České vysoké učení technické v Praze
Fakulta jaderná a fyzikálně inženýrská

Katedra matematiky
Obor: Inženýrská Informatika
Zaměření: Softwarové inženýrství a matematická informatika

Výpočetní prostředí pro asimilaci disperzních atmosférických modelů

Environment for Assimilation of Atmospheric Dispersion Models

DIPLOMOVÁ PRÁCE

Vypracoval: Matěj Laitl
Vedoucí práce: Ing. Václav Šmídl, Ph.D.
Rok: 2014

Před svázáním místo téhle stránky vložíte zadání práce s podpisem děkana!!!!

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškerou použitou literaturu.

V Praze dne .....

.....
Matěj Laitl

Poděkování

Děkuji Ing. Václavu Šmídlovi, Ph.D. za vedení mé práce, cenné konzultace a houževnatost, s níž mne motivoval posunovat práci kupředu. Dále děkuji doc. Ing. Jiřímu Fürstovi, Ph.D., jehož přednáška Numerický software mě inspirovala k vytvoření knihovny Ceygen.

Matěj Laitl

Název práce:
Výpočetní prostředí pro asimilaci disperzních atmosférických modelů

Autor: Matěj Laitl

Obor: Inženýrská Informatika
Druh práce: Diplomová práce

Vedoucí práce: Ing. Václav Šmídl, Ph.D.
Oddělení adaptivních systémů
Ústav teorie informace a automatizace
Akademie věd České republiky

Abstrakt: Tato práce je zaměřena na návrh a implementaci softwarového řešení pro asimilaci atmosferických disperzních modelů, které simulují únik radioaktivních látek. Důraz je kladen na výpočetní efektivitu řešení. K asimilaci je přistupováno pomocí sekvenčního Monte Carlo vzorkování, které je v textu speciálně upraveno pro daný problém. Softwarová analýza vyústí ve vytvoření projektu Asim, který implementuje asimilaci, vylepšení knihovny pro Bayesovskou filtraci PyBayes a vytvoření knihovny pro lineární algebru Ceygen, která je použita ke zvýšení výkonu výpočtů v použitém implementačním prostředí — Pythonu, který je kompilován projektem Cython. Je ukázáno, že Ceygen přináší 2- až 10-násobný nárůst výkonu algebraických metod pro určité velikosti vstupů. Nakonec je ověřena správná funkčnost celého asimilačního řešení pomocí dvojného experimentu.

Klíčová slova: asimilace, disperzní model, sekvenční Monte Carlo, radioaktivní únik, softwarová analýza, numerické výpočty, Python, Cython

Title:
Environment for Assimilation of Atmospheric Dispersion Models

Author: Matěj Laitl

Abstract: This text is devoted to designing and implementing software solution for assimilation of atmospheric dispersion models that simulate radioactive pollutant release, with accent on computational efficiency. The assimilation is approached using sequential Monte Carlo methods, which are tailored to this specific problem in the text. Software analysis results in creation of the Asim project to perform the assimilation, improvements to the PyBayes Bayesian filtering library and creation of the Ceygen linear algebra library that is used to accelerate computations in the implementation environment of choice, Cython-compiled Python. Ceygen is later shown to provide 2× to 10× performance increase of algebraic methods for certain problem sizes. Functionality of the assimilation solution is successfully verified using a twin experiment.

Key words: assimilation, dispersion model, sequential Monte Carlo, radioactive release, software analysis, numerical computing, Python, Cython

Contents

Notation

Following notation is used throughout this text:

$\mathbb{N}$	set of natural numbers excluding zero
$\mathbb{N}_0$	set of natural numbers including zero
$\mathbb{R}$	set of real numbers
$\mathbb{R}_0^+$	set of non-negative real numbers
t	discrete time moment; $t \in \mathbb{N}_0$
a_t	value of quantity a at time t ; $a_t \in \mathbb{R}^n, n \in \mathbb{N}$
$\mathbf{s}$	vector quantities are emphasized where it improves readability
$a_{t:t'}$	sequence of quantities $(a_t, a_{t+1}, \ldots, a_{t'-1}, a_{t'})$
$p(a_t)$	probability density function of quantity a at time t; for the purpose of this text, probability density function p is multivariate non-negative function $\mathbb{R}^n \to \mathbb{R}$; $\int_{\operatorname{supp} p} p(x_1, x_2, \ldots, x_n)\, \mathrm{d}x_1 \mathrm{d}x_2 \cdots \mathrm{d}x_n = 1$
$p(a_t \vert b_{t'})$	conditional probability density function of quantity a at time t given value of quantity b at time t'
$\delta(a)$	Dirac delta function; used exclusively in context of probability density functions to denote discrete distribution within framework of continuous distributions, so that $\int_{-\infty}^{\infty} f(x)\delta(x-\mu)\, \mathrm{d}x = f(\mu)$ and more complex expressions can be derived using integral linearity and Fubini's theorem
$\mathcal{N}(\mu, \Sigma)$	multivariate normal (Gaussian) probability density function with mean vector μ and covariance matrix Σ
$\mathcal{N}(x; \mu, \Sigma)$	random variable (x) is explicitly mentioned along with distribution parameters (μ, Σ) where it is not clear from the context
$t\mathcal{N}\left(\mu, \sigma^2, \langle a, b\rangle\right)$	normal probability density function with original mean value μ and variance σ^2 truncated to interval $\langle a, b\rangle$
$\mathcal{G}(k, \theta)$	gamma probability density function with shape parameter k and scale parameter θ
$i\mathcal{G}(\alpha, \beta)$	inverse gamma probability density function with shape parameter α and scale parameter β

Introduction

Assimilation of dispersion models using sequential Monte Carlo techniques has recently become attractive thanks to methods that improve its efficiency. Monte Carlo, when approached as a form of Bayesian filtering, allows to combine diverse measurement sources naturally to produce informed estimation of a hidden state. This text focuses mainly on design and development of a clean software solution implementing its algorithms in the field of atmospheric assimilation. Special emphasis is taken on computational efficiency.

Chapter 1 presents a general mathematical apparatus that our solution is based on (the recursive Bayesian estimation) along with more specific method, particle filtering, that can be viewed as an application of the framework. Their inclusion in this text serves two purposes: completeness and forming a basis for the software design presented later, which happily borrows concepts from the underlying mathematics.

Chapter 2 describes how the simulation of the atmospheric pollutant dispersion is modeled and how it can be measured for assimilation purposes. Later it constructs the task of assimilation using the mathematical language from chapter 1. It also devotes some space to presentation of a technique employed to optimize efficiency of particle filtering.

Chapter 3 is devoted to software analysis. After discussion of requirements posed on desired software solution, it summarizes available projects and frameworks. It then offers conclusions from the analysis in form of suggestions for layout of the target software stack and interactions between its components.

Chapter 4 presents the design and implementation of these components in greater detail. It starts by description of the Ceygen library which was conceived to provide highly efficient linear algebra operations for the latter projects; continues by presentation of enhancements done to PyBayes to support the umbrella project Asim, which was developed to combine Bayesian filtering from PyBayes with dispersion models to form the desired dispersion model assimilation solution.

Chapter 5 verifies functionality of the software stack. It presents results from

benchmarks of the Ceygen library to confirm that it indeed brings significant overhead reduction compared to previous solutions. Second part of this chapter is devoted to a twin experiment, where a simulation of pollutant release under realistic meteorologic conditions is performed, noise is added to simulated measurements, assimilation is run with the noisy observations and its results are compared against the original simulation.

Chapter 1

Sequential Monte Carlo Methods

We are concerned with assimilation of radioactive pollutants as they disperse in the atmosphere; that is to give the best estimate of the pollutant distribution in the environment and its progress in time, given a set of measurements with varying degree of accuracy. The problem is approached using the mathematical framework of recursive Bayesian estimation, which is briefly presented in the first section. This general problem statement and solution is then followed by description of a so-called Particle Filter,[1] a variant of which is employed to perform the assimilation. We also briefly discuss the Marginal Particle Filter, a derivate which is designed to counter some inherent problems of Bayesian particle filtering.

1.1 Recursive Bayesian Estimation

Recursive Bayesian estimation[2] yields an estimate (in form of probability density functions) of a sequence of hidden state vectors $x_{1:t}$ at discrete time steps $1:t$ given sequence of available measurements (observations) $y_{1:t}$, known (or assumed) noisy state model (1.1) and known (or assumed) noisy observation model (1.2). State model (1.1) describes how state evolves in time; more precisely it prescribes that state x at time t, x_t, depends on previous state x_{t-1} and random state noise v_{t-1} (whose statistical properties need to be known) through the means of function f_t, which often does not vary with time and is therefore labeled just f. Similarly, measurement model (1.2) prescribes that visible measurements y at time t, y_t, depend via function h_t (often just time-independent h) on hidden state x_t and random observation noise w_t, probability density function of which is likewise known. Summarizing

[1] also known as the Sequential Importance resampling (SIR) or Sequential Monte Carlo method.
[2] also known as Bayesian filtering.

this, we have

$$x_t = f_t(x_{t-1}, v_{t-1}) \tag{1.1}$$
$$y_t = h_t(x_t, w_t). \tag{1.2}$$

The Bayesian solution can now be derived. Both x_t ($\forall t$) and y_r ($\forall t$) are treated as (multivariate) random variables from now on. First, let us observe that the *state probability density function* $p(x_t|x_{t-1})$ can be derived from (1.1) and the *observation probability density function* $p(y_t|x_t)$ can be derived from (1.2) likewise. So-called *prior* probability density function $p(x_t|y_{1:t-1})$ can be expressed using only known densities by applying reverse marginalization over x_{t-1}, applying chain rule for probability density functions and finally taking into account that x_t does not depend on y_{t-1} when already conditioned on x_{t-1} (i.e. we assume Markov property)

$$\begin{aligned} p(x_t|y_{1:t-1}) &= \int p(x_t, x_{t-1}|y_{1:t-1})\ \mathrm{d}x_{t-1} \\ p(x_t|y_{1:t-1}) &= \int p(x_t|x_{t-1}, y_{1:t-1}) p(x_{t-1}|y_{1:t-1})\ \mathrm{d}x_{t-1} \\ p(x_t|y_{1:t-1}) &= \int p(x_t|x_{t-1}) p(x_{t-1}|y_{1:t-1})\ \mathrm{d}x_{t-1}. \end{aligned} \tag{1.3}$$

Next straightforward step in the solution is to derive *posterior* probability density function $p(x_t|y_{1:t})$, which is done by applying the Bayes' theorem and taking the fact that observation y_t depends only on the newest state x_t so that $p(y_t|x_t, y_{1:t-1}) = p(y_t|x_t)$

$$\begin{aligned} p(x_t|y_{1:t}) &= \frac{p(y_t|x_t, y_{1:t-1}) p(x_t|y_{1:t-1})}{p(y_t|y_{1:t-1})} \\ p(x_t|y_{1:t}) &= \frac{p(y_t|x_t) p(x_t|y_{1:t-1})}{p(y_t|y_{1:t-1})}, \end{aligned} \tag{1.4}$$

where the denominator of (1.4) can be computed using marginalization over x_t or not at all, because it does not depend on x_t and serves only the purpose of a normalization constant. Formulas (1.3) and (1.4) together form the solution of the problem posed above.

On the other hand, this solution in its general form has only limited application, because the integral (1.3), unless tractable analytically, constitutes a major computational challenge. The situation when the analytical solution is known is represented mainly by the famous Kalman filter[10] and its variants, which applies

when state and observation models are linear and noises are normally distributed. If these models are highly non-linear (which is our case in radioactive release assimilation), one of the viable approaches is to resort to some kind of approximation of the solution. One such approximation is a particle filter described below.

1.2 Particle Filter

The idea of the particle filter[7] is to represent the approximation of its posterior probability density function using weighted empirical probability density function

$$p(x_t|y_{1:t}) \approx \sum_{i=1}^{N} \omega_t^{(i)} \delta(x_t - x_t^{(i)}), \tag{1.5}$$

where N is the total number of particles (an important parameter of the filter), $\omega_t^{(i)}$ is the weight of the i-th particle at time t[3] and finally $x_t^{(i)}$ is the value of the i-th particle at time t, a hypothetical state the model could be in. The algorithm to recursively estimate posterior density is bootstrapped by i.i.d.[4] sampling initial particles $x_0^{(1:N)}$ from a given *initial* probability density function $p(x_0)$. When new observation y_t is available, recursive update is performed as described in Algorithm 1.1. [4]

First part of step 1. in Algorithm 1.1 is usually called *prediction* as it foresees potential value of $x_t^{(i)}$ based on previous value $x_{t-1}^{(i)}$ and, in general case, observation y_t. Second part of step 1. performs *update* of the predicted posterior probability density function through the means of adjusting weight $\omega_t^{(i)}$ based on the actual observation and its probability density function $p(y_t|x_t)$, taking into account likelihood of the prediction made by a so-called *proposal* density. — Apart from probability density functions that come directly from the Bayesian filtering (the state density $p(x_t|x_{t-1})$ and observation density $p(y_t|x_t)$), the algorithm makes use of the proposal density $q(x_t|x_{t-1}, y_t)$, which is an instrument to optimize the algorithm to a specific problem. The most straightforward choice of the proposal density (and the one originally described in [7]) is directly the transition density $p(x_t|x_{t-1})$ (which is sometimes called the *naive* proposal) that ignores the observation, which simplifies the weight calculation in (1.6) to $\omega_t^{(i)} := p(y_t|x_t)\omega_{t-1}^{(i)}$. Such algorithm is known as the *bootstrap filter*. On the other hand, discarding observation y_t, possibly the source of most valuable information, can degrade the efficiency of the algorithm in a very severe way for certain systems. Our problem of atmospheric assimilation is an example of one such system as it is prohibitively computationally demanding without the help of carefully chosen proposal density.

[3] constraint $\sum_{i=1}^{N} \omega_t^{(i)} = 1 \quad \forall t$ must hold.
[4] independently identically distributed.

Algorithm 1.1 Recursive update of the particle filter

1. for each i in $\{1, \ldots, N\}$ do:

   - draw $x_t^{(i)}$ from the proposal density $q(x_t|x_{t-1}, y_t)$, where x_{t-1} is substituted by $x_{t-1}^{(i)}$ and y_t is the observation.
   - update weight using (1.6) where x_{t-1} is substituted by $x_{t-1}^{(i)}$, x_t is substituted by $x_t^{(i)}$ and y_t is the observation,

$$\tilde{\omega}_t^{(i)} := \frac{p(y_t|x_t)p(x_t|x_{t-1})}{q(x_t|x_{t-1}, y_t)}\omega_{t-1}^{(i)}. \tag{1.6}$$

2. for each i normalize weights according to

$$\omega_t^{(i)} := \frac{\tilde{\omega}_t^{(i)}}{\sum_{j=1}^{N} \tilde{\omega}_t^{(j)}}.$$

3. resample $x_t^{(1:N)}$ from posterior probability density function 1.5 eliminating low-weight particles, but maintaining posterior statistics.

The popularity of the particle filter may come from the fact that it is distinctively general and easy to apply to a wide range of Bayesian problems, regardless of analytical tractability of integrals that occur in generic solution: the only performed operations with involved densities is sampling from the proposal density and evaluation of the proposal, state and observation densities. However, this convenience comes with a cost. First and foremost, the particle filter is an approximate and stochastic method. While convergence to the optimal Bayesian solution (1.4) as N approaches infinity is proven under a set of assumptions,[3] real-world applications must find a trade-off between accuracy (which improves as N grows as with other Monte Carlo methods) and performance (where computation time linearly depends on N).

Another peculiarity of the particle filter is a *sample impoverishment* phenomenon. It is described as a tendency of the weights of all-but-one particle to drop to zero as time progresses (unless counter-measured), i.e. the variance of the weights tends to raise without bounds. Untreated, this property effectively degenerates the posterior density to a single Dirac delta function, making the filter anything but useful. Common approach to eliminate this is the resampling described in step 3. of the Algorithm 1.1. *Number of effective particles* N_{eff} is commonly used as a measure of the sample impoverishment and can be estimated using (1.7).[4]

$$N_{\text{eff}}(t) \approx \left(\sum_{i=1}^{N} \left(\omega_t^{(i)} \right)^2 \right)^{-1}, \qquad 1 \leq N_{\text{eff}} \leq N. \tag{1.7}$$

N_{eff} close to N denotes near-uniform weight distribution while N_{eff} approaching 1 is a sign of the degenerate case. The resampling step mitigates some effects of the sample impoverishment, but if $N_{\text{eff}} \ll N$, available resources are used inefficiently, as accuracy of the estimate is only improved by the "effective" particles, the rest serves as (to some extent inevitable) computational overhead. This problem is most pronounced when the proposal density does not match the resulting posterior density sufficiently (often being too broad) a condition often observed in the bootstrap filter. In accordance with literature, our tests have shown that careful choice of the proposal density can render sample impoverishment much more subtle problem.

1.3 Marginal Particle Filter

Numerous techniques to address shortcomings of the particle filter have been described in literature, we have found it interesting to pick up the *Marginal Particle Filter* proposed in [12]. It is a modification of the particle filter where filtering is done directly on the marginal distribution $p(x_t|x_{t-1})$, more precisely on its estimate

Algorithm 1.2 Recursive update of the Marginal Particle Filter

1. for each i in $\{1, \ldots, N\}$ do:
   - draw $x_t^{(i)}$ from the modified proposal density $\sum_{i=1}^{N} \omega_{t-1}^{(i)} q(x_t|y_t, x_{t-1}^{(i)})$ using stratified sampling, where y_t is the observation.
   - update weight using

$$\tilde{\omega}_t^{(i)} := \frac{p(y_t|x_t) \sum_{i=1}^{N} \omega_{t-1}^{(i)} p(x_t|x_{t-1}^{(i)})}{\sum_{i=1}^{N} \omega_{t-1}^{(i)} q(x_t|y_t, x_{t-1}^{(i)})}.$$

2. for each i normalize weights according to

$$\omega_t^{(i)} := \frac{\tilde{\omega}_t^{(i)}}{\sum_{j=1}^{N} \tilde{\omega}_t^{(j)}}.$$

$\sum_{i=1}^{N} \omega_{t-1}^{(i)} p(x_t|x_{t-1}^{(i)})$. The authors suggest proposal probability density function in a similar form $q(x_t|y_{1:t}) = \sum_{i=1}^{N} \omega_{t-1}^{(i)} q(x_t|y_t, x_{t-1}^{(i)})$, where $q(x_t|y_t, x_{t-1}^{(i)})$ would be the proposal density normally used in the original particle filter design. Along with adapted updating step, Marginal Particle Filter is characterized in Algorithm 1.2.

The authors argue that sampling from a more complete proposal density (that takes other weighted particles into account) effectively fights the sample impoverishment problem and relieves the need to perform the resampling step. These claims are supported by comparisons of the Marginal Particle Filter against its precursor on synthetic and real-world examples that indeed confirm improvement in *unique particle count*[5] and variance of the weights. Additionally it is shown (in the case of Auxiliary Marginal Particle Filter, an analogous extension of the auxiliary variable particle filter) that variance of the weights can be only less or equal than in non-marginal variant. It should be noted that when *naive* proposal is used, Marginal Particle Filter is equivalent to the traditional particle filter described above.

On the other hand, the presented method comes with its own drawback, which lies in $O(N^2)$ algorithmic complexity[6] caused by the need to evaluate the proposal N times *for each particle*. This is a serious performance hit compared to $O(N)$ complexity of the traditional particle filter. Given that one can compensate unsuitably low number of effective particles simply by increasing N, any method that

[5] a measure similar to *number of effective particles* used in this text.
[6] where N is the total number of particles.

improves accuracy (which directly relates to the number of effective particles) at the cost of increased complexity of the algorithm must, in our opinion, prove that it brings an improvement even when this trade-off is taken into account. The authors of the Marginal Particle Filter suggest using approximate methods from N-body learning that can reduce the cost to $O(N\,log(N))$[7] and argue that additional error introduced by this approximation can be made insignificant with regards to inherent Monte Carlo error while still offering substantial performance improvement.

For some cases it is however advantageous to have the dependency on x_{t-1} integrated out in sampling and weight update step, especially when the proposal density is complex. Such approach applied to atmospheric assimilation is studied in [19].

Marginal Particle Filter should not be confused with so-called *marginalized particle filter*, a different technique more commonly known as the *Rao-Blackwell particle filter* that exploits cases when the transition density $p(x_t|x_{t-1})$ can be expressed as a (conditional) product of two or more participating densities, some of which can be estimated using using more tailored and efficient Bayesian filtering methods, e.g. using the Kalman filter.

[7] or even to $O(N)$ when some favorable conditions are met.

Chapter 2

Dispersion Model Assimilation

This chapter formulates the mathematical model of the physical phenomena we are concerned with and describes how it is applied to recursive Bayesian estimation. After summarizing the models, it continues by construction of assimilation methodology. State space with its transition density and measurement model with its observation density are defined. It finishes by derivation of proposal densities for the particle filtering methods tailored to this specific problem.

Unless noted otherwise, this whole chapter is based on [21].

