libprofit Documentation

Release 1.9.3

Aaron Robotham, Rodrigo Tobar

Jan 08, 2021

Contents

libprofit is a C++ library that produces images based on different luminosity profiles.

CHAPTER 1

Getting *libprofit*

libprofit is currently hosted in GitHub. To get a copy you can clone the repository:

```
git clone https://github.com/ICRAR/libprofit
```

1.1 Compiling

libprofit depends on:

- GSL
- R

Both dependencies satisfy the same requirements, so they are mutually exclusive, but at least one of them is necessary. If both are present GSL takes precedence.

Optional requirements are:

- An OpenMP-enabled compiler
- An OpenCL installation
- FFTW

libprofit's compilation system is based on cmake. `cmake` will check that you have a proper compiler (anything supporting some basic C++11 should do), and scan the system for all required dependencies.

To compile *libprofit* run (assuming you are inside the `libprofit` directory already):

```
$> mkdir build
$> cd build
$> cmake ..
$> make
$> # optionally for system-wide installation: sudo make install
```

With `cmake` you can also specify additional compilation flags. For example, if you want to generate the fastest possible code you can try this:

```
$> cmake .. -DCMAKE_CXX_FLAGS="-O3 -march=native"
```

You can also specify a different installation directory like this:

```
$> cmake .. -DCMAKE_INSTALL_PREFIX=~/my/installation/directory
```

Other `cmake` options that can be given in the command-line include:

- `LIBPROFIT_USE_R`: prefer R libraries over GSL libraries
- `LIBPROFIT_TEST`: enable compilation of unit tests
- `LIBPROFIT_DEBUG`: enable debugging-related code
- `LIBPROFIT_NO_OPENCL`: disable OpenCL support
- `LIBPROFIT_NO_OPENMP`: disable OpenMP support
- `LIBPROFIT_NO_FFTW`: disable FFTW support
- `LIBPROFIT_NO_SIMD`: disable SIMD extensions usage

Please refer to the `cmake` documentation for further options.

CHAPTER 2

Using *libprofit*

2.1 From the command-line

libprofit ships with a command-line utility `profit-cli` that reads the model and profile parameters from the command-line and generates the corresponding image. It supports all the profiles supported by *libprofit*, and can output the resulting image as text values, a binary stream, or as a simple FITS file.

Run `profit-cli -h` for a full description on how to use it, how to specify profiles and model parameters, and how to control its output.

2.2 Programatically

As it name implies, *libprofit* also ships a shared library exposing an API that can be used by any third-party application. This section gives a brief overview on how to use this API. For a full reference please refer to *API*.

At the core of *libprofit* sits `Model`. This class holds all the information needed to generate an image. Different profiles (instances of `Profile`) are appended to the model, which is then evaluated.

The basic usage pattern then is as follows:

1. Add the profit include:

```
#include <profit/profit.h>
```

2. Initialize the library with the `init()` function. This needs to be called *only once* in your program:

```
profit::init();
```

3. First obtain a model instance that will generate profile images for a given width and height:

```
profit::Model model(width, height);
```

4. Create a profile. For a list of supported names see *Profiles*; if you want to support a new profile see *Adding a profile*. If an unknown name is given an `invalid_parameter` exception will be thrown:

```
profit::ProfilePtr sersic_profile = model.add_profile("sersic");
```

5. Customize your profile. To set the different parameters on your profile call `Profile::parameter()` with the parameter name and value:

```
sersic_profile.parameter("xcen", 34.67);
sersic_profile.parameter("ycen", 9.23);
sersic_profile.parameter("axrat", 0.345);
sersic_profile.parameter("nser=3.56");
// ...
```

   A complete list of parameters can be found on and *Profiles* and *API*.

6. Repeat the previous two steps for all profiles you want to include in your model.
7. Evaluate the model simply run:

```
profit::Image result = model.evaluate();
```

8. If the resulting image needs to be cropped (see *Image cropping* for full details) an additional argument needs to be passed to `Model::evaluate()` to receive the offset at which cropping needs to be, like this:

```
profit::Point offset;
profit::Image result = model.evaluate(offset);
profit::Image cropped_image = result.crop({width, height}, offset);
```

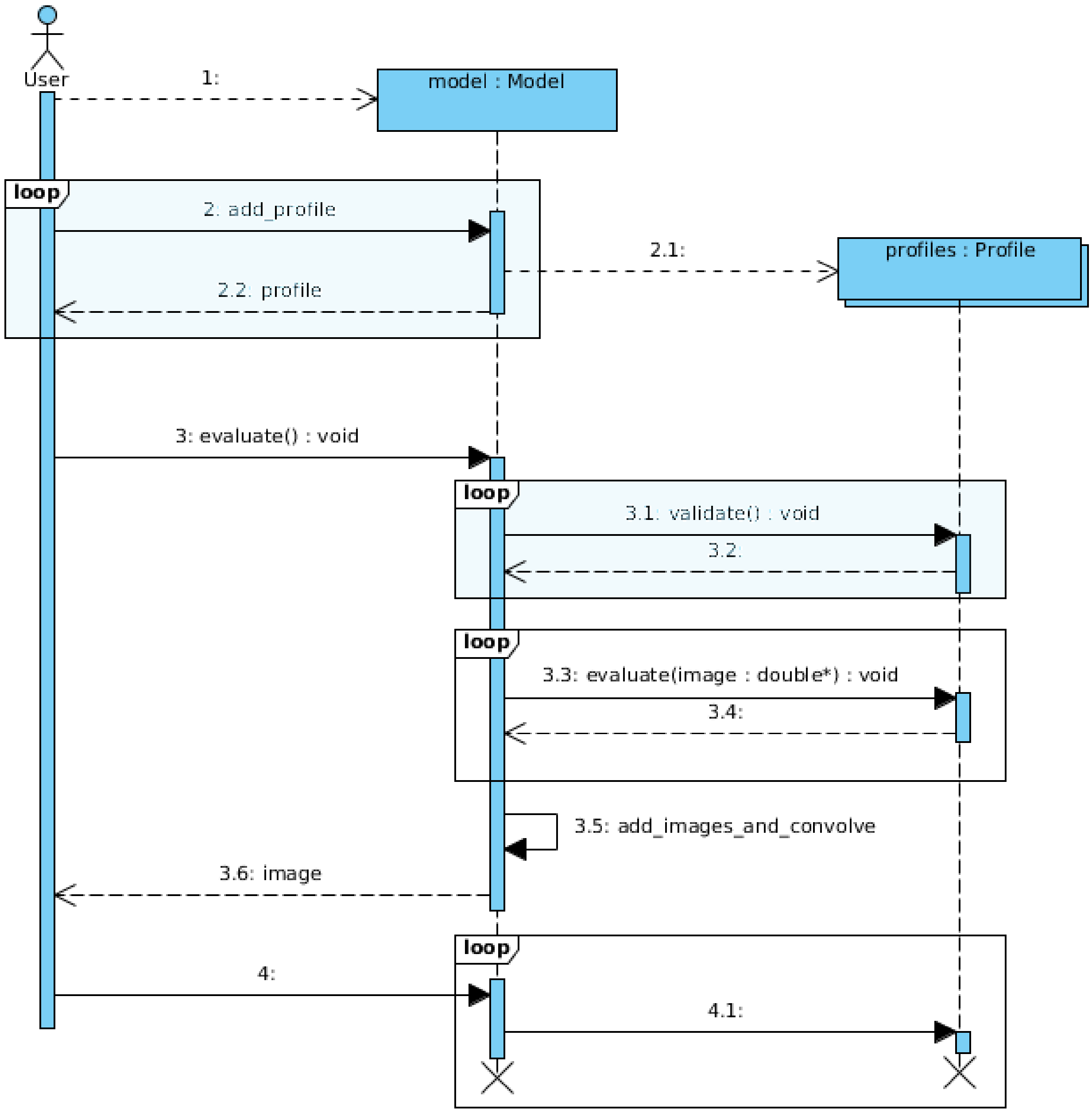
9. If there are have been errors while generating the image an `invalid_parameter` exception will be thrown by the code, so users might want to use a `try/catch` statement to identify these situations:

```
try {
    auto result = model.evaluate();
} catch (profit::invalid_parameter &e) {
    cerr << "Oops! There was an error evaluating the model: " << e.what() << endl;
}
```

