Environmental demogenetic model

Arnaud Becheler

NNT: 2018SACLS145

Environmental demogenetic models

Thèse de doctorat de l'Université Paris-Saclay
préparée à l'Université Paris-Sud

École doctorale n°567 Sciences du Végétal

Spécialité de doctorat: Biologie

Thèse présentée et soutenue à Gif-sur-Yvette, le 30 Mai 2018, par

Arnaud Becheler

Après avis des rapporteurs :

Solenn Stoeckel (Chargé de Recherche, INRA Rennes)
Lacey L. Knowles (Professeur, Université du Michigan)

Composition du Jury :

Myriam Harry Professeur Paris-Sud, Paris-Saclay (UMR EGCE)	Présidente
Solenn Stoeckel Chargé de Recherche, INRA Rennes (UMR IGEPP)	Rapporteur
Christine Dillmann Professeur, INRA (UMR GQE-Le Moulon)	Examinatrice
Mathilde Carpentier Maître de Conférences, Sorbonne Université (UMR ISYEB)	Examinatrice
Renaud Vitalis Directeur de Recherche, INRA (CBGP Montferrier-sur-Lez)	Examinateur
Stéphane Dupas Chargé de Recherche, IRD (UMR EGCE)	Directeur de thèse
Camille Coron Maître de Conférences, Paris-Saclay (LMO)	Co-encadrante

Thèse de doctorat

UNIVERSITÉ PARIS-SACLAY

DOCTORAL THESIS

Environmental demogenetic models

Author:
Arnaud BECHELER

Supervisor:
Dr. Stéphane DUPAS

Co-supervisor:
Dr. Camille Coron

A thesis submitted in fulfillment of the requirements
for the degree of Doctor of Biology

in the

Laboratoire Evolution, Génomes, Comportement, Ecologie
Pôle Evolution et Ecologie

September 16, 2018

We shall not cease from exploration
And the end of all our exploring
Will be to arrive where we started
And know the place for the first time.
Through the unknown, remembered gate
When the last of earth left to discover
Is that which was the beginning;
At the source of the longest river
The voice of the hidden waterfall
And the children in the apple-tree
Not known, because not looked for
But heard, half-heard, in the stillness
Between two waves of the sea.

T.S. Eliot, Four Quartets

UNIVERSITÉ PARIS-SACLAY

Abstract

Université Paris-Saclay
Pôle Evolution et Ecologie

Doctor of Biology

Environmental demogenetic models

by Arnaud BECHELER

Résumé

Les invasions biologiques sont un aspect majeur du changement global et une des principales menaces pesant sur la biodiversité. Leur étude est donc fondamentale pour proposer aux décideurs les scenarios prédictifs dont ils ont besoin pour mettre au point des mesures de conservation adaptées. Pour renseigner ces scénarios, des données doivent être collectées. Une source de données de moins en moins coûteuse consiste en un échantillon génétique de la population à étudier, où des individus sont échantillonnés en divers points géographiques et/ou temporels puis génotypés en différents points du génome.

Renseigner des modèles populationnels spatiallement explicites sur la base de données génétiques peut être extrêmement complexe à mettre en oeuvre mathématiquement si le modèle n'est pas trivial (par exemple si l'hétérogénéité du milieu est prise en compte dans les patrons de croissance ou de migration des populations). Au prix de quelques approximations, des méthodes de simulation numérique comme le Calcul Bayésien Approché (ABC) permettent alors l'étude de ces modèles complexes. Toutefois, les bases de code informatique existante pour la simulation de génétique de populations en paysage explicite sont assez rigides et permettent peu de flexibilité dans la définition des modèles simulatoires. Notamment, les lois de migration et de croissance des populations sont très difficilement modifiable : cela est problématique pour adapter les ressources de code existantes à de nouveaux scénarios biologiques spécifiques. De plus, la comparaison de modèles étant une étape clé de la méthodologie ABC, cela impose que différentes versions de simulateurs de coalescence puissent être rapidement développées et maintenues pour permettre de trouver le meilleur modèle représentant le mieux les données. Enfin, lorsque l'invasion étant assez récente, les mutations génétiques peuvent être considérées comme un facteur négligeable de la structuration de la diversité génétique de l'échantillon: cette hypothèse ouvre la voie à des méthodes statistiques originales qui ne sont pas implémentables avec les simulateurs actuels.

L'apport majeur de cette thèse est Quetzal: une bibliothèque `C++` de code ouverte, assez générale pour pouvoir aisément s'adapter à un grand nombre de modèles possibles et qui permet d'intégrer facilement ces hypothèses dans le code de simulation sans toutefois s'y limiter. Quetzal inclue des algorithmes originaux et génériques.

Les principaux concepts de programmation permettant d'utiliser et d'étendre Quetzal sont exposés à travers les différents chapitres.

La crétation de Quetzal a été initialement motivée par l'étude de l'invasion du frelon asiatique (*Vespa velutina*) en Europe, étude pour laquelle les simulateurs existants ne sont pas adaptés car faisant des hypothèses trop spécifiques. Quetzal a donc été conçue pour permettre d'implémenter facilement un modèle où la croissance des populations dans chaque unité paysagère est décrite par n'importe quelle fonction des conditions environnementales locales, tandis que les flux migratoires entre populations sont tirés dans des lois dont les densités peuvent également être librement définies par l'utilisateur (par exemple comme une fonction de la distance géographique à parcourir ou comme fonction de l'hétérogénéité paysagère). Certains paramètres de ces fonctions sont inconnus et doivent être estimés. Conditionnellement à la démographie, un processus de coalescence permet de simuler l'histoire génétique de l'échantillon. Une fois la simulation achevée, la procédure ABC permet de accepter/rejeter les valeurs de paramètres en fonction de la plausibilité des données génétiques qu'ils permettent de générer.

Enfin, nous mettons à profit les particularités du contexte d'étude du frelon asiatique pour développer une méthodologie spécifique reposant sur le formalisme des partitions floues et qui permet, en recentrant l'analyse sur les processus démographiques très récents et en négligeant les mutations, de réduire le nombre d'hypothèses, le nombre de paramètres et le coût simulatoire de l'analyse. L'estimation des paramètres d'un modèle réaliste de génétique des populations pour le frelon asiatique n'est pas présenté ici.

Mots-clés : invasions biologiques, *Vespa velutina*, génétique des populations, coalescence, Calcul Bayésien Approché, simulation, bibliothèque `C++`, développement logiciel, partitions floues.

Abstract

Biological invasions are a major aspect of global change and one of the most challenging threats to biodiversity. Studying them is then fundamental in order to provide relevant predictive scenarios for conservation policies guidance. Data have to be collected to inform these scenarios. Genetic sampling of populations is a possible source of information: individuals are sampled at various geographic or temporal points, and genotyped at various points in the genome.

Informing spatially explicit populations models based on genetic data can be mathematically extremely complex if the model is not trivial (for example if accounting for the environmental heterogeneity effects on migration or growth patterns). At the cost of some approximations, numerical simulations methods like the Approximated Bayesian Computation (ABC) allow to study complex models. However, the existing computer code bases for simulating population genetic diversity in explicit landscapes are quite rigid and do not allow to easily modify the simulation models definition. Importantly, the dispersal and growth laws are hardly modifiable: this complicates the adaptation of the existing simulation resources to new and specific biological scenarios. Furthermore, models comparison is a key step of the ABC methodology that requires various simulators versions to be easily developed and maintained in order to identify the best model that best represent the data. Finally, when the invasion is recent enough, genetic mutations can be neglected in shaping the genetic sample structure: this hypothesis opens the way for new statistical methods that are currently not implementable with the available simulators.

The major contribution of this thesis is Quetzal: an open-source `C++` library that is general enough to be easily adapted to a wide range of possible models and that allows to integrate easily specific modeling hypothesis in the simulation code. Quetzal includes original generic algorithms. The main programming concepts allowing to use and extend Quetzal are exposed through the various chapters.

Quetzal design was initially motivated by the study of the Asian hornet (*Vespa velutina*) in Europe, for which the existing simulators were not relevant because of their specific hypothesis. Consequently, Quetzal has been designed to allow the user to easily implement a model where the population growth in each landscape unit is described by a user-defined function of the local environmental features, whereas

the migration flux between populations are sampled in laws which densities can be freely defined by the user (for example as a function of the geographical distance or of the environmental features). Some of the parameters of these functions are unknown and should be estimated. Conditionally to the demography, a coalescence process allows for simulating the genetic history of the sample. Once the simulation done, the ABC method allows for accepting/rejecting the parameters values as a function of the data plausibility they generate.

Finally, we take advantage of the biological context to develop a specific methodology built on the fuzzy partitions formalism. It allows to focus only on the very recent demographic processes by neglecting the mutational process, and consequently to reduce the number of hypothesis, the number of parameters, and the simulation cost of the analysis. Estimating the parameters of a realistic demogenetic model that relevant for the hornet invasion is beyond the scope of this thesis.

Keywords: biological invasions, *Vespa velutina*, population genetics, coalescence, Approximate Bayesian Computation, simulation, `C++` library, software development, fuzzy partitions.

Synthèse en français des travaux présentés

Les invasions biologiques sont un aspect important du changement global et représentent une des principales menaces pesant sur la biodiversité. Leur étude est donc fondamentale pour proposer aux décideurs publics ou privés les scenarios prédictifs dont ils ont besoin pour mettre au point des mesures de conservation adaptées. Renseigner ces scénarios nécessite la mise au point de modèles dont certains paramètres doivent être renseignés par des données. Une source de données de moins en moins coûteuse consiste en un échantillon génétique de la population à étudier, où des individus sont échantillonnés en divers points géographiques et/ou temporels puis génotypés en différents points du génome.

Renseigner des modèles populationnels spatiallement explicites sur la base de données génétiques (c'est à dire construire des estimateurs pour les paramètres considérés) peut être extrêmement complexe à mettre en oeuvre mathématiquement si le modèle n'est pas trivial. Par exemple la prise en compte de l'hétérogénéité spatiotemporelle du milieu dans les patrons de croissance des population, de dispersion des individus ou de mutation des génomes complexifie énormément l'analyse mathéamtique. Dès lors, il peut être impossible de construire mathématiquement ces estimateurs.

Au prix de quelques approximations, des méthodes de simulation numérique comme le Calcul Bayésien Approché (ABC) permettent l'étude de ces modèles complexes. Les méthodologies ABC consistent à simuler des données sous un modèle dont les paramètres sont tirés dans des lois connues a priori qui résument les connaissance à priori sur le modèles. Par un algorithme d'acceptation/rejet, les paramètres des simulations menant à des données simulées très proches des données observées permettent de reconstruire des distributions a posteriori qui affinent la connaissance des paramètres du modèle.

Toutefois, les bases de code informatique existantes pour la simulation de génétique de populations en paysage explicite sont assez rigides et permettent peu de flexibilité dans la définition des modèles simulatoires. Notamment, les lois de migration et de croissance des populations sont très difficilement modifiables : cela est problématique pour adapter les ressources de code existantes à de nouveaux scénarios biologiques. Par exemple, lorsque l'invasion biologique est très récente, les

mutations génétiques peuvent être considérées comme un facteur négligeable de la structuration de la diversité génétique de l'échantillon: cette hypothèse ouvre alors la voie à des méthodes statistiques originales qui ne sont pas implémentables avec les simulateurs actuels qui sont tous orientés vers la simualtion explicite de mutations. De plus, la comparaison de modèles étant une étape clé de la méthodologie ABC, cela impose que différentes versions de simulateurs de coalescence puissent être rapidement développées et maintenues pour permettre de trouver le meilleur modèle représentant le mieux les données. Idéalement, le développement inforamtique d'un nouveau modèle simulatoire devrait pouvoir s'appuyer sur des bibliothèques de composantes suffisamment abstraites et générales pour pouvoir être réutilisées.

L'apport majeur de cette thèse est Quetzal: une bibliothèque `C++` de code ouverte, dont chaque composante est assez générale pour pouvoir aisément s'adapter à un grand nombre de modèles possibles. Quetzal permet d'intégrer facilement des hypothèses variées dans le code de simulation, sans toutefois s'y limiter grâce à une extensibilité importante. Quetzal inclue des algorithmes originaux et génériques. Les principaux concepts de programmation permettant d'utiliser et d'étendre Quetzal (programamtion générique, abstractions, classes de traits, classes de politiques) sont exposés à travers les différents chapitres.

La crétation de Quetzal a été initialement motivée par l'étude de l'invasion du frelon asiatique (*Vespa velutina*) en Europe, étude pour laquelle les simulateurs existants ne sont pas adaptés car faisant des hypothèses trop spécifiques. Quetzal a donc été conçue pour permettre d'implémenter facilement un modèle où la croissance des populations dans chaque unité paysagère est décrite par n'importe quelle fonction des conditions environnementales locales, tandis que les flux migratoires entre populations sont tirés dans des lois dont les densités peuvent également être librement définies par l'utilisateur (par exemple comme une fonction de la distance géographique à parcourir ou comme fonction de l'hétérogénéité paysagère). Certains paramètres de ces fonctions sont inconnus et doivent être estimés. Conditionnellement à la démographie, un processus de coalescence permet de simuler l'histoire génétique de l'échantillon. Une fois la simulation achevée, la procédure ABC permet de accepter/rejeter les valeurs de paramètres en fonction de la plausibilité des données génétiques qu'ils permettent de générer.

Enfin, nous mettons à profit les particularités du contexte d'étude du frelon asiatique pour développer une méthodologie spécifique reposant sur le formalisme des partitions floues et qui permet, en recentrant l'analyse sur les processus démographiques très récents et en négligeant les mutations, de réduire le nombre d'hypothèses, le nombre de paramètres et le coût simulatoire de l'analyse. L'estimation des paramètres d'un modèle réaliste de génétique des populations pour le frelon asiatique n'est pas présenté ici.

Acknowledgements

Après quelques milliers de lignes de code écrites, plus encore d'effacées, après trois années de thèse (et quelques mois grapillés), voici le moment de remercier les gens qui, d'une manière ou bien d'une autre, y ont participé.

Merci tout d'abord à mes deux directeurs, Stéphane et Camille. Je vous dois énormément. Je garde pour souvenir de vos qualités et de votre encadrement cette réunion que nous avions eu sur la distance de transfert floue: Stéphane parlant de Biologie, Camille de Mathématiques et moi d'Informatique. Nous trouvions enfin le langage dans lequel nous comprendre et j'apprenais enfin le sens de l'interdisciplinarité.

Stéphane, pour la confiance que tu m'as accordée dès le début de ce projet, un immense merci. Je regarde le chemin parcouru pendant les trois dernières années, et je me dis que je ne serais pas ici à cet instant si tu ne m'avais placé sur ce chemin, ou si tu ne m'y avais pas accompagné. Ton inventivité, ta sympathie, ta créativité et ta grande intuition sont des qualités essentielles à tout projet. Je suis heureux d'avoir choisi ton sujet de thèse et d'avoir appris à te connaître. Je garde de ton encadrement l'image d'un jaillissement permanent d'idées nouvelles, d'un sentiment de liberté et d'humanité parmi des encouragements toujours renouvellés.

Camille, merci de ta très grande aide. Tu as été le pilier sur lequel s'appuyer, et "pertinence" me semble être le maître-mot de ton encadrement. Pertinence de tes conseils scientifiques évidemment, quand tu venais à compléter les intuitions de Stéphane avec une méthode toute mathématique. Pertinence de tes conseils professionnels, en m'exhortant sans relâche à garder le lien biologique. Mais pertinence de tes conseils humains aussi, que je garde bien présents à l'esprit. Charles Elton, en travaillant avec Lotka, a laissé échappé ces mots à son propos: "*Comme la plupart des mathématiciens, il mène le biologiste plein d'espoir au bord d'une mare, fait remarquer qu'une bonne technique de nage l'aidera beaucoup dans sa tâche, puis le pousse et le laisse là à se noyer*". Tu m'as montré que cette barrière inter-disciplinaire, pouvait, comme toute autre barrière, être dépassée. Tu m'as offert un masque, un tuba, des palmes, une bouée, et m'a bien fait comprendre qu'on est toujours libre de sauter ou pas dans la pataugeoire, et que, quoi qu'il advienne, il n'y a jamais d'échec. Merci.

Ambre, l'encadrante dans l'ombre, merci de m'avoir guidé sur les chemins tortueux du C++ moderne et du paradigme générique, sans rien attendre en retour. Je me souviens t'avoir rencontrée avec un code obscur, rigide, au bord de l'effondrement. Tu m'as appris à diviser, isoler, structurer, organiser, généraliser, abstraire: concevoir. Tu m'as appris qu'un bout de code pouvait être simple, beau, et émouvant aussi. Tu m'a appris que le C++ était une langue qui avait son propre rythme et sa propre poésie, ses propres paysages de conscience. Ton temps, tes conseils, ton amitié ont été des alliés précieux ces deux dernières années, et le Quetzal d'aujourd'hui te dois bien des plumes. Je n'oublierai pas ces heures passées chez toi, ni la sensation de ma

tête vacillant sous la densité de tes propos, ni le grisement né de l'imagination d'un dragonneau en plein vol (Fifi forever).

Merci à Olivier Dangles, au labex BASC et à la Chaire MMB pour les financements qui ont aidé à l'aboutissement de ce projet.

I wish to express all my thanks to Lacey Knowles and Solenn Stoeckel for having accepted to be rapporteurs of this thesis.

Merci à mon équipe et à mon laboratoire pour chacune des heures de chacun des jours que j'ai passé à EGCE. Laure, Catherine, vous êtes de remarquables reponsables hiérarchiques. Merci aux secrétaires pour leur travail admirable. Romain, mon cher co-bureau/détenu/galérien/locataire (rayer la mention inutile), merci pour avoir supporté mes feuilles volantes, mes piles de publis mal alignées, mes feutres étalés et mon sempiternel yaourt tiède. Andreas, Cécile, Hannah, Damien, Florian, Vincent et tous les autres, vous rencontrer a été une grande joie. Cher EGCE, merci pour ton amitié, ta tolérance, tes conseils. Tu vas me manquer et j'ai beaucoup appris. J'espère trouver pareille bienveillance à l'avenir.

Lucie, je ne sais pas bien comment te remercier de ces 371 heures. Sans doute que pour toi, ces quelques mots suffiront: Lapinou, Colton, FDBP, MCMC, Stephen, théière volante, pompier, "*e*" jaune comme euro, filtres à mâchoires, boldemort, monsieur tartine, majorrrr tom to grrround contrrrrol, nina talbet, tamalatamala, rise up, Derek et Léonin, et bien sûr ... Nolwenn (celui-là m'a pris du temps).

Flonasse & Jujuille, merci pour tous ces fou rires et ces bons moments. J'ai un joli bouquet de souvenirs à emporter en post-doc: 12 poussins empaillés, 12 petits pouces qui dansent et 12 frelons du nord. Merci pour votre soutien incessant (*mon beau sapiiiiin*), vos encouragements (*regarde, on dirait ta thèse !*), et votre patience (je ne vous en veux presque plus pour la fausse conférence et le piscine-gate).

Merci à Iryna, Alexandre, Elodie, Claire, Alix, Julien, Irène, pour la fidélité de votre amitié, et pour m'avoir accompagné toutes ces années, avec patience, confiance et bienveillance, à devenir ce que j'étais déjà.

Aymeric, Jordane: sans cette soirée rock il y a quatre ans, je n'aurais sans doute jamais osé faire de thèse.

Maïlys, Sacha, merci pour votre *support* (lol), la bassine restera toujours ma copine. Je saurai être un meilleur adc et gagner les défis.

Merci à Nathan, pour toutes ces pauses houblonnées parsemées de conseils en papillon et d'envolées métaphysiques aux touches de fatalisme russe.

Merci à mes compagnons de natation (Amandine, Alex, Jojo, Quentin ...) et de Water-Polo (Petit Putois, Fred, Clacla...) pour votre amitié, votre soutien, nos éclats de rire et d'eau mêlés. Didine la combine, promis on ira jouer des maracasses à Cuba chez Pepito (tchiktchikiboum). Merci au CAO pour l'apaisement que me procuraient les bassins. Mes plus plates excuses à toutes les personnes à qui j'aurais pu dire un jour "*je peux pas j'ai piscine*", personnes que je serais d'ailleurs bien en peine de citer

tant la liste doit être longue (mais qui inclue toutefois mes deux directeurs): pour toute réclamation, veuillez vous adresser au CAO sus-cité.

Merci aux copains du master B2E-parcours-écologie-et-biodiversité-mention-darwin-environnement-je-sais-plus-très-bien-les-qualificatifs. Copains de galère un jour, copains de galère toujours: Lucie, Alain, Marine, Rémi, Camille, Yoann, Blaise, Théo et Félise, c'est toujours bon de savoir que l'on pagaye ensemble.

Merci aux INHPiens, pour les bonheurs passés, présents, et à venir. Inès, Boris, Benoit, Julia, Fannoch, Antoine, Marion... Justin... Vous trottez et trotterez encore dans les recoins de mon cerveau comme au bord de l'étang Saint-Nicolas.

Chère smallah, je te dois tant. Papa, Maman, les fruits ne tombent jamais bien loin de l'arbre. Mais il arrive que certains roulent quelque temps pour germer plus loin. De la pédologie à la métaprogrammation pour la coalescence en paysage hétérogène, il y a un chemin qui garde votre empreinte. On revient toujours à ce qu'on aime. Ronan, merci de ton exemple, c'en est presque facile de marcher dans tes pas. Enora, merci d'être le lien indéfectible qui me ramène souvent sur Terre. Orane, merci d'être le petit soleil du matin qui récompense toutes les nuits de labeur (avec ou sans crème solaire IP 180). Merci à Lise et Mylio pour me permettre de voir plus loin devant, à Mimi et Grand-Père pour pouvoir regarder loin derrière, et à tous les autres pour me faire me sentir feuille parmi les feuilles d'un arbre bien plus grand.

Etre arbre. Un arbre ailé. Dénuder ses racines
Dans la terre puissante et les livrer au sol
Et quand, autour de nous, tout sera bien plus vaste,
Ouvrir en grand nos ailes et nous mettre à voler.

Pablo Neruda - Cuadernos de Temuco (1919-1920)

Contents

For Justin, who taught me perseverance, faith and friendship.

Chapter 1

Introduction

1.1 Global change and loss of biodiversity

There is growing evidence that the Earth system has recently moved away from the range of its natural variability during the last half million years: lands, polar regions, atmosphere, life, societies, oceans and other Earth system constituents have seen their state and the interactions between them dramatically altered (Pachauri et al., 2014).

The progressive understanding of this phenomenon started in 1980 with the World Climate Research Programme (WCRP) foundation, aiming at "a better understanding of the climate system and the causes of climate variability and change": it progressively became clear that climate change was only a part of a larger phenomenon, termed "global change", affecting all Earth system constituents at all scales (Vitousek, 1994). If in the past, planetary changes were driven by processes like volcanism, plate tectonic, solar variation, meteorits or variations in the orbital characteristics of the Earth, there is large evidence showing that the current changes are driven by human resources collection (Matson et al., 1997), transportation, transformation and waste production (Pachauri et al., 2014).

Human societies built on a given Earth system state: important cities were built on rather stable coastlines, population distribution levels followed water availability, agricultural systems and their related social systems developed accordingly to the local ecological properties (soil, water, climate, surrounding lifeforms). Global change is concerning for societies, because it questions the implicit assumption that this state will remain the same and consequently it actually questions our ability to adapt to a new Earth system trajectory: sea level rise threatens coastal cities development (Church and White, 2006; Nicholls and Cazenave, 2010), water resources

FIGURE 1.1: First results from Google for the *biodiversity* keyword as an illustration of the socio-cultural representation of the concept. Using variety of colors and forms are key in the concept representation. Species shown in pictures are attractive or remarkable, like lions, giraffes, orchids. In several pictures, moral values are associated to the concept, with the underlying idea that biodiversity is beneficial for the human well-being.

management systems need to adapt to the new context of global change (Pahl-Wostl, 2007), and agricultural systems have to face soils erosion (Montgomery, 2007), higher climate instability (Howden et al., 2007; Schlenker and Lobell, 2010), local extinction of some life forms and incoming of undesirable ones (Thrupp, 2000; Frison, Cherfas, and Hodgkin, 2011).

Among other important perturbations, the causes and consequences of the various life forms distribution modification across the planet has been widely studied, and raised remarkable awareness about the *biodiversity* concept at multiple levels of the society (Escobar, 1998; Spash et al., 2009). For non-scientific audiences, the term *biodiversity* is generally related to the abundance and diversity of life form: numerous, colorful, exotic and attractive species representations are central in communicating the concept (Figure 1.1). This diversity type (the species diversity) is only one part of the definition in the scientific representation of the concept, that also includes the diversity of the interactions between life forms and their environment, as well as the genetic diversity between individuals or populations.

Biodiversity brings many amenities (see Figure 1.2) called *ecosystem services* which value can be quantified (De Groot et al., 2012), like food production (Bommarco, Kleijn, and Potts, 2013), water purification (Van Houtven, Powers, and Pattanayak,

2007), pollination (Kremen et al., 2007), climate or disease regulation. Altering its state is generally considered as risky (Díaz et al., 2006). This naturally leads to the development of management programs for conversation, protection and restoration of ecosystems (Bullock et al., 2011). Consequently, considerable efforts have been made during the last decade to understand the past, present and future state of biodiversity (Dirzo and Raven, 2003), to decide what the reference state should be (Nielsen et al., 2007), to understand how human activities affect the system trajectory (Hooper et al., 2005) and to propose predictive scenarios for decision-makers (Pereira et al., 2010; Fisher, Turner, and Morling, 2009). Defining a reference state is uneasy, because systems are inherently dynamics (Blois et al., 2013).

For example, there have been in the past spectacular shifts in the distribution of biological diversity, known as massive extinctions. About 375–360 millions years ago, the apparition of the first massive photosynthesizing forests on the continents may have caused a global cooling by removing carbon dioxide (a greenhouse gas) from the atmosphere: interestingly, the apparition of our dearest forests coincide with the extinction of 75% of the former species (Algeo, Scheckler, and Maynard, 2001).

The colonization process (when some individuals reach a new territory, settle and proliferate) is a fundamental process by which biodiversity is redistributed: all species witnessed, to some extent, a modification of its distribution. However, the rate and scale at which this process occurs increased dramatically during the last decades, mainly due to human activities (Vitousek et al., 1997). As modern advances in the transport technologies allow to travel further, faster and more often, biological material can easily be spread all around the world, like plankton transported along sea routes in the ships ballasts. For some cases and under some circumstances, this phenomenon is perceived negatively by the society, that then uses the term of *biological invasion* to designate it.

1.2 Biological invasions

1.2.1 Definition

There has been some debate around the *invasive species* definition (see *e.g.* Valéry et al., 2008; Shah and Shaanker, 2014), certainly because the desired degree of generalization of the concept is very high (*i.e.* and that it aims at representing a very large

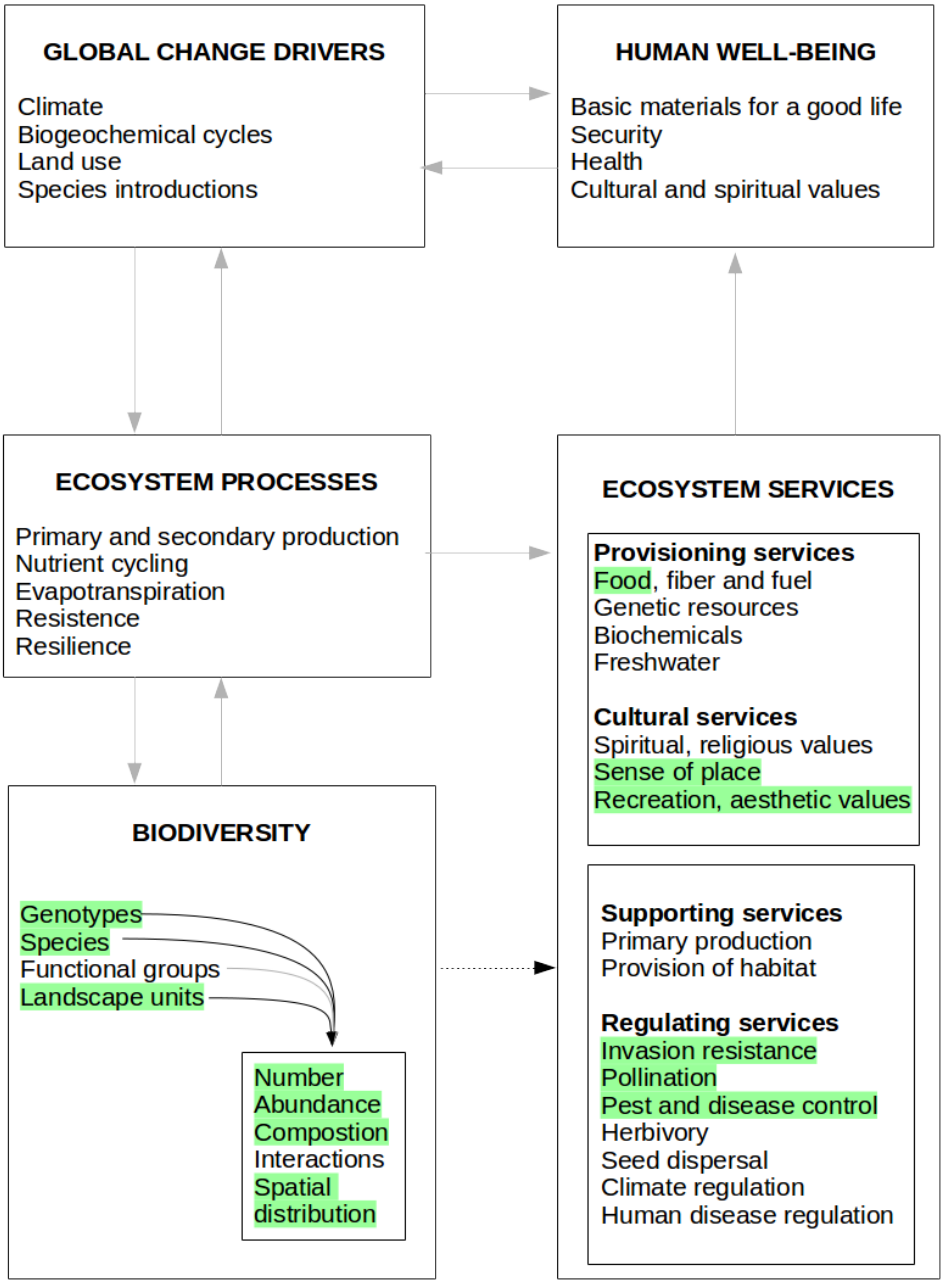


FIGURE 1.2: Relationships between biodiversity and human well-being. Biodiversity is affected by global change drivers, but is as well a factor affecting the levels of human well-being. Regulating services are obtained from ecosystem processes regulation. Supporting services are necessary for all other services to be provided. Dark arrows (resp. highlighted text) indicate the links (resp. the concepts) that are tightly related to the thesis: the invasion of *Vespa velutina* results in a modification of the biodiversity components state in terms of number, abundance, composition, spatial distribution of species and genes, the modalities of which can vary according the landscape local features and must be elucidated. The dark dashed arrow indicates the links that motivates the thesis, but that are not directly answered by the research questions: the predation behavior of *Vespa velutina* threatens the pollinators, especially the local honeybee *Apis mellifera* that provides honey and cultural amenities. Consequently this invasion process causes fears, worries and passionate reactions at various levels of the society. Modified from Díaz et al. (2006) and Dıaz et al. (2005).

variety of complex and particular instances of the phenomenon). Consequently the concepts used in any of its possible definitions are very abstract and subjects to interpretation and subjectivity (Valéry, Fritz, and Lefeuvre, 2013; Colautti and Richardson, 2009).

While some definitions stress out the importance to account for the impact of invasive populations in the new area, others insist on the underlying dispersal pattern, as the following one.

Definition 1.2.1. (Biological invasion, Valéry et al., 2008) A biological invasion consists in a species acquiring a competitive advantage following the disappearance of natural obstacles to its proliferation, which allows it to spread rapidly and to conquer novel areas within recipient ecosystems in which it becomes a dominant population.

As stated by Wilson et al. (2009), definition 1.2.1 fails to identify an intuitive concept. For example, definition 1.2.1 assumes population dominance, but this can not be guaranteed. Indeed, even what we intuitively would like to class as invasive species can suddenly undergo a dramatic population size reduction, and even witness repeated collapses (Simberloff and Gibbons, 2004).

As a number of points debated in the biological invasion literature are irrelevant in our context study, we prefer to use definition 1.2.2.

Definition 1.2.2. (Invasive population, Estoup and Guillemaud, 2010) Set of individuals that has been introduced into a new area, in which these individuals have established themselves, increased in number and spread geographically.

1.2.2 The yellow legged hornet invasion in Europe

Generalities

The dramatic development of the yellow-legged hornet (*Vespa velutina*) in Europe is a remarkable example of biological invasion. *Vespa velutina* is a social insect originating from Asia (Perrard et al., 2014, Figure 1.3).

There is reasonable evidence that the first queens arrived in Europe at a single point (Nerac, South-West France) in 2004, as Chinese bonsai poteries were imported by a local horticulturist (Villemant, Haxaire, and Streito, 2006b; Villemant, Haxaire, and Streito, 2006a). According to genetic studies (Arca et al., 2015), the introduced individuals originate from the provinces of Jiangsu and Zhejiang (China).

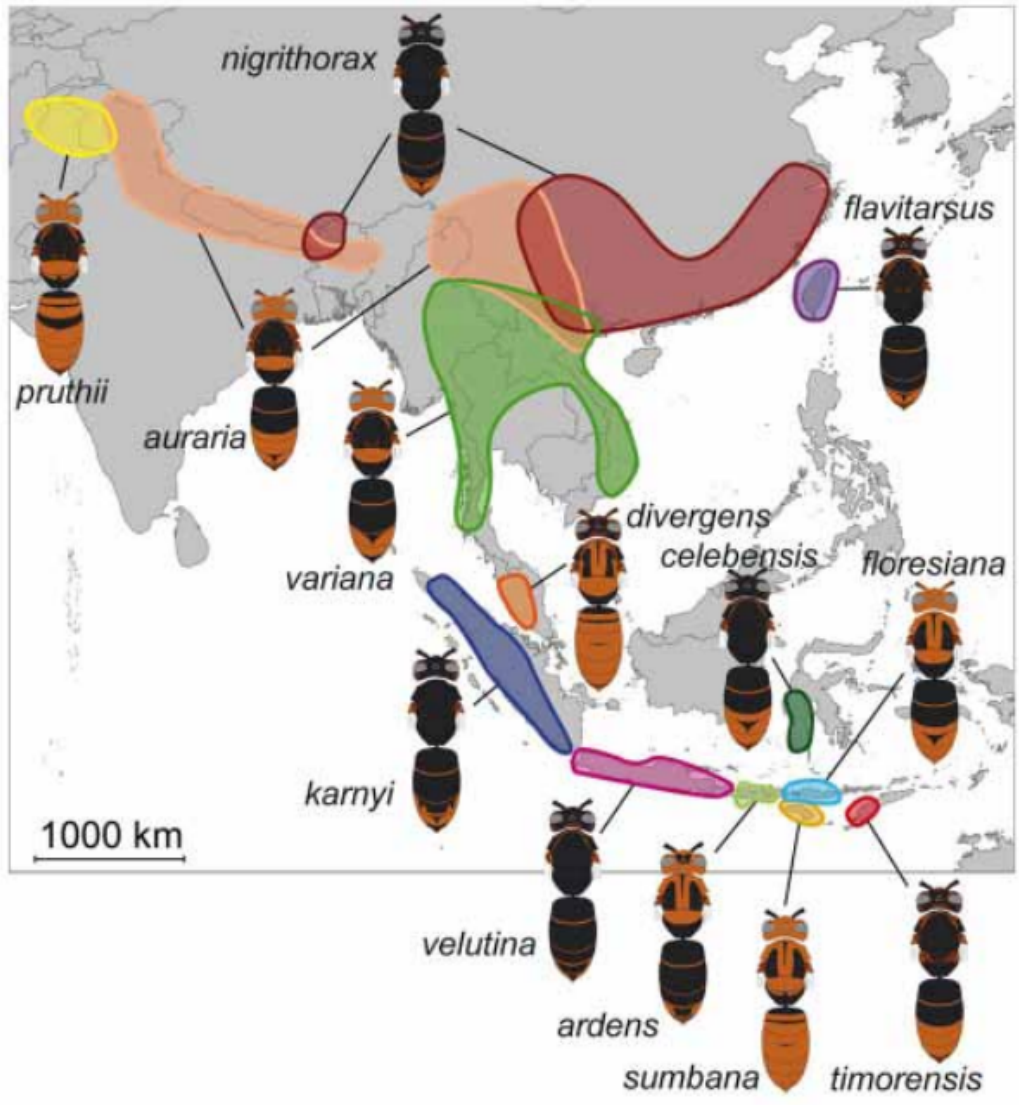


FIGURE 1.3: Known distribution of the different colour morphs of Vespa velutina across south-east Asia, borrowed from Perrard et al. (2014)

Subsequently, colonies expanded rapidly, reaching Spain (López, González, and Goldarazena, 2011), Portugal (Grosso-Silva and Maia, 2012), Belgium (Rome et al., 2013), Italy (Bertolino et al., 2016) and Great-Britain (Keeling et al., 2017).

Life cycle

Nests are initiated by young females, called foundresses. They start the construction of a structure large enough to allow the development of some eggs into grown workers able to expand the nest and to collect food. In the fall, young foundresses are mated and overwinter. The colony dies as winter comes. In spring, the foundresses disperse and found new nests. Using mills experiments where foundresses are attached to a mill for measurement of flight duration, it has been shown that they could actually fly an average of 18 km/day, and possibly fly over 200 km over 10 days (Robinet, Suppo, and Darrouzet, 2017). The importance of both human-mediated dispersal and self-mediated dispersal has been stressed out, as foundresses are suspected to be able to disperse along long distances, possibly transported by human means (Robinet, Suppo, and Darrouzet, 2017). These dispersal features could explain the rapid rate at which the yellow-legged hornet spread in France (around 100 km/year) and why various nests have been found more than 200 km from the invasion front (Villemant et al., 2011; Robinet, Suppo, and Darrouzet, 2017). Notably, three locations at which colonization have been observed are considered as resulting from long-distance dispersal events (Villemant et al., 2011).