2.1 Atmospheric Dispersion Simulation

This section is concerned with the means of representing pollutant dispersion in atmosphere. We use a model based on a state-space representation presented in [9] with only slight modifications in order to perform the assimilation in the early phase of an atmospheric radioactive pollutant release from a facility that contains radioactive material.[1] The model assumes that so-called *puffs* containing varying levels of the pollutant are periodically released from an approximately known location within the facility. Such puffs are then carried by the wind and disperse according to atmospheric conditions; more specifically, the spatial concentration of the pollutant (2.1), which is expressed in terms of total radioactive activity in our case, is assumed to be 3D Gaussian distributed around its center,

[1] we use a nuclear power plan Temelín as an example of such facility in this text.

$$C(\mathbf{s},\,\tau) = \frac{Q_\tau}{(2\pi)^{\frac{3}{2}}\sigma_x\sigma_y\sigma_z}\exp\left[-\frac{(s_x - l_{x,\tau})^2}{2\sigma_x^2} - \frac{(s_y - l_{y,\tau})^2}{2\sigma_y^2} - \frac{(s_z - l_{z,\tau})^2}{2\sigma_z^2}\right], \quad (2.1)$$

where $\mathbf{s} = (s_x, s_y, s_z)$ denotes spatial coordinates where the concentration is to be determined, $\mathbf{l}_\tau = (l_{x,\tau}, l_{y,\tau}, l_{z,\tau})$ coordinates of the puff center at time τ, Q_τ [Bq] total puff activity at given time. The dispersion manifests as growth of the σ parameters. The rate of growth depends on total distance flown by the puff and on current Pasquill's stability category.[2] In case of radioactive release the total puff activity decreases over time by radioactive decay. The frequency of puff release must be carefully chosen, it must be high enough so that the effects of a series of puffs approximate effects of a contiguous plume well enough, but the higher the value the greater computational power is needed and, more importantly, release frequency greater than the one of important measuring devices would introduce harmful ambiguity and should be, in our belief, avoided. In our case, a radioactive monitoring network around the facility, the most critical source of the pollutant release rate information, measures time-integrated dose in 10-minute intervals, we therefore set release interval to 10 minutes in our experiments.

To summarize trajectory development of each puff, its coordinates $\mathbf{l}_{t+1}$ at time step $t+1$ given previous coordinates $\mathbf{l}_t$ and wind field are given by

$$\begin{aligned} l_{x,t+1} &= l_{x,t} - \Delta t\, \dot{v}_t(\mathbf{l}_t)\sin(\dot{\phi}_t(\mathbf{l}_t)), \\ l_{y,t+1} &= l_{y,t} - \Delta t\, \dot{v}_t(\mathbf{l}_t)\cos(\dot{\phi}_t(\mathbf{l}_t)), \\ l_{z,t+1} &= l_{3,\kappa,t}\ , \end{aligned} \quad (2.2)$$

where $\dot{\phi}_t(\mathbf{s})$ denotes given wind direction at time t and spatial coordinates $\mathbf{s}$ and $\dot{v}_t(\mathbf{s})$ wind speed.

Note that this model can be considered too simplistic for a serious simulation, however this is much less a concern for assimilation, where real-world measurements constantly correct imperfections of the model. As this model is only one of the many practical ones, we aim to abstract out its details where feasible in software design so that it can be replaced without need to adapt other parts of the resulting software stack.

2.2 Dose and Wind Field Measurements

As atmospheric activity concentration is not feasibly directly measurable, it needs to be observed indirectly using radioactive monitoring network, which in our scenario

[2] a meteorologic measure of atmospheric turbulence, an important factor affecting dispersion.

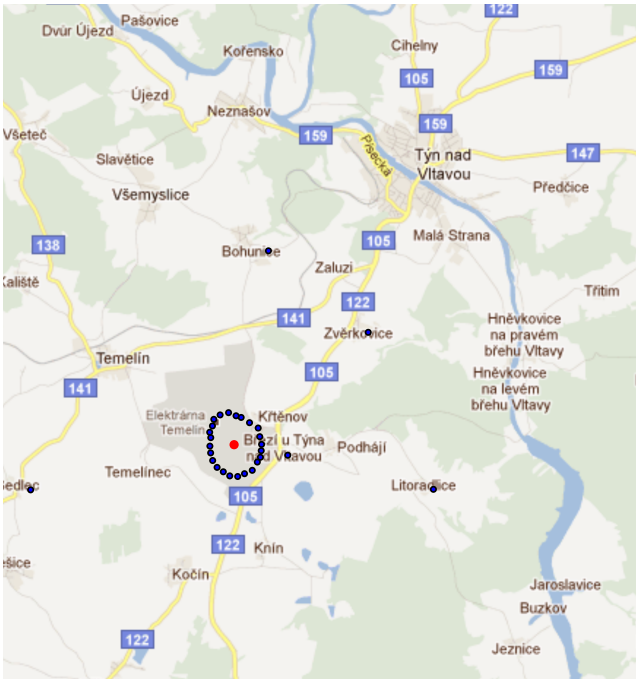

Figure 2.1: Temelín power plant and adjacent monitoring network. Assumed source of release is marked by red circle, blue circles represent dose receptors.

consists of 24 on-line absorbed ionizing dose sensors laid out in a circle around the facility plus additional such devices scattered in nearby villages. These receptors, which are marked using blue circles in the Figure 2.1, periodically send out their readouts. Given spatial pollutant concentration from our model, the dose at given receptor location $\mathbf{s}_r$ can be calculated using

$$D(\mathbf{s}_r, t_1, t_2) = K_c \int_{t_1}^{t_2} \Phi(\mathbf{s}_r, \tau)\, \mathrm{d}\tau, \tag{2.3}$$

where K_c is a physical constant depending on the radionuclide and $\Phi(\mathbf{s}_r, \tau)$ fluency rate that can be computed using

$$\Phi(\mathbf{s}_r, \tau) = \int_{\Omega} \frac{C(\mathbf{s},\, \tau)\,(1 + k\,\mu r)\exp(-\mu r)}{4\pi r^2}\, \mathrm{d}\mathbf{s}. \tag{2.4}$$

Here, k and μ are physical constants, $r = ||\mathbf{s}_r - \mathbf{s}||$ distance from receptor and Ω is the volume in which there is not-negligible concentration. Both dose and concentration from a series of puffs is additive. It should be noted that analytical solution of (2.4) is not known and must be evaluated numerically and presents the most resource-demanding part of the model even if techniques to reduce problem

size are employed.[3] Integral in (2.3) is computed as a simple addition of the element of integration at chosen time sub-steps.

Meteorologic situation is another category of measurements. We assume that forecasts from a numerical model are available and denote wind speed and direction prognoses by $\tilde{v}_t(\mathbf{s})$ and $\tilde{\phi}_t(\mathbf{s})$ respectively, where t and $\mathbf{s}$ have their usual meaning in this text of time and spatial coordinates. Inspired by real-world constraints, we assume that meteorologic forecasts are computed on a rather sparse grid both temporally and time-wise[4] and, because we are concerned with an early phase of a pollutant release, that local wind field deviations in the vicinity of the facility are important, we take another observation into account: an anemometer located in the facility that measures wind speed, denoted as $v_t(\mathbf{s})$ in this text, and wind direction, denoted as $\phi_t(\mathbf{s})$.

2.3 Assimilation: The State Model

With mathematical assimilation apparatus and physical models being set, last step is to combine them. This section formulates the assimilation of the atmospheric pollutant release in the terms of recursive Bayesian estimation, i.e. we construct the state variable $\mathbf{x}_t$ with its transition density $p(\mathbf{x}_t|\mathbf{x}_{t-1})$. In later sections the observation variable $\mathbf{y}_t$ along with observation density $p(\mathbf{y}_t|\mathbf{x}_t)$ is build. Finally we also derive proposal densities for particle filter specific to our problem.

2.3.1 Wind Field Corrections

A simple scheme seen in [8] to combine local measurements with numerical forecast was employed. The corrected wind speed and direction quantities $\dot{v}_t(\mathbf{s})$ and $\dot{\phi}_t(\mathbf{s})$, respectively, are given by

$$\dot{v}_t(\mathbf{s}) = a_t \tilde{v}_t(\mathbf{s}), \tag{2.5}$$

$$\dot{\phi}_t(\mathbf{s}) = b_t + \tilde{\phi}_t(\mathbf{s}), \tag{2.6}$$

where $a_t \in \mathbb{R}_0^+$ and $b_t \in \mathbb{R}$ are biases (the former being relative, the latter additive) of the forecast, hidden artificial dimensionless random variables that are assimilated. All mentioned quantities are time-dependent. Both a_t and b_t random variables are expected to have mean value of their previous realization, i.e.

[3] in particular, we have employed approximation using Gauss quadratures[6] and a "n/μ" problem-reduction technique[15] when computing (2.4) in our test runs.

[4] in our tests we have used a source with 9 km spatial and 1 hour temporal grid step.

$\text{mean}(a_t) = a_{t-1}$, $\text{mean}(b_t) = b_{t-1}$, with a_t having relative standard deviation γ_a and b_t having standard deviation of σ_b. Example conforming distributions with additional favorable properties could be

$$\begin{aligned} p(a_t|a_{t-1}) &= \mathcal{G}(\gamma_a^{-2}, \gamma_a^2 a_{t-1}), \\ p(b_t|b_{t-1}) &= t\mathcal{N}\left(b_{t-1}, \sigma_b^2, \langle b_{t-1} - \pi, b_{t-1} + \pi \rangle\right). \end{aligned}$$

The favorable properties lie in availability of so-called conjugate priors (shown later in this text) in analytical forms for these probability density functions; truncated normal distribution chosen for b_t is in fact only an approximation of formally more appropriate wrapped normal distribution. We argue that truncated normal distribution is a good approximation for practical choices of $\sigma_b \leq \pi/6$; for significantly larger deviations a closer approximation in form of von Misses distribution could be employed.[5]

2.3.2 Release Model

While many previous works assume temporally correlated release rate, where released pollutant quantity Q_t depends on quantity Q_{t-1} released in the previous step, we consider them to be uncorrelated. Among suitable probability density functions with non-negative support, Gamma distribution was used so that

$$p(Q_t|Q_{t-1}) = p(Q_t) = \mathcal{G}\left(\alpha_Q, \beta_Q\right), \tag{2.7}$$

where the choice of $\alpha_Q = 1$ and $\beta_Q \to 0$ serves the purpose of making this an uninformative prior, the one that assumes as little as possible about the system apart from objective constrains such as non-negativeness or continuousness. In our case this can be done without impacting the feasibility of the assimilation because the measurements supply enough information.

2.3.3 State Model Summary

To conclude construction of the state variable, we need to add the rest of quantities that fully represent the state of pollutant dispersion in the atmosphere — the locations of individual puffs, their dispersion parameters and activities, hence

$$\mathbf{x}_{t\,\text{full}} = (a_t, b_t, Q_t,\ Q_{1,t}, \mathbf{l}_{1,t}, d_{1,t}, \boldsymbol{\sigma}_{1,t},\ Q_{2,t}, \mathbf{l}_{2,t}, d_{2,t}, \boldsymbol{\sigma}_{2,t}, \ldots, Q_{K,t}, \mathbf{l}_{K,t}, d_{K,t}, \boldsymbol{\sigma}_{K,t}), \tag{2.8}$$

[5] although not effortlessly as this would invalidate our derivation of conjugate prior for b_t.

where K stands for total puff count and k^{th} puff at time t is characterized by its activity $Q_{k,t}$, spatial coordinates $\mathbf{l}_{k,t}$, total flown distance $d_{k,t}$ and spatial dispersion coefficients $\boldsymbol{\sigma}_{k,t}$. Under this formalism, the state space is extremely high-dimensional (for the purpose of Monte Carlo sampling) and its dimension grows without bounds in time as new puffs are released. Neither of these properties pose a real problem in practice, because

1. puffs can be discarded when certain criteria are met and their contribution to the environment is negligible, usually residual activity when radioactive decay is taken into account, distance from the area of interest and total dispersion can be considered; with this mechanism, no puffs are simulated under normal conditions where no pollutant release occurs in continuous monitoring scenario,

2. apart from the (a_t, b_t, Q_t) triplet, all other quantities in (2.8) are deterministic given the triplet, previous state and variables considered external such as wind field forecast and Pasquill's stability category. In particular, assuming puff number K is to be released in time step t, its variables could be put as

$$\begin{aligned} Q_{K,t} &= Q_t, \\ \mathbf{l}_{K,t} &= \mathbf{l}_{\text{facility}}, \\ d_{K,t} &= 0 \\ \boldsymbol{\sigma}_{K,t} &= (0,0,0)\,, \end{aligned}$$

   where $\mathbf{l}_{\text{facility}}$ is expected location of the release within the facility. Similarly, location of an already flying puff (numbered k) is updated according to (2.2) which in turn uses wind field (2.5)(2.6) corrected using a_t, b_t; activity, distance and dispersion coefficients are given by

$$\begin{aligned} Q_{k,t} &= e^{-\lambda \Delta t} Q_{k,t-1}, \\ d_{k,t} &= d_{k,t-1} + \|\mathbf{l}_{k,t} - \mathbf{l}_{k,t-1}\|\,, \\ \boldsymbol{\sigma}_{k,t} &= \boldsymbol{\sigma}(t-k, d_{k,t}, \mathbf{l}_{k,t}), \end{aligned} \tag{2.9}$$

   where λ is exponential decay constant[6] of the radionuclide causing the pollution and Δt time since the last step.

Given the deterministic property of large part of the full state, we can shrink the state space for the purposes of assimilation into

$$\mathbf{x}_t = (a_t, b_t, Q_t) \tag{2.10}$$

[6] given half-life $t_{1/2}$ exponential decay constant is $\lambda = \frac{t_{1/2}}{\ln 2}$.

along with the transition density

$$p(\mathbf{x}_t|\mathbf{x}_{t-1}) \quad = \quad p(Q_t)p(a_t|a_{t-1})p(b_t|b_{t-1}), \tag{2.11}$$

which makes this a more conceivable task for sequential Monte Carlo estimation.

Still, the full state needs to be accounted for when implementing actual representation of the model for assimilation or when strictly adhering to mathematical formalism. The rest of this text will usually omit deterministic part of the state vector for the sake of simplicity.

2.4 Measurement Model

Defining the observation vector $\mathbf{y}_t$ takes more straightforward path, as it is just a concatenation of available measurements, we may state that

$$\mathbf{y}_t = (v_t, \phi_t, y_{t,1}, \ldots, y_{t,M}),$$

where v_t and ϕ_t are wind speed and wind direction anemometer measurements, M total number of radioactive dose receptors and $y_{t,k}$ is total dose detected by the k^{th} receptor in time interval $\langle t-1, t\rangle$. With state and observation quantities defined, probabilistic description of the observation model (1.2) can be factored to conditionally independent densities and expressed as

$$p(v_t, \phi_t, y_{t,1}, \ldots, y_{t,M}|a_t, b_t) \quad = \quad p(v_t|a_t)p(\phi_t|b_t)p(y_{t,1}, \ldots, y_{t,M}|a_t, b_t). \tag{2.12}$$

2.4.1 Wind Field Measurement Model

We assume that anemometer velocity measurements have non-biased relative error, denoted as γ_v in this text, therefore it can be modeled using a probability density function with non-negative support, mean value $\dot{v}_t(\mathbf{s}_0)$ and standard error $\gamma_v \dot{v}_t(\mathbf{s}_0)$, where $\mathbf{s}_0$ are spatial coordinates of the anemometer. For its favorable properties, we have used the inverse gamma distribution $i\mathcal{G}(\gamma_v^{-2}+2, (\gamma_v^{-2}+1)\dot{v}_t(\mathbf{s}_0))$ that fulfills these demands, hence, taking (2.5) into account,

$$p(v_t|a_t) = i\mathcal{G}\left(\gamma_v^{-2}+2, (\gamma_v^{-2}+1)\dot{v}_t(\mathbf{s}_0)\right) = i\mathcal{G}\left(\gamma_v^{-2}+2, (\gamma_v^{-2}+1)a_t\tilde{v}_t(\mathbf{s}_0)\right). \tag{2.13}$$

Wind direction was decided to be treated as an unbiased measurement with constant standard deviation σ_ϕ. The natural choice for for such situations being the normal density $\mathcal{N}(\dot{\phi}_t(\mathbf{s}_0), \sigma_\phi^2)$, substituting wind direction corrections (2.6) we have

$$p(\phi_t|b_t) = \mathcal{N}(\dot{\phi}_t(\mathbf{s}_0), \sigma_\phi^2) = \mathcal{N}(b_t + \tilde{\phi}_t(\mathbf{s}_0), \sigma_\phi^2). \tag{2.14}$$

2.4.2 Dose Measurement Model

To derive measurement model for radioactive doses, $p(y_{t,1}, \ldots, y_{t,M}|a_t, b_t)$ can be further factored into conditionally independent densities

$$p(y_{t,1}, \ldots, y_{t,M}|a_t, b_t) = \left(\prod_{i=1}^{M} p(y_{t,i}|d_{t,i})\right) p(d_{t,1}, \ldots, d_{t,M}|a_t, b_t), \tag{2.15}$$

where $d_{t,i}$ is expected dose measured on i-th receptor in time slot t (including expected natural background dose y_{nb}). $p(d_{t,1}, \ldots, d_{t,M}|a_t b_t)$ represents computation of the expected doses using formulas (2.3), (2.4) and (2.1), where puff trajectories develop according to corrected wind field model (this is where a_t and b_t enter the calculation). $p(d_{t,1}, \ldots, d_{t,M}|a_t b_t)$ is therefore fully deterministic (i.e. Dirac delta distribution). Parameters of each of the $p(y_{t,i}|d_{t,i})$ correspond to possibilities of dose measuring devices, and according to [18] their error is proportional to the measured dose with constant of proportionality γ_y usually between 7–20%. Dose measurements were again simulated using inverse gamma distribution,

$$\forall i \in 1, \ldots, M \qquad p(y_{t,i}|d_{t,i}) = i\mathcal{G}\left(\gamma_y^{-2} + 2, (\gamma_y^{-2} + 1)d_{t,i}\right). \tag{2.16}$$

2.5 Problem-specific Proposal Density

The densities defined in subsection 2.3.3 and section 2.4 fully describe a recursive Bayesian estimation problem. As however mentioned earlier, our task of environmental assimilation is a very demanding one for Monte Carlo estimation, mainly because our observation density is extremely narrow compared to transition density, which must be wide enough to account for rapid changes of weather or release rate. If we used the bootstrap variant of the particle filter directly, we would end up with overwhelming majority of particles being sampled into areas where whey would be assigned negligible likelihoods by the observation model. In other words, we would face severe case of the sample impoverishment problem.

It is therefore of great benefit to steer particle sampling into regions with better overlap with the observation density. This can be done by employing a better-suited proposal density $q(\mathbf{x}_t|\mathbf{x}_{t-1}, \mathbf{y}_t)$. As shown in [14] on a somewhat simpler problem with the same measurement model, it is undoubtedly advantageous to spend resources on generation of "better" particles than to simply increase the number of particles in the particle filter. Such claim does not hold universally for all uses of sequential Monte Carlo sampling, but holds in our case because analytically intractable time-spatial integration in the measurement model greatly overweights complexity introduced by more elaborate particle sampling.

We start construction of our proposal density by assuming conditional independence of some variables (this assumption is an approximation, it is conceivable because of loose requirements posed on proposal density),

$$\begin{aligned} q(\mathbf{x}_t|\mathbf{x}_{t-1},\mathbf{y}_t) &= p(a_t,b_t,Q_t|a_{t-1},b_{t-1},Q_{t-1},v_t,\phi_t,y_{t,1},\ldots,y_{t,M}) \\ &\approx p(Q_t|a_t,b_t,y_{t,1},\ldots,y_{t,M})p(a_t|a_{t-1},v_t)p(b_t|b_{t-1},\phi_t). \end{aligned} \quad (2.17)$$

As can be seen, wind field variables are made conditional on the most informative measurements — wind direction correction b_t depends on measured direction ϕ_t and speed correction a_t likewise depends on observed velocity v_t. Released dose expectation takes receptor read-outs into account while simultaneously coping with proposed wind field. Forms of individual proposal factors are derived below for completeness.

2.5.1 Wind Direction Conjugate Proposal Density

For the choice of wind direction transition density $p(b_t|b_{t-1}) = \mathcal{N}(b_t; b_{t-1}, \sigma_b)$ and wind direction observation density $p(\phi_t|b_t) = \mathcal{N}(\phi_t; \tilde{\phi}_t + b_t, \sigma_\theta)$, the conjugate proposal can be derived as follows:

$$p(b_t|b_{t-1},\phi_t) = \frac{p(\phi_t|b_t,b_{t-1})p(b_t|b_{t-1})}{p(\phi_t|b_{t-1})} \quad (2.18)$$

$$= \frac{\mathcal{N}(\phi_t;\tilde{\phi}_t+b_t,\sigma_\theta)\mathcal{N}(b_t;b_{t-1},\sigma_b)}{p(\phi_t|b_{t-1})}$$

$$= \frac{1}{p(\phi_t|b_{t-1})}\frac{1}{2\pi}\frac{1}{\sigma_\theta}\exp\left(-\frac{1}{2}\left(\frac{\phi_t-\tilde{\phi}_t-b_t}{\sigma_\theta}\right)^2\right)\frac{1}{\sigma_b}\exp\left(-\frac{1}{2}\left(\frac{b_t-b_{t-1}}{\sigma_b}\right)^2\right)$$

$$= \mathcal{N}\left(\left(\sigma_b^{-2}+\sigma_\phi^{-2}\right)^{-1}\left(\sigma_b^{-2}b_{t-1}+\sigma_\phi^{-2}\left(\phi_t-\tilde{\phi}_t\right)\right),\left(\sigma_b^{-2}+\sigma_\phi^{-2}\right)^{-1}\right). \quad (2.19)$$

Note that (2.18) is obtained by applying Bayes rule and that $p(\phi_t|b_{t-1})$ does not have to be computed at all because it serves as a normalizing constant in a distribution with well-known (Gaussian) form. $\tilde{\phi}_t$ was used instead of $\tilde{\phi}_t(\mathbf{s}_0)$ to denote wind direction forecast at time t and location of the anemometer for brevity.

