Chapter 1

Digital_Coevolution
An agent-based simulation for host-parasite coevolution

Authors: Robert P. Dünner[1,2], Jukka Jokela[1,2].

Author affiliations: ETH Zurich, Institute for Integrative Biology[1], Eawag, Department of Aquatic Ecology[2].

Corresponding author: Robert P. Dünner; robert.duenner@eawag.ch, Eawag ECO BU-G-12, Uberlandstrasse 133, 8600 Dubendorf, Switzerland.

Keywords: Agent-based simulation, resource allocation hypothesis, host-parasite coevolution, negative frequency-dependent selection, matching alleles model, metapopulation structure, genetic drift, maintenance of genetic diversity.

Author contributions: RD conceived the ideas and designed the work, RD authored the simulation, RD and JJ led the writing.

Acknowledgements: I would like to thank Jukka Jokela, Nadine Tardent, Oliver Dressler, Felix Moerman and Ewa Merz for helpful comments on the manuscript.

Data accessibility: The simulation is available on GitHub.

Abstract

Host-parasite coevolution is highly context dependent. Agent-based simulations can be used to implement and control multiple aspects of a hosts and parasites life history. Such simulations are used for hypothesis testing and to generate expectations for empirical systems. Here I present the "Digital_Coevolution" simulation, a highly flexible agent-bases simulation that was created to explore questions in negative frequency-based host parasite coevolution.

Introduction

The growth of a host population and its infection dynamics with parasites is a highly dynamic and context dependent process. Each host is an individual that is born, competes for resources, grows, and ideally also reproduces. The fitness of a host individual is determined by its life history, trade-offs and interactions with its biotic and abiotic environment.
The host might be unlucky and encounter a parasite. Whether or not this encounter leads to an infection depends on many factors. Do the genotypes of the host and the parasite match, how genotype specific is the parasite, how infectious is the parasite? How many parasites where encountered by the host, and how competent are the hosts defenses? Does the host have enough resources to mount an appropriate immune defense? Does the encounter happen in an environment that benefits the host or in one that benefits the parasite? These and and many more factors can determine the success of an infection.

The outcome of interactions between host and parasite individuals depends on more than just the properties of the individuals themselves. All organisms are part of a population. Some populations might be larger, some smaller, some denser, others sparse, some isolated, others very well connected and part of a metapopulation.Those population properties can influence the infection dynamics between hosts and parasites. Larger host populations are less affected by genetic drift and can potentially maintain a larger genetic diversity. In small populations selection by parasites can interfere with drift and increase the speed of degradation of genetic diversity (see chapter 2). Metapopulation structure impacts coevolutionary dynamics as well (Ladle, Johnstone, and Judson 1993; Lion and Gandon 2015) and influences the genetic structure of both the host and the parasite (Judson 1995, 1997). The amount of migrants between subpopulations and the migration distance can, for example, influence virulence evolution (Boots and Sasaki 1999; Boots and Mealor 2007), enable local adaptation of the parasite on the host (Gandon 1996; Gandon and Michalakis 2002; S. Gandon 2002) or can select for increased genetic divergence between host subpopulations (see chapter 3).

It is challenging to account for all these individual and population level factors in order to conduct experiments on coevolution. Only few examples of well controlled systems with different comparable parameter settings that enable research on coevolution exist in the wild. A famous example would be the Trinidad guppy system (Oosterhout et al. 2006).
The alternative of establishing mesocosms, that would allow for more control over the factors influencing infection while keeping a certain level of ecological realism, is a logistical challenge in itself and can not be done in many institutions.
It is still challenging to get conditions right for laboratory populations to sustain parasite infection in just the right way to be able to do experiments. This process can sometimes absorb more time than the actual experiment itself (personal observation).
Simulations provide an alternative to this challenge. They can provide a way to implement theoretical expectations and biological assumptions for difficult-to-manipulate systems. Simulations can provide novel insights themselves or can be used to generate hypotheses that are more feasibly testable in natural systems.

Invaluable insights into many areas of host-parasite coevolution have been gained by simulation studies or theoretical models (see citations above). This includes the first hypotheses on how negative frequency-dependent selection by parasites maintains genetic diversity (Haldane 1949; Haldane and Jayakar 1963; Clarke 1976, 1979) or sexual reproduction (Jaenike 1978; Hamilton 1980; Hamilton, Axelrod, and Tanese 1990). More recent developments try to understand the process of evolution itself by using "digital organisms" like the AVIDA platform (Ofria and Wilke 2004; McKinley et al. 2008). The power and capabilities of computer simulations have increased steadily over time, as hardware became more powerful and software advanced. This increase in computational potential opens the opportunity to embed more and more biological realism into simulations, trying to position simulations closer to well controlled laboratory experiments. All models are wrong, but some are useful (Box 1976, 1979). A simulation can never be as realistic as the real world, as much as a map can never be the real landscape. Nevertheless one can try and include plausible details of the hosts and the parasites life cycles into a simulation in order to increase "realism". That way one would try to enhance the precision of predictions generated by this simulation Similarly, one can draw more information from a more detailed map.

One area that is especially promising in this respect are agent based based models (for example see (Boëte,

Seston, and Legros 2019) or (Bonabeau 2002)). These models simulate the behaviour of each individual organism explicitly, allowing for a great amount of detail. Instead of implementing required properties at the population level, those simulations describe rules for individual behaviour and development. All further higher level dynamics, for example interaction with other individuals or population and metapopulation behaviour then result from the sum of individual level interactions.

The simulation that is presented here, called the "Digital_Coevolution" simulation, is a population of digital organisms in the sense that it aims at simulating the dynamic behaviour of groups of single organisms (see for example (Lenski et al. 1999)). Population level behaviour emerges from interactions between several of these simulated organisms. This means Digital_Coevolution falls into the category of agent-based simulations. The Digital_Coevolution simulation was created as a tool to study how metapopulation structure influences coevolutionary dynamics between hosts and parasites. Due to the level of control over the life histories of both the host and the parasite as well as their interactions, the Digital_Coevolution simulation can also be extended to the study of other questions in host-parasite coevolution, like for example evolution of virulence.

This chapter serves as the documentation for the Digital_Coevolution simulation. It describes the inner workings and the reasoning behind the Digital_Coevolution simulation and should enable the reader to thoroughly understand and use the simulation. The first part describes the "biological logic" behind the code. It describes the functions that govern the behaviour of the agents in the simulation. The second part then describes the implementation of the Digital_Coevolution, moving from biological logic towards R logic. Throughout the document, the term agent and individual are used interchangeably.

This document is written in R markdown (Xie, Allaire, and Grolemund 2018; Allaire et al. 2020) in R (Team 2020) using R Studio (Team 2019) as IDE. It consists of blocks of text that are followed by snippets of commented code when appropriate. The text and the comments do show substantial overlap, trying to reinforce a thorough understanding of what the simulation does. The code snippets are not complete though, and serve mostly as explanations. If you would like to have a look at the complete source code or even contribute, you are very welcome to do so and invited to visit the Digital_Coevolution GitHub repository of the simulation (cite GitHub repository).

Part One: Biology of the Digital_Coevolution simulation

Overview

The Digital_Coevolution simulation simulates individual host and parasite agents in detail. It has instructions on within agent life history as well as for between agent behaviour. When groups of those agents are simulated, population level behaviour emerges. It is implemented as a time-forward simulation. The state of each individual agent is calculated at discrete time-steps and each individual lives for multiple time-steps. Each time-step is dependent on the state of the simulation at the time-step before, similar to a Markov chain. The description below gives an overview over the dynamics that are evaluated for each individual agent at each time step. Not all events do necessarily take place every time-step, as nearly all are context-dependent. Not all agents will experience the same events each time step. One time-step simulates one "day" in the life of an agent.

The agents that make up the Digital_Coevolution simulation have internal resource allocation trade-offs among fitness correlated life-history traits.

Each host agent has a distinct immune genotype and a set of life-history traits that it has inherited. Host agents are born as small individuals with enough starting resources for the first day of their life. They immediately start foraging for resources, for which they compete with other host agents. Resources are limited for each population. The outcome of competition for resources between host agents is dependent on the population density, the host agent size distribution and a stochastic component.

Host agents then use those acquired resources to invest into growth, maintenance or reproduction. Excess resources can be used to build up a fat storage for harder times. The older an individual grows, the less resources it invests into growth and the more it invests into reproduction. Reproductive allocation is implemented as iteroparous clonal capital breeding (Houston et al. 2007; Stephens et al. 2009). Each agent has to save up resources until a threshold is reached at which it can start reproducing. Each agent can reproduce several times during its lifetime, resources permitting. The gradual, age-dependent shift of resource investment from growth to reproduction ensures that early in life, the resources invested into reproduction are not sufficient to produce offspring. This creates a distinct juvenile phase that is dominated by growth. Later in life, the majority of resources are invested into reproduction and an individual only grows very slowly. This creates an adult life stage that is dominated by reproduction. An individual dies as soon as it runs out of resources (starves) or if it reaches the pre-set age threshold (senescence) and is removed. Senescence is thus a deterministic event at a certain age of an agent, if that agent has survived up to that age.

Throughout all its life stages, a host agent can encounter parasite agents. Groups of parasite agents are randomly ingested by the host agent with food resources while foraging. An infection may establish within a host individual depending on the host agents genotype and the number of parasite agents ingested per parasite agent genotype. All types of genotype specific infections can be implemented via an infection matrix. Each host individual can potentially ingest several parasites agents each time step while foraging. Parasite agents that are ingested will be removed from the parasite population, independent of their genotype, their infection success or the infection state of the host.

If an infection does establish within a host agent, it starts small. This means that infections do have a size within the infected host individual, reflecting both the physical space occupied and the resources needed by the host immune system to interact with the infection. Infections within host individuals grow and mature over time until they completely fill out the space that is available for them. This mimics a parasite that infects a certain host tissue, for example the gonads, and whose within host growth is therefore limited. The space that is available for a parasite infection is directly related to the size of the host individual. Infections steal resources from the host individual that are then not longer available to be used by the host. Virulence is thus implemented as a resource loss by the host agent towards the parasite agent infection. The amount of resources that a parasite infection steals is dependent on the size of the infection. A younger, smaller infection will steal fewer resources than a larger, more mature infection.