Predation behavior

V. velutina predates various pollinators in its native area, including the local honey bee (*Apis cerana*). In Europe, its full impact on ecosystems is still not assessed, but it remains clear that *V. velutina* predates *Apis mellifera*, that does not show the defense and avoidance behaviors that allow the Asian bee populations to resist the predation pressure. Consequently, beekeepers are worried about the range and impact of the ongoing expansion, especially since a large part of Europe appears to be favorable to the development of new colonies (Villemant et al., 2011).

Questions of interest

The invasion of *V. velutina* raises a variety of questions implying multiple theoretical approaches and data to answer them (see Figure 1.4).

Introduction points Perhaps the most important point is to be able to predict its future range expansion. Accordingly, it requires first to know if the incoming of individuals for Asia is frequent or not, that is to test if the hypothesis of a unique introduction site is relevant. If not, the spatio-temporal coordinates of the multiple introduction points should be estimated.

Dispersal Then, to predict the future of the invasive process, the various dispersal modes should be elucidated. Dispersal distance, that is the geographic distance between 'start' and 'end' points of a dispersal event, is fundamental to describe the dispersal process (see Nathan et al., 2012, for a review). A large number of mathematical descriptions of the distribution of the dispersal distance (dispersal kernel) exists (Nathan et al., 2012). Furthermore, if geographic distance is the most basic spatial descriptor of dispersal, the effect of landscape heterogeneity on dispersal patterns should not be underestimated, since *V. velutina* seems very sensitive to the opening up of the environment and possibly follows rivers and valleys during the dispersal phase (personal communication of Claire Villemant). Several GIS-based approaches like least cost path method and circuit theory allow to identify dispersal corridors across the landscape, given habitat heterogeneity as a friction layer using geospatial and environmental data (Vignieri, 2005; McRae and Beier, 2007).

Population growth patterns The relationship between populations growth and landscape heterogeneity should be investigated. There is a large number of ways to describe this growth process, including using logistic growth models (Currat, Ray, and Excoffier, 2004a; Estoup et al., 2010a) and using species distribution modeling to synthesize the landscape heterogeneity into habitat suitability (He, Edwards, and Knowles, 2013a). Importantly, as climate change would positively affect the habitat quality of European territories for *V. velutina* (Villemant et al., 2011; Barbet-Massin et al., 2013), accounting for the landscape spatio-temporal heterogeneity is fundamental to correctly predict the species' future distribution patterns.

Objectives The present thesis does not aim to answer directly these biological questions. Instead of choosing one sub-question and the related model special case to develop the corresponding methodology (with limited extensibility), the thesis takes the decision to first develop a general framework into which the multiplicity of models and data can be efficiently handled by the research community members

(see Figure 1.5). The application of this framework to the special case of the Asian hornet is an ongoing work that is not presented here, even if the methodologies presented here are rooted in this context. Hopefully, this framework will allow researchers to be more efficient while analyzing data with the statistical models that are most suited to their question of interest.

1.3 Need for prediction: statistical modelling

1.3.1 Intuition of inferencial processes

Being able to predict phenomenons implies to have collected and processed some form of information related to them, that is to have gained some *knowledge* about the system we are trying to forecast. This information can be direct observations of several instances of the process, or more indirect forms of knowledge.

For example, if a magician asks me to predict the color of the first rabbit he will draw out of his hat, and that I have never attended to one of his shows, then I actually have no direct observation to use to forecast the next outcome of this magic trick. However, since I am an average (*i.e.* bayesian) human being, I can entirely rely on the knowledge I have *a priori* of this magical random experience. Interestingly, through my past experience of magic tricks, and through social representations of rabbit magical sampling (movies, books, pictures), I have actually considerable information concerning this process: *usually*, the magician draws a white rabbit out of his hat. Consequently I will try the *white rabbit* answer, because it seems to me the most likely output.

Interestingly, Tarzan, coming from his jungle, and knowing nothing about magicians, hats nor rabbits, has no clue of the correct answer, as he has no *prior knowledge* about this magical process.

The magician laughs at hearing our answers, and he begins to draw a small series of rabbits:

$$\{pink, pink, red, pink, pink\}$$

At looking the observations, it first comes to my mind that this strange magician radically annihilates my *a priori* representation of a traditional rabbit magical

sampling. As the magician ensures us that there is no link between the color of successive rabbits, Tarzan and I quickly agree on betting on *pink* as being the color of the next rabbit.

Then the magician, frustrated by our previous success, defies us to predict the behavior of another hat, where entire groups of polychromous rabbits are randomly sampled out of the hat. As a hint, he tells us that the colors proportion in each group is *in some way* related to the hat temperature oscillations that have happened 10 seconds before the sampling, while mixing each rabbit fur colors should yield to white. With a sardonic smile, he then pulls out a series of two groups of rabbit, and asks to Tarzan and me to guess the next group configuration.

"What a strange magician, . . . " whispers Tarzan to me, *". . . we should look for a mathematician."*. He is right. Or for a statistician. Or both. Guessing accurately the next rabbit group configuration implies analyzing complex relationship between multiple aspects (dimensions) of the process (time, temperature, ...) and multiple aspects (dimensions) of the data we observe (each individuals color composition). Tarzan's brain and mine are totally overwhelmed by these details.

When data and the process generating them are simple enough (one monochrome rabbit is drawn out of a unique hat), our mental performance is often enough to establish sound conclusions of some observations and gain intuition about the characteristics of the future observations. However, when things become trickier our intuition for future observations is rapidly overwhelmed.

Here comes the need for mathematical modeling and statistical methods, that will help us to formulate the problem into clear terms by forgetting minor details, and to carefully manipulate these terms in such a way that finally solid conclusions and predictions can be drawed from observations by running some computations.

1.3.2 Example : infering a demographic feature

The biological problem

Consider now a more biological (but less funny) example: a biologist daring me to guess the number of children of a random female in the population. First he tests me about the European human population. I can easily estimate it from my social experience: I answer 2-3 by remembering and averaging over the various family examples I met during my life. The biologist seems satisfied by my answer, and he

moves to the yellow-legged hornet population: as I have no available prior information, I ask for some data to update my knowledge. Nicely, the biologist allows me to access its hornet database, where he carefully recorded independent observations of the number of eggs laid by 3 hornet queens he collected in the same forest:

x_1	x_2	x_3
12	8	16

A possible mathematical representation

The first step is to formalize the intuition we have of the biological process underlying these data. We are looking for a way to define a little better our intuition, but staying vague enough for not introducing unjustified specifications.

A reasonable assumption is that since the queens are issued from the same population, they likely share same biological properties, including fecundity: we assume that there is not structure or fundamental differences between individuals, and that queens have a fixed life time.

It is not impossible for a female to be sterile, that is not to be able to lay any egg, but from what we know of insect queens, it is unlikely. On the other hand, being confronted to a super-queen able to lay billions of eggs looks like something out of a science fiction movie, and sounds a bit too dramatic for a realistic situation. And we have the intuition that in a given time interval, a particular queen i, with her defined physiological state, energy reserve, stress level, will not lay *any* number of eggs, but rather that she is likely to lay a certain number of eggs (call it λ).

- x is the number of times an egg is layed in the queen life time.
- We assume that the fact that an egg is layed does not affect the probability that a second egg will be layed later: laying events occur independently.
- The rate at which eggs are layed occur is constant. The rate cannot be higher at some periods of time and lower in other periods.
- Two eggs cannot be layed at exactly the same instant; instead, at each very small sub-interval of time exactly one event either occurs or does not occur.
- The probability of an event in a small sub-interval of time is proportional to the length of the sub-interval.

The average number of eggs in the lifetime is designated λ: it is the rate at which eggs are layed. In this model, the λ value can be seen as the fecundity: *i.e* this is a possible, non-unique way to mathematically precise the rather abstract concept of fecundity.

The probability of observing x events in an interval is given by the equation:

$$P(x) = \frac{\lambda^x}{x!}e^{-\lambda}$$

Knowing λ, I can draw the unique associated form. As we are free to change the λ value, this formula still captures an infinite number of forms. The inference problem is to find, among this variety of forms, the one that is most consistant with the observation, that is, to estimate *lambda*.

The likelihood function

The inferencial method is the way we draw conclusions about λ when looking at the data. Intuitively, we can imagine λ as a tuning button of the process (model) that generates the data: changing the tuning changes the properties of the data that are generated, and observing the properties of the data give insights about the tuning. Consequently, it is rather instinctive to look for the value of lambda that maximizes the probability of observing the data. Finding the maximum of a function is an optimization problem we learnt to solve in highschool.

The first step is to formalize what is meant by *"the probability of the data"*. Call this function L (in the statistical literature, it is known as a *likelihood function*). Remember x_i the number of eggs layed by the female i. As we saw in highschool, since all x_i are independent, then the probability of observing the sequence of data $(x_1, \dots x_n)$ is the product of the probability of observing each x_i.

$$L(x_1, \dots x_n; \lambda) = \prod_{i=1}^{n} \frac{\lambda_i^x}{x_i!}e^{-\lambda} = e^{-n\lambda}\prod_{i=1}^{n}\frac{\lambda_i^x}{x_i!}$$

If L is differentiable (this is not always the case), then we learnt that finding the maxima (or minima) of a function can be done by studying the properties of its derivative. We are actually looking for the value of λ for which the derivative is null (graphically, this represents the point at which the curve slope becomes horizontal) that is to solve the following equation:

$$\frac{dL(x_1, \dots x_n; \lambda)}{d\lambda} = 0$$

As the derivative of a product does not allow comfortable manipulation of the formula, this is equivalent (and simpler) to maximize the natural logarithm of the likelihood (the likelihood being positive), because the logarithm properties allow to transform the product into a sum:

$$\begin{aligned}
\ln L(x_1, ..., x_i, ..., x_n; \lambda) &= \ln e^{-\lambda n} + \ln \prod_{i=1}^{n} \frac{\lambda^{x_i}}{x_i!} \\
&= -\lambda n + \sum_{i=1}^{n} \ln \frac{\lambda^{x_i}}{x_i!} \\
&= -\lambda n + \ln \lambda \sum_{i=1}^{n} x_i - \sum_{i=1}^{n} \ln(x_i!)
\end{aligned}$$

Finding the value of λ for which the first derivative is null is done by solving the following equation:

$$\frac{dlnL(x_1, \dots x_n; \lambda)}{d\lambda} = 0$$

For which the solution is:

$$\hat{\lambda} = \frac{\sum_{i=1}^{n} x_i}{n}$$

So the point $\hat{\lambda}$ is an extremum. The study of the second derivative allows to precise if it is a maximum or a minimum of the function.

$$\frac{\partial^2 \ln L(x_1, ..., x_i, ..., x_n; \lambda)}{\partial \lambda^2} = -\frac{\sum_{i=1}^{n} x_i}{\lambda^2} \leq 0$$

As this expression is always negative, then $\hat{\lambda}$ is a maximum of the likelihood function. The probability of the data $(x_1, \dots x_n)$ is maximal when the λ parameter of the model equates $\frac{\sum_{i=1}^{n} x_i}{n}$. We just find a maximum likelihood estimator of the fecundity (it is normal to find it equal to the empirical mean). So finally, I can answer to the biologist, that under our hypothesis, we estimate that the fecundity of the yellow-legged hornet queens is 12.

What for more complex data and models ?

The previous simple data and model do not really reflect the complexity of real-world biological problems.

In the case of the invasion of the yellow-legged hornet, of course that we have the intuition that it should exist some kind of *fecundity*, but we have reasons to believe that the fecundity will depend on the location of the queen. For example, we can reasonably expect a queen installed in a luxuriant forest to be more prolific than a queen lost in a cold and rocky mountain. It means that the numbers of eggs of queens sampled at different locations are sampled in different distributions: this complicates the expression of the likelihood function.

Moreover, obtaining direct data such as the number of children of a yellow-legged hornet is difficult, if not impossible. It is feasible to have access to the number of eggs laid during a short period of time by a queen in its natural conditions, by opening the nest and counting the number of children in each life stage (egg, larva, pupa). It is feasible to directly study the laying dynamics of a queen in a correct laboratory experiment, for example with a continuous camera recording of the laying process. However, in natural conditions, this kind of data are hardly accessible.

On the other hand, as biological invasions are dynamical processes, a more accurate picture of the process can be given by a spatio-temporal sampling (collecting various data a various times and at different locations). But doing so, it introduces spatial and temporal dependencies between the observed data. For example, the number of hornets observed at Bordeaux in 2006 is highly correlated with the number of hornets observed in its suburbs in 2005. Such dependencies means that we can not easily consider the probability of the observations set as being the product of each observation. This complicates again the likelihood function.

Sometimes the likelihood function simply does not exist, or maximum likelihood estimates of the parameters do not exist (for example if L is not differentiable).

1.4 The demogenetic approach

1.4.1 Problem: data scarcity makes it difficult to inform models

Mathematical models are important to predict the invasion process. But the data scarcity concerning the yellow-legged hornet invasion makes it difficult to build and

inform realistic models, especially concerning growth and spread patterns (Keeling et al., 2017). For example, Keeling et al. (2017) used a stochastic yearly time-step difference equation to predict the spread of Asian hornet in Great Britain. The model assumes that each nest produces a Poisson distributed number of new founders (that is a measure of the reproductive potential r). The mean of the Poisson distribution is reduced by different factors, including a competition term C, the local environmental suitability E and the latitude of the nest considered as a climate proxy. They assume a linear decrease in the mean of queens per nest with latitude, acknowledging that there is little data support for this hypothesis. Furthermore, some parameters are estimated using empirical data, but the choice of precise values for several key parameters (foraging behavior parameters and mean flight distance) comes with little justification, possibly because there is little data support, or because infering these parameters is difficult in this modeling framework.

1.4.2 Solution: incorporate genetic data in the analysis

Using genetic data has been proved to bring invaluable support in the analysis of past demographic events: integrating genetic data in species range dynamics allows for more robust predictions of species responses to environmental changes (Fordham et al., 2014). Indeed, the demographic history of species to environmental changes has left marks in the genes (Marske, Rahbek, and Nogués-Bravo, 2013): present genetic data can then be linked to past ecological processes by coupling demographic models accounting for the spatio-temporal landscape heterogeneity with genetic variation models. Genetic models based on coalescent approaches (Nordborg, 2001; Hein, Schierup, and Wiuf, 2004; Wakeley, 2009) are useful when the studied genetic variation is neutral.

The coalescence of two gene copies into a parent copy is simply the replication of the ADN viewed backward in time. The genealogy of the sampled genes copies can be defined backward in time conditionally to the demographic process which itself can be defined before tackling genetical aspects. This is an important theoretical link between a genetic sample and the historical processes that shaped it, and it can be used for constructing statistical models allowing to estimate properties of these past processes on the basis of the present sample.

Constructing such estimates often relies on the study of the likelihood. The likelihood function can be derived under simple coalescence models, but as theoretical advances steered models towards higher levels of complexity (migration (Beerli and Felsenstein, 1999), recombination (Kuhner, Yamato, and Felsenstein, 2000), selection), the likelihood function became harder and harder to calculate, especially because the sampled genes are not independent (they share a common genealogical history), and because considering spatial heterogeneity dramatically increases the likelihood complexity.

1.4.3 Bypass the likelihood function analysis with ABC

Approximate Bayesian Computation (ABC) methods (see Marin et al., 2012, for a review) has considerably extended the range of Ecology and Evolution models under which inference was possible (Beaumont, 2010; Csilléry et al., 2010). ABC bypasses the complex task of evaluating the likelihood function by combining two approximations making the problem computationally tractable: (i) observed data are reduced to lower-dimensional quantities (the so-called summary statistics), (ii) the inference is tolerant to small distortions of the observed summary statistics. More formal explanations can be found in Blum et al. 2013 and Section 4.1.1.

These approximations make possible for ABC procedures to estimate posterior densities of the parameters by simulating data under the model while exploring the parameter space conditionally to a prior distribution, and accepting only the values of the parameters for which simulated data are close enough to the observations. Despite its apparent ease, ABC methods present important methodological pitfalls (one of them is the choice of the dimensional reduction function), but many studies have paved the way for the non-statisticians (see Bertorelle, Benazzo, and Mona, 2010, for an excellent methodological guide).

1.4.4 On the need of simulation resources

The popularity of ABC methods encouraged the development of more complex coalescence-based simulation computer programs, and their authors put remarkable efforts in successfully delivering novative, usefull and user-friendly products to the community of population geneticists. SPLATCHE (Currat, Ray, and Excoffier, 2004b) simulates coalescents based on complex demographic simulations in a spatially explicit landscape, incorporating landscape heterogeneity. Various versions

of SPLATCHE largely fostered the rapid expansion of the so-coined iDDC modeling approach (integrated distributional, demographic and coalescent modeling, He, Edwards, and Knowles 2013b). iDDC uses Approximate Bayesian Computation with spatially explicit demographic simulation model (possibly integrating landscape heterogeneity) to estimate quantities of interest such as populations growth rate or dispersal law parameters (Lacey Knowles and Alvarado-Serrano, 2010; Estoup et al., 2010b; Massatti and Knowles, 2016). DIY ABC (Cornuet et al., 2014) is an open-source program that provides the ability to conduct inference under a wide range of complex biological scenarios combining an arbitrary number of admixture, divergence or demographic change events. It offers very strong ABC support and an intuitive Graphic User Interface (GUI). IBDsim (Leblois, Estoup, and Rousset, 2009) is an open-source program for simulating genetic variation under isolation by distance, and provides much flexibility in the choice of dispersal kernels. MSMS (Ewing and Hermisson, 2010) puts emphasis on incorporating selection and proposing extensible design. These programs, and others, have provided invaluable support to the non-developer communities for a wide range of applications and studies.

However, most of these programs were aimed at biologists rather than at computer developers: consequently they are much more black boxes than plateforms for future development. For example, SPLATCHE does not allow to change the dispersal kernel or the local growth model, and most of programs aim to write only standard genetic summary statistics in output files. This current state of the art does not scale with the virtually infinite number of arbitrarily complex evolutionary or demographic models and the ever-growing number of statistical methods variants. We need standard, general, reusable tools for helping us to quickly build programs that can simulate and analyze new models.

1.5 Thesis outline

The research presented here focuses on studying the contemporaneous demographic history of invasive populations, using coalescence approaches and simulation methods and ABC to draw inference from genetic data collected on the field and taking advantage of possible theoretical assumptions (see Figure 1.5). The project is especially focused on studying how spatial and temporal landscape heterogeneity affects

patterns of growth and dispersion by coupling niche models with the considered demographic models.

1.5.1 Chapter 2

With no existing reusable codebase, simulating such new coalescence models is a task of a daunting magnitude. That is why time has mainly been invested in the development of Quetzal, an open-source C++ library designed to ease the implementation of a wide range of spatially explicit coalescence-based stochastic models. It now allows to simulate new models of invasions and to test new methods of inference. However, the journey towards this acceptable software solution has been long, tortuous and full of pitfalls. Considerable time has been lost in writing code that experienced programmers would have known doomed at first sight. Considerable time has been spent in acquiring the required expertise with the software engineering community, of which our own biology-oriented community is little aware. It is counter-intuitively hard to precisely identify what is bad in codes, mainly because codes can be considered as *bad* even if it actually works. A bad code (even without bugs) provokes bad feelings when working with it, because the code is cryptic, or complicated, or too long... The community uses the word *bad smell* to designate such bad feelings, a term defined by Martin Fowler as *"a code smell is a surface indication that usually corresponds to a deeper problem in the system"*. Code smells are not bugs, but rather certain structures in the code that slow development and increase the risk of future bugs. The reasons of many bad feelings we can have when writing or reading code have been identified and solved years ago by developpers, who have much to share about. The intended objective of this chapter is to be, if not a bridge between two worlds, at least a trail of breadcrumbs left for the following biologist across the jungle of the programming problems and solutions.

As models comparison is a corner stone of ABC, it is very important to be aware of the multiplicity of the possible models of coalescence to pass on a same dataset. And at the same time, the community of researchers can not lose time in continuously re-implementing programs from scratch each time a new model version is needed. Yet, multiplicity is very challenging to tackle when writing the code, because it requires to express things in a programming language in such a way that the following objectives are met:

- simulate the envisioned models
- keep the door open for unplanned models
- propose an easy replacement or reuse of code components
- allow the behavior of a component to be customized without having to modify its code
- guaranty acceptable performances

The Chapter 2 takes the position that that these objectives can be achieved by investing efforts in learning advanced programming languages features to design not programs, but generic tools helping to write particular instances of programs. It is advocated that writing code in terms of the most basic language features is frustrating, disappointing, confusing and deceptive in the long-run, both for the programmer and its community. In contrast, advanced language features allow to express things in a clear, flexible and secured way. Amusingly, far from being unnecessarily technical, learning to write high-quality code is more of a philosophical and conceptual adventure: after all programming languages are still languages, and as such they allow to create words to express ideas, concepts, and communicate subtle intentions to the programmer's first interlocutor: the machine. Consequently, many charms of natural languages manipulation find in some way an equivalent in programming languages, and the programming activity can be funny and recreative: *computer programming is an art, because it applies accumulated knowledge to the world, because it requires skill and ingenuity, and especially because it produces objects of beauty.* (Donald Knuth, 1974). We explore further these similarities in Chapter 2.

Section 2.1 presents very general thoughts about natural languages and how they allow us to design and manipulate abstract concepts. Key concepts are creation of new words, generalization by loss of details, and progressive accumulation of abstraction. Section 2.2 presents the principal characteristics of a programming language, and the fundamental importance of designing collections of small code components (called *library*) that can be combined in more complex components. Ideas exposed in these two first sections are non-technical, simple enough for any reader, and essential to understand how programming languages imitate natural languages ways to allow elegant forms of expression .

Section 2.3 is written as a step-by-step guide for the beginner to understand how a programming language can be used to express ideas that are closer from the human thought than from the machine representations. As concrete examples were needed to illustrate ideas, the C++ language is introduced. As a C++ tutorial is out of the scope of this thesis, and even if the code was intended in its simplest form, the novice will surely meet unknown code syntax or vocabulary. This is generally not too important, as we hope that reading this section will enable the reader to step-aside from the cliché that code as to be a never-ending succession of details and to appreciate it more like a literary form of expression.

Section 2.4 is written to help the programmer to identify a common problem in the programming activity, namely the frequency at which objectives change, and to understand both that this instability is inherent to its activity, and that without precautions it can have disastrous effects on the code base. This effects are presented here as symptoms of a bad software design, for which solutions are presented in section 2.5 as a well-known set of five engineering principles (the SOLID principles). This section is inevitably abstract and technical, and consequently will surely make little sense to the novice: it still remains of fundamental important for those who are interested in identifying and solving bad design effects in their code base.

Section 2.6 is in substance the article draft presenting Quetzal, the C++ template library for coalescence. Quetzal as a set of thoroughly tested generic components with extensive documentation, which algorithms and data structures can be parametrized to match any user-specific context, and can be freely combined to simulate an open-ended number of coalescence models accounting for arbitrary user-defined dispersal kernel, environment, growth models or data structures. Quetzal integrates ABC components to embed efficiently any simulation model for simpler and faster analysis. Using Quetzal, simpler, safer, faster code can be written in hundreds of lines instead of thousands. The section briefly justifies the key abstractions as design decisions driving the code production, and illustrates the flexibility of the code components through various application examples.

1.5.2 Chapter 3

Most current statistical tools aiming to estimate demography are based on the Kingman coalescent, that neglects the probability that more than two lineages merge during a coalescence event (Kingman, 1982): this assumption holds if the number

of lineages is much smaller than the number of parents. However, there is a large range of problems for which this assumption does not hold, because the population size is not constant in time.

For example, several programs (notably various versions of SPLATCHE, Currat, Ray, and Excoffier 2004a) implementing one or another variation of an environmental demogenetic model have been used in the context of bioinvasions. In this context, the population size distribution over space and time is constructed by some complex stochastic process accounting for environmental dynamics, and is not meant to be directly estimated. A limited number of introduced individuals spread and reproduce across the landscape, so N_x^t, the population size in deme x at time t, is highly stochastic. If the population size depends on parameters to estimate, the random parameter sampling which is part of the ABC procedures makes it harder to guarantee that $n_x^t << N_x^t$. Actually, at some point of the simulation (for example for long-distance dispersal events) N_x^t can be close to 1. Obviously the probability that more than two gene copies have the same parent is no more negligible, so a binary merge algorithm (BMA) is irrelevant and a simultaneous multiple merge algorithm (SMMA) should be used, because it improves demographic estimations (Montano, 2016).

Current implementations do not allow to flexibly and efficiently change simulation features accordingly to the theoretical framework. The first part of the chapter 3 presents the programming techniques that allow the simulation strategies to be decided before the program even runs. Briefly, it is actually possible for the programmer to express choices (that is, to pass options) that are evaluated at compile time (and not at run time!): this enables the compiler to rewrite the code in an optimal way, leading to very high performances.

Section 3.1 presents the design of small algorithms able to build genealogies under (i) the multiple merge coalescence hypothesis or (ii) the binary merge coalescence hypothesis. We hope that they can be considered as standard generic implementations and easily reused. We used standard generic paradigm techniques to enable the user to use the genealogical object type that is most suited to its context and manipulate it accordingly. As these algorithms are expected to be used in the deepest parts of the programs, changes in the coalescence hypothesis can have great impacts in the whole code stability minimized whenever the simulation hypothesis change. We anticipate this by designing the algorithms as small components with

a common interface allowing them to be freely exchangeable at compile-time. This common interface is explained in section 3.1.2. We believe that the programing elements presented in this section can be useful for the C++ beginner willing to add flexibility in its code, or for the potential user of the library. The algorithms fulfill the standard logic of the modern C++ approach, what is beneficial for the users.

Section 3.1.3 (respectively section 3.1.4) presents the behavior and the C++ implementation of a binary merge (respectively simultaneous multiple merge) coalescence event. Their reading can be enlightening to fully understand how the algorithms work (that is, how operations are performed), and to understand the programming techniques that enable the user to freely define the exact nature of these operations (that is, to achieve true genericity).

Section 3.1.5 presents guidelines for the developer willing to write code that can adapt to changes in the coalescence model hypothesis, that is writing code where switching from BMA to SMMA is straightforward and requires no code modification.

While designing these algorithms, a small simulation test program was designed to check the validity of the estimation method presented in Chapter 4, section 4.2.8: the true parameter estimation completely failed. Coalescence with multiple merges is known to dramatically slow down the simulation (what is opposed to the performance constraint imposed by ABC procedures), so I assessed that more simulations were needed for the posterior to converge and I began to search if some reasonable optimization was possible in the simulation of the coalescence events. It appeared that instead of assigning a parent to each coalescence line then merge lines with same parent, it was possible to directly sample a coalescence configuration in a precomputed, memoized probability distribution, and I designed a new algorithm to generate the distribution support. This new algorithm fixed the error and allowed correct estimation, but for the wrong reason: it appeared some time later that the initial algorithm was in fact bugged ! However, it was not in vain, as the design effort required here allowed important enhancements in the coalescence algorithms genericity, and that the new version of the coalescence algorithm leads to 50% faster simulations in our study. The rest of the chapter 3 (section 3.2) relates this little adventure.

1.5.3 Chapter 4

Usually in ABC coalescent studies of complex demographic processes, the model parameters inference is made possible by finding the sampled genes most recent common ancestor (MRCA) through the reconstruction of their genealogy. Doing so, the modeler has to account for an history that can be far more ancient than the processes of interest: the MRCA can indeed be found in a remote spatio-temporal window, leading to a number of problems that are related to the two ABC main approximations. These approximations are detailed in section 4.1.1.

First the quality and the quantity of available information drop when going backward in time: it will result difficult to reliably inform the model, whether for model specification (prior distributions, model hypothesis) or data (availability and consistency).

Second the rejection rate of ABC will unnecessarily increase since the remote-time topology of the coalescent strongly conditions the data likelihood: a genealogy which bottom topology is consistent with observations can be rejected if its top topological properties are inconsistent with the observations. This is a waste of computational time because respectively to the recent history process, it should have been accepted.

To avoid the simulation of the top genealogy, a solution would be to randomly draw the allelic state of ancestors before the MRCA is found. This is not satisfying as generally the prior distribution of the states is unknown, with little information available in the literature. Furthermore, infering the ancestral allelic states would increase the dimensionality of the model.

The idea developed here is to build on the very recent genealogical clustering process to assess its consistency with the observed genetic clustering. Whenever the non-mutation hypothesis holds, it allows to conserve the anonymous nature of the allelic states, as they are conserved along the bottom branches fragments. The non-mutation hypothesis is justified in section 4.2.1

By stating that as the mutational process can be negligible compared to the recent genealogical process in shaping the sample configuration, we can consider that data at one locus actually entirely reflects a partition of the dataset by the underlying coalescent hanging subtrees. Briefly, the coalescence process does not have to be simulated until MRCA is found.

Observed data (based on true allelic states) and simulated data (having anonymous allelic state) can be thought as being a certain type of mathematical objects -fuzzy partitions. Classical (respectively fuzzy) partitions formalism is presented in section 4.2.2 (respectively section 4.2.3). Using this general mathematical framework allows to reuse the existing fuzzy partitions comparison methods to compare simulated and observed data in the ABC method.

Doing so, the ancient history and the ancestral allelic states do not have to be inferred anymore, while the inference can be done without summary statistics using the Fuzzy Transfer Distance presented in section 4.2.7. The implementation of this method will soon be available in Quetzal after publication.

1.5.4 Conclusion

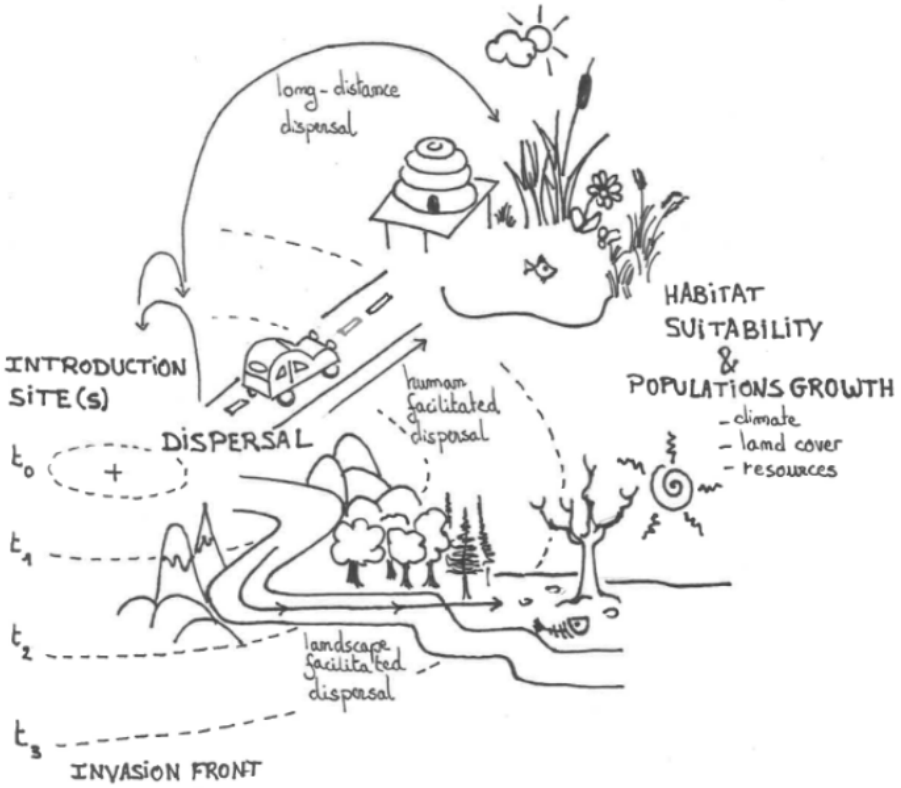
In this thesis is presented a set of methodological and computational tools for the ABC inference of the modern history of invasive population.

Fist, we presented Quetzal, a C++ library that aims at easy the development of new programs. It allows to widden the range of models that can be simulated, while ensuring high performances, and allows inference by offering an ABC module. Joint benefits of Quetzal are not met by current avalaible software solutions. Even if the code base is already quite important, important features that are useful for the community are still missing. This is not alarming: in the long run, future applications to different biological models should lead to a natural extension of the library features, and the open-ended design of the library makes it possible for users to combine the existing features with their own. If the collaborative effort follows, I expect that it should end up with an incremental gathering of fundamental features that are presently dispersed in various coalescence simulators. For example, taking into account coalescence for linked loci would allow to treat sequence data, but time was not spent in implementing non-critical features not required by the current project: this is probably one of the first features that would be implemented in the next versions, along with a summary statistics library based on available code (Arlequin). The documentation website traffics is regular, but still too modest and too sparse to draw useful insights for project development. It is generally admitted that it can take years for open-source projects to reach the critical mass of features able to attract users and collaborators.

Then, we presented various algorithms that provide flexibility in the behavioral details of coalescence algorithms. Their interest is not limited to discrete-time models, even if discrete-time models are used as a general framework all along the thesis. Continuous-time models of coalescence with simultaneous multiple collision exist, and they can be interfaced with Quetzal's policy classes.

Finally, we presented an original manner to conduct inference of very recent demographic processes.

As said earlier, their development is grounded in the study of the invasion of the Asian hornet (*Vespa velutina*) in Europe, a predator of honeybees. Using these tools to infer the dispersal and ecological features of *V. velutina* and to reconstruct its invasive history is still in development, and is not presented here. It involves possibly collaborating with a biologist specialist of *V. velutina* (Juliette Poidatz, Université de Bordeaux) and integrating species distribution models by collaborating with Alice Fournier (Université d'Orsay).



DIVERSITY OF QUESTIONS, MODELS AND DATA

UNIQUE OR MULTIPLE INTRODUCTIONS ?

Estimate the number of introduction sites ?
Estimate their coordinates ?

DISPERSAL DRIVERS ?

Geographic distance
Gaussian kernel
Leptokurtic kernels (long-distance dispersal)

Human drivers
Kernel as a function of transport networks
Uniform kernel (high connectivity)

Landscape drivers
Kernel as a function of landscape heterogeneity
Least cost path models

HABITAT SUITABILITY ?

Population growth modeling
Logistic growth model and variants
Density-dependence, stochasticity
Mortality after wintering

Integrating landscape heterogeneity
Niche population models (NPM)
Insights from species distribution models (SDM)

DATA AND GIS

Mortorway and rail networks
Bioclimatic variables
Apiaries distribution
Land cover

PREDICTION OF THE FUTURE DISTRIBUTION ?

FIGURE 1.4: The invasion of *V. velutina* raises a variety of questions. First, it is still to be thoroughly tested that the hypothesis of the unique introduction site is relevant. If not, the coordinates of the multiple introduction points should be estimated. Then, to predict the future of the invasive process, the various dispersal modes should be elucidated, and the relationship between populations growth and landscape heterogeneity should be investigated. This diversity of questions leads to an important number of possible models and data to inform them. This thesis does not directly answer the biological questions, but it rather prepares a methodological framework into which this multiplicity can be efficiently handled.

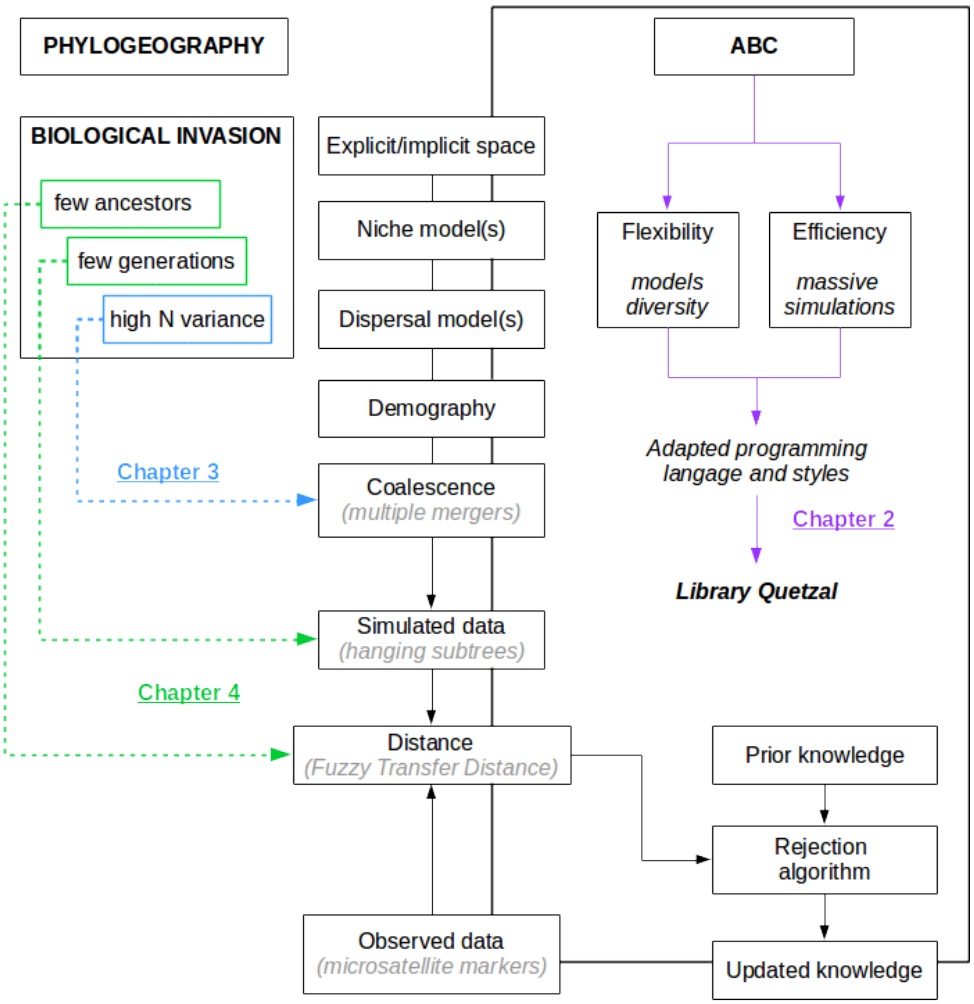


FIGURE 1.5: The theoretical framework presented along the thesis has strong connections with a number of phylogeographic approches (see e.g. He, Edwards, and Knowles, 2013b). Tackling this variety of questions, models and data has strong consequences on the simulation resources development methods, that are explained in Chapter 2. The biological invasion setup comes with interesting features: few ancestors were introduced few generations before sampling, and population sizes are (both spatially and temporally) very heterogeneous. These features have important impacts on the assumptions that can be made, (i) for the coalescence process (explored in Chapter 3), (ii) for the simulation of data for ABC, (iii) for the distance-based comparison in the ABC rejection algorithm (explored in Chapter 4).

