National Defence Défense nationale

DEFENCE RESEARCH AND DEVELOPMENT CANADA (DRDC)
RECHERCHE ET DÉVELOPPEMENT POUR LA DÉFENSE CANADA (RDDC)

PyCoMod (Python Compartment Modelling) Programming Reference

Stephen Okazawa
Josée van den Hoogen
Steve Guillouzic
DRDC – Centre for Operational Research and Analysis

Defence Research and Development Canada
Reference Document
DRDC-RDDC-2023-D111
October 2023

Canada

IMPORTANT INFORMATIVE STATEMENTS

This document was reviewed for Controlled Goods by Defence Research and Development Canada (DRDC) using the Schedule to the *Defence Production Act*.

Endorsement statement: This publication has been published by the Editorial Office of Defence Research and Development Canada, an agency of the Department of National Defence of Canada. Inquiries can be sent to: Publications.DRDC-RDDC@drdc-rddc.gc.ca.

Abstract

This Reference Document describes the PyCoMod (Python Compartment Modelling) code package. PyCoMod is used to build and run compartment models, such as susceptible-infectious-recovered (SIR) models of infectious disease. The package was initially developed to support analyses of the spread of Coronavirus Disease 2019 (COVID-19) in specific scenarios relevant to the Canadian Armed Forces (CAF) during the pandemic in 2020 and 2021. Over the course of multiple studies conducted during this period in collaboration with the Canadian Forces Health Services Group, the package evolved to include many features making it useful as a general modelling and simulation tool. The use of PyCoMod and its features will be described in detail in this Document.

Significance to defence and security

PyCoMod was developed to reduce the time and effort spent on creating, solving, and analyzing epidemiological compartment models. It was developed and used in multiple analyses that informed CAF decision making during the first two years of the COVID-19 pandemic. In addition to modelling COVID-19, PyCoMod can be used to develop models of other diseases relevant to military operations, such as vector-borne diseases (e.g., malaria), water-borne disease/illness (e.g., schistosomiasis, hepatitis), influenza, and others. Furthermore, PyCoMod compartment models fall into a category of general-purpose numerical models known as system dynamics. As a result, it is a tool that can be readily used outside the realm of epidemiological modelling, including areas such as logistics and resource management. As an open-source Python package hosted publicly on GitHub, it also highly suitable to collaborative development and modelling efforts.

Résumé

Le présent document de référence décrit le code du paquet PyCoMod (modélisation à compartiments Python). PyCoMod est utilisé pour créer et exécuter des modèles à compartiments, comme un modèle Susceptible-Infecté-Rétabli relatif à une maladie infectieuse. Le paquet a été élaboré initialement à l'appui des analyses sur la propagation de la COVID-19 dans le cadre de scénarios précis qui concernaient les Forces armées canadiennes pendant la pandémie en 2020 et en 2021. Au cours de multiples études effectuées pendant cette période en collaboration avec le Centre des services de santé des Forces canadiennes, le paquet a évolué et comprend maintenant de nombreuses fonctionnalités qui le rendent utile comme outil de modélisation et de simulation général. L'utilisation de PyCoMod et de ses fonctionnalités sera décrite en détail dans le présent document.

Importance pour la défense et la sécurité

PyCoMod a été mis au point dans le but de réduire le temps et les efforts consacrés à la création, à la résolution et à l'analyse de modèles épidémiologiques à compartiments. Le paquet a été élaboré et a ensuite été utilisé dans de multiples analyses qui ont orienté la prise de décisions au sein des Forces armées canadiennes pendant les deux premières années de la pandémie de COVID-19. En plus de modéliser la COVID-19, PyCoMod peut servir à élaborer des modèles d'autres maladies qui ont un rapport avec les opérations militaires, comme des maladies à transmission vectorielle (p. ex. paludisme), des maladies d'origine hydrique (p. ex. schistosomiase, hépatite), l'influenza, etc. En outre, les modèles à compartiments de PyCoMod font partie de la catégorie des modèles numériques généraux également connus sous le nom de « dynamique des systèmes ». Par conséquent, il s'agit d'un outil qui peut être facilement utilisé en dehors de la modélisation épidémiologique, y compris dans des domaines comme la logistique et la gestion des ressources. En tant que paquet Python libre hébergé publiquement sur GitHub, PyCoMod est parfaitement adapté aux efforts concertés d'élaboration et de modélisation.

Table of contents

List of figures

List of tables

1 Introduction

PyCoMod (Python Compartment Modelling) is a Python package for building and running compartment models derived from systems of differential equations such as the susceptible-infectious-recovered (SIR) model of infectious diseases. PyCoMod was developed to support analyses of the spread of the severe acute respiratory syndrome coronavirus 2 (SARS-CoV-2), which is the virus that causes Coronavirus Disease 2019 (COVID-19), in scenarios relevant to the Canadian Armed Forces (CAF) during the pandemic in 2020 and 2021 [1].

The package uses object-orientated design to efficiently build and run compartment models. PyCoMod is not a model of a specific system; rather, it is Python package to create and analyze systems that can be well-represented by a set of compartments (equivalently, pools or stocks) interconnected with flows defined by mathematical expressions. Compartment models are an epidemiological application of a broader numerical modelling approach known as system dynamics.

In order to accommodate more-realistic scenarios and practical aspects of modelling and simulation, PyCoMod includes several capabilities that go beyond the basics of compartment modelling, including stochastic flows, nested models, dynamic model parameters, vectorized models, Monte Carlo simulation, and efficient simulation management using initialization files and multi-run automation.

The purpose of this Reference Document is to provide a coding reference for PyCoMod. It will describe the installation procedure and provide numerous code examples covering basic to advanced features of the package. For ease of reference, PyCoMod's built-in objects and functions are summarized in Annex A. SIR models and extensions thereof will be used throughout the Document for illustrative purposes, but PyCoMod's applications are not limited to these types of models.

2 Python Compartment Modelling Installation

The PyCoMod package is publicly available on the Defence Research and Development Canada (DRDC) open science (OS) GitHub site:

```
https://github.com/DND-DRDC-RDDC/OS_PyCoMod
```

To install PyCoMod directly from GitHub to a local Python environment (requires Git version control system), run the following from the command line:

```
pip install git+https://github.com/DND-DRDC-RDDC/OS_PyCoMod.git
```

To install PyCoMod in Google Colab,[1] run the following in a code cell:

```
! pip install git+https://github.com/DND-DRDC-RDDC/OS_PyCoMod.git
```

After installing the package, import PyCoMod into your code:

```
import pycomod as pcm
```

The examples that follow were tested in Google Colab and assume that PyCoMod has been installed and imported as above using the abbreviated name *pcm*. They are also assumed to be executed sequentially from start to finish so that earlier imports and definitions are available in later examples.

[1] https://colab.research.google.com (accessed date: 6 September 2023).

3 Python Compartment Modelling examples

This section will introduce PyCoMod's main features by way of a series of examples and descriptions starting with a basic SIR model and proceeding to more advanced modelling scenarios.

Note that these examples were designed to demonstrate the features of PyCoMod; they are not necessarily appropriate models for real situations.

3.1 A simple susceptible-infectious-recovered model

The SIR model compartmentalizes a population based on the disease state of each individual. There are three compartments (Susceptible [S], Infectious [I], and Recovered [R]) and two flows that move individuals from Susceptible to Infectious and from Infectious to Recovered [2]–[4]. The compartments and flows are illustrated in Figure 1, where variables S, I and R represent the number of individuals in the Susceptible, Infectious, and Recovered compartments, respectively.

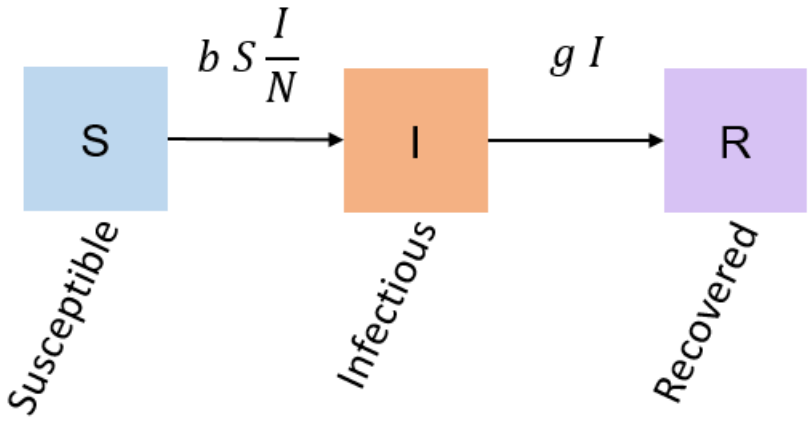

Figure 1: *Compartments and flows in the basic SIR model.*

The flow of individuals from S to I is given by the rate

$$F_{SI} = bS\frac{I}{N}, \tag{1}$$

where b is the transmission rate and N is the total population, equal to $S + I + R$.

The flow of individuals from I to R is given by the rate

$$F_{IR} = gI, \tag{2}$$

where g is the recovery rate and the reciprocal, g^{-1}, is the average time spent in the infectious compartment.

