Appendix A
phy$_{++}$: A C++ Library for Numerical Analysis

A.1 Introduction

A.1.1 A Brief Overview

phy$_{++}$ is a set of library and tools written in C++ that I developed during my PhD. The goal is to provide user-friendly vector data manipulation, as offered in interpreted languages like IDL,[1] its open source clone GDL,[2] or `python` & `numpy`,[3] but with the added benefit of C++: increased robustness, and optimal speed.

The library can be split into two components: the *core* library and the *support* library. The core library introduces the *vector* type, which is at the heart of *phy*$_{++}$, while the support library provides functions and other tools to manipulate these vectors and do some common tasks, ranging from low level mathematics and programming (sorting, integrating, binning, …) to higher level astrophysics-related tasks (such as cross-matching, stacking, SED fitting, …). You can think of the core library as "the language" (the equivalent of IDL or `python`), and the support library as "the function library" (the equivalent of the `IDLastro`,[4] `numpy` or `astropy`[5] libraries).

Below is an code sample written in *phy*$_{++}$ that illustrates its most basic functionalities.

[1] http://www.exelisvis.com/ProductsServices/IDL.aspx.

[2] http://gnudatalanguage.sourceforge.net/.

[3] http://www.numpy.org/.

[4] http://idlastro.gsfc.nasa.gov/.

[5] http://www.astropy.org/.

C. Schreiber, *A Statistical and Multi-wavelength Study of Star Formation in Galaxies*, Springer Theses, DOI 10.1007/978-3-319-44293-8

```
vec2f img = fits::read("img.fits"); // read a FITS image
img -= median(img);                 // subtract the median of the whole image
float imax = max(img);              // find the maximum of the image
vec1u ids = where(img > 0.5*imax);  // find pixels at least half as bright
float sum = total(img[ids]);        // compute the sum of these pixels
img[ids] = log(img[ids]/sum);       // modify these pixels with a logarithm
fits::write("new.fits", img);       // save the modified image to a FITS file
```

A.1.2 Why Write Something New?

The immediate goal of phy_{++} is to provide a syntax as close as possible to that of IDL. IDL is an interpreted language that is widely used in the scientific community, in particular in astrophysics. Born in the late 1970s, this language provides intuitive manipulation of large arrays of data using vectorized operations: applying an operation on a given array does not require the user to write a loop to iterate over its elements and apply the operation. This leads to very concise code that easy to write and read. Unfortunately, IDL suffers from a number of problems. I will start with the *political* and *ethical* problems.

- It is a proprietary, mostly[6] closed-source program. This means that IDL is a black box and that people using it have no choice but to rely on the IDL developers for writing accurate code. While there is an extensive documentation, the algorithms used by the procedures are not always described. This is hardly acceptable for scientific code.
- IDL, like C++, combines several languages into one: a functional language and an object-oriented language. It also contains a huge support library providing many features (having used IDL for more than two years, I could not list them all). For this reason, and because it is proprietary, maintaining this language and adding new features costs a lot of money to its owner, Exelis. This money, in turn, is provided by science labs all around the world, who pay a yearly fee for a bunch of IDL licenses. This is totally fine in itself, but the fact is that most IDL users I have seen only make use of a small sub-set of IDL, one that has barely evolved in twenty years. In this context, the price that is paid is not justified.
- On top of that, the licensing model is that of *floating* licenses: only a fixed, maximum number of simultaneously running IDL instance is allowed in the whole lab. With the now common budget restrictions in research, labs typically buy fewer licenses than there are users. Even worse, it is often needed to run multiple instances of IDL on a single computer, e.g., when working on two projects simultaneously. This will consume two licenses, even though there is a single user. This leads to silly situations, typically when approaching specific deadlines (e.g., deadlines for requesting observing time on large telescopes) where everyone needs to use IDL at the same time, but there is not enough license available. Even worse, we have

[6]The procedures from the IDL library that are written in IDL language are actually open-source, but all the procedures written in native language are compiled and only the binary is provided.

seen cases in our lab of users being unable to run IDL on their new shiny computer because of incompatibility, not with IDL itself, but with the licensing software. Lastly, it should be noted that this licensing model relies on having network connection with a license server. This means that one cannot use IDL while traveling unless a proper SSH tunneling is in place.

