Manual for the FCS/FCCS simulator software (`diffusion4`)

Dr. Jan W. Krieger <jan@jkrieger.de>

October 28, 2015

Contents

1 Availability and Compilation **3**
- 1.1 Availability . . . 3
- 1.2 Download and Setup . . . 3
- 1.3 Compilation . . . 4
- 1.4 Running the Software DIFFUSION4 . . . 4
- 1.5 Running DIFFUSION4 from a Makefile . . . 4

2 Introduction to DIFFUSION4 **6**
- 2.1 Summary of DIFFUSION4 . . . 6

3 Reference of the DIFFUSION4-Modules **9**
- 3.1 Fluorophore Dynamics Modules . . . 9
  - 3.1.1 Fluorescence Detection Modules . . . 9
- 3.2 Introduction and Registration . . . 9
- 3.3 Implementing `FluorophorDynamics`-Classes . . . 10
- 3.4 Implementing `FluorescenceMeasurement`-Classes . . . 10

Chapter 1

Availability and Compilation

1.1 Availability

This software is available on GitHub:

`https://github.com/jkriege2/FCSSimulator`

In addition this simulator is also a part (based on the same repository) of the FCS/FCCS data evaluation software QUICKFIT 3.0 (`https://github.com/jkriege2/QuickFit3` and `http://www.dkfz.de/Macromol/quickfit/`), which also offers an integrated editor-GUI for the simulator configuration scripts.

1.2 Download and Setup

DIFFUSION4 is a C++ program, so you will need a working C++ compiler (e.g. the GCC in version ≥4.7, for Windows you can use MINGW-BUILDS from `http://sourceforge.net/projects/mingwbuilds/`). The program only relies on the GNU scientific library (GSL), available from `http://www.gnu.org/software/gsl/`, but also integrated in the repository (see below). In addition you will need a working BASH-shell (on Windows use MSYS available from `http://www.mingw.org/wiki/msys`) and GIT to access the repository (if you don't want to download the code manually). In general this program should compile on WINDOWS, LINUX and MACOS X.

In order to install DIFFUSION4, follow these steps:

1. check out the git repository:

```
$ git clone --recursive "https://github.com/jkriege2/FCSSimulator.git"
```

2. ensure that all submodules are checked out:

```
$ cd FCSSimulator
$ git submodule update --init --remote --recursive --force
```

3. if GSL is not available on your system, we'll have to build it now. The repository contains a BASH-script for this purpose:

```
$ cd extlibs
$ . build_dependencies.sh
$ cd ..
```

This script will build a local version of GSL. It is not installed in the system, but only in the directory ./extlibs/gsl/. It will ask several questions: Generally you should not keep the build-directories (n), use as many processor, as you like (e.g. 2 for a dual-core, or 4 for a quad-core machine), you the best optimizations for your local machine, if the programm will only be used locally (twice y, or for safe-settings y and then n). Finally answer y when asked whether GSL should be compiled.

1.3 Compilation

the rest of the compilation is done using a Makefile in the base directory:

```
$ make
```

1.4 Running the Software DIFFUSION4

After the compilation, an command-line based executable diffusion4 is created in the base directory. This is the simulator, which can then be started as follows:

```
$ diffusion4 [--spectra SPECTRADIRECTORY] CONFIG_SCRIPT.ini
```

Here CONFIG_SCRIPT.ini is a configuration file that tells the simulator what to do and the optional -spectra SPECTRADIRECTORY can be used to provide another directory with absorption and emission spectra. A version of this directory is available in the repository under ./spectra/, which is also loaded automatically, if the option --spectra is not given. DIFFUSION4 reads absorption and emission spectra for its simulation from this directory. Note that DIFFUSION4 will not work, if a spectra-directory is not provided!