Parasite agent reproduction is implemented with the same life-history strategy as that of the host, as iteroparous clonal capital breeding. It is not free parasite agents that reproduce, but the infection within a host individual that acts as the reproducing unit. This means that new parasite agents are created by infected host agents and not by free parasite agents. This simulates a parasite behaviour that is inspired by that of micro-parasites. They have no substantial life history outside of their hosts, and in order to reproduce

have to obligatorily infect a host. Infections can only be established by one genotype at a time, there are no co-infections possible. As soon as an infection has stolen enough resources from a host to pass the capital breeding threshold, it will produce a clutch of parasite offspring that are shed without killing the host. Each infected host will therefore repeatedly shed bursts of new parasite agents into the parasite population. The amount of parasite agents produced that way is dependent on the parasite-infection size within the host agent, which is dependent on the age of the infection and the host size. Large infections in large host individuals can hence have a huge influence on the parasite population. An infection will kill its host agent as soon as the host agent does not have enough resources left to invest into maintenance. This happens when a parasite infection is absorbing a lot of resources and there are not enough resources coming in from foraging or the fat storage of the host.
The infection spreads horizontally through the host population, as parasite agents are passively ingested by host individuals while foraging. This, together with the non-lethal shedding of new parasite agents by infected host agent, creates a faeco-oral horizontal transmission.

The parasite epidemics spreads horizontally through the host populations, with host agent and parasite agent encounters being random. One should imagine a well-mixed population in terms of host agents, resources and also parasite agents. There can be more than one population though. The Digital_Coevolution simulation allows to specify any number of discrete populations with any amount of resources per population. Population size cannot be set exactly, but is an emergent property of the simulation. Population size is nevertheless strongly linked to the amount of resources that are simulated per population.
Several populations form a metapopulation when there is a distinct migration pattern between those populations. That is why the Digital_Coevolution simulation allows to specify any type of migration pattern between populations, including source-sink dynamics. The migration structure can be specified for both the host and the parasite separately. This can be used to create complex metapopulation structures through which hosts and parasites, and hence also epidemics, can spread.

Within agent dynamics

The within agent dynamics are the heart of the Digital_Coevolution simulation. Here the rules for the life histories of each individual agent are defined. This is where each host agent takes shape as an individual.
The host agents that make up the Digital_Coevolution simulation can best be imagined as small, mobile, aquatic filter feeders, i.e. zooplankters like *Daphnia* or *Cyclops*. Thinking of the agents as zooplankters can help with imagining some of the biology that was implemented. Host agents are called hosts in the code of the simulation itself.
During the simulation, each agent moves through different life-history stages, struggles for resources and gets potentially exposed to parasites. Those parasites are horizontally transmitted, passive dispersing obligate parasites, which might best be imagined as either a bacterial or fungal pathogens (microsporidians for example). Parasite agents are called parasites in the code of the simulation itself.
The following part gives an overview over the different functions, called "dynamics functions", that each govern the behaviour of a different life-history trait for each agent in the simulation. Each of the "dynamics functions" gets evaluated once every time-step.

Size

Size is one of the central parameters of the Digital_Coevolution simulation. It is based on the observation that most organisms grow over their lifetime, and that their size defines some of their interactions with their biotic and abiotic environment. In the Digital_Coevolution simulation size influences many life-history traits but especially the interaction with food resources. Larger agents consume more resources (eat more) but also have a larger metabolic requirement (need more food). This influences several other parameters that are linked either proportionally or with absolute thresholds against the resource availability within an agent. The feeding rate of an agent and the amount of parasite spores that this agent will ingest each time-step also are directly linked. A larger agent will be exposed to more parasite spores than a small one, per time-step.
A larger organism will, on the other hand, also be able to produce more offspring per time-step, as it can acquire more resources faster and the production of one offspring always costs the same in term of resources.
Size is not implemented as one single "dynamics function", but as a trait (of an agent) that is influencing many other "dynamics functions". Size itself is influenced in the metabolism function when resources get allocated between different life history processes. Growth is one of these processes that can get resources. Digital_Coevolution agents are born with size = 1 and then invest resources into growth depending on their age. The younger an individual is, the more resources it will invest into growth as opposed to reproduction. The proportion of resources invested in reproduction declines negatively proportional to numerator 2 with age. Numerator 2 creates an infant growth boost, meaning that the resources that are available at the first time step are counted twice for reproduction.
Size can also be switched off by setting the host.size parameter to "OFF", which then skips the growth step and keeps all individuals at size = 1 at all times. This is especially useful when one would like to compare results to other, less complex, simulations.

```
# Depending on the age of the host, resources are funneled more towards reproduction or
# more towards growth.
if(host.size == "ON"){
    Host[Alive.Hosts$Is.Alive, Size :=
           Size + (pmin(Reproduction.Allocation * Size, Resource.Work) * (2 / Age))]
}
```

In the Digital_Coevolution simulation, only host agents have an individual size. Parasite agents have no size themselves but the infection within the host agents does. The infection size within the host agent is directly linearly limited by the host agent size. Smaller individuals can only maintain smaller infections.

```
Host[Alive.Hosts$Is.Alive, Infection.Size := Infection.Size * infection.growth.factor]

Host[Alive.Hosts$Is.Alive, Infection.Size :=  pmin(Infection.Size, Size)]
```

Age

All things age. The agents in the Digital_Coevolution simulation should be no exception. There is an age variable built into both host and parasite agents. It is a simple integer counter whose value increases by one each time-step. There is an old-age cut-off, essentially creating senescence, implemented as a threshold value for both host and parasite. Once an agent reaches this threshold, it is culled and removed from the simulation.

```
# Takes the Age vector and adds one everywhere, then truncates by a pre-set threshold.
Host[Alive.Hosts$Is.Alive, Age := Age + 1L]

Host[Alive.Hosts$Is.Alive & Age > age.threshold.host, Alive := 0L]

Parasite[Alive.Parasites$Is.Alive, Age := Age + 1L]

Parasite[Alive.Parasites$Is.Alive & Age > age.threshold.parasite, Alive := 0L]
```

Within host resources

All host agents compete with other host agents for a shared pool of finite resources that renew every time-step. Each host agent will obtain a discrete amount of resources that is dependent on the size of the host agent and the size of the host population respective to the available food resources.
The resource allocation within each agent works as follows.
The resource pool within each agent is split in three "containers". One container is for incoming resources gained through foraging ("Resource.In"), one for resources that are currently available for the metabolism ("Resource.Work"), and one is for storing excess resources in a fat storage (Resource.Have). Each time-step, resources are moved between those containers. The resources dynamics take place over three time-steps.
Resources that are gained by a host agent through foraging will remain in the "Resource.In" container for the duration of one time-step and only become metabolically available in the next time-step. After one time-step, the resources are moved to the "Resource.Work" container. If only few resources are coming in, host agents have the ability to also recruit resources from the "Resource.Have" container to the "Resource.Work" container. From the "Resource.Work" container, resources are distributed to different life-history traits according to this agents allocation scheme. Any resources that left over in the "Resource.Work" container at the beginning of the next time-step are moved into the "Resource.Have" container. The amount of resources that can be stored in the "Resource.Have" container grows with a power function with the size of the host agent. If excess resources are available but the "Resource.Have" container is full, those resources are lost. Whether or not there are resources remaining in the "Resource.Work" container at the beginning of a time-step is dependent on the settings for the life-history allocations. These settings are explained a bit further below in the "Host metabolism" section. This three step process of resource movement to different containers within host agents was implemented to reflect natural resource dynamics, where consumed resources are not immediately metabolically available but have to be digested first. The combination of the time-lag and the "Resource.Have" container also allows host agents to feather of variability in resource availability to some extent.
The last step in the function is a starvation process, which culls all individuals that have less than a pre-set metabolic resource threshold available.

```
# This moves the leftover resources from the resource.work container to the resource.have
# container. So leftover metabolic resources to the "fat" storage.
Host[Alive.Hosts$Is.Alive, Resource.Have := Resource.Have + Resource.Work]

# I will make the resource.have threshhold, so basically the fat content, size dependant.
# As larger individuals should be able to proportionally store more fat, a qubic increase
# instead of a linear increase with size migth be applicable.
Host[Alive.Hosts$Is.Alive, Resource.Have := pmin(Resource.Have, (Size ^ 2))]

# Then the resource.work (metabolic resource) gets filled by the resource.in container of
```

```
# the last timestep, creating a delay between feeding and availability of energy.
# This part also lets hosts that get less external resources use some stored resources to
# fill resource.work
Host[Alive.Hosts$Is.Alive, Resource.Work :=
       Resource.In + pmin((round(Size) - Resource.In), Resource.Have)]
Host[Alive.Hosts$Is.Alive, Resource.Have :=
       Resource.Have - pmin((round(Size) - Resource.In), Resource.Have)]

# This part removes individuals that have run out of resources (starving).
Host[Alive.Hosts$Is.Alive & Resource.Work < resource.threshold.host, Alive := 0]
```

Host metabolism

The agents that make up the Digital_Coevolution simulation have internal resource allocation trade-offs among fitness correlated life-history traits.
Each host has a metabolic resource availability that is a result of the ingested resources a time-step earlier and resources available from the fat storage (see "within host resources" section above). These resources now need to be allocated to different life-history traits. In the case of the Digital_Coevolution simulation, there are three life-history traits that can receive resources.
The "Immune.State" is a variable that integrates the overall maintenance state of the host including its immune system. It is the baseline investment that has to be made by every organism into maintenance of tissue, body temperature, innate immune response etc. Basically this is the metabolic cost of being alive. It is the first trait that any agent will have to invest resources in. The amount of resources invested is dependent on the hosts size and an adjustable immune allocation proportion. The value of "Immune.State" resets every time-step, which means that the host will have to invest resources to immunity every time-step again.
Next the host will invest into reproduction ("Reproduction.Have"). The amount of resources invested into reproduction is determined by the hosts size and the reproduction allocation setting. If the host individual has fewer internal resources available than could maximally be invested into reproduction, all remaining resources are invested into reproduction.
An individual has to grow as well, and depending on the age of the individual, a certain proportion is invested into growth of the "Size" parameter rather than reproduction. This proportion is declining negatively proportional to age, so that younger individuals invest proportionally more into growth instead of reproduction. Because a young individual invests few resources into reproduction and reproduction is implemented as capital breeding, young individuals will not reproduce for the first few time-steps. Older individual do invest more into reproduction than growth. This size step can also be switched off to create a simulation with agents of constant size.

```
# This part drains resource.work and assigns to immune.state, refreshes every round, so no
# immune buildup.
Host[Alive.Hosts$Is.Alive, Immune.State :=
       pmin(Immune.Allocation * Size, Resource.Work)]

Host[Alive.Hosts$Is.Alive, Resource.Work :=
       Resource.Work - pmin(Immune.Allocation * Size, Resource.Work)]

Host[Alive.Hosts$Is.Alive, Reproduction.Have :=
       Reproduction.Have +
       (pmin(Reproduction.Allocation * Size, Resource.Work) * (1 - (2 / Age)))]

# This part drains resource.work and moves to reproduction have, cummultative.
# Depending on the age of the host, resources are funneled more towards reproduction or
# more towards growth.
if(host.size == "ON"){
  Host[Alive.Hosts$Is.Alive, Size :=
```

```
          Size + (pmin(Reproduction.Allocation * Size, Resource.Work) * (2 / Age))]
}

Host[Alive.Hosts$Is.Alive, Resource.Work :=
       Resource.Work - pmin(Reproduction.Allocation * Size, Resource.Work)]
```

