# Towards *in silico* studies of antibiotic resistance

Peter Mann and Simon Dobson

pm78@st-andrews.ac.uk
simon.dobson@st-andrews.ac.uk

University of St Andrews | FOUNDED 1413

# Introduction

We are interested in building computational (*in silico*) models of antibiotic resistance in disease pathogens

- How do epidemics spread mutations?
- Do changes in disease parameters lead to significantly different disease dynamics?
- What effects do different treatment regimes have?

At present we're developing the mathematical and computational preliminaries

- Models of multi-disease infection
- Better models of larger and longer-lived contact networks
- A simulation framework to explore the space

# The problem

Antibiotic resistance is one of the major challenges of the 21st century

- Decreasing efficiency of existing chemotherapies
- Excessive use of the current inventory promotes evolution

*In silico* studies

- Look for general patterns and factors affecting epidemics
- The effects of different scenarios
- Gradually introduce complexity rather than focusing on data from the wild, which will be largely uncontrolled

# Overview

Background

Simulation framework

Example

Conclusions

# Compartmented models

## Compartmented models of disease [1]

- Progress of disease modelled as a set of compartments (states)
- Individuals move between compartments according to stochastic rules
- Compartments, rules, and rates of transition define the disease dynamics

## Example: *S*usceptible-*I*nfected-*R*emoved (SIR)

- Most individuals start off in the $S$ compartment, with a small number seeded into $I$
- Removed individuals can't become re-infected

[1] H. Hethcote. The mathematics of infectious diseases. **SIAM Review**, 42(4):599–653, December 2000. URL doi://10.1137/S0036144500371907

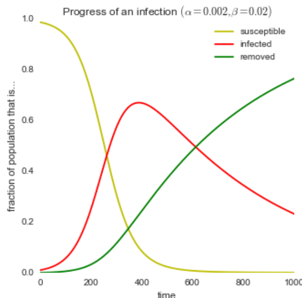
# Traditional modelling

## Rate-controlled ODEs

- Infection ($\beta$) and removal ($\alpha$) parameters
- Rate equations

$$\frac{dS}{dt} = -\beta S \quad \frac{dI}{dt} = \beta S - \alpha I$$

- Assumes a well-mixed population
- Basically a good model of a submarine or a nursery…



# Networked simulation

Use a network to introduce structure into the population over which the epidemic process runs [2]

- Build a network with an appropriate topology
- Nodes are people, edges are contacts
- *I* nodes infect adjacent *S* nodes with probability $\beta$, recover to *R* with probability $\alpha$
- A stochastic process governed by the topology of the network

[2] M. Newman. Spread of epidemic disease on networks. **Physical Review E**, 66, July 2002. URL doi://10.1103/PhysRevE.66.016128

# Multiple strains

Diseases change over time through selection pressures

- Different disease at the end of an epidemic compared to the start
- Dynamical parameters vary with time and space
- Some have different susceptibility, recover more easily

Multiple disease strains interact

- Having one pre-disposes you to another
- Partially immunity

# Richer model of disease

We're interested in making three changes

1. Disease parameters vary with time: how do different strains spread?
2. Long-term modelling: what happens when the network itself is in flux?
3. Treatment: how do different treatment regimes affect the disease dynamics?

Goals

- Extended mathematical models
- …backed-up by numerical networked simulation

# What do we need for convincing results?

Simulation at scale

- Typically want large ($10^5$-node) networks
- Explore parameter space fairly finely, maybe 100 points in each dimension
- These are stochastic processes, so need maybe $10^3$ repetitions averaged for each parameter combination

This means a *lot* of simulation

- The same experiment, repeated many times, almost certainly needs parallelism
- Quite a difficult thing to program efficiently, robustly, and correctly

# Reporting the results

Development practices of many scientific codes leave a lot to be desired (from a computer scientist's perspective)