10. When the model is destroyed the underlying profiles are destroyed as well.
11. When you are finished using the library, call the `finish()` function:

```
profit::finish();
```

To illustrate this process, refer to the following figure:


User
1:
model : Model
loop
2: add_profile
2.1:
profiles : Profile
2.2: profile
3: evaluate() : void
loop
3.1: validate() : void
3.2:
loop
3.3: evaluate(image : double*) : void
3.4:
3.5: add_images_and_convolve
3.6: image
loop
4:
4.1:

CHAPTER 3

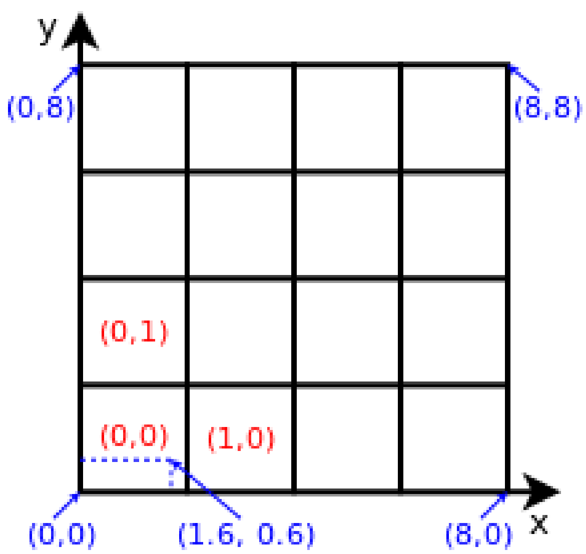
Coordinates

libprofit differentiates pixels from actual image coordinates. Pixels are the individual dots that make up an image, but from the profiles' point of view the area where the image is drawn is a continuum coordinate space. Profiles use this coordinates to perform their calculations, as they are more fine grained than individual pixels.

At the model level there are two sets of parameters that control these two different aspects of the image:

- The `width` and `height` parameters indicate the width and height of the image in *numbers of pixels*.
- The `scale_x` and `scale_y` parameters indicate the horizontal and vertical scale to convert the `width` and `height` parameters into image coordinate sizes.

This is shown in the following example:



In the example an image is shown using both their pixels and their image coordinates. Each square on the grid represents a pixel on the image, which are indexed in red. This image's `width` is 4, and its `height` is also 4. Shown in blue on the other hand are the image coordinates. The image's `scale_x` is 2 and its `scale_y` is also 2. Finally a point is indicated in the image. Its image coordinates are `(1.6, 0.6)`, and is contained within the `(0,0)` pixel.

Profiles use image coordinate to perform their calculations but still need to store only one value per pixel. For this purpose the quantities `scale_x` and `scale_y` are stored by *libprofit* at the model level, making them available to all profiles to use. They indicate the width and height of each pixel in image coordinates. In most cases profiles evaluate a pixel's value using the pixel's center point in image coordinates; that is, evaluating at `x = i*scale_x + scale_x/2`, where `i` is the horizontal pixel index, and likewise for the vertical coordinate. In other cases, like in the `sersic` profile, sub-sampling needs to be performed to achieve an accurate result.

CHAPTER 4

Profiles

Contents

This section lists the profiles currently supported by *libprofit*.

4.1 sersic

An implementation of the Sersic luminosity profile. The sersic profile describes the intensity of a galaxy depending on its distance to the center.

The sersic profile accepts the following parameters:

- **xcen**: x centre of the Sersic profile (can be fractional pixel positions).
- **ycen**: y centre of the Sersic profile (can be fractional pixel positions).
- **mag**: Total magnitude of the Sersic profile. Converted to flux using `flux = 10^(-0.4*(mag - magzero))`, where `magzero` is that of the containing model.

- **re**: Effective radius
- **nser**: Sersic index of the Sersic profile.
- **ang**: The orientation of the major axis of the Sersic profile, in degrees. The starting point is the positive Y image axis and grows counter-clockwise.
- **axrat**: Axial ratio of the Sersic profile defined as minor-axis/major-axis, i.e. `axrat = 1` is a circle and `axrat = 0` is a line.
- **box**: The boxiness of the Sersic profile that traces contours of iso-flux, defined such that r = (x^(2+box)+y^(2+box))^(1/(2+box)). When `box = 0` the iso-flux contours will be normal ellipses, but modifications between `-1 < box < 1` will produce visually boxy distortions. Negative values have a pin-cushion effect, whereas positive values have a barrel effect (the major and minor axes staying fixed in all cases).