1.5 Running DIFFUSION4 from a Makefile

Often it is useful to run DIFFUSION4 from a Makefile, if several simulations should be done (in parallel) on a computer. here is an example Makefile for this task (see the repository directory ./example_configs/ for further examples:

```
SCRIPTS= gyuri_isat_multigfp0.ini\
            gyuri_isat_multigfp1.ini

SHELL = sh

SCRIPTS_TARGET = $(subst .ini,.target,$(SCRIPTS))

TERMINAL_COMMAND=

ifeq ($(findstring Msys,$(OS)),Msys)
EXE_SUFFIX=.exe
TERMINAL_COMMAND=
else
EXE_SUFFIX=
#TERMINAL_COMMAND=konsole -e
#TERMINAL_COMMAND=x-terminal-emulator -e
endif

all: ${SCRIPTS_TARGET}

%.target: %.ini
        @echo -e "random␣wait␣before␣starting␣$<␣..."
        @bash random_sleep.sh
        @bash -c "sleep␣$$[␣(␣$$RANDOM␣%␣10␣)␣␣+␣1␣]s"
        @echo -e "starting␣on␣$<␣..."
        ${TERMINAL_COMMAND} ./diffusion4${EXE_SUFFIX} $< > $<.log
        @echo -e "work␣on␣$<␣DONE!"
```

You can run such a Makefile with the command:

```
$ make -f Makefile -j4
```

where `4` specifies the number of processors to use. This makefile also uses the BASH-script `random_sleep.sh`, which waits a random number of seconds (up to 10 or 20) before running the simulation. This ensures that the random number generator of each simulation is initialized with a different seed (the seed is taken from the system time!).

Chapter 2

Introduction to DIFFUSION4

This software has been developed to simulate fluorescence correlation spectroscopy (FCS) and fluorescence cross-correlation spectroscopy (FCCS) measurements, based on particle trajectories.

2.1 Summary of DIFFUSION4

Note: This section is a copy from the appendix of my PhD thesis!

This software can simulate FCS and FCCS correlation curves for different focus geometries and is closely related to the FCS/FCCS theory as presented in chapter **??** and especially the modeling of fluorescence detection described in section **??**. It starts from a set of N particle trajectories $\{\vec{r}_i(t)\}$, where i numbers the particles and $t = 1, 2, ...$ numbers the equidistant timepoints with resolution Δt_{im}. The trajectories are either read from an external data file or are created internally by a configurable random walk. First $N_i \geq 1$ fluorophores are assigned to each particle, leading to an overall number of fluorophores

$$F = \sum_{i=1}^{N} N_i$$

fluorophores f, which are each characterized by the following set of properties (the functions $i(f)$ is the trajectory ID for every fluorophore f):

1. a position $\vec{r}_f(t) = \vec{r}_{i(f)}(t) + \Delta\vec{r}_f$, where $\vec{r}_{i(f)}(t)$ is the position of the particle and $\Delta\vec{r}_f$ is an arbitrary, but constant shift from this position. In the simplest case there is only one fluorophore per particle and $\Delta\vec{r}_f = 0$. Using $\Delta\vec{r}_f \neq 0$, moving finite-sized objects may be simulated that are e.g. labeled with a set of fluorophores on their surface or in their interior.

2. each fluorophore may be in one of S_f states. Each state may have different spectroscopic properties. The current state at time t is denoted by $s_f(t)$.

3. a wavelength-dependent absorption crosssection spectrum $\sigma_{\text{abs},i}(\lambda)$.

4. a normalized fluorescence spectrum $\eta_{\text{fl},f}(\lambda)$ and a fluorescence quantum yield $q_{\text{fluor},f,s_f(t)}$

5. a dipole orientation vector $\vec{p}_f(t)$ with $|\vec{p}_f(t)| = 1$.

The state trajectory $s_f(t)$ for each fluorophore either does not change (the default case), is read from an external file, or is simulated using a matrix of transition rates and a random decision in each simulation step. In this way photophysical blinking transitions may be simulated, if e.g. $s_f(t) \equiv 1$ is a bright state and $s_f(t) \equiv 2$ is a dark state with $q_{\text{fluor},f,2} = 0$. Also a simple bleaching process is implemented, by switching off (but never on again) a fluorophore with a certain low probability.

