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 was supported by the adaptation to a changing environment center (ACE) at ETH. I would like to thank Jukka Jokela for his generous support and his inspiring supervision. I would like to thank Kyra Gähwiler, Nadine Tardent, Oliver Dressler and Felix Moerman for helpful comments on the manuscript.

Data accessibility: The simulation is available on GitHub.

Abstract

Host-parasite coevolution is highly context-dependent. Population size, demography, metapopulation structure and the details of both the hosts and the parasites life-histories influence the process of coevolution.
This context-dependency is difficult to be completely accounted for in empirical systems. Agent-based simulations can be used to implement and control multiple aspects of host-parasite coevolution while retaining a degree of "realism". Such simulations can be used for hypothesis testing and to generate expectations for empirical systems.
Here I present the "Digital_Coevolution" simulation, a novel, highly flexible, agent-based, time-forward simulation that I created to explore questions in negative frequency-dependent host-parasite coevolution. It allows the simulation of detailed host and parasite agent life-histories based on resource allocation trade-offs, their population structure, demography and migration patterns. Different infection systems anywhere between gene-for-gene to matching-allele models can be implemented. This makes the "Digital_Coevolution" simulation a powerful tool for research and teaching on the context-dependency of host-parasite coevolution.

Introduction

The growth of a host population and its infection with parasites is a highly dynamic, 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 their 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 and infectious is the parasite? How many parasites were 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 environmental context beneficial for the host or the parasite? These and many more factors can determine the success of an infection.

The outcome of host-parasite interactions 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 larger 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 higher 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 et al. 1993, Lion and Gandon 2015) and influences the genetic structure of both the host and the parasite (Judson 1995, 1997). The number of migrants between subpopulations and the migration distance can, for example, influence virulence evolution (Boots and Sasaki 1999, Boots and Mealor 2007), can enable local adaptation of the parasite on the host (Gandon 1996, 2002, Gandon and Michalakis 2002) or it 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.
Establishing highly controlled laboratory populations is challenging as well. Getting the conditions right to sustain parasite infection in order to be able to do experiments can sometimes absorb more time than the actual experiment itself (personal observation).
Simulations provide an alternative to these challenges. They can present a way to implement theoretical expectations and biological assumptions for difficult-to-manipulate systems. Simulations can help 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. 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 et al. 1990). More recent developments aim to understand the process of evolution itself by using "digital organisms", like for example the AVIDA platform (Ofria and Wilke 2004, McKinley et al. 2008). The power and capability 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 include plausible details of the hosts and the parasites life cycles into a simulation in order to increase realism. More detailed simulations can potentially give rise to more precise predictions. Similarly, one can draw more information from a more detailed map.

One attempt to capture realism is by the use of agent-based models. 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 population and metapopulation behaviour, then result from the sum of individual level interactions (for example see (Lenski et al. 1999), (Boëte et al. 2019) or (Bonabeau 2002)).

The simulation that is presented here, called the "Digital_Coevolution" simulation, belongs to the category of agent-based models. It simulates detailed host and parasite agents as well as their interactions. Population level behaviour then emerges from the sum of individual-level interactions in groups of these agents.
The aim of the Digital_Coevolution simulation was to create a tool to study how metapopulation structure influences coevolutionary dynamics between hosts and parasites. Due to the high 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 of and the reasoning behind the simulation and should enable the reader to thoroughly understand and use it. The first part describes the "biological logic" behind the code. It outlines the functions that govern the behaviour of the agents in the simulation. The second part then describes the implementation of the Digital_Coevolution simulation, 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 et al. 2018, Allaire et al. 2020) in R (Team 2020) using R Studio (Team 2019) as IDE. It consists of paragraphs of text explaining the simulation that are immediately followed by chunks of the according, commented R code. The text and the comments in the code do show substantial overlap, trying to reinforce a thorough understanding of what the simulation does. The code is not complete though and serves only as demonstration. 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 (Dünner 2020).