This produces the following system of differential equations:

$$\frac{dS}{dt} = -bS\frac{I}{N}; \tag{3}$$

$$\frac{dI}{dt} = bS\frac{I}{N} - gI; \text{ and} \tag{4}$$

$$\frac{dR}{dt} = gI. \tag{5}$$

Given a population of size 100, where 5 individuals are infected (I) and the remaining 95 individuals are susceptible (S), we can model this simple system in PyCoMod with the following code:

```
class SimpleSIR(pcm.Model):

    def build(self):
        # Pools
        self.S = pcm.Pool(95)
        self.I = pcm.Pool(5)
        self.R = pcm.Pool(0)

        # Equations
        self.N = pcm.Equation(lambda: self.S() + self.I() + self.R())

        # Parameters
        self.b = pcm.Parameter(0.2)
        self.g = pcm.Parameter(0.1)

        # Flows
        self.Fsi = pcm.Flow(lambda: self.b() * self.S() * self.I() / self.N(),
                            src=self.S, dest=self.I)
        self.Fir = pcm.Flow(lambda: self.g() * self.I(),
                            src=self.I, dest=self.R)

        # Output
        self.set_output('S', 'I', 'R')
```

The first two lines begin the definition of a custom class (i.e., a user-defined object type) that inherits properties from the PyCoMod base class for models and overrides the model's `build` method to define the elements of the SIR model. In this case, we create the three population compartments (S, I and R) using the PyCoMod `Pool` class (pool is the word used in PyCoMod for compartment) and specify the initial value of each pool (e.g., self.S = pcm.Pool(95)). We define the value `N` (the total population) as a PyCoMod `Equation`. Equations are defined by a function referencing other model elements, and we have used lambda functions [5] for their syntactical compactness. To obtain the value of a model element, we call the object by adding open- and close-parentheses; for example, the current number of susceptible individuals is obtained by `self.S()`. Using the PyCoMod `Parameter` class, we create and specify values for the model's parameters: the transmission rate, `b`, and recovery rate, `g`. Next, we define the movement between the compartments using the PyCoMod `Flow` class. Flows are defined by a function that returns the instantaneous flow rate. In this case, the flow functions correspond to the rate equations, $\boldsymbol{F_{SI}}$ and $\boldsymbol{F_{IR}}$, defined in Equations (1) and (2). Flows must also specify a source pool and a destination pool using the `src` and `dest` named arguments. Note that when specifying source and destination pools, we reference the pool object itself rather than calling it (e.g., `src=self.S`, not `src=self.S()`). A final step in specifying the model is to let PyCoMod know which values we want to capture for output. This is done by calling the model's `set_output` method and providing the names of the model elements that we want to track.

Having defined the `SimpleSIR` model class, we can now create an instance of it.

```
m = SimpleSIR()
```

We use another PyCoMod object called a `RunManager` to run it. The run manager keeps track of multiple models, run settings and outputs so that batches of runs can be automated. First, we create an instance of the run manager.

```
mgr = pcm.RunManager()
```

Now we can tell the run manager to `run` the `SimpleSIR` model. We can supply run settings (such as the duration in this example), and we must provide a label as a key to access the run results later.

```
mgr.run(m, duration=150, label='My run')
```

Finally, we can plot the results of the run using the PyCoMod `Plotter`. First, we create an instance of the plotter, which internally creates a Matplotlib Figure, and then we can plot outputs from the run on the figure axes. The result is shown is Figure 2.

```
plt = pcm.Plotter(title='SIR Time Series', ylabel='Population', fontsize=14)
plt.plot(mgr['My run'], 'S', color='blue', label='S')
plt.plot(mgr['My run'], 'I', color='orange', label='I')
plt.plot(mgr['My run'], 'R', color='green', label='R')
plt.plot(mgr['My run'], 'S + I + R', color='black', label='Total')
```

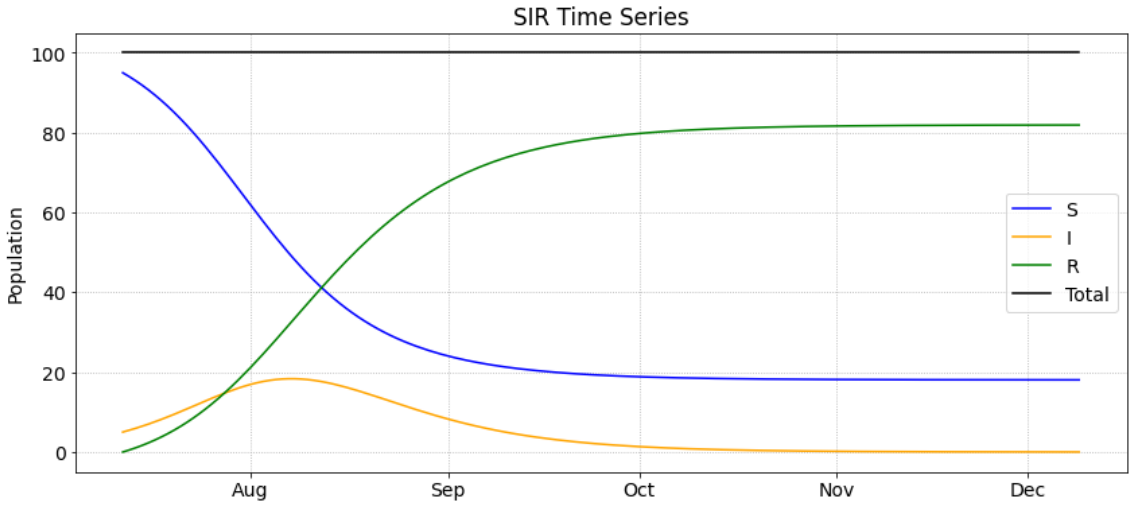

Figure 2: *Plot of the S, I and R model outputs over time.*

Each call to the plotter's `plot` function must specify a run and an output. The run is identified by indexing the run manager with the label that we specified when we ran the model (e.g., `mgr['My run']`). The output must be one of the outputs that was specified in the model using `set_output`. Outputs can be summed together in a plot, e.g., $S + I + R$ in the last line of the code above.

3.2 Adding model elements

To incorporate additional model elements, we simply add more pools, parameters, and flows to the model's `build` method. For example, we can expand the SIR model by incorporating an exposed compartment (E), thus creating a delay between the time of infection and the time of becoming symptomatic and infectious toward others, which models the virus' incubation period. This addition produces the common susceptible-exposed-infectious-recovered (SEIR) model [2]–[4] as seen in Figure 4.

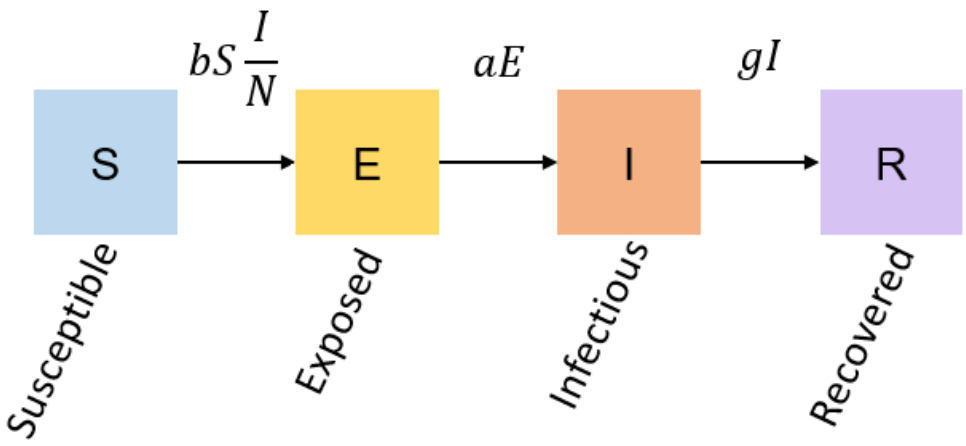

Figure 3: *Compartments and flows in the basic SEIR model.*

In the SEIR model, the parameter a controls the flow from E to I, where the reciprocal, a^{-1}, is the average incubation period for the disease.

This addition of the exposed compartment to the simple SIR example from Section 3.1 is shown in the following code:

```
class SimpleSEIR(pcm.Model):

    def build(self):
        # Pools
        self.S = pcm.Pool(95)
        self.E = pcm.Pool(0)
        self.I = pcm.Pool(5)
        self.R = pcm.Pool(0)

        # Equations
        self.N = pcm.Equation(
                lambda: self.S() + self.E() + self.I() + self.R())

        # Parameters
        self.b = pcm.Parameter(0.2)
        self.a = pcm.Parameter(0.1)
        self.g = pcm.Parameter(0.1)

        # Flows
        self.Fse = pcm.Flow(lambda: self.b() * self.S() * self.I() / self.N(),
                            src=self.S, dest=self.E)
        self.Fei = pcm.Flow(lambda: self.a() * self.E(),
                            src=self.E, dest=self.I)
        self.Fir = pcm.Flow(lambda: self.g() * self.I(),
                            src=self.I, dest=self.R)

        # Output
        self.set_output('S', 'E', 'I', 'R')


# Instantiate model
m = SimpleSEIR()

# Run model
```

```
mgr.run(m, duration=150, label='My run')

# Plot results
plt = pcm.Plotter(title='SEIR Time Series', ylabel='Population', fontsize=14)
plt.plot(mgr['My run'], 'S', color='blue', label='S')
plt.plot(mgr['My run'], 'E', color='red', label='E')
plt.plot(mgr['My run'], 'I', color='orange', label='I')
plt.plot(mgr['My run'], 'R', color='green', label='R')
plt.plot(mgr['My run'], 'S + E + I + R', color='black', label='Total')
```