- Don't publish the code (so we can't review it)
- Not well-abstracted (so we can't run it on different infrastructure)
- Not properly tested (so we can't trust it)

A lot of details go unreported in papers

- A lot of "folklore" that you only work out over time, increasing the learning curve
- Seems to be a physics thing…

# Overview

Background

Simulation framework

Example

Conclusions

# Software frameworks

We identified two distinct tasks we performed frequently

1. Define and run epidemic simulations over networks, using different compartmented models
2. Run these lots of repetitions of these models across multi-dimensional parameter spaces

So we've built two interoperating software packages to simplify them

- Code each thing once (and *only* once), and re-use
- Hide the details of the use of parallelism

# `epydemic`: Epidemic simulation in Python

## Epidemic processes

- Built on top of `networkx` for creating and manipulating networks

## Elements

- Compartmented models
  - Where do changes happen? (loci)
  - How does the model change? (events)
- Simulation dynamics
  - Synchronous: discrete "rounds" of dynamics
  - Stochastic (Gillespie): when will the next event happen? what will it be?

# Defining SIR – Compartments and rates

```
class SIR(CompartmentedModel):
    # the model parameters
    P_INFECTED = 'pInfected'  #: Parameter for probability of initially being infected.
    P_INFECT = 'pInfect'      #: Parameter for probability of infection on contact.
    P_REMOVE = 'pRemove'      #: Parameter for probability of removal (recovery).

    # the possible dynamics states of a node for SIR dynamics
    SUSCEPTIBLE = 'S'         #: Compartment for nodes susceptible to infection.
    INFECTED = 'I'            #: Compartment for nodes infected.
    REMOVED = 'R'             #: Compartment for nodes recovered/removed.

    # the edges at which dynamics can occur
    SI = 'SI'                 #: Edge able to transmit infection.

    def __init__( self ):
        super(SIR, self).__init__()
```

# Defining SIR – Building the model

```
def build( self, params ):
    pInfected = params[self.P_INFECTED]
    pInfect = params[self.P_INFECT]
    pRemove = params[self.P_REMOVE]

    self.addCompartment(self.SUSCEPTIBLE, 1 - pInfected)
    self.addCompartment(self.INFECTED, pInfected)
    self.addCompartment(self.REMOVED, 0.0)

    self.addLocus(self.SUSCEPTIBLE, self.INFECTED, name = self.SI)
    self.addLocus(self.INFECTED)

    self.addEvent(self.INFECTED, pRemove, lambda d, t, g, e: self.remove(d, t, g, e))
    self.addEvent(self.SI, pInfect, lambda d, t, g, e: self.infect(d, t, g, e))
```

# Defining SIR – The events

The two transitions in SIR

- Infection along an SI edge

```
def infect( self, dyn, t, g, e ):
    (n, m) = e
    self.changeCompartment(g, n, self.INFECTED)
    self.markOccupied(g, e)
```

- Recovery

```
def remove( self, dyn, t, g, n ):
    self.changeCompartment(g, n, self.REMOVED)
```

# epyc: Computational experiment modelling in Python

The infrastructure to run experiments

- Parameterised computational code to run (experiments)
- Simulation management (labs)
- Storing the results for analysis (notebooks)

Transparent use of `ipyparallel` compute clusters

- Same experiment runs locally or distributed
- Transparent scale-out
- Might even work in the cloud, eventually…

# Simple example

```
# build a notebook and a lab to populate it
nb = epyc.JSONLabNotebook('my-sir.json')
lab = epyc.Lab(nb)

# set the parameter space for the experiment
lab[epydemic.SIR.P_INFECTED] = 0.01
lab[epydemic.SIR.P_INFECT] = numpy.linspace(0.0, 1.0, num = 100)
lab[epydemic.SIR.P_REMOVE] = numpy.linspace(0.0, 1.0, num = 10)

# create a random network to run over
g = networkx.erdos_renyi_graph(10000, 5.0 / 10000)  # mean degree 5

# create and run the experiment
e = epydemic.StochasticCompartmentedDynamics(epydemic.SIR(), g)
lab.runExperiment(e)

# retrieve all the results as a pandas DataFrame
df = nb.dataframe()
```