2.5.2 Wind Speed Conjugate Proposal Density

In order to derive formula for $p(a_t|a_{t-1},v_t)$, let us first show general (and well known) result for for inverse gamma density and gamma density pair with appro-

priate parametrization. Suppose that $p(y|a) = i\mathcal{G}\left(y; \alpha, (\alpha-1)\,a\right)$, $p(a) = \mathcal{G}(a; k, \theta)$ and that y is known. Then $p(a|y)$ is also a gamma distribution and can be again obtained using the Bayes rule (2.20), yielding (2.22). The transition from (2.21) to (2.22) makes use of the fact that the whole term in (2.21) is a probability density function, therefore it must integrate to 1; however, the non-constant parts (with regards to quantity a) have the form of the gamma distribution, therefore the whole term must necessarily be a gamma distribution itself

$$
\begin{aligned}
p(a|y) &= \frac{p(y|a)p(a)}{p(y)} && (2.20)\\
&= \frac{1}{p(y)} i\mathcal{G}\left(y; \alpha, (\alpha-1)\,a\right)\mathcal{G}(a; k, \theta)\\
&= \frac{1}{p(y)} \frac{\left((\alpha-1)\,a\right)^{\alpha}}{\Gamma(\alpha)} y^{-\alpha-1} \exp\left(-\frac{(\alpha-1)\,a}{y}\right) \frac{1}{\theta^k \Gamma(k)} a^{k-1} \exp\left(-\frac{a}{\theta}\right)\\
&= \frac{(\alpha-1)^{\alpha}\, y^{-\alpha-1}}{p(y)\Gamma(\alpha)\,\theta^k\Gamma(k)} a^{\alpha+k-1} \exp\left(-a\left((\alpha-1)\,y^{-1} + \theta^{-1}\right)\right)\\
&= C\left(y, \alpha, \theta, k\right) a^{\alpha+k-1} \exp\left(-a\left((\alpha-1)\,y^{-1} + \theta^{-1}\right)\right) && (2.21)\\
&= \mathcal{G}\left(a; \alpha+k, \left((\alpha-1)\,y^{-1} + \theta^{-1}\right)^{-1}\right). && (2.22)
\end{aligned}
$$

In our case (again with $\tilde{v}_t\left(\mathbf{s}_0\right)$ forecast being denoted simply as $\tilde{v}_t$):

$$
\begin{aligned}
a &= a_t\tilde{v}_t\\
y &= v_t\\
p(y|a) &= p(v_t|a_t\tilde{v}_t)\\
&= i\mathcal{G}\left(v_t; \gamma_v^{-2}+2, (\gamma_v^{-2}+1)a_t\tilde{v}_t\right)\\
p(a) &= p(a_t\tilde{v}_t|a_{t-1})\\
&= \mathcal{G}(a_t\tilde{v}_t; \gamma_a^{-2}, \gamma_a^2 a_{t-1}\tilde{v}_t)\\
p(a|y) &= p(a_t\tilde{v}_t|a_{t-1}, v_t)\\
&= \mathcal{G}\left(a_t\tilde{v}_t; \gamma_v^{-2}+\gamma_a^{-2}+2, \left(\left(\gamma_v^{-2}+1\right)v_t^{-1} + \gamma_a^{-2}a_{t-1}^{-1}\tilde{v}_t^{-1}\right)^{-1}\right)\\
p(a_t|a_{t-1}, v_t) &= \mathcal{G}\left(a_t; \gamma_v^{-2}+\gamma_a^{-2}+2, \left(\left(\gamma_v^{-2}+1\right)v_t^{-1}\tilde{v}_t + \gamma_a^{-2}a_{t-1}^{-1}\right)^{-1}\right). && (2.23)
\end{aligned}
$$

2.5.3 Released Activity Proposal Density

Derivation of the approximation of the proposal factor $p(Q_t|a_t, b_t, y_{t,1}, \ldots, y_{t,M})$ is less straightforward. Let us reformulate the $\prod_{i=1}^{M} p(y_{t,i}|d_{t,i})$ part of dose measure-

ment density (2.15). We extract activity[7] of the just released puff Q_t (marked simply as Q in this subsection for simplicity) from expected doses $d_{t,i}$ and fix other parameters (such as activities of other puffs and wind field) so that

$$\begin{aligned}\prod_{i=1}^{M} p(y_{t,i}|d_{t,i}) &= \prod_{i=1}^{M} i\mathcal{G}\left(\gamma_y^{-2}+2, (\gamma_y^{-2}+1)d_{t,i}\right) = \\ = p(y_{t,1},\ldots,y_{t,M}|Q) &= \prod_{i=1}^{M} i\mathcal{G}\left(\alpha_i, \beta_i Q + m_i\right). \end{aligned} \tag{2.24}$$

Applying Bayes rule to $p(Q_t|a_t, b_t, y_{t,1}, \ldots, y_{t,M})$ and fixing a_t, b_t parameters we get

$$\begin{aligned} p(Q|y_{t,1:M}) &= \frac{p(y_{t,1:M}|Q)p(Q)}{p(y_{t,1:M})} \\ &\propto Q^{\alpha_Q - 1}\exp\left(-\beta_Q Q\right)\prod_{i=1}^{M}(\beta_i Q + m_i)^{\alpha_i} y_{t,i}^{-\alpha_i - 1}\exp\left(-\frac{\beta_i Q + m_i}{y_{t,i}}\right), \end{aligned} \tag{2.25}$$

where we have substituted from (2.24) and (2.7). Since this density has no standard known form, the normalization factor cannot be inferred analytically, as it was done above. We can approximate it using normal distribution

$$p(Q|y_{t,1:M}) \approx \mathcal{N}(\mu_Q, \sigma_Q^2), \tag{2.26}$$

whose parameters are obtained using Laplace's method.[11] Its mean μ_Q is aligned with local maxima, denoted as $\hat{Q}$, of (2.25), which is obtained by computing derivative of its logarithm

$$\begin{aligned}\frac{d\log(p(Q|y_{t,1:M}))}{dQ} &= \frac{d}{dQ}\left(\sum_{i=1}^{M}\left[\alpha_i\log(\beta_i Q + m_i) - \frac{\beta_i Q + m_i}{y_{t,i}}\right] + (\alpha_Q - 1)\log Q - \beta_Q Q\right) \\ &= \sum_{i=1}^{M}\left[\frac{\alpha_i\beta_i}{\beta_i Q + m_i} - \frac{\beta_i}{y_{t,i}}\right] + \frac{\alpha_Q - 1}{Q} - \beta_Q, \end{aligned} \tag{2.27}$$

which yields a sum of decreasing rational functions, allowing us to find the only possible zero crossing using efficient numerical methods. Second derivative of (2.27) is then computed and made equal to inverse covariance matrix of the approximated normal distribution (2.26) at point of its maxima,

$$\frac{1}{2}\sigma_Q^{-2} = \sum_{i=1}^{M}\left[\frac{\alpha_i\beta_i^2}{(\beta_i\hat{Q} + m_i)^2}\right] + \frac{\alpha_Q - 1}{\hat{Q}^2}.$$

[7] this is possible because $\forall i\ d_{t,i}$ depends on linear combination of puff activities.

As negative released activity is nonsensical, the activity proposal approximation (2.26) is truncated to $\mathbb{R}_0^+$ before it is sampled from.

Chapter 3

Software Analysis

This chapter is devoted to analysis of suitable software solution for performing assimilation of atmospheric dispersion models such as the one presented in chapter 2 using (a variant of) sequential Monte Carlo methods described in chapter 1. Requirements posed on such solution (including foreseen use cases) are discussed first so that we can consult them when making software design decisions. The chapter continues by overview of existing projects that were deemed useful as building blocks of the desired software stack; the Python software environment enhanced by Cython Python-to-C compiler, Eigen C++ template library for linear algebra and PyBayes Python/Cython library for Bayesian filtering are presented. It is concluded by our suggestion of high-level design of the desired software stack, interaction of individual components and enhancements to existing projects.

3.1 Requirements

We are concerned with designing and developing software stack to perform assimilation according to theoretical chapters above. Intended uses of such stack include (but are not limited to):

Research platform The resulting project, when paired with suggested components, should form a solid environment for rapid prototyping and validating new variants of assimilation methods and/or dispersion models. This has implications on the used development language and development iteration time, which needs to be rather low.

Continuous monitoring tool The desired software should allow to be adapted to serve as a computational back-end for hypothetical tool dedicated to contin-

uous monitoring of pollutant releases; such tool would form additional information source for decision-makers in the early phase of a radioactive release accident.

Distributed system worker It is desired that the target project is be usable as a worker in a distributed web-based assimilation simulation system, which is currently being developed. The user interface allowing to specify simulation parameters runs in a web server and distributes the work to computing nodes, which feed back the results.

With these use-cases in mind, we can infer general requirements on the desired solution:

High-Level Language The need for ability to rapidly prototype new variants of assimilation and environmental modeling methods calls for use of a high-productivity programming language. Such language should be easy to learn or well-known so that is is attractive also for researchers without interest in programing on its own.

Extensibility While we implement methods as described in chapter 1 and chapter 2 (with some insignificant details being omitted in the theoretical chapters for clarity), the software should expect that improvements to the methodology will arise and should allow them to be incorporated. It should allow to use implemented assimilation methods with a different suitable dispersion model, allow user to plug in custom weather forecast source etc.

Documentation Up-to-date, complete and readable API[1] documentation is needed so that the software can be used and enhanced by anyone, not just its author. Such documentation can assume that users already have mathematical understanding of the problem.

Efficiency Uses such as continuous monitoring or batch processing in distributed system dictate for high performance given available hardware resources. Inherent property of Monte Carlo methods is that additional resources can be converted to more samples, which in turn increases accuracy. More efficient computation therefore means ability to achieve better justified results, therefore efficiency should be high on the requirements list.

Interoperability The desired software stack should allow for effortless cooperation with supportive software. This includes for example data visualization utilities or tools that prepare inputs. Well-known or standardized formats should be used for data exchange.

[1] Application Programming Interface, a set of rules defining how a software library is used.

Portability The software should build and run on major server and desktop platforms: GNU/Linux (compatibility with other UNIX-like systems being a plus) and Microsoft Windows. Installation of the library on modern systems should be straightforward, no non-standard components should be required.

Openness Self-contained and reusable parts of the software solution should be released to public under open-source license. The authors believe that this approach leads to highest-quality software in the long run. This requirement is not applied to problem-specific parts of the software stack if they are not considered reusable.

It should be noted that many of the demands are in direct conflict with each other, the most prominent example being high programmer productivity and computational efficiency (to be achieved at the same time). The task of following analysis is to find reasonable trade-off between these conflicting requirements or to propose innovative solutions that remove the conflict to some extent.

Because of significant overlap with demands described in [13], we will reuse programming language feasibility analysis performed there for the PyBayes library. The authors chose Python as the preferred implementation environment. The rationale is its developer convenience, readability of code, interoperability with C and MATLAB, availability of 3rd party topical software and dynamic and interpreted nature. It was shown that Cython compiler can mask the only significant draw-back of Python, which is poor performance of CPU-intensive code. This implementation environment is presented in the next section.

3.2 Foundations

The desired software is not built on a green field, we base it on existing environments and projects to form a complete and usable stack. This section presents these projects so that proper design of their interaction can be developed later on. We ascend from the lower-level parts, the Python language and software ecosystem surrounding it, the Cython compiler, the newly incorporated project — Eigen (a C++ template library for linear algebra), to the employed library for Bayesian filtering, PyBayes.

3.2.1 Python

Python[2] is a very high-level language actively developed by the community of volunteers and professionals centered around the Python Software Foundation.[3] Python encourages object-oriented programming paradigm, incorporates many concepts of functional programming and is quick to adopt emerging constructs. It is strongly but dynamically-typed[4] with call-by-object[5] function argument passing and assignment semantics[6] and immutable basic types such as numbers and strings. Python provides automatic memory management, which is done through reference counting in its reference implementation (other implementations are free to implement it differently).

Implementations

The name Python may refer to the language, its standard library which is part of it, or even the principal implementation, CPython. While many implementations exist, for example Jython,[7] IronPython[8] (that are based on Java and .NET runtime environments, respectively) and promising just-in-time compiling PyPy,[9] CPython is shipped with the standard Python installation as is by far the most used. It is written in C and provides a relatively stable C API that exposes its internals. This API is exploited by Cython described below.

Versions

Python (the language) comes in two major versions, 2 and 3. Python 2 is considered legacy, its specification is frozen and the CPython implementation only receives bug fixes. Python 3, which introduces a couple of backward-incompatible changes to the language and standard library, is the current and promoted version. Still, many useful Python projects have not yet finished their Python 3 port, so Python 2 is, at the time of writing, the version with widest palette of 3rd party components available. Methods for almost automatic conversion between the 2 versions of source

[2] https://www.python.org/

[3] https://www.python.org/psf/

[4] types are bound to values, not variables; one variable can assume multiple non-covariant types during its lifetime.

[5] http://effbot.org/zone/call-by-object.htm

[6] objects are referenced, rather than copied, when passed around and assignments in nested scope are invisible to parent scope.

[7] http://www.jython.org/

[8] http://ironpython.net/

[9] http://pypy.org/

code exist: 2to3, 3to2 tools and even Cython can be used as such. We propose developing in the latest revision of the Python 2 language for greatest compatibility while simultaneously avoiding deprecated functionality (which improves accuracy of the automated conversion tools).

Software Ecosystem

NumPy[10] is the de facto standard Python library for numerical computing. It provides a powerful n-dimensional array data-type, the `ndarray`, and a wide range of algebraic operations on top of it. `ndarray` forms the compatibility bridge between nearly all Python libraries related to numerical computing.

The *SciPy library*[11] provides more specialized scientific computing methods not found in NumPy and is in fact just one part of the SciPy ecosystem, which contains *Matplotlib*[12] for data visualization and other projects for symbolic computing, interactive development and so. All these projects are published under permissive open-source licenses and thus freely usable and modifiable.

Drawbacks

Using Python (without additional measures) comes however with some drawbacks. First and foremost, it incurs significant interpreter and dynamic typing overhead. About 60-fold speed improvements were observed when this overhead was removed using Cython on a simple numerical integration example.[13] Such massive overhead is not compatible with our requirement of computing efficiency and must be avoided in the desired software. This conclusion aligns with a general agreement in the literature that Python alone is not suitable for numerically intensive computations.[16, 2]

Another problematic area is thread-based parallelization. While Python supports many approaches to writing multi-threaded code, CPU-bound computations written for multiple threads (but a single process) in pure Python seldom make us of all available CPU cores because of the so-called GIL, the Global Interpreter Lock. Each CPython interpreter thread by default holds this lock and releases it only explicitly for some I/O operations. This results in just one thread (per each CPython interpreter process) running at given time, i.e. completely circumventing parallelization. Methods combating this limitation exist, for example the `multiprocessing`[13] module or Cython-based GIL releasing.

[10] http://www.numpy.org/
[11] http://www.scipy.org/
[12] http://matplotlib.org/
[13] https://docs.python.org/library/multiprocessing.html

3.2.2 Cython

Cython[1] was employed to counter performance problems of pure Python-based solutions when developing PyBayes and it proved extremely useful for this purpose. We present it briefly in this section, concentrating on recent developments not yet available when PyBayes was conceived.

Cython is both an extension of the Python language (a strict superset of it) and its implementation. Its mode of operation is to translate Python source code (`.py` Python or `.pyx` Cython files) into C or C++ code that uses CPython API extensively. The translated files are then compiled into binary Python modules (shared libraries from the operating system point of view: `.so` files on Linux and `.dll` files on Windows) that are treated transparently and indistinguishably[14] by CPython. At the time of writing, Cython achieves virtually 100% compatibility with Python language, meaning that nearly every Python module can be compiled by Cython and then successfully used as a binary module in place of the original one. Cython is however not a replacement of CPython or any other Python implementation: the resulting binary modules depend on CPython environment and its shared library. In fact, the dependency on this specific Python implementation is one of the significant Cython limitations.[15]

Static Typing

Compilation to machine code nullifies the interpreter overhead of traditional Python implementations, which was as high as 290% in the numerical integration example. This should be viewed as an extreme case with reports mentioning 30% performance improvement (with no modification to the code) being more common for general code. The main strength nonetheless lies in the ability to statically type *some* variables, which avoids dynamic-typing overhead mandated by the Python language semantics; typing all variables involved in the accumulation loop of numerical integration example resulted in additional 21× execution time reduction. This version had virtually the same performance as the one hand-written in C; this Cython's ability to come near to C code in terms of performance was shown also with a more complex algorithm.[20] To express typing information for given variables Cython extends the Python language with additional constructs:

`cdef int i` declares a variable of type `int`

[14] with some limitations like inability to programmatically retrieve executed source code.

[15] efforts to make Cython compatible with other implementations exist, though, e.g. PyPy support is approached using a `cpyext` layer: http://docs.cython.org/src/userguide/pypy.html

`cdef bint func(double x):` defines a function with C calling semantics taking `double` floating-point value and returning a boolean value encoded as integer (`bint`)

If `cpdef` was used instead of `cdef` in the function definition above, Cython would generate a Python wrapper around func() so that it can be called from Python code (with usual overhead coming from Python function call semantics). Object-oriented paradigm is supported, class definitions can be annotated using again the `cdef` modifier to form so-called *extension types*, Cython-extended classes whose attributes and methods can be typed. Inheritance is supported, extension types can even by extended by Python code (not the other way around); Cython implements virtual function support in C to keep Python semantics. Extension types, which behave like built-in types such as `str` to Python code, can in turn appear in further Cython typing constructs.

Pure Python Mode

Adding typing annotations to code makes it incompatible with Python and associated development tools such as documentation extractors and static code analysis utilities. This is often undesirable. When using Cython, this can be avoided by placing the annotations outside of `.py` module files. It supports so-called *pure Python* mode, where these would-be incompatible annotations are put into augmentation `.pxd` files. For example a `module.py` file can be kept Python-compatible while still providing necessary typing information in a `module.pxd` file. Another purpose of putting declarations into `.pxd` files is to share them between multiple modules, in a direct analogy to `.h` header files in C. PyBayes uses pure Python mode to resolve the ambivalent requirements of supporting rapid prototyping *and* high performance.

Interfacing C/C++ Code

Another property, and in fact the original rationale behind Cython, is its ability to interface external C and (since version 0.13) C++ code. Given its design (compilation into C/C++), Cython only needs to know signatures of accessed functions and data type layouts. Special syntax is provided to achieve this (Cython is not able to parse C header files, but a range of external projects provide such functionality and generate appropriate Cython declarations). Supported C++ features include polymorphism, templates, bundled interface to C++ standard library and automatic conversion of C++ exceptions to Python equivalents.

Parallelism and GIL

In order to ease development or parallel algorithms that make use of multiple processor cores simultaneously, Cython brings support for parallelizing algorithms using OpenMP-like interface. Internally Cython implements this support by actually emitting OpenMP pragmas into generated C code. It additionally performs control flow analysis of the the variables involved in parallelized blocks to infer their thread-locality and whether they are used as reduction quantities. Hence it suffices to replace a `range` loop by a `prange` (which stands for parallel range) one in simple cases to achieve parallelism.

Parallelization feature would be of little use if Cython did not solve the Global Interpreter Lock issue described above. The GIL is used by CPython to ensure consistency of otherwise non-thread-safe data structures, most prominently reference counters attached to each living Python object. Each variable assignment or function call adhering to Python semantics causes several updates of reference counts of involved objects. Cython can, if types are known and suitable, skip the Python semantics and perform desired operation (like a function call) in pure C without touching CPython data structures, thus removing the need to hold the GIL. Moreover it provides means to express that given code block can execute with the GIL released[16] (which turns on compile-time checking whether this is possible) and means to label entire functions as not requiring the GIL to be held. It is then required that `prange` appears only at places where the GIL is released (in fact, it can be instructed to release the GIL itself using a parameter), which in turn ensures that the nested block can execute in a truly parallel manner.

Templates

Since its 0.16 version Cython allows programmers to avoid some sorts of code duplication by providing simple templating support. The feature called *fused types* is somehow restricted compared template support as known e.g. from the C++ language: all possible specializations of a fused type need to be known at compile time. The advantage is that this approach does not change build system semantics: functions using fused types are still generated in the same phase. Cython generates a typed code block for each possible combination (for public code; in case of publicly inaccessible parts, only actually used variants are generated), which can lead to combinatorial explosion unless used carefully. When interfacing C++ code, fused types can be combined with C++ templating system, so that for example wrappers around generic C++ container types can be written relatively easily.

[16] ability to transiently hold the GIL inside a "nogil" block is also provided.

Typed Memoryviews

Cython 0.16 also introduced support for so-called *typed memoryviews*, a special Cython-specific data type representing a view on a multidimensional numeric array. Unlike NumPy `ndarrays`, memoryviews encode their base data type (e.g. `double`, `int`) and number of dimensions statically instead of dynamically, which along with implementation directly in Cython leads to significantly reduced overhead. Memoryviews provide very fast member access (indexing), transposing and slicing (creating a sub-view on part of the array, potentially of different dimension) while still keeping great flexibility, the format is for example able to express a view on a matrix of third components of a tensor, without copying data:

```
cdef double[:, :] third_components = tensor[:, :, 2]
```

another inherent advantage is that above operations can be done without holding the GIL, e.g. inside parallel blocks.

