Scaling Hybrid Mean-Field/Master-Equation Models to Arbitrary Dimension with Programmatic Model Generation

Kirby Gordon[1,*]

[1]Vermont Complex Systems Institute, University of Vermont, Burlington, VT, USA
[*]kirby.gordon@uvm.edu

ABSTRACT

Mean-field models provide a computationally efficient method of tracking population dynamics. Master-equation models use much more computational effort to track probability distributions over any possible state of the population, thereby taking into account extinction events missed entirely by mean-field models. Hybrid models use fine-grained master-equation accounting for states below a population threshold, then switch to mean-field above the threshold. These models require a large number of equations, especially when scaling into higher-dimensional models, which has severely limited their use. This work introduces a programmatic model-generation system allowing easy scaling of hybrid models into arbitrary dimension.

1 Introduction

Hebert-Dufresne et al. introduce the hybrid mean-field/master-equation model in the context of approximate master equations (AMEs). That work models stochastic diffusion processes using a metapopulations model. In this case, taking account of extinction probabilities is critical, since occupied vs. unoccupied subpopulations play distinct roles in population-level diffusion/migration processes. Hebert-Dufresne et al. demonstrate use of hybrid models in one and two dimensions, requiring four and twelve equations, respectively. In three dimensions, thirty-four equations would be required, representing a qualitative shift into infeasibility.

This proliferation of equations comes from the need to treat states distinctly depending on their proximity to the mean-field regime. For each dimension of a cell's position that falls within this critical regime, the cell must track both an occupation probability, and a mean-value. Transitions out of a critical regime in some dimension are scaled by a poisson-based estimate of the likelihood that the mean-field system actually takes on exactly the critical value. This leads to a combinatoric explosion in the number of equations needed to describe the system.

Note, though, that the logic for a transition remains the same regardless of the dimension. There is an algorithmic recipe for writing down the equations governing the flow of probability mass starting from the transitions that define the system dynamics. Therefore, it is quite tractable to develop software that consumes a structured-data representation of the state transitions and produces a function modeling the probability flows of the system. This model-builder generalizes over any set of state transitions and any number of dimensions.

2 Methods

The problem

To understand the challenges of building hybrid models, we will consider a Lotka-Volterra model in two dimensions, shown in 1.

The interactions between species are represented by the following equations.

Fish increase at rate $\mu F(K-F)$, where K is the carrying capacity of fish and μ is a parameter controlling their reproduction rate

Sharks decrease at rate νS, where S is the number of sharks and ν is a death-rate parameter.

Sharks increase and fish decrease at rate βFS, where parameter β controls the predation rate.

The model

The dashed line separates critical from non-critical states. Within the critical regime, F and S track the mean-field values of fish and sharks. There are five types of transitions that must be handled slightly differently in code. First, arrows shown in black are basic transitions, which use the indexes of the origin cell to scale the transition. Red arrows exit the critical regime. In

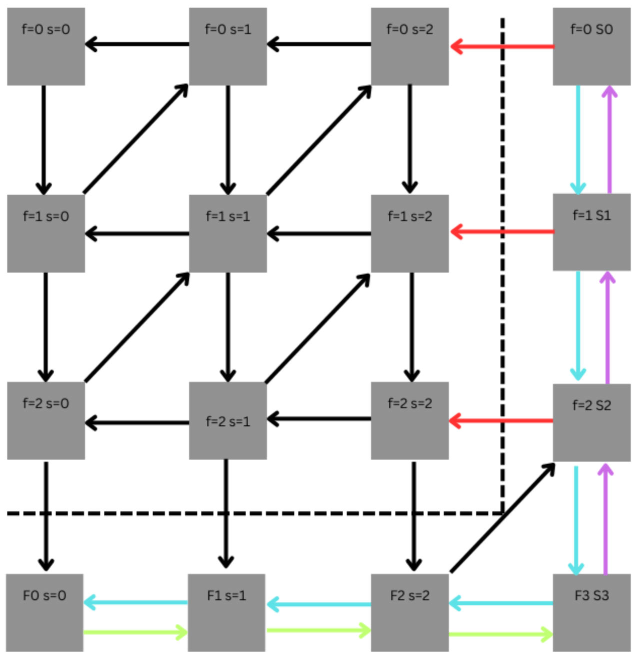

Figure 1. f=# of fish, s=# of sharks. Red arrows exit critical regime. Blue arrows remain inside critical regime. Green and purple arrows are virtual transitions.

the relevant dimension, the transition also scales with a poisson distribution evaluated at the critical value. Blue arrows are transitions that remain within a critical dimension. These scale based on the mean-field value, not the index.