Part One: Biology of the Digital_Coevolution simulation

Overview

The Digital_Coevolution simulation models individual host and parasite agents in detail. It has instructions for within-agent life-history as well as for between-agent behaviour. When groups of those agents are simulated, population level behaviour emerges. The simulation is implemented in a time-forward fashion. 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 of the dynamics that are evaluated for each individual agent at each time-step. One time-step can be thought of as one "day" in the life of an agent. Not all events do necessarily take place every time-step and nearly all are context-dependent. Not all agents will experience the same events each time-step.

The agents that make up the Digital_Coevolution simulation have internal resource-allocation trade-offs among fitness-correlated life-history traits (Stearns 1980, Noordwijk and Jong 1986, Perrin and Sibly 1993). Each host agent has a distinct immune genotype and a set of life-history traits that it 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 depends 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 income breeding (Sibly and Calow 1984, Houston et al. 2007, Stephens et al. 2009) after a maturation threshold. Each agent has to save up resources until the maturation threshold is reached after which it can start reproducing. Each agent can reproduce several times during its lifetime, if resources permit it. 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). 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 host and parasite genotype matching and the number of parasite agents ingested. All types of genotype-specific infections can be implemented via an infection matrix. Infections can only be established by one parasite genotype at a time, there are no co-infections possible. Each host individual can potentially ingest several parasite 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 no longer available to the host. Virulence is thus implemented as a resource transfer from the host agent towards the parasite 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. 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 from the fat storage of the host.

Parasite agent reproduction is implemented with the same life-history strategy as that of host agents. Parasite reproduction only happens within infected host agents, with the infection acting 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 obligatorily have to infect a host. As soon as an infection has stolen enough resources from the host mature, it will produce a clutch of parasite offspring that is 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 this way is dependent on the parasite infection size within the host agent, which itself is dependent on the age of the infection and the host size. Mature infections in large host individuals can hence have a significant influence on the parasite population.

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. Host 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. Nevertheless it is 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. Therefore 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-history 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, for example zooplankters like *Daphnia* or *Cyclops*. Thinking of the agents as zooplankters can help with imagining some of the biology that has been implemented. Host agents are simply called hosts in the code of the simulation.
During the simulation each agent moves through different life-history stages, struggles for resources and is potentially exposed to parasites. Those parasites are horizontally transmitted, passively dispersed, obligate parasites, which might best be imagined as either bacterial or fungal pathogens, for example microsporidians. Parasite agents are simply called parasites in the code of the simulation.
The following part gives an overview over the different functions, called "dynamics functions", that each govern the behaviour of a different life-history trait. Each of the "dynamics functions" is 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 during 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 ingested parasite spores are directly linked. A larger agent will be exposed to more parasite spores per time-step than a small one as it has a higher feeding rate.
A larger organism will, on the other hand, also be able to produce more offspring per time-step, as it can acquire resources faster and the production of one offspring always costs the same amount of resources.

Size is not implemented as one single "dynamics function", but as a trait (of an agent) that is influencing many "dynamics functions". Size itself is influenced in the metabolism function when resources are allocated between different life-history processes.
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 into growth declines negatively proportional to numerator 2 with age $(2/AGE)$. Numerator 2 causes the resources that are available at the first time-step for growth to be counted twice. This was done to create a pronounced infant growth boost and a juvenile phase without reproduction (see section "Host metabolism").
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 for comparing 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 agent 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 1 each time-step. There is an old-age cut-off, creating senescence, implemented as a threshold value for both hosts and parasite agents. Once an agent reaches this threshold, it is culled and removed from the simulation.

```
# Takes the Age vector and adds 1 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 these 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 also have the ability to 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 the agents resource allocation scheme. Any resources that are 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 the squared 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 time-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 cope with 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 in the "Resource.Work" container.