3.3 Stochastic model elements

In PyCoMod, we can also introduce stochastic model elements and run Monte Carlo simulations. For example, two improvements to the simple SIR model would be to sample the transmission rate from a distribution reflecting the uncertainty in this parameter, and to make the flows stochastic and discrete. We show these changes below in a new model class called `MonteCarloSIR`.

```
import numpy as np
rng = np.random.default_rng()

class MonteCarloSIR(pcm.Model):

    def build(self):
        # Pools
        self.S = pcm.Pool(95)
        self.I = pcm.Pool(5)
        self.R = pcm.Pool(0)

        # Equations
        self.N = pcm.Equation(lambda: self.S() + self.I() + self.R())

        # Transmission rate parameters
        self.b_m = pcm.Parameter(0.2)
        self.b_s = pcm.Parameter(0.05)

        # Transmission rate random sample
        self.b = pcm.Sample(lambda: rng.normal(self.b_m(), self.b_s()))

        # Recovery rate parameter
        self.g = pcm.Parameter(0.1)

        # Flows
        self.Fsi = pcm.Flow(
                lambda: rng.binomial(self.S(), self.b() * self.I() / self.N()),
                src=self.S, dest=self.I)
        self.Fir = pcm.Flow(
                lambda: rng.binomial(self.I(), self.g()),
                src=self.I, dest=self.R)

        # Output
        self.set_output('S','I','R')


m2 = MonteCarloSIR()
```

The first two lines, above, import `NumPy`[2] and instantiate its default random number generator (RNG). We now specify the transmission rate b in Equation (3) with two parameters, a mean value `b_m` and a standard deviation `b_s`. Then we create a variable `b` for the transmission rate as a PyCoMod `Sample`, defined by a lambda function that calls `NumPy`'s normal (or Gaussian) RNG, passing `b_m` and `b_s` as parameters. This will resample the transmission rate from the normal distribution at the start of each model run.

The flow F_{SI} has been updated such that, rather than being a deterministic rate, each susceptible person has a probability of remaining susceptible or being infected in one unit of time based on the number of infected people in the population and the transmission rate. Therefore, we use the binomial RNG to generate a discrete, random number of new infections that will move from the susceptible population to the infectious population in one time step: `rng.binomial(self.S(), self.b()*self.I()/self.N())`. The flow F_{IR} has similarly been updated such that each infected person has a probability of recovering (or not) in each time step, again using the binomial RNG to generate a discrete, random number of people to move from the infectious compartment to the recovered compartment.

Lastly, we create an instance of the new model and call it `m2`. These modifications produce the same average behaviour as the deterministic model, but introduce variability based on the uncertainty in the transmission rate and the randomness of transmission and recovery events.

We can now run the model in Monte Carlo mode using the run manager's `run_mc` method, passing the number of replications (`reps`) in the run settings, and giving the run a new label.

```
mgr.run_mc(m2, duration=150, reps=100, label='My run - mc')
```

We can plot the results of a Monte Carlo run using the plotter's `plot_mc` method. The optional interval parameter specifies the percentile range from the distribution of outputs to be displayed. An interval of 50 means the middle 50% of the distribution, or the inter-quartile range. An interval of 90 would display the region from the 5th to 95th percentile. The result is shown in Figure 4.

```
plt = pcm.Plotter(title='SIR Time Series - Monte Carlo', ylabel='Population',
                  fontsize=14)
plt.plot_mc(mgr['My run - mc'], 'S', color='blue', interval=50, label='S')
plt.plot_mc(mgr['My run - mc'], 'I', color='orange', interval=50, label='I')
plt.plot_mc(mgr['My run - mc'], 'R', color='green', interval=50, label='R')
plt.plot_mc(mgr['My run - mc'], 'S + I + R', color='black', interval=50,
            label='Total')
```

[2] https://numpy.org (accessed date: 6 September 2023).

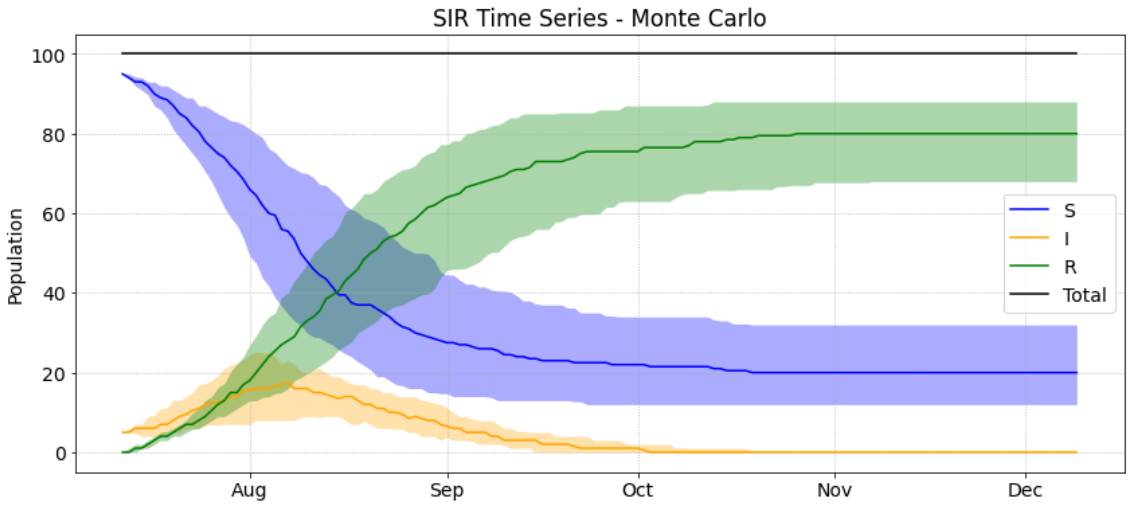

Figure 4: *Plot of the median and inter-quartile range for S, I and R over time from a Monte Carlo simulation where the transmission rate and infection and recovery events are stochastic.*

3.4 Nested models and model initialization

PyCoMod models support nesting, so any PyCoMod model can be used as an element inside another model. For example, if we have two sub-populations with different transmission dynamics and a certain degree of mixing between them, we can create a new model, `MixSIR`, that contains two instances of the `MonteCarloSIR` model defined previously in Section 3.3.

```
class MixSIR(pcm.Model):

    def build(self):

        # Sub models
        self.GrpA = MonteCarloSIR()
        self.GrpB = MonteCarloSIR()

        # Transmission parameter between groups
        self.b_mix = pcm.Parameter()

        # Cross-infection flows
        self.Fsi_GrpA = pcm.Flow(
                lambda: rng.binomial(self.GrpA.S(),
                                     self.b_mix() * self.GrpB.I()
                                                  / self.GrpB.N()),
                src=self.GrpA.S, dest=self.GrpA.I)
        self.Fsi_GrpB = pcm.Flow(
                lambda: rng.binomial(self.GrpB.S(),
                                     self.b_mix() * self.GrpA.I()
                                                  / self.GrpA.N()),
                src=self.GrpB.S, dest=self.GrpB.I)

        # Output
        self.set_output('GrpA','GrpB')


m3 = MixSIR()
```

In the code above, the two sub-populations, `GrpA` and `GrpB`, are both defined as instances of the `MonteCarloSIR` model. Each group behaves internally as before according to its parameters and initial conditions, but we introduce the possibility of cross-infection between these groups. The cross-infections occur with a different transmission rate, `b_mix`, defined as a `Parameter` in the `MixSIR` model. The cross-infection flows result in new infections within each group as a result of an interaction with an individual from the infectious population in the other group. Note that in order to save the output from a sub-model, the sub-model must be listed in the parent model's output list, `self.set_output(GrpA, GrpB)`; then all elements of the sub-model will be accessible when plotting.

While `GrpA` and `GrpB` are the same model, we will supply them with different parameter values and initial conditions. Previously, we specified these values while defining the model, but it is often preferable to separate model inputs from the model itself. Therefore, we can supply the inputs for the model at run-time using a Python dictionary. For the `MixSIR` model, the initialization dictionary would look something like `init_mix` below.

```
init_GrpA = {'S': 95, 'I': 5, 'R': 0, 'b_m': 0.2, 'b_s': 0.05, 'g': 0.1}
init_GrpB = {'S': 30, 'I': 0, 'R': 0, 'b_m': 0.3, 'b_s': 0.05, 'g': 0.1}
init_model = {'b_mix': 0.05, 'GrpA': init_GrpA, 'GrpB': init_GrpB}
init_run = {'reps': 100, 'end': 150}
init_mix = {'run':init_run, 'model':init_model}
```

The initialization dictionary consists of two entries: `run` contains a dictionary of run inputs, and `model` contains a dictionary of model inputs. In this case, the supplied run inputs are the number of repetitions (`reps`) and the `end` time. The model dictionary contains keys corresponding to the names of the model elements, and values to be used to initialize each element. The only model elements that accept input are pools, parameters, and sub-models. The entry value for a pool is the initial population of the pool. The entry value for a parameter is the parameter's value which is a constant. To initialize a sub-model, such as GrpA above, the entry value is another dictionary designed to initialize the sub-model, `init_GrpA = {'S': 95, 'I': 5, 'R': 0, 'b_m': 0.2, 'b_s': 0.05, 'g': 0.1}`. Hence, nested models are initialized with equivalently nested dictionaries. In this example, GrpA is given the same initialization values as in the MonteCarloSIR model while GrpB is a smaller population (Size 30) with a higher mean transmission rate, but with no initial infections. We then run the model using the dictionary to set both the model inputs and the run inputs.

```
mgr.run_mc(m3, init=init_mix, label='My run - mix')
```