The sersic profile implements recursive sub-pixel sampling for better results in areas closer to the profile center. This sub-sampling can be controller by the following additional parameters:

- **rough**: Don't perform any sub-sampling, ignore the rest of the parameters.
- **rscale_switch**: Radius scale fraction within which sub-sampling is performed. Pixels outside this radius are not sub-sampled.
- **max_recursions**: The maximum levels of recursions allowed.
- **resolution**: Resolution (both horizontal and vertical) to be used on each new recursion level.
- **acc**: Accuracy after which recursion stops.

The sersic profile also implements far-pixel filtering, quickly zeroing pixels that are too far away from the profile center. This filtering can be controller by the following parameters:

- **rscale_max**: Maximum *re*-based distance to consider for filtering.
- **rescale_flux**: Whether the calculated profile flux should be scaled to take into account the filtering performed by **re_max**.

Finally, an **adjust** parameter allows the user whether adjustments of most of the parameters described above should be done automatically depending on the profile parameters. *libprofit* makes a reasonable compromise between speed and accuracy, and therefore this option is turned on by default.

4.2 `moffat`

The moffat profile works in exactly the same way as the sersic profile. It also supports sub-pixel sampling using the same parameters. Because of the nature of the profile the `re` and `nser` parameters from the `sersic` profiles are not present, and instead the following new parameters appear:

- **fwhm**: Full-width at half maximum of the profile across the major-axis of the intensity profile.
- **con**: Profile concentration.

4.3 `ferrer`

Again, the ferrer profile works in exactly the same way as the sersic profile. It replaces the `re` and `nser` parameters from the `sersic` profile with:

- **rout**: The outer truncation radius.
- **a**: The global power-law slope to the profile center

- **b**: The strength of truncation as the radius approaches **rout**.

4.4 coresersic

The coresersic profile works in exactly the same way as the sersic profile. In addition to the `re` and `nser` parameters from the `sersic` profile it also adds:

- **rb**: The transition radius of the sersic profile.
- **a**: The strength of the transition from inner core to outer sersic
- **b**: The inner power-law of the core sersic.

4.5 king

The king profile works in exactly the same way as the sersic profile. It replaces the `re` and `nser` parameters from the `sersic` profile with:

- **rc**: The effective radius of the sersic component.
- **rt**: The transition radius of the sersic profile
- **a**: The power-law of the King.

4.6 brokenexp

The broken exponential profile works in exactly the same way as the sersic profile. It replaces the `re` and `nser` parameters from the `sersic` profile with:

- **h1**: The inner exponential scale length.
- **h2**: The outer exponential scale length (must be equal to or less than `h1`).
- **rb**: The break radius.
- **a**: The strength of the truncation as the radius approaches `rb`.

4.7 sky

The sky profile provides a constant value for an entire image.

- **bg**: Value per pixel for the background. This should be the value as measured in the original image, i.e. there is no need to worry about the effect of the model's `magzero`.

4.8 psf

The psf profile adds the model's psf to the model's image at a specific location and for a given user-defined magnitude.

- **xcen**: The x position at which to generate the centred PSF (can be fractional pixels).
- **ycen**: The y position at which to generate the centred PSF (can be fractional pixels).
- **mag**: The total flux magnitude of the PSF.

4.9 `null`

The null profile leaves the image area untouched. It is only useful for testing purposes.

CHAPTER 5

Convolution

Contents

Image convolution in *libprofit* happens optionally as part of a `Model` evaluation. Internally, the `Model` uses a `Convolver` to perform convolution.

5.1 Supported convolution methods

Convolvers are objects that carry out convolution (via their `Convolver::convolve()` method). Depending on the size of the problem, and on the libraries available on the system, different convolver types will be available to be used:

- `BRUTE_OLD` is the simplest convolver. It implements a simple, brute-force 2D convolution algorithm.
- `BRUTE` is a brute-force convolver that performs better that `BRUTE_OLD`, but still implements simple, brute-force 2D convolution. It is the default convolver used by a `Model` that hasn't been assigned one, but requires one.
- `FFT` is a convolver that uses Fast Fourier transformations to perform convolution. Its complexity is lower than the `BRUTE`, but its creation can be more expensive.
- `OPENCL` is a brute-force convolver implemented in OpenCL. It offers both single and double floating-point precision and its performance is usually better that that of the `BRUTE`.

5.2 Creating a Convolver

Instead of manually selecting the class that should be used, users create `Convolver` instances via the `create_convolver()` function. `create_convolver()` lets the user specify which type of convolver should be created (either using an enumeration, or a standard string value), and a set of creation preferences that apply differently to different types of Convolvers.

If a `Model` needs to perform convolution and a `Convolver` has been set on its `Model::convolver` member then that convolver is used. If no convolver has been set, it creates a new `BRUTE` and uses that to perform the convolution.

5.3 Using a convolver

Once created, users can call the `Convolver::convolve()` method directly on the resulting convolver, (or assign it to a `Model` instance for it to use it). The `Convolver::convolve()` methods needs at least three parameters: an image, a kernel and a mask. Convolvers will convolve the image with the kernel only for the pixels in which the mask is set, or for all pixels if an empty mask is passed. This implies that the mask, if not empty, must have the same dimensions that the image.

5.4 Image cropping

Some convolvers internally work with images that are larger than the original source image (mostly due to efficiency reasons). After this internal image expansion occurs, and the convolution takes place, the resulting image is usually cropped at the corresponding point to match original source image size and positioning before being returned to the user.

However, users might want to pick into this internal, non-cropped result of the convolution process. To do this, an additional `crop` parameter in the `Convolver::convolve()` method determines whether the convolver should return the original, and potentially bigger, image. When a non-cropped image is returned, an additional `offset_out` parameter can be given to find out the offset at which cropping would have started. The cropping dimensions do not need to be queried, as they always are the same of the original source image given to the convolver.

5.5 Model convolution

During model evaluation (i.e., a call to `Model::evaluate()`) users might want to be able to retrieve the non-cropped result of the internal convolution that takes place during model evaluation (as explained in *Image cropping*).

To do this, users must first call `Model::set_crop()` with a `false` argument. When calling `Model::evaluate()`, users must then also give a `Point` argument to retrieve the offset at which cropping should be done to remove the image padding added by the convolution process.