Chapter 2

Development of Quetzal, a C++ library for coalescence

Approximate Bayesian Computation methods require that models can be simulated efficiently. The state of the art does not allow to easily develop the new coalescence-based simulators that are essential in answering questions related to biological invasions. This chapter attempts to solve this problem by presenting a C++ library that is general enough to simulate data under an open-ended number of coalesence-based models. The first part of this chapter presents guidelines through the abstraction principle, that enables the production of code with enhanced flexibility, efficiency and reusability. Library production shares similarities with the development of new words and concepts in a natural language: these similarities are exploited through the chapter to ease an intuitive understanding of the most important programming concepts. The second part presents Quetzal, a C++ library helping to rapidly develop coalescence-based simulators, possibly in explicit heterogeneous landscapes.

2.1 Reflexion about natural languages and their abstractions

> The world was so recent that many things lacked names, and in order to indicate them it was necessary to point.
>
> *One Hundred Years of Solitude*
> *Gabriel García Márquez*

> Nouns, verbs and adjectives. Lightness and rigour form the secret of this precious alloy: the language. Express as precisely as possible, without heaviness, let the beauty appear. The language is music made of liberty and unbreakable rules.
>
> *Une étoile qui danse sur le chaos, free translation*
> *Eve Ricard*

2.1.1 Words creation

Take a look at the nearest window: details and variations are a fundamental and overwhelming component of the natural world. Even two neighbouring trees of a same species will have remarkable differences in all dimensions: height, colors, number of branches or number of leaves. Let us go back to the time where words were not yet. Without words, I can distinguish these two objects by pointing to one or another. But as soon as I will move away and loose eye-contact with these objects, I will loose this ability to point, and I will be unable to communicate about them. Here comes the support of the language, as creating words make us able to refer to distant things: if I can at least emit two different sounds, I can give a name for each of these two trees, like Aba and Ada. I can instruct an interlocutor, pointing at one tree by pronouncing “Aba“, then pointing to the other pronouncing “Ada“. As we would meet again in a distant place at a distant time, at the evocation of a name our memory will invoke the appropriate object. As language is open-ended, I can easily name a third tree Abada, and a fourth Abadaba. But as we go forward in life, the number of things that have to be referred grows beyond our capacity to remember or pronounce the names designing each thing and sets of things. We reasonably can not rely on the enumeration of the sole name of each particular instance of objects to address the set formed by the reunion of these objects. In other words, how to communicate efficiently about the four trees Aba, Ada, Abada and Abadaba if we did not define the concept of a *tree species* ? And how to communicate about this set without forgetting what makes each particular tree distinguishable from the others ? Here comes the need for oblivion, for loss of details and generalization.

2.1.2 Generalization and loss of details

When observing Aba and Ada, despite all their differences, some of our intuitive mental process will discard the details that make them distinct entities, retaining what they share in common rather than what they do not share, leading us to class

them as being instances of a same set. I can invent a name designing this set: *Quercus*. I can instruct my interlocutor by pointing successive trees and repeating "*Quercus, Quercus ...*". Among their details and variations he will rapidly capture their common properties and associate their conjunction with the definition of the word I just created. Later, he shall encounter a new object and be able to determine if this instance fulfills the requirements of being a *Quercus*. Based on the decision, he will initiate some appropriate action, like harvesting the comestible fruits or ignore them. As we meet again later, the evocation of the *Quercus* will encompass the full generality of this set of common properties. Of course we can refine the definition if we discover that new constraints define new usefull subsets: *Quercus robur* and *Quercus petraea* provide long-lasting heartwood and comestible acorns, both are a (fulfill the constraints of being a) *Quercus* but are distinguishable by the form of their leaves.

Quercus does not design Aba nor Ada, it designs the idea of both of them. It is at the same time more and less than Aba. More, because it represents also Ada and any possible tree fulfilling its constraints. Less, because it is unaware of all the details that make them different and identifiable. In some strange, cryptic but unavoidable way, our brain alleviates us from the overwhelming mass of details by constructing artificial and simplified representations of objects: concepts, that we can name and share, refine and extend. This process has been termed *abstraction*.

Definition 2.1.1. Abstraction The process of formulating generalized ideas or concepts by extracting common qualities from specific examples.

Amusingly, abstraction is so intuitive and powerful that it requires tremendous efforts to formalize the species concept definition, even though we could distinguish an oak from a poplar from our earliest years.

2.1.3 Complexity reduction

Abstraction is also a natural way to decrease the complexity of a problem to a reasonable level. If we try to compare men and women salaries, the observed distributions of salaries would certainly be abstracted to (approximated by) some distribution law (*e.g.* the normal distribution). As the details of the reality of the sample will be abstracted away, its dimension will dramatically drop to the comfortable number of parameters of the law (two in the case of the normal distribution: mean and variance). Instead of being overwhelmed by the smallest details of the sample, our brain

can now focus on its principal variations and proceed to extra work, for example mean comparison. The quantity of information lost is hopefully compensated by the information gained by being able to manipulate this approximation in a rigorous, mathematically formalized statistical framework.

2.1.4 Incrementality of abstraction: defining new abstractions with old abstractions

Words creation coupled to generalization is a fundamental aspect of intellectual work and communication. Science communication makes full use of abstraction as new concepts are created daily from pre-existing concepts. The word *ecosystem* was created and loosely defined in 1935 by the botanist Arthur George Tansley, and its definition has been progressively refined to its present accepted formulation:

Definition 2.1.2. Ecosystem All the living things in an area and the way they affect each other and the environment (Cambridge online dictionary).

First we can appreciate again the benefit of defining new words by imagining how it would be difficult to exchange about ecology if the word *ecosystem* had not been created and that we had to repeatedly enounce its full definition instead. Secondly we can appreciate the full generality of the definition, as it is defined in terms of secondary abstractions: *living things*, *affect*, *environment*. We will see that a main purpose of programmers is to imitate this linguistic process that creates new powerfull abstractions.

2.2 Programming languages

2.2.1 "An absolute gulf between intelligence and bullshit."

> Fondamentalement, l'ordinateur et l'homme sont les deux opposés les plus intégraux qui existent. L'homme est lent, peu rigoureux et très intuitif. L'ordinateur est super rapide, très rigoureux et complètement con. On essaie de faire des programmes qui font une mitigation entre les deux. Le but est louable. Mais de là à y arriver ...
>
> *Gerard Berry*

Solving problems with a computer requires a communication system between the human and the computer; that is a programming language. As a human programmer, then my first interlocutor is a computer, and my first task is to write a set of instructions that are understandable by both humans and computers. This is complicated and discussion is incredibly uneasy. As often with communication, the incomprehension is largely due to cultural differences. In our human culture, all communications are based on small memory, great intuition. No need to mention our ability to deal with homonyms and ambiguities, that leads to word games, humour and poetry. It is amazing that humans are able to communicate in languages that allow for so much uncertainty and ambiguities. We expect that our interlocutor will be able to detect small errors in our speech, correct them, and clarify ambiguities by choosing the most appropriate meaning or by asking. Above all, we expect that the discussion will not be overwhelmed by quantities of useless details.

Any person who once wrote some code would surely have a mirthless laughter by reading these lines. Indeed, a computer will likely never be this ideal companion: computers are way too dumb. Even if computers are incredibly fast and handle tons of details, they perform poorly at generalizing, guessing, infering meaning and intentions.

An interesting human metaphor was given by Jorge Luis Borges in *Funes the Memorious* (1942), a tale where Ireneo Funes acquires after a head injury the ability to remember everything. His thinking leads the narrator, a version of Borges himself, to state:

> I suspect, nevertheless, that he was not very capable of thought. To think is to forget a difference, to generalize, to abstract. In the overly replete world of Funes there were nothing but details, almost contiguous details.

For humans, abstraction of details is *everything*. For computers, the *details* are everything. Humans and computers are so different that communication between them (that is, programming) is uneasy, requires years of training, decades of practice and that generations of researchers and engineers spent their time in trying to ease it. We will see that solutions have been found, as modern programming languages propose features that allow human developers to build powerful abstractions that

can be then used to write code that hopefully is more than *"almost contiguous details"* and that can be understood both by humans and machines.

2.2.2 Unbreakable rules: Vocabulary, Grammar and Semantics

A programming language is formed by:

- an alphabet (*e.g.* ASCII or Unicode)
- a formal grammar (defining the syntactic rules between elements of the language)
- a semantics (that gives the meaning of a sentence in a language).

Remark. As in natural languages, semantics and grammar can be thought independently: the well-known sentence "Colorless green ideas sleep furiously" (see Chomsky, 1957, p.15) is grammatically correct but semantically nonsensical.

A programming language provides a basic vocabulary, that is a set of primary abstractions hiding the details of a computer functioning. For example, manipulating numbers is expected by humans to be easy and intuitive, and one expects to perform additions or divisions by writing without caring about number how numbers are represented in the computer, or about the algorithms defining how mathematical operations are performed. This basic vocabulary is essential, because number representation and computer arithmetic constitute a distinct discipline requiring years of study (see *e.g* Parhami, 1999).

2.2.3 Liberties: if the word doesn't exist, invent it

Writing a non-trivial program only in terms of the basic vocabulary of a language would be as cumbersome and inefficient as writing a thesis only with a 5 years old child's vocabulary: it is possible, but it would require a huge amount of large and complex sentences carrying little meaning and expressiveness for the human interlocutor. A funny counterpart in natural languages is given by the *Definitional literature*. It is a poetic form and technique coming from OuLiPo (a group of mathematicians, authors and painters aiming at inventing a new literature) that starts from a statement and rewrites it by recursively replacing each word by its definition. The statement:

> The study and management of ecosystems represents the most dynamic field of contemporary ecology (Publishing, 2018)

is transformed into:

> The application of the mind to the acquisition of knowledge, as by reading, investigation, or the act or manner to take charge of a system, or a group of interconnected elements, formed by the interaction of a community of organisms with their environment, designate a very active area of activity or interest of relationships between the air, land, water, animals, plants, etc.

It is naturally hardly understandable due to the many details obfuscating the primal meaning.

Just as prehistoric human proto-languages did not know (nor need) the concept of *ecosystem* or the intermediary subconcepts *organisms* or *environment*, programming languages are fairly young and natively do not know anything about them. But just as a strength of human languages is to conceal the potential for creating an open-ended number of words, a strength of programming languages is to give the ability to define new abstractions in terms of other abstractions so that they can be used to express ideas in a synthetic and efficient way. We will see in the further sections the various tools that are given to build them. Gathering a set of abstractions suitable to a given field (*e.g the coalescence*) is exactly the role of libraries.

2.2.4 Libraries: Invent it; But first be sure it doesn't exist

Motivations

A library can be seen as a semantic field that is required to conveniently address in the programming language the family of problems arising from the discipline. The abstractions contained by the library allow to represent directly into the code concepts and behaviors that are familiar and well-defined into the field, rather than manipulating inappropriate vocabulary coming from lower abstractions. Importantly, these abstractions are designed to be highly reusable (see Figure 2.1).

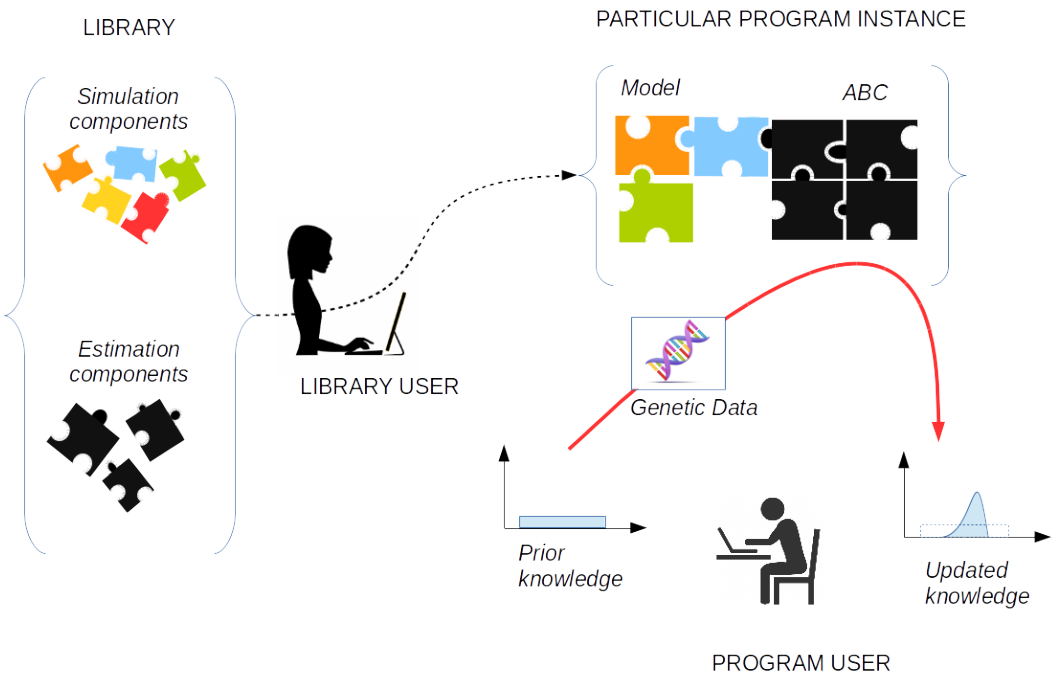


FIGURE 2.1: Library seen as a collection of components (abstractions) that have a defined behavior (they have a special purpose) and that can be combined if they present compatible interfaces. The library user writes a program by picking the desired behaviors in the library, so a program can be written. The program itself has a defined behavior that depends on the input given by its user. In this example, it conduces some inference using genetic data as input. The program user and the library user may not be the same person.

Definition 2.2.1. (Library) A collection of implementations of behavior, written in terms of a language, that has a well-defined interface by which the behavior is invoked (Wikipedia, 2018). Figure 2.1 gives an intuition of the difference between a program and a library.

When a library adressing a problem does not exist, it is generally worth to design it, because in the long way much time will be spared. The main goal and difficulty of designing the library will be to construct abstractions implementing small, atomics behaviors that are easy to understand and easy to reuse.

Benefits of libraries: correctness, efficiency, maintainability

Using libraries has a well-known number of benefits that were summarized in Stroustrup (2003):

> The key to fast development, correctness, efficiency, and maintainability is to use a suitable level of abstraction supported by good libraries.

Fast development comes from the fact that since the administrative tasks (organization of the program into small pieces and memory management) are services offered by the library, the programmer is free too focus on describing its own computational problem in terms of the library concepts.

Correctness comes from the fact that as the library code is regularly reused, its correctness is more likely to be tested, and if an undesirable behaviors is raised, it can be reported and corrected by the library maintainers.

Maintainability stems from the fact that by definition, a library decomposes complex tasks into much smaller and simpler behaviors, allowing to easily perform changes (to fix a bug or add a feature, to enhance performance or usability) without introducing new bugs in the system. Since complexity has been breaked down, the system is understandable by new developers that can make a progressive exploration of the collection.

Efficiency is due to the fact that since a library code is likely to be reused many times, an important quantity of time can be invested in the optimization of its most critical components. Shared knowledge coming from users feedbacks and collaborators suggestions participate to progressively increase the efficiency of the most important library components, at no cost for the user.

What is a "suitable level of abstraction" ?

> Call him Voldemort, Harry. Always use the proper name for things. Fear of a name increases fear of the thing itself.
>
> ---
>
> *Harry Potter and the Sorcerer's Stone*
>
> *J.K. Rowling*

Being able to use a suitable vocabulary with the right level of abstraction to expose a problem is obviously a quality for natural languages speakers. For example in giving a talk, a young phD student in landscape genetics will naturally manipulate *landscapes* because it is a convenient abstraction in the natural language. The *landscape* word is charged with meanings, but it does not specify which aspects (topography, soil occupancy ...) are considered, and it does not precise the spatial extent of the landscape. It does not even precise the geometry of the geographical space. Far from being a problem, this lack of precision helps the audience to focus on more advanced points.

Reciprocally, using an inappropriate vocabulary or a wrong level of abstraction to expose a problem is obviously disruptive and inefficient. Most of communication problems between scientists of different fields come from the fact that the levels of abstractions have to be constantly adjusted, and most of the efforts in interdisciplinary projects consist in acquiring the abstraction level of the others fields (or at least informing a common and comfortable subset), so ideas can be efficiently appropriated and passed around. As each field has built very high abstractions, communicating with the newcomer is commonly associated with a lowering of the abstraction level that sometimes does not reflect anymore the true nature of things. This does not necessarily threaten the quality of short interactions, but it can prevent any attempt to treat more advanced problems.

Similarly in programming, very short programs do not necessarily suffer from using low-level of abstraction, but long ones will. If the problem has been *exhaustively* specified, that there is *absolutely no risk* that requirements change, and that *very few* lines of code are necessary to describe the problem in terms of the language most basic feature, obviously the problem *is* a low-level problem, so using higher level abstractions will not help. But if the problem is not that obvious, then programmers can greatly benefit from finding the right level of abstraction. When they fail to identify it, problems arise.

For example, if the previous student has to write a program for solving non-trivial landscapes problems and that he finds no library offering a *landscape* abstraction, the risk exists that, ignorant or reluctant about the tools allowing to represent such a degree of abstraction in the code, he will be tempted to over-specify details and to use built-in and low-level abstractions, or inappropriate abstractions coming from another field. He will likely use matrices from some mathematical library to represent lattice landscapes, and row index and column index to indicate coordinates. Instead of writing code using *landscapes*, he will write code using *matrices*. At the beginning of the project, it will seem *simple enough*. But very soon, all his code will be pervaded with matrices of values. As matrices are charged with a strong matrix semantics, manipulating them is going to be of an absurd complexity. As matrices ignore the concept of time, he will be tempted to use a list of matrices to represent a time changing landscape. As a landscape can have more than one dimension (altitude, temperature ...), a list of list of matrices is going to replace the previous definition. And paradoxically, there is going to be no other choice than

using this utterly complex and counter-intuitive aggregate to represent landscapes of one dimension that is constant in time and space (where one value should have been enough). This is paradoxical, because using matrices seemed initially a simple and sound option.

Indeed, problems come from the semantic gap between a *landscape* and a *matrix*. We expect from a lattice landscape to have some notion of width or resolution (for example in kilometers), but a matrix semantics gives no indication about this crucial information. We expect to easily compute geographical distances across two random points in a landscape, but a matrix does not propose any related feature. Obviously this is not a lack in the matrix definition: they are just meant to be used as matrices, not landscapes. The problem is that we are using the *wrong level of abstraction*: even if in some cases a matrix can be an acceptable representation for a physical value sampled in a two-dimensional lattice landscape, a matrix is definitely not a landscape. They do not present the same behaviors and multiplying a landscape by a vector makes no sense. This difference should be stated in the code as clearly as the fact that a whale is not a fish even if they both swim.

Making this distinction explicit is essential, because we are intuitive beings and when reading a text we tend to perform projections on the potential behavior and qualities of things. Obviously these projections will be profoundly different is the *thing* is a *matrix* or a *landscape*: using the wrong term for an object is almost always confusing and deceitful. "*If names be not correct, language is not in accordance with the truth of things*" said Confucius. Experience tends to show how much he was true about programming languages too.

In conclusion, when willing to represent a concept in a code, one should first search for the suitable abstraction in existing libraries. If not found, it should not be feared to build this abstraction using the programming language features (they exist for this very purpose) and to give it a proper name.

2.3 Constructing abstractions with the C++ language

2.3.1 Why C++ ?

At some point, one has to decide which programming language will be used to address the computational problem. There is a wide range of available languages and choosing among them is not straightforward: comparing languages is a very tricky

issue. In most cases, the choice of a language is related to situational (rather than technical) aspects like the possibility to run the software on the material, licenses, developers experience or the community culture.

The C++ language is a royalty-free portable language, that has been widely used to implement coalescence simulation programs. This popularity can be explain by the fact that C++ design is particularly relevant for applications requiring high performances (Stroustrup, 1994). It allows both for low-level manipulation of data representations (pointers, manual memory management) and for building of higher-level structures using the language features such as classes, class hierarchies, templates, exceptions, and namespaces. These features allow to build and use efficient libraries. As C++ has static typing (that is, types are known and checked during compilation, and not at runtime), it allows the compilers to optimize the code for better runtime efficiency. C++ supports various styles of programming (among them procedural programming, object-oriented programming, generic programming) that allow to choose the appropriate style to solve a problem in the most elegant way (that is, the simplest and the most efficient one).

2.3.2 A step-by-step abstraction design

Primitive built-in types and good variable names

At some point in writing the program, it will be necessary to store some intermediary result in a *variable*, that is to reserve space in memory: variables are memory locations where values can be stored. The *type* of the variable determines how much memory the operating system should reserve, the *variable name* allows to refer to it.

The C++ language offers several built-in types: boolean types, character types, integers types and floating point types. As said earlier, these types act as primary abstractions hiding the details of the data representation : for example, one can declare two floating point values to represent the longitude and latitude of a sample point without having to feel concerned about how floating point values are represented in memory. Furthermore, the *variable name* is a very basic liberty for the programmer to elevate a bit the level of abstraction of its code:

```
double latitude = 45.5;
double longitude = 1.2;
```

Type aliasing for expressive types

The *type aliasing* is a feature allowing to add a bit of abstraction that will help the understanding of the code in an explicit and secured way. For example, imagine we want to precise that the coordinates are expressed in decimal degrees. We could of course add a line of comments:

```
// double represents decimal degrees
double latitude = 45.5;
double longitude = 1.2;
```

The use of comments (*when*, *where* and *how* to write them) is still a very active debate. In my opinion, the main problem is that comments are by definition not checked by the compiler, they are only written, read, and checked by humans, and that humans are fallible and very hurried beings. Because of this, after having modified a code snippet, the programmer forgets often (intentionally or not) to modify the comment explaining the code. Consequently, the histories of comments and real code tend to diverge, and that often leads to comments that are in contradiction with the implemented behaviors. This occurs a lot, and consequently a recurring motto is to write code that is self-explanatory: no one should rely on comments to express key concepts if the language features allow to express the same concept in the very language, in an explicit, concise and secured way. Indeed, type aliasing allows to add some information to a type, in a way that is compiler-proof:

```
using decimal_degree = double;
decimal_degree latitude = 45.5;
decimal_degree longitude = 1.2;
```

Standard Library for more advanced abstractions

The Standard Library provides key components that are pivotal in the code production: algorithms, containers, functions and iterators. Containers are used to store values of any built-in types or any user-defined type fulfilling some basic constraints. For reducing the library complexity, algorithms and functions are designed to be independent of containers on which they operate. This is accomplished by using iterators, an abstraction that allows to represent any type of container.

The Standard Library is widely used for building the very first levels of abstraction in a project. For example, any function using geographic coordinates would

have to take as an argument the longitude and the latitude. This can be cumbersome if several coordinates are needed:

```
void foo(decimal_degree lon1, decimal_degree lat1
         decimal_degree lon2, decimal_degree lat2);
```

Code complexity can be reduced by associating latitude and longitude in a same structure, like a pair of values: the Standard Library offers the `std::pair` container to represent it. Arbitrarily, the first element (accessed by `x.first` will be set as the longitude, and the second element (accessed by `x.second`) as the latitude. Each (longitude, latitude) couple is then represented by a `coord_type` object (that type is an alias on a `std::pair<double>`). This reduce complexity and function calls begin to look better:

```
using decimal_degree = double;
using coord_type = std::pair<decimal_degree>;
void foo(coord_type x, coord_type y);
```

However, depending on the context, it may be that the application needs to evolve (what happens often in research), and that the altitude should finally be part of the coordinate definition. We can anticipate this evolution by not constraining a coordinate to be a couple of value. Like many problems in software development, this can be done in more than one way. Using a `std::vector<decimal_degree>` would certainly be the first reflex of the beginner willing to represent a set of values (*i.e.* abstracting away the number of elements).

In the code lines above, only the type aliasing definition has to be changed:

```
using coord_type = std::vector<decimal_degree>;
```

Because of type aliasing, the signature of `foo` does not need to change, as it was not defined in terms of `std::pair` but in terms of an abstraction (the `coord_type` type alias). We see here a first advantage of using abstractions: that leads to code that is more robust to details changes.

A "wrong level of abstraction" feeling

However, this solution is still unsatisfying, as it is not clear if longitude or latitude is referred first. The standard EPSG:4326 recommends to use the latitude, longitude ordering. But an appreciable number of software use longitude, latitude ordering, so

there is much confusion. Apparently trivial, this problem caused so serious damages in the geospatial community that GeoTools, a GeoSpatial library, found it worth to document the history of the problem (GeoTools, 2018).

The problem comes from the ordering of information elements (latitude, longitude) that are not meant to be ordered. Obviously using an abstraction based on an ordering is the wrong level, as it offers an information that is misleading. Just like the `std::pair` implementation did the arbitrary contract that the first element would be the longitude and the second element would be the latitude, so does the `std::vector` implementation, adding that the third element (if it exists, what still needs to be checked) would be the altitude. As these contracts are implicit (that is not specified as compiled code), it becomes the task of the developer to remember and to respect all these points, loosing time and energy by constantly asking the code he is writing (or reading):

```
using coord_type = std::vector<decimal_degree>;
coord_type x = {45.5, 1.2, 10.0}

// much further in code ...
x[0] = 9855625; // does it access to longitude or latitude ?
                // is the new value relevant ?
x[2] = -10;     // what was the elevation unit again ?
                // decimal degree ? Really ?
                // was the third element defined ?
                // if not, this is a bug.
```

Moreover, there is no tenable reason that could ever justify why the altitude shares the same value type with longitude and latitude: altitude unit should be precised independently. This point could be naively fixed using a `std::tuple` (that is a container of heterogeneous types) to define an alias for `coord_type`, but it would not improve the readability and the expressiveness of the code using coordinates, nor would it improve the security checks:

- is longitude in its validity range, that is $[-180, 180]$?
- is latitude in its validity range, that is $[-90, 90]$?
- is elevation relevant relatively to Earth diameter ?

These questions must hold true during execution of the program: they are called *invariants*. To enforce these requirements *by design* in a clear and explicit way, the best solution is to define a new type, a class, where it is guaranteed that invariants, expressed in the code using *assertions*, are always respected.

Classes for enforcing invariants

At some point, one wants to represent in the code an object of the real-world that has some internal state, some internal functioning and some actions that are possible to perform on it. As the previous example of geographic coordinates is rather abstract, we will first consider a more trivial object, say a lamp that we got at our favorite shop. The lamp switch can be *on* or *off*, the lamp can be connected or disconnected, and the bulb can be functional or burnt-out. If the lamp is disconnected or if the bulb is burnt-out, the lamp can not be alight, even if the switch is on. Of course at all times the lamp is either alight or not. To model the state of the lamp and the actions one can perform on the lamp, using only basic types will not be satisfying: indeed several dependent variables are involved (alight state, switch position, connection state...), and modifying one variable has repercussions on the others (connecting the lamp lets the switch position untouched but will alight the lamp if the switch was on). If not hidden and carefully managed, these dependencies among variables will seriously hinder code reliability and readability. Rather than cryptic variables manipulations, one would likely prefer to write (or read) something like this:

```
Lamp lamp = our_favorite_shop::buy_lamp();
lamp.connect();
lamp.turn_on();
if( !lamp.is_alight() & !lamp.is_bulb_burnt()){
  std::cout << "defective␣item" << std::endl;
}
```

The main task of the programmer will be to define the `Lamp` class in such a way that these actions can be performed and that the state of the lamp is *always valid*, from its construction to its destruction. Any action performed on the lamp should never put the lamp in an invalid state (for example an lamp alight with its switch off). Consequently, one should avoid to enable the lamp user to turn off the switch

without feeding back this action on the lamp state. This can only be done by forbidding the direct access to the state, and imposing the user to access the state by calling carefully designed functions. This is the role of the private and public fields of a class:

```
class Lamp{
private:
  // Implementation details, generally state
  bool is_switch_off;
  bool is_alight;
  bool is_connected;
  // ... more details

public:
  // Public interface, generally behaviors
  bool is_alight() const;
  void connect();
  void turn_on();
}
```

The private field contains the member variables (and member functions) that describe the internal details of the class. They can not be directly accessed from outside the class: only the members described in the public field are accessible. The implementation of the `turn_on` function will ensure that the `is_alight` member is set to true if and only if the `is_switch_off` member is evaluated to false and the `is_connected` member is evaluated to true. This secures the access and modification of the information by the user, as he would not be able to directly set the `is_switch_off` member. Instead he will will have to pass by a secured public member function that automatically avoids dysfunctions.

Reconsidering the coordinate data type previously presented, creating a class that is dedicated to its representation allows to design quite easily a set of services that is both very expressive and difficult to corrupt (Figure 2.2). An example is the following very minimalist implementation of a GeographicCoordinates:

```
class Coordinates{
public:
```

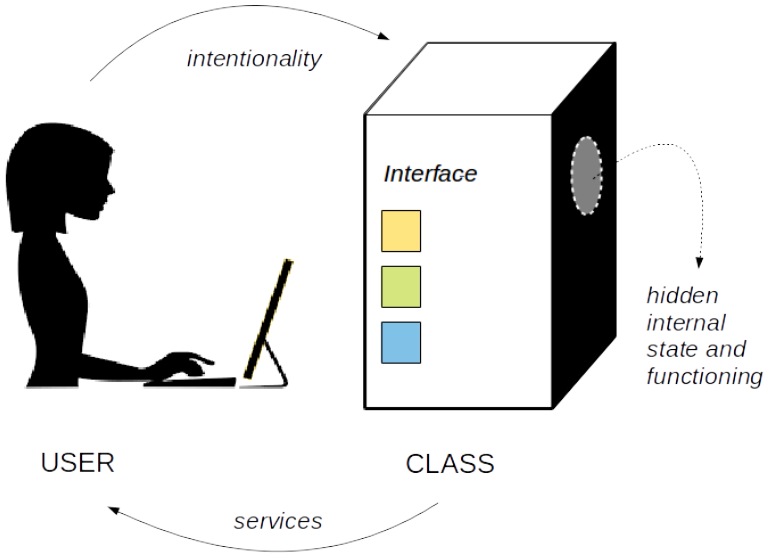


FIGURE 2.2: Class services: a set of complex behaviors is hidden behind a class that offers a clear and intuitive interface that the user can easily predict, understand and use. A metaphor can be given by a coffee machine, where complex details (pipes, electronics, pumps, water alimentation...) are hidden behind few intuitive buttons and an electric plug.

```
  // type aliasing
  using decimal_degree = double;
  using km = double;

  Coordinates(decimal_degree lat, decimal_degree lon);
  decimal_degree lat() const;
  decimal_degree lon() const;
  km great_circle_distance_to(Coordinates other) const;
  bool operator==(const Coordinates& other) const;

private:
  // details the user do not care about
}
```

When creating a coordinate object, the constructor will be called, and invariants will be tested here. As the class has only `const` methods (that is methods that do not modify the state of the instance), the invariants are ensured to hold true. The class gives a read-only access to longitude and latitude values through public methods `lat()` and `lon()`, so the client code depending on these methods is ensured to be

self-explanatory : `coord.lat()` or `coord.lon()`. The only place where confusion is possible is pushed back to the constructor, where the place of arguments could be exchanged, even if it follows the recommendations of EPSG:4326. This design choice can holds if the coordinates objects can not be constructed by the library user, but only accessed for read-only purposes (this kind of scenario happens when the construction is actually hidden by another producer class, a geospatial dataset reader for example).

Information hiding

The principle to distinguish implementation details (the private field) and the public interface (the public field) is called *information hiding*, that is often assimilated to the term of *encapsulation*. *Information hiding* is the way computer science performs abstraction and it is quite different from the way mathematical science does. Mathematics rely on *information neglect* to build complex inference systems whose generality comes from the elimination of irrelevant information, whereas computer science manages the complexity of systems by hiding (not neglecting) information details, that are crucial for the processing, but irrelevant in the current programming context (Colburn and Shute, 2007).

Indeed, they are irrelevant because the details of the data representation change often, but the general behaviors that we expect from the class do not change that easily: although we can not guarantee that the state of a lamp will always be represented by three booleans, we can *always* expect from a lamp that it is alight or not, connected or not: they are *services we can expect from the class*. This design approach that emphasizes behaviors rather than data allows to design more robust systems, where objects state are manipulated via carefully designed interfaces. It also enables flexible programming, as various kinds of objects (that is, classes) differing only by details can be abstracted to the subset of their common behaviors. For example, both lamps and televisions have in common that both can be connected to the electrical network, then it is possible to model a third software component that buys and installs an electrical device in an house and that applies to both lamps and television.

2.4 Writing code that is resistant to changes

2.4.1 Motivations

Changes are everywhere. Perhaps even more in science research where the very dynamic activity is to constantly try new ways to solve new problems. As a landscape geneticist, you are working on a small butterfly: you design the model, write some code, process the data, begin to have some preliminary results. You present them to colleagues, and based on their suggestions you change some details in the hypothesis of the model. Thanks to a collaboration, you enrich your dataset. A statistician remarks that some details in the statistical framework could finally benefit from some rearrangement. You think you can reuse the code you wrote for the first paper: why not ? after all you just performed *minor changes in some details*. Very soon you understand that these changes have such repercussions on the implementation that your code is beyond salvation. Depending on the project, hours, days, months, or years of hard work can be lost (in my case I lost eleven months of code base, on a three years project). *How did this happen ?* The reason is simple: for computers, details are absolutely *everything*. The number of parameters of your model, their type (integers, decimals), their name, the dimensionnality of your data, the type of the variables in your data: you did not realize that you wrote the code in terms of the most ultimate details of your problem. Facing the change, you barely begin to appreciate everything that made the specificity of the problems you were willing to solve. Do not panic: you are just becoming a developer. Generations of programmers have faced the same problem: textbooks, blogs, articles were written, conferences, seminaries were organized, even languages evolved to address the same issue: writing code that resists to changes. Problems have been identified and solutions exist.

2.4.2 The problems comes from code dependencies.

Willing to represent geospatial coordinates, you wrote the entire code in terms of the `Coordinates` class, because it seemed a very solid abstraction. In any function were geographic coordinates were involved, you used the `Coordinates` class in the signature declaration, and the function definition. In any class composed of geographic coordinates, you used the `Coordinates` class in the member declaration. Doing so, a tight *dependencies* have been introduced between the class and its *client* (the code using it). It can seem reasonable, as obviously at some point geographic functions have

to interact with geographic coordinates. But it must be acknowledged that this class is very basic and that sooner or later it will reach its limits, for example because it is for now totally unaware of the altitude, or of the geodetic datum modelling the form of the Earth. Anyone thinking it is a detail that will never change should have a look to the complexity and the diversity of the existing coordinates systems to get convinced that it can not reasonably be guaranteed that a more advanced representation of a geographic coordinate will ever be needed. Moreover you will certainly want to test the simulation code with simplistic models were the geographic space can be just two points, or some regular points along a line: in that case using geographic coordinates will bring many difficulties, as it is the wrong level of abstraction.

Inevitably at some point the meaning of a geographic coordinate will slightly change: if the design of the application is robust, a single line will need to be changed. If the design is rotten, the application will break at every dependency point between the system and `GeographicCoordinate`. Thus, a dependency should never be established if it can not not be guaranteed it will never change. Dependencies propagate very rapidly in the code and have a number of toxic side effects. They are rigidity, fragility, immobility, and viscosity, defined in **martin_design** as symptoms of a poor design.

2.4.3 Symptoms of a poor design

Rigidity

It is defined by the difficulty to make even slight modifications to a software. When dependencies have pervaded a system, a modification at a given place triggers further modifications in the dependent parts of the software. The snowball effect can be very serious, transforming what should be a minor change into a major reconsideration of various modules. As the dependencies chains grow rapidly beyond possible appreciation, the time costs of a modification can no longer be estimated, and deadlines becomes untenable. Because of fear to break the system, no one dares to bring modifications. At some point, rigidity can become so strong that non-critical modification to the software is tacitly or explicitly forbidden.

Fragility

This the propensity of a system to break at multiple points when a single modification is done. The unpredictability of the break points location (that can be in very remote parts of the system) makes it very risky to even fix one single bug, as it raises multiple other bugs. When fragility is high, the system is not maintainable, as fixing bugs causes even more bugs.

Immobility

Immobility is characterized by the fact that the parts of a system can not be reused in another context (another project, or another module of the same project). Immobility arises when dependencies to the context are too numerous in the system, so a part can not be easily isolated from its context and it becomes preferable to rewrite the entire functionality rather than reusing it.

Viscosity

It designate the tendency of a system to favour changes that have a negative impact of the design. In a design with high viscosity, solving a problem in a quick and dirty way is easier than using a clear and sound solution.

As dependencies is the root of all these symptoms, various techniques have been proposed to manage them, aiming at designing variation points in the application to contain dependencies.