Memoryviews can be created out of any Python type that supports PEP[17] 3118[18]-style buffer interface — without copying any data. This includes NumPy `ndarrays`, CPython `arrays`, newly introduced Cython arrays (provided just for the purpose of backing memoryviews). Automatic copy-free coercion to NumPy `ndarray` is also implemented, which means that one can freely mix NumPy and memoryview methods. However, even without copying the data, each conversion incurs overhead caused by boilerplate Python code and requires holding the GIL.

It was shown by Jake Vanderplas in https://jakevdp.github.io/blog/2012/08/08/memoryview-benchmarks/ that optimized usage of Cython memoryviews led to to a an algorithm with virtually the same performance as the fastest solution using raw pointers on a nearest neighbor search example. This version was significantly faster than a different one that used legacy direct Cython support for NumPy `ndarray` type. This support still exists, but it cannot do slicing as efficiently as memoryviews, requires holding the GIL for all but element access and the syntax cannot be used for class attributes, among other limitations.

Memoryview feature of Cython is not yet perfect, though. The biggest obstacle is, in our opinion, the lack of support of algebraic operations of top of memoryviews. A mere attempt to add two matrices `a` and `b`

```
cdef double[:, :] a, b
result = a + b
```

[17] Python Enhancement Proposal.
[18] http://www.python.org/dev/peps/pep-3118

results in a compile-time error as of Cython 0.20.1 with it complaining about invalid operands for the + operator. It is of course possible to exploit bidirectional coercion to NumPy types and use NumPy methods,

```
import numpy as np

cdef double[:, :] a, b
np.add(a, b)
np.asarray(a) + np.asarry(b)
```

but implied overhead and the need for the GIL negates virtually all advantages of using memoryviews in the first place. A project called *array expressions* had been worked on by memoryview support author, Mark Florisson, as part of his Master's thesis.[5] Array expressions would add functionality to perform algebraic operations with memoryviews directly into Cython, through use of embedded *minivect*[19] expression compiler. Unfortunately, a request to merge this support into Cython code-base was recalled by its author after discussions whether and how to include the minivect project.

Back in the days when memoryviews did not exist and custom support for NumPy `ndarray` was state-of-the-art for computations in Cython, one could use Shane Legg's *Tokyo*[20] project as a Cython wrapper for a range of functions of system BLAS[21] implementation and a couple of system LAPACK[22] functions. While not even close to flexibility and elegance that would have been offered by array expressions (Tokyo inherited many legacy properties from BLAS and LAPACK, most prominently a fixed-function API which is painful to use), Tokyo did its job and offered remarkable overhead reduction. Still, BLAS and LAPACK compatibility problems that have been experienced when deploying Tokyo-based software along with its API motivated us to investigate different possibilities than porting Tokyo to memoryviews. A viable project, Eigen, was found and is presented in the next section.

[19] https://github.com/markflorisson88/minivect

[20] http://www.vetta.org/2009/09/tokyo-a-cython-blas-wrapper-for-fast-matrix-math/

[21] Basic Linear Algebra Subprograms, a de facto standard API for computer linear algebra dating back to 1979.

[22] Linear Algebra PACKage, extension of BLAS with higher-level methods like equation solving or decompositions.

3.2.3 Eigen

Eigen[23] is a mature C++ template library for linear algebra written entirely in header files. Eigen makes extensive use of some of the lesser known features offered by the C++ language to achieve maximum performance, flexibility and elegance of use. Eigen employs templates to implement lazy evaluation,[24] so that expressions like

```
MatrixXd A, B, C, D;
(...)
A = 2*B - 3*(D - B));
```

expand only to one for loop (something that is impossible with a fixed-function API like the one of BLAS) and do not use any temporary variables because the computation happens as late as in the assignment operator (`operator=()`), where memory of the matrix `A` is already available. Eigen also performs explicit vectorization so that features of modern processors[25] are properly utilized for maximum performance.

Supported features include all common algebraic operations, linear problem solving, a variety of decompositions (Cholesky, LU, QR, SVD) and methods for solving eigenvalue problems. Large part of this functionality is also provided for sparse matrices, for which a custom efficient storage format is noted. Sparse matrices however present no advantage for our software solution and are thus not further studied.

The basic `Eigen::Matrix` type has six template parameters (many of them implicit if unspecified) and allows programmer to specify individual dimensions either statically (at compile time) or dynamically (at runtime). Base scalar type and dimensionality (tensor order) are always specified statically, as in Cython memoryviews. Convenience subclasses are provided for row and column vectors (that simply fix row or column count to 1, respectively). To share code without the overhead of virtual C++ methods, Eigen exploits so-called *Curiously Recurring Template Pattern*, where a base class takes the type of the derived class as its template parameter:

```
template<class Derived>
class Base {
    (...)
};

class Derived : public Base<Derived> {
```

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

[24] deferring of the actual computations to the point where they are first really needed.

[25] like the SSE or AVX extensions.

```
        (...)
};
```

Thanks to this strangely-looking construct the `Base` class can statically access methods and attributes of the `Derived` class, which allows the C++ compiler to perform unprecedented range of optimizations. With this mechanism, element-wise complex conjugate operation on a real matrix becomes a true no-op *implicitly*, without any help from application or library developer.

Thanks to this approach, Eigen's single-threaded performance surpasses all freely available BLAS/LAPACK implementations and matches state-of-the-art proprietary ones[26] like Intel MKL[27] or ACML.[28] Since Eigen 3.1.0 an optional support is provided for transparently using Intel MKL for functions and problem sizes that are known to be computed more efficiently using it.

Mapping Type

For our potential use of implementing linear algebra on top of Cython memoryviews, a subclass of the fundamental `Eigen::Matrix` type comes into mind: the `Eigen::Map` class enables one to use any piece of memory anywhere where `Eigen::Matrix` is accepted if storage layout of the underlying numeric array is known. `Eigen::Map` has the following signature:

```
template<typename PlainObjectType, int MapOptions,
         typename StrideType> class Map
    : public MapBase<Map<PlainObjectType, MapOptions,
                         StrideType> >,
```

where `PlainObjectType` is the base scalar type, `MapOptions` allow to specify additional flags like alignment constraints and `StrideType` determines layout of the stored data. To illustrate: when a matrix element at row i and column j is to be accessed, `data` is a pointer to first element of the array and `inner`, `outer` being inner, respectively outer strides, following code holds:

```
PlainObjectType *data;
(...)
PlainObjectType element = data[outer*i + inner*j];
```

[26] http://eigen.tuxfamily.org/index.php?title=Benchmark

[27] Intel Math Kernel Library, https://software.intel.com/en-us/intel-mkl

[28] AMD Core Math Library, http://developer.amd.com/tools-and-sdks/cpu-development/amd-core-math-library-acml/

This example applies to row-major (C-style) matrices. Eigen again allows its users to specify inner and outer stride statically or dynamically (independently from each other).

Important special cases exist: when the inner stride is 1, the array is said to be C-contiguous (neighboring elements in a row are adjacent to each other in memory); when outer stride is 1, the array is said to be Fortran-contiguous (neighboring elements in a row are adjacent to each other in memory).[29] Computations using Eigen are only efficient in case of C- or Fortran-contiguous arrays whose inner, respectively outer, stride is statically specified (to 1). This is an inherent limitation of the vectorization support in processors and cache locality principle; Eigen merely allows the general case of non-contiguous arrays. We note that this maps perfectly to Cython memoryviews that can also be non-contiguous.

3.2.4 PyBayes

PyBayes is, as the name suggests, a Python library for recursive Bayesian estimation written by the author of this text as part of the Bachelor thesis[13] and actively maintained since. *This section describes state of PyBayes* at the time when work on this thesis began. It was modified and enhanced as part of this thesis, so this section is necessarily outdated. Changes that have been done to PyBayes are presented in later sections; past tense is used here for statements that no longer hold in their entirety as a result of these modifications.

PyBayes is developed publicly using git[30] version control system on GitHub project hosting service at the address `https://github.com/strohel/PyBayes`. PyBayes is completely documented,[31] and extensively unit-tested[32] (with a total of 124 unit test at the time of writing this; the test-suite is auto-run on public Travis CI[33] server upon every git push). PyBayes released under the GNU GPL[34] v2+ strong copyleft open-source license and is compatible with both Python 2 and Python 3.

As mentioned earlier, PyBayes can be used in two ways:

1. as an ordinary Python library with very simple installation and no dependencies apart from NumPy, or

[29] in fact, Eigen allows to express whether the storage order is row-major or column major, specifying the other storage variant has the same effect as swapping inner and outer stride.
[30] `http://git-scm.com/`
[31] `https://strohel.github.io/PyBayes-doc/`
[32] `https://travis-ci.org/strohel/PyBayes`
[33] Continuous Integration.
[34] GNU General Public License, `https://www.gnu.org/licenses/gpl-2.0.html`

2. compiled using Cython into optimized binary Python modules, which makes installation slightly more complicated, but brought 30% to 200% speed improvements (depending on methods used and problem sizes).

This dual behavior is achieved without code duplication using the *pure Python* Cython mode, where implementation is put into valid and functional `.py` files (Python modules) and extra declarations that allow Cython to do its optimizations are separated in corresponding `.pxd` augmenting files. Such approach not only lowers the barrier of trying and getting started with PyBayes, but perhaps more importantly it preserves compatibility with tools that expect Python source code, in particular the *coverage*[35] test coverage analysis tool and the *Sphinx*[36] documentation generation system.

The Python build uses NumPy `ndarray` of Python `floats`[37] is its base type, while the Cython build used the legacy direct Cython support for NumPy `ndarrays` as its default vector (and matrix) type — development of PyBayes predates introduction of Cython memoryviews, so no viable alternative was available at the time. Efficient algebra with vectors and matrices in the Cython build was implemented using the *Tokyo* BLACS/LAPACK wrapper, which was bundled as a git submodule in the source code repository for convenience.

We continue by brief presentation of actual functionality that the PyBayes library offers for solving problems of recursive Bayesian estimation: the way how random variables are represented and why, the framework of probability density functions and finally the available Bayesian filtering methods. Next sections do not form a complete reference to PyBayes and omit some classes, they merely pick up classes and methods that were deemed important when considering its involvement and enhancement.

Random Variable Meta-Representation

While PyBayes was designed to avoid symbolic computing where feasible, general probability density function products like

$$p(a, b|c) = p(a|b)p(b|c)$$

had to be expressed. A concept of random variables and their components was designed to facilitate this. The two classes can be found in the `pybayes.pdfs`[38] module because of their closeness to probability density functions.

[35] http://nedbatchelder.com/code/coverage/
[36] http://sphinx-doc.org/
[37] Python `float` type corresponds to C `double` type.
[38] pdfs stands for probability density functions.

`RVComp` class represents a component of a random variable, for example a in the example above. It has one mandatory parameter, `dimension` (integer), which denotes dimension of this component and an optional parameter `name` (string), which is only used for display purposes.

`RV` class represents a random variable, and is in fact just a sequence of `RVComp`s coupled with convenience attributes `name` and `dimension` that cumulate child components. It also provides a couple of helper methods:

`contains(component)` returns true when component is part of this random variable.

`contains_all(components), contains_any(components)` convenience extensions of `contains()`.

`contained_in(components)` method dual to `contains()`, return true if this random variable is entirely contained in `components`.

`indexed_in(super_rv)` returns index array that can be used to select all components in this random variable within `super_rv`, which must be its superset.

Two `RVComp`s are considered equal in comparisons within `RV` if and only if they reference the same instance. This is best shown on an example of interactive Python session:

```
>>> rv = RV(RVComp(1, "a"))
>>> ...
>>> rv.contains(RVComp(1, "a"))
False
```

Such semantics may seem unexpected for new users, but Python's behavior to reference, rather than copy, object in assignments makes this approach convenient. It also avoids problems of an alternative to compare random variable components by their names: possible unwanted collisions and slower comparisons.

Random variables are of no use on their own, they become useful when attached to probability density functions as shown in the next section.

Probability Density Functions

Recursive Bayesian estimation is built on probability density functions, and so is PyBayes. Its framework of distributions forms large part of the library; it is implemented in the `pybayes.pdfs` module. All distributions are assumed to be multivariate (even ones that are limited to one dimension, for symmetry) and represent

their variables as vectors. The physical type employed for vectors depends on used build type of PyBayes and is denoted simply as `vector`. Following Python syntax, superclasses of presented classes are marked in parentheses.

`CPdf` is the fundamental prototype (abstract base class) for all possibly conditional probability density functions. Strict abstractness is in fact not abode as `CPdf` contains 2 attributes:

- **`rv`** stores the random variable (`RV`) that is associated with this distribution. Subclasses take `rv` as an optional constructor argument; if it is not specified, conventions dictate that a random variable with one anonymous component of suitable size is created.
- **`cond_rv`** is an analogy of `rv` that stores the *conditioning* random variable. Similar rules for `cond_rv` construction apply, with the difference that empty random variable (with no components) is created in case of unconditional probability density functions.

the main purpose of `CPdf` is however to declare method prototypes that its subclasses implement:

- **`shape() → int`** returns shape of the random variable (number of its dimensions); subclasses are encouraged to use the default implementation which simply returns `rv.dimension`.
- **`cond_shape() → int`** is analogous to `shape()`, but applies to condition instead. Similarly, default implementation returns `cond_rv.dimension`.
- **`mean(cond=None) → vector`** returns mean value assuming the the conditioning variable takes value `cond` (which is of type `vector`). In case of unconditional distributions, cond does not need to be specified, hence the default value of `None`.
- **`variance(cond=None) → vector`** computes variance (diagonal elements of covariance) of this distribution. `cond` is again ignored for unconditional probability density functions.
- **`eval_log(x, cond=None) → double`** return logarithm of the (conditional) likelihood function in point x. Logarithm is used instead of untransformed value in order to achieve more accurate representation of near-zero values and to enforce non-negativeness. It also allows simpler formulas for distributions of the exponential family.
- **`sample(cond=None) → vector`** draws one sample from this probability density function (given condition `cond`).

samples(n, cond=None) → matrix convenience helper to draw n samples.

Pdf(CPdf) is a thin subclass of `CPdf` that forms a base class for unconditional probability density functions. It implements just `cond_shape()` so that 0 is returned quickly and one convenience helper method for subclasses.

UniPdf(Pdf) is a straightforward implementation of the multivariate uniform density on interval $\langle \mathbf{a}, \mathbf{b} \rangle$

$$f(\mathbf{x}) = \Theta(\mathbf{x} - \mathbf{a})\Theta(\mathbf{b} - \mathbf{x}) \prod_{i=1}^{n} \frac{1}{b_i - a_i}.$$

AbstractGaussPdf(Pdf) is a container to store mean value μ and covariance matrix R.

GaussPdf(AbstractGaussPdf) represents multivariate normal distribution

$$f(\mathbf{x}) \propto \exp\left(-\left(\mathbf{x} - \mu\right) R^{-1} \left(\mathbf{x} - \mu\right)\right).$$

LogNormPdf(AbstractGaussPdf) models log-normal probability density function. I.e. the distribution of random variable Y when

$$Y = \exp(X); \quad X \sim \mathcal{N}(\mu, R).$$

AbstractEmpPdf(Pdf) is a base class for weighted empirical probability density functions. It has one attribute, `weights` (`vector`).

normalise_weights() multiplies weights by appropriate constant so that $\sum \omega_i = 1$, where ω_i is weight of i^{th} particle.

get_resample_indices() calculates indexes of the "new" particles in the "old" particle array if resampling step of the particle filter is to be performed. Roughly speaking, each particle would be copied approximately n-times according to this index array, where n is the ratio of particle weight ω and uniform weight $1/N$ (N being particle count). This method does not change the distribution itself in any way it is merely a helper for subclasses.

EmpPdf(AbstractEmpPdf) implements the standard weighted empirical probability density function as needed by the particle filter,

$$f(\mathbf{x}) = \sum_{i=1}^{n} \omega_i \delta(\mathbf{x} - \mathbf{x}^{(i)}),$$
$$\sum \omega_i = 1,$$

where $\mathbf{x}^{(i)}$ is the value of i^{th} particle.

`resample()` performs actual resampling as described in chapter 1: particles are re-indexed with the help of `get_resample_indices()` and copied as needed, weights are reset to uniform ($\forall i \in N \quad \omega_i := \frac{1}{N}$).

`ProdPdf(Pdf)` represents a product of factor probability density functions, which need to be independent:

$$p(a, b, c) = p(a)p(b)p(c).$$

This way one can represent for example a 2-dimensional random variable which is uniform along first axis and Gaussian along second axis

```
import pybayes as pb

prod = pb.ProdPdf([pb.UniPdf(...), pb.GaussPdf(...)]).
```

`MLinGaussCPdf(CPdf)` is a conditional normal distribution whose mean value μ depends on the condition through affine transformation:

$$f(x|c) \propto \exp\left(-(x-\mu)' R^{-1} (x-\mu)\right),$$

where $\mu = Ac + b$ and c is the value of conditioning variable. `MLinGaussCPdf` takes optional parameter `base_class` (`AbstractGaussPdf` subclass) which can be used to specify alternate base probability density function to work on. This way one can create e.g. log-normal conditional density by passing `LogNormPdf` as `base_class`.

`LinGaussCPdf(CPdf)` models similar, but univariate, conditional normal distribution whose mean value and variance depend on 2-dimensional conditioning variable so that

$$f(x|c_1c_2) \propto \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right),$$

where $\mu = ac_1 + b$ and $\sigma^2 = cc_2 + d$. `LinGaussCPdf` accepts `base_class` optional argument (which again defaults to `GaussPdf`) in a manner similar to `MLinGaussCPdf`.

`GaussCPdf(CPdf)` is the most general one-dimensional normal probability density function available in PyBayes. It exploits Python's "function is a first-class citizen" principle and takes 2 functions f and g to perform general transformation of the condition so that

$$\mu = f(\mathbf{c}),$$
$$R = g(\mathbf{c}),$$

where **c** is the conditioning variable. `GaussCPdf` also takes `base_class` optional argument as described above.

`ProdCPdf(CPdf)` represents general conditional product (application of the chain rule for probability density functions) of 2 or more (conditional) factor distributions. An example to describe

$$p(x_1 x_2 | y_1 y_2) = p(x_1 | x_2 y_1) p(x_2 | y_2 y_1)$$

can look like this in the code:

```
# prepare random variable components:
x_1, x_2 = RVComp(1), RVComp(1, "name is optional")
y_1, y_2 = RVComp(1), RVComp(1, "but recommended")

p_1 = SomePdf(..., rv=[x_1], cond_rv=[x_2, y_1])
p_2 = SomePdf(..., rv=[x_2], cond_rv=[y_2, y_1])
p = ProdCPdf((p_2, p_1), rv=[x_1, x_2],
             cond_rv=[y_1, y_2]).
```

`ProdCPdf` then constructs the data-flow automatically (with the help of utility methods of the `RV` class) and raises an exception if this is impossible to do.

Bayesian Filters

With frameworks modeling probability density functions and associated random variables in place, the road is paved to construct Bayesian filters. As it is common in PyBayes, there is abstract filter prototype and individual implementations. All filters are found in the `filters` module of PyBayes.

`Filter` represents a Bayesian filter, i.e. any solution of the recursive Bayesian estimation problem as defined in section 1.1 or its approximation. Filters usually take a description (of some form) of the transition and observation probability density functions as constructor parameters.

`bayes(yt, cond=None)` performs the Bayesian update, i.e. the transition from $t-1$ to t, taking observation at time t, which is passed through the `yt` (`vector`) argument, into account. For flexibility, optional argument `cond` (`vector`) can be also supplied. Its meaning (as well as whether it is required or not) and depends on individual subclasses.

`posterior() → Pdf` returns posterior probability density function: $p(x_t|y_{1:t})$ however with $y_{1:t}$ already substituted by values passed to preceding `bayes()`

calls so that it takes unconditional form $p(x_t)$. One might expect that posterior distribution would be returned right by the `bayes()` method, but this is not done because constructing posterior may spend some additional resources and some callers may not need to know it after each `bayes()` call. It also makes calling code more explicit, which considered as more "Pythonic" way of designing software.[39] Subclasses are allowed to return a reference (for performance reasons) to their internal attribute, it is therefore forbidden to modify the returned object.

`evidence_log(yt) → double` returns logarithm of the *marginal likelihood* (sometimes also called *evidence*)[17] of the observation y_t, which is passed as `yt` (`vector`), given previous observations $y_{1:t-1}$, which are taken from preceding `bayes()` calls (excluding the most recent one), i.e.

$$p(y_t|y_{1:t-1}) = \int p(y_t|x_t)p(x_t|y_{1:t-1})\,\mathrm{d}x_t.$$

This quantity can be used as a measure of confidence in the filter. Logarithm is returned instead of plain value for symmetry with `CPdf.eval_log()` method. As closed formula for this quantity may not be known, subclasses are not required to implement this method.

`KalmanFilter(Filter)` implements the Kalman filter[10] (formulated in language of recursive Bayesian estimation). To make PyBayes suitable also for decision-making applications, control vector can be optionally specified through the `cond` argument of the `bayes()` method (along with related matrix parameters in the constructor). `GaussPdf` is (naturally) used as a posterior distribution and evidence_log() is implemented.