```
# 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]

# The resource.have, so basically the fat content, is size dependant.
# As larger individuals should be able to proportionally store more fat, a qubic increase
# instead of a linear increase with size might 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 time-step, 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 (Stearns 1980, Noordwijk and Jong 1986, Perrin and Sibly 1993). 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 summarizes 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 that need to be 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 anew.
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 and changes with age. The same formula that has modified the investment into growth of an individual is also applied here. Younger individuals invest less into reproduction with proportion $2/AGE$. Numerator 2 creates a negative value in the "Reproduction.Have" container at the first time-step that has to be neutralized by subsequent investment. This means that the first batch of offspring costs more resources than following batches. Reproduction allocation was implemented this way to create a distinct juvenile phase without reproduction (see section "Size").
Finally, a host will invest into growth. Depending on the age of the individual, a certain proportion is invested into growth (increasing the value of the "Size" parameter). This proportion is declining proportional to age, so that younger individuals invest proportionally more into growth instead of reproduction (see section "Size"). 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 build-up.
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.
# 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

Each host individual first must accumulate resources until it reaches an adjustable threshold in order to mature. After it has reached this threshold, it reproduces as a iteroparous clonal income breeder (Sibly and Calow 1984, Houston et al. 2007, Stephens et al. 2009). The number of offspring produced per agent per batch is dependent on the resources invested in reproduction and an adjustable multiplicative reproduction factor. It is also dependent on the size of the individual, as larger individuals can obtain more resources which then can be invested into reproduction.

Reproduction is implemented in the Digital_Coevolution simulation by first calculating the identities of host agents that have matured. Those host agents are then copied multiple times. The amount of times they are copied is dependent on the amount of resources each individual had accumulated in excess of the maturation threshold and the multiplicative reproduction factor.
The life-history traits are heritable on the "genotype-level" 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, the last immune state of the parent agent and the population assignment of the parent agent. Other traits of an agent are dependent on its individual background and are not heritable. Those traits are reset to the starting conditions in the offspring. Those traits include age, size, infection state and all resource traits ("Resource.In", "Resource.Have", "Resource.Work"). The parent agent remains in the population and looses the resources it has used for reproduction.

```
# Calculate the number of reproducing individuals and how many offspring each
# individual produces.
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 within a host agent ("Infection.Size"). The size of an infection defines how many resources are withdrawn from the hosts resource budget ("Resource.Work") towards the infection. 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 infections 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 are withdrawn by the infection are used by the parasite to reproduce. Resources stolen by the parasite 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 in the same way as that of the host. Parasite reproduction only happens within infected host agents, with the infection acting as the reproducing unit. This means that the parasite infection within the host agent must accumulate resources until an adjustable threshold value is reached and the infection matures. After that, resources stolen from the host can be used for iteroparous clonal income breeding. As larger infections accumulate resources faster, larger infections will produce more

offspring parasite agents per clutch.

The implementation of parasite reproduction is identical to that of host reproduction. The amount of resources that an infection has stolen from the host individual is checked against the parasite maturation threshold to calculate which infection will have a 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 reset 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. 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 medium and a few zooplankters in it. Environmental conditions in those glass jars are fairly stable and feeding happens once a day. 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 also affects interactions with parasite agents, as they are 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 a batch of resources per population that renew every time-step. The host agents then compete for a portion of those resources. 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 individual 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, which equals 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 size 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 host and parasite genotypes. This can be specified so that the resulting infection model ranges anywhere between a gene-for-gene model to a matching-allele type infection model. Most simulation models of host-parasite coevolution are run as a perfect matching-allele 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 the most important part of the simulation. Here the host agents are exposed to parasites. The exposure within populations is completely random, i.e. there is no host seeking behaviour 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 or the infection success of the parasite agent. 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 a host agent, which is dependent on the size of the host population relative to the available resources (is there over- or underfeeding?). Then each ingested parasite agent gets assigned the specific host agent by which it has been ingested. The assignment is dependent on the size of an individual host agent relative to the summed-up size of the host population. This is because the feeding rate of host agents is directly linked to host agents 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. 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 spore 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$Is.Ingested vector for faster look-ups.
Alive.Parasites[Alive.Parasites$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 calculates 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 is 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 a migration pattern to create any type of metapopulation. The resources 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 are 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 zooplankton, 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. Accordingly can host and parasite agents in the Digital_Coevolution simulation migrate between different host and parasite subpopulations. This migration takes place once every time-step (once a day). Like in an experiment, migration can happen at different rates between different populations. Last but not least, those subpopulations can also be of different sizes, i.e. 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 one population another, both for the host and the parasite. This allows for the implementation of arbitrary metapopulation structures ranging 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 migrates substantially more. This makes the Digital_Coevolution simulation a valuable tool to investigate questions of host-parasite coevolution in metapopulations.

