument response function [OConnor12], pile-up artifacts [Coa68], differential non-linearity of the micro time channels [Bec05] and more.

Below the basic usage of the `Decay` class is outlined with a few application examples. These examples can be used as starting point for custom analysis pipelines, e.g. for burst-wise single-molecule analysis, pixel-wise FLIM analysis or analysis over larger ensembles of molecules or pixels.

Decay histograms

Decay class

Fluorescence decays can be either computed using the static method provided by the `Decay` class or Create an instance the `Decay` class

Single-molecule

```
import matplotlib.pylab as plt
import scipy.optimize
import scipy.stats
import numpy as np
import tttrlib
import fit2x


def objective_function_chi2(
        x: np.ndarray,
        decay_object: fit2x.Decay,
        x_min: int = 20,
        x_max: int = 150
):
    scatter, background, time_shift, irf_background = x[0:4]
    lifetime_spectrum = x[4:]
    decay_object.set_lifetime_spectrum(lifetime_spectrum)
    decay_object.irf_background_counts = irf_background
    decay_object.scatter_fraction = scatter
    decay_object.constant_offset = background
    decay_object.irf_shift = time_shift
    # wres = decay_object.get_weighted_residuals()
    # return np.sum(wres[x_min:x_max]**2)
    return decay_object.get_score(x_min, x_max)


def objective_function_mle(
        x: np.ndarray,
        x0: np.ndarray,
        decay_object: fit2x.Decay,
        x_min: int = 20,
        x_max: int = 500,
        use_amplitude_prior: bool = True,
        use_initial_prior: bool = True,
        amplitude_bias: float = 5.0,
        initial_sd: float = 5.0,
        verbose: bool = False
):
    scatter, background, time_shift, irf_background = x[0:4]
    lifetime_spectrum = np.abs(x[4:])
    decay_object.set_lifetime_spectrum(lifetime_spectrum)
    decay_object.irf_background_counts = irf_background
    decay_object.scatter_fraction = scatter
    decay_object.constant_offset = background
    decay_object.irf_shift = time_shift
    chi2_mle = decay_object.get_score(x_min, x_max, score_type="poisson")
    # d = decay_object.get_data()[x_min:x_max]
    # m = decay_object.get_model()[x_min:x_max]
    # return np.sum((m - d) - d * np.log(m/d))
    ap = 0.0
    if use_amplitude_prior:
        p = lifetime_spectrum[::2]
        a = np.ones_like(p) + amplitude_bias
        ap = scipy.stats.dirichlet.logpdf(p / np.sum(p), a)
        chi2_mle += ap
    ip = 0.0
    if use_initial_prior:
```

(continues on next page)

(continued from previous page)

```
        ip = -np.sum(np.log(1./((1+((x - x0) / initial_sd)**2.0))))
        chi2_mle += ip
    if verbose:
        print("Total log prob: %s" % chi2_mle)
        print("Parameter log prior: %s" % ip)
        print("Dirichlet amplitude log pdf: %s" % ap)
        print("---")
    return chi2_mle
```

Prepare data

```
# load file
spc132_filename = '../../test/data/bh_spc132_sm_dna/m000.spc'
data = tttrlib.TTTR(spc132_filename, 'SPC-130')
ch0_indeces = data.get_selection_by_channel([0, 8])
data_ch0 = data[ch0_indeces]

n_bins = 512
# selection from tttr object
cr_selection = data_ch0.get_selection_by_count_rate(
    time_window=6.0, n_ph_max=7
)
low_count_selection = data_ch0[cr_selection]
# create histogram for IRF
irf, _ = np.histogram(low_count_selection.micro_times, bins=n_bins)

# Select high count regions
# equivalent selection using selection function
cr_selection = tttrlib.selection_by_count_rate(
    data_ch0.macro_times,
    0.100, n_ph_max=5,
    macro_time_calibration=data.header.macro_time_resolution / 1e6,
    invert=True
)
high_count_selection = data_ch0[cr_selection]
data_decay, _ = np.histogram(high_count_selection.micro_times, bins=n_bins)
time_axis = np.arange(0, n_bins) * data.header.micro_time_resolution * 4096 / n_bins
```