Host reproduction

Host reproduction is implemented as an iteroparous clonal capital breeder (Houston et al. 2007; Stephens et al. 2009). Each host individual must accumulate resources until it reaches an adjustable threshold in order to produce a batch (a clutch) of offspring. The number of offspring produced per agent is dependent on the resources saved up for reproduction and an adjustable multiplicative reproduction factor. It is also indirectly dependent on the size of the individual, as larger individuals can obtain more resources and reach the reproduction threshold more often.
Reproduction is implemented in the Digital_Coevolution simulation by first calculating the identities of all host agents that have accumulated enough resources to reproduce. Those host agents are then copied multiple times. The amount of times those host agents are copied is dependent on the amount of resources each individual had accumulated in excess of the reproduction threshold and the multiplicative reproduction factor.
The life-history traits that represent "genotype-level" heritability are directly copied from the parent agent. Those include the "Reproduction.Allocation" and "Immune.Allocation" traits that detail which proportion of resources is invested into reproduction and immune system respectively. It includes the immune genotype of the agent and the last immune state of the parent agent. Other traits of an agent are dependent on its individual background and are not heritable. For those traits the offspring gets re-set to the starting conditions. Those traits include age, size, infection state and all resource traits ("Resource.In", "Resource.Have", "Resource.Work"). Other properties of host agents like their population assignment are copied as well. The parent agent remains in the population and looses the resources it has used for reproduction.

```
# First calculates the number of reproducing individuals, and how many offspring each
# individual should produce.
reproducing.hosts <- Host[, rep(
  .I[Alive.Hosts$Is.Alive & Reproduction.Have > reproduction.threshold.host],
  times =
    round(
      Host[Alive.Hosts$Is.Alive &
             Reproduction.Have > reproduction.threshold.host,Reproduction.Have] -
        ((reproduction.threshold.host) * reproduction.factor.host)
  )
  ]

# If there are reproducing individuals, proceed, otherwise, skip.
# Open if statement.
if (Host[reproducing.hosts, .N] > 0) {

  # In the subset of empty (dead) host rows, fill host rows according to the reproducing
  # hosts.
  Host[Host[, .I[!Alive.Hosts$Is.Alive][seq(Host[reproducing.hosts, .N])]],
         `:=`
       (Alive = 1L,
         Host.Replicate = Host[reproducing.hosts, Host.Replicate],
         Host.Population = Host[reproducing.hosts, Host.Population],
         Host.Infection.Genotype = NA,
         Age = 1,
         Resource.Have = 1,
```

```
          Reproduction.Allocation = Host[reproducing.hosts, Reproduction.Allocation],
          Immune.Allocation = Host[reproducing.hosts, Immune.Allocation],
          Immune.Genotype = Host[reproducing.hosts, Immune.Genotype],
          Resource.In = 1,
          Resource.Work = 0,
          Reproduction.Have = 0,
          Immune.State = Host[reproducing.hosts, Immune.State],
          Infection.State = 0L,
          Infection.Size = 0,
          Parasite.Resources = 0,
          Host.TempID = NA,
          Size = 1,
          Host.Generation = (Host[reproducing.hosts, Host.Generation] + 1L),
          Origin = Host[reproducing.hosts, Host.Population])
          ]

  # Withdraw the resources used for reproduction.
  Host[Alive.Hosts$Is.Alive & Reproduction.Have > reproduction.threshold.host,
       Reproduction.Have :=
         Reproduction.Have - (round(Reproduction.Have) - reproduction.threshold.host)]

# Close if statement.
}
```

Disease and virulence

Infections have a size, the "Infection.Size" trait, within a host agent. The size of an infection defines how many resources get withdrawn from the hosts resource budget ("Resource.Work") towards the infection by the parasite. Virulence is hence implemented as a resource loss for the host agent. Infections all start at size = 1 and then grow at an exponential rate, mimicking natural infection with micro-parasites. Infections can grow until they fill the space that is available, which is linked to the size of the host individual. The resources that get withdrawn by the infection can then be used by the parasite to reproduce. Resource stolen by the parasite in this way are stored in the host level variable "Parasite.Resources".

```
# First the infection matures.
Host[Alive.Hosts$Is.Alive, Infection.Size := Infection.Size * infection.growth.factor]

Host[Alive.Hosts$Is.Alive, Infection.Size := pmin(Infection.Size, Size)]

# Then the parasite draws resources, dependant on infection size.
Host[Alive.Hosts$Is.Alive, Parasite.Resources :=
       Parasite.Resources + pmin(Resource.Work, Infection.Size * virulence)]

Host[Alive.Hosts$Is.Alive, Resource.Work :=
       Resource.Work - pmin(Resource.Work, Infection.Size * virulence)]
```

Parasite reproduction

The reproduction of the parasite is implemented as an iteroparous clonal capital breeder as well. It is not the free parasite agents that reproduce, but the infections within infected host agents that act as the reproducing unit. This means that the parasite infection within the host agent must accumulate resources until an adjustable threshold value is reached, upon which a batch of offspring parasite agents is produced. The threshold value is independent of the infection size, but the amount of offspring parasite agents produced is proportional to the accumulated resources. As larger infections accumulate more resources faster, larger

infections will produce more offspring parasite agents.
The implementation of parasite reproduction is identical to that of host reproduction. The amount of resources that a infection has stolen from the host individual is checked against the parasite reproduction threshold to calculate which infection will have an reproduction event. Then the according number of parasite agents is created. The newly created parasites inherit the genotype of their "parent" infection, as well as the population assignment. Other values like age are re-set to starting values.

```
# The first step selects the subset of the host population that is infected and where the
# infections have accumulated enough resources to reproduce. And then multiplies this
# subset by the reproduction factor.
reproducing.parasites <-
  Host[
    ,
    rep(.I[Alive.Hosts$Is.Alive & Parasite.Resources > reproduction.threshold.parasite],
        times = round(
          Host[Alive.Hosts$Is.Alive &
                 Parasite.Resources > reproduction.threshold.parasite,
               Parasite.Resources] - reproduction.threshold.parasite) *
          reproduction.factor.parasite)]

# If there are reproducing individuals, proceed, otherwise, skip.
# Open if statement.
if (Host[reproducing.parasites, .N] > 0) {

  # This part selects the "dead" rows in the parasite data.table, then calcualtes how
  # many offsping are produced, and updates the values from the corresponding
  # host data.table
  Parasite[
    Parasite[, .I[!Alive.Parasites$Is.Alive][seq(Host[reproducing.parasites, .N])]],
             `:=`
             (Alive = 1L,
               Parasite.Replicate = Host[reproducing.parasites, Host.Replicate],
               Parasite.Population = Host[reproducing.parasites, Host.Population],
               Parasite.Infection.Genotype =
                 Host[reproducing.parasites, Host.Infection.Genotype],
               Attack.Host.TempID = NA,
               Attack.Host.Genotype = NA,
               Success.Parasite.Infection.Genotype = NA,
               Ingested = 0,
               Age = 1L)
               ]

  Host[Alive.Hosts$Is.Alive & Parasite.Resources > reproduction.threshold.parasite,
       Parasite.Resources :=
         Parasite.Resources -
         (round(Parasite.Resources) - reproduction.threshold.parasite)]

  # Uptdate parasite alive vector
  set(Alive.Parasites, j = "Is.Alive", value = Parasite[, Alive == 1])

# Close if statement.
}
```

Between agent dynamics

A great benefit of agent-based simulations is that between-agent dynamics can be observed at the interaction level. When the agents in the Digital_Coevolution simulation are imagined as zooplankters, distinct groups of agents form populations of such zooplankters. This simulation was not written with natural populations in mind, but with laboratory populations. A population in the Digital_Coevolution simulation can be imagined as a glass jar with some media and a few *Daphnia* in it. Environmental conditions in those glass jars are fairly stable and once a day food becomes available. With limited amounts of medium in a glass jar available to filter feeding zooplankters, a large enough population can potentially "over-filter" the available media volume. This affects interactions with parasite agents as well, as they can be thought of as being suspended in the same media volume as the food particles. As parasites are passively ingested during food consumption, resource dynamics and parasite epidemiological dynamics are closely linked.

Host resources population wide

The host resources are implemented as batch resources per population that renew every time-step. The hosts agents then compete for a part of those resources. In order to avoid infinitely small resource consumption, resources are partitioned out according to a transformed Poisson distribution. Each time-step there is a random Poisson vector calculated, with one element per host. As the amount of resources available per host is dependent on the population size and the summed-up filtering capacity of the host individuals, the expected mean of the distribution is proportional to the fraction of total filtering capacity per individual. This fraction is calculated by dividing the host individuals size (equal its filtering capacity) by the sum of sizes over the population, and that fraction is multiplied by the size of the individual again. This means that the population is density dependent with a soft upper border.

```
# This part takes all the available resources, and redistributes them according to
# host size. It adjusts the resources by relative filtering capacity if the population
# gets less than maximal food.
Host[Alive.Hosts$Is.Alive, Resource.In :=
       (rpois(
         n = .N,
         lambda = (Size * min(1, (resources.host[Host.Population[1]] / sum(Size)))) *
           resource.grain) / resource.grain),
     by = list(Host.Population, Host.Replicate)]

# This part restricts resource in to Size + 10% in order to avoid unrealistic overfeeding
Host[Alive.Hosts$Is.Alive, Resource.In := pmin(Resource.In, (Size * 1.1))]
```

Infection model

The infection model in the Digital_Coevolution simulation is implemented as a genotype-by-genotype look-up table that defines the infection affinity between the two genotypes. This can be specified so that the resulting infection model ranges anywhere between a gene-for-gene model to a matching-alleles type infection model. Most simulation models of host-parasite coevolution are run as a perfect matching-alleles model, with complete parasite specificity.

```
# Infection dynamic parameters, row is host, column is parasite
infection.table <- matrix((1 - parasite.specificity),
                          nrow = host.genotypes,
                          ncol = parasite.genotypes)

diag(infection.table) <- 1
```

Host exposure to parasites

This is arguably the most important part of the simulation. Here the host agents get exposed to new parasites. The exposure within populations is completely random, i.e. there is no host seeking behavior by the parasite and no parasite avoidance by the host. The simulated transmission is faeco-oral, so the transmission is linked to the feeding rate. As the parasite population can be both larger or smaller than the host population, the number of parasites each host is exposed to varies largely. Host agents consume parasite agents which are then removed from the population, independent of the infection state of the host. Infection success is dose and genotype dependent. Co-infections are not possible.