These issues can be solved by switching to one of the free and open-source alternatives, like GDL. The downside is that these implementations are lacking behind IDL in terms of features, as some useful functions are still to be implemented. Worse, some functions cannot *legally* be implemented because they would violate IDL's copyright.

But that's only half of the story. Indeed, IDL and GDL also suffer from technical issues. I will list below the most important ones.

- Designed in the 1970s, IDL was born in an era where the available RAM was scarce, and that great care had to be taken to consume as few bytes of memory as possible. For this reason, the default integer type in IDL is a `short`, i.e., it occupies only two bytes in memory, while most languages (including some that are older than IDL itself) encode their integers on four bytes by default. The biggest issue with this choice is that the largest number one can store in a `short` is `32768`. Being the default integer type, this creates quite a few surprises to the unexperienced user, and will fool even the expert from time to time.
- IDL is an interpreted language, meaning that the code you write is continuously read and interpreted by the IDL executable. While this is not an issue if you make good use of vectorization (the art of writing IDL code), performances are severely degraded once you write loops explicitly, because the content of the loop has to be *interpreted* and then *executed* on each iteration. And this is sometimes unavoidable.
- Like many interpreted languages, IDL is dynamically typed. This means that the type of a variable can change from one line to another, and that a variable containing a string can be assigned a number. While sometimes convenient, this comes at a cost: performance. Most IDL programs I have seen do not use this feature, yet they have to pay for it anyway.
- But worse than dynamic typing, and this is my main concern, variables in IDL are not *declared* before they are used. This means that if you do a typo in the name of one of your variables, chances are that the code will still run. Indeed, IDL cannot know that this was not intended, and will think that you want to create or modify a new variable. It will then do its best to carry on, and the result will be unpredictable. This, together with the fact that variables are almost not *scoped* (i.e., a variable created inside a `for` loop is still valid outside of the loop) makes it very easy to write confusing and buggy code. The most frightening part is that, in a good fraction of the cases, the output will be meaningful, and you can go on with your calculation never realizing that something went wrong. And publish that.

Avoiding the aforementioned issues is possible, but it requires coding with a fair amount of rigorousness and self discipline. My limited experience with astronomers

taught me that these are not particularly common character traits in the field, probably because we are all self taught programmers, but also because most of the code we write never goes out of our own computer and therefore does not get the chance to be reviewed an corrected by someone else. My conclusion is that, when it comes to checking the validity of a code, as much work as possible has to be done by the *language* itself (or its compiler), e.g., by being designed so that some errors cannot even be made, and that most of the remaining ones are identified *before* running the program and reported to the programmer so that he/she can fix them.

Switching to more modern interpreted languages like `python` or Julia[7] would solve a few of these issues, in particular the first one. But the other items on this list are unfortunately inherent to most interpreted languages.[8] To avoid these traps, the only solution today is to use *statically typed*, compiled languages, like C++.

Now, there are already some libraries in C++ that are addressing the topic of vector data manipulation. One can cite Eigen[9] or the more recent `blaze-lib`.[10] These are powerful libraries that have inspired phy_{++} in some way, but their issue is that they are more oriented toward algebra, meaning that they have vectors and matrices, but no data type for arrays of higher dimensions (i.e., tensors[11]).

Therefore, seeing that a gap had to be filled, phy_{++} was created.

A.1.3 Why C++?

There are many different compiled languages that offer similar or better performances than C++. In particular, the most famous ones are Fortran and C. C is impractical to use because it has not been developed with user-friendliness in mind, and no mechanism exist to improve that. This is a system language, and it does that perfectly, but not much more. Fortran is known as the fastest of all, and it is particularly well suited for numerical analysis. While few languages are harder to read than Fortran 77, things have become much better since Fortran 90 (which is not used as often as it should be). However, Fortran is relatively bad at doing anything else than numerical analysis, which is annoying the moment you want to do something that is a bit off the tracks. C++ on the other hand, with all its disadvantages, is probably the best fit thanks to its almost unlimited capacity for adaptation. And it also happens to be the language I am most familiar with.

Since the beginning, C++ has always been good at performances, first because it is a language that compiles directly into assembler instructions, but also thanks to its philosophy: "you only pay for what you ask for". But its main disadvantage is its *complexity*: it contains almost the whole C language, plus all the layers that

[7] http://julialang.org/.