# Repetitions

That example will perform 1,000 repetitions of the experiment, one per parameter combination

- Stochastic effects may introduce lots of artefacts

```
# Repeat the experiment several times at each point
re = epydemic.RepeatedExperiment(e, 100)
lab.runExperiment(re)
```

- Separate the experimental logic from the repetition logic
- Other experiment combinators

# Parallelism

We will now be doing 100,000 runs of the experiment

- Waiting a *long* time on a laptop
- . . . so we'll use a cluster instead

```
# build a lab that will use a cluster
clab = epyc.ClusterLab(nb, profile = 'myexperiment')
clab[epydemic.SIR.P_INFECTED] = lab[epydemic.SIR.P_INFECTED]
clab[epydemic.SIR.P_INFECT] = lab[epydemic.SIR.P_INFECT]
clab[epydemic.SIR.P_REMOVE] = lab[epydemic.SIR.P_REMOVE]

# run the experiment
clab.runExperiment(re)
clab.wait()
```

- The same experimental code runs locally or remotely

# Disconnected working

Even on a cluster, that much computation is hardly likely to be quick

- Awkward for a laptop user, especially when using an interactive system such as Jupyter notenooks
- Would like to "fire and forget"
- Come back later to collect complete or partial results

```
# run the experiment, don't wait for the results
clab.runExperiment(re)

# check how far we've got
print('{c} jobs completed ({f:.2#%}% of total)'.format(c = clab.numberOfResults(),
                                                       f = clab.readyFraction()))
```

- Results land in notebook as the cluster finishes the experiments

# Overview

Background

Simulation framework

Example

Conclusions

# Addition-deletion networks

We want to build random networks to model large-scale populations

- Birth and death, independent of the disease process
- A preliminary to looking at disease evolution, already studied in non-networked settings [3]

Build simulations in our framework

- Reproduce known network-only results [4]

[3] M. Meehan, D. Cocks, J. Trauer, and E. McBryde. Coupled, multi-strain epidemic models of mutating pathogens. **Mathematical Buisciences**, 296:82–92, 2018. URL doi://10.1016/j.mbs.2017.12.006

[4] C. Moore, G. Ghoshal, and M. Newman. Exact solutions for models of evolving networks with addition and deletion of nodes. **Physical Review E**, 74, September 2006. URL doi://10.1103/PhysRevE.74.036121

# The theory: the simplest case

Add and remove nodes to a base network

- Add a node linking randomly to $c$ neighbours
- Remove a random node (and all incident edges)

Degree distribution should converge to an equilibrium

- Compute $p_k$, the probability that a random node has degree $k$
- Analytic results are asymmetric on either side of $c$

# Process

Paper contains theory and simulation results

- …but in the way of these things, doesn't present the code, or a lot of the numerical detail

So we reproduced the results within our framework

- OK, we already know they're right…
- Infrastructure we need to do our own extensions, integrated with `epydemic`

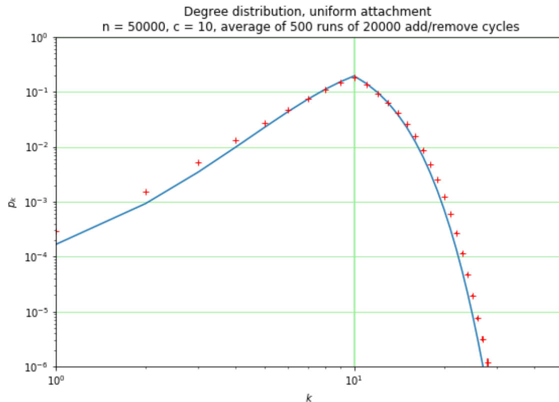
# The main part of the experiment

```
def do( self, params ):
    N = params[self.N]
    c = params[self.NODE_DEGREE]
    tmax = params[self.CYCLES]

    # run the experiment
    g = self._network
    ns = list(g.nodes())
    for t in xrange(tmax):
        self.add_node(g, N + t + 1, ns, c)
        self.remove_node(g, ns)

    # compute the degree distribution as fractions of the node population
    degrees = map(lambda id: id[1], g.degree())
    counts = numpy.bincount(degrees)
    pks = (counts + 0.0) / N

    # return the degree distribution
    rc = dict()
    rc[self.DEGREE_DIST] = list(pks)
    return rc
```