Make decay object

```
decay_object = fit2x.Decay(
    data=data_decay.astype(np.float64),
    irf_histogram=irf.astype(np.float64),
    time_axis=time_axis,
    excitation_period=data.header.macro_time_resolution,
    lifetime_spectrum=[1., 1.2, 1., 3.5]
)
decay_object.evaluate()

# A minimum number of photons should be in each channel
# as no MLE is used and Gaussian distributed errors are assumed
sel = np.where(data_decay > 1)[0]
```

(continues on next page)

(continued from previous page)

```
x_min = 10 #int(min(sel))

# The BH card are not linear at the end of the TAC. Thus
# fit not the full range
x_max = 450 #max(sel)

# Set some initial values for the fit
scatter = [0.05]
background = [2.01]
time_shift = [2]
irf_background = [5]
lifetime_spectrum = [0.5, 0.5, 0.5, 3.5]
```

Optimize

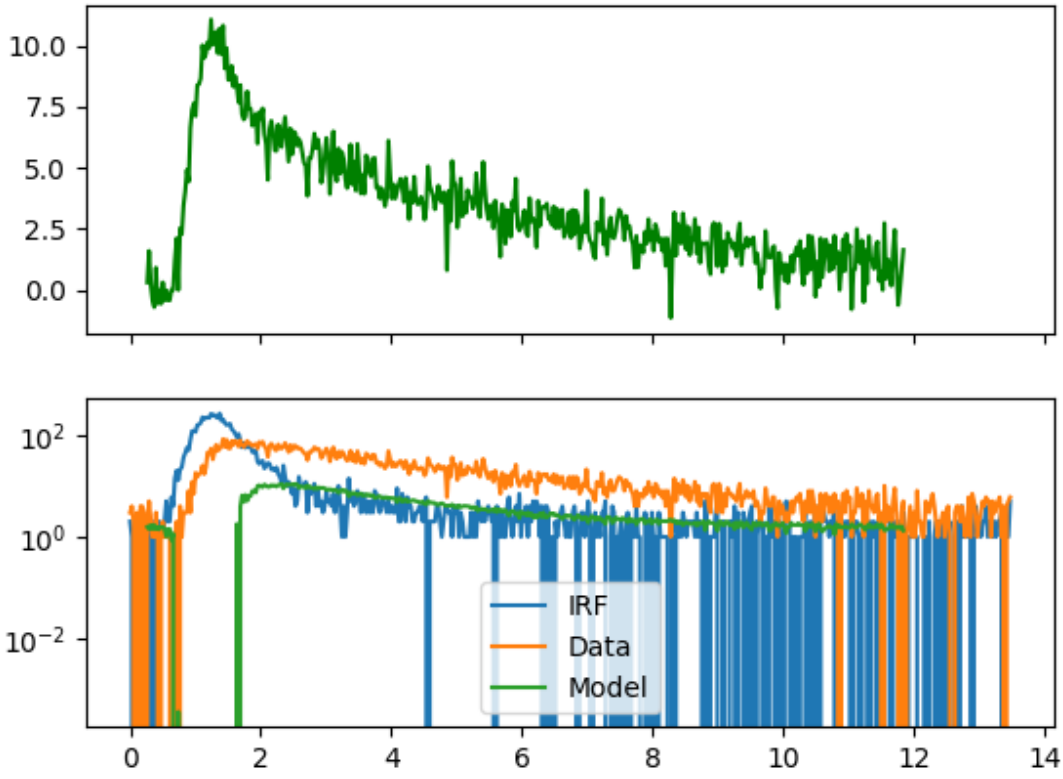
```
x0 = np.hstack(
    [
        scatter,
        background,
        time_shift,
        irf_background,
        lifetime_spectrum
    ]
)
fit = scipy.optimize.minimize(
    objective_function_mle, x0,
    args=(x0, decay_object, x_min, x_max, False, False)
)
fit_mle = fit.x

x0 = np.hstack([scatter, background, time_shift, irf_background, lifetime_spectrum])
fit = scipy.optimize.minimize(
    objective_function_mle, x0,
    args=(x0, decay_object, x_min, x_max)
)
fit_map = fit.x
```

Out:

```
/home/docs/checkouts/readthedocs.org/user_builds/fit2x/conda/latest/lib/python3.7/
↪site-packages/scipy/optimize/_numdiff.py:497: RuntimeWarning: invalid value␣
↪encountered in subtract
  df = fun(x) - f0
/home/docs/checkouts/readthedocs.org/user_builds/fit2x/conda/latest/lib/python3.7/
↪site-packages/scipy/optimize/_numdiff.py:497: RuntimeWarning: invalid value␣
↪encountered in subtract
  df = fun(x) - f0
```