2.5 Design principles: managing dependencies with S.O.L.I.D.

2.5.1 Motivations

SOLID is largely considered as one of the most important acronyms in object-oriented design. It designates five principles initially identified by Robert Martin in the late 1990s as cornerstones of sound class-level design (Martin, 2002a). Together, they allow developers to identify more clearly, to reason and to communicate about the feelings the have of a *clean* or a *bad* code. When an application is perceived to suffer from bad design symptoms (rigidity, fragility, immobility, viscosity), it generally turns out that several of these principles (or even all of them) are violated. Refactoring the application to respect these principles generally leads to better feelings

about the code. There is no formal proof that these principles actually work, but accumulated experience of the developers community shows that, in some way, better design and less problems result from writing code that respect SOLID principles. Consequently, they are not meant to be absolute rules to follow at any cost, but are rather meant as good advices and practical heuristics:

> They are common-sense solutions to common problems. They are common-sense disciplines that can help you stay out of trouble. (Martin, 2009)

2.5.2 S - Single Responsibility Principle (SRP)

Definition 2.5.1. Single Responsibility Principle A class should have one, and only one, reason to change. (Martin, 2002a, p.95)

In other terms, a class should be related to unique responsibility, defined as "a reason to change". Because it requires experience, it can be difficult, when designing a class, to anticipate what could later change. However there are some aspects that are known to be highly unstable like the format of input or output, or the user interface. For this reason, the output formatting of an object `Coordinates` is not define in the class, because printing output format depends on a context unknown when designing the class. There are actually many reasons one would like to print a coordinate object:

- to constitute a bug report.
- to use as a *spy*, that is a momentary output one use as a debug tool to print out the state of variables.
- to use as part of a more complex structure, for data exchange purpose.

It is not clear which part of the information is relevant for printing (should the coordinates unit, *i.e.* decimal degrees, be part of the information printed ?), it is not clear how to organize the relevant information (we saw that the longitude, latitude ordering was an important problem), and it is not clear if an approximated information is relevant or not (that is how to decide the the floating point precision of a latitude ?). Obviously, to answer these questions, one should know the context in which a coordinate needs to be printed. And as contexts tend to change extremely fast in a

software life cycle, there will be a constant back-and-forth in the `Coordinates` class to modify the way a coordinate information is displayed, even if the others class services work fine. Each new printing context will be a new reason for the class to change.

Obviously, printing is not a responsibility of a coordinate: the unique responsibility one should reasonably expect from a geographic coordinate is to locate a point in a space. Printing should be the responsibility of another class, like a `Printer` class, that has access to the relevant information through the `Coordinates` interface, and organizes it in a manner specified by the context before to display it.

SRP is the developer first tool for managing the complexity of a system, as it helps to compartmentalize the services of a system into manageable smaller pieces: responsibilities, that have each as little reason to change as possible (that is, ideally, only one). SRP is perhaps the most important principle of SOLID, because it enables the others: it will be difficult to respect the four other principles if SRP is violated.

2.5.3 O - Open/Closed Principle (OCP)

Write a code once for all

Adding new features is a key step in the life cycle of any software. For example a biologist studying migration patterns is likely to change the migration model from time to time according to its current biological model. However, changing the requirements of an application generally causes many modifications in the source code. This is very unsatisfying, as the pre-existing code has normally been tested and its behaviors proved to be correct, so there should be nos reason to compromise all these precious and solid results. Source code modification only bears uncertainty and potential bugs, so it should be avoided (Martin, 1996d):

Definition 2.5.2. Open/Closed Principle Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification (Meyer, 1988).

That is, when requirements change (for example when a new migration model is needed), the new feature should be implemented by adding new code, not by modifying old code. It can bee seen as contradictory: how to add a feature without modifying the code ? Various language features make it possible to achieve flexibility by adding *extension points*. As these extension points can complicate the design, an useless extension points is source of problems and should be avoided. The decision

to place an extension point should then be based on the probability that the need for extension will appear in the future. The knowledge of the study domain surely helps a lot in taking this decision: migration models, mutational models, growth models, coordinate systems are known to be unstable across studies, so in these cases it is generally worth to place an extension point rather than a dependency. This is done by using more abstractions.

A random walk case

Imagine we want to represent a random walk in a landscape. A key step is to sample the next location conditionally to the present location. A naive approach to tackle multiple migration models would be to use case statements:

```
Coordinates walk(Coordinate x, std::string model){
  Coordinate y;
  switch ( model )
  {
    case 'gaussian':
      // sample y in gaussian density conditionnally to x
      break;
    case 'uniform':
      // sample y in uniform density conditionnally to x
      break;
    default:
      // error : unknown model
  }
  return y;
}
}
```

This code does not respect the Open Close Principle. If a new migration model is needed, the code using migration models will need to be updated, a new case will need to be added in the `switch` statement. Not only does it compromise old working code, but also it adds much complexity to the code, as it leads to monolithic switch (or if/else) statements. Any user of the random walk simulation function would

have to find, read, understand and update the code if he wants to add a new model. This is untenable. There is much better to do.

Manipulating different forms of a same idea: polymorphism

Generally speaking, statements with many different cases is a potential symptom of a bad design. The first thing to do is to think in terms of abstractions, in what is common across all anticipated cases, and to use this common point to design the algorithm once for all. In other words, we want to make the computer understand that all possible migration models, are, in some way, just various forms of a same general concept, and we want to write the algorithm in terms of this general concept, not in terms of all possible forms. What we want to achieve is called *polymorphism*. It can be done in two orthogonal ways: using generic programming (with templates) or object-oriented programming (that is, subtyping). Generic programming allows the compiler to have all the type information at compile time. As there is no extra operation to perform at runtime, this technique is known to be very efficient. In another hand Subtyping makes use of class hierarchies to achieve polymorphism at runtime, what can be very useful if the type of an object changes according runtime conditions, but it is known to hinder performances unnecessarily if generic programming can be used instead (Stroustrup, 2014).

Static polymorphism with generic programming

All we want to do is to sample a new location in a model. We actually want to write something like:

```
Location walk(Location x, Model model){
  return model.sample(x)
}
```

This is much simpler than the previous version. However, this code does not work: the `Location` and `Model` classes have not be defined. The generic programming, enabled by the `template` keyword, allows you to write code in terms of types (or equivalently, classes) that are unknown at the moment to write the code but that will be known at the moment where the code should be compiled. This is very handy to reduce dependencies, and very simple to write:

```
template<class Location, class Model>
```

```
Location walk(Location x, Model model){
  return model.sample(x);
}
```

This code will work with any class that defines a function in its interface taking a `Location` that has the following signature: `Location sample(Location)` . There is much generality in this definition. Further generality can still gained by replacing the `sample` function with the call operator:

```
template<class Location, class Model>
Location walk(Location x, Model model){
  return model(x);
}
```

An open-ended number of possible migration functions and coordinate systems can then be considered, at the (little) price to use the `template` syntax. It enables small unit tests, as you can readily use and test the `walk` function with simplistic models (a two-state markov chain) and basic types (booleans representing coordinates):

```
// type aliasing
coord_type = bool;
// lambda expression are small anonymous functions
auto move_away = [](coord_type from){return !from;};

coord_type x = true;
for(int t = 1; t < 10; ++t){
  x = walk(x, move_away);
  std::cout << x << "␣";
}
std::endl;
```

Indeed the `walk` algorithm can work with any types fulfilling basic constraints:

- `Model` and `Location` must be *copiable* types
- `Model` must be *callable*

This set of constraints enabling the walk algorithm to work is called a *concept*, that is defined (implicitly in the code, explicitly in the documentation) in terms of the

`Callable` and `Copiable` subconcepts. As many other basic concepts, they are defined in the C++17 standard (ISO/IEC 14882) (see *e.g.* cppreference.com, 2018). As explicit is better than implicit, the C++ normalization committee is working hard on a way to express these concepts directly in the code, rather than in the documentation. This should be done with the C++20 norm coming soon.

Runtime polymorphism with subtyping

Another way to achieve polymorphism is to make explicit in the code the idea that a given abstraction is a special case of an even more general abstraction. It makes use of *inheritance*, a technique allowing to build hierarchies of concepts, where *base classes* are parent nodes and *derived classes* (subtypes) are children nodes, and the inheritance a *IS A* relationship. For example one could think that a gaussian model *is a* dispersal model. It allows to write code only in terms of the most general abstraction, without caring about the variety of subcases:

```
Coordinate walk(Coordinate x, Model& model){
  return model->sample(x);
}
```

The syntax is a little bit different than for the generic programming. First we need to use pointer or references to enable runtime polymorphism. Then the `Model` class has to be defined:

```
class Model{
  // pure virtual function
  virtual Coordinate sample(Coordinate x) = 0;
};
```

The `virtual` keyword is essential, as it says to the compiler that the `sample` behavior in the general case can be overriden by the subtypes (that is, that the computer has to use the derived classes dispersal models methods).

Of course, in the context of dispersal models it makes no sense to define a general implementation for the `sample` method. That is precisely the meaning of the `=0` syntax, that is used to define a *pure virtual function*:this is just the C++ manner to express that the `Model` class is a pure abstraction because there is no general way to sample a coordinate.

Remark. The `= 0` syntax seems strange and cryptic. Stroustrup explains that "it was chosen over the obvious alternative of introducing a new keyword `pure` or `abstract` because at the time I saw no chance of getting a new keyword accepted. [...] I used the tradition C and C++ convention of using 0 to represent *not there*." (Stroustrup, 1994, chapter 13.2.3)

Now that the the general model has been defined, as well as the way to manipulate it, various subtypes of a dispersal model can be created, for example:

```
class GaussianDispersion : public Model{
  Coordinate sample(Coordinate x) override {
    // use gaussian density to return a coordinate
  }
};

class UniformDispersion: public Model{
  Coordinate sample(Coordinate x) override {
    // use uniform density to return a coordinate
  }
};
```

Now, any of these subtypes can be passed to the `walk` algorithm, and at runtime the appropriate behavior will be invoked. The effects can seem quite similar to these obtained using static polymorphism. However they are a number of inconvenients using runtime polymorphism.

Choose the right type of polymorphism

Types hierarchies are quite rigid. Because all problems are not of hierarchical nature, it will often be difficult to add a type in a hierarchy, so constructing the inheritance hierarchy to represent the problem will result in a counter intuitive design.

Then, as more subtypes are needed, it happens that hierarchies become deeper and deeper. This should absolutely be avoided, because the inheritance (that is, a *is a* relationship) is the strongest form of dependency that is possible to establish between to types, and this dependency is propagated across all classes of the inheritance tree, so bringing modifications to the design will be extremely difficult.

Finally, runtime polymorphism can be very inefficient, because the `virtual` keyword prevents the compiler to make optimizations like inlining, and this can lead to code up to 50 times slower (Stroustrup, 2014). Runtime polymorphism can be useful for example to decide a type according to a simulation context, but it should not be used carelessly. Actually it should be avoided when no strictly necessary:

> Polymorphism is one of the main reasons why object oriented programs can be less efficient than non-object oriented programs. If you can avoid virtual functions then you can obtain most of the advantages of object oriented programming without paying the performance costs. (Fog, 2012)

2.5.4 L - Liskov Substitution Principle (LSP)

The LSP is a particularly strong definition of a subtyping relation, allowing to build inheritance relationship in a more robust way. It is a strong behavorial subtyping principle based on the following requirement:

Definition 2.5.3. Liskov Substitution Principle Let $\phi(x)$ be a property provable about objects x of type T. Then $\phi(y)$ should be true for objects y of type S where S is a subtype of T (Liskov and Wing, 1994).

In other words, that means that a subtype should extend, not substitute, the base class behavior. A classic example is the relation between a *rectangle* and a *square*. A programmer deciding to formalize the fact the a square *is a* rectangle will surely make a `Square` class derive from a `Rectangle` class. It can seem reasonable, as mathematically speaking a square is indeed a particular case of a rectangle. However, establishing this relationship in the code will totally break the intuition one has about a square, because a square does not *behave* like a rectangle: a rectangle can change its height and its width independently, but a square as a supplementary invariant, that is that height and width must be equal in any case. That means that a `setHeight` method inherited from the `Rectangle` class does not make any sense in the context of the `Square` class. As said earlier, all concepts do not fit well in a class hierarchy ...

A somewhat easier definition was given by Robert C. Martin:

Definition 2.5.4. Liskov Substitution Principle Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it (Martin, 1996c).

With this definition, it is easier to understand that a class that does not respect LSP is violating the OCP, as it would have to be modified each time that a new derived class is added. Runtime Type Information (RTTI) is a common violation of LSP arising often when one aims at identifying the real type of an object in order to call the appropriate function. Imagine that we would like to print the parameters values of the model `m` used for modeling the dispersal. We would like to print something like `mu=0, sigma=1` for a gaussian model instance, and `a=0, b=10` for a uniform model. As the names of the parameters change from one model to another, one could be tempted to write the following code to obtain the desired behavior:

```
void PrintModel(const Model& m)
{
  if (typeid(m) == typeid(Gaussian))
    PrintGaussian(static_cast<Gaussian&>(m));
  else if (typeid(s) == typeid(Circle))
    PrintUniform(static_cast<Uniform&>(m));
}
```

The implementation of `PrintGaussian` and `PrintUniform` would be something like:

```
void PrintGaussian(Gaussian m){
  std::cout << "mu=" << m.mu() << ",␣"
            << "sigma=" << m.sigma() << std::endl;
}


void PrintUniform(Uniform m){
  std::cout << "a=" << m.a() << ",␣"
            << "b=" << m.b() << std::endl;
}
```

The design seems to respect the SRP as the responsabilities to represent a mathematical model and to print the information about this model are separated. However it clearly violates the OCP, as adding a new class to the hierarchy (like a fat tail dispersal kernel) would break this code and force the developer to modify the existing `PrintModel` function to update the decision tree. Generally speaking, just

as deep inheritance trees, decision trees based on types should be treated with high suspicion.

2.5.5 I - Interface Segregation Principle (ISP)

Definition 2.5.5. Interface Segregation Principle Clients should not be forced to depend upon interfaces that they do not use (Martin, 1996b).

The best way to understand this principle is by identifying the signs indicating that ISP is broken.

We defined the `Model` abstract class to represent any kind of dispersal model. The `sample` method was first added because the first desired behavior of a dispersal model was to sample a new coordinate. However, as the project grows, we begin to use dispersal models in an ABC context, where the internal parameters of the dispersal model need to be sampled in some prior distribution: so the `resample_parameters` method is added to the `Model` interface. During the first steps of the project, only very standard laws are considered, like the gaussian kernel or the uniform kernel. A bit later, it appears that the density of the underlying probability distribution may also be known, and that leads to add a third public method, a `get_density_distribution`:

```
class Model{
  virtual Coordinate sample(Coordinate x) = 0;
  virtual void resample_parameters() = 0;
  virtual void get_density_distribution() = 0;
};
```

Problems arise when designing a new dispersal model that does not need or can not define the totality of the interface. For example, for a Unit Test on dispersal features (implemented in terms of the `Model` abstraction), it is likely that a handy toy-model would be to implement a two demes dispersal with a Bernouilli law. Obviously ABC resampling is irrelevant in the context of unit testing, and the density of a Bernouilli law is not defined. But because the dispersal algorithms have been defined in terms of the `Model` abstraction, it forces the user to define all the interface methods, even if some of them are not needed or possible. That leads to loose time in a forced implementation or to increase the code complexity with strange code where runtime errors are thrown because the desired behavior makes no sense:

```
class Bernouilli : public Model{
  Coordinate sample(Coordinate x) override {
    // sample and return a coordinate
  }

  void resample_parameters() override {
    // No time for implementing this behavior
    throw std::runtime_error("Not␣implemented␣here");
  }

  void get_density_() override {
    // It makes no sense for discrete distributions
    throw std::runtime_error("No␣density");
  }
};
```

Obviously the problem grows as more methods are added to the interface. The solution is to decompose the big `Model` interface into smaller interfaces (called *role interfaces*), so that the client code can depend only on the interfaces that it uses, what reduces dependencies and code complexity. In substance, this is the advice given by the Interface Segregagtion Principle.

2.5.6 D - Dependency Inversion Principle (DIP)

Definition 2.5.6. Interface Segregation Principle High level modules should not depend upon low level modules. Both should depend upon abstractions. Abstractions should not depend upon details. Details should depend upon abstractions. (Martin, 1996a).

In an application, the direction of dependencies is generally going from high level components (that reflect the main ideas of the implementation model) to low level components (that implement the details of a system). If high level components are defined in terms of low level components, then a change affecting low level components will force the higher level to be modified too. It must seem counter-intuitive (and undesirable!) that the core functionalities of an application could be jeopardized by changes in minor details. Furthermore, this form of dependency prevents

high level components to be reused in other contexts. The Dependency Inversion Principle advises to reverse this conventional direction of dependencies: the dependency is not reduced, but it is rather shifted so that the components that are worth to be reused (that is the high level components) can be reused.

The way to achieve it is by designing high level components that express what they need (their dependency) in terms of an interface (like an abstract class). It will then be the user's responsibility to provide the desired implementation to the component.

For example, a demographic simulator is likely to have some *high-level design* (like a loop over time, a loop over space, some form of population growth and some form of dispersal). The high level component describing this simulator should not depend on the `Gaussian` class defined in a *dispersal models* submodule, but it should depend on an interface representing any dispersal model. In doing so, the simulator can be reused in another context, where the *dispersal models* submodule is not available.

2.6 QUETZAL - an open source C++ template library for coalescence-based environmental demogenetic models inference

2.6.1 Abstract

The purpose of this article is to introduce an implementation framework enabling us, using available genetic samples, to understand and foresee the behavior of species living in a fragmented and temporally changing environment. To this aim, we first present a model of coalescence which is conditioned to environment, through an explicit modeling of population growth and migration. The parameters of this model can be infered using Approximate Bayesian Computation techniques, which supposes that the considered model can be efficiently simulated. We next present Quetzal, a C++ library composed of reusable generic components and designed to efficiently implement a wide range of coalescence-based environmental demogenetic models.

2.6.2 Introduction

Motivations

Understanding how species react to spatio-temporal environmental heterogeneity and how this conditions the patterns of genetic variation is of great importance in the context of conservation biology, for example to predict future species distributions under global climate change (Pauls et al., 2013). Spatially explicit simulation studies have proven to be of fundamental importance when tackling such dynamical processes, especially in the context of range expansions (Excoffier, Foll, and Petit, 2009). Despite a growing number of simulation programs dedicated to coalescence-based models of genetic variation, code reuse is still limited. We present Quetzal, a new C++ library with reusable generic components designed to ease the implementation of a wide range of coalescence-based environmental demogenetic models, and to embed the simulation in an Approximate Bayesian Computation (ABC) framework. The code is open-source, and available at `https://github.com/Becheler/quetzal` (see Becheler, 2017).

Context

Present genetic data can be linked to past ecological processes by coupling demographic models accounting for the spatio-temporal landscape heterogeneity with models of genetic variation (see Figure 2.3). When the studied genetic variation is neutral, genetic models based on coalescent approaches (Nordborg, 2001; Hein, Schierup, and Wiuf, 2004; Wakeley, 2009) can be used. In this framework, the coalescence of two gene copies into a parent copy is simply the replication of the ADN viewed backward in time. The genealogy of the sampled genes copies can be defined backward in time conditionally to the demographic process which itself can be defined before tackling genetical aspects. This is an important theoretical link between a genetic sample and the historical processes that shaped it, and it can be used for constructing statistical models allowing to estimate properties of these past processes on the basis of the present sample.

Constructing such estimates often relies on the study of the likelihood, that gives the probability of data to arise, as a function of the parameter θ of the statistical model. The likelihood function can be derived under simple models, but as theoretical advances steered models towards higher levels of complexity (migration (Beerli

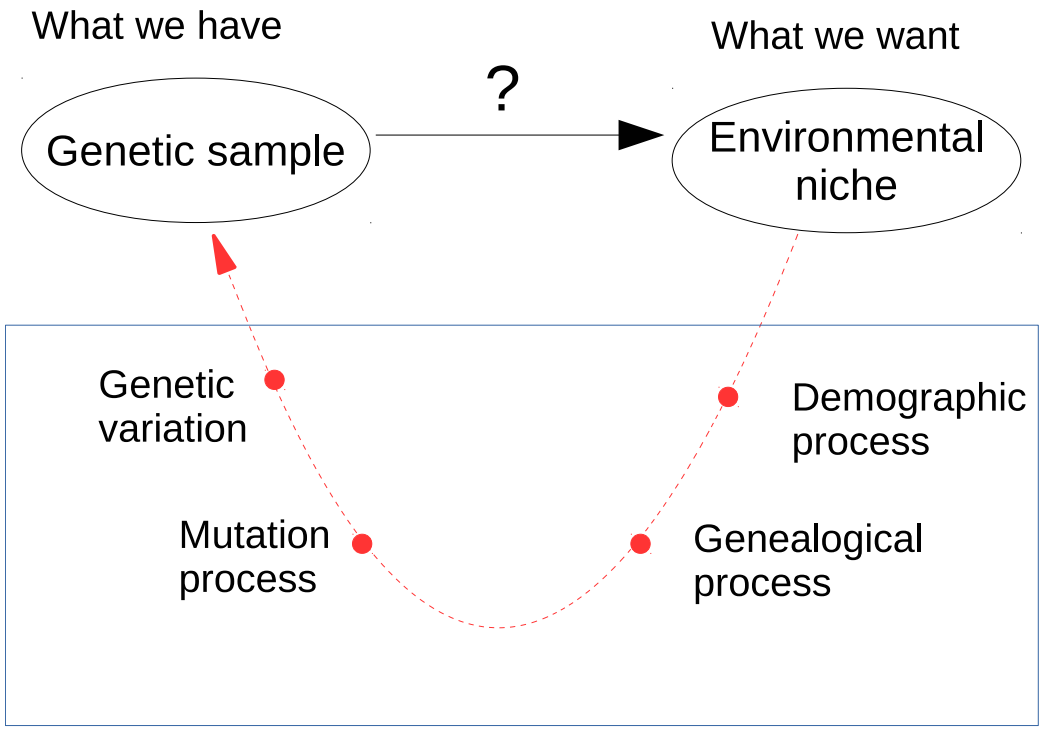


FIGURE 2.3: Inferencial framework from which Quetzal stems. The red arrow represents the dependency structure under the hypothesis of neutral variation, where processes condition the observed data. The black arrow represents the inferencial framework, where data allow to shed light on processes.

and Felsenstein, 1999), recombination (Kuhner, Yamato, and Felsenstein, 2000), selection), the likelihood function became harder and harder to calculate.

Approximate Bayesian Computation (ABC) methods (see Marin et al., 2012) dramatically extended the complexity limits of the models under which inference was possible. ABC bypasses the complex task of evaluating the likelihood function by combining two approximations making the problem computationally tractable: (i) observed data are reduced to lower-dimensional quantities (the so-called summary statistics), (ii) the inference is tolerant to small distortions of the observed summary statistics (see Blum et al. 2013 for more formal explanations). This makes possible for ABC procedures to estimate posterior densities of the parameters by simulating data under the model while exploring the parameter space conditionally to a prior distribution, and accepting only the values of the parameters for which simulated data are close enough to the observations. Despite its apparent ease, ABC methods present important methodological pitfalls (for example the choice of the dimensional reduction function), but many studies have paved the way for the non-statisticians (see Bertorelle, Benazzo, and Mona, 2010) for an excellent methodological guide), and ABC became very popular in Ecology and Evolution (Beaumont, 2010; Csilléry et al., 2010).

The popularity of ABC methods encouraged the development of more complex coalescence-based simulation computer programs, and their authors put tremendous efforts in successfully delivering novative, usefull and user-friendly products to the community of population geneticists. SPLATCHE (Currat, Ray, and Excoffier, 2004b) simulates coalescents based on complex demographic simulations run in a spatially explicit landscape, incorporating landscape heterogeneity. Various versions of SPLATCHE largely fostered the rapid expansion of the so-coined iDDC modeling approach (integrated distributional, demographic and coalescent modeling, He, Edwards, and Knowles 2013b). iDDC uses Approximate Bayesian Computation with spatially explicit demographic simulation model (possibly integrating landscape heterogeneity) to estimate quantities of interest such as populations growth rate or dispersal law parameters (Lacey Knowles and Alvarado-Serrano, 2010; Estoup et al., 2010b; Massatti and Knowles, 2016). DIY ABC (Cornuet et al., 2014) is an open-source program that provides the ability to conduct inference under a wide range of complex biological scenarios combining an arbitrary number of admixture, divergence or demographic change events. It offers very strong ABC

support and an intuitive Graphic User Interface (GUI). IBDsim (Leblois, Estoup, and Rousset, 2009) is an open-source program for simulating genetic variation under isolation by distance, and provides much flexibility in the choice of dispersal kernels. MSMS (Ewing and Hermisson, 2010) puts emphasis on incorporating selection and proposing extensible design. These programs, and others, have provided invaluable support to the non-developer communities for a wide range of applications and studies.

Paradoxically, these programs collectively failed to help the software developers community to write new programs, mainly due to very low rate of reusability in their source code. Because they put emphasis on the non-developer user, they act as rigid black boxes taking an input, processing it in some way configured by some form of User Interface (*e.g.* command line or GUI) and delivering the output. Indeed they can not be used if the underlying theoretical model does not belong to their predefined set of possible options (*e.g.* SPLATCHE does not allow to change the dispersal kernel or the local growth model) or if their computational solution does not answer to the question (*e.g.* if they write standard genetic summary statistics in output files when we would need to analyze genealogical properties). This current state of the art does not scale with the virtually infinite number of arbitrarily complex evolutionary or demographic models and the ever-growing number of statistical methods variants. We need standard, general, reusable tools for helping us to quickly build programs that can solve new problems. As written in Stroustrup (2003): "The key to fast development, correctness, efficiency, and maintainability is to use a suitable level of abstraction supported by good libraries". Reusable code such as library's relies on abstractions, syntactic constructions often opposed to performance. However, when it comes to ABC and massive simulations, performances become critical. C++ offers the template mechanism, a key feature allowing to build very high levels of abstraction without loss of efficiency, so we do not have to choose between reusability and performance.

As a first attempt to offer reusable components to the coalescence software community, we present here Quetzal, an open-source C++ template library. Quetzal offers powerful abstractions for building coalescence-based simulation programs. It contains several independent modules, each directed towards a general simulation purpose (demography, geography, coalescence, genetics, ABC ...) in which are

located the files containing the sources of generic components. The extensive documentation and the wiki (Becheler, 2017), both avalaible on the github project, ease to pick and combine the desired functionalities and the template mechanism allows to adapt it to a new problem with minimal recoding (if no recoding at all). We insist on the possibility to extend the behaviors of the Quetzal algorithms with very few new lines of code. The high level of abstraction in Quetzal allows its generic components to be used to write expressive and maintainable code with appreciable terseness and efficiency. Their genericity make them applicable to a large range of programs and the user can change the design of its application without major changes in the code (what typically arises when changing a finally-not-so-minor detail in the theoretical model). Each documented functionality comes with a small demonstration program and its output, providing valuable and intuitive insights on the way to manipulate the component. The library design insists heavily on modularity, extensibility and efficiency, and is intended to respect the standards of the Standard Template Library with STL-like algorithms and interfaces. Template rules make the library header-only.

This first version focuses on coalescence-based environmental demogenetic models. Consequently, Quetzal first features are designed to bring efficiency and flexibility in defining demographic quantities as functions of space and time of landscape heterogeneity, to couple these quantities to the coalescence process, and to use ABC methodology to conduct inference.

First we present a simple mathematical ecological model for estimating ecological features of a species from a genetic sample with the ABC procedure. This model is purposely general, as it will serve as an illustration to facilitate the understanding of Quetzal functionalities by enforcing its genericity, but it is still of strong biological interest as it relaxes several constraints that were previously made in the literature. Of course Quetzal is flexible and its use is not limited to this model as other features can be added or removed. Then we present a core feature of Quetzal for the coalescence, the abstraction of the ancestry relationship between a child gene copy and its parent, and we point out its importance to use and extend the library. Lastly, we give an overview of Quetzal functionalities and concepts and we apply them in a demonstration program implementing a fully-specified version of the previous general theoretical model.

2.6.3 Ecological and mathematical demogenetic model

Motivations

Let be a spatial sample S of n haploid individuals that have been sampled at time t_S across the landscape and that have been genotyped at a microsatellite locus, and a dataset giving the values of environmental quantities across the same landscape. From these data, we want to infer ecological properties of the species such as the *niche functions* (defined here as the functions relating the environmental quantities to demographic quantities, for example the local growth rate) and the dispersal function, so we need a statistical model to link observed data and processes to infer (see Figure 2.3). We present here a description of a general bayesian model for estimating these functions using an ABC framework (see Figure 2.4): a demographic history is simulated forward in time conditionally to the features of the heterogeneous landscape, then the genealogy of the sampled gene copies is simulated backward in time conditionally to the demography. The uncertainty on the parameters $\theta \in \mathbb{R}^p$ to estimate is defined by the prior distribution Π from which a parameter θ is sampled at each simulation.

Geography

Let consider a given set of demes $\mathbb{X}$ (typically reduced to the geographic coordinates of their centroïd). The environment E is defined by i known ecological quantities which are functions of space and time, typically climate layers from the WorldClim global climate database (www.worldclim.org), or a niche suitability dataset estimated from an external niche modeling step (He, Edwards, and Knowles, 2013b).

$$\begin{array}{rcccl} E & : & \mathbb{X} \times \mathbb{N} & \mapsto & \mathbb{R}^i \\ & & (x,t) & \mapsto & (E_i(x,t))_{i \in \mathbb{I}} \, . \end{array}$$

Demography

The demographic simulation process goes from time t_0 to t_S and iteratively constructs the function N giving the number of individuals in deme $x \in \mathbb{X}$ at time t:

$$\begin{array}{rcccl} N & : & \mathbb{X} \times \mathbb{N} & \mapsto & \mathbb{N} \\ & & (x,t) & \mapsto & N(x,t) \, . \end{array}$$

N is initialized by setting $N(.,t_0)$ the initial distribution of individuals across demes at the first time t_0. Typically for a biological invasion, this is restricted to the introduction site(s) with the number of introduced individuals (Estoup et al., 2010b). For endemic species, paleoclimatic distribution can be considered as starting point. The number of descendants $\tilde{N}_x^t$ in each deme is sampled in a distribution conditionally to a function of the the local density of parents, for example $\tilde{N}_x^t \sim Poisson(g(x,t))$, where g can be for example a discrete version of the logistic growth as in Currat, Ray, and Excoffier (2004b).

$$g \; : \left| \begin{array}{ccc} \mathbb{X} \times \mathbb{N} & \mapsto & \mathbb{R}^+ \\ (x,t) & \mapsto & \frac{N_x^t \times (1+r(x,t))}{1+\frac{r(x,t)\times N_x^t}{K(x,t)}} \end{array} \right. .$$

The r (respectively k) term is the growth rate (respectively the carrying capacity), defined as a function of the environmental quantities with parameter θ:

$$\begin{array}{rccl} K & : & \mathbb{X} \times \mathbb{N} & \mapsto & \mathbb{R}_+ \\ & & (x,t) & \mapsto & f_K^\theta(E(x,t)) \, , \end{array}$$

$$\begin{array}{rccl} r & : & \mathbb{X} & \mapsto & \mathbb{R} \\ & & (x,t) & \mapsto & f_r^\theta(E(x,t)) \, . \end{array}$$

Non-overlapping generations are considered (the parents die just after reproduction). The children dispersal is done by sampling their destination in a multinomial law, that defines $\Phi_{x,y}^t$ the number of individuals going from x to y at time t:

$$(\Phi_{x,y}^t)_{y\in\mathbb{X}} \sim \mathcal{M}(\tilde{N}_x^t, (m_{xy})_y) \, .$$

The term $(m_{xy})_y$ denotes the parameters of the multinomial law, giving for an individual in x its proability to go to y. These probabilities are given by the dispersal law with parameter θ:

$$\begin{array}{rccl} m & : & \mathbb{X}^2 & \mapsto & \mathbb{R}_+ \\ & & (x,y) & \mapsto & m^\theta(x,y) \, . \end{array}$$

After migration, the number of individuals in deme x is defined by the total number of individuals converging to x:

$$N(x,t+1) = \sum_{i\in\mathbb{X}} \Phi_{i,x}^t \, .$$

Coalescence

These quantities are used for defining the coalescence process which is defined by the following stochastic process going from t_s to t_0: knowing that a child node c is found in $j \in \mathbb{X}$, the probability for its parent p to be in $i \in \mathbb{X}$ is :

$$P(p \in i \mid e \in j) = \frac{\Phi^t_{i,j}}{\sum_k \Phi^t_{k,j}} .$$

Knowing that the parents p_1 (p_2) of nodes c_1 (c_2) are in x at time t, the probability for the children to coalesce in the same parent is :

$$P(p_1 = p_2 \mid p_1 \in i, p_2 \in i) = 1/N^t_i .$$

A forest of random coalescent trees is then constructed backward in time, until t_0 is reached. Note that at this point the Most Recent Common Ancestor is not necessarily found, and assuming that $t_S - t_0$ is small enough no neglect mutations, the simulation can end with a collection of trees rather than a complete genealogy.

2.6.4 Abstraction of the ancestry relationship

Motivations

When exposing the concept of coalescent trees above in the mathematical model, it has been useless to define exhaustively the tree properties or the exact nature of its nodes and branches. These are details humans typically *abstract away*, which leads to high generalization and low intellectual overhead. However, a computer program has to deal with an impressive number of details, and if these details are not carefully separated from the general concerns when writing the code, it leads to poor generalization (see Alexandrescu, 2001, p.xvii). Indicators of a lack of generalization are typically numerous dependencies across code, monolithic classes and a high rate of code duplication. Consequences are defined by Martin (2000) and Alexandrescu (see 2001, p.5) as being *rigidity* (the software is difficult to change), *fragility* (the software breaks at several points after a small change), *immobility* (the software can not be reused in another context, so it is entirely rewritten), gruesome intellectual overhead and poor performances. Since the devil is in the details, a natural solution

is to write code in terms of general *abstractions* rather than in terms of *implementation details* (Dependency Inversion Principle,Martin 2002b), so the designed generic components can be reused in various contexts.

Object-oriented paradigm

In C++, the genericity of the implementation can be realized by using inheritance and dynamic binding enable by the `virtual` keyword (Object Oriented Programming, OOP). Algorithms manipulating trees would then rely on an interface defined in terms of an abstract class `AbstractTree`, but will be applied on instances of concrete classes that inherit from `AbstractTree` and that present the specific desired behaviors, for example `TreeForStoringCoalescenceTimes` or `TreeForStoringCoalescenceDemes`. This design avoids the well-known problems of a class that would expose a monolithic interface to store all possible features like demes, times, mutations and others, (Single Reponsability Principle, Martin 2002b) but it has a number of well-known drawbacks. First, if inheritance can be very useful when the set of classes to be treated by the algorithm can naturally be thought as a hierarchy of concepts, in most cases this is not the case: it would result very unnatural to order them into a class hierarchy, and, importantly, it would lead to hardly maintainable code design (Stroustrup, 2014). Second, it is sometimes natural to expect the algorithms to work with primitive types or with STL containers, but as primitive types are not classes and as STL containers are not designed for dynamic polymorphism (they have no virtual destructor), there is no hope to see the object-oriented approach work with these types. Finally, the use of inheritance and virtual functions can have runtime overhead because of an extra lookup in the virtual table when a virtual function is called, and because virtual methods can not be inlined (Stroustrup, 2014).

Generic paradigm

All the data type manipulated by a same general algorithm do not have to be linked by the rigid hierarchical relation imposed by OOP : the generic programming allows for uniform manipulation of independent types. In generic programming an algorithm is not defined in terms a particular type, but in terms of a set of constraints wielded on the type by the algorithm internals; this set is frequently defined as a *concept* and represents an implicit interface. Thus the algorithm will work with any type fulfilling these constraints, allowing for high abstraction level without loss of

efficiency. Generic programming allows to implement the coalescence algorithms with great generalization, making minimal assumptions on the type handled.

The simple task to merge two nodes uniformly at random in a sequence of nodes can be defined as follow:

1. randomly permute the k elements of the sequence
2. create of a new node P (the parent)
3. designate the first element as child of P
4. designate the last element as child of P
5. assign P to the first element.
6. return the $k-1$ first elements.

When implementing this binary merge algorithm in Quetzal, several details need to be abstracted away to preserve the generality of the algorithm definition: the nature of the nodes, the nature of the sequence, of the nature of the inheritance relationship between a child node and its parent.

The nodes could be integers, character strings, a user-defined class or, actually, *anything else*. No constraint on this type comes from the algorithm, but various constraints can come from the sequence type used to store them. The sequence could be a standard container (`std::vector`, `std::list` ...) or a user-defined type. The classical way to abstract containers in C++ is passing as argument two iterators (one pointing to the first element of the sequence and the other pointing to the past-the-end element) giving the range of data on which the algorithm will operate. As the algorithm can not modify the external container, the reduction in size caused by the merge is signaled by returning an iterator pointing to the new past-the-end element. The only explicit effort the user can have to do is just to precise, conditionally to the chosen node type, what is meant by "*designating node c as child of the parent node p*". This can done by passing to the algorithm a function-object taking as argument a reference on the parent and another on the child, returning the result of the branching event. Or, if no function-object is given, the sum operand is used by default (thus requiring the expression $c + p$ to be defined).

This abstraction of the ancestry relationship is expected to allow (i) to efficiently generalize the existing Quetzal algorithms to an open-ended number of specific,

user-defined kinds of genealogies and (ii) to give guidance to the developers who need to write new generic coalescence algorithms.

Counting hanging subtrees leaves

Achieving genericity is of fundamental importance to efficiently tackle a wide range of situations. For example most of the current simulation softwares focuse on generating genetic variation samples, because this is most of the time the only information available. However, in some cases the mutational process can be negligible compared to the recent genealogical process in shaping the sample configuration (see Becheler et al., in preparation), so topological properties like the number of leaves of hanging subtrees (see Hein, Schierup, and Wiuf, 2004, p.78) become the desired output. This is very unlikely that the developer of a program could ever foresee this specific need. Fortunately, it does not mean one should recode everything from scratch each time a new simulation behavior is needed by a new methodological advance: Quetzal allows the user to inject the desired behaviors into its generic components.

When implementing the simulation, instead of building complex genealogical objects, then counting their leaves by tree traversals algorithms, a much more efficient approach is to directly make the coalescence algorithm sum the number of leaves of the hanging subtrees, updating it at each coalescence event. The type of nodes is thus defined as being integers, and the sampled nodes value is set to 1. Conveniently and by default, the merging algorithm will initialize the new parents to their default constructor value (which is 0 for integers), and define the branching event of two nodes by the sum function.