We can then plot the Monte Carlo simulation of `GrpA`, as shown in Figure 5.

```
plt = pcm.Plotter(title='SIR Time Series - Monte Carlo - GrpA',
                  ylabel='Population', fontsize=14)
plt.plot_mc(mgr['My run - mix'], 'GrpA.S', color='blue',
            interval=50, label='S')
plt.plot_mc(mgr['My run - mix'], 'GrpA.I', color='orange',
            interval=50, label='I')
plt.plot_mc(mgr['My run - mix'], 'GrpA.R', color='green',
            interval=50, label='R')
plt.plot_mc(mgr['My run - mix'], 'GrpA.S + GrpA.I + GrpA.R', color='black',
            interval=50, label='Total')
```

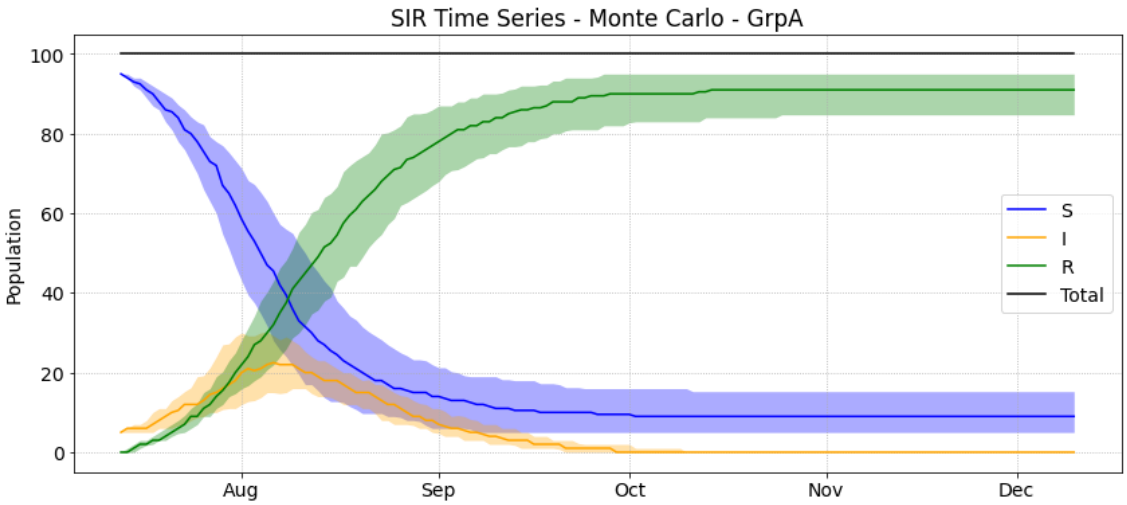

Figure 5: *Plot of the median and inter-quartile range for S, I and R over time for GrpA.*

Similarly, we can plot what happens to `GrpB`, as shown in Figure 6.

```
plt = pcm.Plotter(title='SIR Time Series - Monte Carlo - GrpB',
                  ylabel='Population', fontsize=14)
plt.plot_mc(mgr['My run - mix'], 'GrpB.S', color='blue',
            interval=50, label='S')
plt.plot_mc(mgr['My run - mix'], 'GrpB.I', color='orange',
            interval=50, label='I')
plt.plot_mc(mgr['My run - mix'], 'GrpB.R', color='green',
            interval=50, label='R')
plt.plot_mc(mgr['My run - mix'], 'GrpB.S + GrpB.I + GrpB.R', color='black',
            interval=50, label='Total')
```

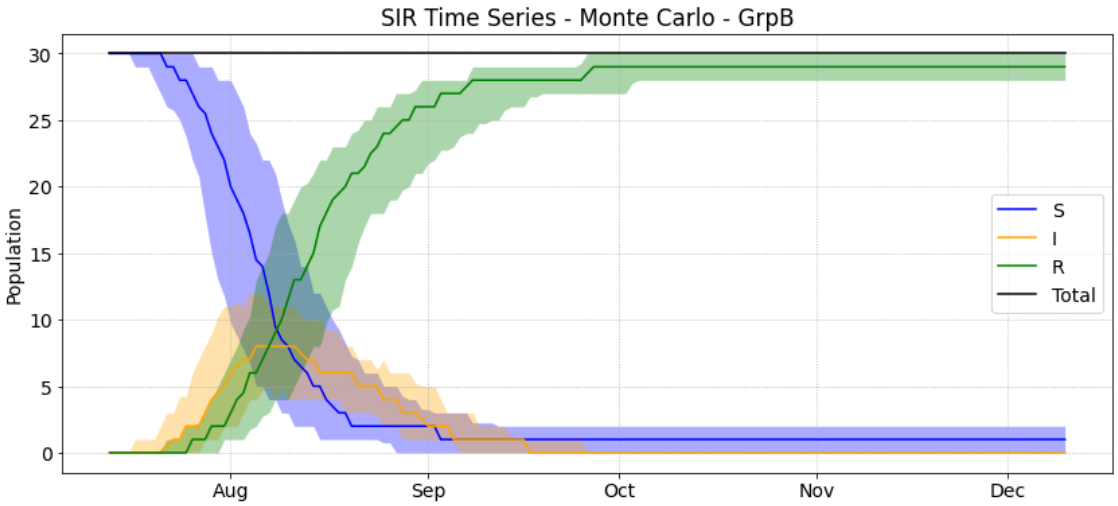

Figure 6: *Plot of the median and inter-quartile range for S, I and R over time for GrpB.*

Note in the above code that to specify the output we want to plot in a nested model, we use dot-notation to navigate the sub-models. E.g., `GrpB.S` plots the susceptible population within `GrpB`.

3.5 Initialization files

Initialization dictionaries are useful when we want to set up the model in Python code, but it is often more practical to specify the initialization data in a separate file. This allows different model setups to be saved and edited by hand. For this purpose, PyCoMod models can also be initialized from an Excel file. The Excel file template to initialize a particular model can be generated by the model itself by calling `write_excel_init` and providing a file name.

```
m3.write_excel_init('init_mix.xlsx')
```

In Google Colab, the initialization file will be written to the temporary session storage and can be downloaded, modified and re-uploaded. In a local Python environment, the file is written to local storage.

The Excel initialization file is structured in a similar way to the initialization dictionary. The first tab contains run inputs, the second tab contains the top-level model inputs, and subsequent tabs contain sub-model inputs if sub-models are present. In the case of the example provided in Section 3.4, there are four tabs as shown in Figure 7.

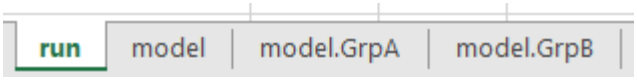

Figure 7: *Tab structure of an Excel initialization file.*

The content of the run tab is shown in Figure 8 and always consist of the following run settings:

- *t*—the initial simulation time (usually 0),
- *date*—the initial simulation date,
- *dt*—the simulation time step,
- *end*—the simulation end time, and
- *reps*—the number of replications for Monte Carlo runs.

	A	B	C	D	E
1	**t**	**date**	**dt**	**end**	**reps**
2	0	2023-02-03 00:00:00	1	300	100

Figure 8: *Content of the run tab.*

The model tab contains the initialization inputs for the elements of the top-level model. In this case, they are *GrpA*, *GrpB*, and *b_mix*, which are shown in Figure 9.

	A	B	C	D
1	**GrpA**	**GrpB**	**b_mix**	**out**
2	model.GrpA	model.GrpB	0.05	GrpA
3				GrpB

Figure 9: *Content of the model tab.*

We can edit the value for the cross-infection parameter *b_mix,* here.

Because *GrpA* and *GrpB* are sub-models, the value under these labels is the name of the tab that contains the initialization data for that sub-model. So under *GrpA*, the value is *model.GrpA* which is the name of the third Excel tab. Tab names contain the full path from the model hierarchy to avoid naming collisions in the event that two sub-models have the same name. It should not be necessary to change the sheet-name entry under a sub-model within the *model* tab. In the *model.GrpA* tab, shown in Figure 10, we find the inputs for the elements of the *GrpA* sub-model: *S*, *I*, *R*, *b_m*, *b_s*, and *g*.

	A	B	C	D	E	F	G
1	**S**	**I**	**R**	**b_m**	**b_s**	**g**	**out**
2	95	5	0	0.2	0.05	0.1	S
3							I
4							R

Figure 10: *Content of the model, GrpA tab.*

The same applies to the *GrpB* sub-model tab. Each model tab also contains an *out* entry which is used to list the desired outputs for the model or sub-model. This has the same function as calling `set_output` within the model definition. Recall that the outputs of a sub-model will only be saved if the parent model includes the sub-model in its output list.

We can edit the values in the Excel file, for example, by changing *b_mix* to 0.025 (cutting the transmission rate between the two populations in half) and then saving the changes.

In Google Colab, we then must upload the edited file to the session storage.

Now we can run the model using the Excel file to initialize it.

```
mgr.run_mc(m3, init='init_mix.xlsx', label='My run - mix - xls')
```

The output can then be viewed in the same way as shown in the previous section.