The implementation starts by calculating for each parasite agent whether or not it has been ingested by any host agent, which is dependent on the size of the host population relative to the available resources (is there over- or underfeeding?). Then each parasite agent that has been ingested gets assigned the specific host agent by which it has been ingested. This is dependent on the size of an individual host agent relative to the summed-up size of the population because the feeding rate of host agents is directly linked to host agent size. One host agent can ingest several parasite agents. Not all ingestion events lead to a successful infection. Only one parasite can infect the host. The probability for each of the ingested parasite agents to successfully infect is weighted by both the number of parasite agents with a certain genotype and by the matching of the host and parasite genotype in the infection table. In other words, with perfect specificity, only parasite agents that match the host agents genotype have a chance at establishing an infection. If the specificity of the infection system is not perfect, then there is a dose response in the probability of the infection identity. Finally, host agents that have already been infected in a prior time-step cannot acquire a new infection (vaccination effect), but will still consume and remove parasite agents from the population.

```
# This part sets an identifier to be used later on.
Host[Alive.Hosts$Is.Alive, Host.TempID := 1 : .N]

Host[! Alive.Hosts$Is.Alive, Host.TempID := NA]

  set(Parasite, j = "Ingested", value = 0)
  set(Parasite, j = "Attack.Host.TempID", value = NA)
  set(Parasite, j = "Attack.Host.Genotype", value = NA)
  set(Parasite, j = "Success.Parasite.Infection.Genotype", value = NA)

# This part removes the identifier from those hosts that already are infected, creating a
# vaccination effect.
Host[Alive.Hosts$Is.Alive & Infection.State == 1, Host.TempID := NA]

# This part does assign to each parasite score if it has been ingested by a host,
# dependant on resource availability, host and parasite population size.
Parasite[Alive.Parasites$Is.Alive,
           Ingested :=
           rbinom(n = .N,
                  size = 1,
                  prob = min(1,
                             (sum(
                               Host[Alive.Hosts$Is.Alive &
                                      Host.Population == Parasite.Population[1] &
                                      Host.Replicate == Parasite.Replicate[1], Size]) /
                                resources.host[Parasite.Population[1]]))),
         by = list(Parasite.Population, Parasite.Replicate)]

# Updating the Alive.Parasites$Is.Ingested vector for faster lookups.
Alive.Parasites[Alive.Parasites$Is.Alive,
                Is.Ingested := Parasite[Alive.Parasites$Is.Alive, Ingested == 1]]
```

```
# This part does assign to each parasite spore the host tempID it has been ingested by
Parasite[Alive.Parasites$Is.Ingested,
         Attack.Host.TempID :=
           base:::sample(x = Host[Alive.Hosts$Is.Alive &
                                    Host.Population == Parasite.Population[1] &
                                    Host.Replicate == Parasite.Replicate[1],
                                  Host.TempID],
                         size = .N,
                         replace = TRUE,
                         prob = c(
                           Host[Alive.Hosts$Is.Alive &
                                  Host.Population == Parasite.Population[1] &
                                  Host.Replicate == Parasite.Replicate[1], Size] *
                             (.N / resources.host[Parasite.Population[1]]) *
                             min(1, resources.host[Parasite.Population[1]] /
                                   sum(
                                     Host[Alive.Hosts$Is.Alive &
                                            Host.Population == Parasite.Population[1] &
                                            Host.Replicate == Parasite.Replicate[1],
                                          Size]))
                         )
                        ),
         by = list(Parasite.Population, Parasite.Replicate)]

# This part assigns to each parasite the host genotype it has been ingested by.
Parasite[Alive.Parasites$Is.Ingested, Attack.Host.Genotype :=
           Host[Alive.Hosts$Is.Alive][Attack.Host.TempID, Immune.Genotype]]

# This part calculated which parasite successfully infects, takes into account the
# relative abundance of ingested parasite spores and their genetic specificity.
Parasite[Alive.Parasites$Is.Ingested & ! is.na(Attack.Host.TempID),
         Success.Parasite.Infection.Genotype :=
           sample(c(NA, Parasite.Infection.Genotype),
                  size = 1,
                  prob = c(1,
                           infection.table[Attack.Host.Genotype[1],
                                           Parasite.Infection.Genotype]),
                  replace = TRUE),
         by = list(Attack.Host.TempID, Parasite.Population, Parasite.Replicate)]

# And the last thing to do would be to assign the infection genotype back to the host.
infected.hosts <- unique(Parasite[Alive.Parasites$Is.Ingested &
                                    ! is.na(Success.Parasite.Infection.Genotype)],
                         by = "Attack.Host.TempID")$Attack.Host.TempID

infected.hosts.infection.genotypes <-
  unique(Parasite[Alive.Parasites$Is.Ingested &
                    ! is.na(Success.Parasite.Infection.Genotype)],
         by = "Attack.Host.TempID")$Success.Parasite.Infection.Genotype

Host[Host[, .I[Alive.Hosts$Is.Alive]][infected.hosts],
     Host.Infection.Genotype := infected.hosts.infection.genotypes]
```

```
# And the very last thing is to update the infection status of the host that got assigned
# a infection.genotype.
Host[Alive.Hosts$Is.Alive & ! is.na(Host.Infection.Genotype) & Infection.State == 0,
     c("Infection.Size", "Infection.State") := 1]

# And kill the parasites that have been ingested
Parasite[Alive.Parasites$Is.Ingested, Alive := 0]
```

Host resources simulation wide

Resources indirectly control the size of a simulated population. Therefore the resources can be used together with the migration pattern to create any type of a metapopulation. The resources simply control the size of the subpopulations, while the migration pattern controls the movement of agents between the subpopulations. The resources that are available per subpopulation implemented as an integer vector, where the number of vector elements is the number of populations that are going to be simulated, and the value of the vector elements is the size of the simulated population. This way it is possible to simulate a metapopulation that consists of subpopulations of different population sizes.

```
# Here at the same time the number of populations and their size are set. For each
# population define explicitly the amount of resources it receives daily.
resources.host <- c(500, 50, 50)

###
number.populations.host <- length(resources.host)

populations.host <- c(1 : number.populations.host)

starting.population.sizes.host <- ceiling(resources.host / 2)

###
number.populations.parasite< - number.populations.host

populations.parasite <- c(1 : number.populations.parasite)

starting.population.sizes.parasite <- ceiling(resources.host * 10)
```

Between population dynamics

A central feature of the Digital_Coevolution simulation is that it can simulate several interconnected subpopulations of both host and parasite agents. When a host agent population is imagined as a glass jar full of zoo-plankton, a metapopulation can be imagined as the climate chamber in which the glass jars are standing. Depending on the migration schedule, host or parasite individuals are moved between different glass jars in the climate chamber. Similarly can host and parasite agents in the Digital_Coevolution simulation migrate between different host and parasite populations. This migration happens once a time-step (once a day). And like in an experiment, migration can happen at different rates between different populations. Last but not least, those populations can also be of different sizes, with different amounts of food resources available to them.

Hierarchical metapopulation

Population inter-connectivity is implemented as a population-by-population fully crossed look-up table that defines how many individuals move from which population to which, both for the host and the parasite. This allows for the implementation of arbitrary metapopulation structures from only lightly connected large metapopulations to a network of closely knitted subpopulations. Combined with the explicit resource availability per population, this can also create asymmetrical metapopulations, as for example a mainland-island model.
Most importantly, by enabling the user to specify the metapopulation structure for both the host and the parasite separately, situations can be created where either the host or the parasite migrate substantially more. This enables the use of the Digital_Coevolution simulation in questions of host-parasite coevolution in metapopulations, which are research questions of unrivaled interest.

```
# Migration matrix for the host
migration.matrix.host <- matrix((host.migration / number.populations.host),
                                nrow = number.populations.host,
                                ncol = number.populations.host)

diag(migration.matrix.host) <-
  diag(migration.matrix.host) + (1 - host.migration)

# Migration matrix for the parasite
migration.matrix.parasite <- matrix((parasite.migration / number.populations.parasite),
                                    nrow = number.populations.parasite,
                                    ncol = number.populations.parasite)

diag(migration.matrix.parasite) <-
  diag(migration.matrix.parasite) + (1 - parasite.migration)
```

Parasite migration

The “parasite.migration.function” allows the parasite to migrate between different populations. It does so simply by picking a random subset of the free parasite population, and randomly assigning it to a new population. Migration is thus completely random and independent of any parasite properties. By supplying a migration matrix, the probabilities for each population of origin to end up in a certain population of destination can be manipulated, creating specific migration patterns.

```
parasite.migration.function <- function(){
  Parasite[Alive.Parasites$Is.Alive,
           Parasite.Population :=
             base:::sample(1 : number.populations.parasite,
                                              size = .N,
                                              prob =
                             migration.matrix.parasite[Parasite.Population[1], ],
```

```
                            replace = TRUE),
          by = list(Parasite.Population, Parasite.Replicate)]
  }
```

Host migration

The host migration works analogously to the parasite migration. A set fraction of the host population is randomly assigned to a new population according to the probabilities from the migration matrix.

```
host.migration.function <- function(){
  Host[Alive.Hosts$Is.Alive,
       Host.Population :=
         base:::sample(1 : number.populations.host,
                       size = .N,
                       prob = migration.matrix.host[Host.Population[1], ],
                       replace = TRUE),
       by = list(Host.Population, Host.Replicate)]
  }
```

Part Two: Implementation of the Digital_Coevolution simulation

Overview

The Digital_Coevolution simulation has two logical areas. The most important part of the simulation describes all the rules for the behaviour within and between individuals. This "biological logic" is mostly implemented in the functions and parameters that have been described in part one of this document. The second area of the simulation is the implementation of the simulation itself. How an agent actually is simulated, how time is implemented, how results are recorded. This is described in part two of this document.
The core structure of the Digital_Coevolution simulation is the data.table data structure (Dowle and Srinivasan 2019). A data.table is also an data.frame but much more powerful for data handling. Each row of the data.table contains all information on one individual agent. Each column of the data.table then represents one trait, for example the immune genotype, the reproduction allocation proportion, the amount of resources already saved for reproduction and the infection state.
The whole data.table then is the state of all traits of all agents at a certain point in time. At each time-step, each of the columns gets updated according to the "dynamics functions" that have been described in part one of this document. This utilizes R's vectorized structure, as the same operation can be applied to all elements of a column simultaneously in a very computationally efficient way. At specified intervals, a copy of the data.table gets saved and which allows for the analysis of host-parasite coevolutionary dynamics over time.

The Digital_Coevolution simulation currently consists of three to four interdependent R scripts.
The "Digital_Coevolution_Dynamics_Functions.R" script, where the within and between agent dynamics are defined and where the time-forward simulation process is defined.
The "Digital_Coevolution_Parameterspace.R" script, where different within and between agent parameters can be set. Those parameters for example define the type of agent that is simulated.
The "Digital_Coevolution_Run.R" script, which coordinates the simulation.
The "Digital_Coevolution_User.R" script, which is where the user interacts with and runs the Digital_Coevolution simulation.