[8] The best counter example is probably Java.

[9] http://eigen.tuxfamily.org.

[10] http://code.google.com/p/blaze-lib/.

[11] Eigen actually has a tensor module, but it is unsupported.

were added on top of it, one year after another, starting from classes, exceptions, then templates. The end result is that it is a challenging task to master all the aspects of this language.

But the good news is: you do not have to master all of C++, and for your sanity you probably should not. Indeed, there are a number of *sub-languages* made out of a subset of C++ that are completely self-sufficient, i.e. you can use them to write any program. In other words, there are multiple, very different ways of writing the same program in C++. Typically, modern programs only use a small fraction of the whole language, e.g., leaving aside most of what was inherited from C (raw arrays, raw pointers, explicit memory management, etc.). A special class of such sub-languages are those that are tailored specifically to address a given task, as opposed to being open to any purpose. These are called *domain-specific languages* (DSL), and only require learning a few of C++'s rules and concepts, plus the rules introduced by the sub-language itself. The phy_{++} library is an example of such domain-specific languages, its domain being vector data manipulation.

In short, although C++ is a very complex language, it is only necessary to learn a fraction of it to be able to use phy_{++} correctly. Of course, the more one knows about C++, the more one will be able to take advantage of all the features of phy_{++} in an optimal way.

A.1.4 Documentation

In this thesis, I do not include the library's full documentation. I figured this would be pointless for one major reason: the library, although fairly mature, is still being conceived. New functions and features are added on a regular basis. Therefore, the documentation is still very much unstable, and if I was to include it here, it would become obsolete several months after the publication of this manuscript. Because it currently consists of more than a hundred pages, I realized this would be a waste of time and resources.

If you are interested, you can of course read the current, updated and full[12] documentation online. It is available either in a web-oriented format[13] or as a compiled PDF document.[14] I give in Fig. A.1 a screenshot and description of the web interface.

[12] Actually, at the time of writing this sentence, only half of the functions are documented.

[13] http://cschreib.github.io/phypp/doc/category_support_01_intro.html.

[14] http://github.com/cschreib/phypp/raw/master/doc/latex/phypp.pdf.