`ParticleFilter(Filter)` implements the particle filter as presented in section 1.2. Before it was enhanced as part of this thesis, `ParticleFilter` did not allow specifying the *proposal* distribution, i.e. it implemented only the *bootstrap* variant of the particle filter which cannot take most recent observation into account when sampling new particles. It also did not support specifying custom empirical probability density function — `EmpPdf` was used unconditionally, because the implementation was tied to it. `ParticleFilter` constructor took number of particles N (`int`), initial probability density function $p(x_0)$ (`Pdf`) to sample starting particles from, transition density $p(x_t|x_{t-1})$ and observation density $p(y_t|x_t)$ (both `CPdf`) as its arguments.

[39] http://www.python.org/dev/peps/pep-0020/

Linear Algebra Wrappers

We finish the section about PyBayes with brief mention of its `wrappers` module.[40] While it is not intended to be used externally and can be considered an implementation detail, it is a workhorse of dual nature (Python-interpreted and Cython-combined) of PyBayes.

Instead of using NumPy functions and data types directly, other PyBayes components use shim layers found in the `wrappers` package: `_numpy`, which mimics (to certain extent) the `numpy` module of NumPy project, and `_linalg`, which mimics the `numpy.linalg` module of NumPy (that contains more specialized linear algebra functions). Unlike other PyBayes components, modules in the `wrappers` package have separate implementation for each of the 2 PyBayes build types, e.g. `_numpy.pxd` + `_numpy.pyx` for Cython build and `_numpy.py` for Python build. The PyBayes build-system ensures that correct implementation is picked up.

The Python versions of wrapper modules had straightforward implementation, they just imported appropriate NumPy objects and defined aliases that were needed to make static typing in Cython possible. What follows was an actual implementation (with non-crucial lines omitted) of the `_numpy` wrapper:

```
from numpy import *

dotvv = dot  # vector-vector dot product
```

Here, `dot()` was aliased to `dotvv()` as the code differentiated between vector-vector dot product, which returns a scalar, and other dot products that result in an `ndarray`.

Cython versions contained more logic. First, they defined the base vector and matrix type, `ndarray`, through a special Cython import that ensured that direct `ndarray` support in Cython kicked in. Fast implementations of frequently-called algebraic functions were provided through the Tokyo wrapper for system BLAS/LAPACK libraries. These were `dot()` and `dotvv()` in the `_numpy` wrapper and `inv()`[41] in the `_linalg` wrapper. It should be noted that these implementations were less than elegant in some cases. For example the `dot()` implementation had multiple code paths for various matrix/vector and C-contiguous/Fortran-contiguous combinations and simply failed for non-contiguous vectors and matrices. Other NumPy functionality that PyBayes used was simply imported from NumPy, which incurred the inevitable Python overhead.

[40] in fact a package in Python nomenclature, https://docs.python.org/distutils/introduction.html

[41] matrix inversion.

3.3 Analysis Results

The aim of this section is to draw conclusions from feasibility analysis of the projects presented above and to suggest high-level structure of the desired software solution. This structure is visualized in Figure 3.1 on page 50.

3.3.1 Asim

We propose that a separate project that focuses on radioactive pollutant atmospheric dispersion assimilation is created. This project, which became known as Asim, would consist of two main parts: dispersion model part and assimilation part. The *dispersion model* part should contain:

- Dispersion model prototype, an interface that could be implemented by any model that simulates radioactive pollutant release into the atmosphere. Any such model should be able to simulate development of spacial pollutant concentration in time given meteorologic situation and release conditions. Additionally it should be able to compute absorbed doses at given points of interest and time intervals. Initial implementation would be puff-based dispersion model as presented in chapter 2.
- Meteorologic model prototype that would provide all necessary information to the dispersion model. Such abstraction would allow for different sources of meteorologic data to be plugged in when performing simulation of assimilation.
- Release model prototype which would supply data such as release location, rate and type of the radioactive pollutant (which determines parameters such as decay half-life and released energy).
- A simple runner that would perform simulation with predefined meteorologic and release data and visualize the release.

The *assimilation* part should combine the dispersion model with the particle filter from PyBayes to create the assimilation algorithm. The transition and observation models need to be formulated in terms of PyBayes-compatible probability density functions. It is assumed that PyBayes' particle filter can be enhanced to support proposal density, so that again needs to be expressed in a suitable manner. It is expected that subclasses of some PyBayes classes will have to be implemented; if these are general enough to be useful for other PyBayes users, we suggest to add such classes directly to PyBayes, otherwise to Asim.

As planned usage of PyBayes implies, Asim is to be implemented in Cython-enhanced Python implementation environment, which proved very well suitable for intended use in case of PyBayes. While the stress does not need to be put so much on ease of first use, we still propose that *pure Python* approach is used where feasible: the implementation should be done in valid Python `.py` files and Cython-specific enhancements should go into separate `.pxd` files. The motivation for this is *a)* compatibility with existing Python development tools (IDEs,[42] documentation extractors such as Sphinx, testing and test coverage tools...) and *b)* possibility that projects like PyPy and/or Numba[43] will render use of Cython unnecessary in the future.

As the Asim project is rather specific in its purpose, we do not see any advantage in developing it in public. Of course, any improvements into its upstream projects that are already openly developed that arise during Asim development should be propagated to these projects for public benefit.

3.3.2 PyBayes Improvements

While PyBayes was found generally suitable as a recursive Bayesian estimation backend for dispersion model assimilation, it needs to be enhanced in a couple of areas for perfect fit. It was identified that truncated normal, gamma and inverse gamma distributions have to be added along with conditional variants. The conditional variants should model the case where the condition specifies their mean value and their standard error is proportional to the mean value.

Another area that requires improvement is the `ParticleFilter` class. It needs to be extended to allow using custom proposal distribution (which is crucial in cases like ours where the observation density is significantly narrower than the transition density). Another modification arises from a special form of our state variable: while only 3 dimensions are actually assimilated, the full state variable consists of much more (deterministic) variables whose count can grow in time. We suggest that such complexity is hidden in a custom `EmpPdf` subclass (which would too specific to be useful outside Asim); `ParticleFilter` should be therefore modified to allow specification of `EmpPdf` subclass to work with. It is also proposed that the transitioning step (when $x_t^{(i)}$ is sampled using $x_{t-1}^{(i)}$ and $q(x_t|y_t, x_{t-1})$) is delegated to `EmpPdf` (as it is already done with the resampling and weight normalizing step) so that the special cases like ours can be handled.

[42] Integrated Development Environments.

[43] http://numba.pydata.org/

3.3.3 Ceygen

Due to recent developments in Cython compiler and language, most prominently its support for typed memoryviews, fused types (lightweight templating) and advancements in the C++ support, we opinion that implementing vector/matrix algebra using NumPy `ndarray`s and resorting to Tokyo/BLAS/LAPACK solution for speeding it up should be considered obsolete.

Memoryviews come with fewer limitations (such as ability to appear as class attributes, in global scope), are more parallelization-friendly due to reduced need of holding the GIL and have been shown to bring better performance in general. With the array expressions project unfortunately not materializing, only roadblock left was unavailability of linear algebra operations on top of them. After finding that the Eigen C++ library can be interfaced through Cython with minimal overhead and a proof-of-concept test implementation, we became convinced that implementing a linear algebra library for typed memoryviews using Eigen would be well possible with reasonable effort.

The project was named Ceygen (a rather poor wordplay on Cython + Eigen) and turned out a success. Its design and implementation is presented in the next chapter, so are the Asim project and improvements to PyBayes. It was deemed appropriate and advantageous to develop Ceygen[44] publicly and release it under a free open-source license. Rationale being *a)* to give back to Cython community *b)* to attract testers and possible contributors so that maintenance and development is distributed.

It was decided that Cython build of PyBayes will be ported to use memoryviews + Eigen, dropping the dependency on Tokyo and BLAS/LAPACK along the way. Beyond the `wrappers` module, which was rewritten, only small changes (such as changing method signatures and some function calls) had to be done to functional parts of PyBayes. Asim project then inherently has to use the same vector/matrix type.

[44] which could be useful to anyone who performs linear algebra with Cython memoryviews.

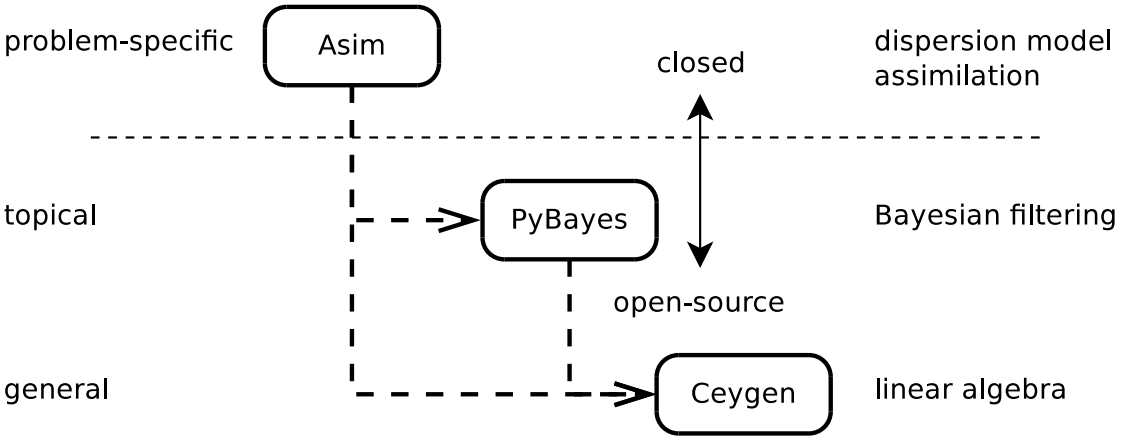

Figure 3.1: High-level overview of Asim, PyBayes and Ceygen projects. Dashed lines with arrows mark dependencies, e.g. Asim depends on PyBayes and Ceygen.

Chapter 4

Design and Implementation

In this chapter, we present the design of developed software components, their implementation and rationale behind it. The design builds on suggestions made in the previous chapter with occasional enhancements aimed mainly to make the software more future-proof or efficient. It is shown how the software stack is built from the bottom up; the chapter starts with the Ceygen linear algebra library that has been developed to improve performance, continues with description of enhancements made to PyBayes and finalizes with presentation of the top-level project Asim. We consider the implemented software one of the main results of this thesis.

In the whole chapter a Python-like syntax for class and method description is used along with function annotations as defined by PEP 3107:[1] `ClassA(ClassB)` denotes that `ClassA` is a subclass of `ClassB` and `func(p : int) → bool` denotes a function/method `func` with parameter `p` of type `int` that returns a boolean value. Functions with no return value mentioned do not return semantically valuable information; they are usually designed to return `True` to allow for efficient exception passing when compiled by Cython.[2]

4.1 Ceygen

Ceygen was developed to fill the missing gap in usability of Cython typed memoryviews for numerical software: linear algebra on top of them. Per software analysis from chapter 3, the Eigen library (combined with recent improvements to C++

[1] https://www.python.org/dev/peps/pep-3107

[2] when functions with no return value are called, Cython has to use Python API function to check to possible raised exceptions every time, which induces overhead: http://docs.cython.org/src/userguide/language_basics.html

support in Cython) was identified as a perfect back-end to perform the actual computations. Eigen is a compile-time only dependency of Ceygen.

Ceygen is being publicly developed and actively maintained by the author of this text using the git version control system at the GitHub source code repository at https://github.com/strohel/Ceygen. It is released under a free and open-source GNU GPL v2+ license and packages of released versions are available through the PyPI Python package repository at https://pypi.python.org/pypi/Ceygen.

It should be noted that Ceygen is not useful for interpreted Python code, for which using NumPy is, in our opinion, totally appropriate given already present interpreter and dynamic-typing overhead. It is rather designed to be used from Cython-compiled code (which should be enhanced with type declarations) that requires high performance and/or ability to parallelize. In fact, Ceygen functions are only visible from Cython-compiled code to prevent such ill-optimization.

4.1.1 Achieved Design Goals

Design goals, which have been fulfilled, prescribe that Ceygen...

is fast: Ceygen would be of no use if it was not highly performant; it does not implement functionality that is *not available*, it implements functionality that is *not available without considerable overhead*, especially for small to moderate-sized matrices and vectors. For this reason, every Ceygen function has a code-path where no Python function is called, no memory is allocated on heap and no data (beyond auxiliary variables of fixed size) is copied.

is documented: To be usable for its users without the need to study the source-code, software must be properly documented. Every public Ceygen method is therefore described in detail in its reference manual, which is generated using Sphinx from out-of-source documentation[3] and is available on-line at https://strohel.github.io/Ceygen-doc/ or on the enclosed DVD-ROM.

works with a range of scalar types: Thanks to *fused types* support in Cython and templated nature of Eigen, Ceygen can be designed in a way that various data types are supported using single implementation. Types offered out-of-the box are `char`, `short`, `int`, `long`, `float` and `double`; complex type support is planned.

[3] generation from in-source declarations is not currently possible due to Cython-enhanced syntax beyond the Python language.

is tested: Every public function has a unit test that assures its correctness and also proper behavior when invalid input is passed. At the time of writing, the test-suite consists of 98 individual tests (of which 11 are benchmarking ones). Thanks to Travis CI, a publicly available continuous integration server, the test-suite is run every time under multiple Python versions when source code is changed.[4]

supports multi-threading: Every Ceygen function does not acquire the GIL unless it needs to create a Python object (which is always avoidable); all functions are declared `nogil`[5] so that they can be called in `prange`[6] blocks without losing parallelism.

reports errors consistently: Care is taken to propagate all errors properly (down from Eigen) to fulfill semantics expected by Python programmers. Ceygen functions do not crash on invalid input, but rather raise reasonable errors. Steps taken to achieve this are described below.

is compatible with NumPy, but does not depend on it at compile or run-time (although its test suite uses it). Cython makes memoryviews and NumPy `ndarray`s fully interchangeable (without copying data, but with some overhead), so Ceygen can be plugged into an existing project without need to modify all code paths — just those performance-critical.

In order not to make false expectations, we find it fair to mention that Ceygen...

is not an interface to Eigen: while Eigen is used internally, Ceygen does not wrap Eigen types for others to use.

does not bring Eigen elegance to Python: Because of the way how Ceygen is implemented, it does not support lazy evaluation or pseudo-code-like expressions that make Eigen beautiful when used in C++ code.

does not help code readability: Because Ceygen does not integrate with memoryviews at Cython level,[7] expressions where operators would be normally used have to be written as function calls. Users therefore have to find a trade-off between readability and performance.

To illustrate usage of Ceygen, we show a sneak peek from its README file:

[4] https://travis-ci.org/strohel/Ceygen
[5] a special Cython-specific decorator that marks the function as callable with the GIL released.
[6] parallelized for loop, a multi-threading construct provided by Cython.
[7] which would be doable, but out of scope of this text.

```
>>> cimport ceygen.core as c
>>> cdef double[:, :] big = np.array(
>>>                     [[1.,  2.,   2.,  0.,   0.,  0.],
>>>                      [3.,  4.,   0., -2.,   0.,  0.]])
>>> c.dot_mm(big[:, 0:2], big[:, 2:4], big[:, 4:6])
[[ 2. -4.]
 [ 6. -8.]]
>>> big
[[ 1.  2.   2.  0.   2. -4.]
 [ 3.  4.   0. -2.   6. -8.]]
```

The `dot_mm()` call above does not copy any data, allocates no memory on heap and does not need need the GIL. If the passed array was bigger, it would use vectorization (SSE, AltiVec) if the processor architecture it is compiled for supported it.

4.1.2 Interfacing Eigen

As mentioned above, Ceygen uses `Eigen::Map` class to make Cython memoryviews accessible from Eigen. More specifically, Eigen::Map is subclassed (C++ -wise) in Ceygen file `eigen_cpp.h`. Template arguments are omitted for brevity and mentioned in description if significant:

`BaseMap(Eigen::Map)` exists to overcome Cython C++ support limitation that all instantiated classes must have a default constructor. Its default constructor sets Eigen:Map values to null as nothing else is available at that time. It has 2 template parameters, `BaseType` to specify underlying Eigen type (matrix, vector and its scalar type, `Scalar`) and `StrideType` to specify striding (Fortran- or C-contiguity) at compile-time. The initialization is done by a later called method:

`init(data : Scalar*, shape : const int*, strides : const int*)` sets vital `Eigen::Map` variables extracted from Cython memoryview base pointer `data`, `shape` (number of rows and/or columns) and `strides` (memory layout along axes; contiguity). It uses a trick with placement new operator (as `Eigen::Map` otherwise does not allow late initialization).

Other convenience subclasses are provided with no implementation that just transform their template parameters. All following classes take `dtype` template argument to specify base scalar type (denoted as `Scalar` in `BaseMap`) and `ContiguityType`

template argument, which is our way to specify compile-time striding (whether individual axes are contiguous or specified as run-time) and storage order (column-major or row-major):

`VectorMap(BaseMap)` fixes `BaseType` to $n \times 1$ `Eigen::Matrix` (column vector) for matrix-based operations.

`RowVectorMap(BaseMap)` fixes `BaseType` to $1 \times n$ `Eigen::Matrix` (row vector).

`Array1DMap(BaseMap)` fixes `BaseType` to $n \times 1$ `Eigen::Array` (column vector) for element-wise operations.

`MatrixMap(BaseMap)` fixes `BaseType` to $m \times n$ `Eigen::Matrix` for matrix-based operations such as dot product.

`Array2DMap(BaseMap)` fixes `BaseType` to $m \times n$ `Eigen::Array` for element-wise operations with matrices.

With C++ classes being prepared, last step is to make them known to Cython. As it cannot parse C or C++ header files itself, signatures and interfaces have to be communicated to it using its own syntax. The `eigen_cython.pxd` file in the source code does exactly that, it declares `BaseMap` and its defined methods and operators (including the ones that are inherited from Eigen)[8] and mentions that `VectorMap`, `RowVectorMap`, ..., `Array2DMap` are its subclasses. This suffices to make the whole machinery accessible from Cython-compiled Python code.

4.1.3 Contiguity-based Dispatching

Because Cython memoryviews can be both contiguous (in row or column sense) or noncontiguous, but efficient computation is only possible with the contiguous, it was decided that Ceygen should support both without incurring performance penalty in the favorable case.

Because of that, every computational Ceygen function is divided into 2 parts: a wrapper and worker, which takes contiguity type of its arguments as a template parameter. Wrapper just prepares the arguments and calls the worker through a so-called *dispatcher*, which determines what variant of worker is to be called depending on runtime inspection of passed vector and matrix layouts. Scary-looking implementation of dispatchers in `dispatch.h`[9] has in fact little effect on performance,

[8] Cython does not require to declare all methods, just the ones used.
[9] which is again presented to Cython through `dispatch.pxd`.

because their code is embedded directly into wrappers by the compiler, furthermore function pointer assignments are used instead of deep branching involving function calls. Thanks to this approach, dispatcher code adds only tens of instructions into wrapper function object code when compiled using optimizing C++ compiler.[10] We also expect that CPU branch predictors to be reasonably effective in this case.

All this is best shown on an example implementation of matrix dot product in Ceygen:

```
cdef void dot_vv_worker(
        dtype *x_data, Py_ssize_t *x_shape,
        Py_ssize_t *x_strides, XVectorContiguity x_dummy,
        dtype *y_data, Py_ssize_t *y_shape,
        Py_ssize_t *y_strides, YVectorContiguity y_dummy,
        dtype *o) nogil:
    cdef VectorMap[dtype, XVectorContiguity] x
    cdef VectorMap[dtype, YVectorContiguity] y
    x.init(x_data, x_shape, x_strides)
    y.init(y_data, y_shape, y_strides)
    o[0] = x.dot(y)

@cython.boundscheck(False)
@cython.wraparound(False)
cdef dtype dot_vv(dtype[:] x, dtype[:] y) nogil except *:
    # vector x vector -> scalar operation dispatcher:
    cdef VVSDispatcher[dtype] dispatcher
    cdef dtype out
    dispatcher.run(&x[0], x.shape, x.strides,
                   &y[0], y.shape, y.strides, &out,
                   dot_vv_worker, dot_vv_worker,
                   dot_vv_worker, dot_vv_worker)
    return out
```

We remind that all dispatcher code is actually compiled into `dot_vv()` and that 4 identically-looking passings of `dot_vv_worker` in fact instantiate different contiguity variant each time.

[10] this was verified in the generated assembly code.

Exception Passing

Another purpose of dispatchers is to facilitate exception propagation from Eigen (C++) to Python caller. First, a macro `eigen_assert(statement)` is defined before including Eigen in `eigen_cpp.h` to code that throws a standard C++ library exception instead of aborting the program being run (which is the default Eigen behavior and which would be unexpected by Python programmers).

Second, `run()` methods of all dispatchers are decorated using special `except +` flag which signals Cython to enclose the call into `try { } catch` block and convert compatible C++ exceptions to Python ones. For performance reasons it is important to do this just once per each function call, so dispatchers are perfect place to do this even though the C++ exception is raised much lower in the stack (the fact that BaseMap methods may raise C++ exceptions themselves is purposefully concealed from Cython so that generated worker code is leaner).

4.1.4 Ceygen Modules

Finally, with all important technical details being thoroughly discussed, we can present useful functionality implemented by Ceygen. Se further divide the the functionality to subsections which correspond to Ceygen modules. This section does not form a reference manual of Ceygen (which is available on-line or on the enclosed DVD-ROM), it only mentions implemented functionality, omitting unimportant functions.