In order to be able to run the Digital_Coevolution simulation, simply download or copy the relevant scripts from the GitHub repository to your computer (link to github). On the GitHub repository there are detailed instructions on how to install and use the Digital_Coevolution simulation on either normal personal computers or on high performance cluster computers.
The Digital_Coevolution simulation is implemented in R, so a R installation is needed as well. The newest R version for your system can be downloaded from CRAN.

Setting up the Digital_Coevolution

Data structure

The Digital_Coevolution simulation is implemented in a way that each host or parasite agent is represented as a vector. Each element of the vector corresponds to a certain trait of that agent, as for example age or genotype. Those vectors are then stacked to make up the populations. Populations are then represented as tables, with each individual agent corresponding to one row. Reading out a column of this table, perpendicular to the rows, gives a vector that represents the variable/trait state of the whole population at a certain time-point. This allows for the manipulation of all individuals simultaneously if necessary.
As agents can reproduce and also die, the population sizes will vary throughout the simulation. This means that the number of rows in the data.table will have to vary throughout the simulation as well. Having to vary row numbers of data.tables (or worse, data.frames) in a loop is depreciated in R, as R will create a new copy of the old data.frame whenever the number of rows or columns changes. This can slow code down compared to situations when a data.frame of constant size is used. Therefore, the data.table used for the Digital_Coevolution simulation is created to be able to accommodate the largest possible population size in the simulation. Adding a variable that records whether or not the agent that is residing in this row is currently alive or dead allows to subset this larger data.table to just the alive population. When new agents are created (born) they are assigned to a row that has either been empty or that contains an individual with the marker "dead". The rows in the data.table are thus constantly recycled. This circumvents the growing

data.table problem and speeds up the simulation.
The Digital_Coevolution simulation relies heavily on the data.table package and hence its syntax. The data.table syntax is analogous to SQL and has three elements: Where, order by / select, and update group / by. View below for the first two rows of the host data.table at the initializing state of the simulation.

```
##    Alive Host.Replicate Time Host.Population Host.Infection.Genotype Age
## 1:     1              1    0               1                     <NA>   1
## 2:     1              1    0               1                     <NA>   1
##    Resource.Have Reproduction.Allocation Immune.Allocation Immune.Genotype
## 1:             1                   0.35              0.35               3
## 2:             1                   0.35              0.35               3
##    Resource.In Resource.Work Reproduction.Have Immune.State Infection.State
## 1:           1             0                 0            0               0
## 2:           1             0                 0            0               0
##    Infection.Size Parasite.Resources Host.TempID Size Host.Generation Origin
## 1:              0                  0          0    1               1      1
## 2:              0                  0          0    1               1      1
```

The creation of the first agents

The simulation process is initialized by the creation of the host and parasite agents.
This is done by a call to the "individual.creator.function" that is defined in the "Digital_Coevolution_Dynamics_Functions" script. It uses variables that are set in either the "Digital_Coevolution_User" script or the "Digital_Coevolution_Parameterspace" script. That includes variables that will regularly be changed like the number of replicates, and variables that will be changed less often, such as the allocation of resources to reproduction. The "individual.creator.function" is automatically called when the simulation is started.
The individual.creator.function creates two data.tables, one for the host, one for the parasite, where each line corresponds to one individual.

```
#########################################################################
# In this step I preallocate all the data.table space that I could be using in an
# attempt to speed up the simulation.
individual.creator.function <- function(){

  preallocation.margin <- 10 # How many times larger the data.table size.

  parasite.margin <- 1

  preallocation.length <-
    sum(starting.population.sizes.host * replicates * preallocation.margin)

  preallocation.parasite <-
    sum(starting.population.sizes.parasite * replicates *
          preallocation.margin * parasite.margin)


  if(exists("Host")) {rm(Host, pos = ".GlobalEnv")}


  Host <<- data.table(
    Alive = integer(preallocation.length),
    Host.Replicate = integer(preallocation.length),
    Time = integer(preallocation.length),
    Host.Population = integer(preallocation.length),
    Host.Infection.Genotype = factor(NA, levels = c(1 : parasite.genotypes)),
```

```
  Age = integer(preallocation.length),
  Resource.Have = numeric(preallocation.length),
  Reproduction.Allocation = numeric(preallocation.length),
  Immune.Allocation = numeric(preallocation.length),
  Immune.Genotype = factor(sample(c(1 : host.genotypes),
                                  size = preallocation.length,
                                  prob = rep(1 / host.genotypes,host.genotypes),
                                  replace = T),
                           levels = c(1 : host.genotypes)),
  Resource.In = numeric(preallocation.length),
  Resource.Work = numeric(preallocation.length),
  Reproduction.Have = numeric(preallocation.length),
  Immune.State = numeric(preallocation.length),
  Infection.State = integer(preallocation.length),
  Infection.Size = numeric(preallocation.length),
  Parasite.Resources = numeric(preallocation.length),
  Host.TempID = integer(preallocation.length),
  Size = numeric(preallocation.length),
  Host.Generation = integer(preallocation.length),
  Origin = integer(preallocation.length)
)

# Here I initialize the starter populations.
# Replicate, this way I hope to paralellize the replicate calculations.
Host[, Host.Replicate :=
       c(rep(1 : replicates,
             each = sum(starting.population.sizes.host)),
         integer(preallocation.length - sum(starting.population.sizes.host) *
                   replicates))]

# Alive variable, 1 = alive, 0 = not alive
Host[Host.Replicate != 0, Alive := 1L]

# Population
Host[Alive == 1, Host.Population :=
       as.integer(rep(1 : number.populations.host,
                      times = starting.population.sizes.host)),
     by = Host.Replicate]

# Age
Host[Alive == 1, Age := 1L]

# Resource.Have
Host[Alive == 1, Resource.Have := 1]

# Resource.In, meaning that the starters start with a full belly
Host[Alive == 1, Resource.In := 1]

# Reproduction.Allocation
Host[Alive == 1, Reproduction.Allocation := reproduction.allocation]

# Immune.Allocation
Host[Alive == 1, Immune.Allocation := immune.allocation]
```

```
# Immune.Genotype, has been set when initializing, needs to be cleaned
Host[Alive == !1, Immune.Genotype := NA]

# Infection.Genote
Host[, Host.Infection.Genotype := NA]

# Size
Host[Alive == 1, Size := 1]

# Host.Generation
Host[Alive == 1, Host.Generation := 1L]

# Origin
  Host[Alive == 1, Origin := Host.Population]

###
# Here I start a vector that just contains if a host is alive or not, to circumvent all
# the lookups.
if(exists("Alive.Host")) {rm(Alive.Host, pos = ".GlobalEnv")}

Alive.Hosts <<- data.table(Is.Alive = Host[, Alive == 1])

# Here I initialize the parasite data.table structure
if(exists("Parasite")) {rm(Parasite, pos = ".GlobalEnv")}


Parasite <<- data.table(
  Alive = integer(preallocation.parasite),
  Parasite.Replicate = integer(preallocation.parasite),
  Time = integer(preallocation.parasite),
  Parasite.Population = integer(preallocation.parasite),

  Parasite.Infection.Genotype =
    factor(sample(c(1 : parasite.genotypes),
                  size = preallocation.parasite,
                  prob = rep(1 / parasite.genotypes, parasite.genotypes), replace = T),
           levels = c(1 : parasite.genotypes)),

  Attack.Host.TempID = integer(preallocation.parasite),
  Attack.Host.Genotype = factor(sample(c(1 : parasite.genotypes),
                  size = preallocation.parasite,
                  prob = rep(1 / parasite.genotypes, parasite.genotypes), replace = T),
           levels = c(1 : parasite.genotypes)),
  Success.Parasite.Infection.Genotype = factor(NA, levels = c(1 : parasite.genotypes)),
  Ingested = integer(preallocation.parasite),
  Age = integer(preallocation.parasite)
)


# Here I initialize the starter populations for the parasite Replicate
Parasite[, Parasite.Replicate :=
           c(rep(1 : replicates, each = sum(starting.population.sizes.parasite)),
             integer(preallocation.parasite -
```

```
                          sum(starting.population.sizes.parasite) * replicates))]

  # Alive variable, 1 = alive, 0 = not alive
  Parasite[Parasite.Replicate != 0, Alive := 1L]

  # Population
  Parasite[Alive == 1,
           Parasite.Population :=
             as.integer(rep(1 : number.populations.parasite,
                            times = starting.population.sizes.parasite)),
           by = Parasite.Replicate]

  # Age
  Parasite[Alive == 1, Age := 1L]

  # Infection.Genotype, has been set when initializing, needs to be cleaned
  Parasite[Alive != 1, Parasite.Infection.Genotype := NA]

  # Atack.Host.Rownumber
  Parasite[, Attack.Host.TempID := NA]

  # Atack.Host.Genotype
  Parasite[, Attack.Host.Genotype := NA]

  # Here I start a vector that just contains if a Parasite is alive or not, to circumvent
  # all the lookups
  if(exists("Alive.Parasite")) {rm(Alive.Parasite, pos = ".GlobalEnv")}

  Alive.Parasites <<- data.table(Is.Alive = Parasite[, Alive == 1], Is.Ingested = FALSE)

# Close the individual.creator.function.
}
```

Within agent parameters

When the first agents are created they need to be assigned trait values. Most within-agent parameters, like the resource allocation proportion or the old-age threshold can be set in the "Digital_Coevolution_Parameterspace.R" script. They define the life-history of the agents that are simulated. Depending on the settings in the "Digital_Coevolution_Parameterspace.R" script, the agents can for example be parametrized to behave more like an k-strategist or more like an r-strategist. If one for example changes the old age threshold and the reproduction allocation, one can vastly change the nature of the agents simulated. This amount of control over the behaviour of the agents allows the Digital_Coevolution simulation to be fine-tuned to the hypothesis that is currently examined.

```
###### Parameter space
# Internal dynamic parameters
# Host internal dynamic parameters
host.size <- "OFF"
age.threshold.host <- 30
resource.threshold.host <- 0.2
reproduction.threshold.host <- 2
reproduction.factor.host <- 4
reproduction.allocation <- 0.35
immune.allocation <- 0.35
```

```
# Parasite internal dynamic parameters
parasite.genotypes <- host.genotypes
age.threshold.parasite <- 60
reproduction.threshold.parasite <- 2
reproduction.factor.parasite <- 23

# This factor gives the per time unit growth of a infection, in percent
infection.growth.factor <- 1.15
```