Plot

```
fig, ax = plt.subplots(nrows=2, ncols=1, sharex=True, sharey=False)
ax[1].semilogy(time_axis, irf, label="IRF")
ax[1].semilogy(time_axis, data_decay, label="Data")
ax[1].semilogy(
    time_axis[x_min:x_max],
    decay_object.model[x_min:x_max], label="Model"
)
ax[1].legend()
ax[0].plot(
    time_axis[x_min:x_max],
    decay_object.weighted_residuals[x_min:x_max],
    label='w.res.',
    color='green'
)
plt.show()
```



Total running time of the script: (0 minutes 0.531 seconds)

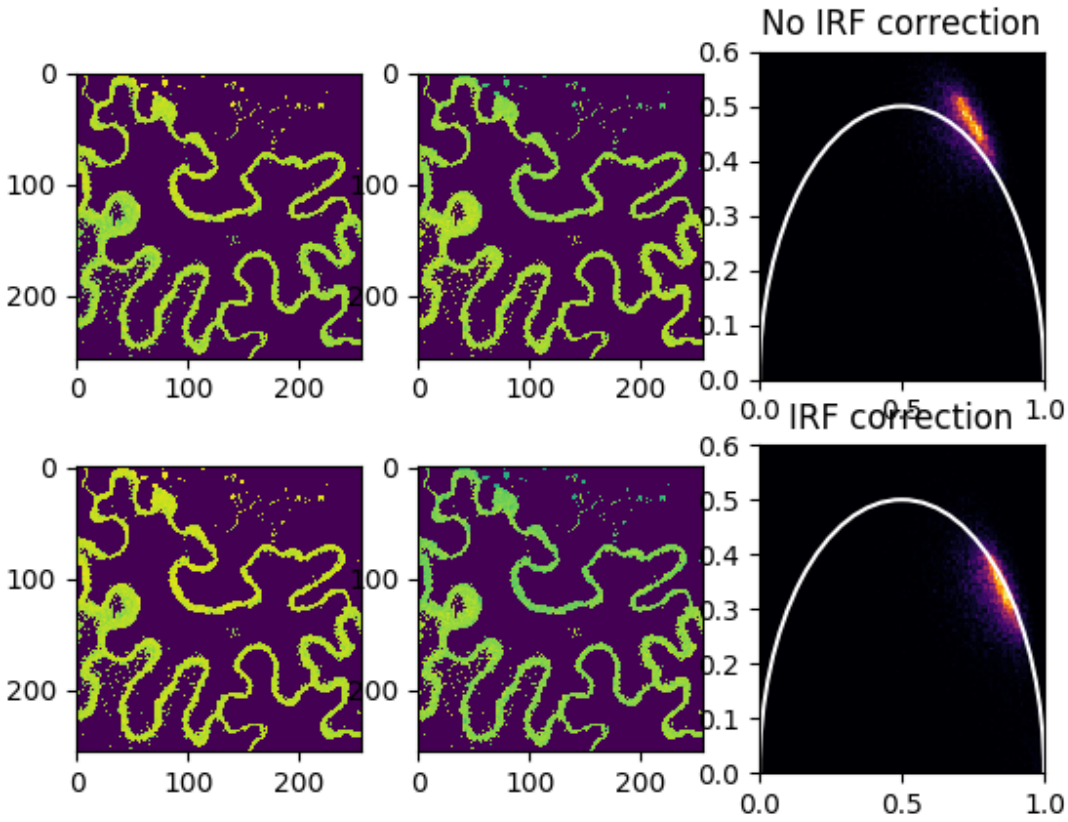
3.4.2 Imaging

Examples

FLIM imaging

Phasor analysis of images

For phasor image analysis the library fit2x provides functions




```
from __future__ import print_function
import tttrlib
import fit2x
import pylab as p

data = tttrlib.TTTR('../../test/data/imaging/pq/ht3/crn_clv_img.ht3')
data_mirror = tttrlib.TTTR('../../test/data/imaging/pq/ht3/crn_clv_mirror.ht3')
data_irf = data_mirror[data_mirror.get_selection_by_channel([0, 1])]

ht3_reading_parameter = {
    "marker_frame_start": [4],
    "marker_line_start": 1,
    "marker_line_stop": 2,
```