```
# 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 parasites from one population and randomly assigning it to a new population. Migration is thus completely random and independent of any parasite properties. The migration matrix defines the probabilities for each possible migration path.

```
# Parasite migration
# Here a random subset of each parasite population is picked
# and moved to another population. Size of subset and probability
# of destination population are defined in the migration matrix.
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 subset of the host population is randomly assigned to a new population according to the probabilities from the migration matrix.

```
# Host migration
# Here a random subset of each host population is picked
# and moved to another population. Size of subset and probability
# of destination population are defined in 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. It specifies how an agent actually is simulated, how time is implemented and 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 (Dowle and Srinivasan 2019). A data.table is also a 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 represents one trait, for example the immune genotype, the reproduction allocation proportion, the amount of resources already saved for reproduction or the infection state.
The whole data.table therefore comprises the state of all traits of all agents at a certain point in time. Every time-step each column is updated according to the "dynamics functions" that have been described in part one of this document. This utilizes R's vectorized structure whereby the same operation can be applied to all elements of a column simultaneously in a computationally efficient way. At specified intervals a copy of the data.table is saved, allowing 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 as well as 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 characteristics of agents that are 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. 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.

Setup of the Digital_Coevolution simulation

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 represented as tables, with each individual agent corresponding to one row. Reading out a column of this table 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 size will vary throughout the simulation. This means that the number of rows in the data.table will vary throughout the simulation as well. Having data.tables (or worse, data.frames) with varying numbers of rows within a loop is problematic in R, as R automatically creates a copy of the old data.table whenever the number of rows changes. This can slow the code down compared to loops where a data.table 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 of 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. See below for the first two rows of the host data.table at the initializing state of the simulation.

The Digital_Coevolution simulation relies heavily on the data.table package and hence its syntax (Dowle and Srinivasan 2019). The data.table syntax is analogous to SQL and has three elements: Where, order by / select, and update group / by.

```
##    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               4
## 2:             1                   0.35              0.35               1
##    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 invoked when the simulation is started. The "individual.creator" function creates two data.tables, one for the host and one for the parasite.

```
#########################################################################
# Individual.creator.function. Here the host and parasite agents are
# created upon initializing of the simulation.

# Open function.
individual.creator.function <- function(){

# In this step all the data.table space that might be necessary is preallocated
# in an attempt to speed up the simulation.
  preallocation.margin <- 10

  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")}

# Here the data.table of the host is created.
```

```
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 the starter populations are initialized with values.
# Replicate
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.Genotype
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 a vector that just contains if a host is alive or not is started
# to circumvent all the look-ups.
if(exists("Alive.Host")) {rm(Alive.Host, pos = ".GlobalEnv")}

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

# Here the parasite data.table structure is initialized.
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 the starter populations for the parasite are filled with values.
  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]