All computational Ceygen functions share a common signature pattern: their suffix determines on what combination of vectors/matrices they operate, e.g. `dot_mv()` is matrix-vector dot product while `dot_mm()` is a matrix-matrix one. This is done because Cython memoryviews encode their dimensionality statically. We believe this does not restrict users significantly as we have only rarely seen numerical code that accepts tensors of varying dimensionality. Disregarding the suffixes, function names mimic those of NumPy to allow for easy back and forth code transitions.

Another pattern is optional `out` parameter (available in all but scalar-returning or in-place operations). Familiar to NumPy users, this parameter allows Ceygen to eliminate significant overhead of creating a new vector/matrix for return value. If specified, Ceygen uses it to fill the operation results and returns it back for symmetry. Some element-wise methods support repeating one of the input parameters in `out`, effectively forming in-place operations.

Core Data-type: `dtype` Module

The base scalar templated type, dtype, is defined in this module. There is also `nonint_dtype`, which omits integer types and is used in functions where integer scalars are not feasible: decompositions, matrix inversion etc. Both are implemented as Cython *fused types*, meaning that all latter functions using them are generated for various scalars: `float`, `double`, `int` and so on. In theory, adding support for a new type (for example complex ones) to Ceygen involves just extending `dtype` and providing implementation for helper methods to create new instances:

`vector(size : int, like : dtype*) → dtype[:]` creates a new vector; `like` is a dummy parameter used to specify desired base type. Individual components are uninitialized.

`matrix(rows : int, cols : int, like : dtype*) → dtype[:, :]` similarly creates a new matrix.

Linear Algebra: `core` Module

The `ceygen.core` module implements matrix dot product for various matrix/vector combinations. As Cython memoryviews do not differentiate between row and column vectors (contrary to Eigen), Ceygen assumes that vectors are transposed in a way so that operation result is of smaller dimensions. $n \times 1$ or $1 \times n$ matrices can always be specified to force desired behavior. Functions in this module are (omitting their obvious arguments): `dot_vv() → dtype`, `dot_mv() → dtype[:]`, `dot_vm() → dtype[:]`, `dot_mm() → dtype[:, :]`.

Element-wise Operations: `elemwise` Module

As not even element-wise operations are provided by Cython for memoryviews (apart from assignment which even supports broadcasting), they are supplied by Ceygen. Unlike more complex operations, element-wise operations allow its users to repeat one of the input arrays in the out parameter to create in-place versions of the methods.

Vector-scalar operations implemented: `add_vs() → dtype[:]`, `multiply_vs() → dtype[:]`, `power_vs() → dtype[:]`. Note that there are no `subtract_vs()` or `divide_vs()`: users are instead simply advised to add opposite and multiply by inverse number, respectively.

Strictly analogous are matrix-scalar functions `add_ms()`, `multiply_ms()`, `power_ms()` that all return `dtype[:, :]` (matrix).

Vector-vector element-wise methods follow similar pattern: `add_vv()`, `subtract_vv()`, `multiply_vv()`, `divide_vv()` all return a vector.

For completeness matrix-matrix functions are listed: `add_mm()`, `subtract_mm()`, `multiply_mm()`, `divide_mm()`.

Decompositions: `lu` and `llt` Modules

Functions based on lower-upper and Cholesky $\left(L \cdot L^T\right)$ decompositions are more interesting. `ceygen.lu` module provides:

`inv(x : nonint_dtype[: :], out=None) → nonint_dtype[: :]` computes inverse of the matrix x using LU factorization with partial pivoting. Produces undefined results on non-invertible matrices.

`iinv(x : nonint_dtype[: :]) → nonint_dtype[: :]` is a similar method that inverts the matrix in-place. This is faster than using `A = ceygen.lu.inv(A)` because the temporary workspace is allocated in C++ domain rather than Python domain.

`det(x : dtype[: :]) → dtype` computes matrix determinant using LU decomposition.

The Cholesky factorization-powered `ceygen.llt` module implements this method:

cholesky(`x : nonint_dtype[: :], out=None) → nonint_dtype[: :]` computes decomposition of the matrix `x` (which must be square, Hermitian and positive-definite) so that $x = \texttt{out} \cdot \texttt{out}^H$ ($\texttt{out}^H$ being conjugate transpose of `out`).

Reductions: `reductions` Module

The last-presented `ceygen.reductions` module contains, at the time of writing, just various variants of summing:

`sum_v(x : dtype[:]) → dtype` returns sum of the vector `x`.

`sum_m(x : dtype[:, :]) → dtype` similarly returns sum of the matrix `x`.

`rowwise_sum(x : dtype[:, :], out=None) → dtype[:]` reduces all rows of the matrix `x` to scalars by summing.

`colwise_sum(x : dtype[:, :], out=None) → dtype[:]` analogously reduces all columns of the matrix `x` to scalars by summing.

4.2 PyBayes Enhancements

Improvements made to the PyBayes Bayesian filtering library can be split into three separate parts: porting its Cython build to use Cython memoryviews and Ceygen for linear algebra, implementing newly needed probability density functions with their conditional variants and extending its particle filter implementation. We present these modifications one by one.

4.2.1 Porting to Cython memoryviews

As soon as Ceygen project proved to be useful, it was decided that Python-compiled version of PyBayes should be ported away from using NumPy `ndarray`s and problematic Tokyo wrapper to new Cython memoryviews with Ceygen backing. The reasons included:

1. relief from dependency on (C interfaces to) system BLAS and LAPACK libraries. The dependency was found to be peculiar roadblock for portability especially when optimized PyBayes was to be run on Apple Mac platforms. Given that standardized BLAS and LAPACK implementations are only Fortran-based, C interfaces to them vary.[11]

2. ability to off-load linear algebra implementation to more appropriate projects. Due to low-level nature of the 2 numerical libraries and in extension of the Tokyo wrapper, PyBayes contained logic that was not deemed appropriate for a topical library.

3. Cython memoryviews have been shown to be more elegant, less limited and more performant than exploiting Cython's ad-hoc support for NumPy `ndarray`s. Sticking to legacy interfaces was found to be a blocker in PyBayes' integration with emerging projects.

The porting itself happened in two steps. First, Tokyo was replaced with Ceygen, but usage of NumPy `ndarray`s was kept. In second step, signatures of methods, types of class attributes and appropriate local variables have been changed to memoryviews. Operators were replaced by functions calls; in-code comments were added so that adverse effects on code readability are minimized.

[11] there are for example at least 2 different C LAPACK interfaces: the one produced by f2c and LAPACKe one, with incompatible method signatures.

This was accompanied by rewrite of the wrapper modules so that Python-only build of PyBayes is not affected. The wrappers became more straightforward, what follows is side-by-side comparison of Cython (`_linalg.pxd`) and Python (`_linalg.py`) version of the `_linalg` wrapper that mocks `numpy.linalg`:

```
from ceygen.lu cimport *     |  from numpy.linalg import *
from ceygen.llt cimport *    |
```

I.e. this wrapper became, due to Ceygen mimicking NumPy, a simple switcher between the 2 implementations. Note that in case of Cython build, the imports are in `.pxd` "header" file so that they are resolved at compile-time. Associated `_linalg.pyx` file is empty.

The _numpy wrapper was changed similarly. Convenience methods `vector()`, `index_vector()`, `index_range()` and `matrix()` to abstract creation of new objects have been added with NumPy-, respectively Ceygen-backed implementations. Similar layer was created for advanced indexing within arrays. This was needed because Python and Cython versions now use different types. Other PyBayes modules now use more specialized function variants such as `add_vv()` and `dot_mm()` — the Python variant of the `_numpy` wrapper aliases them as shown in the following except:

```
add_vv = add
add_mm = add
dot_vv = dot
```

Given comprehensive test and stress-testing suite of PyBayes, the transition could have been made without fear of breaking functionality. Only problem worth mentioning that was faced during the transition was Cython's inability to work with zero-length memoryview matrices of vectors. Logic of some algorithms was therefore changed slightly to avoid these corner-cases.

From user point of view, nothing has changed in the Python version of PyBayes. Only user-facing change of the Cython-compiled build are different (but compatible) return types of methods and increased performance.

4.2.2 New Probability Density Functions

Adding new probability density functions was more straightforward as was sufficient to implement the defined interfaces. Following distributions have been added:

`TruncatedNormPdf(Pdf)` implements uniform truncated normal probability density function, i.e. distribution of Y, where $Y = X|a \leq x \leq b$ and $X \sim$

$\mathcal{N}(\mu, \sigma^2)$. The `sample()` method is implemented using rejection sampling. More efficient implementation is being considered.

`GammaPdf(Pdf)` models gamma distribution $\mathcal{G}(k, \theta)$ with shape parameter k and scale parameter θ:

$$f(x) = \frac{1}{\Gamma(k)\theta^k} x^{k-1} e^{\frac{-x}{\theta}}.$$

`InverseGammaPdf(Pdf)` implements inverse gamma distribution $i\mathcal{G}(\alpha, \beta)$ with shape parameter α and scale parameter β,

$$f(x) = \frac{\beta^\alpha}{\Gamma(\alpha)} x^{-\alpha-1} e^{\frac{-\beta}{x}}.$$

`GammaCPdf(CPdf)` provides conditional `GammaPdf` variant as needed by wind speed transition density. Given fixed relative error coefficient γ and conditioning variable μ, it is tuned to form gamma distribution with mean value μ and standard error $\gamma\mu$.

`InverseGammaCPdf(CPdf)` is inverse analogy of the `GammaCPdf` distribution above. It is needed for absorbed dose and wind speed observation densities. Given fixed γ and conditioning μ, `InverseGammaCPdf` produces inverse gamma probability density function of mean μ and variance $(\gamma\mu)^2$.

4.2.3 Extending PyBayes Particle Filter

Last step to make PyBayes fully usable for dispersion model assimilation was to make its particle filter more flexible and capable.

First, support for the proposal density has been added into `ParticleFilter` class. The proposal density takes the form $q(x_t|x_{t-1}, y_t)$ which coincides with posterior probability density function of a Bayesian filter (which is in PyBayes represented using unconditional probability density function with x_{t-1} and y_t already substituted) and it is evaluated twice in particle filter algorithm (once during sampling, for the second time during weight update). Because of this, it was decided that proposal would be represented by a nested Bayesian filter (the alternative had been a conditional distribution). This has a favorable side-effect that filters can be chained this way. Filter's `bayes()` method is called for each particle and its posterior density is then used for sampling and weight evaluation.

More technical change was to make `ParticleFilter` accept custom implementation of the underlying `EmpPdf` distribution. Such change was needed to accommodate specially-crafted `PuffModelEmpPdf` which is described below in the Asim section.

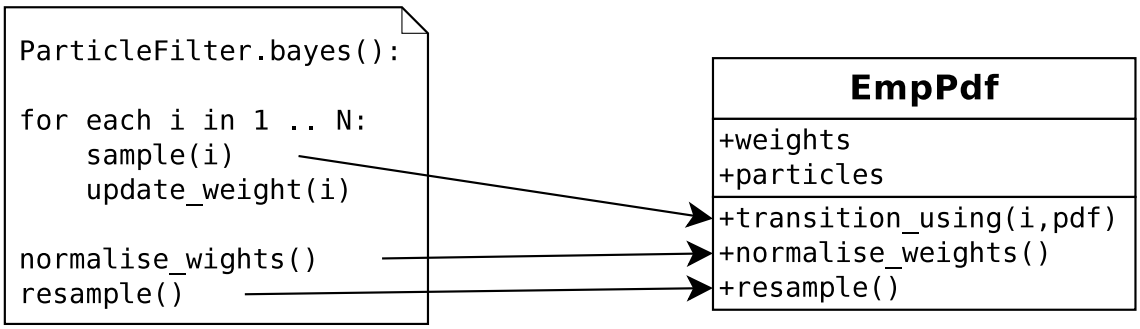

Figure 4.1: `ParticleFiler.bayes()` pseudo-code and interaction with `EmpPdf`

This request has been implemented without increasing number of `ParticleFilter` constructor parameters: if the passed `init_pdf` is already an instance of (a subclass of) `EmpPdf`, it is used directly for the whole lifetime of the `ParticleFilter` instance. Previous implementation used it just to sample initial particles.

Third and last change relates to the second one. Since `ParticleFilter` no longer knowns all internal details of its newly polymorphic `EmpPdf` instance, it delegates more work to it, namely the task to sample i-th particle given a proposal density. This was facilitated by introducing the `transition_using()` method to `EmpPdf` as depicted in Figure 4.1.

4.3 Asim

Asim is the functionally highest-level project presented here. It is written in Cython-enhanced Python language (pure Python mode approach) and makes use of PyBayes for Bayesian filtering, Ceygen to perform linear algebra and employs Cython typed memoryviews as its base vector/matrix type (the basic scalar type being `double`). Asim uses standard Python `distutils` file-system layout and build system (it is intended to be installed as a Python site-package) so that it is compatible with Python package management tools such as *pip*[12] or *easy_install*.[13] Asim documentation is generated using Shinx[14] from directives embedded in code (available on the enclosed DVD-ROM). Asim uses coding style conventions expected by vast majority of Python developers as defined in PEP 8.[15]

[12] http://www.pip-installer.org/en/latest/
[13] https://pypi.python.org/pypi/setuptools
[14] http://sphinx-doc.org/
[15] https://www.python.org/dev/peps/pep-0008

4.3.1 Dispersion Model Prototype and Supportive Models

The purpose of defining prototypes for central models used in assimilation is to *a)* have a common defined set of features that any alternative models need to implement and *b)* to be able to implement assimilation in a manner agnostic to specific model; in other words, to be able to swap dispersion, meteorologic and release model implementations with as little changes as possible to assimilation algorithms.[16]

All classes described in this section lie in the `dispmodel.iface` module of Asim.

`Location` is a simple container for spatial coordinates, a non-abstract class.

- **`x, y, z`** attributes contain x (increases in the East direction), y (increases in the North direction) and z (height) coordinates in meters.
- **`distance_to(loc : Location) → double`** computes distance between this location and another one, `loc`.

`Nuclide` class describes physical properties of one radionuclide such as half-life, energy radiated during decay or coefficients describing tendency to deposit. It is a simple container for attributes with no methods. Instantiated `Nuclide` objects should be treated as immutable.

`SourceModel` is a prototype for description of the source of the pollutant release. It should provide location of the release, composition of the pollutant (what radionuclides are leaking) and release rate.

- **`inventory() → Nuclide array`** returns composition of the pollutant as an array of `Nuclide` objects. Currently the composition is assumed not to change in time for simplicity.
- **`release_rate(time) → vector`** returns rate of the release in $\mathrm{Bq \cdot s^{-1}}$for each pollutant at given time step `time` (all time `step` arguments are integers in Asim).
- **`location() → Location`** should return spatial coordinates of the expected pollutant leakage place.

`MeteoModel` represents, as the name suggests, a model of meteorologic conditions.

[16] depending on difference between original and alternative models, we expect that modifications may still be needed, for example the presented released dose proposal density is rather tied to puff-based dispersion model.

`wind_speed_at(loc, time) → double` is designed to return wind speed in $\mathrm{m} \cdot \mathrm{s}^{-1}$ at given location `loc` (all arguments named `loc` are of type `Location` in Asim) and time step.

`wind_direction_at(loc, time) → double` similarly returns wind direction in radians; 0 rad means North, $\frac{\pi}{2}$ East and so on.

`mixing_layer_height_at(loc, time) → double` returns height in meters of the atmospheric layer where air that rises from the ground clashes with colder air in the upper parts (at given time-spatial coordinates, ignoring the height). Such quantity is needed by certain dispersion models including our puff-based one.

`dispersion_xy(loc, time, total_distance) → double` returns dispersion coefficients σ_x, σ_y (which are assumed to be same) as presented in chapter 2 for a puff that has flown `total_distance` in meters.

`dispersion_z(loc, time, total_distance) → double` is a similar method that returns σ_z.

`DispersionModel` is the prototype for atmospheric pollutant dispersion models. Apart from simulating the pollutant propagation, it also needs to be able to compute pollutant concentration at given spatial coordinates and absorbed pollutant dose at any point of interest during the last time step. It does not contain any logic for meteorologic or release simulation, such data are rather passed through separate models. `DispersionModel` is required not to make any assumptions about these models apart from their conformance to the specified interface. `DispersionModel` is designed to keep track of time; implementations are expected to accept a parameter denoting time step Δt and advance internal time counter during each propagation step. Because continuous infinite assimilation is among use-cases, it is not recommended that dispersion models keep track of history; they are not required to provide any data or computations from all but the most recent time step.

`propagate(meteo_model, source_model)` advances to next time step by simulating atmospheric effects on the pollutant in the air. It sources meteorologic information exclusively from the `meteo_model` argument, which is an instance of the `MeteoModel` interface. It also simulates ongoing pollutant release (which may be zero) from the facility based on data provided by the `source_model` parameter (instance of `SourceModel`).

`concentration_at(loc) → vector` should return the concentration in $\mathrm{Bq} \cdot \mathrm{m}^{-3}$ of each pollutant component at given spatial coordinates `loc` and current time step.

`dose_at(loc) → vector` is designed to return time-integrated absorbed dose in Sv at given location `loc` per the most recent time step. Individual vector components correspond to pollutant components.

As an extension to theory in chapter 2, the dispersion model prototype assumes that multiple pollutant components (with different radioactive characteristics) may be released simultaneously. While this was not thoroughly tested, we considered supporting it in the API beneficial so that future extensions do not need to make drastic changes to it.

For performance reasons, implementations may return a reference to internal workspace attribute in vector-returning methods. This means that callers cannot make modifications to vectors they get. Given intended uses, this was considered to be more elegant than the alternative to let callers (optionally) pass the "out" parameter that would be returned back.[17]

4.3.2 Puff-based Dispersion Model

When implementing a concrete realization of the dispersion model prototype, the puff-based model described in chapter 2, it was noted that each puff could be viewed as a simple dispersion model of its own, and given that measured quantities are additive,[18] the whole plume-simulating model can be represented as a simple collection of puffs. This has been exploited in the software design in order to make the implementation clearer.

Note: arguments of implemented prototype methods are not repeated for brevity. Classes mentioned in this section can be found in the `dispmodel.puffmodel` Asim module.

`Puff(DispersionModel)` represents a single puff, a rather constrained realization of the dispersion model. It keeps track of its location, dispersion state, pollutant activity (accounting for radioactive decay) and composition, and total flown distance.

`__init__(time_step, inventory, Q, loc, time)` constructs the Puff. Beyond auxiliary variables `time_step` and `time`, it takes natural parameters `inventory` (composition of the released pollutant represented as a sequence of `Nuclides`), `Q` (per-component activities in Bq) and appearance location `loc`.

[17] we consider employing at least one of these approaches crucial for high computational efficiency as allocating memory and creating new objects upon each call constitutes significant overhead.

[18] with regards to puffs.

propagate() advances the puff in time according to (2.2) and (2.9), with wind field and other atmospheric parameters supplied by the `meteo_model`. `Puff` ignores `release_model`, which makes it unsuitable when used on its own.

concentration_at() computes pollutant concentration at given location in the atmosphere using (2.1). As an extension to the referenced formula, the implementation also accounts for reflection from and deposition on the ground and reflection from the atmospheric mixing layer.

dose_at() computes dose absorbed at given location (which is assumed to be on the ground) per the last `time_step` according to (2.3). As the same computation of concentration is used internally, same notes apply about reflecting additional physical phenomena such as ground and mixing layer reflection and ground deposition.

unit_dose_at(loc) is same as `dose_at()`, but ignores tracked activity `Q` and assumes it is unitary instead. This is an extension beyond `DispersionModel` to help the released dose proposal distribution.

PuffModel(DispersionModel) consists of a series of puffs that together form a plume.

__init__(time_step, puff_sampling_step, source_model) initializes the `PuffModel`. `puff_sampling_step`, which must be a multiple of `time_step` determines how frequently new puffs are created. `source_model` is used only to determine number of pollutant components, which is then assumed to be constant.

propagate() propagates child puffs and samples new ones (as determined by `puff_sampling_step`) using `source_model`.

concentration_at() just sums child puff concentrations.

dose_at() again only performs simple summing.

unit_puff_dose_at(loc, out=None) is an extension for proposal density that computes per-puff unit doses (does not sum them). The `out` parameter has the same meaning as in Ceygen.

puff_activities(out=None) is another extension that returns per-puff and per-component activities in a matrix.

Relationship between dispersion model, its implementations and related classes is depicted in Figure 4.2.