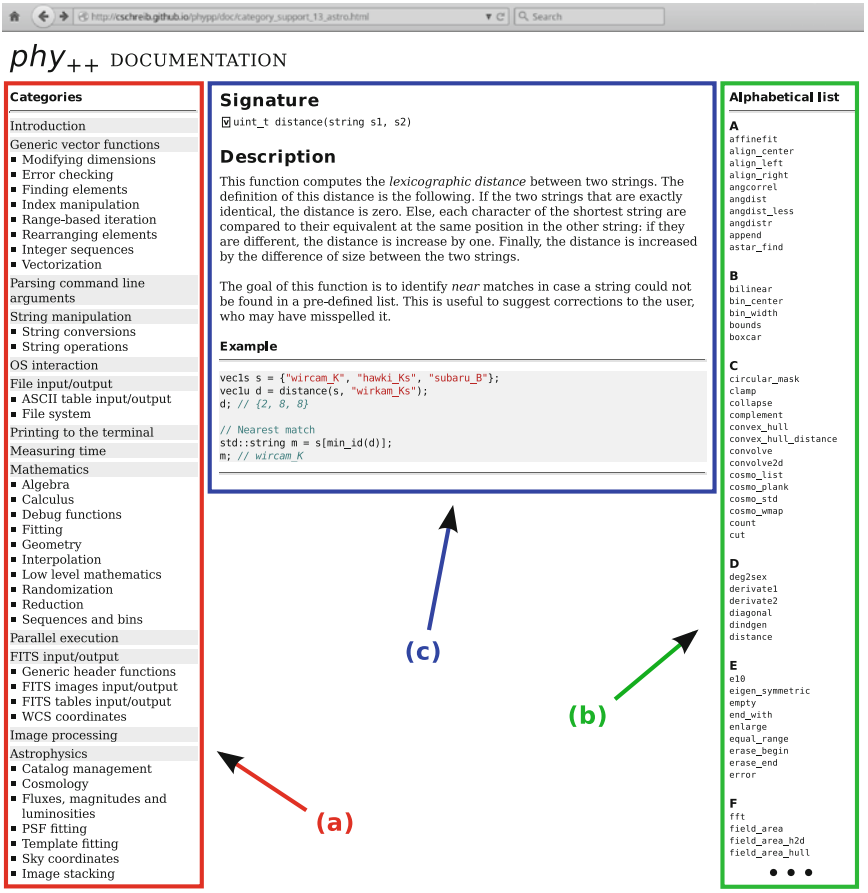

Fig. A.1 Example web page in the online documentation of the phy_{++} library (http://cschreib.github.io/phypp/doc/category_support_01_intro.html). Three main areas are highlighted on this screenshot: **a** the *category* menu, where the functions of the library are grouped by themes and sub-themes to ease the discovery of new functions; **b** the *alphabetical* menu, which lists all the functions of the library by alphabetical order to allow quick access to the documentation of a known function; and **c** the central panel where the documentation is displayed, giving the *signature* of the function (i.e., what arguments it expects), a short descriptive text, and a code sample to illustrate the usage of the function

A.2 Application: `pixfit` and `gfit`

Using the phy_{++} library, I have written most of the important codes involved in this thesis, for example the EGG tool that I introduce in Chap. 4. In this section I describe two other codes that I have written at the end of my PhD.

Most of the galaxies that we detected with ALMA (see Chap. 6) should be relatively bright in the *Herschel* SPIRE images. However, because of the poor angular resolution, interpreting these images is challenging. To obtain more precise flux estimations, I developed two programs, `pixfit` and `gfit`. These are still in the process of being tested, and I did not have time to reach a stable solution at the time of writing this manuscript. Still, I hope to be able to publish the codes in the near future. In the following, I give a brief description of the philosophy behind this novel approach, and postpone a more detailed assessment of the performances and robustness to a future work.

Conventional tools used to extract FIR fluxes (like FASTPHOT, Béthermin et al., 2010) perform point-source fitting at various pre-determined positions of the image simultaneously using linear algebra, assuming that the noise of the image is Gaussian. If there is no strong overlap between two extracted objects (or, alternatively, if the positions of the emitting sources are known perfectly), the resulting fluxes and error estimates have been shown to be reliable (see, e.g., Wang et al. in prep.). However, extracting fluxes in the highly confused SPIRE maps remains a challenge, since most objects are blended. In Wang et al. (in prep.), the situation is improved by bringing additional prior information on the expected fluxes of the faintest galaxies, but this comes at a price: the output flux catalog becomes model dependent. Even then, the number of SPIRE 500 μm sources extracted in a typical *Herschel* deep field does not exceed a hundred, compared to the thousands of MIPS 24 μm detections that we know are contributing, to some extent, to the observed 500 μm emission.

The approach that I chose with these new tools is to think of the flux catalog as only an intermediate product in the chain of data analysis: what we have in input is an observed map, and what we want in output is a catalog of SFR, $L_{\rm IR}$, or $M_{\rm dust}$. In fact, the flux catalog is only a translation of the observed map into a format that is easier to manage, but the issue is that this translation, as I argue above, is not unique. In most cases, we do not know what fraction of a given 500 μm flux should be attributed to this or that galaxy, and building a flux catalog requires making assumptions (e.g., "the brightest galaxy at 24 μm will be the brightest at 500 μm").

However, if we give up on the idea of building a conventional flux catalog, where each galaxy has either its own flux or no flux at all, one can get rid of these assumptions. For example, the idea behind `pixfit` and `gfit` is the following: for galaxies that are too close to one another on a given image (e.g., the SPIRE 500 μm map), I give up measuring their individual fluxes, and combine them into a single "flux group", for which I can measure the total flux accurately (e.g., with aperture photometry after subtracting the neighboring sources). In this case, "too close" can be defined arbitrarily, for example by choosing a given fraction of the width of the PSF,

or a fraction of pixels on the rasterized image.[15] The measured flux is then stored into a separate list, and each galaxy that belongs to the group is linked to this measurement. This first task of extracting the fluxes and making the flux groups is performed by `pixfit` on each FIR image independently. In particular, this means that two galaxies can be grouped in one image, where the angular resolution is poor, but not in another, where the resolution is sharper. This is made in a fully automatic way, by just specifying in input a list of prior positions, and defining the distance threshold below which two sources must be grouped. An example is show in Fig. A.2.