The simulation proceeds in steps of Δt_{sim}. For each time step and each focus in the simulation, first the expected number of fluorescence photons is calculated:

$$\overline{N}_{\text{phot}}(t)=\sum_{f=1}^{F} q_{\text{fluor},f,s_f(t)}\cdot\sigma_{\text{abs},i}(\lambda_{\text{ex}})\cdot q_{\text{det}}\cdot\frac{\Delta t_{\text{sim}}\cdot I(\vec{r}_f(t))}{hc_0/\lambda_{\text{ex}}}\cdot\Omega(\vec{r}_f(t))\cdot h_{\text{pol}}(\vec{p}_f(t)), \tag{2.1}$$

where h is Planck's constant, c_0 is the velocity of light in vacuum and λ_{ex} is the excitation wavelength.

The shape of the illumination profile is described by the function $I(\vec{r})$ and the respective shape of photon collection efficiency by $\Omega(\vec{r})$. Several models are implemented for them:

1. **Gaussian**: The shapes of $I(\vec{r})$ and $\Omega(\vec{r})$ are cigar-like Gaussian functions with equal x- and y-width w_0 and z-width z_0:
$$I(\vec{r}),\Omega(\vec{r})\propto\exp\left(-2\cdot\frac{x^2+y^2}{w_0^2}-2\cdot\frac{z^2}{z_0^2}\right) \tag{2.2}$$
2. **Gaussian beam**: The illumination/detection focus is described by a Gaussian beam, which has a lateral width $w(z)$ increasing with distance z from the focus and a Laurentzian shape in z-direction:
$$I(\vec{r}),\Omega(\vec{r})\propto\left(\frac{w_0}{w(z)}\right)^2\cdot\exp\left(-2\cdot\frac{x^2+y^2}{w^2(z)}\right),\quad\text{with}\quad w(z)=w_0\cdot\sqrt{1+\left(\frac{z}{z_0}\right)^2} \tag{2.3}$$
3. **Gaussian light sheet**: A simple model for a lightsheet is a Gaussian in z-direction, which does not depend on x or y:
$$I(\vec{r})\propto\exp\left(-2\cdot\frac{z^2}{z_0^2}\right) \tag{2.4}$$
4. **Slit pattern light sheet**: To model the sidelobes observed in typical light sheets a slit function can be used:
$$I(\vec{r})\propto\left(\frac{\sin(\pi\cdot z/z_0)}{\pi\cdot z/z_0}\right)^2 \tag{2.5}$$

The first two patterns can be used for both, the illumination and detection foci, whereas the last two are designed to model the light sheet illumination.

The remaining influence of the detection process (signal loss at optical interfaces and filters, detector quantum efficiency, ...) is described by the factor

$$q_{\text{det}}=q_{\text{det},0}\cdot\frac{\int\limits_{\lambda_{\text{det,min}}}^{\lambda_{\text{det,max}}}\eta_{\text{fl},f}(\lambda)\,\mathrm{d}\lambda}{\int\limits_0^\infty\eta_{\text{fl},f}(\lambda)\,\mathrm{d}\lambda}, \tag{2.6}$$

summarizing the loss of light due to optics and detector quantum efficiency $q_{\text{det},0}$, as well as the spectral width of the fluorescence detection window $\lambda_{\text{det,min}}...\lambda_{\text{det,max}}$. This detection window allows to also take into account crosstalk between two detection channels. The influence of the dipole direction $\vec{p}_f(t)$ and a possible laser polarization is modeled by the factor

$$h_{\text{pol}}(\vec{p}_f(t))=(1-\theta_{\text{pol}})+\theta_{\text{Pol}}\cdot\left(\vec{\epsilon}_{\text{ex}}\bullet\vec{p}_f(t)\right)^2, \tag{2.7}$$

where $\bullet$ is a scalar product, $\theta_{\text{Pol}}\in[0,1]$ is the fraction of linear polarization of the excitation light source and $\vec{\epsilon}_{\text{ex}}$ (with $|\vec{\epsilon}_{\text{ex}}|=1$) is the linear polarization direction of this light source.

