Particle filters are about 25 years old. Initially confined to the so-called “filtering problem” (the sequential analysis of state-space models), they are now routinely applied to a large variety of sequential and non-sequential tasks and have evolved to the broader Sequential Monte Carlo (SMC) framework. This greater applicability comes at the price of a greater technicality. To make matters worse, literature on particle filters spans several scientific fields, mainly engineering (in particular signal processing), but also statistics, machine learning, probability theory, operations research, physics and econometrics. Of course, each field uses slightly different notations and terms to describe the same algorithms. As a result, tracking this literature has become a challenge for non-experts.

It thus seems timely to write an introduction to particle filters for a large audience, which would cover their potential uses (and misuses), the underlying theory, and even their computer implementation.

This book is our attempt at writing such an introduction. It relies on several deliberate choices on our parts, which we hope the reader will appreciate:

• We describe the different SMC algorithms found in the literature as special cases of one generic algorithm. What really varies from one case to the next is the underlying Feynman-Kac model.
• We do not take the view that Feynman-Kac models are scary beasts that only Probabilists should look into the eyes. Properly tamed, they are simply the proper way to describe the distributions a particle filter approximates. We spend some time in this book to explore the properties of these models. The big advantage of doing so is that many practical algorithms may be derived directly from such properties.
• In the same spirit, and although we tried to keep mathematical sophistication to a low level, we did not try to avoid at all prices measure-theoretic concepts such as Radon-Nikodym derivatives and Markov kernels. Several important practical problems feature Markov processes \( \{X_t\} \) such that \( X_t \) given \( X_{t-1} \) does not admit a density, and trying to explain how the corresponding algorithms work (or do not...
work!) in this case becomes very confusing if everything is expressed in terms of “densities”. For readers less familiar with these concepts, we provide a gentle introduction to these notions.

Speaking of mathematical prerequisites, this book is, we hope, accessible to any reader who had a previous exposure to non-trivial Probability; some familiarity with probability spaces, $\sigma$-algebras, and related concepts should help but is not essential.

1.1 Relation to Other Books

As said above, our objective is to provide a general introduction to all the facets of SMC: the underlying theory, the actual algorithms, their complexities, their limitations, how to implement them in practice, and the wide range of their applications.

We hope that this book may also serve as an introduction to books that go deeper into the theory, such as the books of Del Moral (2004) and Cappé et al. (2005).

Regarding the former, we recommend in particular Chaps. 2–5 and Chap. 8, which are particularly relevant for the theoretical study of SMC algorithms. Regarding the latter, we recommend in particular Part II, as it covers a topic which we only allude to in this book: the asymptotic theory of parametric estimation for state-space models.

1.2 Structure of the Book

Most chapters have the following plan: a short abstract, followed by the core of the chapter than contains the main methods and theory, a section on numerical experiments, a set of exercises (for self-study, or for covering more advanced points), a bibliography, and a “Python corner” that discusses the implementation of the considered methods in Python; see the end of this chapter for more information on the Python corners.

The first half of the book covers the prerequisites for describing particle algorithms at a certain level of generality:

- Chapters 2 and 3 give a general, non-technical overview of the applications of particle filters: Chap. 2 focuses on the sequential analysis of state-space models, while Chap. 3 describes non-sequential problems that may be tackled with these algorithms.
- Chapters 4 and 5 describe the mathematical tools and concepts that underlie particle algorithms; e.g. Markov processes, Feynman-Kac models, the related forward-backward recursions, and so on. These two chapters may require more efforts from the readers than the rest, but the upside is that many ideas and
algorithms found in subsequent chapters become obvious when seen as an application of such tools.

- Chapters 6 and 7 cover respectively the forward-backward algorithm (for finite state-space models), and the Kalman filter (for linear-Gaussian state-space models). These two algorithms correspond to the two most common cases where the Feynman-Kac distributions may be computed exactly (without resorting to particle filtering). These two chapters may be skipped at first reading.
- Chapters 8 and 9 give prerequisites in Monte Carlo sampling, particularly importance sampling (Chap. 8) and resampling (Chap. 9).

The core of the book consists of Chaps. 10 (on particle filtering, and its application to state-space models), 12 (on particle smoothing), and 13 (quasi-Monte Carlo particle filter). The rest of the book covers various advanced topics that the readers may be read selectively based on their interests:

- Chapter 11 gives an introduction to the asymptotic theory of particle filters.
- Chapters 14 and 16 cover respectively maximum likelihood and Bayesian estimation of state-space models. Since the latter relies on MCMC (Markov chain Monte Carlo) algorithms, Chap. 15 contains a brief introduction on MCMC.
- Chapter 17 covers SMC samplers, that is, SMC algorithms that may be applied to a general sequence of distributions.
- Chapter 18 describes SMC$^2$, an advanced SMC sampler for the sequential estimation of state-space models.