The following small program applies the approach by merging two nodes uniformly at random in a sequence of four nodes, updating the leaves number information. It can of course be extended to much more complex simulation frameworks.

```
#include "quetzal/coalescence.h"
#include <random>     // std::mt19937
#include <iostream>   // std::cout
#include <algorithm> // std::copy
#include <iterator>

using namespace quetzal::coalescence;
```

```
int main(){
  using node_type = int;
  std::vector<node_type> nodes(4,1);
  std::mt19937 rng;
  auto last = binary_merge(nodes.begin(), nodes.end(), rng);

  std::ostream_iterator<node_type> it(std::cout, "␣");
  std::copy(nodes.begin(), last, it);
  return 0;
}
```

The output gives the number of leaves of each hanging subtree after one generation of coalescence:

```
2 1 1
```

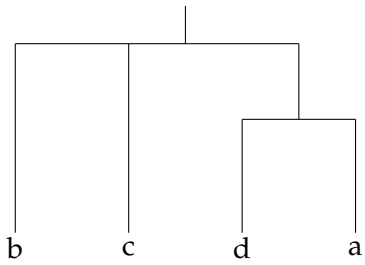
Construct a Newick tree format

Code with a suitable level of abstraction allows to readapt old code to new problems with ease. Studying genealogies topological properties can be the main statistical focus, but visualizing genealogies is still the most instinctive way to shed light on some properties, to assert correctness of algorithms generating them, or to present results. However, when it comes to data visualization, C++ is not the most suited platform. Many tree visualizer use a Newick tree format (see Olsen, 1990) as an input for nice plot rendering. The implementation is straightforward when using Quetzal abstractions: the type of the nodes is now a character string, the parent node is by default constructed as an empty string, and the branching event is defined as a formating function taking the parent node p and the child node c as argument to build the Newick format character string piece by piece. There are very few lines to change in the code to entirely redefine the meaning of a coalescence event:

```
#include "quetzal/coalescence.h"
#include <random>     // std::mt19937
#include <iostream>   // std::cout
#include <algorithm> // std::copy
```

```
#include <iterator>
#include <string>
using namespace quetzal::coalescence;

int main(){
  using node_type = std::string;

  std::vector<node_type> nodes = {"a","b","c","d"};
  std::mt19937 rng;

  auto branch = [](auto p, auto c){
    if(p.size() == 0)
      return "(" + c;
    else
      return p + "," + c + ")";
  };

  auto first = nodes.begin();
  auto last = nodes.end();

  while(distance(first,last)>1){
    last = binary_merge(first, last, rng, branch);
  }

  std::ostream_iterator<node_type> it(std::cout, "␣");
  std::copy(nodes.begin(), last, it);
  return 0;
}
```

The output will give the Newick format character string `((d,a),b,c)` representing the following coalescent tree.



The output can be exported for example on an online tree viewer such as iTol (Letunic and Bork, 2006). Quetzal is not restricted to this simple example and any arbitrarily more complex formatting functions can be considered, for example to represent branches length or nodes position in a landscape.

2.6.5 Quetzal components for simulation

The manipulation of the genealogies is the most fundamental aspect of all coalescence-based application programs, so the abstraction of the ancestry relationship is expected to be always useful, and the algorithms written in terms of this abstraction are expected to be highly reusable. However, an open-ended number of generative model variants can be considered: we present here a number of components that are most likely to be necessary when implementing them. Note that these components are intended to be independent. For example, if a demographic simulation is usually run after reading some environmental quantities in a geographic file, this does not have to be the case. Indeed any user-defined set of coordinates can be used to represent the demic structure, and environmental quantities can for example be represented by any mathematical function of the geographic space. Accordingly, the type of the geographic coordinates used in the *geography* module does not pervade the other modules.

Discrete landscape construction

In the *geography* module, Quetzal uses the Geospatial Data Abstraction Library (GDAL Development Team, 2017) to read grids of ecological data through the instantiation of a `DiscreteLandscape` object. In this class, the demes are represented by the grid cells and identified by the geographic coordinate of their centroid. The class allows to retrieve the set of demes centroid geographic coordinates (so it can be used to represent the demic structure to run spatially explicit simulations), to reproject a set of sampled coordinates to the nearest centroid (so compatibility is ensured between

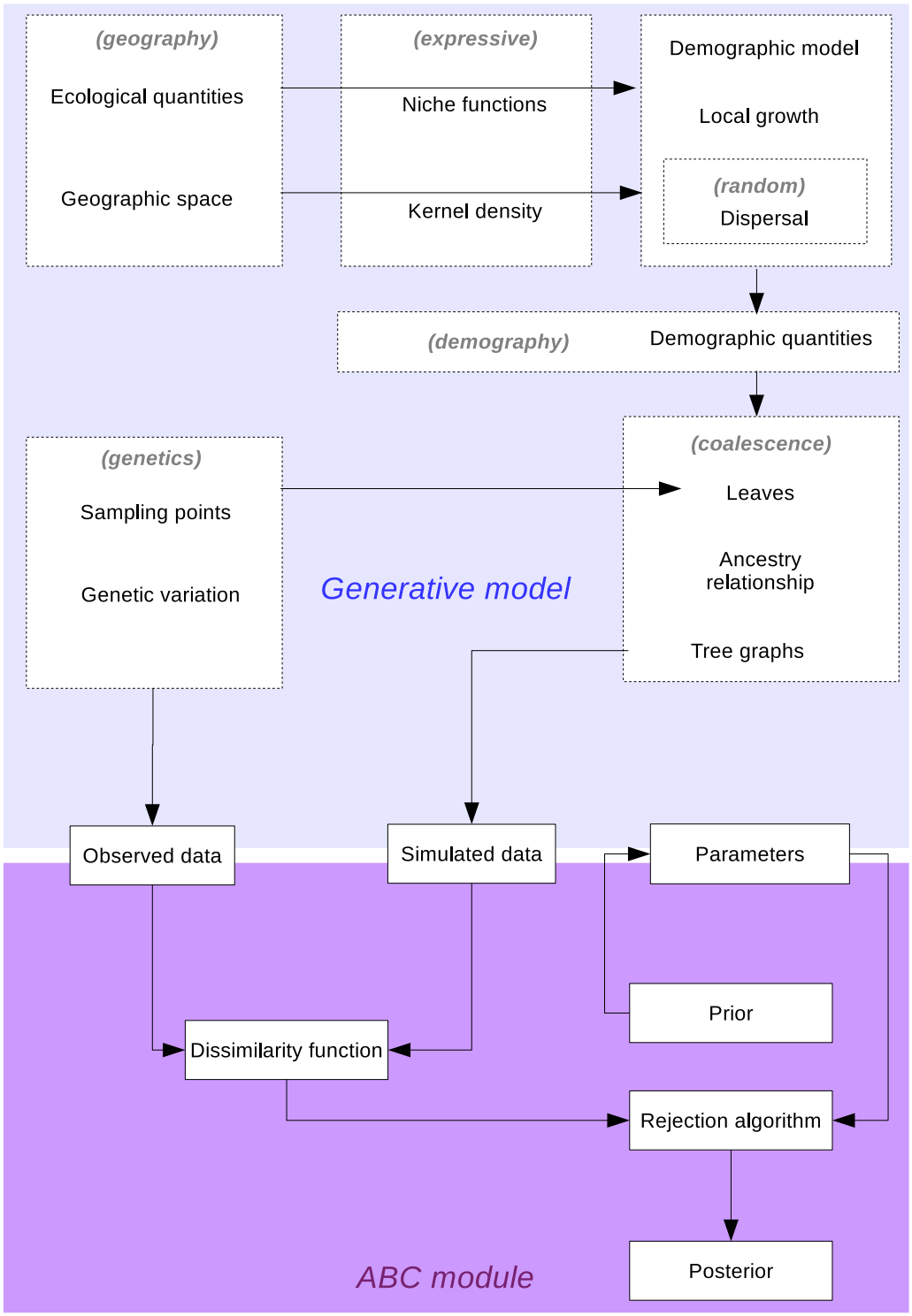


FIGURE 2.4: Flow chart illustrating the information flows between components of a general environmental demogenetic model (see section 2.6.3). In grey parenthesis are indicated the Quetzal modules (see section 2.6.5) that allow to represent these components in the code. The way the landscape conditions the demographic processes form the main focus of a number of approaches (landscape-ABC (Estoup et al., 2010b), iDDC modeling (He, Edwards, and Knowles, 2013a)) in the literature, such that the inference is usually driven on the underlying niche and/or dispersal model. Infering such ecological properties from a spatial genetic sample is made possible by using a coalescence model to link the sample to the demographic processes that shaped it. Inference is run in an ABC framework, where parameters to estimate are sampled in a distribution, allowing a dataset simulated by the generative model to be compared to the observed data by some dissimilarity function to build the posterior.

a spatial sample and the geographic support), or to deliver lightweight function-objects that give access to the underlying ecological quantities and that are susceptible to be coupled to the demographic model by composing them into arbitrarily complex mathematical expressions of space and time. The `GeographicCoordinates` class allows secured manipulations of longitude and latitude coordinates, and computation of great circle distances (often useful to define dispersal kernels).

Demographic variables definition

In the *demography* module, Quetzal defines two class templates (`PopulationSize` and `PopulationFlux`) to construct and consult (N_x^t) and (Φ_{ij}^t), providing expressive interface for secured and intuitive manipulation. Both class templates do not depend on *geography* module as they are templated on the type of the locations (the demes) and on the type of values that are stored. It makes possible to use any geographic coordinate system (for example longitude/latitude using the `geography::GeographicCoordinates`) to spatialize the model. Any arithmetic type can be chosen to define the set in which (N_x^t) and (Φ_{ij}^t) take value (typically $\mathbb{N}$ or $\mathbb{R}+$). Indeed, the type of the stored time values is not necessarily an integer but can be a more complex date type.

Compile-time functions composition

Because growth patterns are species-specific features, the expression of N_x^{t+1} is typically user-defined and hence can be any arbitrarily complex function, for example constant values or discrete version of logistic growth model (Currat, Ray, and Excoffier, 2004b). Quetzal offers tremendous facilities to build these functions, composing function-objects into an expression that can be efficiently passed around. To this purpose, Quetzal integrates *expressive* (Marques, 2017), a library making use of metaprogramming techniques to enable compile-time optimization of function composition with high expressiveness.

Consider the discrete logistic growth version (see section 2.6.3) to define N_x^{t+1} and let us pretend there is strong biological motivations to want the growth rate r to be constant ($r = 3$) over space and time, and the carrying capacity K to be the mean of heterogeneous environmental quantities E_1 and E_2, $K = \frac{E_1+E_2}{2}$. As C++ has strong static typing, it is impossible to directly sum constants (literals) with functions, as it would be possible under others languages. So the first step is to transform the

constants as functions with the same definition space as the other functions we want to combine:

```
literal_factory<space_type,time_type> lit;
auto r = lit(3);
```

Here r is now callable with time and space arguments, and can be composed with other *expressive* callable objects. Assuming that E_1 and E_2 are function-objects callable with time and space arguments (for example the function-objects delivered by the `DiscreteLandscape` class), the function `use` allows *expressive* to manipulate the expressions and gives them an enriched mathematical interface, so the addition or division operator can be applied on them:

```
auto K = (use(E_1)+use(E_2))/lit(2);
```

And finally, assuming that N is a function-object callable with space and time arguments, the whole growth expression can be built:

```
auto g = (use(N)*(lit(1)+r)/(lit(1)+((r * use(N))/K));
```

This expression can be captured in a lambda expression to simulate the number of gene copies after reproduction in deme x at time t:

```
auto sim_growth = [g](auto x, auto t, auto& gen){
    poisson_distribution<N_type> dis(g(x,t));
    return dis(gen);
}
```

This object can be passed around conveniently for further invocation with space and time arguments:

```
for(auto t : times){
  for(auto x : space){
    // ...
    auto N_tilde=sim_growth(x,t,gen);
    // ...
  }
}
```

This last code snippet illustrates an important benefit of *expressive*: to allow the separation of concerns when writing the application code. In other terms, the details

of the logistic growth expression are not intricate with the code where the expression is invoked, what would result in an obfuscated and hardly maintenable code. Moreover, as the expression is known at compile-time, it is a perfect candidate for all compile-time optimizations done by the compiler such as inlining: as the compiler knows exactly which functions will be called when g is called, he should be able to replace a function call directly by the body of the function, potentially leading to extremely efficient code with very few indirections.

Dispersal patterns

In the *random* module, the `TransitionKernel` class is an implementation of a markovian kernel for sampling the next state x_1 of a markovian process conditionally to the present state x_0. It can be used in the dispersal context (in that case the states type will represent demes coordinates, for example `geography::GeographicCoordinates`). The underlying markovian probability distributions associated with each present state do not need to have the same arrival space, and they are built only if needed by the simulation context. The weights are computed with an arbitrary mathematical function conditionally to the present state (for example when only geographic distance affects dispersal) or to the present state and time (for example when environmental spatio-temporal heterogeneity affects dispersal).

Coalescence features

For coalescence under the Wright-Fisher model, a binary merge algorithm is proposed to be used in the simulation contexts where the sample size is small relative to the population size. A simultaneous multiple merge algorithm can be used when this approximation does not hold.

The benefit of abstracting the inheritance relationship when simulating the genealogical graph was presented above along with two examples showing that an explicit representation of the coalescent was not necessarily desirable. However in many standard cases, it is needed, for example to save arbitrary information from the simulation context (times of coalescence events) and access them later for updating some quantities while descending the genealogy (for example apposing mutations with a probability conditional to the time spent between two nodes). For these cases, the `Tree` class template allows to construct such object, encapsulating an arbitrary user-defined data field into each node, defining the inheritance relationship between

a parent node and a child node in a secured way, proposing topological manipulation operations and tree traversal algorithms.

The `Forest` class template is designed to ease the manipulation of spatial collections of trees (of arbitrary type) when using spatially explicit coalescence simulation.

2.6.6 Quetzal components for inference

The Quetzal *abc* module provides abstractions allowing to embed efficiently an open-ended range of simulation models into an ABC framework. To this purpose, an ABC object associates a simulation model and the prior distribution of its (possibly multidimensional) parameter to conduct inference. We present here key elements of the *abc* module, notably the concept of *GenerativeModel* used to abstract out the model-specific details.

As ABC-based inference on spatial coalescents involves complex functions for dimensional reduction and distance computation that are far beyond the scope of this article, the ABC inference examples will be presented with a toy generative model (the poisson distribution), a toy dimensional reduction function η (identity) and a toy distance ρ (absolute value of the difference).

Then we step away from the toy-model and propose a concrete example of a class satisfying *GenerativeModel* and implementing a fully-specified version of the general theoretical model of coalescence presented in section 2.6.3. This example will make use of Quetzal components to illustrate how to build original coalescence simulations objects with ABC-compatible interface.

Features

The *GenerativeModel* concept

To achieve genericity and propose clear, standard and uniform ways to manipulate models and parameters, specific simulation models are abstracted to the concept of *GenerativeModel*, a Quetzal C++ concept that has been designed to be a generalization of the standard C++ *RandomNumberDistribution*. It notably generalizes the type of the result that is no more restricted to be an arithmetic type, so more complex type values (coalescents, genetic data, or summary statistics) can be generated. Furthermore, the returned values are not necessarily generated from a simple probability density function or a discrete probability distribution, as generally in ABC a complex

stochastic simulation function is involved. The list of all requirements can be found in the documentation. Any model object whose type `D` satisfies *GenerativeModel* and any prior object able to randomly produce an object of type `D::param_type` can be used to build an `ABC` object. Consequently, all the STL random number distributions are compatible with the *abc* module, which is very convenient for testing and demonstration purposes:

```
using model_type = poisson_distribution<>;
uniform_real_distribution<double> prior(1.,100.);
model_type model;
auto abc = make_ABC(model, prior);
```

Here an ABC object is constructed by associating the STL poisson distribution with a prior on its parameter, the STL uniform distribution, for sampling parameters in [0,100].

Prior predictive distribution sampling

The generation of the reference table is done by sampling n results in the prior predictive distribution.

```
mt19937 g;
auto n = 1000000;
auto table = abc.sample_prior_predictive_distribution(n,g);
```

The generated `ReferenceTable` object can compute other table objects. Considering a function-object representing the summary statistics function η, the raw data table can produce a second `ReferenceTable` object associating the parameter value to generated summary statistics.

```
auto eta = [](auto x){return x;};
auto sumstats = table.compute(eta);
```

Generated data can be accessed, for example to be used as pseudo-observed data in ABC validation methodology:

```
auto pod_value = sumstats.begin()->value();
auto pod_param = sumstats.begin()->param();
```

Then, considering a function object representing ρ, the distance function between observed and simulated dataset,

```
auto rho = [](auto obs, auto sim){return abs(obs - sim);};
auto distances = sumstats.compute_distance_to(pod, rho);
```

Finally the syntax of the various interfaces make it intuitive to design a quick rejection algorithm, sending to output only the parameter values for which the generated summary statistics was less than a threshold:

```
double threshold = 2.0;
for(auto const&it : distances){
  if(it.value() <= threshold){
   cout << it.param().lambda() << endl;
  }
}
```

Rejection samplers

The simplest samplers is the Rubin rejection sampler (Rubin, 1984). It accepts a parameter value only if the generated data is strictly equal to the observed data (the data type has to be *EqualityComparable*, *i.e.* having a built-in or a user-defined comparison operator `operator==`).

The implementation of the Pritchard rejection sampler (Pritchard et al., 1999) generalizes the dimensional reduction function (that traditionally computes summary statistics) and the distance function (that evaluates the distance between observation and simulation). Therefore, any object-function can be used to transform data into summary statistics and any type of distance can be used.

More complex sampling algorithms like MCMC-ABC (Marjoram et al., 2003a), SMC-ABC (Del Moral, Doucet, and Jasra, 2006), or PMC-ABC (Beaumont et al., 2009) are yet not implemented, but we do not expect that it will hinder Quetzal reliability. Indeed these algorithms are known to be challenging to calibrate (Marin et al., 2012), while embedding the simulation model and the inference framework in the same C++ application code has the benefit to make all the type information available for the compiler, decreasing the computational cost, so the generation of the reference table alone is expected to be useful for a wide range of situations. Furthermore, we expect that if more sophisticated versions of algorithms are needed, the Quetzal existing concepts will greatly ease their implementation.

2.6.7 Implementing a custom generative model

We present here how to construct a class `ExampleModel` that meets the requirements of the *GenerativeModel*. The main general ideas are highlighted here and the program can be found in the supplementary material.

We consider a landscape reduced to two demes A and B. At time t_0, $N_0 = 10$ haploid individuals were introduced in deme A. The local growth rate and the local carrying capacity K are assumed to be constant across the landscape. The growth rate is known ($r = 100$) while the local carrying capacity K is unknown and assumed to belong to $[1, 500]$. The aim of the program is to estimate this value starting with a uniform prior distribution on $[1, 500]$. For each individual there is a probability $m = 0.1$ to migrate to the other deme. After $g = 10$ generations, $n = 30$ individuals were sampled in B and genotyped at one locus. We assume that each introduced individual had different allelic states and that mutational process is negligible. Under these hypotheses, the observed clustering of the data is only shaped by the genealogical process, so we can reject all simulated coalescent forests that do not clusterize the dataset into as many subsets of same cardinality than in the observed clustering. Consequently, we just need to construct the vector of the hanging subtrees leaves count (a way to do it efficiently was presented above) and to compare it to the observed vector of clusters size. We accept the parameter used for the simulation only if the two vectors are equals. For a demonstration purpose, we construct a reference table by sampling 5×10^5 simulated data into the prior predictive distribution, and we generate 100 pseudo-observed data under the parameter $K = 50$ for validation. Note here that if the pseudo-observed data do not contain 30 individuals in deme B, no posterior will be estimated. The prior and posterior distributions are shown in Figure 2.5.

ABC-compatible interface

Declaring the following interface is sufficient to capt the generality of all possible generative models and enable the `ExampleModel` class to interact with an `ABC` object:

```
class ExampleModel{
public:
  using param_type = Param ;
  using value_type = ... ;
```

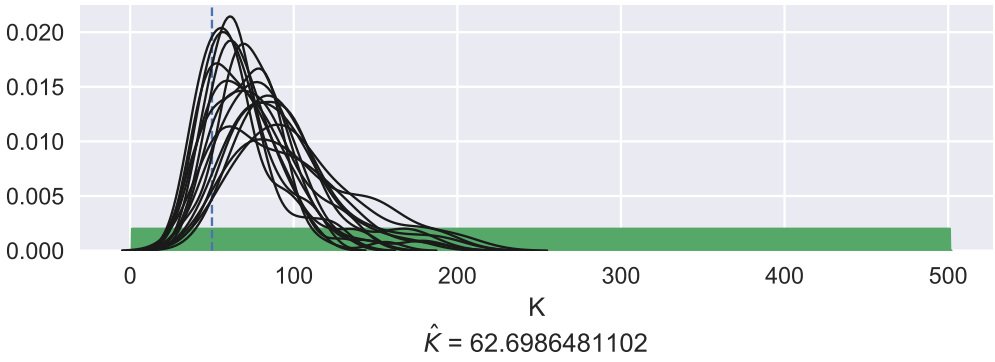


FIGURE 2.5: Posterior densities obtained by ABC inference conducted on pseudo-observed data generated under the *ExampleModel* model shown in section 2.6.7. True parameter value $K = 50$ is shown as the vertical dashed line. The prior distribution is shown in green. The mean of all posteriors is given as $\hat{K}$.

```
  template<typename Generator>
  value_type operator()(Generator& gen, param_type const& p) const;
};
```

The member type `value_type` describes the type of value generated by the model. The member type `param_type` encapsulates the details of θ, the multidimensional parameter to estimate.

Encapsulating θ

The point here is to hide useless details (such as dimensionality) from ABC procedures while allowing the user to have control over its implementation. We warn against using vectors or arrays to represent multidimensional parameters: it would impose all dimensions to be of same type, and their index-based value access interface would later favor confusion between dimensions. Instead, we suggest to follow the STL standards and to implement user-defined small classes, with expressive getter/setter syntax. Here we show an extract of the `Param` class, where the member `k` represents K, the carrying capacity of each deme in the landscape. Other dimensions are not given here, but can be set to constants for better code maintainability, as found in the supplementary material.

```
class Param{
private:
  double K;
public:
  double K() const; // getter
  void K(double); // setter
  // ... other dimensions
};
```

Constructing the prior

Instances of this parameter class can be created in a prior distribution *i.e* a function that can be called with a random generator and that randomly produces a `param_type` object, manipulating the `Param` object *via* its interface to set its dimension values:

```
auto prior = [](auto& gen){
  ExampleModel::param_type params;
  params.k(std::uniform_int_distribution<>(1,500)(gen));
  params.r(100);
  params.m(0.1);
  return params;
};
```

More guidance in the design of this second-order function can be found in the project wiki. This function-object, representing the parameter joint distribution, will be passed to the `ABC` object, that will use it to generate random parameters and pass them to the model `ExampleModel::operator()` member function to generate random `value_type` objects. The model details lay in the definition of `ExampleModel::operator()` member function, and a possible implementation is proposed in the supplementary material.

2.6.8 Acknowledgements

We thank Ambre Marques who importantly contributed to the present Quetzal state by taking on her free-time to provide advice on generic paradigm and design issues, and to develop the *expressive* library.

We thank Florence Jornod who participated as an intern.

Arnaud Becheler was funded by a multidisciplinary project founded by the French Government (LabEx BASC, ANR-11-LABX-0034) that aims to provide new knowledge regarding the drivers of species distribution and to design innovative guidelines toward sustainable land management.

This work was partially supported by the Chair "Modélisation Mathématique et Biodiversité" of VEOLIA-Ecole Polytechnique-MNHN-F.X., by the Mission for Interdisciplinarity at CNRS and by the Institute for the Diversity, Ecology and Evolution of the Living World.

2.6.9 Data Accessibility

Quetzal source code can be found on github project (https://github.com/Becheler/quetzal). The README file redirects towards Quetzal resources (documentation, wiki, IRC channel). This program is a free software; you can redistribute it and/or modify it

under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

2.6.10 Authors Contribution

All authors participated equally in the mathematical model design. Arnaud Becheler implemented the C++ library. The article and the documentation of the Quetzal project were written by Arnaud Becheler in cooperation with the other authors.

Chapter 3

Strategies for coalescence simulation

Approximate Bayesian Computation methods require that various models can be efficiently simulated. However, changing hypothesis or details in a model can have important impacts on the underlying simulation code. This is notably the case of the algorithms used to perform coalescence. We present here technical solutions to write generic algorithms for genealogical manipulations. Furthermore, under certain conditions, the common approximation of considering only binary coalescence events does not hold, and alternative algorithms must be developed to consider simultaneous multiple coalescence events. According to the simulation conditions, different versions of these algorithms can be considered to increase ABC efficiency. These algorithms are presented, along with the technical solutions allowing to make them exchangeable components of higher-level code structures.

3.1 Wright-Fisher sampling algorithms

The word *coalescence* comes from the latin *coalescere* (*unite*, *join*). It is used across different fields to refer to the merging of two or more elements. For example, the word is used to designate the tendency of two or droplets to merge together. Or two bubbles, or particles. It can be used to designate the fusion of two biological tissues. In population genetics, it is used to refer to the merging of genetic lines backward in time to their most recent common ancestor. As the word *coalescence* in natural language is linked to a variety of use context, we will see how to define a `C++` concept of coalescence that is able to represent a wide variety of particular meanings.

This section is for pedagogical and documentation purpose. First, the theoretical coalescence setup in which the algorithms are developed is presented. Then we show how the details of a particular programming context are abstracted to design generic algorithms that can adapt to all contexts. This genericity is not found in the coalescence-based simulators state of the art, but the techniques enabling it is well-known in the modern `C++` community. We bridge this gap in this section by presenting concepts that can be reused to develop new generic coalescence algorithms.

Coalescence algorithms are likely to be unstable, because algorithms are relative to hypothesis which relevance is tightly linked to a changing user context. Besides, these algorithms are usually parts of larger structures in the code. We show how to design both algorithms and higher-level code structures in such a way that algorithms becomes options that can be passed at compile-time to higher-level structures, in order to choose the appropriate simulation behavior. It allows to reach high flexibility and code efficiency.

3.1.1 Theoretical setup

The neutral Wright-Fisher model

A central model in population genetics is the Wright-Fisher model, that describes the evolution of a haploid population through time, by describing how alleles are transmitted between generations. The model was implicitly defined by Fisher (1923) and explicitly defined by Wright (1931). This model (defined here for haploid populations) makes a number of assumptions:

- population size is constant in time
- the population described by the model is not spatially structured (panmixia).
- No selection or mutation, *i.e.* as individuals are equally fitted, their number of descendants follow the same law.
- Discrete and non-overlapping generations : all individuals reproduce/die at the same discrete time.

The model can be generalized for variable population size. In that case, these assumptions allow to define the genealogical process in the following terms: at each generation t, parents die by giving birth to N_t new individuals that pick their parent uniformly at random.

Under this model, the demographic process is defined *forward in time*. In other words, the number of descendants is defined conditionally to the number of parents. In contrast, the genealogical process is defined *backward in time*: each descendant picks its parent uniformly at random and independently. Interestingly, it allows to consider each process separately under the neutral hypothesis: first the demographic process to determine the number of parents and descendants, and second the genealogical process to assign each child at each generation to a parent at the previous generation. This model allows to trace back the ancestry of sampled genes (or sequences) in the demographic history of the population. As the more similar genes are expected to find rapidly their common ancestor, it allows to shed light on the underlying demographic process.

When looking at n sampled genes, some of them may originate from the same reproduction event in the previous generation, where the parental gene was replicated into several children genes. Forward in time, it looks like a classical reproduction event. Backward in time and looking at genes lines, everything happens as if some of the sampled lines merged into a common ancestor: this is called a *coalescence event*. Coalescence events lead the building of coalescence trees, where vertices are common ancestors of children nodes and where edges are ancestry relationship.

The large population size approximation

Coalescence theory has a long tradition of approximations. Whenever the considered number of lines k is much smaller than the parental population size N, it is very unlikely to observe more than one coalescence event. Furthermore, the number of lines that are coalescing is then unlikely to be superior to 2 (as children pick parents uniformly at random, the probability that three children pick the same parent is $1/N^2$). Based on this approximation, a variety of models give the distribution of the coalescence times: events times can then be efficiently simulated and two lines can be sampled uniformly at random to be merged. The popularity of this approximation in a wide range of models motivates that we implemented an algorithm that is general enough to merge two nodes at random in a sequence, independently from the specific modeling framework (that is, we do not care here about coalescence times, coalescence demes locations, allelic states ...).

In many cases this approximation can not be made. For example, in a biological invasion context, just after a long distance dispersal event the population size is very

small. In this case the probability that more than one coalescence is no longer negligible. When 8 lines are coalescing into 3 parents, it is likely to observe simultaneous events of multiple coalescence (that is, to observe for example a configuration like 4 children coalescing in the first parent, three children in the second parent and 1 not coalescing). It motivates the design of an algorithm that coalesce lines uniformly at random according to a given configuration of the collisions. Again, to increase generality and reusability this algorithm should be designed independently from the modeling context.

3.1.2 Common abstractions for coalescence algorithms

Abstracting the concept of data sequences: iterators

A common feature of coalescence algorithms under the Wright-Fisher uniform sampling is that they operate on a sequence of nodes. We said earlier that the Standard Library uses various containers to represent information sequences: the `vector` class or the `list` class are generally good candidates for basic sequences. As different types of containers are likely to be passed to the algorithm, this difference needs to be abstracted so the algorithm can work in all cases. The Standard Library makes use of iterators to abstract containers:

Definition 3.1.1. Iterator The Iterator concept describes types that can be used to identify and traverse the elements of a container. Iterator is the base concept used by other iterator types: InputIterator, OutputIterator, ForwardIterator, BidirectionalIterator, and RandomAccessIterator. Iterators can be thought of as an abstraction of pointers (memory adresses where are stored data)

Using iterators, we can move forward in the sequence using the `++`, some iterators can move backward using the `--` operator, the element pointed by the iterator can be accessed with the `*` operator. As they allow to manipulate any kind of containers, they are of fundamental interest in the design of coalescence algorithms.

Abstracting the tree data structure: template variable type

When writing the algorithm, it is impossible to foresee which `C++` class the user will want to use to represent a coalescent tree in the code. Indeed, previous programs like simcoal uses a complex class (the `TNode` class) to represent coalescent trees that involve the representation of very model-specific features: deme to which

the node belongs, mutations, mutation rates and range constraints for microsatellite data. Other coalescence approaches rather aim at simply representing the partition process that operates on coalescing nodes and do not worry about all these details (see for example Chapter 4). The nodes type can be abstracted simply by using a variable type with the *template* keyword.

Abstracting behavioral details: function objects

The variable node type alone allows to give flexibility on the type accepted by the algorithm, but it does not help much in calling the desired behavior on nodes to coalesce. For example, in the following example, two nodes of unknown type `T` have to be merged in a newly created parent node. Let's assume that the tree type is a binary tree data type like in simcoal (that is each parent node has two children : left and right):

```
template<typename T>
T merge(T node1, T node2){
  T parent; // parent node constructed
  parent.set_left_child(node1);
  parent.set_right_child(node2);
  return parent;
}
```

The problem with this code is that the behavior called on the node type is tightly dependent on the type itself: a *n-ary* tree type (with multiple potential children nodes) has no reason to offer the `set_left_child` method, so the algorithm will not work with this type. This problem often leads to code duplication. For example the following code snippet is a clumsy attempt to "generalize" the merge to more than two children nodes:

```
template<typename T>
T merge(T node1, T node2, T node3){
  T parent;
  parent.set_children(node1, node2, node3);
  return parent;
}
```

But again, if native data types like `int` are used for representing partitions of data, the algorithm will not work as the integer type does not offer a `set_children` in its interface (it would be very ugly to implement its own integer type). It seems that a new algorithm should be defined each time that a new class is used for the coalescence: obviously, this approach rapidly does not scale with the open-ended number of possible types used for representing trees. As the desired behavior is not defined at the moment where the algorithm is written, it should be abstracted too, and left at the discretion of the library user to define the right behavior to call. Abstracting behaviors can be done by passing as arguments functions that implement the desired behavior. There are many types of objects that can behave like functions, and they are called function objects:

Definition 3.1.2. (Function objects) A function object is any object for which the function call operator is defined. C++ provides many built-in function objects as well as support for creation and manipulation of new function objects.

Due to their ability to inject flexible behaviors in algorithms, function objects are central in the design of the Standard Library. Among all function objects, lambda functions (a feature brought by C++11) are very handy. This feature allows to build function objects very easily, with high locality (that is, the function can be declared next to the point it is used). Their benefits were so important that they radically changed the way programmers used function objects.

Instead of declaring something like `parent.set_left_child(child)`, a function object `binop` (a binary operator, that is an operator taking two arguments) will be used instead: `binop(parent, child);` The exact operations that are performed (and that define the parent-child relationship) are decided in the declaration of the `binop` function object, that is *outside* the merge algorithm (since the object function is passed as an argument of the algorithm). This way, the merge algorithm can be designed independently of all context-specific information:

- which object type will be used to represent a genealogy
- the parent-child relation type
- how the parent will be created (its initial internal state)

3.1.3 Binary merge algorithm

The aim is here to present an algorithm able to realize the Wright-Fisher sampling for merging two nodes uniformly at random, using the abstractions previously presented to guarantee a high level of genericity and reusability.

Expected behavior

We want this algorithm:

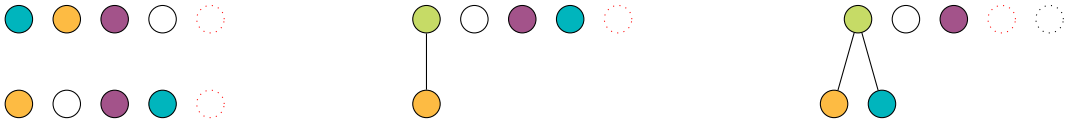
1. to perform the sampling of two nodes in a set of nodes
2. to create a new parent node
3. to set the sampled nodes as children of the parent
4. to remove the children from the nodes set
5. to add the parent in the nodes set
6. to return the transformed set

Sampling uniformly two nodes at random in a sequence is equivalent to rearrange uniformly at random the sequence and then to sample its first and last element. The Standard Library offers the `shuffle` algorithm to perform this rearrangement. All standard containers offer the `begin` and the `end` methods that give iterators pointing respectively on the first and on the past-the-end element, the sequence can be traversed from the beginning to the end by incrementing the iterator. Passing a container to a generic algorithm can then be done by passing these two iterators (generally called `first` and `last`). As the algorithm modifies the length of the container, it can signal it by returning an iterator pointing to the new end of the container. Figure 3.1 illustrates the behavior of the algorithm.

Generic implementation

Once the algorithm behavior is specified, it can be implemented with techniques respecting its genericity:

```
template <class It, class T, class Binop, class Generator>
It binary_merge(It first, It last, T init, Binop op, Generator& g)
{
```



(A) (B) (C)

FIGURE 3.1: Steps of the binary merge algorithm. 3.1a: a four nodes sequence is given to the algorithm by passing two iterators as arguments: one pointing to the first element of the sequence, the other pointing to the past-the-end element (this *end* iterator is represented as a dotted red node). As nodes have to be picked uniformly at random, the nodes in the sequence are first shuffled uniformly at random. 3.1b: A parent is initialized with the desired initial state (passed as an argument), and the first element of the sequence is set to be its child. 3.1c: then the last element is set to be the second child, and the reduction of the sequence size is indicated by moving the end iterator backward and returning it as a result of the algorithm. The updated sequence can then be accessed by iterating between the begin/end pair of iterators.

```
    std::shuffle(first, last, g);
    *first = op(init, *first);
    *first = op(*first, *(--last));
    return last;
}
```

In the code above, the most important line is the second line, giving the signature of the function: the function `binary_merge` takes two iterators (giving the range of a sequence of nodes) and returns an iterator (others arguments will be inspected further). This means that the merge algorithm can accept any kind of containers and that it modifies its size. Interestingly, all types manipulated by the algorithm are unknown (this is the meaning of the first line `template< ... >`) : these types will be decided at the compilation, as the user will apply this algorithm to well-defined types of nodes, containers, and function objects.

The first operation is the random rearrangement of the elements between `first` and `last`: this is done by using the standard `std::shuffle` algorithm, called with the random number generator (of unknown type) given by the user as argument. Abstracting the random number generator is important, as multiple implementations exist, so the library should not rely on a specific type.

Then the binary operation of unknown type is called with the `init` and the first element of the sequence. The result of this operation represents the parent node having as child the first element of the sequence. This result is then designated as

the new first element of the sequence (that is, the child is not anymore contained in the sequence).

In the third line, the last element of the sequence is accessed by decrementing then dereferencing the past-end iterator `last` and is used as a child for the parent (the new first element). As this child should not be part of the nodes sequence anymore, its iterator is returned to design it as the new past-end iterator: the number of elements between `first` and `last` is finally reduced by 1.

3.1.4 Simultaneous multiple collisions algorithm

The aim is here to present the implementation of an algorithm able to realize the Wright-Fisher sampling for merging several nodes in several parents uniformly at random. If the logic of the algorithm differs a bit from the binary merge algorithm (because it is more general), it makes use of the same abstractions previously presented. The important difference is that the algorithm needs a supplementary information to perform coalescence. This information is a description of the coalescence configuration, given by an occupancy spectrum (see section 3.2.3). An occupancy spectrum is a vector of numbers containing the description of how many new parents should be created, and the number of children each parent has.

Expected behavior

Let us consider a vector $(M_1, \ldots, M_n)$ representing an occupancy spectrum. We want this algorithm, for each $i \in [\![0; n]\!]$ of a given occupancy spectrum:

1. to create M_i new parent nodes
2. for each parent to perform the sampling of i nodes
3. to set the sampled nodes as children of the parent
4. to remove the children from the nodes set
5. to add the parent in the nodes set
6. to return the transformed set

A way to understand the behavior of this algorithm is to imagine that as new parents are created, the sequence of nodes is progressively emptied by its end, an iterator tracking the end of the sequence. Figure 3.2 illustrates the behavior of the

algorithm where the occupancy spectrum $M = 101$ has been given: this spectrum means that over 4 nodes, 1 node will no coalesce, 0 binary merge will occur, and 1 coalescence event with 3 children will occur.