3.6 Dynamic model parameters

It is often necessary to adjust model parameters over time. This can be accomplished using PyCoMod's `equation` class. For example, we might want to modify the `SimpleSIR` model to make the transmission rate decay over time, reflecting increasing adherence to public health measures. So, we could replace the transmission parameter *b* with an equation implementing an exponential decay (i.e., $b(t) = 0.2(0.98)^t$).

```
class ModSIR(pcm.Model):

    def build(self):
        # Pools
        self.S = pcm.Pool(95)
        self.I = pcm.Pool(5)
        self.R = pcm.Pool(0)

        # Equations
        self.N = pcm.Equation(lambda: self.S() + self.I() + self.R())

        # Parameters
        self.b = pcm.Equation(lambda: 0.2 * (0.98)**self.t())
        self.g = pcm.Parameter(0.1)

        # Flows
```

```
        self.Fsi = pcm.Flow(lambda: self.b() * self.I() * self.S() / self.N(),
                            src=self.S, dest=self.I)
        self.Fir = pcm.Flow(lambda: self.g() * self.I(),
                            src=self.I, dest=self.R)

        # Output
        self.set_output('S', 'I', 'R', 'b')


m4 = ModSIR()
```

The current simulation time can be accessed and used in the equation for `b` by calling the special variable `self.t`. We can view the modified transmission rate over time by including `b` in the list of outputs, running the model, and then plotting it; see Figure 11.

```
mgr.run(m4, duration=150, label='Mod SIR')

plt = pcm.Plotter(title='Dynamic transmission rate',
                  ylabel='Value', fontsize=14)
plt.plot(mgr['Mod SIR'], 'b', color='blue', label='Transmission rate')
```

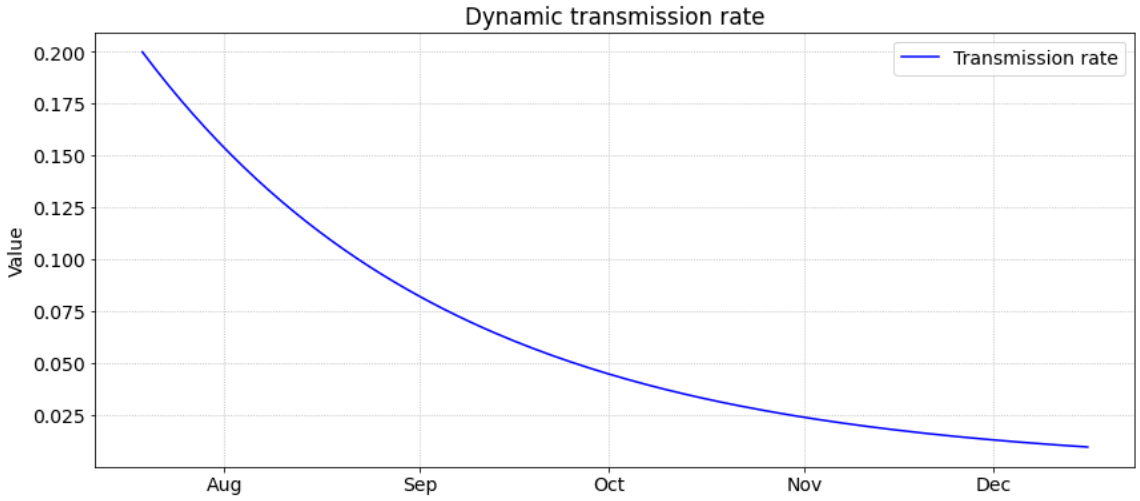

Figure 11: *A dynamic transmission rate modelled with an exponential decay equation.*

Sometimes we want a parameter to change to specific values at specific times, in other words, a step function. It is possible to implement a step function as a PyCoMod `Equation`, but this is not trivial. As this is a common requirement in modelling and simulation, PyCoMod provides a built-in equation sub-class called `Step`. For example, we can change the `ModSIR` model from Section 3.6 such that the transmission rate increases and decreases at certain times, reflecting specific public health measures coming into and out of force.

```
self.b = pcm.Step([0.2, 0.13, 0.2], [0, 7, 21])
```

When initializing the PyCoMod step object, we provide a list of values and a corresponding list of times. In this case, the transmission rate is initially 0.2 at time 0, it then reduces to 0.13 on day 7 for a period of two weeks, after which it returns to 0.2 on day 21, shown in Figure 12. Note that the default time unit in PyCoMod is 1 day.

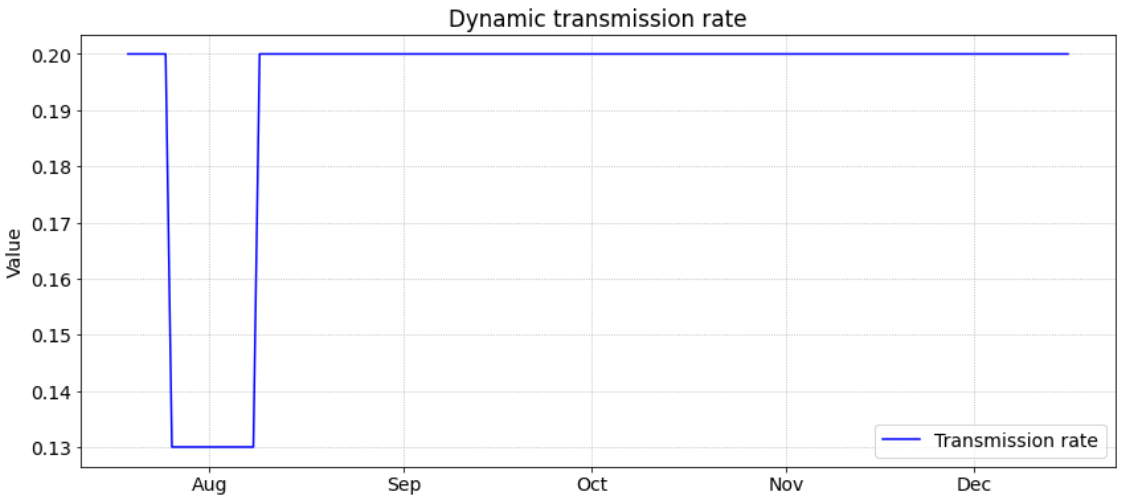

Figure 12: *A dynamic transmission rate modelled as a step function.*

In the above examples, the numerical constants used to define the changing transmission rate, `b`, could be replaced with PyCoMod `Parameters` which would register them as model inputs allowing them to be adjusted via an initialization dictionary or initialization file. This is an important advantage of using parameters rather than literals in a model.

In the case of the `Step` class, we need two vectors, and PyCoMod `Parameter` objects support vector inputs. So, we can create a parameter `b_v` for the values of the transmission rate and a parameter `b_t` for the times at which they will be applied.

```
self.b_v = pcm.Parameter([0.2, 0.13, 0.2])
self.b_t = pcm.Parameter([0, 7, 21])
self.b = pcm.Step(self.b_v(), self.b_t())
```

The initialization dictionary for this model would then specify lists for the values of `b_v` and `b_t`.

```
init_mod = {'run': {'end': 150},
            'model':{'S': 95, 'I': 5, 'R': 0, 'b_v': [0.2, 0.13, 0.2],
                     'b_t': [0, 7, 21], 'g': 0.1}}
```

If we would rather create an Excel initialization file for this model, we will see two vector inputs for the parameters `b_v` and `b_t`, shown in Figure 13.

D	E
b_v	**b_t**
0.2	0
0.13	7
0.2	21

Figure 13: *Vector inputs in the initialization Excel file for the transmission rate step function.*

Whichever method is used, we can now edit the timing and magnitude of changes to the transmission rate. The size of the vector is not restricted to the initial dimension of three in this example. More values and times can be added so long as there is always a corresponding time for each value.

The PyCoMod `Impulse` is another type of dynamic function similar to `Step`. The `Impulse` class generates specified values at specified times, but only at those times. In other words, the impulse value is held only for the timestep that contains the impulse time, otherwise it returns 0 or an optional default value. For example, the transmission rate in our model could be 0.2 under normal circumstances, but on certain dates there may be events that are expected to result in elevated transmission.

```
self.b = pcm.Impulse([0.5, 0.5, 0.5], [10, 25, 45], 0.2)
```

When initializing PyCoMod `Impulse`, we provide a list of impulse values, a list of impulse times, and an optional default value. In this case, it produces an elevated transmission rate of 0.5 on days 10, 25 and 45, but it otherwise produces the nominal rate of 0.2, shown in Figure 14.