The output of this procedure is very similar to a conventional flux catalog, since each galaxy *can* have its individual flux extracted from each image, provided that it was not grouped with any other galaxy. If this is not the case, then for each band there is an additional column that indicates the ID of the flux group that contains the flux of this galaxy, and a second catalog is built to store these flux groups. It contains four columns: the group ID, a reference to the image this group was extracted from, the extracted flux and the associated uncertainty.

The next step is to properly interpret this data. Standard SED fitting codes assume that one has access to individual flux measurements in all bands, and these codes do not know how to deal with the flux groups I introduced above. Some particular codes can be given upper limits in case of a non-detection, but treating these in a statistically correct way is not trivial, and requires non-linear fitting algorithms. Indeed, while the likelihood associated to a measurement is a Gaussian, that associated to an upper limit is an error function. Therefore, the contribution of an upper limit to the χ^2 is:

$$\chi^2 = -\frac{1}{2}\log\left[\frac{1}{2}\left(1+\mathrm{erf}\left(\frac{limit-model}{error\sqrt{2}}\right)\right)\right], \tag{A.1}$$

where *limit* is the estimated upper limit, *model* is the attempt at modeling the corresponding flux, and *error* is the uncertainty on the upper limit.[16] If a galaxy is grouped in an image, the flux of the corresponding group can be used as an upper limit. As written at the beginning of this section, not only is this suboptimal, but this approach is also incorrect since each galaxy will be fitted independently. Indeed, while the upper limit will ensure that no individual model goes above the flux present on the map, there is no constrain on the *sum* of all the model fluxes: if the measured flux on

[15] Actually a similar approach is used in the extraction code of (Magnelli et al., 2009), where sources that are distant by less than a pixel are not fitted individually. The main difference with the approach I introduce here is that only one of their galaxies is kept in the prior list and arbitrarily "wins" all the observed flux.

[16] This expression is numerically unstable for large deviations above the upper limit. Setting $d \equiv (limit - model)/error$, then for $d < -3$, this formula can be approximated with good accuracy by $d^2 + 2\,\log(-2\,d\sqrt{\pi/2.0})$. Note the similarity with the regular formula for a Gaussian weight, which is just d^2.

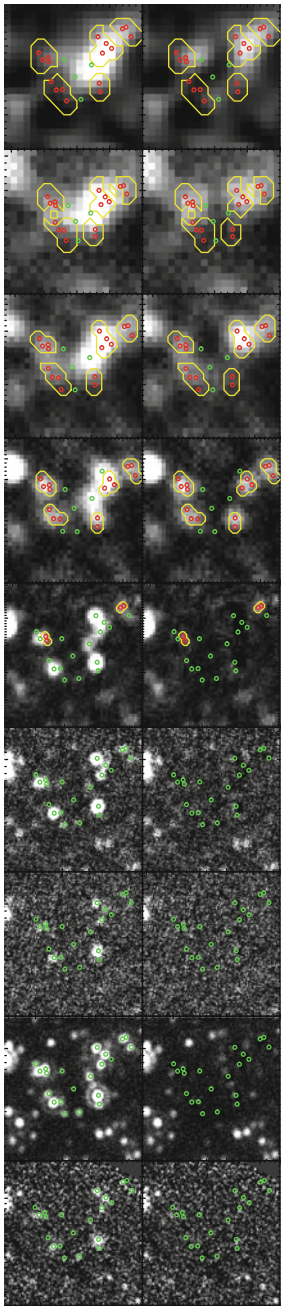

Fig. A.2 Example application of `pixfit` in GOODS–*South*. In the top row are the observed images. From *left to right*: *Spitzer* 16 and 24 μm, *Herschel* PACS 70, 100 and 160 μm, *Herschel* SPIRE 250, 350 and 500 μm, and LABOCA 870 μm. Each postage stamp covers the same region of the sky. The *bottom row* are the same image after subtracting the galaxies that have individual flux measurements, leaving only the fluxes of the groups. Each *open circle*, whether *green* or *red*, is a prior position used to extract the fluxes. *Green circles* are galaxies that have an individual flux measurement, while *red circles* show galaxies that where grouped with their neighbors for being too closely packed. A *yellow contour* indicates the extent of the corresponding flux group, and the area that is used to perform aperture photometry

the map is 20 mJy, and we use this value as an upper limit for two galaxies that lie in this region, then each galaxy can reach 20 mJy individually, for a combined flux of 40 mJy that will clearly overshoot what is observed.