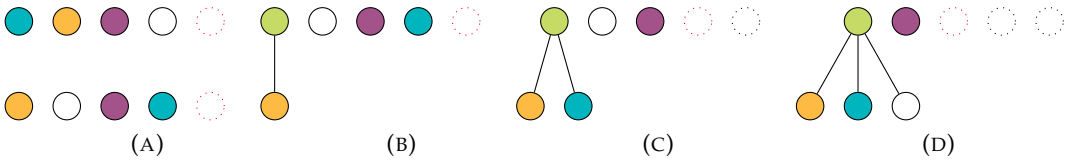


FIGURE 3.2: Steps of the simultaneous multiple merge algorithm. 3.2a: a four nodes sequence is given to the algorithm by passing two iterators as arguments: one pointing to the first element of the sequence, the other pointing to the past-the-end element (this *end* iterator is represented as a dotted red node). As nodes have to be picked uniformly at random, the nodes in the sequence are first shuffled uniformly at random. 3.2b: A parent is initialized with the desired initial state (passed as an argument), and the first element of the sequence is set to be its child. 3.1c: then the last element is set to be the second child, and the reduction of the sequence size is indicated by moving the end iterator backward. 3.2d the new last element is set as being the third child, and the end iterator is moved backward again. Once the algorithm is over, the end iterator is returned. The updated sequence can then be accessed by iterating between the begin/end pair of iterators.

Generic implementation

Once again, various template arguments allow to design an algorithm that respects both the expected behavior previously presented and a full genericity. The abstractions are mainly identical to those used precedently but their name were changed for aesthetical purpose only. An abstraction was nevertheless added to represent the occupancy spectrum. The motivation for not using a simple `std::vector` is that I wanted to reserve the possibility to use a dedicated class to represent the occupancy spectrum invariants (see section 3.2.3, equation 2 and 3).

```
template<class It, class T, class F, class S, class G>
It simultaneous_multiple_merge(It first, It last, T init, S sp, F op, G&
{
  std::shuffle(first, last, g);
  // directly go to binary merge
  auto m_it = sp.cbegin();
  std::advance(m_it, 2);
  int j = 2;
```

```
  // iteration on the occupancy spectrum
  while(m_it != sp.cend()){
    // loop on the m_j parents
    for(unsigned int i = 1; i <= *m_it; ++i){
      *first = op(init, *first);
      // loop on the others children
      for(int k = 1; k < j; ++k){
        *first = op(*first, *(--last));
      }
      ++first;
    }
    ++j;
    ++m_it;
    }
  return last;
}
```

Like the binary merge algorithm, this algorithm needs a sequence of nodes on which to operate: the sequence is contained between the two iterators passed as arguments. Then, a node model is given to initialize the parents at their initial state, and a random number generator is given for the random permutation of elements. The occupancy spectrum is given too, so the algorithm is independent from the random process that generates this coalescence configuration.

The first step is to randomly rearrange the elements. Then, the iteration on the occupancy spectrum numbers begins directly at the binary merge events (j=2). A first loop is run that creates the desired number of parents with initially one child, and a second loop affects to the newly created parent the desired number of children. When a parent has the right number of children, the `first` iterator is moved forward, so another cycle can begin. In the meanwhile, the last iterator is moved backward to signal that the sequence size decreases.

3.1.5 Guidelines for designing interchangeable strategies at compile-time

Multiplicity of possible designs

The precedent algorithms, deterministic in that they necessarily perform coalescence, can be used in higher-level stochastic algorithms. This higher level logic is associated to a more specific modeling setup. For example in a discrete-time model, the logic will decide if a coalescence event happens at a given generation. In a continuous-time setup, the logic will probably sample the time at which coalescence occurs. But in both cases, there are reasons to want to use either the binary merge algorithm or its multiple collision variant. This choice results from the approximation that can be (or not) done if the parental population size is large enough relatively to the number of lines. In discrete-time models, more than 2 children can have the same parent is the parental population size is small enough. Some continuous-time models incorporate the possibility for simultaneous multiple collision of lines. In any cases this choice is the user's choice, and is likely to change from time to time: this variability should not compromise the existing code base, and the user should be able to pass the algorithm version (binary or multiple merge) to use as an argument, and to do this efficiently.

Combinatorial explosion of possible behaviors

Often there are multiple ways to perform the same task, and this multiplicity is particularly difficult for the programmer to tackle. For example, in ABC models comparison methodology, the same data are simulated according to different models, and the quality of models is then assessed. In that case, models are a combination of various submodels (growth, dispersal, mutation ...) for which several variants exist. The direct and naive solution to represent each general model version by a class implementing a given set of options does not scale with the number of options: as the options combinations number grows exponentially with the number of options, the programmer would have to define an exponential number of classes differing only by some behavioral details. For example, to test the effect of the coalescent topology (2 modalities: binary tree or k-ary tree), the dispersal kernel (2 modalities: gaussian or fat-tail kernel) and the growth pattern (2 modalities: logistic or exponential models), 8 classes would have to be defined, with hardly readable names reflecting what they exactly do:

- `class ModelBinaryCoalescenceGaussianDispersalLogisticGrowth`
- `class MultipleCoalescenceGaussianDispersalLogisticGrowth`
- `class ModelBinaryCoalescenceFatTailDispersalLogisticGrowth`
- `class ModelBinaryCoalescenceFatTailDispersalExponentialGrowth`
- ...

These classes need to be maintained and are likely to evolve, which will appear quite complicated. For example, if a supplementary option is envisaged (like a submodel to represent the link between the environment and the growth rate), the number of classes will raise to at least 16. Obviously, this is unmaintainable.

Decomposing a complex behavior into policy classes

Instead of defining multiple classes each implementing a fully specified set of many behaviors, this is better to define one general-purpose class (the simulation skeleton) and to decompose the simulation complex behavior into small independent behaviors that can be composed at compile time (the key for efficiency). The classes implementing these orthogonal behaviors are called *policy classes*.

```
template<class CoalPolicy, class DispersalPolicy, GrowthPolicy>
class DemogeneticModel;
```

And assuming some of these policy classes were implemented, they can be passed as template arguments to obtain the desired behavior:

```
DemogeneticModel<BinaryMerge, Gaussian, Logistic> model_1;
DemogeneticModel<BinaryMerge, FatTail, Exponential> model_2;
ABC::compare(model_1, model_2); // for example
```

Importantly, the user can define its own policy and give it to the `DemogeneticModel` class (assuming its policy implements the correct interface). The DemogeneticModel class is then open for extension, but closed for modifications, as the variables parts have been exported.

3.2 Algorithms for fast simulation of discrete-time coalescents with simultaneous multiple merger.

3.2.1 Abstract

Motivation

There has been a growing interest in discrete-time coalescent simulation models mixing binary and simultaneous multiple mergers. This allowed to widen the range of biological models under which genetic data can inform demographic processes without sacrificing performance. However, generating a multiple collisions configuration is still costly and code reuse is still limited.

Results

We address these two problems by presenting a set of generic, reusable simulation components for coalescent simulation with simultaneous multiple collisions. We use the *occupancy spectrum* which is a vector of numbers summarizing a coalescence configuration as a common interface for various simulation strategies. Among others, we notably propose a new algorithm allowing to efficiently sample a spectrum directly in its probability distribution.

Availability

All components are integrated as parts of Quetzal, an open-source C++ library for coalescence available at github.com/Becheler

3.2.2 Introduction

Genetic data allow to inform the dynamics of biodiversity in relation to environmental changes, giving access to the phylogeographic history of species (Brown and Knowles, 2012) or to biological invasions drivers (Estoup et al., 2010a). Considered the generally high-level complexity of the related models, Approximate Bayesian Computation (ABC, see *e.g.* Marin et al., 2012) techniques have shown to be very useful for estimating ecological parameters (Beaumont, 2010). These techniques rely on massive simulation capacity and on an efficient implementation. Coalescence approaches (see *e.g.* Wakeley, 2009) allow to dramatically lighten the computation load by focusing on simulating the genetic history of the sample (the *coalescent tree*)

conditionally to a given demography, rather than simulating both the demographic and the genetic dynamics of the whole population.

We focus here on coalescence models resulting from discrete-time Wright-Fisher models, where individuals reproduce and die immediately with no generation overlap. When implementing the simulation program, the choice of the algorithms generating the coalescent and the reliability of the underlying assumptions heavily depend on the order relation between the population size N and the sample size n. Whenever $n << N$, the probability that more than two gene copies have the same parent is extremely low, so a fast binary merge algorithm can be used.

However, there is a large range of problems for which this assumption does not hold. For example, several programs (notably various versions of SPLATCHE, Currat, Ray, and Excoffier 2004a) implementing one or another variation of an environmental demogenetic model have been used in the context of bioinvasions. In environmental demogenetic approaches (that encompass iDDC modeling, see He, Edwards, and Knowles, 2013b and **Becheler214767**), the population size distribution over space and time is constructed by some complex stochastic process accounting for environmental dynamics, and is not meant to be directly estimated. In this context, a limited number of introduced individuals spread and reproduce across the landscape, so N_x^t, the population size in deme x at time t, is highly stochastic. If the population size depends on parameters to estimate, the random parameter sampling which is part of the ABC procedures makes it harder to guarantee that $n_x^t << N_x^t$. Actually, at some point of the simulation, N_x^t can be close to 1. Obviously the probability that more than two gene copies have the same parent is no more negligible, so a simultaneous multiple merge algorithm (SMMA) should be used. However SMMA is known to dramatically slow down the simulation, what is opposed to the performance constraint imposed by ABC procedures.

In this article we first introduce the term of *occupancy spectrum* as an abstraction of a simultaneous multiple coalescence configuration. An occupancy spectrum resumes all the information needed to coalesce n indistinguishable lineages in N parents. It has multiple benefits in terms of implementation design. Notably, it acts as an interface and allows to consider interchangeable strategies for the simulation, opening the door for user-friendly compile-time customization of the simulation behavior.

We then compare different strategies for simulating coalescence configurations.

First we present a straightforward implementation of a ball-to-urn assignment. Then we present a second strategy that samples a configuration directly in a distribution according to its probability. This strategy requires the construction of the discrete probability distribution of all possible configurations susceptible to arise when coalescing n lineages in N parents. So it demands that i) the support of the distribution can be efficiently constructed, ii) the probability of a configuration can be computed iii) the distribution can be constructed only once and sampled many times to scale with ABC procedures where massive simulations are needed. We present a new algorithm that allows to efficiently generate this support. The probability of a spectrum can be computed using urn and balls problems arguments. Memoization techniques allow to store the constructed distribution in memory for later sampling leading to fast coalescent simulations. As the support of the distribution can be huge, approximation strategies aiming at reducing its size are proposed, and their performance assessed.

All the algorithms and strategies presented are implemented in C++ as part of the Quetzal template library (**Becheler214767**). The project documentation presents how to easily switch between strategies using the policy classes available in Quetzal, and how to design customized composite strategies as policy classes for enhanced efficiency in the user-specific context.

3.2.3 Approach

Occupancy spectrum

Simulating the coalescence of n indistinguishable lines into m parents is actually an urn and balls experience where n indistinguishable balls are randomly placed in m urns, each urn having an assignment probability of m^{-1}. If an urn contains r balls at the end of the random experience, r is said to be the *occupancy number* of the urn (see Johnson and Kotz, 1977, p. 115). We then count the number of urns having the same occupancy number r (*i.e* the number of urns containing exactly r balls) is denoted M_r. We introduce the term *occupancy spectrum* to design the vector $M_0, M_1, \ldots, M_k$. We show in Figure 3.3 a possible output of throwing 8 balls in 5 urns, and how the configuration can be summarized by the occupancy spectrum. This abstraction allows to disentangle the generation of a coalescence configuration from the concrete implementation details of the coalescence event representation,

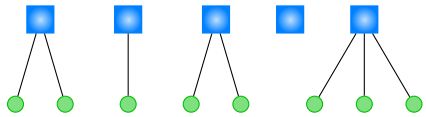


FIGURE 3.3: Possible output resulting from throwing 8 balls (green balls) into 5 urns (blue boxes), or equivalently from coalescing 8 lineages in 5 parents. The following terms can be calculated: $M_0 = 1$ is the number of urns (resp. parents) with no balls (resp. children), *i.e.* the number of parents with no descent in the sample. $M_1 = 1$ is the number of lineages that will not coalesce. $M_2 = 2$ is the number of simultaneous binary merge events. $M_3 = 1$ is the number of simultaneous ternary merge events. The concatenations of M_0, M_1, M_2, M_3 terms form the *occupancy spectrum* $M = 1121$ that summarizes a coalescence configuration with simultaneous multiple collisions. Such an occupancy spectrum can be randomly generated with Algorithm 1. All possible occupancy spectrum output can be generated using Algorithm 2.

and, as an interface, it allows to consider interchangeable strategies for generating a coalescence configuration.

Algorithm 1 is an on-the-fly occupancy spectrum sampling algorithmm. It takes as an entry the number of balls and urns and by a ball-to-urn random assignment experience: it assigns uniformly at random a parent among N to each one of the k lineages, counting those with same parent for constructing the spectrum.

Algorithm 1 On-the-fly occupancy spectrum sampling algorithm (OTF)

1: **procedure** SAMPLE(n,m)
Require: n number of balls, m number of urns
Ensure: returns an occupancy spectrum.
2: M zero-vector of length $n+1$
3: π zero-vector of length m
4: **for** $k \in [1, n]$ **do**
5: $i \sim unif(0, m-1)$
6: $p_i \leftarrow p_i + 1$
7: **for** $j \in p$ **do**
8: $M_j \leftarrow M_j + 1$
9: $printM_j$

Direct sampling in the occupancy spectrum probability distribution

Instead of repeatedly reconstructing an occupancy spectrum each time a sample is needed, we propose to sample an occupancy spectrum directly in its probability distribution. Generating the support of the distribution is not straightforward and

is expected to be costly, but memoization techniques allow it to be constructed once, sampled many times, what scales well with ABC methods.

Let D_n^m be the joint distribution of the occupancy spectrums arising when throwing n balls in m urns. Interestingly, the expression of D_n^m has been obtained by von Mises (1939):

$$\Pr[\bigcap_{j=0}^{n}(M_j = m_j)] = \frac{m!n!}{m^n \Pi[(j!)^{m_j} m_j!]} \tag{3.1}$$

$$\text{with} \quad \sum_{j=0}^{n} m_j = m \quad \text{(conservation of the number of urns)} \tag{3.2}$$

$$\text{and} \quad \sum_{j=0}^{n} j m_j = n \quad \text{(conservation of the number of balls)} \tag{3.3}$$

This result means that if we had an algorithm able to efficiently generate Ω_n^m, the support of D_n^m, we could directly sample an occupancy spectrum according to its probability instead of entirely reconstruct it from the ball-by-ball urn assignment described in Algorithm 1.

As far as we know, such an algorithm has not been published yet, and we present a version of it in Algorithm 2

3.2.4 Methods

The various components are implemented in C++ as parts of the Quetzal library. The `simultaneous_multiple_merge` algorithm takes as arguments a sequence of lines and the number of parents, and returns the sequence of lines after coalescence events dictated by an occupancy spectrum created in the function internals. The behavior of the algorithm concerning the spectrum creation is controlled by a policy class: by default it invokes the `on_the_fly` policy that is an implementation of Algorithm 1. As policy design allows compile-time customization, the spectrum generator can be set to another policy, included user-defined ones. Quetzal proposes the alternative policy `in_memoized_distribution` that will sample an occupancy spectrum directly in D_n^m, possibly loosing time in constructing D_n^m if the algorithm was never called before with the couple $\{n, m\}$.

Algorithm 2 Ω_k^m generative algorithm

```
1: procedure GENERATE(n,m)
Require: n number of balls, m number of urns
Ensure: for each v printed v ∈ Ω_k^m.
2:     v zero-vector of length n + 1
3:     f(n, m, n,v)
4: function F(n, m, λ, v)
Require: n number of balls, m number of urns, λ largest occupancy number of the
    spectrum, v an updating occupancy spectrum solution
Ensure: for each v printed v ∈ Ω_k^m.
5:     if m = 0 & n = 0 then
6:         print v return
7:     if ! m = 0 then
8:         if n = 0 then
9:             w ← v
10:            w_0 ← m
11:            print w
12:            return
13:        else
14:            if λ > 0 then
15:                i ← ⌊n/λ⌋
16:                while i ⩾ 1 do
17:                    w ← v
18:                    w_λ ← i
19:                    if m ⩾ i then
20:                        b ← n − i ∗ λ
21:                        if b < λ then
22:                            f(b, m − i, b, w)
23:                        else
24:                            f(b, m − i, λ − 1, w)
25:                    i ← i − 1
26:    if λ = 0 & n > 0 then
27:        return
28:    w ← v
29:    w_λ ← 0
30:    f(n, m, λ − 1, w)
```

Truncated spectrum

As the support of the memoized distribution can be huge, Quetzal offers various possibilities to reduce its memory footprint. By default, the length of the occupancy spectrums in Ω_k^m is set to $k+1$ for homogeneity of behavior with the `on_the_fly` policy. However, the last zero-elements of an occupancy spectrum represent an important part of the memory loads of Ω_k^m, in addition to induce useless iterations in the `simultaneous_multiple_merge` algorithm, so they could be removed. For example the occupancy spectrum coded by 631000 (6 urns with 0 balls, 3 urns with 1 ball, and 1 urn with 2 balls) can be summarized by 631. The `RemoveLastEmptyUrns` policy allows to customize the behavior of `in_memoized_distribution` by shortening the size of the generated occupancy spectrums.

Approximated distribution

Many spectrums have a very low probability of occurence. For example, with $n = 5$ and $m = 10$, out of 7 possible spectrums, the occupancy spectrum 900001 (9 urns with 0 balls, 1 urn with 5 balls) has a probability of 10^{-4}. Quetzal proposes a policy to control the behavior of `in_memoized_distribution` by deciding wether or not a spectrum should be kept in memory based on its probability of occurence. The default behavior is set to keep all the spectrums.

Performance comparison

The performance of the `simultaneous_multiple_merge` algorithm is compared under three different strategies for generating the occupancy spectrum. These strategies are respectively OTF (*On The Fly*, implemented by the `on_the_fly` policy), IMD (*Memoized*, implemented by the default settings of the `in_memoized_distribution`) and MAT (Memoized, Approximated, Truncated), a composite policy resulting from sampling the truncated spectrum in a distribution where the spectrums with a probability of occurence less than 10^{-6} have been discarded. The effects of the approximation are not assessed here.

For each condition and each strategy, the execution time of one coalescence generation is sampled and discarded 30 times (warmup), then a 10000-sample of execution times is recorded. The execution time distributions for OTF, IMD and MAT strategies are shown in Figure 3.4 for $n = 6$ and $m = 10$. The distributions for

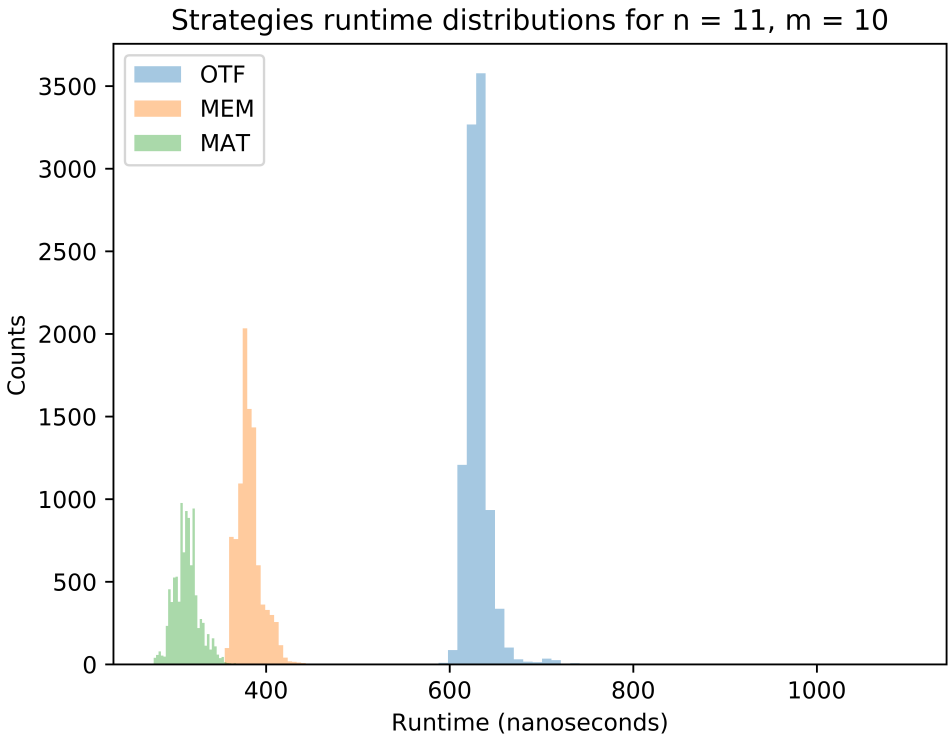


FIGURE 3.4: Distributions of runtimes required by Quetzal algorithms to perform simultaneous multiple collisions of $n = 11$ lines in $m = 10$ parents. Algorithm represented by strategy OTF (in blue) merges the lines conditionally to an occupancy spectrum reconstructed on the fly each time the algorithm is called (see Algorithm 1). MEM (in orange) merges the lines conditionally to a spectrum directly sampled in its exact memoized probability distribution whereas MAT (in green) approximates this distribution by discarding the spectrums with low output probability, and truncates the spectrum to reduce memory loads and avoid useless iterations.

$n \in [\![2\ ;\ 20]\!]$ and $m = 10$ are shown in Figure 3.5 for OTF and MAT strategies. The microbenchmark source code can be found in supplementary material.

3.2.5 Discussion

The memoized versions of the algorithm generating the spectrum (MEM and MAT) are approximately twice faster than the standard procedure (OTF), with less variance for increasing sample sizes. The MAT strategy outperforms MEM for large enough values of n. Similar patterns have been observed for variable values of m (results not shown).

Although our memoized versions of coalescence configuration generation compare favorably with standard ones, care should be taken in the interpretation, as this analysis suffers from the limitations of microbenchmarks, notably concerning

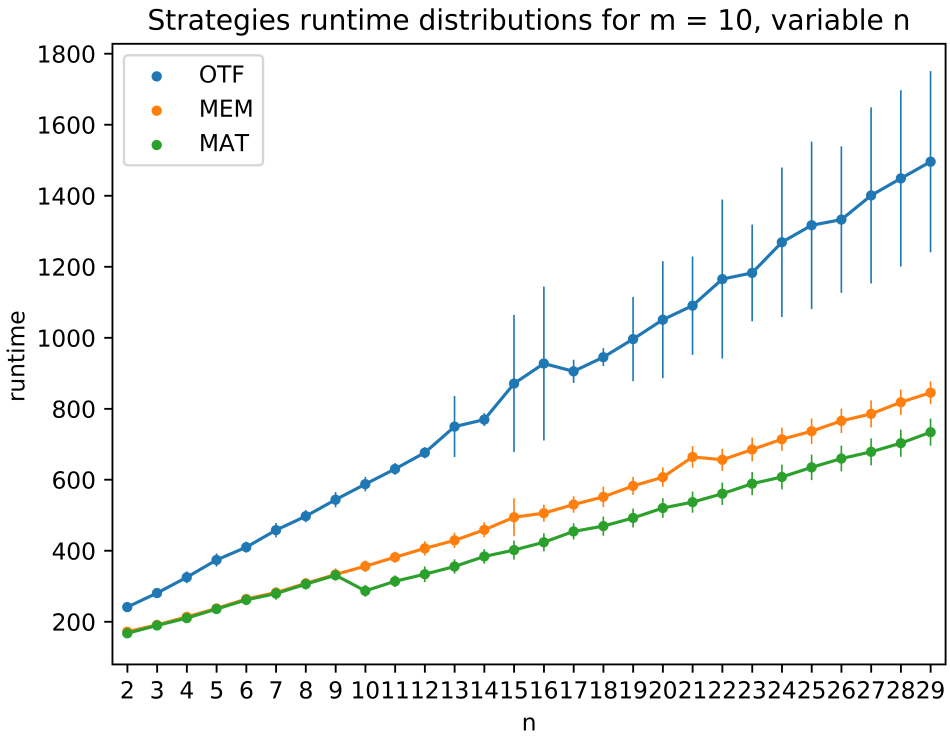


FIGURE 3.5: Mean runtime (in nanoseconds) required by Quetzal algorithms to perform simultaneous multiple collisions of n lines in $m = 10$ parents. The square deviation is represented by vertical bars. See Figure 3.4 for explanations about strategies

reproducibility and extensibility of the conclusions. First, different compilers versions may compile code differently and it may be that the results we show are valid only in our particular hardware and operating system. Furthermore, by definition a microbenchmark does not evaluate the performance of an algorithms in a complex simulation context: in many user-specific contexts we expect cache misses to hinder the performance of the memoized approach. Finally, multithreading contexts are known to be very difficult to evaluate using benchmarks.

Aware of these limitations, we advise the ordinary user of Quetzal to stick with the default settings of the coalescence algorithms, as the on-the-fly procedure is more intuitive, requires no particular C++ policy-based design knowledge, and is susceptible to be satisfying under a vast range of $\{n, m\}$ conditions.

When there is strong control on the domain where $\{n, m\}$ takes value, using memoized approaches can be relevant as a means of last ressort, but it should be carefully assessed that the performance gain under this domain is worth by profiling the code, possibly reusing or adapting the small benchmark C++ and Python libraries available in the supplementary material. Notably, the user should be aware of the effects of cache misses.

In specific contexts, for example when we expect the majority of coalescence events to occur when repeated bottlenecks reduce the population size to a constant N, it may be worth to design a small composite policy class to sample in the memoized distribution if $m = N$, and to generate the occupancy spectrum in all the other cases.

3.2.6 Conclusion

We developed an algorithm able to generate all possible configurations resulting from coalescing n lines uniformly at random in m parents. As the probability of each configuration can be computed, memoization techniques allow to perform random sampling of configurations many times at little cost in coalescence simulators. These methods compare favorably with standard procedures simulating configuration by repeated lines-to-parent assignement, leading to 50% faster simulations in our study. All strategies presented here are implemented as generic components in Quetzal (**Becheler214767**) and are avalaible to be freely reused or combined.

Chapter 4

Using fuzzy partitions for ABC inference of recent demographic processes

This chapter treats the problem of comparing an observed genetic dataset with a simulated forest of incomplete coalescents, a fundamental step in the ABC rejection algorithm for infering parameters of recent demographic processes. First we highlight that if the mutational process can be neglected, the data partition formed by the most recent genealogical fragments (the bottom parts of a coalescent tree) contains all the information needed for inference. Furthermore, the hypothesis of high recombination between loci allows to treat data locus by locus using bayesian prior updating for inference. Then we show that as models can not distinguish gene copies in a same panmictic population unit, it is detrimental no to account for this uncertainty in the data representation. The fuzzy partition formalism allows to express that groups of sampled gene copies indistinguishable by the model belong to different allelic states at some degree. We will finally see that keeping anonymous the simulated clusters is statistically beneficial and that observed and simulated fuzzy partition can be compared computing the fuzzy transfer distance. The fuzzy transfer distance can then be used in the ABC framework, and we show various simulation results validating the approach.

4.1 Introduction

Genetic patterns observed in the field allow to infer demographic processes experienced by the population. One particular case of such demographic processes is

the fast expansion of an initially small population, that is characteristic of biological invasions. As various demographic events leave a specific genetic signature, it enables inference of past processes given present data. A number of inferencial methods focus on computing the likelihood function, what is often not possible under reasonably realistic models. To bypass this difficulty, Approximated Bayesian Computation methods (Beaumont et al. 2002, Marjoram et al. 2003) simulate many datasets to find the parameter values that lead to minimal discrepancy between observation and simulation.

4.1.1 The two main ABC approximations

Let be a statistical parametric model

$$\{f(y_{obs} \mid \theta) : y_{obs} \in \mathscr{Y}, \theta \in \Theta\},\ \mathscr{Y} \subseteq \mathbb{R}^n,\ \Theta \subseteq \mathbb{R}^p,\ p \geq 1,\ n \geq 1\ .$$

Let $p(\theta)$ be the prior distribution on the parameter θ.

Observing the data y_{obs}, Bayesian inference methods aims at determining the posterior distribution giving the probability of the parameters values given y_{obs}, that is $p(\theta \mid y_{obs})$. To bypass the intractability of the likelihood function $p(y \mid \theta)$, ABC methods make two approximations:

Summarizing full data sets using low-dimensional summary statistics

First, the observed data y_{obs} is approximated by a lower-dimensional data $s_{obs} = \eta(y_{obs})$, where $\eta : \mathscr{Y} \to \mathbb{R}^q$ is called a *summary statistics* in the ABC culture, and in practice used to designate both η or s_{obs}. The posterior distribution is then approximated by:

$$p(\theta|y_{obs}) \simeq p(\theta|s_{obs}) \propto p(s_{obs}|\theta)p(\theta).$$

The quality of this approximation depends on how much information is lost by the summary statistics η:

- if s_{obs} is highly informative on θ, then $p(\theta|s_{obs}) \simeq p(\theta|y_{obs})$
- if s_{obs} is sufficient, then $p(\theta|s_{obs}) = p(\theta|y_{obs})$

Ideally, a sufficient statistics should be used to avoid an error in the estimation, but identifying such statistics is known to be a complicated (or impossible) task (Marjoram et al., 2003b).

k-nearest neighbors procedure and kernel density estimation

First we need to present the algorithm 3 introduced by Biau, Cérou, and Guyader (2015). As stated by the authors, it is noteworthy that it is not the algorithm mainly used to present the method (see for example Algorithm 1 and 2 in Marin et al. (2012)), but it reflects how ABC is commonly run by practitioners:

Algorithm 3 ABC sampler

Require: $a \in \mathbb{N}$ and $c1 \leq k_a \leq a$
1: **for** $i = 1$ to a **do**
2: Sample a parameter in the prior: $\theta_i \sim p(\theta)$
3: Simulate a data in the model: $y_i \sim p(y \mid \theta_i)$
return The θ_i's such that $s(y_i)$ is among the k_a-nearest neighbors of $S(y_{obs})$.

In other words, this algorithm performs an *i.i.d* sampling of size a in the prior predictive distribution $p(y \mid \theta)p(\theta)$ (this sample is called the *reference table* in the ABC literature, see e.g. Marin et al. (2016, Algorithm 1)). Then, the k_a simulated couple $\{y_i, \theta_i\}$s with minimal discrepancy to the observation are retained. This discrepancy is evaluated by a distance $\rho : \mathbb{R}^q \to \mathbb{R}$.

Finally, once the k_a-sample T of parameters leading to simulated data nearest to the observation has been returned, a kernel density estimation (see Definition 4.1.1 for a 1-dimensional example) procedure is generally used to estimate the posterior distribution (Biau, Cérou, and Guyader, 2015). The value of the posterior distribution density at a point θ_x is given by:

$$p_{ABC}(\theta_x) = \frac{1}{k_a h} \sum_{j=1}^{k_a} K(\frac{\theta_x - T_j}{h_a}) \, ,$$

where $K : \mathbb{R}^p \to [0,1]^p$ is a standard kernel and h is a vector of positive real numbers (bandwidth).

Definition 4.1.1. (Kernel density estimation) Non-parametric method that estimates the density function of a random variable, extrapolating a sample to a density. Given a sample $x_1, x_2, \ldots, x_n$, the density of the population distribution can be approximated by:

$$\hat{p}_h(x) = \frac{1}{nh} \sum_{i=1}^{n} K(\frac{x - x_i}{h}) \, ,$$

where K is a probability kernel and h a smoothing parameter. Figure 4.1 illustrates the method to estimate the density of the probability distribution from which 6 data were sampled, using a gaussian kernel.

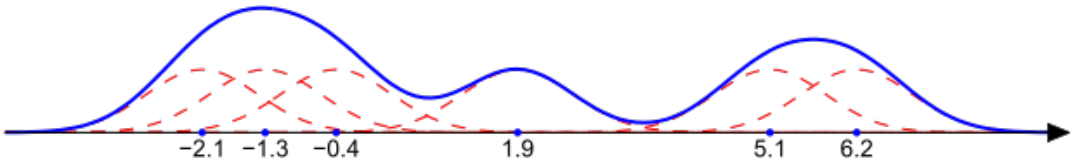


FIGURE 4.1: Illustration of a kernel density estimation (see Definition 4.1.1). Gaussian kernels (red, variance set at 0.5) and their sum (blue). The density is estimated by averaging over kernels. (Oleg Alexandrov, 12 January 2005)

4.1.2 Approximations consequences

If the kernel density estimation step is performed in a high-dimensional space (that is with the number of model parameters p large), counter-intuitive problems arise and threaten the posterior estimation reliability. Problems arising in high dimension are known as *curse of dimensionality*: see *e.g* Aggarwal, Hinneburg, and Keim (2001) for surprising behaviors of distances in high dimensional spaces and Domingos (2012) for examples of counter-intuitive phenomenons in high-dimensional machine-learning problems.

Intuitively, a key problem of high dimensions is that data are *diluted* along dimensions: 10 points sampled in $[0,1]$ are not too far away from each other, but the space between them will grow if they are dispersed in the $[0,1]^3$ cube. Due to data spareness, the kernel density estimator converges more slowly towards the target distribution as dimensionality increases. That is, exponentially more data will are needed if one wants to achieve good accuracy. For example Scott (2008, Section 7.2) shows that to estimate a simple Gaussian distribution using a kernel density estimator, 10^6 observations are necessary in 10 dimensions to conserve the same accuracy than in 1 dimension where estimation was performed with 50 observations only.

In the ABC framework previously presented, curse of dimensionality involves that if the number of model parameters is too high, posterior density estimation will not be reliable unless a tremendous computational effort is made to simulate an exponentialy-growing number of nearest neighbors T. Consequently, this is a real motivation for the modeler to keep p as small as possible.

At the same time, it is worth to study the possibility to design a sufficient statistics, knowing its dimension q does not have to be dramatically reduced relatively to the initial data dimension, as the distance ρ can be defined on a high-dimensional space.

4.1.3 Related decisions in the modeling step

Usually in ABC coalescent studies of complex demographic processes, the model parameters inference is made possible by finding the sampled genes most recent common ancestor (MRCA) through the reconstruction of their genealogy. Doing so, the modeler has to account for an history that can be far more ancient than the processes of interest: the MRCA can indeed be found in a remote spatio-temporal window, leading to a number of problems.

First the quality and the quantity of available information drop when going backward in time: it will result difficult to reliably inform the model, whether for model specification (prior distributions, model hypothesis) or data (availability and consistency).

Second the rejection rate of ABC will unnecessarily increase since the remote-time topology of the coalescent strongly conditions the data likelihood: a genealogy which bottom topology is consistent with observations can be rejected if its top topological properties are inconsistent with the observations. This is a shame because respectively to the recent history process, it should have been accepted.

To avoid the simulation of the top genealogy, a solution would be to randomly draw the allelic state of ancestors before the MRCA is found. This is not satisfying as generally the prior distribution of the states is unknown, with little information available in the literature. Furthermore, infering the ancestral allelic states would increase the dimensionality of the model.

The idea developed here is to build on the very recent genealogical clustering process to assess its consistency with the observed genetic clustering. Whenever the non-mutation hypothesis holds, it allows to conserve the anonymous nature of the allelic states, as they are conserved along the bottom branches fragments. Doing so, the ancient history and the ancestral allelic states do not have to be inferred anymore, and the inference can be done without summary statistics using an orignal distance.

4.2 Material and methods

4.2.1 Justifying the non mutation hypothesis

For invasion processes, the total number of generations since invasion (beginning of the process) is generally known exactly or approximately. This specific fact allows,

for recent invasions, to neglect mutation, as is explained now.

In the classical Wright-Fisher model with mutation, mutations are assessed to occur with constant probability μ at each offspring creation.

Let X be the number of mutations occurring when coalescing n_g lines at a given generation g. X is distributed as a binomial with parameters the number of lines n_g (the number of Bernoulli experiments) and the mutation probability μ (the probability of success). The probability that there are k mutations occurring at a given generation g is then

$$P(X = k) = \binom{n_g}{k} \cdot \mu^k (1-\mu)^{n_g - k}.$$

Conditionally to a discrete-time coalescent tree $\mathcal{T}$ (or equivalently a forest of random coalescent trees $\mathcal{F}$), the number of mutations along this tree $X_{\mathcal{T}}$ is then the sum of the number of mutations at each generation X_g. As mutations are assumed neutral in the Wright-Fisher model, mutations across generations are independent, so all random variables X_g are independent. Mutation rate is assumed constant through generations.

Assuming $Y_1 \hookrightarrow B(n_1, p)$ and $Y_2 \hookrightarrow B(n_2, p)$, when Y_1 and Y_2 are independent, $Y_1 + Y_2 \hookrightarrow B(n_1 + n_2, p)$.

It follows that:

$$X_{\mathcal{T}} \hookrightarrow B(\sum_g n_g, \mu)$$

and the expected number of mutations is

$$E(X_{\mathcal{T}}) = \mu . \sum_g n_g$$

with variance

$$Var(X_{\mathcal{T}}) = \mu(1-\mu) \sum_g n_g$$

Without conditioning on the coalescent tree, the quantities n_g (the number of lines at each generation), are random variables. Under complex models their distribution is unknown, and so are the quantities $E(X_{\mathcal{T}})$ and $Var(X_{\mathcal{T}})$.

However, as a *worst* case argument, the total branches length of $\mathcal{T}$ is maximal if the n_0 lines do not coalesce during the G generations, or equivalently if they coalesce

at the last generation $G-1$. In that case, $X_{\mathcal{T}} \hookrightarrow B(Gn_0, \mu)$ with $E(X_{\mathcal{T}}) = Gn_0\mu$ and $Var(X_{\mathcal{T}}) = Gn_0\mu(1-\mu)$.