(continues on next page)

(continued from previous page)

```
    "marker_event_type": 1,
    "n_pixel_per_line": 256,
    "reading_routine": 'default',
    "channels": [0, 1],
    "fill": True,
    "tttr_data": data,
    "skip_before_first_frame_marker": True
}
image = tttrlib.CLSMImage(**ht3_reading_parameter)
frequency = 32.0  # MHz
stack_frames = True
frequency_mt = frequency / (1000. / data.header.micro_time_resolution)

# No IRF correction
phasor = fit2x.phasor.get_phasor_image(
    image=image,
    stack_frames=stack_frames,
    frequency=frequency_mt
)

n_frames = 1 if stack_frames else image.n_frames
phasor_1d = phasor.reshape((n_frames * image.n_lines * image.n_pixel, 2))
phasor_x, phasor_y = phasor[:, :, :, 0], phasor[:, :, :, 1]
phasor_x_1d, phasor_y_1d = phasor_1d.T[0], phasor_1d.T[1]

fig, ax = p.subplots(nrows=2, ncols=3)
ax[0,2].set(xlim=(0, 1), ylim=(0, 0.6))
a_circle = p.Circle(
    xy=(0.5, 0),
    radius=0.5,
    linewidth=1.5,
    fill=False,
    color='w'
)
ax[0, 2].add_artist(a_circle)
ax[0, 2].hist2d(
    x=phasor_x_1d,
    y=phasor_y_1d,
    bins=101,
    range=((0, 1), (0, 0.6)),
    cmap='inferno'
)
ax[0,0].imshow(phasor_x[0,:,:])
ax[0,1].imshow(phasor_y[0,:,:])

# IRF correction
data_irf = data_mirror[data_mirror.get_selection_by_channel([0, 1])]
phasor = fit2x.phasor.get_phasor_image(
    image=image,
    tttr_irf=data_irf,
    stack_frames=stack_frames,
    frequency=frequency_mt,
    minimum_number_of_photons=30
)
n_frames = 1 if stack_frames else image.n_frames
phasor_1d = phasor.reshape((n_frames * image.n_lines * image.n_pixel, 2))
phasor_x, phasor_y = phasor[:, :, :, 0], phasor[:, :, :, 1]
```

(continues on next page)

(continued from previous page)

```
phasor_x_1d, phasor_y_1d = phasor_1d.T[0], phasor_1d.T[1]

ax[0, 2].set_title('No IRF correction')
ax[1, 2].set_title('IRF correction')
ax[1, 2].set(xlim=(0, 1), ylim=(0, 0.6))
a_circle = p.Circle(
    xy=(0.5, 0),
    radius=0.5,
    linewidth=1.5,
    fill=False,
    color='w'
)
ax[1,2].add_artist(a_circle)
ax[1,2].hist2d(
    x=phasor_x_1d,
    y=phasor_y_1d,
    bins=101,
    range=((0, 1), (0, 0.6)),
    cmap='inferno'
)
ax[1,0].imshow(phasor_x[0,:,:])
ax[1,1].imshow(phasor_y[0,:,:])

p.show()
```

Total running time of the script: (0 minutes 6.636 seconds)

3.5 C++ API

3.6 Glossary

CLSM confocal laser scanning microscopy

MFD (Multiparameter Fluorescence Detection) A MFD experiments is a time-resolved fluorescence experiment which probes the absorption and fluorescence, the fluorescence quantum yield, the fluorescence lifetime, and the anisotropy of the studied chromophores simultaneously (see [KuhnemuthS01])

IRF IRF stands for instrument response function. In time-resolved fluorescence measurements the IRF is the temporal response of the fluorescence spectrometer to a delta-pulse. Suppose a initially sharp pulse defines the time of excitation / triggers the laser, then recorded response of the fluorescence spectrometer is broadened due to: (1) the temporal response of the exciting light source, (2) the temporal dispersion due to the optics of the instrument, (3) the delay of the light within the sample, and (4) the response of the detector. As the most intuitive contribution to the IRF is the excitation profile, the IRF is sometimes called 'lamp function'. The IRF is typically recorded by minimising the contribution of (3), e.g., by measuring the response of the instrument using a scattering sample, or a short lived dye.