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

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

  # Here a vector that just contains if a parasite is alive or not is started
  # to circumvent all the look-ups.
  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 a k-strategist or more like a r-strategist. If one for example changes the old-age threshold and the reproductive allocation, that can vastly change the nature of the agents simulated. This high amount of control over the behaviour of the agents allows the Digital_Coevolution simulation to be fine-tuned to match a variety of empirical systems.

```
# 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-tep growth of an 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 is reached by using look-up tables that specify the outcome of interactions at the per-combination level.
The infection table contains for all combinations of host and parasite genotypes the infection specificity of that interaction. A value of 0 leads to no infection, a value of 1 means full infection potential. Values in between are a reduced infection specificity. A value of 0.5, for example, means that this host and parasite genotype combination has half the infection probability of a fully specific combination. The infection table is implemented as a full matrix, so it allows for the specification of asymmetrical infection patterns, while the default setting is symmetrical.
The migration matrix is implemented similarly. 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 0 and 1 will randomly assign a proportion of individuals to a new subpopulation. The default migration matrix is also symmetrical. This means all subpopulations are connected with migration of equal strength.

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

###
# Migration matrix
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 dead individuals. Dead individuals are simply agents from past time-steps whose rows have not yet been recycled. 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 made 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 of the queried value to 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 further optimization in execution speed are possible when the comparison operation is circumvented altogether. This is especially valuable when a comparison is done repeatedly.
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 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.

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

# The Alive.Hosts vector is updated by the set function after a host has been born or died.
set(Alive.Hosts, j = "Is.Alive", value = Host[, Alive == 1])

# In the host migration function the Alive.Hosts vector is used to subset the data.table
# to only the alive hosts.
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

The dynamics wrapper is 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- and between-agent dynamics that were explained in part one is defined as a separate function. These functions are called "dynamics functions" throughout the document.
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 invoked can have a big influence on the "biological logic" of the Digital_Coevolution simulation. As an example, the timing of migration 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.
Still, every time-step all the functions that govern the behaviour of the agents have to be called once in order to complete the full life-cycle of an agent. To achieve that, and to ensure consistency in the order of function calling, they are wrapped in another function that is called "dynamics.wrapper". All this "dynamics.wrapper" does, is simply calling the "dynamics functions" 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 all the functions defined above are combined into a wrapper function.
# One call is one time-step. The order of the functions can have an influence on
# the dynamics of the simulation.
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 of the simulation. This allows coevolutionary dynamics to be analysed with great temporal resolution.

There is a piece of code in the "Digital_Coevolution_Run.R" script that saves a complete copy of the host and parasite agent population to an output file at specified intervals. This means that a complete snapshot of every single agent with all its internal states like reproduction, infection and resources, gets saved.
The "saving.interval" setting in the "Digital_Coevolution_Run.R" script allows to control the frequency of this data extraction process in time-steps. With a value of 1 every time-step is saved, otherwise every N-th time-step. For long simulations it is worth to consider if saving every time-step is necessary or if less data is sufficient as well.

In the "Digital_Coevolution_Run.R" script there is also an option that allows to get a summarized report instead of a raw copy of the host and parasite agent states. That summary includes the number of host or parasite individuals per genotype per sub- and 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 time-steps.
# This is done by calling the dynamics.wrapper function within a loop.

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)
    }
# This part is only invoked if the option for a summarized report is enabled.
    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-stamp

Because the Digital_Coevolution 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 an identifier. This is implemented as an integer counter that increases its value each time-step. It is particularly useful for filtering the results.