From the average number of detected photons, the measured number of photons $N_{\text{det}}(t)$ is calculated, taking the detector statistics into account. In the simplest case of a photon counting detector, $N_{\text{det}}(t)$ is drawn from a Poissonian distribution with mean (and variance) $\overline{N}_{\text{phot}}(t)$. Other detection statistics are possible, such as a linear detector, where $N_{\text{det}}(t)$ is drawn from a Gaussian distribution with mean $\langle G\rangle\cdot\overline{N}_{\text{phot}}(t)$ and a variance comparable to (**??**):

$$\sigma_{\text{det}}^2=\langle G\rangle^2\cdot\mathcal{F}^2\cdot N_{\text{det}}(t)+\sigma_{\text{read}}^2, \tag{2.8}$$

where $\langle G\rangle$ is the average detector gain, $\mathcal{F}^2$ the excess noise factor and σ_{read}^2 the read noise variance, summarizing all contributions, not depending on the number of incident photons. Also artifacts, such as a background intensity offset may be included in the detector simulation. Although intermediate results may be floating-point numbers, the finally detected number of photons (or ADU counts in a linear detector) is always an integer number.

Finally the time series $N_{\text{det}}(t)$ is post processed to yield count rate traces with arbitrary binning, auto- and cross-correlation functions (between different foci on the simulation) and other statistical properties. Also several test data sets are saved by the simulation program, such as particle mean squared displacements (MSDs), the raw detector statistics etc.

The complete program is split into modules that may be combined in different ways for a simulation. All these modules are either trajectory sources or sinks. In each time step first all sources generate a new set of fluorophore properties, e.g. by reading a new data set from a file or advancing a random walk simulation. The these new particle properties are forwarded to the sink objects, which simulate the actual detection process, as described above, or generate MSDs and other trajectory statistics. Every sink may be connected to several sources, and one source can feed several sinks. This can be used e.g. for simulations of fluorophore reservoir depletion, as in section **??**, where a single trajectory source is fed into intermediate objects that simulate different bleaching rates on the same particle positions and finally detected by a set of identical sinks, which simulate FCS detection.

This software was initiated in the first year of the thesis and extended and improved in the following years. It was used to simulate different aspects of FCS/FCCS in several publications [1–4].

Chapter 3

Reference of the DIFFUSION4-Modules

3.1 Fluorophore Dynamics Modules

3.1.1 Fluorescence Detection Modules

terExtending DIFFUSION4

3.2 Introduction and Registration

You can extend DIFFUSION4 with your own modules by implementing classes of the type `FluorophorDynamics` for fluorophore dynamics modules, or `FluorescenceMeasurement` for fluorescence detection modules. these virtual base-classes each provide virtual functions that you have to implement in order to give you class a function. Finally you will have to register your new class in `main.cpp`:

- `FluorophorDynamics`-classes have to be added near the text

  ```
  /////////////////////////////////////////////////////
  // add you custom dynamics classes here
  /////////////////////////////////////////////////////
  ```

  e.g.:

  ```
  /////////////////////////////////////////////////////
  // add you custom dynamics classes here
  /////////////////////////////////////////////////////
  } else if (lgname.find("mydyn")==0 && lgname.size()>5) {
          supergroup="mydyn";
          d=new MyDynamics(fluorophors, oname);
  ```

  Here you custom class is called `MyDynamics` and in the configration files it will have the prefix `mydyn`.

- `FluorescenceMeasurement`-classes have to be added near the text

  ```
  /////////////////////////////////////////////////////
  // add you custom detection classes here
  /////////////////////////////////////////////////////
  ```

  e.g.:

  ```
  /////////////////////////////////////////////////////
  // add you custom detection classes here
  /////////////////////////////////////////////////////
  ```

```
} else if (lgname.find("mydetection")==0 && lgname.size()>14) {
        supergroup="mydetection";
        m=new MyDetection(fluorophors, oname);
```

Here you custom class is called `MyDetection` and in the configuration files it will have the prefix `mydetection`.