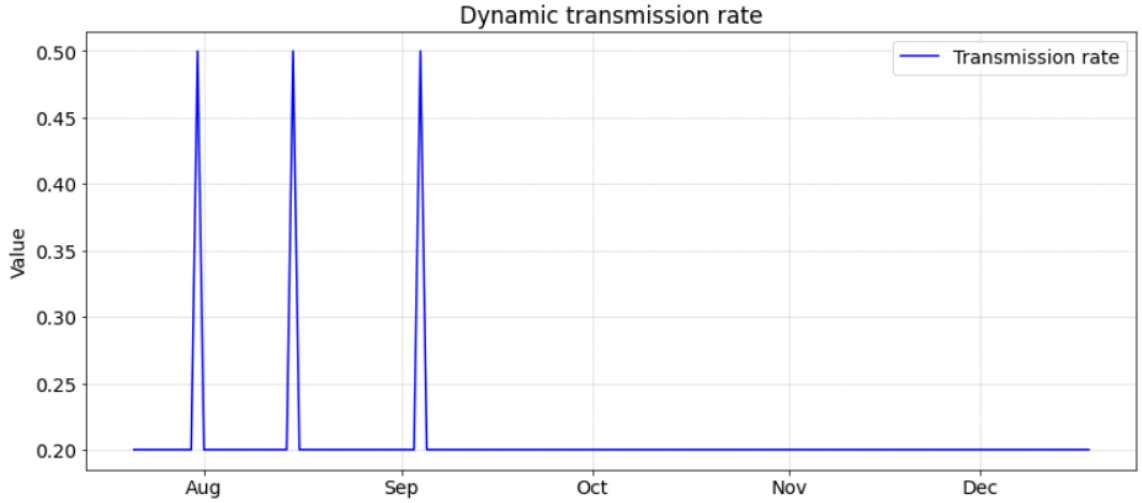

Figure 14: *A dynamic transmission rate modelled as an impulse function.*

The same approach as described above can be used to set these values using an initialization dictionary or Excel file.

3.7 Initial flows

In some cases, it may be useful to incorporate flows into establishing the initial state of the system. For example, we may not know that there are exactly 5 initial infections in the population, as in the preceding examples. Instead, we may only know that there is a 5% chance that any given person is infected, based on some larger population statistics. To model this situation, we can place the entire population in the susceptible, `S`, compartment, and use a stochastic initial flow to move a random number of them to the infectious, `I`, compartment based on the aforementioned 5% probability.

```
class MonteCarloSIR2(pcm.Model):

    def build(self):
        # Pools
        self.S = pcm.Pool(100)
        self.I = pcm.Pool(0)
        self.R = pcm.Pool(0)

        # Equations
        self.N = pcm.Equation(lambda: self.S() + self.I() + self.R())

        # Transmission rate parameters
        self.b_m = pcm.Parameter(0.2)
```

```
        self.b_s = pcm.Parameter(0.05)

        # Transmission rate random sample
        self.b = pcm.Sample(lambda: rng.normal(self.b_m(), self.b_s()))

        # Recovery rate parameter
        self.g = pcm.Parameter(0.1)

        # Flows
        self.Fsi = pcm.Flow(
                lambda: rng.binomial(self.S(),
                                     self.b() * self.I() / self.N()),
                src=self.S, dest=self.I)
        self.Fir = pcm.Flow(
                lambda: rng.binomial(self.I(), self.g()),
                src=self.I, dest=self.R)

        # Initial flow
        self.Pi = pcm.Parameter(0.05)
        self.Fsi_init = pcm.Flow(lambda: rng.binomial(self.S(), self.Pi()),
                                 src=self.S, dest=self.I, init=True)

        # Output
        self.set_output('S','I','R')


m5 = MonteCarloSIR2()
```

In the above code, note that the `S` pool is initialized to contain the whole population, and `I` and `R` are empty. Toward the end of the model definition, we have added a parameter, `Pi`, for the 5% probability of initial infection, and the initial flow `Fsi_init`. This flow uses a binomial RNG to move a random number of individuals from *S* to *I* using the probability *Pi*. To flag this flow as an initial flow, we set the optional `init` parameter to *True*. This flow will now only be executed once at the start of each run.

If we run this model, we can see that the initial state of the system is now uncertain, and there is more variability in the outcome, shown in Figure 15, compared to the first `MonteCarloSIR` model in Section 3.3.

```
mgr.run_mc(m5, duration=150, reps=100, label='My run - mc2')

plt = pcm.Plotter(title='SIR Time Series - Monte Carlo',
                  ylabel='Population', fontsize=14)
plt.plot_mc(mgr['My run - mc2'], 'S', color='blue',
                interval=50, label='S')
plt.plot_mc(mgr['My run - mc2'], 'I', color='orange',
                interval=50, label='I')
plt.plot_mc(mgr['My run - mc2'], 'R', color='green',
                interval=50, label='R')
plt.plot_mc(mgr['My run - mc2'], 'S + I + R', color='black',
                interval=50, label='Total')
```

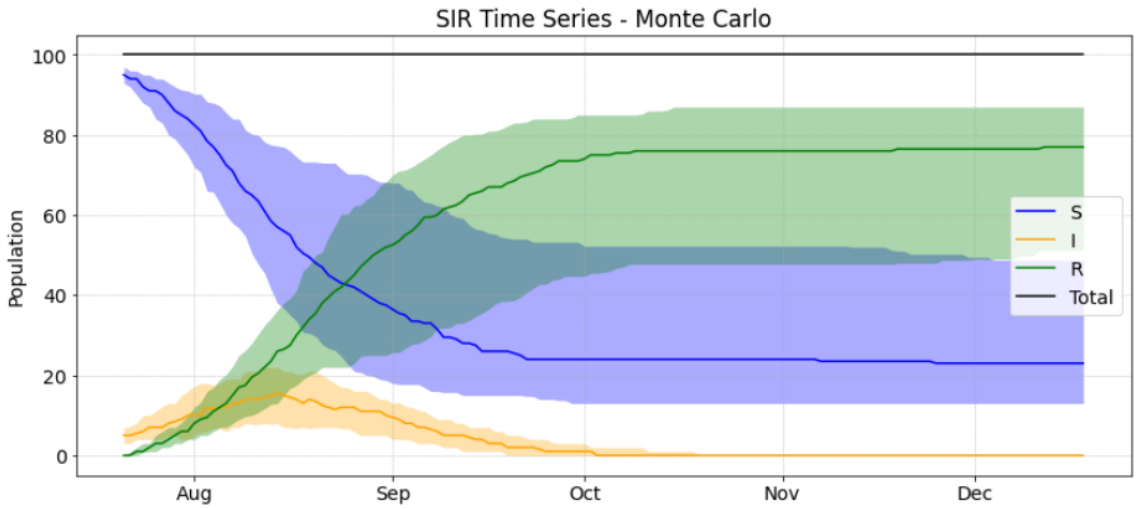

Figure 15: *The result of the model where the initial state is controlled by a stochastic initial flow from S to I.*

3.8 Vectorization

In PyCoMod, the values held by model elements can be vectors. As with vector parameters introduced previously in Section 3.6, pools can also be initialized with a list of values and are stored internally as `NumPy` arrays. Many mathematical operations in `NumPy` are seamlessly compatible with vector values. `NumPy's` RNG functions are also compatible with vector inputs. In many cases a model developed for scalar values will be compatible with vector values with little or no changes. This feature is useful for modelling multiple isolated or semi-isolated populations in parallel, such as a training setting in which students are divided into parallel cohorts. Note that a familiarity with how `NumPy` handles vectors in mathematical expressions is necessary to build vectorized models.

For example, we can vectorize the `MonteCarloSIR2` model from Section 3.7 simply by changing the pool initial values to lists. In this case, the susceptible population is initialized to 10 cohorts containing 10 individuals each, and the infectious and recovered populations are initialized to 10 empty cohorts. Note that the `S`, `I` and `R` pools must all have the same number of cohorts. The rest of the model accommodates the vectorized populations without any changes. So rather than a single SIR model of 100 people, we have 10 parallel SIR models of 10 people each.

```
class VecSIR(pcm.Model):

    def build(self):
        # Pools
        self.S = pcm.Pool([10] * 10)
        self.I = pcm.Pool([0] * 10)
        self.R = pcm.Pool([0] * 10)

        # Equations
        self.N = pcm.Equation(lambda: self.S() + self.I() + self.R())

        # Transmission rate parameters
        self.b_m = pcm.Parameter(0.2)
        self.b_s = pcm.Parameter(0.05)

        # Transmission rate random sample
        self.b = pcm.Sample(lambda: rng.normal(self.b_m(), self.b_s()))

        # Recovery rate parameter
        self.g = pcm.Parameter(0.1)
```

```
        # Flows
        self.Fsi = pcm.Flow(
                lambda: rng.binomial(self.S(), self.b() * self.I() / self.N()),
                src=self.S, dest=self.I)
        self.Fir = pcm.Flow(
                lambda: rng.binomial(self.I(), self.g()),
                src=self.I, dest=self.R)

        # Initial flow
        self.Pi = pcm.Parameter(0.05)
        self.Fsi_init = pcm.Flow(lambda: rng.binomial(self.S(), self.Pi()),
                                 src=self.S, dest=self.I, init=True)

        # Output
        self.set_output('S','I','R')


m6 = VecSIR()
```

If we plot the result, we can see the protective effect of dividing the population into isolated cohorts, Figure 16. Note that when we plot a model output that is vectorized, it is the sum of the vector that is shown on the figure.

```
mgr.run_mc(m6, duration=150, reps=100, label='My run - vec')

plt = pcm.Plotter(title='SIR Time Series - Monte Carlo',
                  ylabel='Population', fontsize=14)
plt.plot_mc(mgr['My run - vec'], 'S', color='blue',
            interval=50, label='S')
plt.plot_mc(mgr['My run - vec'], 'I', color='orange',
            interval=50, label='I')
plt.plot_mc(mgr['My run - vec'], 'R', color='green',
            interval=50, label='R')
plt.plot_mc(mgr['My run - vec'], 'S + I + R', color='black',
            interval=50, label='Total')
```