Green and purple arrows are slightly more complex. We will refer to them as virtual transitions. Note that these arrows do not point in the direction of prey reproduction, predator death, or predation. For green arrows, this occurs because a predation event may occur without exiting the critical regime in the fish dimension. However, for non-critical numbers of sharks, this predation event will result in flow of probability mass to the state to the right. Similarly, when the predation transition originates in a cell which is already at the critical value for predators, the number of fish will decrease, but the predator dynamics will be captured by updates to the mean-field quantities. These are another type of transition that point in a different direction than any physical transition in the system. They are shown in purple.

Any given arrow may, in general, fall into multiple of these transition categories on different axes. We now see how computer software can be made to efficiently manage this complexity. Regardless of the dimensionality of the system, we iterate over each state in an outer loop, look at each transition in an inner loop, and handle each dimension along which the transition operates depending on the case. A listing showing a python implementation of such a program can be found in the Appendix.

3 Results

The software was tested on a one dimensional birth-death process, and a two dimensional Lotka-Volterra system. The one dimensional model was also written explicitly for the sake of comparison. The transitions and parameters governing the birth-death process model were as follows:

```
mu, nu = 1, 0.05
transitions = [
    Transition({Dir('pop', INC)}, lambda n: mu*n),
    Transition({Dir('pop', DEC)}, lambda n: nu*n**2)
]
```

The critical population threshold was set to $n = 8$. 2 shows the hard-coded ground-truth result for the birth-death process. 3 shows the same model, this time generated by the generalized model-builder code. Results are identical which validates the system in one dimension.

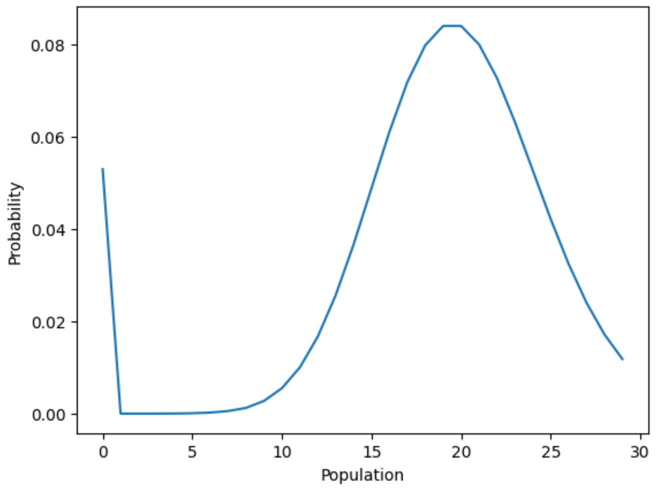

Figure 2. Hard-coded birth-death process model

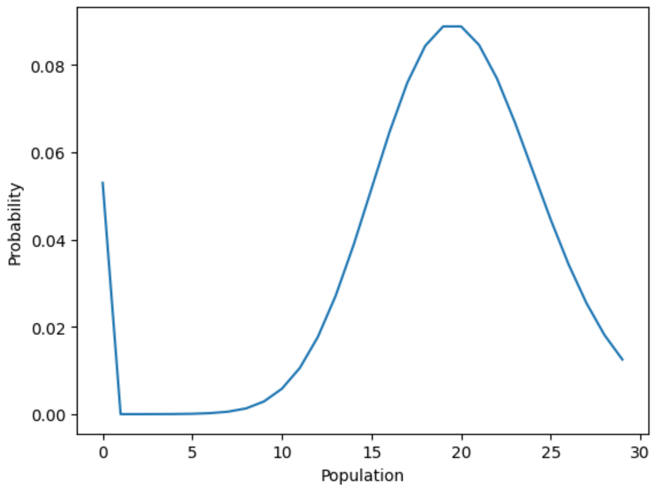

Figure 3. Generated-model birth-death process simulation

Next, a two dimensional Lotka-Volterra model was built and simulated using the model-builder software. The critical population value was set to 3 on each axis, and the following transitions and parameters defined the model:

```
K = 20
mu, nu, beta = 0.04, 0.4, 0.06
transitions = [
    Transition({Dir('fish', INC)}, lambda f: mu*f*(K-f)),
    Transition({Dir('shark', DEC)}, lambda s: nu*s),
    Transition({Dir('fish', DEC), Dir('shark', INC)}, lambda f,s: beta*f*s),
]
```

Note that the predation transition operates across two dimensions, with separate vector components in each.

4 shows the results of the simulating the generated model, compared with a mean-field version. As expected, the mean-field model overestimates the population of sharks because it fails to take into account extinction events, although both curves follow the same contour. Separately, 5 shows the extinction probability of the predator population over time.