3.3 Implementing `FluorophorDynamics`-Classes

```
        virtual void init();
```

This function is called before the simulation and is used to initialize the simulation object.

```
        virtual void propagate(bool boundary_check=true);
```

This function implements the main functionality. It is called in every step and propagates the walkers that are stored in the array `walker_state`. If the parameter `boundary_check` is set `true`, the function body should perform a boundary-check at the borders of the sim-box. Otherwise the sim-box is assumed to be infinite (used for testing).

```
        virtual std::string report();
```

This function reports the object-state in human-readable form.

```
        virtual void read_config_internal(jkINIParser2& parser);
```

This function reads the object-configuration from the given parser, which is already cd'ed to the group to be read. This function is called twice for each object: Once for the super-group and once for the actual object-group.

Note that there are additional functions that can be used in special cases. See the implemented classes in the repository for details.

3.4 Implementing `FluorescenceMeasurement`-Classes

```
        virtual void init();
```

This function is called before the simulation and is used to initialize the simulation object.

```
        virtual void propagate();
```

This function implements the main functionality. It is called in every step and propagates the walkers that can be obtained from the dynamics-objects in the array `dyn`.

```
        virtual std::string report();
```

This function reports the object-state in human-readable form.

```
        virtual void save();
```

This function stores the simulation results.

```
        virtual void read_config_internal(jkINIParser2& parser);
```

This function reads the object-configuration from the given parser, which is already cd'ed to the group to be read. This function is called twice for each object: Once for the super-group and once for the actual object-group.

Note that there are additional functions that can be used in special cases. See the implemented classes in the repository for details.

Index

availability, 3

build, 3
build environment, 3

compilation, 4

Download, 3

FCCS, 6
FCS, 6
fluorescence correlation spectroscopy, 6
fluorescence cross-correlation spectroscopy, 6
fluorescence detection, 9, 10
`FluorescenceMeasurement`, 9, 10
`FluorescenceMeasurement::init()`, 10
`FluorescenceMeasurement::propagate()`, 10
`FluorescenceMeasurement::read_config_internal()`, 10
`FluorescenceMeasurement::report()`, 10
`FluorescenceMeasurement::save()`, 10
`FluorophorDynamics`, 9, 10
`FluorophorDynamics::init()`, 10
`FluorophorDynamics::propagate()`, 10
`FluorophorDynamics::read_config_internal()`, 10
`FluorophorDynamics::report()`, 10
fluorophore dynamics, 9, 10

mean squared displacement, 8
MSD, 8

particle dynamics, 9, 10

repository, 3
running, 4
  Makefile, 4

Bibliography

[1] Tomasz Wocjan, Jan Krieger, Oleg Krichevsky, and Jörg Langowski. Dynamics of a fluorophore attached to superhelical DNA: FCS experiments simulated by brownian dynamics. *Physical Chemistry Chemical Physics*, 11(45):10671, 2009. doi:10.1039/B911857H.

[2] Jan Buchholz, Jan Wolfgang Krieger, Gábor Mocsár, Balázs Kreith, Edoardo Charbon, György Vámosi, Udo Kebschull, and Jörg Langowski. FPGA implementation of a 32x32 autocorrelator array for analysis of fast image series. *Optics Express*, 20(16):17767, 2012. doi:10.1364/OE.20.017767.

[3] Anand Pratap Singh, Jan Wolfgang Krieger, Jan Buchholz, Edoardo Charbon, Jörg Langowski, and Thorsten Wohland. The performance of 2D array detectors for light sheet based fluorescence correlation spectroscopy. *Opt. Express*, 21(7):8652–8668, 2013. doi:10.1364/OE.21.008652.

[4] Jan Wolfgang Krieger, Anand Pratap Singh, Christoph S. Garbe, Thorsten Wohland, and Jörg Langowski. Dual-Color fluorescence Cross-Correlation spectroscopy on a single plane illumination microscope (SPIM-FCCS). *Optics Express*, 22(3):2358, 2014. doi:10.1364/OE.22.002358.