Time-tagged time resolved (TTTR) TTTR stands for time tagged time-resolved data or experiments. In TTTR-datasets the events, e.g., the detection of a photon, are tagged by a detection channel number. Moreover, the recording clock usually registers the events with a high time resolution of a few picoseconds. For long recording times of the detected events, a coarse and a fine clock are combined. The fine clock measures the time of the events relative to the coarse clock with a high time resolution. The time of the coarse and the fine clock is usually called macro and micro time, respectively.

Time correlated single photon counting (TCSPC) Time correlated single photon counting (TCPSC) is a technique to measure light intensities with picosecond resolution. Its main application is the detection of fluorescent light. A pulsed light source excites a fluorescent sample. A single photon detector records the emitted fluorescence photons. Thus, per excitation cycle, only a single photon is detected. Fast detection electronics records the time between the excitation pulse and the detection of the fluorescence photon. A histogram accumulates multiple detected photons to yield a time-resolved fluorescence intensity decay.

SWIG SWIG is a software development tool that connects programs written in C and C++ with a variety of high-level programming languages. SWIG can be used with different types of target languages including common scripting languages such as Javascript, Perl, PHP, Python, Tcl and Ruby and non-scripting languages such as C#, D, Go language, Java, Octave, and R. SWIG is free software and the code that SWIG generates is compatible with both commercial and non-commercial projects. `fit2x` is C/C++ based to provide the capability for a broad variety of languages to interface its provided functionality.

Scatter fraction The scatter fraction $gamma$ is defined by the number of photons that

Anisotropy The steady-state anisotropy r_G in the detection channel G is formally given by the fluorescence intensity weighted integral of the time-resolved anisotropy.

$r_G = \int F_G(t) \cdot r(t) dt \cdot \frac{1}{\int F_G(t) dt}$

where the time-resolved anisotropy is defined by unperturbed the fluorescence intensities of an ideal detection system.

$r_G(t) = \frac{F_{G,p}(t) - F_{G,s}(t)}{F_{G,p}(t) + 2F_{G,s}(t)}$

Through out `fit2x` two distinct anisotropies are computed: (1) background corrected anisotropies, and (2) anisotropies not accounting for the background. In single-molecule experiments the background is mainly scattered light (Raman scattering). The uncorrected anisotropy (without background correction) is computed by:

$r = (S_p - g \cdot S_s) / (S_p \cdot (1 - 3 \cdot l_2) + (2 - 3 \cdot l_1) \cdot g \cdot Ss)$

where S_p is the signal in the parallel (German: parallel=p) detection channel, :math`S_s` the signal in the perpendicular decection channel (German: senkrecht=s), g is the g-factor, l_1 and l_2 are factor mixing that determine the mixing of the parallel and perpendicular detection channel, respectively [KSM95].

The scatter corrected steady-state anisotropy is computed using the scatter / background corrected signals parallel $F_p = (S_p - \gamma \cdot B_p)/(1. - \gamma)$ and perpendicular $F_s = (S_s - \gamma \cdot B_s)/(1. - \gamma)$ fluorescence intensity. $r = (F_p - g \cdot F_s)/(F_p \cdot (1 - 3 \cdot l_2) + (2 - 3 \cdot l_1) \cdot g \cdot F_s)$ The scatter corrected and anisotropy not corrected for scatter are computed by most fits of `fit2x`.

Jordi-format In the Jordi format is a format for fluorescence decays. In the Jordi format fluorescence decays are stacked in a one dimensional array. In a typical polarization resolved Jordi file the first decay is the parallel and the subsequent decay is the perpendicular decay. In the Jordi format both decays must have the same length, i.e., the same number of micro time counting channels.

3.7 References

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- search

CHAPTER

FIVE

LICENSE

fit2x was developed at the Seidel Lab (Heinrich Heine University) and is maintained by Thomas Peulen. fit2x is released under the open source MIT license.

BIBLIOGRAPHY

[AFGS06] Matthew Antonik, Suren Felekyan, Alexander Gaiduk, and Claus A. M. Seidel. Separating structural heterogeneities from stochastic variations in fluorescence resonance energy transfer distributions via photon distribution analysis. *The Journal of Physical Chemistry B*, 110(13):6970–6978, 2006.