# Adding nodes

```
def add_node( self, g, i, ns, c ):
    # add a new node
    g.add_node(i)
    ns.append(i)

    # link to c other nodes with uniform probability
    es = set()
    for m in xrange(0, c):
        # a probably unnecessary test for parallel edges
        while(True):
            j = int(numpy.random.random() * (len(ns) - 1))
            k = ns[j]
            if k not in es:
                break
        es.add(k)
    for k in es:
        g.add_edge(i, k)
```

# Example – Results



# Overview

Background

Simulation framework

Example

Conclusions

# Next steps

Much easier to write simulations

- Compositional approach
- Easier to debug and test (although still fiddly)

Engineering

- Integrate addition-deletion with compartmented models
- Make it easier to spin-up clusters in the cloud

Research

- Develop theory of how these epidemics should behave
- Validate by simulating at scale

# Software

```
pip install epyc
pip install epydemic
```

- https://pypi.org/project/epyc/
  https://epyc.readthedocs.io/en/latest/index.html
- https://pypi.org/project/epydemic/
  https://pyepydemic.readthedocs.io/en/latest/index.html
- (I have some Jupyter notebooks showing more detail of the frameworks in use)

# References

H. Hethcote. The mathematics of infectious diseases. **SIAM Review**, 42(4):599–653, December 2000. URL `doi://10.1137/S0036144500371907`.

M. Meehan, D. Cocks, J. Trauer, and E. McBryde. Coupled, multi-strain epidemic models of mutating pathogens. **Mathematical Buisciences**, 296:82–92, 2018. URL `doi://10.1016/j.mbs.2017.12.006`.

C. Moore, G. Ghoshal, and M. Newman. Exact solutions for models of evolving networks with addition and deletion of nodes. **Physical Review E**, 74, September 2006. URL `doi://10.1103/PhysRevE.74.036121`.

M. Newman. Spread of epidemic disease on networks. **Physical Review E**, 66, July 2002. URL `doi://10.1103/PhysRevE.66.016128`.

# The degree distribution – 1

Addition-deletion with random (unweighted) selection

- The distribution should converge to:

$$p_k = \frac{e_c}{c^{c+1}} \left[ \Gamma(c+1) - \Gamma(c+1, c) \right] \frac{\Gamma(k+1, c)}{\Gamma(k+1)} \qquad k < c$$

$$p_k = \frac{e_c}{c^{c+1}} \Gamma(c+1, c) \left[ 1 - \frac{\Gamma(k+1, c)}{\Gamma(k+1)} \right] \qquad k \geq c$$

# The degree distribution – 2

The distribution uses the gamma function $\Gamma(c+1)$ and the "incomplete" gamma function $\Gamma(c+1,x)$

- For some reason the Python library version (`scipy.special.gammainc`) doesn't have the same definition as the one in the paper
- Make use of

$$\Gamma(c+1,x) = \Gamma(c+1)\, e^{-x} \sum_{m=0}^{c} \frac{x^m}{m!}$$

```
def gammainc(cpo, x):
    a = 0.0
    for m in xrange(0, int(cpo)):
        a = a + (math.pow(x, m) / math.factorial(m))
    return gamma(cpo) * math.exp(-x) * a
```