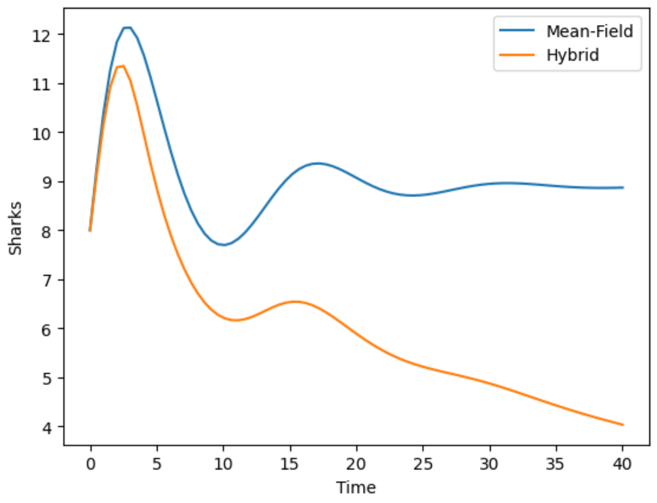

Figure 4. Generated-model Lotka-Volterra system with fish and sharks vs mean-field results for the same

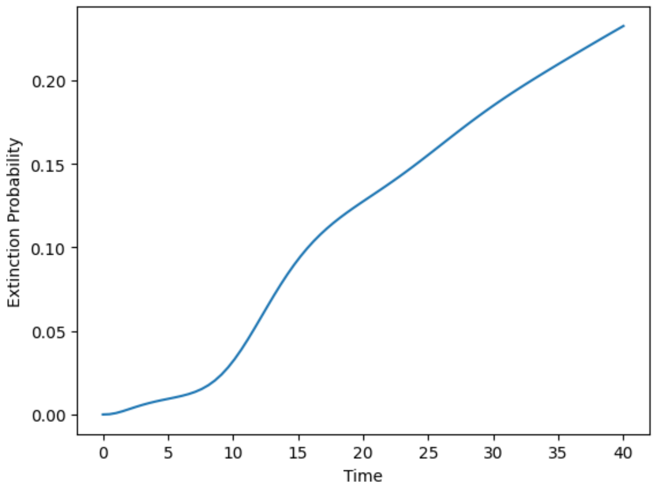

Figure 5. Extinction probability over time for sharks in the generated Lotka-Volterra model

4 Discussion

The model-builder functions correctly for the two test cases presented here. Further testing is very important in order to cover edge cases and validate the ability of the model to scale correctly to higher-dimensional systems. The software would also benefit greatly from reorganization and simplification. Finally, initializing the returned models currently requires knowledge of the internals of the representation scheme used by the model, which is poor design. Carrying out these validations and improvements would transform this system from a prototype to a more complete software tool supporting extinction-sensitive modeling in multi-dimensional systems.

There are numerous potential applications of such a model-building tool. For example, consider the ecosystem of an island habitat with a relatively small number of key species. Through observing populating dynamics and species interactions on the island for several years, ecologists can hypothesize about the interactions governing the trophic web and fit mean-field models to the observed data. A key question for such an ecosystem is to understand the time-scale for species extinction events, and to determine the relative susceptibility of different species to such events. Use of the system introduced in this work would be a highly effective approach to exploring such dynamics.

Alternatively, consider the gut microbiome. This is a many-dimensional ecosystem where population explosions and extinction events may both be observed. Administering antibiotics in response to pathogenic states of the microbiome is a common treatment, known to come with the severe drawback of damaging gut biodiversity. Since the quantity of antibiotic can easily be modeled as an additional dimension in the system, the hybrid model-building tool might be a useful approach to calibrating optimal dosage.

Acknowledgements

Thank you to Laurent Hebert-Dufresne for advising and saving me a lot of time.

Thank you to Derek Michael Devine for helping to find errors and discussing the work.

5 References

References

1. Hébert-Dufresne, Laurent, et al. "Stochastic diffusion using mean-field limits to approximate master equations." arXiv preprint arXiv:2408.07755 (2024).