```
# Time-stamp, counts the number of time-steps that the simulation
# has been looped over.
Host[, Time := Time + 1L]
Parasite[, Time := Time + 1L]
```

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 are 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. This 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 fully examined. 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 for-loop iterates is the number of time-steps that the simulation is run for. This is essentially what runs the Digital_Coevolution simulation.

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

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, insofar as the agents have detailed, individual life-histories. Compared to many other digital organisms, it lacks the self-modificability, 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 et al. 2018, 2019).

Agent-based simulations and digital organisms are widely used in a diverse array of research topics, such as the origin of life (C G et al. 2017), evolution in small populations (LaBar and Adami 2016, 2017), metapopulations (Fortuna et al. 2013, Boëte et al. 2019), Red Queen Hypothesis (Kidner and Moritz 2015), diversity (Zaman et al. 2011) and mutational robustness (Lenski et al. 1999). They are not only used in biology but have, for example, also been applied in crowd control and economics (Bonabeau 2002). Agent-based models should not be seen as an alternative to analytic models but as a complement (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" enables that additional properties of host and parasite agent interactions can easily 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 genotype-by-genotype-by-environment interactions in a metapopulation context. Allowing for variance in the inheritance of life-history traits, like reproduction allocation, would open the simulation to research on optimization of life-history traits under different conditions. Adding further agent types, for example predators, would vastly improve the realism of the simulation and allow for different type of interactions between agents like parasites with complex life-cycles (multi-host life-cycles) or predator prey dynamics.

The Digital_Coevolution simulation can not only be used for research purposes but also for teaching. It can showcase how individual level rules and behaviours can lead to population and ecosystem properties. As many students and researchers are already familiar with R, 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 via the *shiny* package (Chang et al. 2020) could increase usability of the Digital_Coevolution simulation for research and teaching.

References

Allaire, J., Y. Xie, J. McPherson, J. Luraschi, K. Ushey, A. Atkins, H. Wickham, J. Cheng, W. Chang, and W. Iannone. 2020. R markdown: Dynamic Documents for R.

Boëte, C., M. Seston, and M. Legros. 2019. Strategies of host resistance to pathogens in spatially structured populations: An agent-based evaluation. Theoretical Population Biology 130:170–181.

Bonabeau, E. 2002. Agent-based modeling: Methods and techniques for simulating human systems. Proceedings of the National Academy of Sciences 99:7280–7287.

Boots, M., and M. Mealor. 2007. Local Interactions Select for Lower Pathogen Infectivity. Science 315:1284–1286.

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 266:1933–1938.

Box, G. 1976. Science and statistics. Journal of the American Statistical Association 71:791–799.

Box, G. 1979. Robustness in the Strategy of Scientific Model Building. Pages 201–236 *in* Robustness in Statistics. Elsevier.

C G, N., T. LaBar, A. Hintze, and C. Adami. 2017. Origin of life in a digital microcosm. Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences 375:20160350.

Chang, W., J. Cheng, J. Allaire, and Y. Xie. 2020. Shiny: Web Application Framework for R.

Clarke, B. 1976. The ecological genetics of host-parasite relationships. *in* Genetic aspects of host-parasite relationships. Blackwell, London.

Clarke, B. 1979. The Evolution of Genetic Diversity. Proceedings of the Royal Society B: Biological Sciences 205:453–474.

Dowle, M., and A. Srinivasan. 2019. Data.Table: Extension of 'data.Frame'.

Dünner, R. P. 2020. Digital_Coevolution: An agent-based simulation for host-parasite coevolution.

Fortuna, M. A., L. Zaman, A. P. Wagner, and C. Ofria. 2013. Evolving Digital Ecological Networks. PLoS Computational Biology 9:e1002928.

Gandon, S. 1996. Local adaptation and gene-for-gene coevolution in a metapopulation model. Proceedings of the Royal Society of London. Series B: Biological Sciences 263:1003–1009.

Gandon, S. 2002. Local adaptation and the geometry of host-parasite coevolution. Ecology Letters 5:246–256.

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 15:451–462.