For microsatellite loci, $\mu \simeq 10^{-3}$. For a sample of size $n_0 = 100$ coalescing over 10 generations (what is approximately the Asian hornet dataset setup), we expect in the worst-tree case only one mutation. So we can reasonably assume that the mutation process is negligible compared to the genealogical process in shaping the dataset.

4.2.2 Hard partitions

Let us consider a finite set S of elements. We will be interested in this section in grouping these elements in different subsets of S (called *clusters*). To bring structure and information to a set of elements, an intuitive idea will be to group together elements that share common features.

The Köppen–Geiger climate classification system will be used through the first part of the chapter to illustrate important concepts about partitions, as it has intuitive motivations. Later in the chapter, examples will follow with the less-intuitive study of genetic polymorphism distribution through space (and time).

The Köppen–Geiger climate classification system partitions coordinates of the geographic space according to bioclimatic features. For example, Paris belongs to the *Cfb* cluster, that designates a *temperate hot climate without dry season and with temperate summer*. Looking at the map of climates (Figure 4.2), regions and their associated features make much more sense than the underlying raw data list. In this sense, clustering can lead to very intuitive representations of data. Consequently the Köppen system is widely used across multiple disciplines (such as hydrology, agronomy, climatology, biology).

To group together the cities of the set $\{Seattle, Chicago, Houston, Portland\}$ that share the same clusters in the Köppen system, the following partition notation is largely used (but it loses the groups name information):

$$\{\{Chicago\}, \{Seattle, Portland\}, \{Houston\}\}.$$

This defines a hard partition of the cities set into 3 subsets (clusters).

Definition 4.2.1. (Hard partition) A hard partition $P_U = \{P_{U^1}, P_{U^2}, \ldots, P_{U^c}\}$ of a set $\mathcal{S}$ is a set of nonempty subsets and disjoint of S covering S:

- $P_{U^i} \neq \emptyset \forall i \in \{1, 2, \ldots, c\}$ (subsets are nonempty)

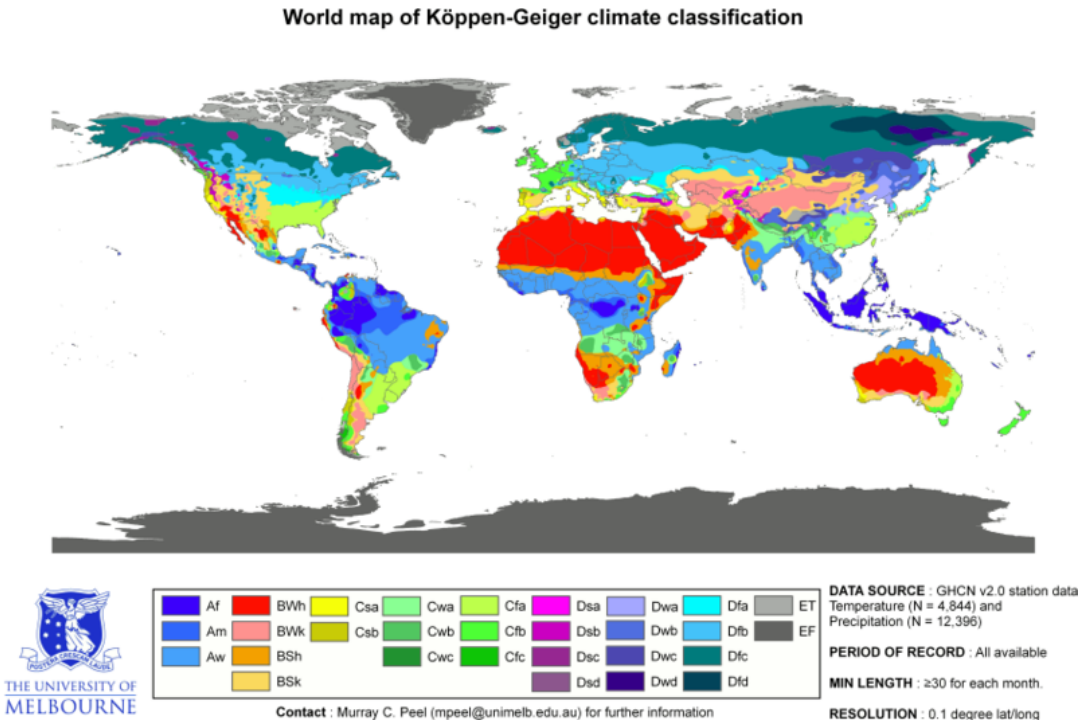


FIGURE 4.2: The Köppen–Geiger climate classification system assigns geographic coordinates to clusters according to local bioclimatic conditions (Peel, Finlayson, and McMahon, 2007). It defines a partition of the geographic space elements in bioclimatic clusters.

- $P_{U^i} \cap P_{U^j} = \forall i, j \in \{1, 2, \ldots, c\}$ (subsets are pairwise disjoint)
- $\bigcap_{i=1}^{c} P_{U^i} = S$ (subsets cover S)

We will use the term of c-partition to indicate that a hard partition is constituted of c clusters. If considering a set S of λ elements $S = \bigcup_i u_i$ belonging to c clusters, a hard partition of S in c clusters can be described in terms of a *hard partition matrix* U of size $c \times \lambda$

$$\mathcal{U} = \begin{array}{c} \\ U^1 \\ U^2 \\ \vdots \\ U^i \\ \vdots \\ U^c \end{array} \begin{array}{c} \begin{array}{cccccc} u_1 & u_2 & \ldots & u_k & \ldots & u_\lambda \end{array} \\ \begin{pmatrix} b_{11} & b_{12} & \ldots & b_{1k} & \ldots & b_{1\lambda} \\ b_{21} & b_{22} & \ldots & b_{2k} & \ldots & b_{2\lambda} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ b_{i1} & b_{i2} & \ldots & b_{ik} & \ldots & b_{i\lambda} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ b_{c1} & b_{c2} & \ldots & b_{ck} & \ldots & b_{c\lambda} \end{pmatrix} \end{array}$$

with the general term b_{ik} being equal to 1 if $u_k \in U^i$, 0 if not. For example, the following hard partition matrix gives the attribution of some cities to the Köppen

climatic clusters:

$$\begin{array}{c} \\ Dfa \\ Csb \\ Cfa \end{array} \begin{array}{c} \begin{array}{cccc} Chicago & Seattle & Portland & Houston \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \end{array}$$

4.2.3 Fuzzy partitions

Hard partitions formalism allows to assign each element at exactly one cluster. However, in many cases, objects can not be partitioned in well-delimited clusters as it is not clear if an element belongs to one cluster or another, because of data imprecision or uncertainty, or because of the very problem definition.

For example, the climatic features of New York city make it possibly belong to both climatic clusters *Cfa* or *Dfa*: the concept of *hot and humid continental climate* that defines *Dfa* is obviously not an absolute, and it is at some point difficult to clearly separate it from a *subtropical humid climate* (*Cfa*). As seen in Chapter 2, this lack of precision is not necessarily a problem, as this kind of abstraction gives space for intuition, generalization and communication: *belongingness of an object in a collection is a matter of degree* (Bodjanova, 2000).

Consequently, different mathematical theories were developed, aiming at incorporating flexibility in representing this kind of data. Zadeh (1965) came with a mathematical description of *belongingness*, or *membership*, introduced in the theory of fuzzy sets (see Definition 4.2.2).

Fuzzy sets

Definition 4.2.2. (Fuzzy set, Zadeh (1965)) A fuzzy set $\mathcal{A}$ in S is defined by a membership function $f_A : S \to [0,1]$ that associates each element of S to a real number in $[0,1]$ representing the *grade of membership* of the corresponding element in $\mathcal{A}$. Value 1 represents full membership of the element to $\mathcal{A}$, value 0 represents nonmembership to $\mathcal{A}$. A fuzzy set is empty if its membership function is null on S. In fuzzy set theory, classical sets are called *crisp sets*.

Consequently, the definition of a hard partition matrix can be extended, and a more general form of matrix can be used to allow elements to partially belong to fuzzy clusters. Among other spaces, Bezdek (1981) defines the space of the fuzzy

partitions as the set of partitions where the membership coefficients of each column belong to $[0,1]$ and sum to 1:

$$\mathbb{M}_{fcn} = \{U \in \mathbb{R}^{c\ n}, u_{ik} \in [0,1], \sum_{i=1}^{c} u_{ik} = 1\}.$$

Each of the λ columns u_λ gives the vector of the membership coefficient indicating to which degree element λ belongs to the each of the c clusters of P_U. For example, the following fuzzy partition represents the fact that New York can possibly belong to two climatic regions:

$$\begin{array}{c} \\ Dfa \\ Csb \\ Cfa \end{array} \begin{array}{c} \begin{array}{ccccc} Chicago & Seattle & Portland & Houston & NewYork \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 0 & 0.5 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0.5 \end{pmatrix} \end{array}$$

The set of the hard partitions $\mathbb{M}_{hcn}$ is included in $\mathbb{M}_{fcn}$ and designates the fuzzy partitions with binary membership coefficients:

$$\mathbb{M}_{hcn} = \{U \in \mathbb{M}_{fcn}, u_{ik} \in \{0,1\}\}.$$

Fuzzy inclusion

Set inclusion is an important structural description of data, providing the basis for comparing two partitions. In fuzzy theory, the first mathematical definition was given by Zadeh (1965) as a first attempt to extend the inclusion relationship to fuzzy sets, see Definition 4.2.3.

Definition 4.2.3. (Zadeh's inclusion, Beg and Ashraf (2012)) Let $F(S)$ be the set of all fuzzy subsets of set S. For all $A, B \in F(S)$, A is said to be a subset of B if for all $u \in S, f_A(u) \leq f_B(u)$, where $f_A(u)$ and $f_B(u)$ represent the membership grades of element u in A and B respectively. In this case, we write $A \subseteq B$ and call it the Zadeh's inclusion. Two fuzzy sets A and B are said to be equal if and only if $A(u) = B(u) \forall u \in S$.

Subcase 1 Ordinary (nonfuzzy) set inclusion is very direct: all elements of V are elements of U, so $V \subseteq U$ (Figure 4.3a). Pursuing with the climatic classification example, if a climatic set V is defined by *"locations where annual temperature mean is less*

than 10°C", then it delimits a geographic region fully enclosed into the climatic set U defined by *"locations where annual temperature mean is less than* 11°C": then obviously $V \subseteq U$. Importantly, inclusion intuition holds even if sets are not constructed using the same rule system, *e.g.* U' may be defined in terms of a soil quality criteria and may still define a region that is included in V, even if V is itself defined in terms of temperature conditions.

Subcase 2 Relaxing the membership bivalency on the set V is intuitively not problematic. This situation can arise for example if a climatic set U has been categorically defined, but that the classification system remains quite elusive on defining another set V, causing an uncertainty about the degree to which coordinates of S belongs to V. But if the Zadeh's inclusion definition holds, that is if $f_V(u) \leq f_U(u)\ \forall u \in S$, as it is the case in Figure 4.3b, then the inclusion of V in U seems natural (if it still seems counter-intuitive to the reader, then considering the non-inclusion case shown in Figure 4.3c helps much in making an intuition of what Zadeh's subsethood is, and is not).

Remark. Several authors (Bandler and Kohout, 1993, *e.g.*) stated that as Zadeh'inclusion made binary decision about being a subset or not, it was too dichotomous for a fuzzy theory concept, and that a more fuzzy description of subsethood would be relevant. Consequently important efforts were made to propose a measure of the degree to which a subset is included in another (see *e.g.* Beg and Ashraf, 2009; Beg and Ashraf, 2012).

Subcase 3 Considering how much a fuzzy set is included in another fuzzy set seems at first sight rather counter-intuitive, because we use to formalize sets in a nonfuzzy way. Intuition comes back when remembering that fuzzy theory aims precisely at formalizing vagueness: the counter-intuitive Zadeh's condition can be understood by progressively switching from Figure 4.3c to Figure 4.3d relaxing full and non membership assumptions first to make cluster U a little fuzzier, then to make V a *reasonable* subset of U. Zadeh's inclusion definition captures an important structural property of subsets, even if it is subject to criticism exposed in the previous remark. For example, considering a climate classification system that remains elusive in all clusters definitions (for example using only natural vocabulary such as *hot*, *cold*, *humid* ... to define climatic regions). Then it should not impede a map to be

drawn, even if partial and imprecise, using transparency to represent membership degrees to which each coordinate belongs to a climatic group. Zadeh's inclusion can help in limiting the number of groups in the classification system, by deciding if the information captured by V is captured more efficiently by U's definition. If it is worth to reduce redundancy, V should be removed from the classification system, as U is sufficient.

Refinement

Refinement is an important structural property of partitions, closely related to the inclusion definition.

Definition 4.2.4. (Refinement) A c-partition U is a refinement of a r-partition V if and only if:

- $c \geq r$,
- $\forall i \in \{1, 2, \ldots, c\}, \exists j / U^i \subseteq V^j$

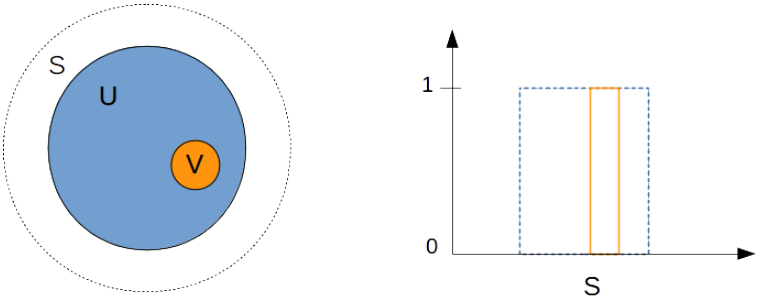
U is said to be finer than V (V is rawer that U), and this relationship is denoted $U \subseteq V$

In the following hard partition example, U_h is finer than V_h:

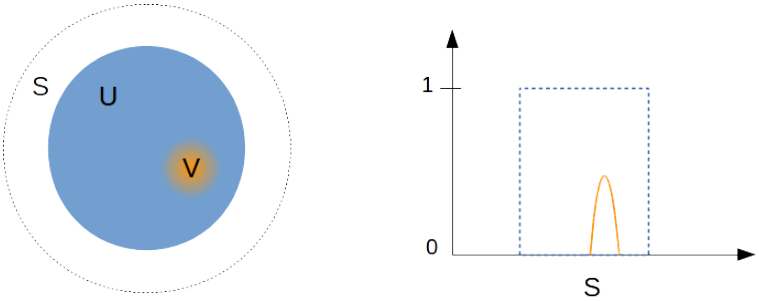
$$\mathcal{U}_h = \begin{array}{c} \begin{matrix} u_1 & u_2 & u_3 & u_4 \end{matrix} \\ \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \end{array} \qquad \mathcal{V}_h = \begin{array}{c} \begin{matrix} u_1 & u_2 & u_3 & u_4 \end{matrix} \\ \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{pmatrix} \end{array}$$

The following example illustrates the refinement concept with two fuzzy partitions, where U_f is finer than V_f:

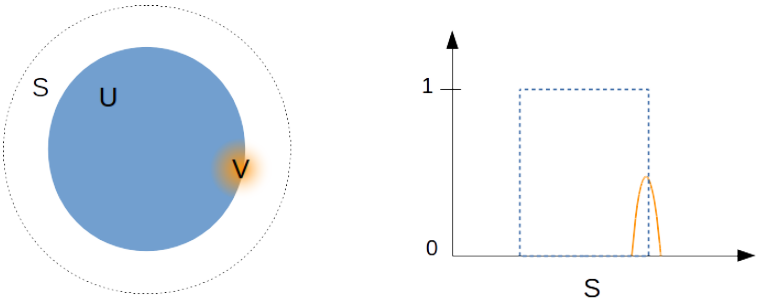
$$\mathcal{U}_f = \begin{array}{c} \begin{matrix} u_1 & u_2 & u_3 & u_4 \end{matrix} \\ \begin{pmatrix} 0.1 & 0.8 & 0.7 & 0.3 \\ 0.9 & 0.1 & 0.2 & 0.2 \\ 0 & 0.1 & 0.1 & 0.5 \end{pmatrix} \end{array} \qquad \mathcal{V}_f = \begin{array}{c} \begin{matrix} u_1 & u_2 & u_3 & u_4 \end{matrix} \\ \begin{pmatrix} 1 & 0.9 & 0.9 & 0.5 \\ 0 & 0.1 & 0.1 & 0.5 \end{pmatrix} \end{array}$$



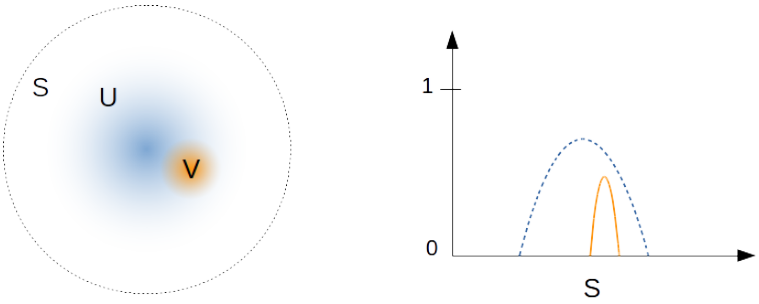
(A) Crisp sets inclusion



(B) Inclusion of a fuzzy set into a crisp set



(C) Non inclusion



(D) Fuzzy sets inclusion

FIGURE 4.3: Simplified examples of various degrees of set inclusion (or subsethood). Left: graphical representation of a set S partitioned in two subsets U (blue) and V (yellow) with $V \subset U$. Right: corresponding (schematized) membership functions f_U (blue) and f_V (yellow) associated to the clusters. Subfigure 4.3a: a crisp set is included in another crisp set. Subfigure 4.3b: a fuzzy set is included in a crisp set. Subfigure 4.3c: a fuzzy set is not included in a crisp set. Subfigure 4.3d: a fuzzy set is included in another fuzzy set.

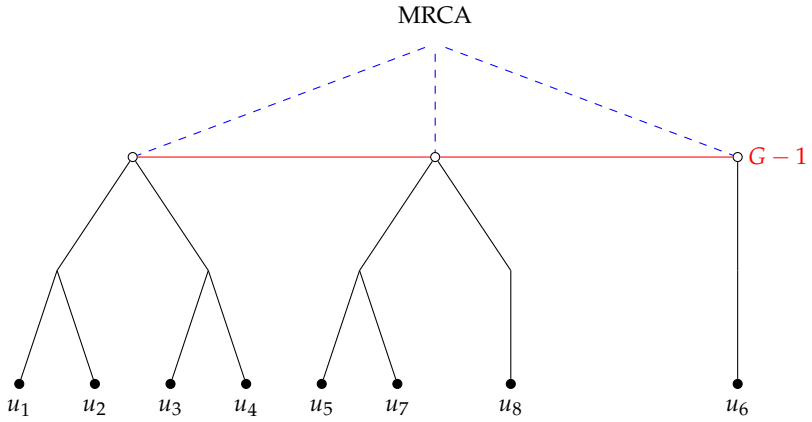


FIGURE 4.4: Coalescence of a forest of hanging subtrees during G generations: unlike traditional coalescence approaches, the biological invasion context allows the coalescence process to be stopped at generation $G-1$, possibly before the most recent common ancestor of the sampled genes copies has been found. The hanging subtrees define a partition of the sampled gene copies: $\{\{x_1, x_2, x_3, x_4\}\{x_5, x_7, x_8\}\{x_6\}\}$

4.2.4 The genealogical partitioning process

Let be S the set of n gene copies (nodes) sampled at generation 0. A forest of random trees is built backward in time from generation 0 to generation $G-1$ by the coalescence process (see Figure 4.4).

Let define the equivalence relationship $\sim$ on S *«are leaves of a same hanging subtree»*. At generation g, the equivalence classes define a n_g-partition P_{G-1} of S, where n_g is the number of hanging subtrees at generation g.

The following hard partition describes the clusters of nodes $\{u_1, \cdots, u_8\}$ formed by the genealogical hanging subtrees in Figure 4.4:

$$\begin{array}{cccccccc} u_1 & u_2 & u_3 & u_4 & u_5 & u_6 & u_7 & u_8 \\ \left(\begin{array}{cccccccc} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{array}\right) \end{array}$$

The observed genetic data define a partition based on an allelic state clustering of the gene copies, while the coalescence process gives a genealogical-based clustering of them. Under the hypothesis that the mutational process can be neglected, the leaves of a same hanging subtree share the same allelic state due to the absence of mutations. Furthermore, two (or more) hanging subtrees can share the same allelic state if their ancestors share same allelic state. In terms of partition structure, it

means that P_g is a refinement of the observed partition. In terms of ABC inference, it means that considering an observed partition P_{obs}, all simulated genealogical partitions P_g refining P_{obs} should be accepted, and that all partitions that do not should (ideally) be rejected.

For example, considering the genealogical process in Figure 4.4 as the *true* genealogy and let's say that the ancestor of $\{u_1, u_2, u_3, u_4\}$ and the ancestor of u_6 have an arbitrary allelic state A, while the ancestor of $\{u_5, u_7, u_8\}$ is a: when looking only at allelic states, two genealogical clusters are aggregated. Then the true genealogical partition $\{\{u_1, u_2, u_3, u_4\}\{u_5, u_7, u_8\}\{u_6\}\}$ is a refinement of the observed partition $P_{obs} = \{\{u_1, u_2, u_3, u_4, u_6\}\{u_5, u_7, u_8\}\}$. Given only P_{obs}, it will not be possible to infere which of its refinements is the true one.

4.2.5 Why fuzzy partitions formalism is useful in the coalescence framework

The example in Figure 4.4 was intentionally simple and leads to crisp partition of data. However, in real-world dataset, various sources of uncertainty appear, making difficult, if not impossible, to affect objects to well-defined clusters. When looking at a gene copy in our dataset, there are two main sources of uncertainty, for which using fuzzy partition to represent and compare data could be useful.

Unphased data

If the dataset is unphased, that is the haplotype is unknown, there is no information about which one of the chromosome holds the allele. Consequently, when genotyping for example a diploid sample and representing it under a table form, any pairwise permutation of the alleles at a given locus actually encodes the same information. This is typically one of many implicit rules that humans (but not computers) abstract away as undesirable properties of the data structure. Unfortunately, at some point in the programming activity, implicit rules have to be made explicit: we expose here how the fuzzy partition formalism cab adresses this point.

Consider the previous example, where the observed partition of gene copies is $P_{obs} = \{\{u_1, u_2, u_3, u_4, u_6\}\{u_5, u_7, u_8\}\}$, and the true genealogical partition underlying the observations is $P_g = \{\{u_1, u_2, u_3, u_4\}\{u_5, u_7, u_8\}\{u_6\}\}$. We mention only now that the sampled diploid individuals were the pairs $\{u_1, u_2\} \dots \{u_7, u_8\}$.

As the two alleles of a same individual can be permuted, the memberships of u_5 and u_6 can be exchanged:

$$P_g = \{\{u_1, u_2, u_3, u_4\}\{u_5, u_7, u_8\}\{u_6\}\} \equiv \{\{u_1, u_2, u_3, u_4\}\{u_6, u_7, u_8\}\{u_5\}\} \equiv P_{obs}$$

In terms of ABC inference, it means that these two genealogical partitions are equivalent regarding P_{obs} and that none can not be rejected without introducing bias in the posterior estimate. Detecting efficiently this form of equivalency is then important. When looking at very few individuals, testing all permutations is not problematic, but it scales poorly with the data size: if d is the ploidy and n the number of individuals genotyped at l loci, they are d^{nl} ways to represent the same information using a a hard partition encoding. Using fuzzy partitions is a way to represent this phase uncertainty. The following fuzzy partition describes the clusters of nodes $\{u_1, \ldots, u_8\}$ formed by the genealogical hanging subtrees in Figure 4.4:

$$\mathcal{A} \begin{pmatrix} \{u_1, u_2\} & \{u_3, u_4\} & \{u_5, u_6\} & \{u_7, u_8\} \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 0.5 & 1 \\ 0 & 0 & 0.5 & 0 \end{pmatrix}$$

Note that A is a refinement of the fuzzy representation of P_{obs}:

$$P_{obs} \equiv \begin{pmatrix} \{u_1, u_2\} & \{u_3, u_4\} & \{u_5, u_6\} & \{u_7, u_8\} \\ 1 & 1 & 0.5 & 0 \\ 0 & 0 & 0.5 & 1 \end{pmatrix}$$

The demic structure hypothesis

When analyzing spatial (or spatio-temporal) samples of genetic material with discrete-space populations models, the gene copies sampled inside a same deme are not distinguished by the model. Then it will be difficult to assess how much a simulated spatial coalescence forest of hanging subtrees agrees with an observed spatial dataset. Using fuzzy partitions to incorporate the model demic structure uncertainty in the data representation avoids brute-force approaches.

4.2.6 Constructing the fuzzy partitions: examples

Observed fuzzy partition

In Figure 4.5, Instead of representing the fact that a gene copy belongs to an allelic state cluster with probability 1, a fuzzy partition is used to represent the degree to which the various groups of gene copies belong to each allelic clusters. In the spatial sampling framework, these groups are defined in terms of a *same location*. The term *same location* can refer to the exact sampling geographic coordinates or to the same panmictic population unit defined by the model. In this case, the fuzzy partition can be seen as a representation of the spatial distribution of allelic frequencies.

Coordinates	A1	A2
x_1	10	10
x_2	20	12
x_3	20	10
x_3	10	12

$\equiv$

	x_1	x_2	x_3
	0	0.5	0.25
	1	0	0.5
	0	0.5	0.25

FIGURE 4.5: Observed genotypes at one locus for 4 individuals at 3 different geographic locations x_1, x_2, x_3. The traditional dataframe representation (left) bears the semantics that the gene copies are distinguishable and each belongs with probability 1 to an allelic state cluster (color), what is actually inexact in the model framework. Using a fuzzy partition (right) allows to confound the gene copies sampled in a same deme that are undistinguishable, and to compute the membership coefficients giving the degree to which groups of undistinguishable gene copies belong to each allelic state cluster.

Simulated fuzzy partitions

Here, we consider the equivalence relationship $\sim$ defined at section 4.2.4. The genealogical process defines equivalence classes. Instead of representing the fact that a singular gene copy belongs to a hanging subtree with probability 1, it is more relevant to give the degree to which each group of undistinguishable gene copies belong to a subtree. So a spatial coalescent can be represent ed using a fuzzy partition, where nodes in a same deme are confounded (see Figure 4.6). As some ancestral gene copies can have same allelic state, various clusters of the partition can be merged (this is equivalent to adding the corresponding rows of the partition matrix).

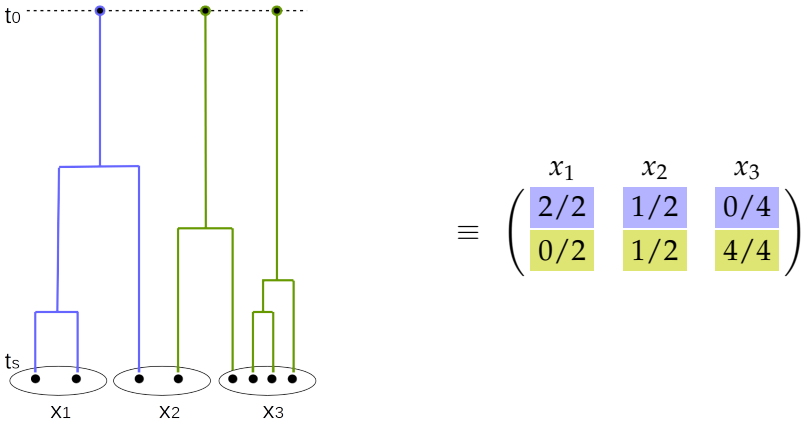


FIGURE 4.6: Simulated anonymous genotypes at one locus for 4 individuals at 3 different geographic locations. Using a fuzzy partition allows to confound the gene copies sampled in a same deme. Membership coefficients give the degree to which demes belong to each allelic state cluster (colors). As mutation is neglected, allelic states do not have to be explicitly simulated.

4.2.7 Comparing simulated and observed fuzzy partitions

The Fuzzy Transfer Distance

Campello (2010) defines $FTD(U,V)$ the *fuzzy transfer distance* between two c-partitions of λ elements U and V by extending the definition of the transfer distance described for hard partitions by Charon et al. (2006):

Definition 4.2.5. (Fuzzy Transfer Distance)

$$FTD(U,V) = \min_{T\in\Gamma} \sum_{ij} \mid V_{ij} - (TU)_{ij} \mid ,$$

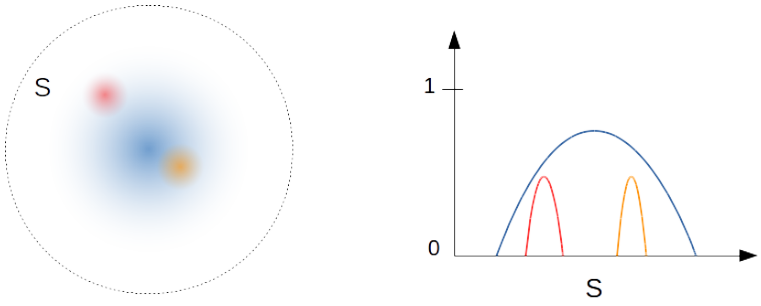
where

$$\Gamma = \{ \ (T_{ij})_{\substack{1\leqslant i\leqslant c \\ 1\leqslant j\leqslant \lambda}} \mid T_{ij} \in \{0,1\} \ , \ \sum_j T_{ij} = 1 \ , \ \sum_i T_{ij} = 1 \ \}$$

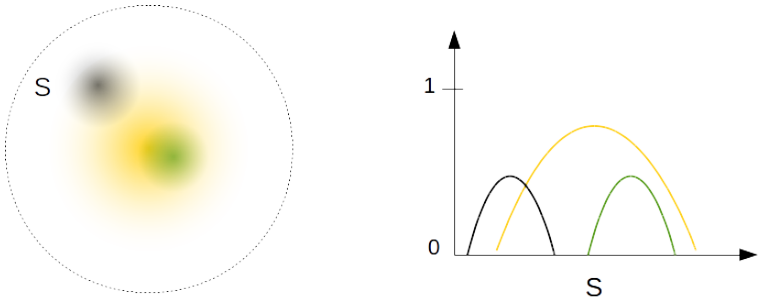
is the set of permutation matrices.

$FTD(U,V)$ describes the minimal quantity of membership coefficients that has to be added and/or removed from the U columns to make them equal to the columns of V, up to a permutation of the rows.

In terms of fuzzy set theory, it allows to find the best way to assign U clusters to V clusters such that the discrepancy between each pair of membership functions is minimal (Figure 4.7).



(A) Fuzzy partition U of S



(B) Fuzzy partition V of S

FIGURE 4.7: Simplified example of Fuzzy Transfer Distance application. Left: graphical representation of a fuzzy partition of a set S in 3 clusters. Right: corresponding (schematized) membership functions associated to each clusters. Computing the Fuzzy Transfer Distance automatically assigns respectively the blue, red, orange clusters of partition U (subfigure 4.7a) to the yellow, black, green cluster of V (subfigure 4.7b) and returns a value indicating how similar are these two partitions.

In terms of allelic distribution and inference, the FTD gives a way to measure the similarity between an observed partition U based on explicit allelic state, and a simulated partition V that does not explicitly defines clusters in terms of allelic state. That is, there is no need to simulate the $\{A, T, G, C\}$ values for a SNP marker, or to simulate the number of repetitions of a microsatellite marker: the FTD automatically finds the best way to compare a coalescent-based clustering to an allelic-based clustering.

Implementation

A bipartite graph between U and V clusters is constructed to find the best assignement between clusters of U and V (Figure 4.8). Each edge is valuated by

$$w_{ij} = \sum_{k=1}^{n} |u_{ik} - v_{jk}|.$$

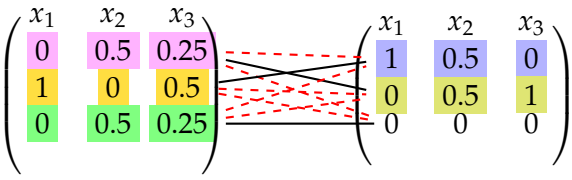


FIGURE 4.8: Application example of the Kuhn-Munkres algorithm: a bipartite graph is constructed for finding a best assignment (dark lines) between clusters (colors) of observed (left) and simulated (right) fuzzy partitions. Edges are valuated by $w_{ij} = \sum_{k=1}^{n} |u_{ik} - v_{jk}|$ (not shown). Minimal cost defines the fuzzy transfer distance (here FTD=2.25) Campello, 2010.

The Kuhn-Munkres algorithm, also called Hungarian algorithm (Kuhn, 1955; Munkres, 1957) allows to find the perfect matching of minimal weight in polynomial time. The minimal weight (dark edges in Figure 4.8) defines the FTD (Campello, 2010).

4.2.8 Method validation

Model

To test the validity of this approach, a first step is to check that it allows sound inference of parameters under a first simple model. We will here focus on estimating the recent history of a population for which we can only assess that the Wright-Fisher hypothesis holds for the last g generations before sampling. That is, the population history older than the last g generations will remain unknown. Mutational process is neglected. The population size parameter N is estimated using ABC approach with FTD taking as data for inference a pseudo-observed (pod) fuzzy partition made of k sampled gene copies, which clustering was simulated by coalescence under a well-known population size N_r:

$$y_o \sim p(y|N_r)$$

The aim is to correctly re-estimate the true parameter value N_r.

Sampling in the predictive prior distribution

A reference table is constituted:

- Parameter is sampled in the prior distribution : $N' \sim p(N)$
- n fuzzy partitions are simulated under the model: $y' \sim p(y|N')$

- The fuzzy transfer distance $FTD(y', y_o)$ is computed.
- The couple $\{N', FTD(y', y_o)\}$ is memorized in the table.

Rejection step

According to the FTD distribution properties, a threshold ϵ is chosen for the ABC rejection step: only those couples (y', θ') for which $FTD(y', y_o) \leqslant \epsilon$ are used for estimating the posterior distribution using kernel density estimation. It is expected that for decreasing ϵ values, the posterior mean converges towards the true value of the parameter N, that is N_r.

Numerical application

Figure 4.9 illustrates this validation test run with $N_r = 200$, $n = 10^5$, $g = 50$, $k = 50$ and for various values of ϵ taken as deciles of the computed FTD values distribution. Prior distribution $p(N)$ is a uniform distribution $U[1, 10000]$.

Posterior stability

It is important to check that the posterior estimation quality is stable across various pseudo-observed data. Since the model is stochastic, we expect the estimation quality to vary among pods. Some pods can be very likely under a parameter value θ and very unlikely under all other parameter values: this leads to very accurate estimation. However, most of the time things are not that extreme, and some generated pods can be not very typical of the true parameter value, leading to some error in the estimation. In Figure 4.10, multiple pseudo-observed data were generated under the model, with $N_{true} = 200$, $n = 10^5$, $g = 50$, $k = 50$ and ϵ taken as deciles of the computed FTD values distribution. Prior distribution $p(N)$ is a uniform distribution $U[1, 10000]$.

4.3 Results

The FTD method has been implemented in Quetzal and will part of the next release after publication. A `Fuzzifier` class allows to transform a genetic dataset into a

`FuzzyPartition`. The `fuzzy_transfer_distance` member function allows to compute FTD between two `FuzzyPartition` objects. The following unit testing small program illustrates how to use the class:

```
#include "FuzzyPartition.h"
#include "RestrictedGrowthString.h"

#include <map>
#include <vector>
#include "assert.h"

// Defining useful type aliases
using membership_function_t = std::vector<double>;
using element_t = std:string;
using coefs_t = std::map<element_type, membership_function_t>;

int main(){

  // Elements a, b, c belong to 4 clusters to various degrees:
  coefs_type c = { {"a",{0.0, 0.1, 0.9, 0.0}},
                   {"b",{0.4, 0.1, 0.2, 0.3}},
                   {"c",{0.0, 0.3, 0.6, 0.1}}
                   };

  // Class with nicer interface
  FuzzyPartition<element_type> A(c);
  assert(A.nElements() == 3);
  assert(A.nClusters() == 4);

  auto B = A;

  // Encodes the partition {{1,2}{3}{4}}
  RestrictedGrowthString RGS({0,0,1,2});

  // Merge clusters 1 and 2 according to RGS
```

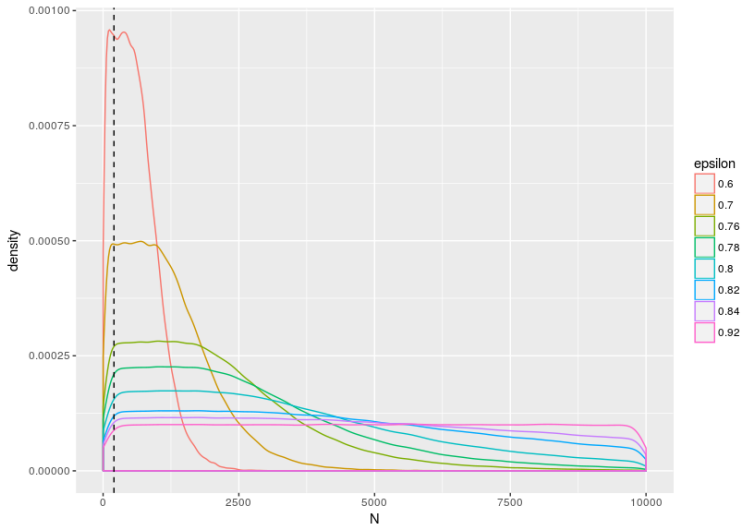


FIGURE 4.9: Posterior densities for various ϵ taken as the computed FTD distribution deciles (1 pseudo-observed data, 10^5 simulations). True population size ($N = 200$) is indicated by the vertical dashed line.

```
  B.merge_clusters(RGS);
  assert(B == FuzzyPartition<int>( { {"a",{0.1, 0.9, 0.0}},
                                     {"b",{0.5, 0.2, 0.3}},
                                     {"c", {0.3, 0.6, 0.1}}
                                   }) );

  double d = A.fuzzy_transfer_distance(B);
  assert(d == 0.4);
  return 0;
}
```

These classes allowed us to design small programs to test the statistical quality of the FTD methodology. Figure 4.9 shows that more stringent rejections (smaller ϵ values) lead to smaller error on the parameter estimate (that is, the mode of the distribution is nearer from the true parameter value). Figure 4.10 shows that the estimation seems stable across multiple pseudo-observed data, with posteriors picked around the true parameter value.