CHAPTER 6

Flux capturing

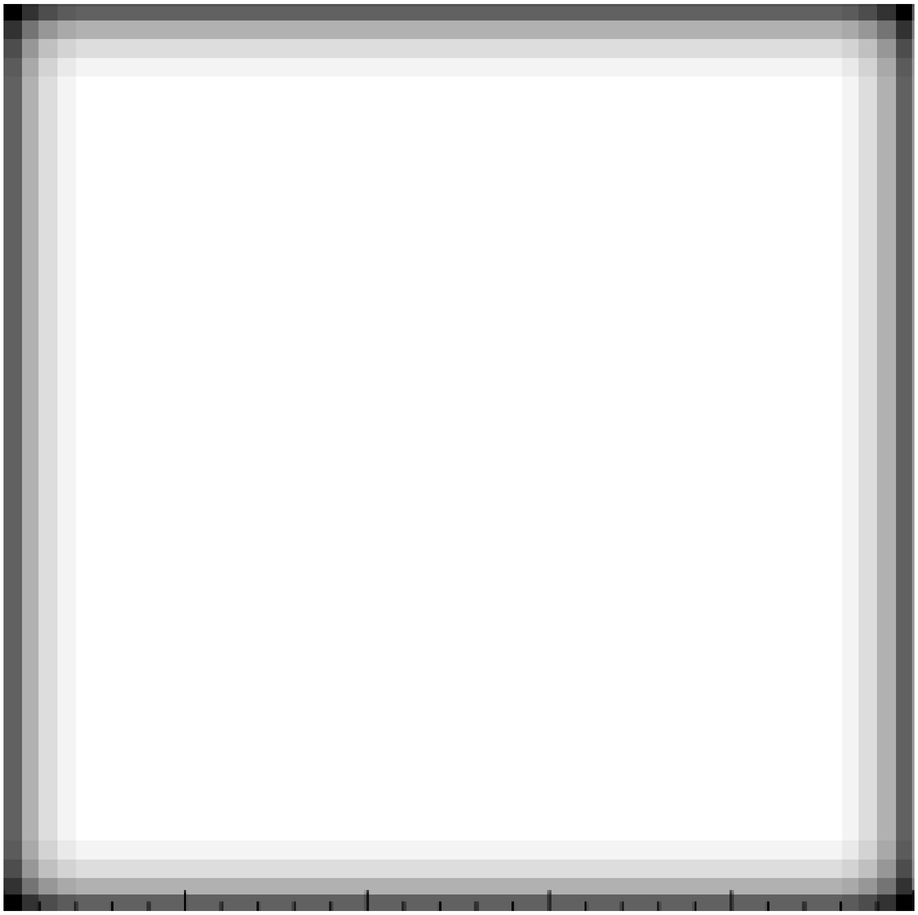
Flux capturing refers to the action of fully considering flux during the convolution process.

6.1 What is it

With no convolution taken into account, profiles already generate the correct luminosity for each pixel in their respective image. After that, if there is an extra convolution step the luminosity in these pixels gets spread out into their neighbours, and vice-versa.

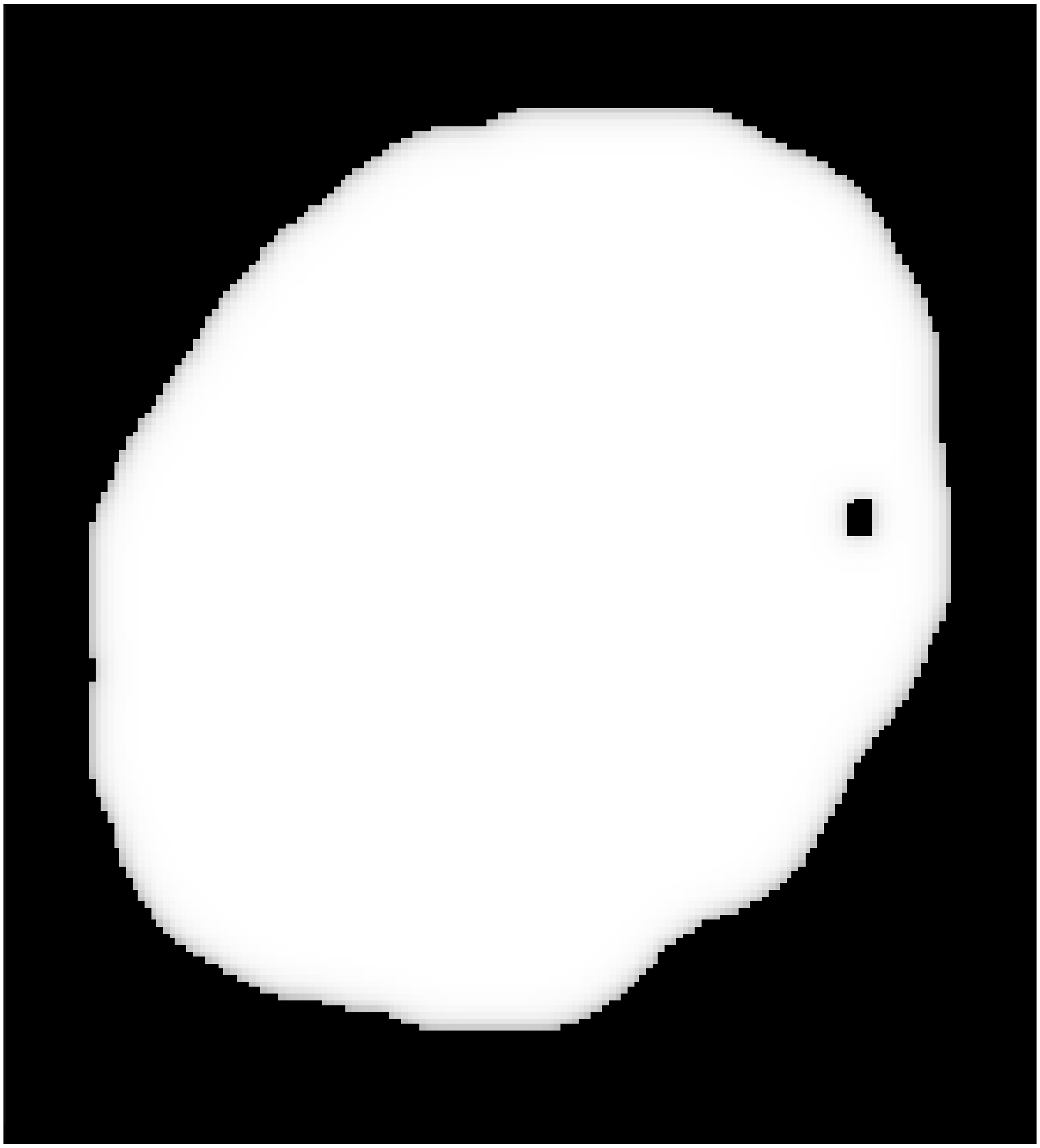
This is all and well, except when pixels are close to the edges of the image. When this happens, some of their neighbours are *outside* the image frame. Because they are outside, they had not been evaluated by the profiles, and their fluxes are considered to be 0. Therefore, when convolution happens for these pixels, their final luminosity will be less than what *would have been* if there was a value calculated for those pixels outside of the image.