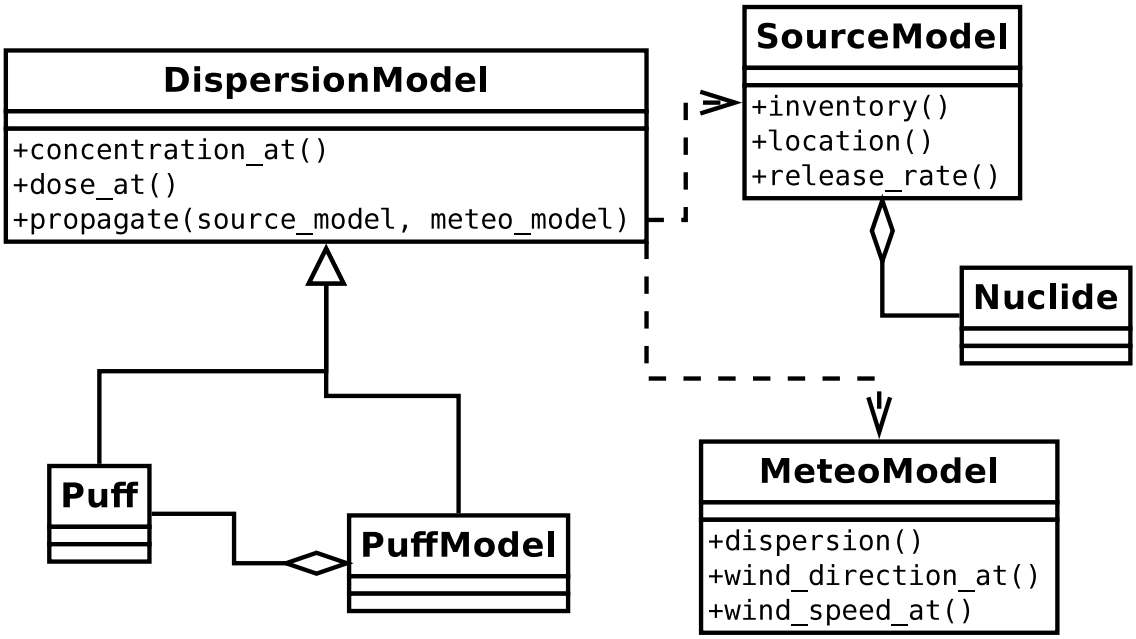

Figure 4.2: UML diagram of dispersion model prototype, supportive models and implementation. `DispersionModel` uses (depends on) `MeteoModel` and `SourceModel`, which is in turn composed of `Nuclide` instances. Both `Puff` and `PuffModel` are implementations of `DispersionModel`, `DispersionModel` being composed of `Puffs`.

4.3.3 Simulation

In order to be able to perform dispersion model validation and the twin experiment, supportive classes and a runner have been implemented in Asim's `simulation.simple` module:

`StaticCompoLocSourceModel(SourceModel)` is a semi-abstract code-sharing `SourceModel` subclass that provides implementations of `inventory()` and `location()` that return predefined values.

`SimulatedSourceModel(StaticCompoLocSourceModel)` simulates pollutant release based on per-`puff_sampling_step` activities that are passed to its constructor.

`PasquillsMeteoModel(MeteoModel)` is another semi-abstract class that exists to share code and definitions between meteorologic models based on Pasquill's stability category.

`StaticMeteoModel(PasquillsMeteoModel)` lets its instantiator specify Pasquill's stability category and a 4-dimensional precomputed grid of the wind field.

`run()` function executes the simulation. It reads meteorologic data from a file and passes it to `StaticMeteoModel` and employs `SimulatedSourceModel`. Finally it instantiates `PuffModel` and runs it against created source and meteorologic models. It also captures total absorbed doses on a $1\,\mathrm{km} \times 1\,\mathrm{km}$ grid and per-10-minute doses on specified receptor locations, which can be later used as data for assimilation. It provides facilities for plotting the results.

It should be noted that up until this point, no code had an explicit dependency on PyBayes or other components taking part in assimilation, and only a couple of places were aware of assimilation. Such functionality separation is intentional.

4.3.4 Assimilation

Finally, PyBayes can be plugged in to perform sequential Monte Carlo sampling on top of dispersion models. The idea is rather simple: create specialized `ReleaseModel` and `MeteoModel` subclasses that would transform and feed assimilated quantities (Q_t, a_t, b_t) into spawned `PuffModel` instances. The design gets slightly more complicated due to the fact that released dose assimilation is infeasible without appropriate proposal density and that the state variable has to be split into non-deterministic and conditionally deterministic part of variable dimension.

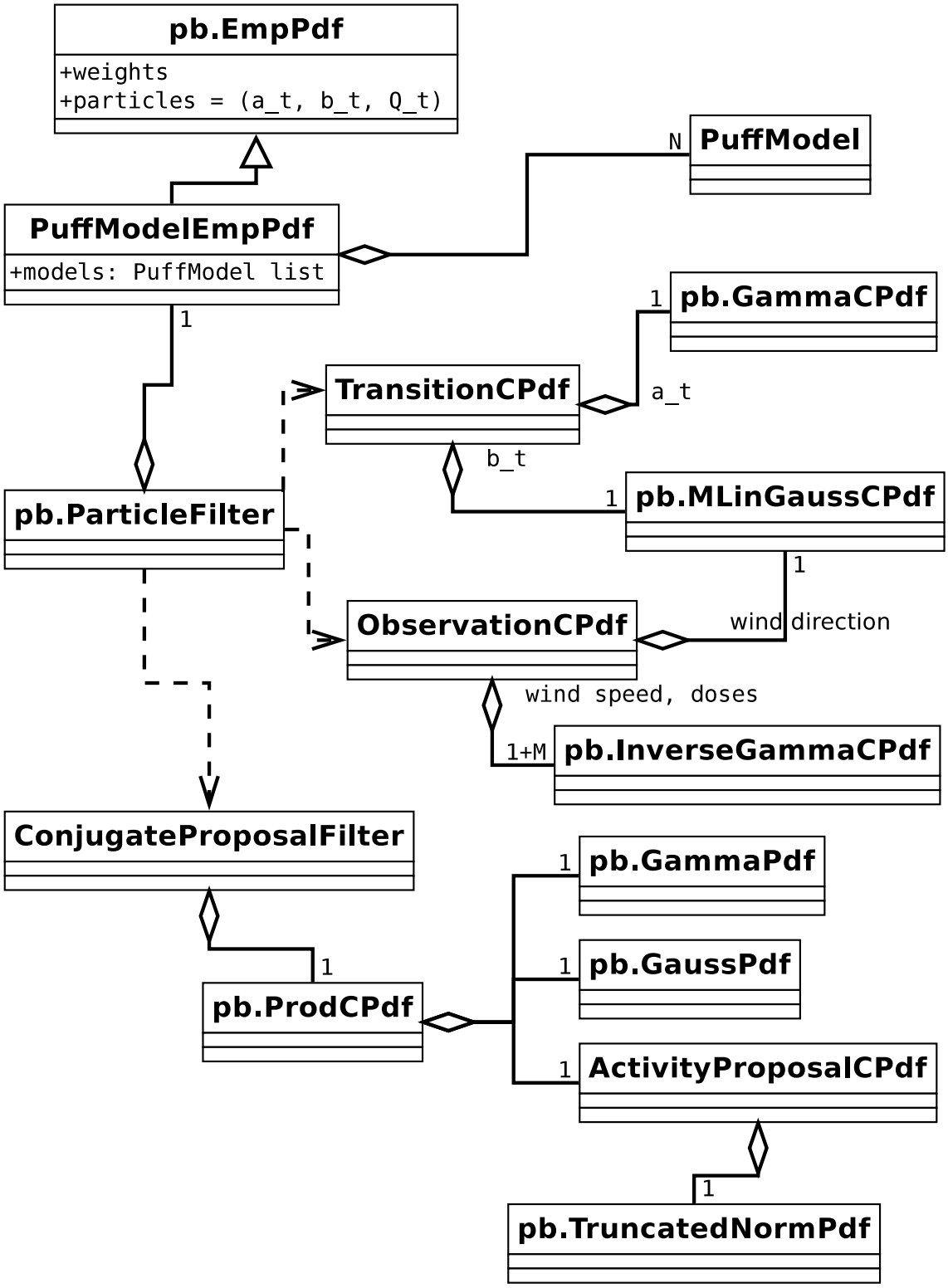

Figure 4.3: Objects participating in assimilation. Relationships between instances (rather than classes) are shown.

Presented classes are implemented in the `assimilation.twin` module of Asim. We alias PyBayes package to `pb` in class descriptions, i.e. `pb.GaussPdf` is PyBayes' implementation of the normal distribution. Overall assimilation design is depicted in the Figure 4.3, where diamond-ending lines represent composition and dashed arrows usage.

Puff-enhanced Empirical Posterior Distribution

To cope with special form of the state vector (2.10), (2.8), a problem-specific subclass of `pb.EmpPdf` has been created. It is described below along with supportive classes.

`PuffModelEmpPdf(pb.EmpPdf)` is a work-horse container for the assimilation; it enhances `pb.EmpPdf` to add a `PuffModel` instance to each particle. The weights $\omega_t^{(i)}$ and assimilated quantities $\left(a_t^{(i)}, b_t^{(i)}, Q_t^{(i)}\right)$ are kept in `pb.EmpPdf`, but `PuffModel` instances are stored in its own `models` attribute. The task of `PuffModelEmpPdf` is to keep them in sync and ensure that it can be used as a posterior density of `pb.ParticleFilter`. To do so, it re-implements two methods:

`resample()` ensures that `models` (in addition to `pb.EmpPdf` attributes) are resampled correctly during the final step of particle filter iteration. It tries to minimize copying, which is expensive.

`transition_using()` is called by particle filter for each particle to sample x_t given x_{t-1}, y_t and passed proposal density. This is where propagation of the dispersion model happens. Absorbed doses at receptors (assuming unitary puff acitivites) are precomputed here for the proposal and observation distributions so that they do not have to be computed twice.[19] Multiple smaller propagations of the dispersion model per each particle filter iteration (`transition_using()` call) happen to achieve adequate precision of the dose calculation.[20] Computing real absorbed doses is then a matter of simple vector multiplication of "unitary" doses and puff activities.

`AssimilationSourceModel(StaticCompoLocSourceModel)` (not shown in the diagram) simply hooks assimilation into dispersion models by returning assimilated released dose. As en exception, `AssimilationSourceModel` in fact just

[19] an optimization that convolutes the design, but cuts needed resources virtually in half as the dose evaluation is the single most demanding operation resource-wise.

[20] which can be, in time domain, viewed as a simple addition-based integration.

provides release rate that results in unitary released puff activity as it is triggered by the activity proposal distribution before the sampled activity is known (which is assigned to the released puff once it is known).

`AssimilationMeteoModel(PasquillsMeteoModel)` (not shown in the diagram) provides wind field to dispersion models given assimilated a_t, b_t and externally supplied numerical forecast according to transformations (2.5) and (2.6).

Transition and Observation Probability Density Functions

PyBayes particle filter expects transition and observation distributions in form of conditional probability density functions. Problem-specific ones are hence implemented in Asim, using building blocks provided by PyBayes.

`TransitionCPdf(pb.CPdf)` represents transition density $p(x_t|x_{t-1})$. `TransitionCPdf` implements (2.11) by employing `pb.GammaPdf` for wind speed correction coefficient a_t and `pb.MLinGaussCPdf` for wind direction correction coefficient b_t. It also formally contains Q_t, but its sampling is skipped (as it is infeasible) and during evaluation equal probability is assigned to all realizations. This is possible because it is the proposal distribution that performs this job. If there were not this special casing (and circular wrapping for b_t) `TransitionCPdf` could have been implemented as `pb.ProdCPdf`.

`ObservationCPdf(pb.CPdf)` models observation density $p(y_t|x_t)$ per (2.12). Anemometer wind direction read-out is evaluated using `pb.MLinGaussCPdf`, while wind speed and absorbed dose read-outs (from M receptors) are modeled using the `pb.InverseGammaCPdf` densities. As an optimization, only one instance of inverse gamma distribution is created for dose evaluation and shared.

Proposal Filter and Densities

The assimilation is made possible (assuming real-world resources) only by employing a suitable form of proposal density. Following classes implement it.

`ConjugateProposalFilter(pb.Filter)` provides problem-specific proposal density as shown in section 2.5 in form of a Bayesian filter as required by `pb.ParticleFilter`. We note that its name is slightly misleading as only a_t and b_t proposals take form of conjugate probability density functions; Q_t proposal is an approximation with arbitrarily chosen distribution. Its `bayes()` method is called by particle filter when it is about to sample a new particle. This method returns

a posterior distribution in form of `pb.ProdCPdf` representing (2.17) that is composed of:

`pb.GammaPdf` whose parameters are given by (2.23) for sampling wind speed correction coefficient a_t.

`pb.GaussPdf` for sampling wind direction correction coefficient b_t, tuned according to (2.19).

`ActivityProposalCPdf(pb.CPdf)` performs released activity proposal approximation as described in subsection 2.5.3. It is conditionally dependent on a_t and b_t. In its `sample()` method, it performs dispersion model propagation with unitary released activity and simulates dose measurements on receptors (computation results are saved for reuse in `PuffModelEmpPdf`). Computed doses are substituted into (2.25), which is then approximated using Laplace's method[21] as (2.26). The final approximation is then represented using `pb.TruncatedNormalPdf`.

Runner

The whole assimilation machinery is orchestrated by the `run()` method, which contains support for running multiple variants of the assimilation on the same data for easy comparison. It also saves assimilation results for reuse and allows them to be visualized.

4.3.5 Supportive Modules

Apart from work-horse classes, Asim contains also some auxiliary objects and functions in its `support` package. Module `support.newton` implements Netwon's method for finding a root of a function with known derivative. It accepts instances of the `Function` class prototype that exists to let Cython fully optimize the algorithm. Data visualization can be performed using convenience functions in the `support.plotters` module, which is implemented in ordinary Python as it is not performance-critical.

4.4 Integration in Distributed Worker System

One of the intended use-case of Asim is to provide so-called *workers* for web-based distributed dispersion modeling system. Instance of such system, which uses a dif-

[21] which involves finding maxima of (2.25) and computing second derivative in it.

ferent dispersion model, HARP, is available at https://dss.utia.cas.cz/. Such workers listen for tasks given by the server. A task assignment is described in a JSON format which contains necessary parameters for the worker and once the worker is done, it sends back the results again in the JSON format.[22] Two workers have been implemented (outside Asim project), one for simulation and one for assimilation.

4.4.1 puff_simulation Worker

The puff_simulation worker, which can be found on the enclosed DVD-ROM under a directory of the same name, performs atmospheric pollutant release simulation using the `asim.simulation` Asim package. It contains a stripped-down version of the runner found in the mentioned package that constructs the required classes with parameters given in the task JSON. Format of this experiment configuration file is best shown on an example (shortened for brevity):

```
{
    "simulation_length": 14400,
    "time_step": 120,
    "source_model": {
        "puff_sampling_step": 600,
        "activities": [
            [0.4E+16],
            (..)
            [1.9E+16]
        ]
    },
    "meteo_model": {
        "stability_category": 3,
        "grid_step_time": 3600,
        "grid_step_xy": 3000,
        "meteo_array": <4D array with wind speed, wind
                        direction on a time-spatial grid>
    },
    "receptors": [
        {"name": "ETE 01", x: 115.36, y: 445.01, z: 0},
        (...)
        {"name": "Sedlec", x: -30.50, y: -77.46, z: 0}
    ]
}
```

[22] https://tools.ietf.org/html/rfc7159

As can be seen, the parameters are organized in a structure similar to object-oriented design of Asim classes. The simulation runner therefore only interprets top-level parameters and passes other ones as constructor arguments to individual components.

puff_simulation worker repeats some of its input parameters in its output JSON file and adds:

- total doses computed on a grid (which is itself also specified),
- per-`puff_sampling_step` doses computed for each receptor specified in `receptors`,
- recorded trajectories for each released puff,
- wind speeds and directions that could have been measured at origin (facility) using anemometer given the meteorologic forecast.

Precise output format is not included here for brevity, but full versions of both files can be found on the enclosed DVD-ROM under the `puff_simulation/` folder.

4.4.2 puff_assimilation Worker

The simulation-performing worker is strictly analogous. It contains simplified version of Asim's `assimilation.twin.run()` runner that supports executing just one assimilation at a time. Input JSON format of the `puff_assimilation` worker is exactly the same as output format of the `puff_simulation` worker so that a twin experiment can be executed trivially. Its input can be however enhanced with more properties that affect assimilation (such as number of particles N etc.) — specified parameters override built-in defaults.

Once assimilation is done, output dataset is produced. It currently lists just estimated released activities (Q_t values) along with its standard deviations, a measure of particle filter effectivity, N_{eff} and time spent in each iteration. It is expected that output data will be extended to allow visualization of the results and better comparison (as much more variables are actually computed). Source code of this worker along with its example input and output formats is available on the enclosed DVD-ROM in the `puff_simulation/` directory.

Both simulation and assimilation workers assume that Asim, PyBayes and Ceygen are installed as Python site-packages.[23]

[23] There are however many standardized ways specify additional Python package search locations.

Chapter 5

Verification

In this final chapter, we verify that the implemented software stack works correctly and performs according to expected performance and accuracy levels on a series of benchmarks and an experiment. After defining the hardware and software environment used to perform the tests, results of the Ceygen benchmark suite are presented and compared against analogous NumPy functions. The second part of this chapter is devoted to verification of the assimilation algorithms as implemented in Asim in a so-called twin experiment.

5.1 Testing Environment

Following equipment and software was used to perform the benchmarking and testing below:

Processor	Intel Core i5-2520M @ 2.5 GHz (turbo boost up to 3.2 GHz)		
Main Memory	8 GB @ 1333 MHz		
Storage	Intel SSD 310 80 GB, mSATA 3Gb/s		
Operating System	Gentoo Linux (Base System release 2.2)		
	Linux kernel 3.10.26-gentoo		
Compiler	gcc 4.7.3	Cython	0.19.2
BLAS, LAPACK	ATLAS 3.10.1	Ceygen	0.3-10-gb90e849
NumPy	1.8.0	PyBayes	0.3-97-gdfd59f3
Python	2.7.5	Asim	0.1-44-g4d14d37
Eigen	3.0.6		

Used ATLAS[1] BLAS/LAPACK implementation was carefully compiled on the

[1] Automatically Tuned Linear Algebra Software: http://math-atlas.sourceforge.net/

target machine so that its automatically-tuning feature is used to maximum extent. NumPy was build with support for calling into BLAS/LAPACK enabled so that maximum expected performance can be achieved.

Note: A regression has been found in very recent versions of Cython (at least 0.20.1) which causes Cython compiler to crash when it is about to compile Ceygen. This is being investigated and until resolved, users are advised to use versions 0.19.1 or 0.19.2 with Ceygen.

5.2 Ceygen Benchmark

We have picked 4 algebraic operations of different nature to benchmark performance of Ceygen against equivalent NumPy functions. The benchmark itself is part of Ceygen and can be found in its `tests.bench` module.

Care is taken to use the function variants that incur least overhead: the `out` parameter is always specified for both NumPy and Ceygen function calls. The benchmarking code is compiled by Cython (to eliminate interpreter overhead) and involved objects are properly typed (in case of NumPy, most efficient type for both arguments and function itself is just `object`). All involved arrays are C-contiguous. To illustrate, only 2 Python API function calls (which cause most overhead) are done inside the loop where e.g. `numpy.add()` is called: `PyTuple_New()` to construct its parameters and `PyObject_Call()` to call it. Number of iterations is chosen so that at least 0.2 s is spent inside the loop.

In Figure 5.1 we compare vector-vector addition for vectors of size 2 up to size 128. As in all latter benchmarks we plot relative wall clock time consumed by one call, i.e. the slowest variant gets assigned time 1 an other variants are scaled relatively to it. Ceygen shines in this $O(n)$ operation and is roughly 3×faster for all examined data sizes.

Square matrix-vector dot product performance is plotted in Figure 5.2. We note that Ceygen performance is about double for problem sizes 2–32, and approaches that of NumPy from size 64.

Figure 5.3 shows that Ceygen consumes approximately half of the resources compared to NumPy in square matrix-matrix dot product for sizes 2–48, and slowly converges to NumPy as problem size increases. Visible bumps can be, in our opinion, explained by CPU cache and alignment effects.

