fpsim

Release 0.19.2

Institute for Disease Modeling

Dec 07, 2022

CONTENTS

FPsim is a stochastic agent-based model, written in Python, for exploring and analyzing family planning. In addition, the repository contains scripts for analyzing model output and tests for testing functionality.

CHAPTER

ONE

FULL CONTENTS

1.1 FPsim overview

FPsim is currently under development.

1.1.1 User Guide

FPsim is designed as an open-source tool for family planning research. However, it is not a silver bullet tool. It is designed to answer complex questions about emerging dynamics in complex social and behavioral systems. Its strength stems from a life-course approach, which allows researchers to examine how compounding and temporal effects unfold over women's lives, and how these individual-level changes lead to macro-level outcomes.

Before using FPsim, please refer to the following guidelines:

- FPsim is only as good as the data and assumptions provided. Be sure you are familiar with both before using FPsim.
- FPsim is not a replacement for good data. The model cannot tell you what demand for a hypothetical method will be.
- FPsim is not a replacement for descriptive statistics. Before using FPsim, assess your primary research question(s). Can they be answered using descriptive statistics?
- FPsim cannot predict exogenous events. Use caution when interpreting and presenting results. For example, FPsim cannot predict regional conflicts or pandemics, nor their impacts on FP services.

1.1.2 Repo Structure

The structure is as follows:

- FPsim, in the folder `fpsim`, is a standalone Python library for performing family planning analyses.
- Within `fpsim`, the `locations` folder contains parameters and input data for all countries currently calibrated.
- Docs are in the `docs` folder.
- Examples are in the `examples` folder.
- Tests are in the `tests` folder.

1.1.3 Installation

Run `pip install fpsim` to install and its dependencies from PyPI. Alternatively, clone the repository and run `pip install -e .` (including the final dot!).

1.1.4 Documentation

Documentation is available at https://docs.fpsim.org.

1.1.5 Contributing

Style guide

Please follow the starsim style guide at: https://github.com/amath-idm/styleguide

Issues

- Everything you're working on must be linked to an issue. If you notice that something needs to be done (even small things or things nearly finished) and there isn't an issue for it, create an issue. This helps track who is doing what and why.
- Label issues you are currently working on with `in progress` for tracking purposes - and to avoid accidental replication of work.
- High priority issues are organized from top (most urgent) to bottom (least urgent) and can be labelled with `urgent` or `blocking` as appropriate. If you are working on something that is urgent or blocks other development, please set a reasonable deadline for review (can be updated, of course!)
- The Hydra Head Effect: Often when you solve one issue, two more pop up in its place. When this happens, close the original issue and start new issues (linked) to be triaged.
- If your issue has more than two distinct tasks associated with it, please include a check list in the text, so that we can track which components of the issue have been resolved and which need to be supported.
- If your issue is a bug that was not caught by test, and includes a specific expected value that can be hard-checked, please either include or request a test patch so that a test fails due to the bug

Pull Requests

- ALL PRs should be linked to at least one issue. As above, if you're working on a PR and there's no issue associated with it, you can create an issue. However, before doing so, ask yourself if it really needs to be done.
- All PRs should have another person assigned for review. If assigned to more than one person, use the comment section to assign an issue owner/main reviewer. Use your best judgement here, as roles shift, but in general:
  - @MOBrien-IDM as FPsim lead (approval required to merge)
  - Anyone you've worked with on this issue 1:1
  - @cliffckerr to ensure new feature performance and compatibility with FPsim
  - @mzimmermann-IDM for subject matter expertise, economic and empowerment questions, questions about modeling best practices
  - @avictorious for questions about historical FPsim decisions and subject matter expertise
- Keep PRs as small as possible: e.g., one issue, one PR. Small PRs are easier to review and merge.
- At times there may be a backlog of issues, but there should never be a big backlog of PRs. (If you're unsure whether to make a PR, write a detailed issue first.)
  - What if there are two people working on PRs at the same time?

  - Take a look at the issue priority. The PR addressing the higher priority issue should merge first. Make sure you pull the new master after that merge before you push changes for your PR. If both issues are high priority, the one with more time-sensitive commits should be merged first. If you're unsure, ask.