Between agent parameters

The Digital_Coevolution simulation allows for a high level of control over the between agent parameters like the infection system or the migration pattern. This level of control might be excessive for most use cases, so the settings of between agent parameters has been split into two compatible methods, depending on the level of control needed.
If a simple level of control is enough, then the between agent parameters will mostly be set in the "Digital_Coevolution_User.R" script. There one can set the number and size of subpopulations, the number of host and parasite genotypes, the specificity of the infection system and the amount of migration of between host or parasite subpopulations. Those simple parameter settings are passed on to the "Digital_Coevolution_Parameterspace.R" script where the infection table and the migration matrix are then automatically initialized.
The default that is used for the infection table is symmetrical. Full specificity will create a perfect matching alleles type of infection where only matching genotypes of host and parasite agents can create an infection. Lower specificity settings will also allow non-matching host and parasite agents to establish an infection with the respective probability.
The default migration matrix is symmetrical as well. This means all subpopulations are connected with migration of equal strength. A migration setting of 0 will create no migrants, and a setting of 1 will randomly re-assign all individuals to any of the subpopulations each time-step. Settings between that will randomly assign a proportion of individuals to a new subpopulation each time-step. Those default settings allow for easy usability of the Digital_Coevolution simulation, while still allowing for more control if needed.

```
#############################################################
## Here you can adjust the parameters that influence the
## digital organisms themselves

# How many host populations should there be, and how large
# should they be? This doesn't directly set the number
# of individuals, but the resources that are available
# per population. More resources will lead to larger
# populations, but the exact number of individuals
# that will result depends on many things.
host.populations <- c(100,100)

# How many host genotypes should there be?
# Note that as default, each host genotype gets its own
# parasite genotype as well.
host.genotypes <- 5

# Which proportion of the host populations should
# randomly migrate each time step
host.migration <- 0

# Which proportion of the parasite populations
# should randomly migrate?
```

```
parasite.migration <- 1

# How genotype specific should the infection
# be in percent?
parasite.specificity <- 1

# How virulent should the parasites be?
# Virulence is implemented as percent resources withdrawn
# per time step, depending on infection size.
virulence <- 0.4

# How much random drift should there be?
# Defined as proportion per timestep.
# This proportion of the host and the parasite population
# will additionally randomly be killed each timestep.
# A value of zero adds no additional random drift.
random.drift <- 0
```

If the between agent parameters should be fine tuned in more detail, for example if a certain asymmetrical migration pattern is desired, that can be done directly in the "Digital_Coevolution_Parameterspace.R" script. The "Digital_Coevolution_Parameterspace.R" script is used downstream of the "Digital_Coevolution_User.R" script. Any parameter settings that are done in the "Digital_Coevolution_Parameterspace.R" script will therefore take precedence over the settings from the "Digital_Coevolution_User.R" script.
In the "Digital_Coevolution_Parameterspace.R" script the infection table can be set explicitly, which defines how likely an infection is per host-agent genotype by parasite-agent genotype combination. This allows the creation of asymmetrical infection patterns. Simply provide a customized infection table instead of the default infection table creation.
The same applies for the migration matrix of both the host and the parasite agents. You can provide the "Digital_Coevolution_Parameterspace.R" script with custom migration matrices for host agents or parasite agents instead of the default created migration matrix.

```
###### Parameter space
# Infection dynamic parameters, row is host, column is parasite
infection.table <- matrix((1 - parasite.specificity),
                          nrow = host.genotypes,
                          ncol = parasite.genotypes)
diag(infection.table) <- 1

# Resource parameters
# This is the modifier that can increase or decrease the variance in resource
# distribution. The higher the number the less variance. With value 1 its a normal
# poisson distribution.
resource.grain <- 10

##############################################################################
##############################################################################
# Here at the same time the number of populations and their size are set. For each
# population define explicitly the amount of resources it receives daily. Currently,
# population size settles at about half the resources given.
resources.host <- host.populations

###
number.populations.host <- length(resources.host)
populations.host <- c(1 : number.populations.host)
```

```
starting.population.sizes.host <- ceiling(resources.host / 2)

###
number.populations.parasite <- number.populations.host
populations.parasite <- c(1 : number.populations.parasite)
starting.population.sizes.parasite <- ceiling(resources.host * 10)

###
# External Population dynamic parameters:
migration.matrix.host <- matrix((host.migration / number.populations.host),
                                nrow = number.populations.host,
                                ncol = number.populations.host)

diag(migration.matrix.host) <-
  diag(migration.matrix.host) + (1 - host.migration)

migration.matrix.parasite <- matrix((parasite.migration / number.populations.parasite),
                                    nrow = number.populations.parasite,
                                    ncol = number.populations.parasite)

diag(migration.matrix.parasite) <-
  diag(migration.matrix.parasite) + (1 - parasite.migration)
```

Dead or alive?

The agents of the Digital_Coevolution simulation can be, much like biological individuals, either dead or alive. The data.table that contains the agents of the Digital_Coevolution simulation always has the same size for technical reasons (see section "data structure"). This means it contains both currently alive as well as currently dead individuals. Dead individuals are simply agents from past time-steps whose rows have not yet been recycled yet. In order to optimize the simulation in terms of computational efficiency, and because dead agents should not interact with alive ones, calculations should only be done with alive agents. This is possible by using the "Alive" variable, which can take the values 0 = dead and 1 = alive to subset the data.table. As a subsetting operation always is a comparison operation between the queried value and all values in the vector that is to be subsetted, it can become computationally demanding. The data.table package already has internal optimisation to make this type of look-up as fast as possible (Dowle and Srinivasan 2019), yet there are still further gains in speed to be had when the comparison operation is circumvented completely. This is especially valuable when a comparison is done repeatedly. To check whether or not an individual agent is still alive is one such operation which can be optimized. In the Digital_Coevolution simulation this is done by creating an external vector (or to be more precise, a one-column data.table) that has the same length as the number of rows of the data.table that contains the agents. This external vector contains the information indicating which agent is currently alive in logical form, alive = TRUE. It is essentially a copy of the trait "Alive" in logical form. This vector can now be used to subset the data.table that contains the agents to only the agents which are alive, without the necessity for a comparison operation. It does need to be carefully updated whenever an agent dies or gets born.
For the host agents, the "Alive.Hosts" vector is created as follows:

```
# Here I start a vector that just contains if a host is alive or not, to circumvent all
# the lookups.
Alive.Hosts <<- data.table(Is.Alive = Host[, Alive == 1])
```

And every time the aliveness status of an agent changes, it is updated as follows:

```
set(Alive.Hosts, j = "Is.Alive", value = Host[, Alive == 1])
```

And it is used as follows:

```
# Host migration function.
host.migration.function <- function(){
  Host[Alive.Hosts$Is.Alive,
       Host.Population :=
         base:::sample(1 : number.populations.host,
                       size = .N,
                       prob = migration.matrix.host[Host.Population[1], ],
                       replace = TRUE),
       by = list(Host.Population, Host.Replicate)]
  }
```

The simulation process

The dynamics wrapper

We are at the heart of the Digital_Coevolution simulation. All of the biological logic that was explained in part one of this document results in functions. Functions in R are collections of code that are defined under a common name and will be executed once the function is called. In the Digital_Coevolution simulation, each of the distinct within agent and between agent dynamics that were explained in part one is defined as a separate function. I have called those functions "dynamics functions" throughout the document. The migration function of the host for example is designed like this:

```
# Host migration function.
host.migration.function <- function(){
  Host[Alive.Hosts$Is.Alive,
       Host.Population :=
         base:::sample(1 : number.populations.host,
                       size = .N,
                       prob = migration.matrix.host[Host.Population[1], ],
                       replace = TRUE),
       by = list(Host.Population, Host.Replicate)]
}
```

Having all the different rules for dynamic behaviour of the agents defined as separate functions does allow to invoke them separately. That is important, as they will be executed consecutively in the simulation. The order in which the functions are called can have a big influence on the biological logic behind the Digital_Coevolution. The timing of migration for example is important. Does migration of parasite agents happen after the parasite reproduces but before there is a new round of hosts being exposed to parasites? Or will the parasites migrate after an exposure event? This simple question of timing will influence whether coevolution has a more local component or a more global component.
Nevertheless, every time-step all of the functions that govern the agents will have to be called once for a full agent life cycle. To achieve that, and to ensure consistency in the order of the calling of functions, they will be wrapped in another function that is called "dynamics.wrapper" in the simulation. All this dynamics wrapper does is simply calling the functions that define the agents behaviour one after another. One call of the dynamics.wrapper function is one time-step in the simulation. Changing the order of the functions in the dynamics.wrapper is a simple yet powerful way to change the behaviour of the Digital_Coevolution simulation.

```
# Here I will combine all the functions defined above into one wrapper function to be
# called. One call is one timestep. The order of the function can have an influence on
# the dynamics of the thing.
dynamics.wrapper <- function(){
  time.function()
  senescence.function()
  host.resource.function()
  infection.function()
  host.exposure.function()
  parasite.reproduction.function()
  metabolism.function()
  host.reproduction.function()
  host.migration.function()
  parasite.migration.function()
}
```

Obtaining Results

A key feature of the Digital_Coevolution simulation is that the dynamic behaviour of each individual host and parasite agent can be examined at each time-step for the duration of the simulation. This allow coevolutionary