### 1.3 Note to Instructors

This book grew out of courses on particle filtering taught at different places, at different levels, and for different durations. Slides may be requested from the authors. An example of such a course (in a very short, two-hour format, at PhD level) may be viewed at: [https://library.cirm-math.fr/Record.htm?idlist=4&record=19285595124910037779](https://library.cirm-math.fr/Record.htm?idlist=4&record=19285595124910037779). The more common format for the first author is a 18-h course taught at a graduate (M2) level at the ENSAE.

We make the following recommendations to potential instructors. First, it is important to spend at least 1 h on the various applications covered in Chaps. 2 and 3. This part is very accessible, and should give the students a good idea of what the course is about and what SMC methods are capable of.

The next step is to cover Chaps. 4 and 5, on Markov processes and Feynman-Kac models respectively. Students may struggle with the notations, so it is important to explain them very carefully, using e.g. simple illustrative examples (where the density of the Markov kernel exists or does not exist, and so on). You may want to skip the second part of Chap. 7 (on backward decompositions) in order to spend more time on the simpler forward recursion.

We typically skip Kalman filtering (Chap. 7) and cover briefly the finite case (Chap. 6), mostly as a way to illustrate the ideas of the previous chapter.
Then, one should cover importance sampling and resampling (Chaps. 8 and 9). This can go quite fast if students have already some background on Monte Carlo.

Once all these notions have been properly introduced, covering particle filtering (Chap. 10) is surprisingly straightforward, and may be done in about 1 h.

At this stage, it is very helpful to discuss with the students how the different particle algorithms may be implemented in practice. We typically spend at least one hour going through the material covered in the Python corners of up to Chap. 10. This part really helps students to get a more concrete understanding of the underlying concepts.

The rest of the course should depend on your preferences and time constraints. Going through all the different approaches for smoothing (Chap. 12) is very tedious, so you may want to skip this chapter entirely, or cover only one algorithm, e.g., FFBS. On the other hand, parameter estimation for state-space models (Chaps. 14 and 16) is a topic which is simple to review, and which students enjoy quite a lot. For the latter chapter, it is nice to focus on PMMH as students find this algorithm particularly intuitive.

Certain years, we manage to cram one more chapter, and go for SMC samplers, as this chapter may be the basis for several interesting projects.

Speaking of projects, we typically provide the students with a recent research paper, ask them to implement one or several SMC samplers (e.g. the one described in the paper plus another one they have to figure out given some indications), and discuss their results. They are allowed (and even encouraged) to use particles, the Python package we developed to go with this book (see below).

1.4 Acknowledgements

As said above, this book is partly based on material developed over the years while teaching particle filters in various places (Université Paris Dauphine, ENSAE Paris, University of Copenhagen, a summer school at Warwick University, and another one at CIRM in Marseille). We are very grateful to our colleagues who kindly invited us to teach this material (Susanne Ditlevsen, Joaquín Miguez, Murray Pollock and Christian P. Robert) and to students who (enthusiastically, or sometimes stoically) attended the courses, and helped us to improve the presentation.

We are also most grateful to all our colleagues who kindly accepted to read and provide feedback on a preliminary version of the book (in alphabetical order): Adrien Corenflos, Hai-Dang Dau, Arnaud Doucet, Gabriel Ducrocq, Pierre Jacob, and Jonty Rougier. Special thanks to Sergios Agapiou, Mathieu Gerber, Umberto Picchini, and Daniel Sanz-Alonso for their detailed comments.

The first author would like to dedicate this book to his family, in particular his father who passed recently, his mother, his wife and his two particles Alice and Antoine. The second would like to acknowledge the inspiration from another tamed FK beast, who has also actively participated in some of the training activities mentioned above, the Female Kanine (sic) Xiska.
The book received its final revision and editing during the SARS-Cov-2 lockdown: ούδεν κακόν αμιγες καλού!

1.5 Python Corner

While writing this book, we developed a Python package, called particles, which implements in a modular way a large class of particle algorithms. This package is available on GitHub (at https://github.com/nchopin/particles); see below for installation instructions.

Each chapter of this book ends up with a Python corner like this one, where we discuss how to implement in Python the methods covered in that chapter, in particular by considering excerpts from particles.

The rest of this Python corner is a very brief introduction to Python and its scientific libraries. It is not a proper tutorial on Python and its numerical libraries, such as e.g. the Scipy lectures (http://www.scipy-lectures.org), the on-line courses at the software carpentry (https://software-carpentry.org/), or the book of McKinney (2012) (which is also a good introduction to some of the scientific libraries discussed in the next section). Rather, it is meant to explain a few quirks of the Python language that, sometimes, confuse beginners. Fortunately, Python has been designed to be highly readable, so the following should be enough to make beginners able to make most of the Python corners.