This is better seen graphically. Here is an example of a plain image (using a *sky* profile) after convolution with a simple, gaussian-like PSF:



In this image, it is very clear how the pixels at the boundaries of the image frame are loosing flux during convolution. Again, this is because pixels *outside the image* have no flux, and therefore don't contribute to the flux of the pixels within the image after convolution.

A similar situation happens when there is a Mask involved:



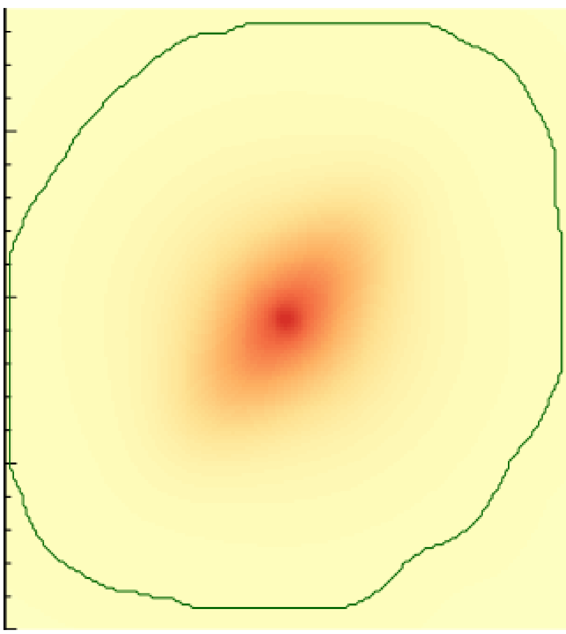
And a zoom into the top-right corner:

Correct flux capturing is important to correctly represent observed images, which *do include* flux coming from outside the field of view.

6.2 How it works

As of *libprofit* 1.9.0, flux capturing is automatically taken into account without the user needing to worry about modifying the inputs to a `Model`. This process takes into account whether convolution is needed at all, the size of the PSF, the original `Mask`, the finesampling factor set into the `Model`, and any other factor affecting this computation.

There are two main transformations that take place when this automatic adjustment happens internally in the `Model`. First, if there is a `Mask` involved, its coverage is expanded. This coverage expansion is a convolution-like operation, which is required so the profiles evaluate pixels outside of the originally-intended area. Like this:



In the figure above, the left-hand image shows the original coverage of the `Mask`, while the right-hand side shows the *expanded* coverage after taking into account that the convolution process will require all the pixels in the expanded area to have values calculated on them.

The second transformation that needs to happen is the expansion of the `Model` dimensions. Like in the first case, if pixels outside of the `Model` original image frame need to be calculated, then the `Model` dimensions need to be adjusted so these pixels receive values during profile evaluation.

6.3 Using pre-calculated Masks

The process described above happens automatically without the user having to adjust any of the inputs of the `Model`.

However, in the case of image fitting, when we know that the original `Mask`, PSF, and other inputs will not change across evaluations of a `Model`, some work can be pre-calculated. In particular, the final form of the `Mask` can be reused across `Model` evaluations. This is done in a two-step process:

- The user first calls `Model::adjust()` with the intended inputs. This results on a pre-calculated `Mask` that works for that set of inputs
- Then, the mask is passed down as usual to the `Model` via `Model::set_mask()`. Additionally, the `Model` is informed that no further adjustment should be done on that mask via `Model::set_adjust_mask()` with a `false` argument.

6.4 Current and previous status

Until libprofit 1.9.0, images produced by *libprofit* failed to correctly capture flux correctly in scenarios when there was a convolution involved. The ProFit R package implemented this as part of its fitting process though, but other users would have been lacking this feature.

CHAPTER 7

Adding a profile

Contents

This section explains the steps required to add a new profile to *libprofit*.

In a nutshell, to add a new profile one must:

- Create a new subclass of `Profile`
- Write the mandatory methods
- Associate the new profile with a standard name

In all steps below, a completely artificial `example` profile is being added, This new profile takes three parameters: `param1` and `param2` are double numbers, while `param3` is an unsigned integer. The profile fills the image by taking the X and Y coordinates and filling the pixel with the value `|(param1 - param2) * param3 * (x - y)|` and requires that all parameters are positive or 0.

The data types used in this example are described in detail in *API*.

7.1 New Class

The first step to add a new profile is to define the C++ class that will hold all its information. Any kind of information can be added to the class, but it is **required** that the class extends the base `Profile` class. The class should be defined in an `.h` file in the `profit` directory so it can be used by others, and should be part of the `profit` namespace.

So far, it should look like this:

```
class ExampleProfile : public Profile {
private:
    double param1;
    double param2;
    unsigned int param3;
};
```

7.2 Methods

Each profile requires a minimum of three methods that need to be written:

- The constructor,
- A method to validate the profile's values, and
- A method to evaluate it.

The two latter are imposed by the base class, and must be called `validate` and `evaluate`.

In addition, to be able to receive parameters given by the user, the `parameter` methods must be overwritten.

7.2.1 Parameters

To receive parameters given by the user the new class must overwrite the necessary `parameter` methods from the parent class. There are several flavours of this methods, depending on the parameter data type, so only the necessary ones are required.

In our example we only have parameters of type *double* and *unsigned int* so we only need to overwrite those two methods. This method must call its parent method to check if it already set a parameter with that name, in which case it should short-cut and return *true*; it then should check the parameter name against its own parameters, and return either *true* or *false* if the parameter was set or not.

In our example, *double* parameters are set like this:

```
void ExampleProfile::parameter(const std::string &name, double value) {

    if( Profile::parameter(name, value) ) {
       return true;
    }

    if( name == "param1" )       { param1 = value; }
    else if( name == "param2" ) { param2 = value; }
    else {
       return false;
    }

    return true;
```

(continues on next page)

(continued from previous page)

```
}
```

7.2.2 Validation

After parameters are all set, *libprofit* will call the validation function. The validation function's responsibility, as its name implies, is to validate the inputs of the profile, checking that they obey the required minimum to make the operation successful.

In the case of the `example` profile it was mentioned that all parameters must be positive, so the code must test for that. If a violation occurs, a `invalid_parameter` exception is thrown. This exception will prevent the profile (and in fact the whole model) from being evaluated.