dynamics to be analysed with great temporal resolution.
In order to do that there is a piece of code in the "Digital_Coevolution_Run.R" script that will save a complete copy of the host agent and parasite agent population to a file on the disc. This means we get a complete snapshot of every single agent, all its internal states like reproduction, infection and resources whenever we wish. This data extraction process is implemented using the fwrite function from the data.table package in append mode. This means that whenever it is invoked, it will take both the host and parasite agent population data.tables and save a copy to disk. Instead of creating a new file for every time a copy is saved, the append mode uses the same data file every time, and simply adds the new data at the bottom of it. This is where all the identifiers for time and replicate become useful, as they allow us to filter the resulting data set very precisely.
Long simulations (> 20'000 time-steps) with large populations (> 5000 resources) can create big data files (tens to hundreds of GB) when the state of the simulation is saved every time step. The "saving.interval" setting in the "Digital_Coevolution_Run.R" script allows to control the granularity of the data extraction process. It controls the interval between saving events as N time-steps. If the setting is left at 1, the default, then every time step will be saved. If you set another value, for example 10, then only every 10th time-steps will be saved. This decreases the size of the final output file by 10, but also decreases the resolution in time. It can also be used to potentially speed up the simulation, as writing to disk can take a lot of time. The fwrite function used is already optimized for speed, but if an older system with a hard drive instead of a solid state drive is used, it can nevertheless use up quite some time. For long simulations, it is worth it to consider if it is necessary to save every time-step, or if less data is sufficient as well. This feature is implemented in the "by =" argument in the "if" statement that wraps the fwrite in the code below.
There is also a toggle that allows you to get a summarized report instead of a raw copy of the host and parasite agent states. If that toggle is enabled, then instead of a raw copy, a summary report will be created at each reporting time step. That summary includes the number of host or parasite individuals per genotype per sub- or metapopulation and the number of infections. It lacks some of the details of the raw data, but is a much more convenient data set.

```
# Because it is a time forward simulation it is necessary to loop through the timesteps.
# We do that by calling the dynamics.wrapper function within a loop.
# The dynamics.wrapper contains all the functions that guide the dynamics of the
# individuals in each timestep (as for example the host.reproduction.function).
for(i in 1 : duration.days){
  dynamics.wrapper()
  # result saving
  if(i %in%
     c(1, seq(from = saving.intervall, to = duration.days, by = saving.intervall))){
    if(raw.results){
      fwrite(Host[Alive.Hosts$Is.Alive], file =
               paste(result.file.location,
                     result.file.name,
                     "_Host_",
                     run.date,
                     "_raw_",
                     ".csv",
                     sep = ""), append = TRUE)
      fwrite(Parasite[Alive.Parasites$Is.Alive], file =
               paste(result.file.location,
                     result.file.name,
                     "_Parasite_",
                     run.date,
                     "_raw_",
                     ".csv",
                     sep = ""), append = TRUE)
    }
```

```
  if(summarized.results) {
    temp.data.host <- copy(Host)
    temp.data.host[, Virulence := virulence]
    temp.data.host[, Popsize := host.populations[1]]
    temp.data.host[, Random.Drift := random.drift]
    temp.data.host[, Parasite.Connection := parasite.migration]
    temp.data.host[, Host.Connection := host.migration]
    temp.data.host[, Host.Time := Time]

    temp.data.host[, Host.Number.Individuals := .N,
                   by = list(Host.Time, Host.Replicate, Host.Population,
                             Immune.Genotype, Virulence, Popsize, Parasite.Connection,
                             Host.Connection)]
    temp.data.host[, Host.Population.Size := .N,
                   by = list(Host.Time, Host.Replicate, Host.Population, Virulence,
                             Popsize, Parasite.Connection, Host.Connection)]
    temp.data.host[, Host.Number.Individuals.Between := .N,
                   by = list(Host.Time, Host.Replicate, Immune.Genotype, Virulence,
                             Popsize, Parasite.Connection, Host.Connection)]
    temp.data.host[, Host.Population.Size.Between := .N,
                   by = list(Host.Time, Host.Replicate, Virulence, Popsize,
                             Parasite.Connection, Host.Connection)]
    temp.data.host[, Epidemic.Size.Within := sum(Infection.State),
                   by = list(Host.Time, Host.Replicate, Host.Population,
                             Immune.Genotype, Virulence, Popsize, Parasite.Connection,
                             Host.Connection)]
    temp.data.host[, Epidemic.Size.Total := sum(Infection.State),
                   by = list(Host.Time, Host.Replicate, Host.Population, Virulence,
                             Popsize, Parasite.Connection, Host.Connection)]

    #########
    temp.data.parasite <- copy(Parasite)
    temp.data.parasite[, Virulence := virulence]
    temp.data.parasite[, Popsize := host.populations[1]]
    temp.data.parasite[, Random.Drift := random.drift]
    temp.data.parasite[, Parasite.Connection := parasite.migration]
    temp.data.parasite[, Host.Connection := host.migration]
    temp.data.parasite[, Parasite.Time := Time]

    temp.data.parasite[, Parasite.Number.Individuals := .N,
                       by = list(Parasite.Time, Parasite.Replicate,
                                 Parasite.Population, Parasite.Infection.Genotype,
                                 Virulence, Popsize, Parasite.Connection,
                                 Host.Connection)]
    temp.data.parasite[, Parasite.Population.Size := .N,
                       by = list(Parasite.Time, Parasite.Replicate, Parasite.Population,
                                 Virulence, Popsize, Parasite.Connection,
                                 Host.Connection)]
    temp.data.parasite[, Parasite.Number.Individuals.Between := .N,
                       by = list(Parasite.Time, Parasite.Replicate,
                                 Parasite.Infection.Genotype, Virulence, Popsize,
                                 Parasite.Connection, Host.Connection)]
    temp.data.parasite[, Parasite.Population.Size.Between := .N,
```

```
                      by = list(Parasite.Time, Parasite.Replicate, Virulence, Popsize,
                                Parasite.Connection, Host.Connection)]

#########
temp.data.host[, Total.Parasite.Number.Individuals :=
                 Epidemic.Size.Within +
                 temp.data.parasite[
                   Parasite.Replicate == Host.Replicate[1] &
                     Parasite.Population == Host.Population[1] &
                     Parasite.Time == Host.Time[1] &
                     Parasite.Infection.Genotype == Immune.Genotype[1], .N],
               by = list(Host.Time, Host.Replicate, Host.Population,
                         Immune.Genotype, Virulence, Popsize, Parasite.Connection,
                         Host.Connection)]
temp.data.host[, Total.Parasite.Population.Size :=
                 Epidemic.Size.Total +
                 temp.data.parasite[
                   Parasite.Replicate == Host.Replicate[1] &
                     Parasite.Population == Host.Population[1] &
                     Parasite.Time == Host.Time[1], .N],
               by = list(Host.Time, Host.Replicate, Host.Population, Virulence,
                         Popsize, Parasite.Connection, Host.Connection)]

temp.data.host[, Total.Parasite.Number.Individuals.Between :=
                 Epidemic.Size.Within +
                 temp.data.parasite[
                   Parasite.Replicate == Host.Replicate[1] &
                     Parasite.Population == Host.Population[1] &
                     Parasite.Time == Host.Time[1] &
                     Parasite.Infection.Genotype == Immune.Genotype[1], .N],
               by = list(Host.Time, Host.Replicate, Immune.Genotype, Virulence,
                         Popsize, Parasite.Connection, Host.Connection)]
temp.data.host[, Total.Parasite.Population.Size.Between :=
                 Epidemic.Size.Total +
                 temp.data.parasite[
                   Parasite.Replicate == Host.Replicate[1] &
                     Parasite.Population == Host.Population[1] &
                     Parasite.Time == Host.Time[1], .N],
               by = list(Host.Time, Host.Replicate, Virulence, Popsize,
                         Parasite.Connection, Host.Connection)]

#########
fwrite(
  unique(temp.data.host[,
                        list(Host.Time, Host.Replicate, Host.Population,
                             Immune.Genotype, Virulence, Popsize, Random.Drift,
                             Parasite.Connection, Host.Connection,
                             Host.Number.Individuals, Host.Population.Size,
                             Host.Number.Individuals.Between,
                             Host.Population.Size.Between, Epidemic.Size.Within,
                             Epidemic.Size.Total, Total.Parasite.Number.Individuals,
                             Total.Parasite.Population.Size,
                             Total.Parasite.Number.Individuals.Between,
```

```
                                    Total.Parasite.Population.Size.Between, Origin)]),
        file = paste(
          result.file.location,
          result.file.name,
          "_Host_",
          run.date,
          "_summarized_",
          ".csv", sep = ""), append = TRUE)

      fwrite(
        unique(temp.data.parasite[,
                                  list(Parasite.Time, Parasite.Replicate,
                                       Parasite.Population, Parasite.Infection.Genotype,
                                       Virulence, Popsize, Random.Drift,
                                       Parasite.Connection, Host.Connection,
                                       Parasite.Number.Individuals,
                                       Parasite.Population.Size,
                                       Parasite.Number.Individuals.Between,
                                       Parasite.Population.Size.Between)]),
        file = paste(
          result.file.location,
          result.file.name,
          "_Parasite_",
          run.date,
          "_summarized_",
          ".csv", sep = ""), append = TRUE)

    }
  }
}
```

Time-forward simulation

The simulation is implemented as a time-forward simulation. This means that the simulation loops through discrete time-steps. Each time-step, all dynamics functions are called once and the host and parasite agent data.tables get updated. Then the next time step is calculated with the updated data.table. This is necessary technically, as the simulation contains a number of stochastic elements. The process is similar to the creation of a Markov chain.
Time-forward simulations also allow for a very fine-grained analysis of temporal dynamics of host and parasite coevolution, as every time-step the status of the simulation can be examined fully. This is especially valuable for systems where at least part of the dynamics have a time-shift property.
The time-forward property of the Digital_Coevolution is implemented as a for-loop. A for loop in R calls its arguments for N times consecutively. The number of times over which the R for loop iterates is the number of time-steps that the simulation is run over. This scales roughly linearly, as every time-step the same functions get evaluated.
This is what runs the Digital_Coevolution simulation.

```
# Now we run the simulation
# Because it is a time forward simulation it is necessary to loop through the timesteps.
for(i in 1 : duration.days){
  dynamics.wrapper()
}
```

Time

Because the simulation is a time-forward simulation, and the result file will contain many agent states from different time points, each agent needs a time-stamp as identifier. This is implemented as a integer counter that increases its value each time-step. It is particularly useful when filtering the results.

```
Host[, Time := Time + 1L]
Parasite[, Time := Time + 1L]
```

Conclusions

The Digital_Coevolution simulation is a novel agent-based simulation that implements detailed life histories of host and parasite agents. It can be used to test hypotheses and create theoretical expectations for natural systems. It is positioned to be close to a digital organism, in that the agents have detailed, individual-life histories. It lacks the self-modificability that is inherent in many other digital organisms but it includes emergent properties that have not been programmed and are a result of individual interactions. Such emergent properties can lead to unanticipated yet insightful results (Lehman, Clune, and Misevic 2018; Lehman et al. 2019). Agent-based simulations and digital organisms are an active field of research and contribute to a wide array of topics from the origin of life (C G et al. 2017), evolution in small populations (LaBar and Adami 2016, 2017), metapopulations (Fortuna et al. 2013; Boëte, Seston, and Legros 2019), Red Queen Hypothesis (Kidner and Moritz 2015), diversity (Zaman, Devangam, and Ofria 2011), mutational robustness (Lenski et al. 1999). They are not only used in biology but have for example been applied in crowd control and economics (Bonabeau 2002). They should not be seen as an alternative to analytic models but as complementary (Gräbner et al. 2019). Researching the same questions with several different approaches will only increase the robustness of findings.

The Digital_Coevolution simulation is a work in progress and shows potential to be further modified. The modular approach with the "dynamics functions" means that additional properties of host and parasite agent interactions can be included in the simulation without altering existing code. A few future additions are already anticipated. Environmental conditions in subpopulations that affect host and parasite competitive rankings could be implemented to allow research on GxGxE interactions in a metapopulation context. Allowing for variance in the inheritance of life-history traits like reproduction allocation would open the simulation to research into optimization of life-history traits under different conditions. Adding further agent types would vastly improve the realism of the simulation and allow for different type of interactions between agents, for example parasites with complex life-cycles (multi-host life cycles) or for predator prey dynamics.

The Digital_Coevolution simulation can not only be used for research purposes but also for teaching purposes. It can showcase how individual level rules and behaviours can lead to population and ecosystem properties. Many students and researchers also are already familiar with R, which means that no new program or syntax has to be learned before the Digital_Coevolution simulation can be used. The approachability of R also means that advanced users can modify the simulation themselves, which makes the Digital_Coevolution simulation a highly flexible tool. Further integration with R towards a CRAN package and the implementation of a graphical user interface for parameter settings and visualisations via the shiny package (Chang et al. 2020) could increase usability of the Digital_Coevolution simulation for research and teaching.