Additional information

Code availability statement: Software is available at https://colab.research.google.com/drive/1kOEoNO6fk_lHkTx0BRYD_54_IUq7eokD?us

6 Appendix

```
from typing import Set, Callable, Tuple
from itertools import chain, combinations

# does not include S or {}
def powerset(iterable):
    s = list(iterable)
    return [set(c) for c in chain.from_iterable(
        combinations(s, r) for r in range(1, len(s))
    )]

class Dir:
    def __init__(self, dim, sign):
        self.dim = dim
        self.sign = sign
    def __eq__(self, other):
        return self.dim == other.dim and self.sign == other.sign
    def __hash__(self):
        return hash((self.dim, self.sign))

DEC, INC = -1, 1
class Transition:
    def __init__(
        self,
        dirs: Set[Dir],
        func: Callable,
        virtual: bool = False,
        fails: Set[Dir] = {}
    ):
        self.dirs = dirs
        self.func = func
        self.virtual = virtual
        self.fails = fails

def build_model(dimensions, shape, base_transitions):
    num_states = np.prod(np.array(shape)+1)
    rank = len(shape)
```

```
def mask(dirs):
    arr = np.zeros(rank, dtype='int')
    for d in dirs:
        arr[dimensions.index(d.dim)] = d.sign
    return arr

# generate virtual transitions
transitions = base_transitions.copy()
for t in base_transitions:
    for subset in powerset(t.dirs):
        transitions.append(Transition(
            t.dirs-subset,
            t.func,
            virtual=True,
            fails=subset
        ))
def J(states, _):
    # reshape states into joint prob and means
    P = np.pad(
        np.reshape(states[:num_states], np.array(shape)+1),
        [(0,1)]*rank,
        mode='constant',
        constant_values=0
    )
    JP = np.zeros(np.array(shape)+1)

    extended_shape = np.append(np.array(shape)+1, [rank])
    M = np.reshape(states[num_states:], extended_shape)
    JM = np.zeros(extended_shape)

    # update means according to mean field equations
    with np.nditer(M, flags=['multi_index']) as it:
        for m in it:
            cell = it.multi_index[:-1]
            if np.any(np.array(shape)==np.array(cell)):
                dim = it.multi_index[-1]
                critical_dims = np.array(shape)-np.array(cell)==0
                if np.any(critical_dims) and critical_dims[dim]:
                    JM[cell][dim] = np.sum([
                        mask(t.dirs)[dim]*t.func(
                            *np.extract(mask(t.dirs), M[cell])
                        ) for t in transitions if not t.virtual
                    ])

    # update probs according to master equations
    with np.nditer(P, flags=['multi_index']) as it:
        for p in it:
            cell = it.multi_index
            if np.any(np.array(shape)+1==np.array(cell)):
                continue
            for t in transitions:
                j_in, j_out = 0, 0
                # incoming
                origin = np.array(cell) - mask(t.dirs)
```

```
                    critical_origins = np.array(shape)-origin==0
                    critical_cells = np.array(shape)-np.array(cell)==0
                    critical_exits = critical_origins & ~critical_cells
                    critical_stays = critical_origins & critical_cells

# virtual transitions originate in critical dimensions that they fail to escape
                    if t.virtual and np.any(critical_stays!=mask(t.fails)):
                        # print(’virtual skip’)
                        continue

                    if not np.any(np.array(shape)+1==np.array(origin)):
                        j_in = P[tuple(origin)] * t.func(*np.extract(
                            mask(t.dirs)+mask(t.fails),
                            np.where(critical_stays, M[tuple(origin)], origin)))
                        # all transitions
                        j_in *= np.prod([
                            Po(n_c, mean_value) for n_c, mean_value in zip(
                                np.extract(critical_exits, shape),
                                np.extract(critical_exits, M[tuple(origin)])
                            )
                        ])
                        # virtual transition exit rolls
                        j_in *= np.prod([
                            1-Po(n_c, mean_value) for n_c, mean_value in zip(
                                np.extract(mask(t.fails)==DEC, shape),
                                np.extract(mask(t.fails)==DEC, M[cell])
                            )
                        ])

# outgoing: the only way to fail is to fail a roll in every changing dimension
                        JP[cell] += j_in
                    if t.virtual:
                        continue
                    target = np.array(cell) + mask(t.dirs)

                    if not np.any(target==-1) and not np.all(np.extract(
                        mask(t.dirs),
                        target==np.array(shape)+1
                    )):
                        critical_cells = np.array(shape)-np.array(cell)==0
                        critical_targets = np.array(shape)-np.array(target)<=0
                        critical_dim_shifts = critical_cells & ~critical_targets
                        critical_stays = critical_cells & critical_targets

                        j_out = t.func(*np.extract(
                            mask(t.dirs),
                            np.where(critical_stays, M[cell], cell)
                        )) * P[cell]
                        if np.sum(critical_dim_shifts) + np.sum(np.extract(
                            mask(t.dirs),
                            target==np.array(shape)+1
                        )) == len(t.dirs):
                            j_out *= 1-np.prod([
                                1-Po(n_c, mean_value) for n_c, mean_value in zip(
                                    np.extract(critical_dim_shifts, shape),
                                np.extract(critical_dim_shifts, M[cell])
                            )
                        ])
                JP[cell] -= j_out

    return np.concatenate((JP.flatten(), JM.flatten()))
return J
```