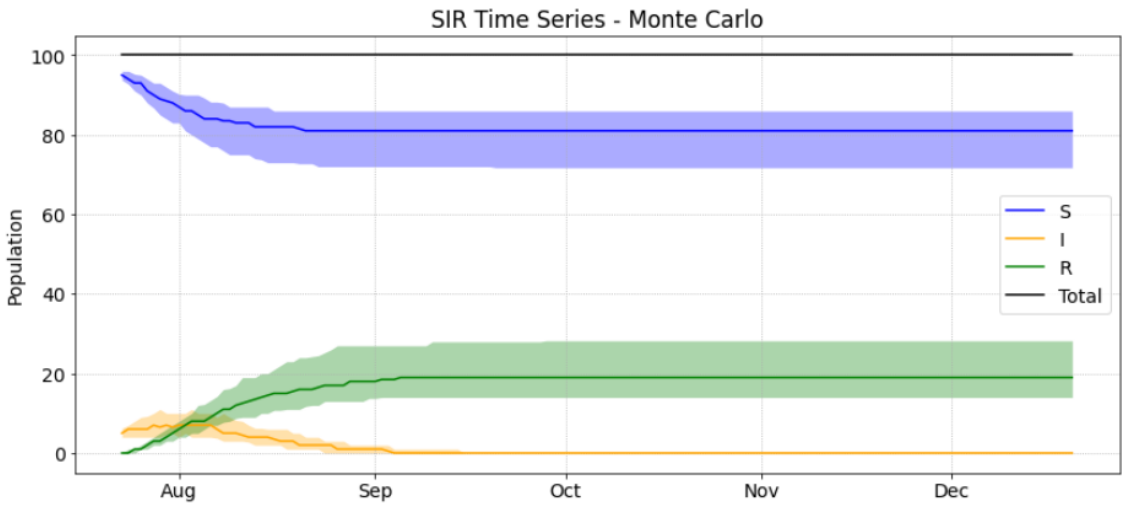

Figure 16: *The result of a vectorized model where all populations are divided into 10 cohorts.*

However, it is usually not realistic to assume that populations are perfectly isolated, so we can introduce a potential for spread between cohorts. At the end of the model definition, we add the parameter `b_mix` which is the smaller rate of transmission between cohorts (one tenth the nominal transmission rate within cohorts), and we add the flow `Fsi_mix` which creates new infections within each cohort as a result of mixing between cohorts. When a susceptible person is in a mixed setting (e.g., a hallway where cohorts share the same space), the probability that they encounter an infectious person is given by the total proportion of infectious people in the population, hence the modified term `self.I().sum()/self.N().sum()` appears in the flow equation. The addition of `.sum()` returns the sum of the vector, in other words, the sum across the cohorts.

```
class VecSIR(pcm.Model):

    def build(self):
        # Pools
        self.S = pcm.Pool([10] * 10)
        self.I = pcm.Pool([0] * 10)
        self.R = pcm.Pool([0] * 10)

        # Equations
        self.N = pcm.Equation(lambda: self.S() + self.I() + self.R())

        # Transmission rate parameters
        self.b_m = pcm.Parameter(0.2)
        self.b_s = pcm.Parameter(0.05)

        # Transmission rate random sample
        self.b = pcm.Sample(lambda: rng.normal(self.b_m(), self.b_s()))

        # Recovery rate parameter
        self.g = pcm.Parameter(0.1)

        # Flows
        self.Fsi = pcm.Flow(
                lambda: rng.binomial(self.S(), self.b() * self.I() / self.N()),
                src=self.S, dest=self.I)
        self.Fir = pcm.Flow(
                lambda: rng.binomial(self.I(), self.g()),
                src=self.I, dest=self.R)

        # Initial flow
        self.Pi = pcm.Parameter(0.05)
        self.Fsi_init = pcm.Flow(lambda: rng.binomial(self.S(), self.Pi()),
                                 src=self.S, dest=self.I, init=True)

        # Mixing
        self.b_mix = pcm.Parameter(0.02)
        self.Fsi_mix = pcm.Flow(
                lambda: rng.binomial(self.S(),
                                     self.b_mix() * self.I().sum()
                                                  / self.N().sum()),
                src=self.S, dest=self.I)

        # Output
        self.set_output('S','I','R')


m6 = VecSIR()
```

If we plot the output, we can see the effect of the limited degree of mixing between cohorts in Figure 17.

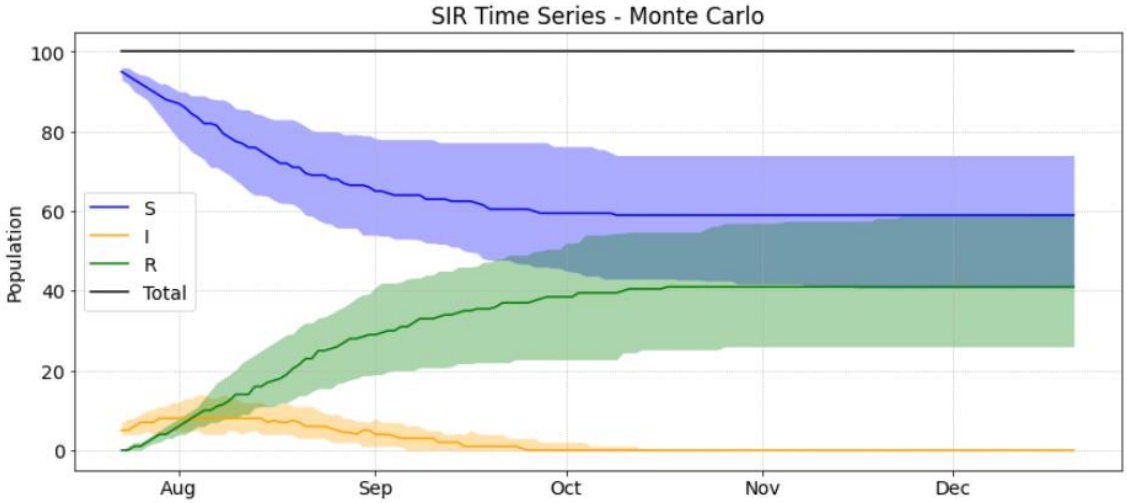

Figure 17: *The result of a vectorized model where all populations are divided into 10 cohorts and a limited degree of mixing between cohorts occurs.*

4 Conclusion

PyCoMod was developed to reduce the time and effort required to build, solve, and analyze detailed, stochastic compartment models. Although initially developed to model COVID-19 transmission in CAF operational settings, PyCoMod can be used to develop a variety of models outside this realm. The most obvious example is employing PyCoMod to model other infectious diseases, whether airborne (e.g., influenza), water-borne (e.g., schistosomiasis, hepatitis) or vector-borne (e.g., malaria, dengue). Several such diseases would be of concern to military operations around the world. Applying PyCoMod to vector-borne diseases would likely involve modelling both the human population and the vector (e.g., mosquitoes) in tandem [6]. Many diseases relevant to the military have previously been modelled using compartment models [7], however they have typically been analyzed in a civilian context and may not consider aspects of the population or environment that are particular to a CAF population, such as the unique facilities and conditions of a deployment or exercise. PyCoMod's features were developed specifically to efficiently model such unique characteristics of populations operating in an infectious disease setting. These features can be employed by other researchers in creating compartment models tailored to other unique populations or situations.

Outside the realm of epidemiology, compartment models fall into the more general class of modelling known as system dynamics, which are used in a wide range of fields including, for example, personnel management and training logistics [8]. PyCoMod's capabilities are not exclusive to disease modelling and may be applied wherever system dynamics models are used, and its unique features may be beneficial over existing tools in those fields. The package has been made publicly available on DRDC's GitHub page [9] in order to give access to as many interested parties as possible. This was especially beneficial during the COVID-19 pandemic given the heightened need for researchers to be able to collaborate remotely.

References

[1] van den Hoogen, J., and Okazawa, S., (2021, October). A Stochastic Model of COVID-19 Infections During a Large-Scale Canadian Army Exercise. NATO Operations Research and Analysis (OR&A) conference. Defence Research and Development Canada, External Literature, DRDC-RDDC-2022-P143.

[2] Brauer, F., (2008). Compartmental Models in Epidemiology. Mathematical Epidemiology, 1945, pp. 19–79. https://doi.org/10.1007/978-3-540-78911-6_2.

[3] Hethcote, H.W., (2000). The mathematics of infectious diseases. SIAM review, 42(4), pp. 599–653.

[4] Wikipedia contributors, (2022, May 29). Compartmental models in epidemiology. In Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Compartmental_models_in_epidemiology&oldid=1090396756. (Accessed date: 1 June 2022).

[5] Python 3.11.4 Documentation: Functional Programming HOWTO, (2023, June). Small functions and the lambda expression. https://docs.python.org/3/howto/functional.html#small-functions-and-the-lambda-expression. (Accessed date: 12 June 2023).

[6] Leggat, P.A., (2010). Tropical diseases of military importance: A Centennial Perspective. *Journal of Military and Veterans Health*, 18(4), pp. 25–31.

[7] Brauer, F., (2017). Mathematical epidemiology: Past, present, and future. Infectious Disease Modelling, 2(2), pp. 113–127.