That is where the `gfit` tool comes in. This program understands the catalogs produced by `pixfit`, and can perform SED fitting of multiple galaxies simultaneously. In particular, if two galaxies have some of their flux grouped, the program will model these fluxes individually, sum them up, and compare the result to the measured flux of the group in the χ^2, like any regular measurement. The fit can then be made using linear algebra, and is therefore very fast.

This main feature of performing simultaneous SED fitting is a double-edged sword though. The major downside is that if I have 100 templates in my SED library (e.g., corresponding to different values of T_{dust}), finding the optimal χ^2 requires testing each and every possible combination of templates for all the galaxies in the group, and each additional galaxy increases the computation time by a factor of 100. Obviously, this means that the problem can become computationally prohibitive. To avoid this, I first sample the parameter space of the library with a coarse grid, say of only 10 templates. I locate the combination of SEDs that produces the best χ^2, and refine the grid around this region with 10 more templates. With this approach, the accuracy on the best-fit parameters is unchanged, but the complexity drops from 100^N to 2×10^N. Without a super computer, this can still be too much if the prior density is too large. In practice though, I never had to fit more than 6 galaxies simultaneously in a given group, although I have only applied this method to a handful of cases. This problem can also be tackled with more sophisticated algorithms for global minimization, but I have not investigated this path any further.

At present, both tools are written and are feature complete. I have tested them on some of our ALMA detections, trying to better constrain their SEDs. The results seemed reasonable, but these tools really have to be tested on simulated images before any output can be trusted. I will do this later, when time permits.

Below is a excerpt from the code of `pixfit`, to illustrate how the phy_{++} library looks like in a "real world" situation.

```
// [...]

// First build the masks of each group.
// We want to make sure that the same pixels are not counted twice in different
// groups, so we have to exclude the regions where two (or more) groups overlap.
for (uint_t i : range(group_cat.ra)) {
    if (group_cat.fit[i]) {
        // This is a 'group_fit'
        // We don't need to care about it any more.
        continue;
    }

    // Locate the sources that are part of this group
    vec1u id = where(old_cat.group_aper_id == group_cat.id[i]);
    phypp_check(!id.empty(), "aper group ", group_cat.id[i], " is empty...");

    // Extract just what we need from the whole map
    uint_t xmi = max(0,             floor(min(tx[id]) - group_aper_size));
    uint_t xma = min(img.dims[1]-1, ceil(max(tx[id])  + group_aper_size));
    uint_t ymi = max(0,             floor(min(ty[id]) - group_aper_size));
    uint_t yma = min(img.dims[0]-1, ceil(max(ty[id])  + group_aper_size));

    vec2i tgrp = grp_map(ymi-_-yma,xmi-_-xma);

    // Convert coordinates to the local map
    tix[id] -= xmi;
    tiy[id] -= ymi;

    // Build the aperture mask
    vec2b mask(tgrp.dims);
    for (uint_t j : id) {
        // Create the aperture for this source
        vec2b taper = translate(aper, tdy[j], tdx[j]) > 0.5;
        vec1u idi, idp;
        subregion(
            mask,
            {tiy[j]-hsize, tix[j]-hsize, tiy[j]+hsize, tix[j]+hsize},
            idi, idp
            );

            // Add it to the mask
            mask[idi] = mask[idi] || taper[idp];
        }

        // Flag pixels that already belong to another group
        vec1u ido = where(tgrp != 0 && mask);
        tgrp[ido] = -1;
        // Remove these pixels from the mask
        mask[ido] = false;
        if (count(mask) == 0) {
            warning("group ", group_cat.id[i], " has empty mask");
        }

        // Set pixels of this group
        tgrp[where(mask)] = group_cat.id[i];
        // Save back in the whole map
        grp_map(ymi-_-yma,xmi-_-xma) = tgrp;
    }

    // [...]
```

References

M. Béthermin, H. Dole, A. Beelen, H. Aussel, A&A **512**, 78 (2010)
B. Magnelli, D. Elbaz, R.R. Chary et al., A&A **496**, 57 (2009)

Index

C. Schreiber, *A Statistical and Multi-wavelength Study of Star Formation in Galaxies*, Springer Theses, DOI 10.1007/978-3-319-44293-8