Final remarks

Writing the Digital_Coevolution simulation has been great fun, and probably one of the most instructive experiences that I ever had. There is some truth in the saying that you have not understood something completely before you can not build it yourself. Trying to build this agent-based simulation and reflecting how those agents have to behave lead me to have some great realisations on how complex and fascinating life is, and how little we actually understand. It made me read far more papers in a far broader array of topics than I would have initially imagined, and I would not miss a bit of it.
The Digital_Coevolution simulation is a purpose-built tool that I have created to answer some questions in host-parasite coevolution. By training I am a biologist, so writing this simulation has given me great exposure to writing functional and fast R code. The source code of this simulation is provided on GitHub and I invite people to use it, to collaborate, and to share alterations of it.
Last but not least: Have fun, explore, stay curious.

Cheers
Robert

References

Allaire, JJ, Yihui Xie, Jonathan McPherson, Javier Luraschi, Kevin Ushey, Aron Atkins, Hadley Wickham, Joe Cheng, Winston Chang, and Winston Iannone. 2020. "R Markdown: Dynamic Documents for R."

Boëte, Christophe, Morgan Seston, and Mathieu Legros. 2019. "Strategies of Host Resistance to Pathogens in Spatially Structured Populations: An Agent-Based Evaluation." *Theoretical Population Biology* 130 (December): 170–81. https://doi.org/10.1016/j.tpb.2019.07.014.

Bonabeau, E. 2002. "Agent-Based Modeling: Methods and Techniques for Simulating Human Systems." *Proceedings of the National Academy of Sciences* 99 (Supplement 3): 7280–7. https://doi.org/10.1073/pnas.082080899.

Boots, M., and M. Mealor. 2007. "Local Interactions Select for Lower Pathogen Infectivity." *Science* 315 (5816): 1284–6. https://doi.org/10.1126/science.1137126.

Boots, M., and A. Sasaki. 1999. "'Small Worlds' and the Evolution of Virulence: Infection Occurs Locally and at a Distance." *Proceedings of the Royal Society of London. Series B: Biological Sciences* 266 (1432): 1933–8. https://doi.org/10.1098/rspb.1999.0869.

Box, G.E.P. 1976. "Science and Statistics." *Journal of the American Statistical Association* 71 (356): 791–99.

———. 1979. "Robustness in the Strategy of Scientific Model Building." In *Robustness in Statistics*, 201–36. Elsevier. https://doi.org/10.1016/B978-0-12-438150-6.50018-2.

C G, Nitash, Thomas LaBar, Arend Hintze, and Christoph Adami. 2017. "Origin of Life in a Digital Microcosm." *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 375 (2109): 20160350. https://doi.org/10.1098/rsta.2016.0350.

Chang, Winston, Joe Cheng, JJ Allaire, and Yihui Xie. 2020. "Shiny: Web Application Framework for R."

Clarke, B C. 1979. "The Evolution of Genetic Diversity." *Proceedings of the Royal Society B: Biological Sciences* 205 (1161): 453–74.

Clarke, Bryan. 1976. "The Ecological Genetics of Host-Parasite Relationships." In *Genetic Aspects of Host-Parasite Relationships*. London: Blackwell.

Dowle, Matt, and Arun Srinivasan. 2019. "Data.Table: Extension of 'data.Frame'."

Fortuna, Miguel A., Luis Zaman, Aaron P. Wagner, and Charles Ofria. 2013. "Evolving Digital Ecological Networks." Edited by Shoshana Wodak. *PLoS Computational Biology* 9 (3): e1002928. https://doi.org/10.1371/journal.pcbi.1002928.

Gandon, S., and Y. Michalakis. 2002. "Local Adaptation, Evolutionary Potential and Host-Parasite Coevolution: Interactions Between Migration, Mutation, Population Size and Generation Time: Local Adaptation and Coevolution." *Journal of Evolutionary Biology* 15 (3): 451–62. https://doi.org/10.1046/j.1420-9101.2002.00402.x.

Gandon, Sylvain. 1996. "Local Adaptation and Gene-for-Gene Coevolution in a Metapopulation Model." *Proceedings of the Royal Society of London. Series B: Biological Sciences* 263 (1373): 1003–9. https://doi.org/10.1098/rspb.1996.0148.

———. 2002. "Local Adaptation and the Geometry of Host-Parasite Coevolution." *Ecology Letters* 5 (2): 246–56. https://doi.org/10.1046/j.1461-0248.2002.00305.x.

Gräbner, Claudius, Catherine S. E. Bale, Bernardo Alves Furtado, Brais Alvarez-Pereira, James E. Gentile, Heath Henderson, and Francesca Lipari. 2019. "Getting the Best of Both Worlds? Developing Complementary Equation-Based and Agent-Based Models." *Computational Economics* 53 (2): 763–82. https://doi.org/10.1007/s10614-017-9763-8.

Haldane, J. B, S. 1949. "Disease and Evolution." *La Ricerca Scientifica Supplemento*, no. 19.

Haldane, J. B, S., and S. D. Jayakar. 1963. "Polymorphism Due to Selection Depending on the Composition of a Population." *Journal of Genetics* 3 (58): 318–23.

Hamilton, W. D., R. Axelrod, and R. Tanese. 1990. "Sexual Reproduction as an Adaptation to Resist Parasites (a Review)." *Proceedings of the National Academy of Sciences* 87 (9): 3566–73. https://doi.org/10.1073/pnas.87.9.3566.

Hamilton, William D. 1980. "Sex Versus Non-Sex Versus Parasite." *Oikos* 35 (2): 282. https://doi.org/10.2307/3544435.

Houston, Alasdair I., Philip A. Stephens, Ian L. Boyd, Karin C. Harding, and John M. McNamara. 2007. "Capital or Income Breeding? A Theoretical Model of Female Reproductive Strategies." *Behavioral Ecology* 18 (1): 241–50. https://doi.org/10.1093/beheco/arl080.

Jaenike, John. 1978. "An Hypothesis to Account for the Maintenance of Sex Within Populations." *Evolutionary Theory*, no. 3: 191–94.

Judson, Olivia P. 1995. "Preserving Genes; a Model of the Maintenance of Genetic Variation in a Metapopulation Under Frequency-Dependent Selection." *Genet. Res.*, 175–91.

———. 1997. "A Model of Asexuality and Clonal Diversity: Cloning the Red Queen." *Journal of Theoretical Biology* 186 (1): 33–40. https://doi.org/10.1006/jtbi.1996.0339.

Kidner, J., and Robin F. A. Moritz. 2015. "Host-Parasite Evolution in Male-Haploid Hosts: An Individual Based Network Model." *Evolutionary Ecology* 29 (1): 93–105. https://doi.org/10.1007/s10682-014-9722-y.

LaBar, Thomas, and Christoph Adami. 2016. "Different Evolutionary Paths to Complexity for Small and Large Populations of Digital Organisms." Edited by Sergei Maslov. *PLOS Computational Biology* 12 (12): e1005066. https://doi.org/10.1371/journal.pcbi.1005066.

———. 2017. "Evolution of Drift Robustness in Small Populations." *Nature Communications* 8 (1): 1012. https://doi.org/10.1038/s41467-017-01003-7.

Ladle, Richard J., Rufus A. Johnstone, and Olivia P. Judson. 1993. "Coevolutionary Dynamics of Sex in a Metapopulation: Escaping the Red Queen." *Proceedings of the Royal Society of London. Series B: Biological Sciences* 253 (1337): 155–60. https://doi.org/10.1098/rspb.1993.0096.

Lehman, Joel, Jeff Clune, and Dusan Misevic. 2018. "The Surprising Creativity of Digital Evolution." In *The 2018 Conference on Artificial Life*, 55–56. Tokyo, Japan: MIT Press. https://doi.org/10.1162/isal_a_00016.

Lehman, Joel, Jeff Clune, Dusan Misevic, Christoph Adami, Lee Altenberg, Julie Beaulieu, Peter J. Bentley, et al. 2019. "The Surprising Creativity of Digital Evolution: A Collection of Anecdotes from the Evolutionary Computation and Artificial Life Research Communities." *arXiv:1803.03453 [Cs]*, November. http://arxiv.org/abs/1803.03453.

Lenski, Richard E., Charles Ofria, Travis C. Collier, and Christoph Adami. 1999. "Genome Complexity, Robustness and Genetic Interactions in Digital Organisms." *Nature* 400 (6745): 661–64. https://doi.org/10.1038/23245.

Lion, S., and S. Gandon. 2015. "Evolution of Spatially Structured Host-Parasite Interactions." *Journal of Evolutionary Biology* 28 (1): 10–28. https://doi.org/10.1111/jeb.12551.

McKinley, P., B.H.C. Cheng, C. Ofria, D. Knoester, B. Beckmann, and H. Goldsby. 2008. "Harnessing Digital Evolution." *Computer* 41 (1): 54–63. https://doi.org/10.1109/MC.2008.17.

Ofria, Charles, and Claus O. Wilke. 2004. "Avida; A Software Platform for Research in Computational Evolutionary Biology." *Artificial Life*, no. 10: 129–229.

Oosterhout, Cock, Domino A. Joyce, Stephen M. Cummings, Jonatan Blais, Nicola J. Barson, Indar W. Ramnarine, Ryan S. Mohammed, Nadia Persad, and Joanne Cable. 2006. "BALANCING SELECTION, RANDOM GENETIC DRIFT, AND GENETIC VARIATION AT THE MAJOR HISTOCOMPATIBILITY

COMPLEX IN TWO WILD POPULATIONS OF GUPPIES (POECILIA RETICULATA)." *Evolution* 60 (12): 2562–74. https://doi.org/10.1111/j.0014-3820.2006.tb01890.x.

Stephens, Philip A., Ian L. Boyd, John M. McNamara, and Alasdair I. Houston. 2009. "Capital Breeding and Income Breeding: Their Meaning, Measurement, and Worth." *Ecology* 90 (8): 2057–67.

Team, R Core. 2020. "R: A Language and Environment for Statistical Computing." Vienna, Austia: R Foundation for Statistical Computing.

Team, RStudio. 2019. "RStudio: Integrated Development Environment for R." Boston, MA: RStudio, Inc.

Xie, Yihui, J.J. Allaire, and Garret Grolemund. 2018. *R Markdown: The Definitive Guide*. Chapman and Hall / CRC.

Zaman, Luis, Suhas Devangam, and Charles Ofria. 2011. "Rapid Host-Parasite Coevolution Drives the Production and Maintenance of Diversity in Digital Organisms." In *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation - GECCO '11*, 219. Dublin, Ireland: ACM Press. https://doi.org/10.1145/2001576.2001607.