Other advantages of Python is that it is free and open source (unlike Matlab), and that it is becoming the most popular language in machine learning and in Engineering (to the detriment of Matlab). In Statistics, R remains more popular, in particular because it features such a large number of statistical libraries. Note however that it is easy to call any R library from within Python, using the rpy2 library.

1.5.1 Syntax

The following piece of Python code defines a function for computing factorials:

```python
def factorial(n, verbose=False):
    "Compute factorial of n (the product of 1, 2, ... to n)."
    p = 1
    for k in range(2, n + 1):
        p = p * k
        if verbose:
            print("factorial of %i is %i" % (k, p))
    return p

print(factorial(5))
```
These few lines reveal some noteworthy aspects of Python:

• Python uses whitespace indentation to delimit blocks (rather than e.g. curly braces). This is a great way to enforce readability. Common practice is to use four spaces per indentation level. It is also possible to use one tab to define an indentation level. Be aware of however that the number of spaces used to represent one tab is arbitrary. As a result, Python is not able to process text files that mix tabs and spaces. The best way to avoid ‘tab headaches’ is to make the following changes in the settings of your favourite text editor: (a) tab width is four spaces; (b) automatically translate tabs into spaces. That way, when you press tab once (resp. backspace), you automatically get to the next (resp. previous) level, without either entering tabs in your document. (These settings are actually sensible for any language: clear indentation always makes code more readable, but this cannot be achieved using tabs, given again their ambiguous meaning.)

• Python is dynamically typed: the type of the arguments of factorial are not specified.

• Named arguments are optional: if verbose is not specified, it is automatically set to False; to set to true, the proper syntax is \texttt{x = factorial(10, verbose=True)}.

• The string below the \texttt{def} statement defines the documentation of function factorial. Once the code above is executed, factorial becomes an object, and \texttt{help(factorial)} returns its documentation.

• Python uses 0-based indexing (like C). For instance, \texttt{range(n)} iterates over integers \(1, \ldots, n - 1\). In the code block above, \texttt{range(2, n+1)} is the half-open interval \(2, \ldots, n\).

### 1.5.2 Scientific Computation in Python

Here are the main scientific libraries in Python:

• \texttt{NumPy} implements ‘ndarrays’ (multidimensional numerical arrays), and functions that operate on them. It also includes routines for linear algebra, Fourier analysis and random number generation. Most scientific libraries are built upon \texttt{NumPy}.

• \texttt{SciPy} is a comprehensive package for scientific computing, with modules for linear algebra, optimization, integration, special functions, signal and image processing, statistics, ODE solvers, among other things.

• \texttt{Matplotlib} is a 2D plotting library with a syntax similar to Matlab.

• \texttt{seaborn} complements \texttt{Matplotlib} with statistical plots, in a spirit somehow similar to the R library \texttt{ggplot}.

• \texttt{pandas} implements data frames (à la R) and various tools for data analysis.

• \texttt{rpy2} makes it possible to call R functions and libraries within Python.
• **Numba** is an experimental library that speeds up the execution of Python; see Performance below.
• **scikit-learn** is a popular machine learning library.

All these libraries and many others are available in any good scientific Python distribution; see below under ‘Installation’. The only strict dependencies for particles are NumPy and SciPy; Matplotlib and seaborn are optional dependencies, as they are required only to run certain examples.

The code blocks provided in the Python corners assume that the following ‘import’ have been performed:

```python
import numpy as np
```

The first line makes all NumPy functions and objects available with the syntax np.func. (If we had typed `import numpy`, then they would be available as `numpy.func`.) For instance:

```python
x = np.arange(100)
y = np.exp(x[:10])
```

creates two 1D arrays, `x` containing 0, 1, ..., 99, and `y` containing \(e^0, \ldots, e^9\); recall that Python uses 0-based indexing, hence `x[:10]` is an array containing `x[0]` to `x[9]`. The second line specifically calls the NumPy version of `exp`, which accepts ndarrays as inputs and returns ndarrays as outputs.

It is of course a bit annoying (especially if you are coming from Matlab or R) to type `np.` in front of all standard functions, e.g. `log`, `sin`, and so on. One may be tempted to use instead an implicit import:

```python
from numpy import *
```

```
x = arange(100)
y = exp(x[:10])
```

so that all NumPy objects and functions are available into the base namespace. This is fine for interactive work, but in general it is better to be explicit about where does any object comes from; importing an awful lot of functions and objects into the namespace is typically called namespace pollution and it is not very Pythonic! It leads to code that it is hard to follow. In particular, implicitly importing two libraries with overlapping functionalities (such as NumPy and SciPy) is a good way to write obscure and buggy code.