Gräbner, C., C. S. E. Bale, B. A. Furtado, B. Alvarez-Pereira, J. E. Gentile, H. Henderson, and F. Lipari. 2019. Getting the Best of Both Worlds? Developing Complementary Equation-Based and Agent-Based Models. Computational Economics 53:763–782.

Haldane, J. B., S. 1949. Disease and evolution. La Ricerca Scientifica Supplemento.

Haldane, J. B., S., and S. D. Jayakar. 1963. Polymorphism due to selection depending on the composition of a population. Journal of Genetics 3:318–323.

Hamilton, W. D. 1980. Sex versus Non-Sex versus Parasite. Oikos 35:282.

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 87:3566–3573.

Houston, A. I., P. A. Stephens, I. L. Boyd, K. C. Harding, and J. M. McNamara. 2007. Capital or income breeding? A theoretical model of female reproductive strategies. Behavioral Ecology 18:241–250.

Jaenike, J. 1978. An Hypothesis to account for the maintenance of sex within populations. Evolutionary Theory:191–194.

Judson, O. P. 1995. Preserving genes; a model of the maintenance of genetic variation in a metapopulation under frequency-dependent selection. Genet. Res.:175–191.

Judson, O. P. 1997. A Model of Asexuality and Clonal Diversity: Cloning the Red Queen. Journal of Theoretical Biology 186:33–40.

Kidner, J., and R. F. A. Moritz. 2015. Host-parasite evolution in male-haploid hosts: An individual based network model. Evolutionary Ecology 29:93–105.

LaBar, T., and C. Adami. 2016. Different Evolutionary Paths to Complexity for Small and Large Populations of Digital Organisms. PLOS Computational Biology 12:e1005066.

LaBar, T., and C. Adami. 2017. Evolution of drift robustness in small populations. Nature Communications 8:1012.

Ladle, R. J., R. A. Johnstone, and O. 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 253:155–160.

Lehman, J., J. Clune, and D. Misevic. 2018. The Surprising Creativity of Digital Evolution. Pages 55–56 *in* The 2018 Conference on Artificial Life. MIT Press, Tokyo, Japan.

Lehman, J., J. Clune, D. Misevic, C. Adami, L. Altenberg, J. Beaulieu, P. J. Bentley, S. Bernard, G. Beslon, D. M. Bryson, P. Chrabaszcz, N. Cheney, A. Cully, S. Doncieux, F. C. Dyer, K. O. Ellefsen, R. Feldt, S. Fischer, S. Forrest, A. Frénoy, C. Gagné, L. L. Goff, L. M. Grabowski, B. Hodjat, F. Hutter, L. Keller, C. Knibbe, P. Krcah, R. E. Lenski, H. Lipson, R. MacCurdy, C. Maestre, R. Miikkulainen, S. Mitri, D. E. Moriarty, J.-B. Mouret, A. Nguyen, C. Ofria, M. Parizeau, D. Parsons, R. T. Pennock, W. F. Punch, T. S. Ray, M. Schoenauer, E. Shulte, K. Sims, K. O. Stanley, F. Taddei, D. Tarapore, S. Thibault, W. Weimer, R. Watson, and J. Yosinski. 2019. The Surprising Creativity of Digital Evolution: A Collection of Anecdotes from the Evolutionary Computation and Artificial Life Research Communities. arXiv:1803.03453 [cs].

Lenski, R. E., C. Ofria, T. C. Collier, and C. Adami. 1999. Genome complexity, robustness and genetic interactions in digital organisms. Nature 400:661–664.

Lion, S., and S. Gandon. 2015. Evolution of spatially structured host-parasite interactions. Journal of Evolutionary Biology 28:10–28.

McKinley, P., B. Cheng, C. Ofria, D. Knoester, B. Beckmann, and H. Goldsby. 2008. Harnessing Digital Evolution. Computer 41:54–63.

Noordwijk, A. J. van, and G. de Jong. 1986. Acquisition and Allocation of Resources: Their Influence on Variation in Life History Tactics. The American Naturalist 128:137–142.

Ofria, C., and C. O. Wilke. 2004. Avida; A software platform for research in computational evolutionary biology. Artificial Life:129–229.

Oosterhout, C., D. A. Joyce, S. M. Cummings, J. Blais, N. J. Barson, I. W. Ramnarine, R. S. Mohammed, N. Persad, and J. Cable. 2006. Balancing selection, random genetic drift, and genetic variation at the major histocompatibility complex in two wild populations of guppies (poecilia reticulata). Evolution 60:2562–2574.

Perrin, N., and R. M. Sibly. 1993. Dynamic Models of Energy Allocation and Investment. Annual Review of Ecology and Systematics:379–410.

Sibly, R., and P. Calow. 1984. Direct and absorption costing in the evolution of life cycles. Journal of Theoretical Biology 111:463–473.

Stearns, S. C. 1980. A New View of Life-History Evolution. Oikos 35:266.

Stephens, P. A., I. L. Boyd, J. M. McNamara, and A. I. Houston. 2009. Capital Breeding and Income Breeding: Their Meaning, Measurement, and Worth. Ecology 90:2057–2067.

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

Team, R. C. 2020. R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austia.

Xie, Y., J. Allaire, and G. Grolemund. 2018. R Markdown: The Definitive Guide. Chapman and Hall / CRC.

Zaman, L., S. Devangam, and C. Ofria. 2011. Rapid host-parasite coevolution drives the production and maintenance of diversity in digital organisms. Page 219 *in* Proceedings of the 13th annual conference on Genetic and evolutionary computation - GECCO '11. ACM Press, Dublin, Ireland.