An example implementation would thus look like this:

```
void ExampleProfile::validate() {

    if ( this->param1 < 0 ) {
       throw invalid_parameter("param1 is negative");
    }
    if ( this->param1 < 0 ) {
       throw invalid_parameter("param2 is negative");
    }
    if ( this->param3 < 0 ) {
       throw invalid_parameter("param3 is negative");
    }

}
```

Note also that the base `Profile` class has a reference to the model this profile is part of. Having access to the model means that one can validate profile-specific values against model-global values as well. For example, if a new restriction is added stating that the `example` profile can only be run on images that are bigger than 20 x 20 then the following code could be added:

```
if ( this->model->width < 20 || this->model->height < 20 ) {
    throw invalid_parameter("can't apply example profile to images less than 20x20");
}
```

Finally, if a profile needs no validation at all a validation function must still be provided with an empty body.

7.2.3 Evaluation

Next, we look to the `evaluate` method. Its `image` argument corresponds to the surface where the pixels must be drawn. All profiles in the model receive **the same** image surface, so care must be taken to *add* values into the image's pixel rather than *setting* them. The image is already initialized with zeros when created, so if your profile doesn't cover the entire image no action needs to occur.

It was mentioned earlier that the `example` profile fills the image by taking the X and Y coordinates and filling the pixel with the value `|(param1 - param2) * param3 * (x - y)|`. An implementation of this would then look like this:

```
void ExampleProfile::evaluate(std::vector<double> &image) {

```

(continues on next page)

(continued from previous page)

```
3     Model *model = this->model;
4     double x, y;
5     unsigned int i, j;
6     double half_xbin = this->model->scale_x/2.;
7     double half_ybin = this->model->scale_y/2.;
8
9     x = 0;
10    for (i=0; i < model->width; i++) {
11        x += half_xbin;
12
13        y = 0;
14        for (j=0; j < model->height; j++) {
15           y += half_ybin;
16
17           if ( !model->calcmask || model->calcmask[i + j*model->width] ) {
18              double val = fabs( (this->param1 - this->param2) * this->param3 * (x - y) );
19              image[i + j*model->width] = val;
20           }
21
22           y += half_ybin;
23        }
24        x += half_xbin;
25    }
26 }
```

The code above performs the following steps:

1. On line 10 we loop around the X axis. `i` is the horizontal pixel index on the image and spans from 0 to `model->width`. At the same time we keep track of `x`, which is a floating point number representing the horizontal image coordinate used to evaluate the profile on that pixel. See *Coordinates* for more details on the coordinate system used by *libprofit*.
2. Similarly, on line 14 we loop around the Y axis.
3. The model might specify a calculation mask, indicating that some pixels should not be calculated, which is checked in line 17
4. Being now on a given X and Y coordinate, we evaluate our profile on line 18.
5. Finally on line 19 we store the evaluated profile on the corresponding pixel of the image.

7.2.4 Constructor

Last but not least we look at the constructor. Its signature looks like this:

```
ExampleProfile(const Model &model, const std::string &name);
```

The constructor arguments must be passed down to the parent class. The constructor is also in charge of populating the profile with its default values. For this example the code would look like this:

```
ExampleProfile::ExampleProfile(const Model &model, const std::string &name) :
    Profile(model, name),
    param1(1.),
    param2(2.),
    param3(3)
{
```

(continues on next page)

(continued from previous page)

```
  // no-op
}
```

7.3 Wiring up

To finally wire up your new profile with the rest of *libprofit* you need to give it a name. This is done at the `profit.cpp` file. Open it in an editor and look for the `Model::add_profile` method. This method creates different profile instances based on the given name. Add a new `else if` statement to create your new profile imitating what is done for the other ones.

To add the `example` profile the following lines should thus be added to the first `if/else if` block:

```
else if ( profile_name == "example" ) {
    profile = static_cast<Profile *>(new ExampleProfile());
}
```

In order to be able to "see" the constructor the `example.h` file must also be included, which is done earlier on in `profit.cpp`:

```
#include "profit/example.h"
```

Finally, you need to manually add the new `.cpp` file to the list of files to be compiled. This is done by adding it to the `PROFIT_SRC` list in the `CMakeLists.txt` file:

```
set(PROFIT_SRC
  [...]
  src/example.cpp
  [...]
)
```

7.4 Full example

Below are the full new files that have been described below. `example.h` contains the new data type definition, plus the signature of the creation function, while `example.cpp` contains the implementation of the creation, validation and evaluation of `example` profiles.

Listing 1: example.h

```
/* copyright notice, etc */
#ifndef _EXAMPLE_H_
#define _EXAMPLE_H_

#include <string>
#include <vector>

#include "profit/profile.h"

namespace profit
{

class ExampleProfile : public Profile {
```

(continues on next page)

(continued from previous page)

```

public:
	ExampleProfile(const Model &model, const std::string &name);
	void validate() override;
	void evaluate(Image &image, const Mask &mask, const PixelScale &scale, const Point &offset, double magzero) override;

protected:
	bool parameter(const std::string &name, double value);
	bool parameter(const std::string &name, unsigned int value);

private:
	double param1;
	double param2;
	unsigned int param3;

};

} /* namespace profit */

#endif
```

Listing 2: example.cpp

```
/* copyright statement, etc */

#include <cmath>
#include "example.h"

#include "profit/exceptions.h"
#include "profit/model.h"

namespace profit {

ExampleProfile::ExampleProfile(const Model &model, const std::string &name) :
    Profile(model, name),
	 param1(1.),
	 param2(2.),
	 param3(3)
{
	// no-op
}

bool ExampleProfile::parameter(const std::string &name, double value) {

	if( Profile::parameter(name, value) ) {
		return true;
	}

	if( name == "param1" )      { param1 = value; }
	else if( name == "param2" ) { param2 = value; }
	else {
		return false;
	}

	return true;
```

(continues on next page)