Last plot, Figure 5.4, where matrix determinant is computed, is most interesting, because it includes parallelized versions. Ceygen `lu.det()` method is run under

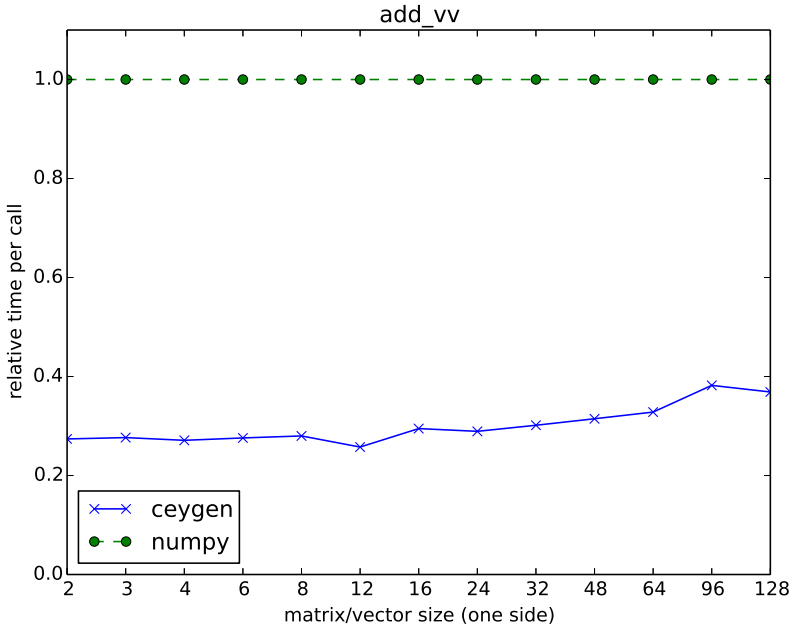

Figure 5.1: Ceygen vector-vector addition benchmark

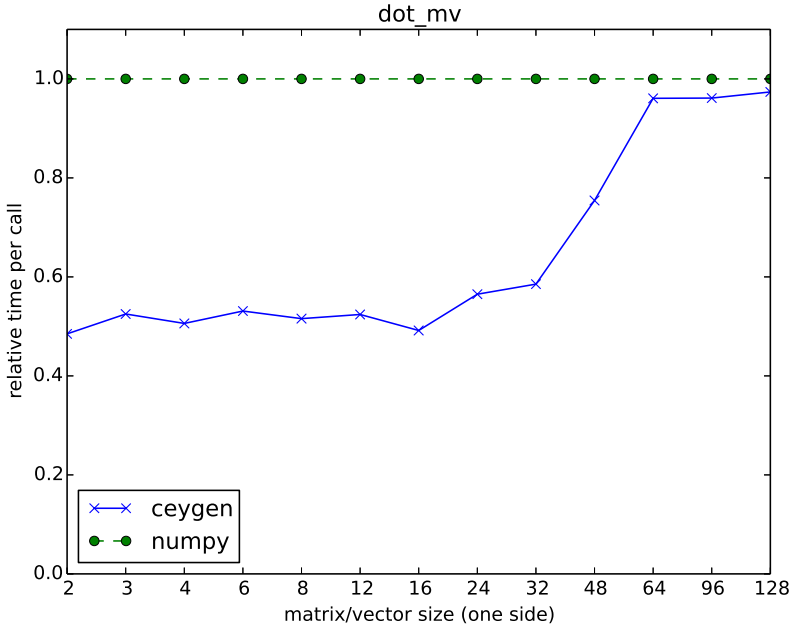

Figure 5.2: Ceygen matrix-vector dot product benchmark

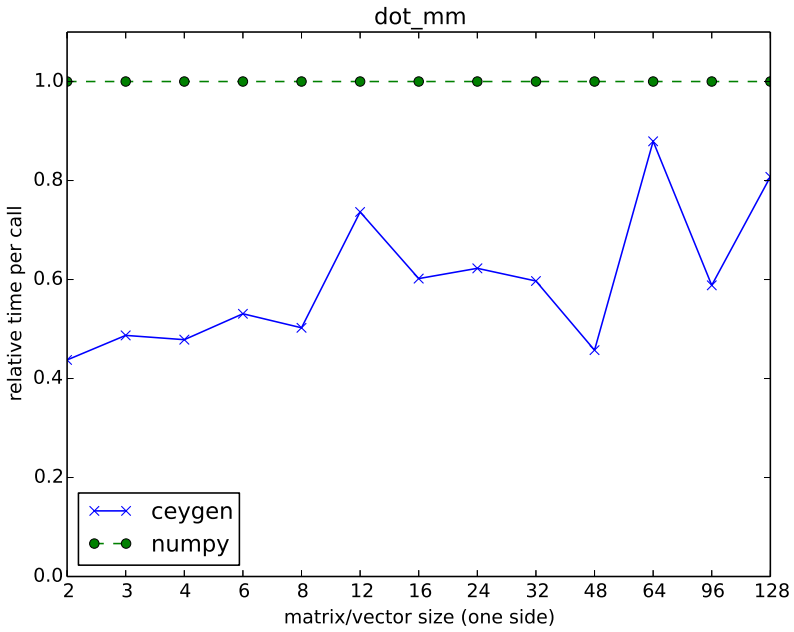

Figure 5.3: Ceygen matrix-matrix dot product benchmark

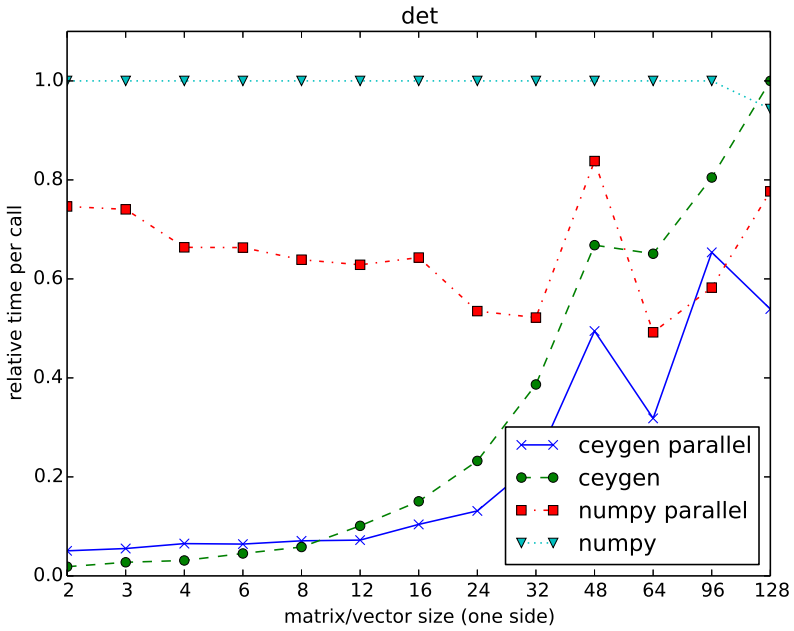

Figure 5.4: Ceygen determinant calculation benchmark including parallel variants

Cython's `prange` block that internally uses OpenMP[2], while NumPy `det()` is run in parallel using Python `multiprocessing` module, which spawns multiple Python processes and facilitates communication between them. Both methods have been checked to use the same computing algorithm, the LU decomposition with partial pivoting. Serial versions are included too for comparison.

Comparing appropriate versions of Ceygen and NumPy, it can be seen that the Ceygen variants are over tenfold faster up to size 12 and gradually come nearer to NumPy variants as problem size approaches 100, where they are even occasionally outperformed.

Worth noting is parallel efficiency of respective algorithms. Python `multiprocessing` seems to be rather successful at parallelizing NumPy determinant computations, with irregular fluctuations (still, parallel NumPy variant is slower than even serial Ceygen function up to 48×48 matrices). Parallelization in Cython has slightly different behavior. It has negative impact in 2–8 problem sizes and slowly starts to be effective only with 24×24 and larger matrices. We speculate that this is caused by the fact that all threads used the same instances of passed arrays in this synthetic test; this has lead to frequent cache flushes as different threads might have written into near memory locations. Also, reference counting (which is implemented using atomic integer operations in Cython memoryviews) might be the culprit. We expect that real-world code would not exhibit such properties and that it would parallelize better.

In general, we believe that we may conclude that Ceygen really brings significant performance increase, at least for operations being considered, especially for matrices and vectors whose one side is smaller than roughly 50 elements; speed improvement usually exceeds factor of 2. Performance beyond this size depends on function benchmarked, but in worst case it gradually approaches that of equivalent NumPy method.

5.3 Twin Experiment

During twin experiment, an atmospheric radioactive pollutant release simulation is performed under realistic conditions, measurements are recorded, noise is added and then assimilation is run against the noisy measurements. Assimilation results are then compared with original simulation.

[2] http://openmp.org/

5.3.1 Setup

A 4-hour simulation of hypothetic pollutant release from the Temelín power plant was used as validation scenario. Realistic wind-field was generated with Northeast direction, which varies slightly during the simulation, with wind speed that fluctuates around $2\,\mathrm{m/s}$. Because real-world dose measurement devices on site perform data read-outs each 10 minutes, the same period was chosen as an interval between 2 puff releases and subsequently as iteration interval in assimilation. Model propagation interval was set to 2 minutes in both runs so that dose measurements are approximated with sufficient precision. The ^{41}Ar isotope with half-life of 109 minutes was released during the first hour of the experiment, with following activities: $[0.1, 3, 4, 2, 3, 1] \times \mathrm{Bq}$.

Once simulation was done, the captured measurements (anemometer read-outs at the facility and dose receptor measurements) were obfuscated by noise of the similar type and same intensity as defined by the observation model: gamma noise of relative variance γ_v was added to wind speed measurements, wind direction was burdened by Gaussian noise with σ_θ standard error and individual dose measurements were obfuscated by gamma distribution with relative error γ_y (in every case, added noise was unbiased).

Parameters for both obfuscation and assimilation measurement model were chosen as follows: $\gamma_v = 0.1$, $\sigma_\theta = 5°$ and $\gamma_y = 0.2$ as recommended by [18]. Transition model parameters were chosen slightly wider to accommodate for possible rapid changes: $\gamma_a = 0.2$ (for a_t coefficient relative error) and $\sigma_b = 15°$ (for wind direction correction coefficient). Number of particles N was fixed to 1000.

5.3.2 Results

Simulation and assimilation results are shown side-by-side in Figure 5.5; absorbed dose contour graph is only shown in simulation.[3] As can be seen, assimilated trajectories match the simulated ones, which means the assimilation succeeded at revealing the hidden state. As puffs move away from the radioactive monitoring network of the facility (whose receptors are marked by blue dots), a distinctive rise in trajectory variance is observed (they become more dispersed). We believe that this can be explained by the fact that at this point, the observation vector is effectively reduced to anemometer measurements (as puffs are out of range of any receptors) and the observation density thus becomes significantly wider.[4]

[3] but could have been computed from available assimilation data as it is deterministic given puff trajectories and activities.

[4] as highly-nonlinear receptor dose measurements are significantly narrower.

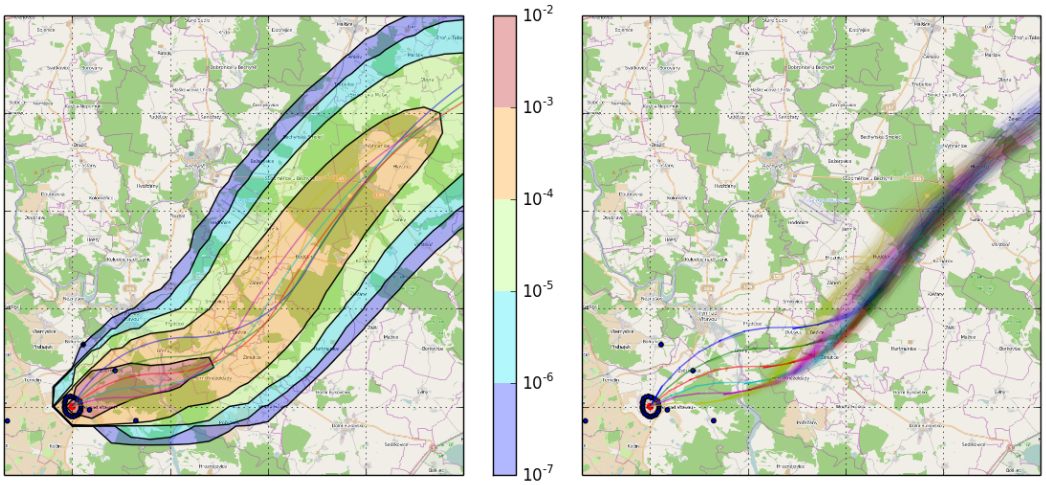

Figure 5.5: Simulation (left) and assimilation (right) trajectories side-by-side. Color curves represent trajectories of individual puffs; curve opacity represents likelihood in case of assimilation.

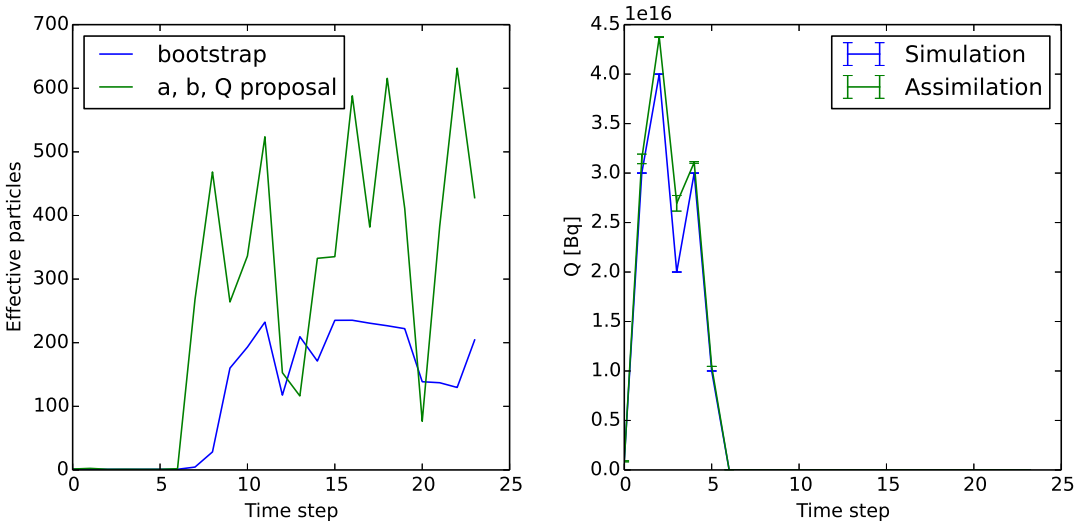

Figure 5.6: N_{eff} and simulated vs. assimilated released activities

Figure 5.6 shows, on the right, assimilated nuclear activities released per individual puffs along originally simulated ones. They fall within expected accuracy so that we can conclude that the implemented assimilation method works correctly. On the left, plot of N_{eff}, a measure of particle filtering efficiency, development in time is included. $1 \leq N_{\text{eff}} \leq N$ is an approximation of how many particles actually contribute to resulting distribution (the rest of them is just overhead). For comparison, results from the bootstrap variant of the particle filter that assimilates just (a_t, b_t)[5] are plotted alongside the studied optimized variant.

N_{eff} is very close to 1 during first 6 iterations for both implementations and raises only after this point. We opinion that this coincides with and could be explained by that fact that during these initial 6 steps, new puffs are released; this means that all 24 receptors arranged in ring around the facility measure non-negligible quantities and therefore contribute to narrowing of the observation density, which becomes sharp-peaked and assigns radically different likelihoods to particles that are in fact very close to each other in the state space.

After sixth iteration, number of effective particles of the optimized variant fluctuates around 500, which could be, in our opinion, considered a good result. We speculate that occasional drops to values around 100 could be caused by outliers in supplied observations, which is supported by the observation that bootstrap variant shows coinciding drops. The fact that the bootstrap variant, which performs less demanding task of assimilating only the wind field, lags behind with number or effective particles slightly below 200 further emphasizes efficiency of the employed proposal density approach.

[5] and is given correct Q_t. The bootstrap variant also correctly revealed puff trajectories.

Conclusion

This text has been divided into several parts. In the first one, well-known theory of recursive Bayesian estimation was briefly reviewed, then emphasis has been put on some of its algorithms: the particle filter and more specialized marginal particle filter.

Second chapter has continued with presentation of the employed model to simulate atmospheric dispersion and devised ways to perform assimilation on top of this model: state and observation models have been defined and discussed. Significant portion of the chapter has been dedicated to derivation of proposal density that has been later shown to substantially increase efficiency.

Next part concentrated on software analysis. First, requirements posed on the desired software stack have been devised from a set of use-cases. Rapid prototyping possibility, extensibility, high computing efficiency and proper documentation have been identified as key demands. The chapter has continued by review of possible building blocks and ended with suggestions. Asim project for assimilation, PyBayes enhancements and a new library for linear algebra employing Eigen have been proposed.

Chapter 4 has presented the design of individual developed software components and has described their implementation. It has tried to clarify relationships between individual classes and finally has shown interaction of objects that span multiple components during a complex task of proposal distribution-enhanced assimilation of dispersion models.

The final chapter has shown results achieved with the implemented software. It has been noted that $2\times$ to $10\times$ performance increase can be achieved by Ceygen for small to moderate problem sizes. Functionality of Asim and other participating projects has been finally verified on a twin experiment; successful assimilation has been demonstrated with adequate particle filtering efficiency.

If we were to identify key contributions of this work, it would be:

- The *Ceygen* library, publicly released software useful for anybody who seeks a

way to perform efficient linear algebra with Cython memoryviews. Ceygen is, to our knowledge, the best performing solution for small to moderate vectors and matrices available for memoryviews.

- The *Asim* project, which brings object-oriented design to dispersion model assimilation research, coupled with improvements to the PyBayes library including optimization resulting from Ceygen porting, from which many may benefit at it is also an open-source project.

Suggestions for possible future improvement areas include mainly integration of emerging techniques that have the potential to further improve efficiency of sequential Monte Carlo sampling. One example of such technique is the AMIS[6] approach that has been studied in the assimilation field in [21] with promising results.

[6] Adaptive Multiple Importance Sampling.

List of Figures

Bibliography

[1] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D.S. Seljebotn, and K. Smith. Cython: The Best of Both Worlds. *Computing in Science Engineering*, 13(2):31–39, march-april 2011. 31

[2] X. Cai, H. P. Langtangen, and H. Moe. On the performance of the Python programming language for serial and parallel scientific computations. *Sci. Program.*, 13(1):31–56, jan 2005. 30

[3] D. Crisan and A. Doucet. A survey of convergence results on particle filtering methods for practitioners. *Signal Processing, IEEE Transactions on*, 50(3):736–746, 2002. 11

[4] A. Doucet, N. de Freitas, and N. Gordon, editors. *Sequential Monte Carlo Methods in Practice*. Springer, 2001. 9, 11

[5] M. Florisson. Techniques for static and dynamic compilation of array expressions. Master's thesis, Edinburgh, 2012. 35

[6] G.H. Golub and J. Welsch. Calculation of Gauss quadrature rules. *Math. Comp.*, (23):221–230, 1969. 17

[7] N.J. Gordon, D.J. Salmond, and A.F.M. Smith. Novel approach to nonlinear/non-Gaussian Bayesian state estimation. *Radar and Signal Processing, IEE Proceedings F*, 140(2):107–113, apr 1993. 9

[8] P.H. Hiemstra, D. Karssenberg, and A. van Dijk. Assimilation of observations of radiation level into an atmospheric transport model: A case study with the particle filter and the ETEX tracer dataset. *Atmospheric Environment*, pages 6149–6157, 2011. 17

[9] G. Johannesson, B. Hanley, and J. Nitao. Dynamic Bayesian Models via Monte Carlo–An Introduction with Examples. Technical report, Lawrence Livermore National Laboratory, 2004. 14

[10] R.E. Kalman. A New Approach to Linear Filtering and Prediction Problems. *Transactions of the ASME — Journal of Basic Engineering*, 82(Series D):35–45, 1960. 8, 45

[11] R.E. Kass and A.E. Raftery. Bayes factors. *J. of American Statistical Association*, 90:773–795, 1995. 24

[12] M. Klaas, N. de Freitas, and A. Doucet. Toward Practical N^2 Monte Carlo: the Marginal Particle Filter. In *Proceedings of the Twenty-First Annual Conference on Uncertainty in Artificial Intelligence (UAI-05)*, pages 308–315, Arlington, Virginia, 2005. AUAI Press. 11

[13] M. Laitl. Implementation environment for Bayesian filtering algorithms. Bachelor thesis, Faculty of Nuclear Sciences and Physical Engineering of the Czech Technical University in Prague, Czech Republic, 2011. 28, 30, 38

[14] M. Laitl. Sequential Monte Carlo methods for atmospheric dispersion model assimilation. Research project, Faculty of Nuclear Sciences and Physical Engineering of the Czech Technical University in Prague, Czech Republic, 2012. 21

[15] P. Pecha and R. Hofman. Construction of observational operator for cloudshine dose from radioactive cloud drifting over the terrain. In *Proc. of 14th International Conference on Harmonisation within Atmospheric Dispersion Modelling for Regulatory Purposes*, 2011. 17

[16] Dag Sverre Seljebotn. Fast numerical computations with Cython. In Gaël Varoquaux, Stéfan van der Walt, and Jarrod Millman, editors, *Proceedings of the 8th Python in Science Conference*, pages 15–22, Pasadena, CA USA, 2009. 30

[17] V. Šmídl. Software Analysis Unifying Particle Filtering and Marginalized Particle Filtering. In *Proceedings of the 13th International Conference on Information Fusion*. IET, 2010. 45

[18] I.M.G. Thompson, C.E. Andersen, L. Bøtter-Jensen, E. Funck, S. Neumaier, and J.C. Sáez-Vergara. An international intercomparison of national network systems used to provide early warning of a nuclear accident having transboundary implications. *Radiation protection dosimetry*, 92(1-3):89, 2000. 21, 81

[19] V. Šmídl and R. Hofman. Adaptive importance sampling in particle filtering. In *Information Fusion (FUSION), 2013 16th International Conference on*, pages 9–16, July 2013. 13

[20] I. Wilbers, H.P. Langtangen, and Å. Ødegård. Using Cython to speed up numerical Python programs. In B. Skallerud and H. I. Andersson, editors, *Proceedings of MekIT'09*, pages 495–512. NTNU, Tapir, 2009. 31

[21] V. Šmídl and R. Hofman. Efficient Sequential Monte Carlo Sampling for Continuous Monitoring of Radiation Situation. *Technometrics*, submitted, 2012. 14, 85

Appendix A

Contents of the Enclosed DVD-ROM

abstract.pdf contains abstract of this thesis in PDF format.

Asim/ contains source code of the Asim project version 0.1-44-g4d14d37.

Asim.pdf is a generated documentation of the Asim project, in PDF hyper-linked form.

Ceygen/ contains source code of Ceygen version 0.3-10-gb90e849.

Ceygen.pdf is hyper-linked PDF documentation of the Ceygen library.

puff_assimilation/ contains source code of the distributed system worker based on Asim performing assimilation; includes example inputs and outputs and usage notes.

puff_simulation/ contains source code of the distributed system worker based on Asim performing simulation; includes example inputs and outputs and usage notes.

PyBayes/ contains source code of the PyBayes library version 0.3-97-gdfd59f3.

PyBayes.pdf is generated API documentation of the PyBayes library in hyper-linked PDF form.

text.pdf is this text in hyper-linked PDF format.