[Bec05] Wolfgang Becker. *Advanced Time-Correlated Single Photon Counting Techniques*. Springer Science & Business Media, December 2005. ISBN 978-3-540-28882-4. Google-Books-ID: fLP0PB9CZ94C.

[BWR+02] Martin Bohmer, Michaek Wahl, Hans-Jurgen Rahn, Rainer Erdmann, and Jorg Enderlein. Time-resolved fluorescence correlation spectroscopy. *Chemical Physics Letters*, 353:439–445, 2002.

[Coa68] P B Coates. The correction for photon `pile-up' in the measurement of radiative lifetimes. *Journal of Physics E: Scientific Instruments*, 1(8):878–879, August 1968. URL: http://stacks.iop.org/0022-3735/1/i=8/a=437?key=crossref.19c54cc94064528a5003e3e0a645afc3, doi:10.1088/0022-3735/1/8/437.

[DBS+05] Michelle A. Digman, Claire M. Brown, Parijat Sengupta, Paul W. Wiseman, Alan R. Horwitz, and Enrico Gratton. Measuring Fast Dynamics in Solutions and Cells with a Laser Scanning Microscope. *Biophysical Journal*, 89(2):1317–1327, August 2005. URL: http://www.sciencedirect.com/science/article/pii/S000634950572779X, doi:10.1529/biophysj.105.062836.

[DSW+05] Michelle A. Digman, Parijat Sengupta, Paul W. Wiseman, Claire M. Brown, Alan R. Horwitz, and Enrico Gratton. Fluctuation Correlation Spectroscopy with a Laser-Scanning Microscope: Exploiting the Hidden Time Structure. *Biophysical Journal*, 88(5):L33–L36, May 2005. URL: http://www.sciencedirect.com/science/article/pii/S0006349505733580, doi:10.1529/biophysj.105.061788.

[DWHG09] Michelle A. Digman, Paul W. Wiseman, Alan R. Horwitz, and Enrico Gratton. Detecting Protein Complexes in Living Cells from Laser Scanning Confocal Image Sequences by the Cross Correlation Raster Image Spectroscopy Method. *Biophysical Journal*, 96(2):707–716, January 2009. URL: http://www.sciencedirect.com/science/article/pii/S0006349508000799, doi:10.1016/j.bpj.2008.09.051.

[EM74] Elliot L Elson and Douglas Magde. Fluorescence correlation spectroscopy. i. conceptual basis and theory. *Biopolymers: Original Research on Biomolecules*, 13(1):1–27, 1974.

[EE97] Jorg Enderlein and Rainer Erdmann. Fast fitting of multi-exponential decay curves. *Optics Communications*, 134:371–378, 1997.

[End12] Jöerg Enderlein. Polymer dynamics, fluorescence correlation spectroscopy, and the limits of optical resolution. *Phys Rev Lett.*, 2012. arXiv:1203.3204v1, doi:10.1103/PhysRevLett.108.108101.

[HDSL16] Jelle Hendrix, Tomas Dekens, Waldemar Schrimpf, and Don C. Lamb. Arbitrary-Region Raster Image Correlation Spectroscopy. *Biophysical Journal*, 111(8):1785–1796, October 2016. URL: http://www.sciencedirect.com/science/article/pii/S0006349516308104, doi:10.1016/j.bpj.2016.09.012.

[IDH73] Irvin Isenberg, Robert D. Dyson, and Richard Hanson. Studies on the Analysis of Fluorescence Decay Data by the Method of Moments. *Biophysical Journal*, 13(10):1090–1115, October 1973. URL: https://linkinghub.elsevier.com/retrieve/pii/S0006349573860473, doi:10.1016/S0006-3495(73)86047-3.

[KFAS07] Stanislav Kalinin, Suren Felekyan, Matthew Antonik, and Claus A. M. Seidel. Probability distribution analysis of single-molecule fluorescence anisotropy and resonance energy transfer. *The Journal of Physical Chemistry B*, 111(34):10253–10262, 2007.

[KWB+07] Peter Kapusta, Michael Wahl, Ales Benda, Martin Hof, and Jorg Enderlein. Fluorescence lifetime correlation spectroscopy. *Journal of Fluorescence*, 17:43–48, 2007.