(continued from previous page)

```
}

bool ExampleProfile::parameter(const std::string &name, unsigned int value) {
        if( Profile::parameter(name, value) ) {
                return true;
        }

        if( name == "param3" ) { param3 = value; }
        else {
                return false;
        }

        return true;
}

void ExampleProfile::validate() {

        if ( this->param1 < 0 ) {
                throw invalid_parameter("param1 is negative");
        }
        if ( this->param1 < 0 ) {
                throw invalid_parameter("param2 is negative");
        }
        if ( this->param3 < 0 ) {
                throw invalid_parameter("param3 is negative");
        }

        /*
        if ( this->model->width < 20 || this->model->height < 20 ) {
                throw invalid_parameter("can't apply example profile to images less than 20x20");
        }
        */
}

void ExampleProfile::evaluate(Image &image, const Mask &mask, const PixelScale &scale, double magzero) {

        double x, y;
        unsigned int i, j;
        auto width = image.getWidth();
        double half_xbin = scale.first/2.;
        double half_ybin = scale.second/2.;

        x = 0;
        for (i=0; i < width; i++) {
                x += half_xbin;

                y = 0;
                for (j=0; j < image.getHeight(); j++) {
                        y += half_ybin;

                        if ( not mask or mask[i + j * width] ) {
                                double val = std::abs( (this->param1 - this->param2) * this->param3 * (x - y) );
                                image[i + j * width] = val;
                        }
```

(continues on next page)

(continued from previous page)

```

                                y += half_ybin;
                        }
                        x += half_xbin;
                }
        }

} /* namespace profit */
```

CHAPTER 8

Language Bindings

Bindings exist to wrap *libprofit* into different languages.

At the moment of writing the following two are available:

- pyprofit: a Python wrapper for *libprofit*.
- ProFit: A package for R that wraps *libprofit* and performs high-level profile fitting against an input galaxy.

Additional language bindings can be easily added in the future if required.

CHAPTER 9

API

9.1 Library

Warning: doxygenenum: Cannot find enum "profit::simd_instruction_set" in doxygen xml output for project "libprofit" from directory: doxygen-output/xml

Warning: doxygenfunction: Cannot find function "profit::init" in doxygen xml output for project "libprofit" from directory: doxygen-output/xml

Warning: doxygenfunction: Cannot find function "profit::finish" in doxygen xml output for project "libprofit" from directory: doxygen-output/xml

Warning: doxygenfunction: Cannot find function "profit::version" in doxygen xml output for project "libprofit" from directory: doxygen-output/xml

Warning: doxygenfunction: Cannot find function "profit::version_major" in doxygen xml output for project "libprofit" from directory: doxygen-output/xml

Warning: doxygenfunction: Cannot find function "profit::version_minor" in doxygen xml output for project "libprofit" from directory: doxygen-output/xml

Warning: doxygenfunction: Cannot find function "profit::version_patch" in doxygen xml output for project "libprofit" from directory: doxygen-output/xml

Warning: doxygenfunction: Cannot find function "profit::has_openmp" in doxygen xml output for project "libprofit" from directory: doxygen-output/xml

Warning: doxygenfunction: Cannot find function "profit::has_fftw" in doxygen xml output for project "libprofit" from directory: doxygen-output/xml

Warning: doxygenfunction: Cannot find function "profit::has_fftw_with_openmp" in doxygen xml output for project "libprofit" from directory: doxygen-output/xml

Warning: doxygenfunction: Cannot find function "profit::has_opencl" in doxygen xml output for project "libprofit" from directory: doxygen-output/xml

Warning: doxygenfunction: Cannot find function "profit::opencl_version_major" in doxygen xml output for project "libprofit" from directory: doxygen-output/xml

Warning: doxygenfunction: Cannot find function "profit::opencl_version_minor" in doxygen xml output for project "libprofit" from directory: doxygen-output/xml

Warning: doxygenfunction: Cannot find function "profit::has_simd_instruction_set" in doxygen xml output for project "libprofit" from directory: doxygen-output/xml

9.2 Exceptions

Warning: doxygenclass: Cannot find class "profit::invalid_parameter" in doxygen xml output for project "libprofit" from directory: doxygen-output/xml

Warning: doxygenclass: Cannot find class "profit::unknown_parameter" in doxygen xml output for project "libprofit" from directory: doxygen-output/xml

Warning: doxygenclass: Cannot find class "profit::opencl_error" in doxygen xml output for project "libprofit" from directory: doxygen-output/xml

Warning: doxygenclass: Cannot find class "profit::fft_error" in doxygen xml output for project "libprofit" from directory: doxygen-output/xml

9.3 Imaging classes

Warning: doxygenclass: Cannot find class "profit::_2dcoordinate" in doxygen xml output for project "libprofit" from directory: doxygen-output/xml

Warning: doxygentypedef: Cannot find typedef "profit::Dimensions" in doxygen xml output for project "libprofit" from directory: doxygen-output/xml

Warning: doxygentypedef: Cannot find typedef "profit::Point" in doxygen xml output for project "libprofit" from directory: doxygen-output/xml

Warning: doxygenclass: Cannot find class "profit::surface" in doxygen xml output for project "libprofit" from directory: doxygen-output/xml

Warning: doxygenclass: Cannot find class "profit::Image" in doxygen xml output for project "libprofit" from directory: doxygen-output/xml

Warning: doxygenclass: Cannot find class "profit::Mask" in doxygen xml output for project "libprofit" from directory: doxygen-output/xml

9.4 Model class

Warning: doxygenclass: Cannot find class "profit::Model" in doxygen xml output for project "libprofit" from directory: doxygen-output/xml

9.5 Profile classes

Warning: doxygenclass: Cannot find class "profit::Profile" in doxygen xml output for project "libprofit" from directory: doxygen-output/xml

Warning: doxygenclass: Cannot find class "profit::RadialProfile" in doxygen xml output for project "libprofit" from directory: doxygen-output/xml

Warning: doxygenclass: Cannot find class "profit::SersicProfile" in doxygen xml output for project "libprofit" from directory: doxygen-output/xml

Warning: doxygenclass: Cannot find class "profit::MoffatProfile" in doxygen xml output for project "libprofit" from directory: doxygen-output/xml

Warning: doxygenclass: Cannot find class "profit::FerrerProfile" in doxygen xml output for project "libprofit" from directory: doxygen-output/xml

Warning: doxygenclass: Cannot find class "profit::CoreSersicProfile" in doxygen xml output for project "libprofit" from directory: doxygen-output/xml

Warning: doxygenclass: Cannot find class "profit::BrokenExponentialProfile" in doxygen xml output for project "libprofit" from directory: doxygen-output/xml

Warning: doxygenclass: Cannot find class "profit::KingProfile" in doxygen xml output for project "libprofit" from directory: doxygen-output/xml

Warning: doxygenclass: Cannot find class "profit::PsfProfile" in doxygen xml output for project "libprofit" from directory: doxygen-output/xml

Warning: doxygenclass: Cannot find class "profit::SkyProfile" in doxygen xml output for project "libprofit" from directory: doxygen-output/xml