[8] Séguin, R., (2015). PARSim, a Simulation Model of the Royal Canadian Air Force (RCAF) Pilot Occupation—An Assessment of the Pilot Occupation Sustainability under High Student Production and Reduced Flying Rates, in proceedings of ICORES 2015, pp. 51–62.

[9] Okazawa, S., van den Hoogen, J., and Guillouzic, S., (2021). OS_PyCoMod. GitHub, https://github.com/DND-DRDC-RDDC/OS_PyCoMod. (Accessed date: 6 September 2023).

Annex A Python Compartment Modelling function and object reference table

For ease of reference, the main elements of the PyCoMod package are summarized in Table A.1 to Table A.4.

Table A.1: *PyCoMod classes.*

Class name	Description	Signature	Usage example	Refer to Section
`Model`	Base class for PyCoMod models.	`Model(self, init=None)`	`class SimpleSIR(pcm.Model):`	3.1
`Pool`	Model compartment with initial value.	`Pool(self, value=1)`	`self.S = pcm.Pool(95)`	3.1

`Equation`	Quantity derived from other elements of the model through an equation.	`Equation(self, eq_func=lambda: 1)`	`self.N = pcm.Equation(`
			`  lambda: self.S() + self.I()`
			`  + self.R())`
				3.1

`Parameter`	Model parameter.	`Parameter(self, value=1)`	`B = self.Parameter(0.2)`	3.1

`Flow`	Rate equation that specifies movement dynamics between source and destination pools.	`Flow(self, rate_func=lambda: 1,`
		`  src=None, dest=None,`
		`  priority=False, init=False)`
			`self.Fir = pcm.Flow(`
			`  lambda: self.g() * self.I(),`
			`  src=self.I, dest=self.R)`
				3.1

`RunManager`	Keeps track of multiple models, run settings and output so that batches of runs can be automated.	`RunManager(self)`	`mgr = pcm.RunManager()`	3.1

`Plotter`	Custom interface to Matplotlib figures to plot the output of PyCoMod runs.	`Plotter(self, figsize=(14, 6),`
		`  fontsize=12, title=None,`
		`  xlabel=None, ylabel=None,`
		`  ylimit=None)`
			`plt = pcm.plotter(`
			`  title='SIR Time Series',`
			`  ylabel='Population',`
			`  fontsize=14)`
				3.1

`Sample`	Stochastic parameter value sampled at the start of each model run from a random number generator of a specified distribution.	`Sample(self, sample_func=lambda: 1)`	`pcm.Sample(`
			`  lambda: rng.normal(self.b_m(),`
			`  self.b_s()))`
				3.3

`Step`	Parameter that changes to specific values at specific times.	`Step(self, values, times,` `  default=0)`	`self.b = pcm.Step([0.2, 0.13, 0.2],` `  [0, 7, 21])`	3.6

Class name	Description	Signature	Usage example	Refer to Section
`Impulse`	Parameter that holds a value only for the timestep that contains the impulse time.	`Impulse(self, values, times, default=0)`	`self.b = pcm.Impulse([0.5, 0.5, 0.5], [10, 25, 45], 0.2)`	3.6

Table A.2: *Model methods.*

Method name	Description	Signature	Usage example	Refer to Section
`build`	Define model by overriding the build method of the Model base class.	`build(self)`	`def build(self):`	3.1
`set_output`	Specify outputs captured for analysis.	`set_output(self, *args)`	`self.set_output('S', 'I', 'R')`	3.1
`write_excel_init`	Generate Excel file template to initialize this model.	`write_excel_init(self, filename=None)`	`m3.write_excel_init('init_mix.xlsx')`	3.5

Table A.3: *RunManager methods.*

Function/ Object name	Description	Signature	Usage example	Refer to Section
`run`	Run model while supplying name and run settings.	`run(self, model, init=None, duration=None, label=None, dt=None, start_date=None, start_time=None)`	`mgr.run(m, duration=150, label='My run')`	3.1
`run_mc`	Run a model in Monte Carlo mode	`run_mc(self, model, init=None, duration=None, label=None, dt=None, start_date=None, start_time=None, reps=None)`	`mgr.run_mc(m2, duration=150, reps=100, label='My run - mc')`	3.3

Table A.4: *Plotter methods.*

Function/ Object name	Description	Signature	Usage example	Refer to Section
`plot`	Create figures of the specified model elements for a given model run.	`plot(self, run, elements, **kwargs)`	`plt.plot(mgr['My run'], 'S',` `color='blue', label = 'S')`	3.1

`plot_mc`	Create figures of a Monte Carlo simulation.	`plot_mc(self, run, elements,`
		`**kwargs)`
			`plt.plot_mc(mgr['My run - mc'], 'S',`
			`color='blue', interval=50,`
			`label = 'S')`
				3.3

List of symbols/abbreviations/acronyms/initialisms

CAF	Canadian Armed Forces
COVID-19	Coronavirus Disease 2019
DRDC	Defence Research and Development Canada
E	exposed compartment
I	infectious compartment
OS	open science
pcm	Python local object name for imported PyCoMod package
PyCoMod	Python Compartment Modelling
R	recovered compartment
reps	replications
RNG	random number generator
S	susceptible compartment
SARS-CoV-2	severe acute respiratory syndrome coronavirus 2
SEIR	susceptible-exposed-infectious-recovered
SIR	susceptible-infectious-recovered

DOCUMENT CONTROL DATA

*Security markings for the title, authors, abstract and keywords must be entered when the document is sensitive

1. ORIGINATOR (Name and address of the organization preparing the document. A DRDC Centre sponsoring a contractor's report, or tasking agency, is entered in Section 8.)

DRDC – Centre for Operational Research and Analysis
Defence Research and Development Canada
Carling Campus, 60 Moodie Drive, Building 7S.2
Ottawa, Ontario K1A 0K2
Canada

2a. SECURITY MARKING (Overall security marking of the document including special supplemental markings if applicable.)

CAN UNCLASSIFIED

2b. CONTROLLED GOODS

NON-CONTROLLED GOODS
DMC A

3. TITLE (The document title and sub-title as indicated on the title page.)

PyCoMod (Python Compartment Modelling) Programming Reference

4. AUTHORS (Last name, followed by initials – ranks, titles, etc., not to be used)

Okazawa, S.; van den Hoogen, J.; Guillouzic, S.

5. DATE OF PUBLICATION (Month and year of publication of document.)	6a. NO. OF PAGES (Total pages, including Annexes, excluding DCD, covering and verso pages.)	6b. NO. OF REFS (Total references cited.)
October 2023	32	9

7. DOCUMENT CATEGORY (e.g., Scientific Report, Contract Report, Scientific Letter.)

Reference Document

8. SPONSORING CENTRE (The name and address of the department project office or laboratory sponsoring the research and development.)

DRDC – Centre for Operational Research and Analysis
Defence Research and Development Canada
Carling Campus, 60 Moodie Drive, Building 7S.2
Ottawa, Ontario K1A 0K2
Canada

9a. PROJECT OR GRANT NO. (If appropriate, the applicable research and development project or grant number under which the document was written. Please specify whether project or grant.)	9b. CONTRACT NO. (If appropriate, the applicable number under which the document was written.)
CVPE_003	

10a. DRDC PUBLICATION NUMBER (The official document number by which the document is identified by the originating activity. This number must be unique to this document.)	10b. OTHER DOCUMENT NO(s). (Any other numbers which may be assigned this document either by the originator or by the sponsor.)
DRDC-RDDC-2023-D111	

11a. FUTURE DISTRIBUTION WITHIN CANADA (Approval for further dissemination of the document. Security classification must also be considered.)

Public release

11b. FUTURE DISTRIBUTION OUTSIDE CANADA (Approval for further dissemination of the document. Security classification must also be considered.)

12. KEYWORDS, DESCRIPTORS or IDENTIFIERS (Use semi-colon as a delimiter.)

Python; Epidemic Models; Disease Transmission; Compartment Models; System Dynamics

13a. ABSTRACT (When available in the document, the French version of the abstract must be included here.)

This Reference Document describes the PyCoMod (Python Compartment Modelling) code package. PyCoMod is used to build and run compartment models, such as susceptible-infectious-recovered (SIR) models of infectious disease. The package was initially developed to support analyses of the spread of Coronavirus Disease 2019 (COVID-19) in specific scenarios relevant to the Canadian Armed Forces (CAF) during the pandemic in 2020 and 2021. Over the course of multiple studies conducted during this period in collaboration with the Canadian Forces Health Services Group, the package evolved to include many features making it useful as a general modelling and simulation tool. The use of PyCoMod and its features will be described in detail in this Document.

13b. Résumé (when available in the document, the French version of the abstract must be included here)

Le présent document de référence décrit le code du paquet PyCoMod (modélisation à compartiments Python). PyCoMod est utilisé pour créer et exécuter des modèles à compartiments, comme un modèle Susceptible-Infecté-Rétabli relatif à une maladie infectieuse. Le paquet a été élaboré initialement à l'appui des analyses sur la propagation de la COVID-19 dans le cadre de scénarios précis qui concernaient les Forces armées canadiennes pendant la pandémie en 2020 et en 2021. Au cours de multiples études effectuées pendant cette période en collaboration avec le Centre des services de santé des Forces canadiennes, le paquet a évolué et comprend maintenant de nombreuses fonctionnalités qui le rendent utile comme outil de modélisation et de simulation général. L'utilisation de PyCoMod et de ses fonctionnalités sera décrite en détail dans le présent document.