[KSM95] Masanori Koshioka, Keiji Sasaki, and Hiroshi Masuhara. Time-Dependent Fluorescence Depolarization Analysis in Three-Dimensional Microspectroscopy. *Applied Spectroscopy*, 49(2):224–228, February 1995. Publisher: SAGE Publications Ltd STM. URL: https://doi.org/10.1366/0003702953963652, doi:10.1366/0003702953963652.

[KuhnemuthS01] Ralf Kühnemuth and Claus A. M. Seidel. Principles of single molecule multiparameter fluorescence spectroscopy. *Single Molecules*, 2:251–254, 2001.

[LFH06] Ted A. Laurence, Samantha Fore, and Thomas Huser. Fast, flexible algorithm for calculating photon correlations. *Optics express*, 31:829, 2006.

[LKK+04] Ted A. Laurence, Achillefs N. Kapanidis, Xiangxu Kong, Daniel S. Chemla, and Shimon Weiss. Photon arrival-time interval distribution (PAID): a novel tool for analyzing molecular interactions. *The Journal of Physical Chemistry B*, 108(9):3051–3067, 2004.

[MEW72] Douglas Magde, Elliot Elson, and Watt W Webb. Thermodynamic fluctuations in a reacting system—measurement by fluorescence correlation spectroscopy. *Physical Review Letters*, 29(11):705, 1972.

[MCH+01] Michael Maus, Mircea Cotlet, Johan Hofkens, Thomas Gensch, Frans C. De Schryver, J. Schaffer, and C. A. M. Seidel. An Experimental Comparison of the Maximum Likelihood Estimation and Nonlinear Least-Squares Fluorescence Lifetime Analysis of Single Molecules. *Analytical Chemistry*, 73(9):2078–2086, May 2001. Publisher: American Chemical Society. URL: https://doi.org/10.1021/ac000877g, doi:10.1021/ac000877g.

[OConnor12] Desmond O'Connor. *Time-correlated single photon counting*. Academic Press, December 2012. ISBN 978-0-323-14144-4. Google-Books-ID: ELQ0Mz6Rq1EC.

[PHW+93] N. O. Petersen, P. L. Höddelius, P. W. Wiseman, O. Seger, and K. E. Magnusson. Quantitation of membrane receptor distributions by image correlation spectroscopy: concept and application. *Biophysical Journal*, 65(3):1135–1146, September 1993. URL: http://www.sciencedirect.com/science/article/pii/S0006349593811731, doi:10.1016/S0006-3495(93)81173-1.

[POS17] Thomas-Otavio Peulen, Oleg Opanasyuk, and Claus AM Seidel. Combining graphical and analytical methods with molecular simulations to analyze time-resolved fret measurements of labeled macromolecules accurately. *The Journal of Physical Chemistry B*, 121(35):8211–8241, 2017.

[Qia90] Hong Qian. On the statistics of fluorescence correlation spectroscopy. *Biophysical Chemistry*, 38(1):49–57, 1990.

[RBC+10] J. Ries, M. Bayer, Csucs, G, Dirkx, R, M. Solimena, H. Ewers, and P. Schwille. Automated suppression of sample-related artifacts in fluorescence correlation spectroscopy. *Optics express*, 18(11):11073–11082, 2010.

[SVE+99] J. Schaffer, A. Volkmer, C. Eggeling, V. Subramaniam, G. Striker, and C. A. M. Seidel. Identification of Single Molecules in Aqueous Solution by Time-Resolved Fluorescence Anisotropy. *The Journal of Physical Chemistry A*, 103(3):331–336, January 1999. Publisher: American Chemical Society. URL: https://doi.org/10.1021/jp9833597, doi:10.1021/jp9833597.

[SRB01] Konstantin Starchev, Jaro Ricka, and Jacques Buffle. Noise on fluorescence correlation spectroscopy. *Journal of Colloid and Interface Science*, 2001.

[WGPE03] Michael Wahl, Ingo Gregor, Matthias Patting, and Jörg Enderlein. Fast calculation of fluorescence correlation data with asynchronous time-correlated single-photon counting. *Optics express*, 11(26):3583–3591, 2003.

[WRV01] Thorsten Wohland, Rudolf Rigler, and Horst Vogel. The standard deviation in fluorescence correlation spectroscopy. *Biophysical Journal*, 80(6):2987–2999, 2001.

INDEX