Warning: doxygenclass: Cannot find class "profit::NullProfile" in doxygen xml output for project "libprofit" from directory: doxygen-output/xml

9.6 Convolvers

Warning: doxygenenum: Cannot find enum "profit::ConvolverType" in doxygen xml output for project "libprofit" from directory: doxygen-output/xml

Warning: doxygenclass: Cannot find class "profit::Convolver" in doxygen xml output for project "libprofit" from directory: doxygen-output/xml

Warning: doxygenclass: Cannot find class "profit::ConvolverCreationPreferences" in doxygen xml output for project "libprofit" from directory: doxygen-output/xml

Warning: doxygenfunction: Cannot find function "profit::create_convolver" in doxygen xml output for project "libprofit" from directory: doxygen-output/xml

Warning: doxygenfunction: Cannot find function "profit::create_convolver" in doxygen xml output for project "libprofit" from directory: doxygen-output/xml

CHAPTER 10

Changelog

Development version

- A new `null` convolver has been added that does no convolution and simply returns the source image unmodified. This is only useful for testing.

1.9.3

- A bug in the OpenCL implementation of the radial profiles prevented Models with multiple profiles from displaying correctly, as the output image would contain only the values of last profile. This was a problem introduced only in the last version of *libprofit*, and not an ongoing issue.
- When using OpenCL, any radial profile specifying `rough=true` caused the output image not to be scaled properly, with values not taking into account the profile's magnitude or pixel scale. This seems to have been an issue for a long time, but since `rough=true` is not a common option it had gone under the radar for some time.

1.9.2

- All profile evaluation has been changed from being absolute (profiles set the final value of a pixel) to be additive (their add their pixel values onto the image). This change in behavior has the effect that one less memory allocation is needed, which can be a big difference when generating big images, while also simplifying the logic of the `Model` evaluation.
- `Model` objects now internally store the normalized version of the PSF given by the user instead of the original, which was never really needed.
- **`profit-cli`** now makes it easier to specify multiple copies of the same profile, useful for scaling tests. Also, writing FITS files in little endian systems doesn't allocate extra memory anymore.
- Minor improvements to imaging classes.

1.9.1

- The implementation of the `Model` class has been improved. In particular it has been made more memory efficient, which is particularly important in scenarios where many profiles (in the order of thousands) are added into it. Previously each profile was allocated its own `Image`, which added both to the memory footprint, and to the total runtime. Now a single scratch space is used for all profiles, and individual results are immediately summed up, respecting the convolution settings of each profile. Experiments with the *null profile* show a significant decrease in runtime when many Model evaluations take place.

1.9.0

- Implemented correct *flux capturing*. This feature was previously implemented in the ProFit R package as part of its fitting process, but it was otherwise unavailable.
- Added explicit support to allow convolution of images against kernels with bigger dimensions than the images themselves. This was previously supported implicitly, and only in certain cases, by the OpenCL convolver, while the FFT convolver threw an proper exception, and the brute-force convolvers usually crashed. This first implementation is not ideal, but the use case is rare.
- Several performance and code improvements, like removing unnecessary code, avoiding unnecessary conversions and avoiding a few dynamic allocations.

1.8.2

- Users can now select the underlying SIMD-capable instruction set to use for brute-force convolution.
- New library method `has_simd_instruction_set()` for users to check whether libprofit was compiled with support for different instruction sets.
- Improved FFTW-based convolver performance by avoiding dynamic memory allocation at convolution time. This brings a noticeable performance improvement of around 20%.

1.8.1

- Adding support for FFTW versions lower than 3.3.

1.8.0

- **`profit-cli`** compiling in Windows.
- New `Profile::parameter()` method to specify parameters and values with a single `name=value` string.
- New utility methods: `trim()`, `split()` and `setenv()`.
- Using SSE2/AVX SIMD extensions to implement brute-force convolution if the CPU supports it, with pure C++ implementation as a fallback. Can be disabled with `-DLIBPROFIT_NO_SIMD=ON`.
- Potentially fixed the importing of FFTW wisdom files in systems with more than one FFTW installation.
- Fixed compilation of `brokenexponential` OpenCL kernel in platforms where it was failing to compile.
- Compiling in release mode (i.e., `-O3 -DNDEBUG` in gcc/clang) by default.
- Lowering OpenMP requirement to 2.0 (was 3.0).
- OpenCL kernel cache working for some platforms/devices that was not previously working.
- Many internal code cleanups and design changes to make code easier to read and maintain.

1.7.4

- FFT convolution using hermitian redundancy. This increases performance of FFT-based convolution by at least 10% in release builds, and addresses some warnings pointed out by `valgrind`.

1.7.3

- Added `init_diagnose()` and `finish_diagnose()` functions to avoid printing to stdout/stderr from within libprofit.

1.7.2

- Fixed `double` detection support for OpenCL devices regardless of the supported OpenCL version.
- Fixed a few compiling issues under Visual Studio compiler.
- Continuous integration in Windows via AppVeyor

1.7.1

- Added `Image::upsample()` and `Image::downsample()` to scale an image up or down (using different modes).
- Added `Model::set_return_finesampled()` to return internally upsampled images.

1.7.0

- Internal implementation dependencies clearly hidden from users. This means that users compiling against libprofit don't need to search for header files other than libprofit's, making it much easier to write code against libprofit.
- `Model` redesigned. No member variables are exposed anymore; instead different setter/getter methods must be used.
- `Image` redesigned. In summary, it looks much more like a standard container now.
- New `Model::set_crop()` specifies whether cropping should be carried out after convolution, if the convolution needs to pad the image.
- `Model::evaluate()` has an extra optional parameter to receive the offset at which cropping needs to happen (if it hasn't, see `Model::set_crop()`) to remove padding from the resulting image.
- FFTW convolution uses real-to-complex and complex-to-real forward and backwards transforms respectively (instead of complex-to-complex transforms both ways), which is more efficient and should use less memory.
- New on-disk OpenCL kernel cache. This speeds up the creation of OpenCL environments by a big factor as compilation of kernels doesn't happen every time an environment is created.
- New on-disk FFTW plan cache. This speeds up the creation of FFT-based convolvers by a big factor as the plans are not calculated every time for a given set of parameters.
- New `null` profile, useful for testing.
- New `init()` and `finish()` calls to initialize and finalize libprofit. These are mandatory, and should be called before and after using anything else from libprofit.

1.6.1

- Brute-force convolver about 3x faster than old version.
- Fixing compilation failure on MacOS introduced in 1.6.0.
- Center pixel in sersic profile treated specially only if `adjust` parameter is on.