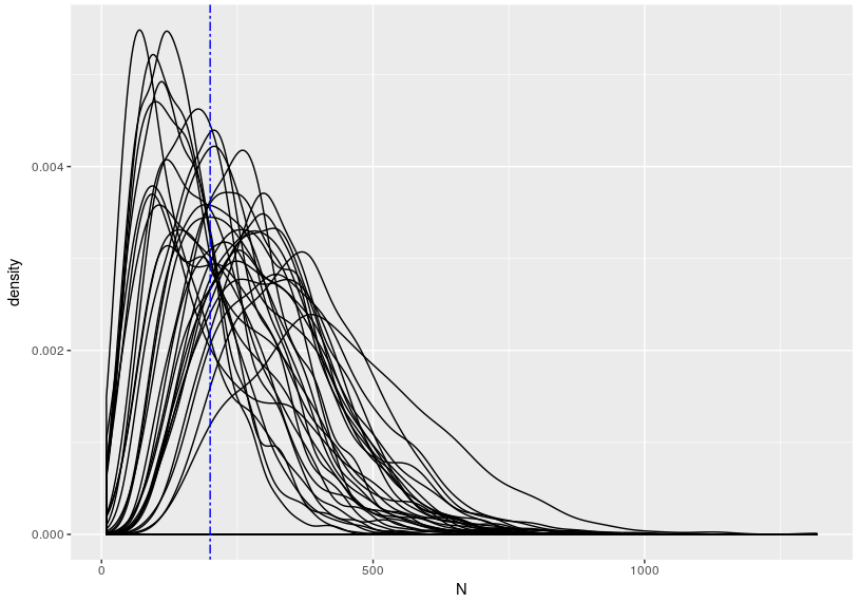


FIGURE 4.10: Posterior densities for 30 pseudo-observed data, with ϵ taken as the first percentile of the FTD distribution (10^5 simulations). The true population size ($N_{true} = 200$) is indicated by the vertical dashed line.

4.4 Discussion

We presented a set of hypothesis that are particularly relevant under a biological invasion context when the recent demographic processes only are the point of interest:

- natural selection is neglected
- mutation can be neglected
- loci are independent (infinite recombination)

The non-mutation hypothesis has been justified, and the independence of loci is currently assumed in microsatellite data set analysis.

Together, these hypothesis enable ABC inference without summary statistics, using the FTD, a function derived from the fuzzy set theory that computes a distance between simulated and observed objects in a highly multi-dimensional space. Each multi-dimensional object is a fuzzy partition that represents the data clustering.

The observed clustering is based on allelic state features and can be considered as a representation equivalent to the full allelic frequencies distribution (frequencies of gene copies with same allelic state in the same deme). The full allelic distribution has already been used to perform ABC inference without summary statistics (Sousa et

al., 2009), but other types of distance was used, and the simulation had to explicitly simulate allelic states, what is not the case of the original approach presented here.

A key point of using the FTD is that it allows to spare hypothesis and/or parameters, while keeping high acceptance rate, by allowing the simulated clustering to be defined in terms of anonymous allelic state. Importantly, the distribution of allelic states at the beginning of the invasion does not need to be infered (reduced number of parameters), and the remote history of the population does not need to be specified (reduced number of hypothesis).

Of course the representation of a genetic data set under the form of a fuzzy partition could be considered as a sufficient summary statistics, but as Sousa et al. (2009), we prefer to use the *"without summary statistics"* terminology to clearly distinguish this approach from the others.

Results of the method validity test (Figure 4.9 and 4.10) show that we can reasonably trust the FTD methodology to estimate very recent demographic processes features. Although this conclusion is for now restricted to the quite simplistic demonstration model used here (a Wright-Fisher population), it is expected that spatial versions will be at least as powerful as traditional methods based on summary statistics. The main reason why more complex models were not tested is that it requires more important code efforts and that extrapolating the method validity across different models is by no means straightforward: it should be kept in mind that the method validity testing is a key part of any ABC analysis that should be assessed each time a new model setup is required.

An important extension of the method is its applicability to the analysis of spatio-temporal sampling schemes: elements (columns) of the partition would not geographic coordinates anymore, but rather spatio-temporal coordinates. Values are still allelic frequencies (implicit or explicit) at each spatio-temporal coordinate. Consequently it allows to make full use of genetic data sets, accounting for temporal dynamics of the allelic state distribution. This will be very useful in the application of the FTD methodology to the analysis of the *Vespa velutina* microsatellite dataset, in which diploid individuals were sampled at various locations at different times.

Although the hypothesis of independent loci is common, traditional ABC analysis use multilocus statistics to summarize data over all loci, possibly inflating the rejection rate. Sousa et al. (2009) proposed to permute loci to find a *best match* between observation and simulation to increase acceptance rate of ABC. Although relevant, it

leads to complicated code design, and brute-force approaches in combinatorial problems is prone to combinatorial explosions making the method to scale poorly with the number of studied loci. Furthermore it is not clear how the FTD computed across multiple loci should be defined. We note that if loci are assumed independent, then data chunks at each of the l loci can be considered as l *i.i.d* observations of the same process. This seems suitable to the use of Sequential Bayesian Updating (SBU) (see *e.g.* Oravecz, Huentelman, and Vandekerckhove, 2016) to sequentially update the knowledge about parameters by analyzing each new locus: posteriors distributions obtained at locus i could used as prior distributions for analyzing locus $i + 1$.

Bibliography

Aggarwal, Charu C, Alexander Hinneburg, and Daniel A Keim (2001). "On the surprising behavior of distance metrics in high dimensional space". In: *International conference on database theory*. Springer, pp. 420–434.

Alexandrescu, Andrei (2001). *Modern C++ design: generic programming and design patterns applied*. Addison-Wesley.

Algeo, TJ, SE Scheckler, and JB Maynard (2001). "Effects of early vascular land plants on weathering processes and global chemical fluxes during the Middle and Late Devonian". In: *Plants Invade the Land: Evolutionary and Environmental Perspectives. Columbia University Press, New York*, pp. 213–236.

Arca, M. et al. (Aug. 2015). "Reconstructing the invasion and the demographic history of the yellow-legged hornet, Vespa velutina, in Europe". en. In: *Biological Invasions* 17.8, pp. 2357–2371. ISSN: 1387-3547, 1573-1464. DOI: 10.1007/s10530-015-0880-9. URL: http://link.springer.com/10.1007/s10530-015-0880-9 (visited on 06/08/2017).

Bandler, Wyllis and Ladislav Kohout (1993). "Fuzzy power sets and fuzzy implication operators". In: *Readings in Fuzzy Sets for Intelligent Systems*. Elsevier, pp. 88–96.

Barbet-Massin, Morgane et al. (2013). "Climate change increases the risk of invasion by the yellow-legged hornet". In: *Biological Conservation* 157, pp. 4–10.

Beaumont, Mark A. (Dec. 2010). "Approximate Bayesian Computation in Evolution and Ecology". en. In: *Annual Review of Ecology, Evolution, and Systematics* 41.1, pp. 379–406. ISSN: 1543-592X, 1545-2069. DOI: 10.1146/annurev-ecolsys-102209-144621. URL: http://www.annualreviews.org/doi/10.1146/annurev-ecolsys-102209-144621 (visited on 03/02/2017).

Beaumont, Mark A et al. (2009). "Adaptive approximate Bayesian computation". In: *Biometrika* 96.4, pp. 983–990.

Becheler, A. (2017). *Quetzal*. https://github.com/Becheler/quetzal. [Online; accessed 28-September-2017].

Beerli, Peter and Joseph Felsenstein (1999). "Maximum-Likelihood Estimation of Migration Rates and Effective Population Numbers in Two Populations Using a Coalescent Approach". In: *Genetics* 152.2, pp. 763–773. ISSN: 0016-6731. URL: http://www.genetics.org/content/152/2/763.

Beg, Ismat and Samina Ashraf (2009). "Fuzzy Inclusion And Fuzzy Similarity With Gödel Fuzzy Implicator". In: *New Mathematics and Natural Computation* 5.03, pp. 617–633.

— (2012). "Fuzzy inclusion and design of measure of fuzzy inclusion". In: *RIMAI J* 8.

Bertolino, Sandro et al. (2016). "Spread of the invasive yellow-legged hornet Vespa velutina (Hymenoptera: Vespidae) in Italy". In: *Applied entomology and zoology* 51.4, pp. 589–597.

Bertorelle, G., A. Benazzo, and S. Mona (June 2010). "ABC as a flexible framework to estimate demography over space and time: some cons, many pros: the ABC revolution in nine steps". en. In: *Molecular Ecology* 19.13, pp. 2609–2625. ISSN: 09621083, 1365294X. DOI: 10.1111/j.1365-294X.2010.04690.x. URL: http://doi.wiley.com/10.1111/j.1365-294X.2010.04690.x (visited on 02/26/2017).

Bezdek, James C (1981). "Objective Function Clustering". In: *Pattern recognition with fuzzy objective function algorithms*. Springer, pp. 43–93.

Biau, Gérard, Frédéric Cérou, Arnaud Guyader, et al. (2015). "New insights into approximate Bayesian computation". In: *Annales de l'Institut Henri Poincaré, Probabilités et Statistiques*. Vol. 51. 1. Institut Henri Poincaré, pp. 376–403.

Blois, Jessica L et al. (2013). "Climate change and the past, present, and future of biotic interactions". In: *Science* 341.6145, pp. 499–504.

Blum, M. G. B. et al. (May 2013). "A Comparative Review of Dimension Reduction Methods in Approximate Bayesian Computation". en. In: *Statistical Science* 28.2, pp. 189–208. ISSN: 0883-4237. DOI: 10.1214/12-STS406. URL: http://projecteuclid.org/euclid.ss/1369147911 (visited on 08/24/2017).

Bodjanova, Slavka (2000). *Fuzzy Partitions*.

Bommarco, Riccardo, David Kleijn, and Simon G Potts (2013). "Ecological intensification: harnessing ecosystem services for food security". In: *Trends in ecology & evolution* 28.4, pp. 230–238.

Brown, Jason L. and L. Lacey Knowles (Aug. 2012). "Spatially explicit models of dynamic histories: examination of the genetic consequences of Pleistocene glaciation and recent climate change on the American Pika: SPATIALLY EXPLICIT MODELS OF DYNAMIC HISTORIES". en. In: *Molecular Ecology* 21.15, pp. 3757–3775. ISSN: 09621083. DOI: 10.1111/j.1365-294X.2012.05640.x. URL: http://doi.wiley.com/10.1111/j.1365-294X.2012.05640.x (visited on 08/23/2017).

Bullock, James M et al. (2011). "Restoration of ecosystem services and biodiversity: conflicts and opportunities". In: *Trends in ecology & evolution* 26.10, pp. 541–549.

Campello, R.J.G.B. (July 2010). "Generalized external indexes for comparing data partitions with overlapping categories". en. In: *Pattern Recognition Letters* 31.9, pp. 966–975. ISSN: 01678655. DOI: 10.1016/j.patrec.2010.01.002. URL: http://linkinghub.elsevier.com/retrieve/pii/S0167865510000048 (visited on 02/28/2017).

Charon, Irene et al. (2006). "Maximum transfer distance between partitions". In: *Journal of classification* 23.1, pp. 103–121.

Chomsky, Noam (1957). "Syntactic structures." In:

Church, John A and Neil J White (2006). "A 20th century acceleration in global sea-level rise". In: *Geophysical research letters* 33.1.

Colautti, Robert I and David M Richardson (2009). "Subjectivity and flexibility in invasion terminology: too much of a good thing?" In: *Biological Invasions* 11.6, pp. 1225–1229.

Colburn, Timothy and Gary Shute (2007). "Abstraction in computer science". In: *Minds and Machines* 17.2, pp. 169–184.

Cornuet, Jean-Marie et al. (Apr. 2014). "DIYABC v2.0: a software to make approximate Bayesian computation inferences about population history using single nucleotide polymorphism, DNA sequence and microsatellite data". en. In: *Bioinformatics* 30.8, pp. 1187–1189. ISSN: 1460-2059, 1367-4803. DOI: 10.1093/bioinformatics/btt763. URL: https://academic.oup.com/bioinformatics/article-lookup/doi/10.1093/bioinformatics/btt763 (visited on 08/24/2017).

cppreference.com (2018). *Library Concepts*. [Online; accessed 27-January-2018]. URL: \url{http://en.cppreference.com/w/cpp/concept}.

Csilléry, Katalin et al. (July 2010). "Approximate Bayesian Computation (ABC) in practice". en. In: *Trends in Ecology & Evolution* 25.7, pp. 410–418. ISSN: 01695347.

DOI: 10.1016/j.tree.2010.04.001. URL: http://linkinghub.elsevier.com/retrieve/pii/S0169534710000662 (visited on 08/24/2017).

Currat, M., N. Ray, and L. Excoffier (Jan. 2004a). "splatche: a program to simulate genetic diversity taking into account environmental heterogeneity: program note". en. In: *Molecular Ecology Notes* 4.1, pp. 139–142. ISSN: 14718278, 14718286. DOI: 10.1046/j.1471-8286.2003.00582.x. URL: http://doi.wiley.com/10.1046/j.1471-8286.2003.00582.x (visited on 08/23/2017).

— (Jan. 2004b). "splatche: a program to simulate genetic diversity taking into account environmental heterogeneity: PROGRAM NOTE". en. In: *Molecular Ecology Notes* 4.1, pp. 139–142. ISSN: 14718278, 14718286. DOI: 10.1046/j.1471-8286.2003.00582.x. URL: http://doi.wiley.com/10.1046/j.1471-8286.2003.00582.x (visited on 08/24/2017).

De Groot, Rudolf et al. (2012). "Global estimates of the value of ecosystems and their services in monetary units". In: *Ecosystem services* 1.1, pp. 50–61.

Del Moral, Pierre, Arnaud Doucet, and Ajay Jasra (2006). "Sequential monte carlo samplers". In: *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 68.3, pp. 411–436. URL: http://onlinelibrary.wiley.com/doi/10.1111/j.1467-9868.2006.00553.x/full (visited on 09/06/2017).

Dıaz, Sandra et al. (2005). "Biodiversity regulation of ecosystem services". In: *Trends and conditions*, pp. 279–329.

Díaz, Sandra et al. (2006). "Biodiversity loss threatens human well-being". In: *PLoS biology* 4.8, e277.

Dirzo, Rodolfo and Peter H Raven (2003). "Global state of biodiversity and loss". In: *Annual Review of Environment and Resources* 28.1, pp. 137–167.

Domingos, Pedro (2012). "A few useful things to know about machine learning". In: *Communications of the ACM* 55.10, pp. 78–87.

Escobar, Arturo (1998). "Whose knowledge, whose nature? Biodiversity, conservation, and the political ecology of social movements". In: *Journal of political ecology* 5.1, pp. 53–82.

Estoup, Arnaud and Thomas Guillemaud (2010). "Reconstructing routes of invasion using genetic data: why, how and so what?" In: *Molecular ecology* 19.19, pp. 4113–4130.

Estoup, Arnaud et al. (Sept. 2010a). "Combining genetic, historical and geographical data to reconstruct the dynamics of bioinvasions: application to the cane toad

Bufo marinus: RECONSTRUCTING BIOINVASION DYNAMICS". en. In: *Molecular Ecology Resources* 10.5, pp. 886–901. ISSN: 1755098X. DOI: 10.1111/j.1755-0998.2010.02882.x. URL: http://doi.wiley.com/10.1111/j.1755-0998.2010.02882.x (visited on 08/23/2017).

— (Sept. 2010b). "Combining genetic, historical and geographical data to reconstruct the dynamics of bioinvasions: application to the cane toad Bufo marinus: reconstructing bionvasion dynamics". en. In: *Molecular Ecology Resources* 10.5, pp. 886–901. ISSN: 1755098X. DOI: 10.1111/j.1755-0998.2010.02882.x. URL: http://doi.wiley.com/10.1111/j.1755-0998.2010.02882.x (visited on 08/23/2017).

Ewing, Gregory and Joachim Hermisson (Aug. 2010). "MSMS: a coalescent simulation program including recombination, demographic structure and selection at a single locus". en. In: *Bioinformatics* 26.16, pp. 2064–2065. ISSN: 1460-2059, 1367-4803. DOI: 10.1093/bioinformatics/btq322. URL: https://academic.oup.com/bioinformatics/article-lookup/doi/10.1093/bioinformatics/btq322 (visited on 08/24/2017).

Excoffier, Laurent, Matthieu Foll, and Rémy J. Petit (Dec. 2009). "Genetic Consequences of Range Expansions". en. In: *Annual Review of Ecology, Evolution, and Systematics* 40.1, pp. 481–501. ISSN: 1543-592X, 1545-2069. DOI: 10.1146/annurev.ecolsys.39.110707.173414. URL: http://www.annualreviews.org/doi/10.1146/annurev.ecolsys.39.110707.173414 (visited on 05/14/2017).

Fisher, Brendan, R Kerry Turner, and Paul Morling (2009). "Defining and classifying ecosystem services for decision making". In: *Ecological economics* 68.3, pp. 643–653.

Fisher, Ronald A (1923). "XXI.—On the Dominance Ratio". In: *Proceedings of the royal society of Edinburgh* 42, pp. 321–341.

Fog, Agner (2012). "Optimizing software in C++ - An optimization guide for Windows, Linux and Mac platforms". In: *Copenhagen University College of Engineering-2012-168 p*.

Fordham, Damien A et al. (2014). "Better forecasts of range dynamics using genetic data". In: *Trends in Ecology & Evolution* 29.8, pp. 436–443.

Frison, Emile A, Jeremy Cherfas, and Toby Hodgkin (2011). "Agricultural biodiversity is essential for a sustainable improvement in food and nutrition security". In: *Sustainability* 3.1, pp. 238–253.

GDAL Development Team (2017). *GDAL - Geospatial Data Abstraction Library, Version 2.2.1*. [Online; accessed 28-September-2017]. Open Source Geospatial Foundation.

GeoTools (2018). *Axis Order — GeoTools*. [Online; accessed 27-January-2018]. URL: \url{http://docs.geotools.org/latest/userguide/library/referencing/order.html}.

Grosso-Silva, José Manuel and Miguel Maia (2012). "Vespa velutina Lepeletier, 1836 (Hymenoptera, Vespidae), new species for Portugal." In: *Arquivos entomolóxicos* 6, pp. 53–54.

He, Qixin, Danielle L. Edwards, and L. Lacey Knowles (Dec. 2013a). "Integrative testing of how environments from the past to the present shape genetic structure across landscapes". en. In: *Evolution* 67.12, pp. 3386–3402. ISSN: 00143820. DOI: 10.1111/evo.12159. URL: http://doi.wiley.com/10.1111/evo.12159 (visited on 08/23/2017).

— (Dec. 2013b). "Integrative testing of how environments from the past to the present shape genetic structure across landscapes". en. In: *Evolution* 67.12, pp. 3386–3402. ISSN: 00143820. DOI: 10.1111/evo.12159. URL: http://doi.wiley.com/10.1111/evo.12159 (visited on 03/22/2017).

Hein, J., M. Schierup, and C. Wiuf (2004). *Gene Genealogies, Variation and Evolution: A primer in coalescent theory*. Oxford University Press, USA. ISBN: 978-0-19-154615-0. URL: https://books.google.fr/books?id=QBC_SFOamksC.

Hooper, David U et al. (2005). "Effects of biodiversity on ecosystem functioning: a consensus of current knowledge". In: *Ecological monographs* 75.1, pp. 3–35.

Howden, S Mark et al. (2007). "Adapting agriculture to climate change". In: *Proceedings of the national academy of sciences* 104.50, pp. 19691–19696.

Keeling, Matt J et al. (2017). "Predicting the spread of the Asian hornet (Vespa velutina) following its incursion into Great Britain". In: *Scientific reports* 7.1, p. 6240.

Kingman, John Frank Charles (1982). "The coalescent". In: *Stochastic processes and their applications* 13.3, pp. 235–248.

Kremen, Claire et al. (2007). "Pollination and other ecosystem services produced by mobile organisms: a conceptual framework for the effects of land-use change". In: *Ecology letters* 10.4, pp. 299–314.

Kuhn, Harold W (1955). "The Hungarian method for the assignment problem". In: *Naval Research Logistics (NRL)* 2.1-2, pp. 83–97.

Kuhner, Mary K., Jon Yamato, and Joseph Felsenstein (2000). "Maximum likelihood estimation of recombination rates from population data". In: *Genetics* 156.3, pp. 1393–1401. URL: http://www.genetics.org/content/156/3/1393.full-text.pdf+html (visited on 08/24/2017).

Lacey Knowles, L. and Diego F. Alvarado-Serrano (Sept. 2010). "Exploring the population genetic consequences of the colonization process with spatio-temporally explicit models: insights from coupled ecological, demographic and genetic models in montane grasshoppers: geneitc consequence of distribution shifts". en. In: *Molecular Ecology* 19.17, pp. 3727–3745. ISSN: 09621083. DOI: 10.1111/j.1365-294X.2010.04702.x. URL: http://doi.wiley.com/10.1111/j.1365-294X.2010.04702.x (visited on 08/23/2017).

Leblois, RaphaëL, Arnaud Estoup, and FrançOis Rousset (Jan. 2009). "IBDSim: a computer program to simulate genotypic data under isolation by distance". en. In: *Molecular Ecology Resources* 9.1, pp. 107–109. ISSN: 1755098X, 17550998. DOI: 10.1111/j.1755-0998.2008.02417.x. URL: http://doi.wiley.com/10.1111/j.1755-0998.2008.02417.x (visited on 08/24/2017).

Letunic, Ivica and Peer Bork (2006). "Interactive Tree Of Life (iTOL): an online tool for phylogenetic tree display and annotation". In: *Bioinformatics* 23.1, pp. 127–128.

Liskov, Barbara H and Jeannette M Wing (1994). "A behavioral notion of subtyping". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16.6, pp. 1811–1841.

López, S, M González, and A Goldarazena (2011). "Vespa velutina lepeletier, 1836 (Hymenoptera: Vespidae): first records in Iberian Peninsula". In: *EPPO Bulletin* 41.3, pp. 439–441.

Marin, Jean-Michel et al. (2012). "Approximate Bayesian computational methods". In: *Statistics and Computing* 22.6, pp. 1167–1180. URL: http://link.springer.com/article/10.1007/s11222-011-9288-2 (visited on 03/02/2017).

Marin, Jean-Michel et al. (2016). "ABC random forests for Bayesian parameter inference". In: *arXiv preprint arXiv:1605.05537*.

Marjoram, Paul et al. (2003a). "Markov chain Monte Carlo without likelihoods". In: *Proceedings of the National Academy of Sciences* 100.26, pp. 15324–15328.

Marjoram, Paul et al. (2003b). "Markov chain Monte Carlo without likelihoods". In: *Proceedings of the National Academy of Sciences* 100.26, pp. 15324–15328.

Marques, Ambre (2017). *expressive*. https://github.com/ambre-m/expressive. [Online; accessed 28-September-2017].

Marske, Katharine Ann, Carsten Rahbek, and David Nogués-Bravo (2013). "Phylogeography: spanning the ecology-evolution continuum". In: *Ecography* 36.11, pp. 1169–1181.

Martin, Robert C (1996a). "The dependency inversion principle". In: *C++ Report* 8.6, pp. 61–66.

— (1996b). "The interface segregation principle: One of the many principles of OOD". In: *C++ Report* 8, pp. 30–36.

— (1996c). "The Liskov substitution principle". In: *C++ Report* 8.3, p. 14.

Martin, Robert C. (2000). "Design principles and design patterns". In: *Object Mentor* 1.34. URL: http://staff.cs.utu.fi/staff/jouni.smed/doos_06/material/DesignPrinciplesAndPatterns.pdf (visited on 08/27/2017).

Martin, Robert C (2002a). *Agile software development: principles, patterns, and practices*. Prentice Hall.

Martin, Robert C (2002b). *Agile software development: principles, patterns, and practices*. Prentice Hall.

Martin, Robert Cecil (1996d). "The Open-Closed Principle". In: Archived August 22, 2006, at the Wayback Machine. URL: \url{https://drive.google.com/file/d/0BwhCYaYDn8EgN2M5MTkwM2EtNWFkZC00ZTI3LWFjZTUtNTFhZGZiYmUzODc1/view}.

— (2009). *Getting a SOLID start - Clean Coder*. Website. [Online; accessed 28-January-2018]. URL: \url{https://sites.google.com/site/unclebobconsultingllc/getting-a-solid-start}.

Massatti, Rob and L. Lacey Knowles (Aug. 2016). "Contrasting support for alternative models of genomic variation based on microhabitat preference: species-specific effects of climate change in alpine sedges". en. In: *Molecular Ecology* 25.16, pp. 3974–3986. ISSN: 09621083. DOI: 10.1111/mec.13735. URL: http://doi.wiley.com/10.1111/mec.13735 (visited on 08/24/2017).

Matson, Pamela A et al. (1997). "Agricultural intensification and ecosystem properties". In: *Science* 277.5325, pp. 504–509.

McRae, Brad H and Paul Beier (2007). "Circuit theory predicts gene flow in plant and animal populations". In: *Proceedings of the National Academy of Sciences* 104.50, pp. 19885–19890.

Meyer, Bertrand (1988). "Object oriented software construction". In:

Montano, Valeria (2016). "Coalescent inferences in conservation genetics: should the exception become the rule?" In: *Biology letters* 12.6, p. 20160211.

Montgomery, David R (2007). "Soil erosion and agricultural sustainability". In: *Proceedings of the National Academy of Sciences* 104.33, pp. 13268–13272.

Munkres, James (1957). "Algorithms for the assignment and transportation problems". In: *Journal of the society for industrial and applied mathematics* 5.1, pp. 32–38.

Nathan, Ran et al. (2012). "Dispersal kernels". In: *Dispersal Ecol Evol*, pp. 187–210.

Nicholls, Robert J and Anny Cazenave (2010). "Sea-level rise and its impact on coastal zones". In: *science* 328.5985, pp. 1517–1520.

Nielsen, SE et al. (2007). "A new method to estimate species and biodiversity intactness using empirically derived reference conditions". In: *Biological Conservation* 137.3, pp. 403–414.

Nordborg, Magnus (2001). "Coalescent theory". In: *Handbook of statistical genetics*. URL: http://onlinelibrary.wiley.com/doi/10.1002/0470022620.bbc21/full (visited on 08/24/2017).

Oleg Alexandrov, Wikipedia contributor (12 January 2005). *Parzen window illustration — Wikipedia, The Free Encyclopedia*. [Online; accessed 27-February-2018]. URL: \url{https://fr.wikipedia.org/wiki/Estimation_par_noyau#/media/File:Parzen_window_illustration.svg}.

Olsen, G. (1990). *Gary Olsen's Interpretation of the "Newick's 8:45" Tree Format Standard*. http://evolution.genetics.washington.edu/phylip/newick_doc.html. [Online; accessed 28-September-2017].

Oravecz, Zita, Matt Huentelman, and Joachim Vandekerckhove (2016). "Sequential Bayesian updating for big data". In: *Big Data in Cognitive Science*, p. 13. URL: https://books.google.com/books?hl=en&lr=&id=qDclDwAAQBAJ&oi=fnd&pg=PA13&dq=%22velocity,+volume,+and+variety+of+big+data+present+both+challenges+and+opportunities+for+cognitive%22+%22in+which+crowd-sourced+data+are+used+to+study+Alzheimer%E2%80%99s+Dementia.+We+%EF%AC%81t+an+extended%22+&ots=a7NRcVsvvn&sig=5wLBPNiXFQmw6SfV6SIdfU_5R64 (visited on 06/09/2017).

Pachauri, Rajendra K et al. (2014). *Climate change 2014: synthesis report. Contribution of Working Groups I, II and III to the fifth assessment report of the Intergovernmental Panel on Climate Change*. IPCC.

Pahl-Wostl, Claudia (2007). "Transitions towards adaptive management of water facing climate and global change". In: *Water resources management* 21.1, pp. 49–62.

Parhami, Behrooz (1999). *Computer arithmetic*. Vol. 20. 00. Oxford university press.

Pauls, Steffen U. et al. (Feb. 2013). "The impact of global climate change on genetic diversity within populations and species". en. In: *Molecular Ecology* 22.4, pp. 925–946. ISSN: 09621083. DOI: 10.1111/mec.12152. URL: http://doi.wiley.com/10.1111/mec.12152 (visited on 08/23/2017).

Peel, Murray C, Brian L Finlayson, and Thomas A McMahon (2007). "Updated world map of the Köppen-Geiger climate classification". In: *Hydrology and earth system sciences discussions* 4.2, pp. 439–473.

Pereira, Henrique M et al. (2010). "Scenarios for global biodiversity in the 21st century". In: *Science* 330.6010, pp. 1496–1501.

Perrard, Adrien et al. (2014). "Geographic variation of melanisation patterns in a hornet species: genetic differences, climatic pressures or aposematic constraints?" In: *PloS one* 9.4, e94162.

Pritchard, J. K. et al. (Dec. 1999). "Population growth of human Y chromosomes: a study of Y chromosome microsatellites". en. In: *Molecular Biology and Evolution* 16.12, pp. 1791–1798. ISSN: 0737-4038, 1537-1719. DOI: 10.1093/oxfordjournals.molbev.a026091. URL: https://academic.oup.com/mbe/article-lookup/doi/10.1093/oxfordjournals.molbev.a026091 (visited on 08/31/2017).

Publishing, Springer (2018). *Ecosystems - Springer*. Website. [Online; accessed 27-January-2018]. URL: \url{https://link.springer.com/journal/10021}.

Robinet, Christelle, Christelle Suppo, and Eric Darrouzet (2017). "Rapid spread of the invasive yellow-legged hornet in France: the role of human-mediated dispersal and the effects of control measures". In: *Journal of Applied Ecology* 54.1, pp. 205–215.

Rome, Quentin et al. (2013). "Spread of the invasive hornet Vespa velutina Lepeletier, 1836, in Europe in 2012 (Hym., Vespidae)". In: *Bulletin de la Société entomologique de France* 118.1, pp. 21–22.

Rubin, DB (1984). "Bayesianly justifiable and relevant frequency calculations for the applied statistician". In: *Annals of statistics* 12.4, pp. 1151–1172.

Schlenker, Wolfram and David B Lobell (2010). "Robust negative impacts of climate change on African agriculture". In: *Environmental Research Letters* 5.1, p. 014010.

Scott, David W (2008). "The curse of dimensionality and dimension reduction". In: *Multivariate Density Estimation: Theory, Practice, and Visualization*, pp. 195–217.

Shah, Manzoor A and R Uma Shaanker (2014). "Invasive species: reality or myth?" In: *Biodiversity and conservation* 23.6, pp. 1425–1426.

Simberloff, Daniel and Leah Gibbons (2004). "Now you see them, now you don't!–population crashes of established introduced species". In: *Biological Invasions* 6.2, pp. 161–172.

Sousa, V. C. et al. (Apr. 2009). "Approximate Bayesian Computation Without Summary Statistics: The Case of Admixture". en. In: *Genetics* 181.4, pp. 1507–1519. ISSN: 0016-6731. DOI: 10.1534/genetics.108.098129. URL: http://www.genetics.org/cgi/doi/10.1534/genetics.108.098129 (visited on 12/08/2016).

Spash, Clive L et al. (2009). "Motives behind willingness to pay for improving biodiversity in a water ecosystem: Economics, ethics and social psychology". In: *Ecological Economics* 68.4, pp. 955–964.

Stroustrup, Bjarne (1994). *The design and evolution of C++*. Pearson Education India.

— (2003). *Abstraction, libraries, and efficiency in C+*. http://www.stroustrup.com/abstraction.pdf. [Online; accessed 28-September-2017]. (Visited on 08/24/2017).

— (2014). *Five Popular Myths about C++*. http://www.stroustrup.com/Myths-final.pdf. [Online; accessed 28-September-2017]. (Visited on 08/26/2017).

Thrupp, Lori Ann (2000). "Linking agricultural biodiversity and food security: the valuable role of agrobiodiversity for sustainable agriculture". In: *International affairs* 76.2, pp. 283–297.

Valéry, Loïc, Hervé Fritz, and Jean-Claude Lefeuvre (2013). "Another call for the end of invasion biology". In: *Oikos* 122.8, pp. 1143–1146.

Valéry, Loïc et al. (2008). "In search of a real definition of the biological invasion phenomenon itself". In: *Biological invasions* 10.8, pp. 1345–1351.

Van Houtven, George, John Powers, and Subhrendu K Pattanayak (2007). "Valuing water quality improvements in the United States using meta-analysis: Is the glass half-full or half-empty for national policy analysis?" In: *Resource and Energy Economics* 29.3, pp. 206–228.

Vignieri, Sacha N (2005). "Streams over mountains: influence of riparian connectivity on gene flow in the Pacific jumping mouse (Zapus trinotatus)". In: *Molecular Ecology* 14.7, pp. 1925–1937.

Villemant, C, J Haxaire, and J Streito (2006a). "The discovery of the Asian hornet Vespa velutina in France". In: *Insectes* 143.4, p. 5.

Villemant, Claire, Jean Haxaire, and Jean-Claude Streito (2006b). "Premier bilan de l'invasion de Vespa velutina Lepeletier en France (Hymenoptera, Vespidae)". In: *Bulletin de la Société entomologique de France* 111.4, pp. 535–538.

Villemant, Claire et al. (2011). "Predicting the invasion risk by the alien bee-hawking Yellow-legged hornet Vespa velutina nigrithorax across Europe and other continents with niche models". In: *Biological Conservation* 144.9, pp. 2142–2150.

Vitousek, Peter M (1994). "Beyond global warming: ecology and global change". In: *Ecology* 75.7, pp. 1861–1876.

Vitousek, Peter M et al. (1997). "Human domination of Earth's ecosystems". In: *Science* 277.5325, pp. 494–499.

Wakeley, John (2009). *Coalescent theory: an introduction*. 575: 519.2 WAK.

Wikipedia (2018). *Library (computing) — Wikipedia, The Free Encyclopedia*. [Online; accessed 27-January-2018]. URL: \url{https://en.wikipedia.org/wiki/Library_(computing)}.

Wilson, John RU et al. (2009). "Biogeographic concepts define invasion biology". In: *Trends in Ecology & Evolution* 24.11, p. 586.

Wright, Sewall (1931). "Evolution in Mendelian populations". In: *Genetics* 16.2, pp. 97–159.

Zadeh, LA (1965). "Fuzzy sets". In: *Information and Control* 8.3, pp. 338–353.

université PARIS-SACLAY
ÉCOLE DOCTORALE
Sciences du végétal : du gène à l'écosystème

Titre : Modèles de démogénétique environnementale

Mots clés : *génétique des populations, dispersion, niche écologique, coalescence*

Résumé : Les invasions biologiques étant des processus raisonnablement limités dans le temps et l'espace, elles fournissent un cadre propice à l'étude de modèles complexes par simulation numérique. Nous mettons à profit la méthode de Calcul Bayésien Approché (ABC) pour étudier l'invasion du frelon asiatique (*Vespa velutina*), en essayant d'estimer les paramètres d'un modèle probabiliste démographique et génétique spatialement explicite. La croissance des populations dans chaque unité paysagère est décrite par une fonction des conditions environnementales locales, tandis que les flux migratoires entre populations sont tirés dans des lois dont les densités sont fonctions de la distance géographique à parcourir. Certains paramètres de ces fonctions sont inconnus et doivent être estimés. Conditionnellement à la démographie, un processus de coalescence permet de simuler l'histoire génétique de l'échantillon. Une fois la simulation achevée, la procédure ABC permet de accepter/rejeter les valeurs de paramètres en fonction de la plausibilité des données génétiques qu'ils permettent de générer. La comparaison de modèles étant une étape clé de la méthodologie ABC, cela impose que différentes versions de simulateurs de coalescence puissent être rapidement développées. A cette fin, cette thèse propose Quetzal, une bibliothèque assez générale pour pouvoir aisément s'adapter à un grand nombre de modèles possibles, et qui inclue des algorithmes originaux et génériques. Les principaux concepts de programmation permettant d'utiliser et d'étendre Quetzal sont également exposés à travers les différents chapitres. Enfin, nous mettons à profit les particularités du contexte d'étude du frelon asiatique pour développer une méthodologie spécifique reposant sur le formalisme des partitions floues et qui permet, en recentrant l'analyse sur les processus démographiques très récents, de réduire le nombre d'hypothèses, le nombre de paramètres et le coût simulatoire de l'analyse.

Title : Environmental demogenetic models

Keywords : *populations genetics, dispersal, niche theory, coalescence*

Abstract : Because biological invasions are processes well delimited in both space and time, they offer a unique framework in which complex models can be studied by numerical simulations. We use the Approximated Bayesian Computation method (ABC) to study the *Vespa velutina* invasion, seeking to estimate the parameters of a spatially explicit demographic and genetic probabilistic model. The population growth in each landscape unit is described by a function of the local environmental features, whereas the migration flux between populations are sampled in laws which densities are functions of the geographical distance. Some of the parameters of these functions are unknown and should be estimated. Conditionally to the demography, a coalescence process allows for simulating the genetic history of the sample. Once the simulation done, the ABC method allows for accepting/rejecting the parameters values as a function of the data plausibility they generate. Models comparison requires that various versions of coalescence-based simulators can be quickly developed. This thesis offers Quetzal, a library general enough to be easily adapted to an open-ended number of models variants. Finally, we take advantage of the biological context to develop a specific methodology built on the fuzzy partitions formalism. It allows to focus only on the very recent demographic processes, and consequently to reduce the number of hypothesis, the number of parameters, and the simulation cost of the analysis.

Université Paris-Saclay
Espace Technologique / Immeuble Discovery
Route de l'Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France