### 1.5.3 Random Number Generation

NumPy has a module called `random`, which contains procedures to sample from many probability distributions. To generate an array containing 100 draws from the \(\mathcal{N}(2, 3^2)\) distribution:

```python
from numpy import random
```

```
x = random.normal(loc=2., scale=3., size=100)
```
SciPy has a module called `stats`, which represents probability distributions as objects, from which may sample from, compute the probability density, and so on.

```python
from scipy import stats

law = stats.norm(loc=2., scale=3.)
x = law.rvs(size=100)
y = law.logpdf(x)  # log of probability density function at x
z = law.cdf(x)  # cumulative distribution function at x
```

The second approach is clearly more versatile. `particles` implements probability distributions in a similar way; see the documentation of module `distributions` for more details.

### 1.5.4 Performance

Let’s return to our factorial example; here is a more concise and more efficient version based on NumPy arrays:

```python
def factorial(n):
    return np.prod(np.arange(2, n+1))
```

Conventional wisdom is that loops are very slow in interpreted languages (such as Python, Matlab or R), and that they should be ‘vectorized’, i.e. replaced by array operations. This is true to some extent, and it certainly applies to our factorial example. But it is more accurate to say that loops with a slim body are slow. Loops with a body that does expensive operations are typically fine, and should be left alone. Another point to take into account is that ‘vectorizing’ a piece of code often makes it less readable, sometimes very much less so. For instance, try to figure out what result you get if you type `factorial(-3)`, by looking at both versions of `factorial`.

We will discuss this point more in detail when practical examples come up in the following chapters. For now, we mention that loops may also be sped-up using Numba, which makes it possible to compile on the fly selected parts of a program; see Chap. 9 for an illustrative example.

### 1.5.5 Installing Python

A simple way to install in one go Python and its main scientific libraries is to install the Anaconda distribution from Continuum Analytics, which is available on all major platforms; see [https://www.continuum.io](https://www.continuum.io). Two versions are proposed: 2.7 and 3.7 at the time of writing. We recommend the latter. Python 2 will soon be
1.5 Python Corner

deprecated, and most libraries are now compatible with Python 3. particles is currently compatible with both, but we may drop support for Python 2 in the future.

The Anaconda distribution comes with all the libraries mentioned above (and many more), and also contains the following useful tools:

- ipython: a replacement for the standard Python shell, which is very popular among scientists. It also has a notebook mode, which makes it possible to mix live code, equations and figures in a browser window. Very handy for teaching. See also jupyter.
- spyder: a development environment in the spirit of Rstudio or the Matlab interface.
- pip: a utility to install packages available in the huge repository pypi (the Python package index, see https://pypi.python.org).

### 1.5.6 Installing particles

You may download particles from https://github.com/nchopin/particles, (manually or by cloning the repository, if you are familiar with the version control system git).

The package is organised as follows: the scripts that were used to perform the numerical experiments discussed in this book (and to generate the plots) are in folder book; the various datasets used in these experiments are in datasets; the documentation is in docs, although it is more convenient to consult it directly online, at https://particles-sequential-monte-carlo-in-python.readthedocs.io/en/latest/.

The package itself is made of modules which may be found in folder particles. To use these modules, you must first install the package, using for instance pip; i.e. on the command line (assuming you are already inside folder particles):

```
    pip install --user .
```

Once the package is installed, you should be able to import any module from particles; e.g.

```
    from particles import resampling as rs
```

See page https://github.com/nchopin/particles, for more details on the installation.

### 1.5.7 Other Software

particles is meant to be easy to use and extend. For raw performance, we particularly recommend LibBi (http://libbi.org, see also its successor at https://
birch-lang.org/) which takes as an input a description of the considered model, and implements in C++ the corresponding particle algorithms, using multi-core CPUs or even GPUs.

Other noteworthy pieces of software are SMCTC, a C++ template class library, see Johansen (2009); pomp, a R package that focuses on parameter estimation of state-space models, see https://kingaa.github.io/pomp/; and SequentialMonteCarlo.jl, a Julia package developed by Antony Lee, see https://github.com/awllee.

**Bibliography**

Cappé, O., Moulines, E., & Rydén, T. (2005). *Inference in hidden Markov models*. Springer series in statistics. New York: Springer.

Del Moral, P. (2004). *Feynman-Kac formulae. Genealogical and interacting particle systems with applications*. Probability and its applications. New York: Springer.

Johansen, A. M. (2009). SMCTC: Sequential Monte Carlo in C++. *Journal of Statistical Software*, 30(6), 1–41.

McKinney, W. (2012). *Python for data analysis*. Sebastopol, CA: O’Reilly Media.