- If we do have a backlog of PRs, it's fine to make a new branch off your current PR, and make a new PR from that. These "cumulative PRs are not ideal, but they are better than creating merge conflicts with yourself!
- Before starting work, always ensure you've pulled from master. If you spend more than a few days on your PR, make sure you pull from master regularly. Before making a PR, ensure that your branch is up to date with master.
- Please create a draft PR on an active branch as soon as you're ready. Be generous in creating draft PRs. It helps with transparency and allows for quicker support if you run into a problem.
- Make sure tests pass on your PR. If they don't, mark the PR as draft until they do.
- Even if your work isn't ready for a PR, push it regularly. A guiding principle is to commit every few minutes and push to your branch every 1-2 hours.
- Every PR that adds a new feature or new functionality which can be hard-checked (so, excluding plotting functionality etc.) should include a corresponding unittest

Testing

- Development and Debugging
  - **Developers are responsible for ensuring the functionality of new features they develop**
    * Debugging and testing code are core features of ensuring functionality
    * When debugging in active development mode, ensure that your new feature is compatible with not only a single run of FPsim, but also the multisim scenarios
    * Ensure new features are compatible with introducing a novel method in scenarios
    * Use example_scens.py to quickly debug your new feature during development
- **Test Coverage**
  - Every time a new feature is added, the developer should develop a unittest which checks the basic implementation of the feature
  - A unittest is simply a function starting with "test" that implements a feature as succinctly as possibly, and checks the expected output with an assertion
  - If you're having trouble starting a unittest feel free to look at some examples here
  - Some test suites organize the tests into a class with a configuration function called `setUp()`. After implementing a unittest in such a class you may want to take advantage of the shared assets defined in `setUp()` to minimize the number of lines of code in your test.
  - The new unittest should follow style guidelines laid out in the starsim style guide
  - The new test should contain a docstring that details what is being tested, how it is tested (what it's being checked against), and the expected value
  - The test should display error message information that is sufficient to create a bug report (summary, expected value, and actual value)

1.1.6 Disclaimer

The code in this repository was developed by IDM and other collaborators to support our joint research on family planning. We've made it publicly available under the MIT License to provide others with a better understanding of our research and an opportunity to build upon it for their own work. Note that FPsim depends on a number of user-installed Python packages that can be installed automatically via `pip install`. We make no representations that the code works as intended or that we will provide support, address issues that are found, or accept pull requests. You are welcome to create your own fork and modify the code to suit your own modeling needs as contemplated under the MIT License. See the contributing and code of conduct READMEs for more information.

1.2 Tutorials

These tutorials walk through how to use FPsim. If you want to explore them interactively, you can run them on Binder via http://tutorials.fpsim.org. To run locally, start a Jupyter environment in this folder (`docs/tutorials`). You can use either `jupyter lab` or `jupyter notebook` to run these tutorials.

1.2.1 T1 - Getting started

Installing and getting started with FPsim is quite simple.

To install, just type `pip install -e .`. If it worked, you should be able to import FPsim with `import fpsim as fp`.

The basic design philosophy of FPsim is: **common tasks should be simple**. For example:

- Defining parameters
- Running a simulation
- Plotting results

This tutorial walks you through how to define paramaters and run the simulation. The next tutorial will show you how to plot the results of a simulation.

1.2.1.1 Hello world

To create, run, and plot a sim with default options is just:

[1]:
```
import fpsim as fp

sim = fp.Sim()
sim.run()
fig = sim.plot()
```

```
/home/docs/checkouts/readthedocs.org/user_builds/institute-for-disease-modeling-fpsim/
↪envs/v0.19.2/lib/python3.9/site-packages/tqdm/auto.py:22: TqdmWarning: IProgress not␣
↪found. Please update jupyter and ipywidgets. See https://ipywidgets.readthedocs.io/en/
↪stable/user_install.html
  from .autonotebook import tqdm as notebook_tqdm
```

```
  Running 1960 of 2020 ( 0/721) (0.09 s)  ---------------------- 0%
  Running 1961 of 2020 (12/721) (0.21 s)  ---------------------- 2%
  Running 1962 of 2020 (24/721) (0.35 s)  ---------------------- 3%
  Running 1963 of 2020 (36/721) (0.48 s)  •--------------------- 5%
```

(continues on next page)

(continued from previous page)

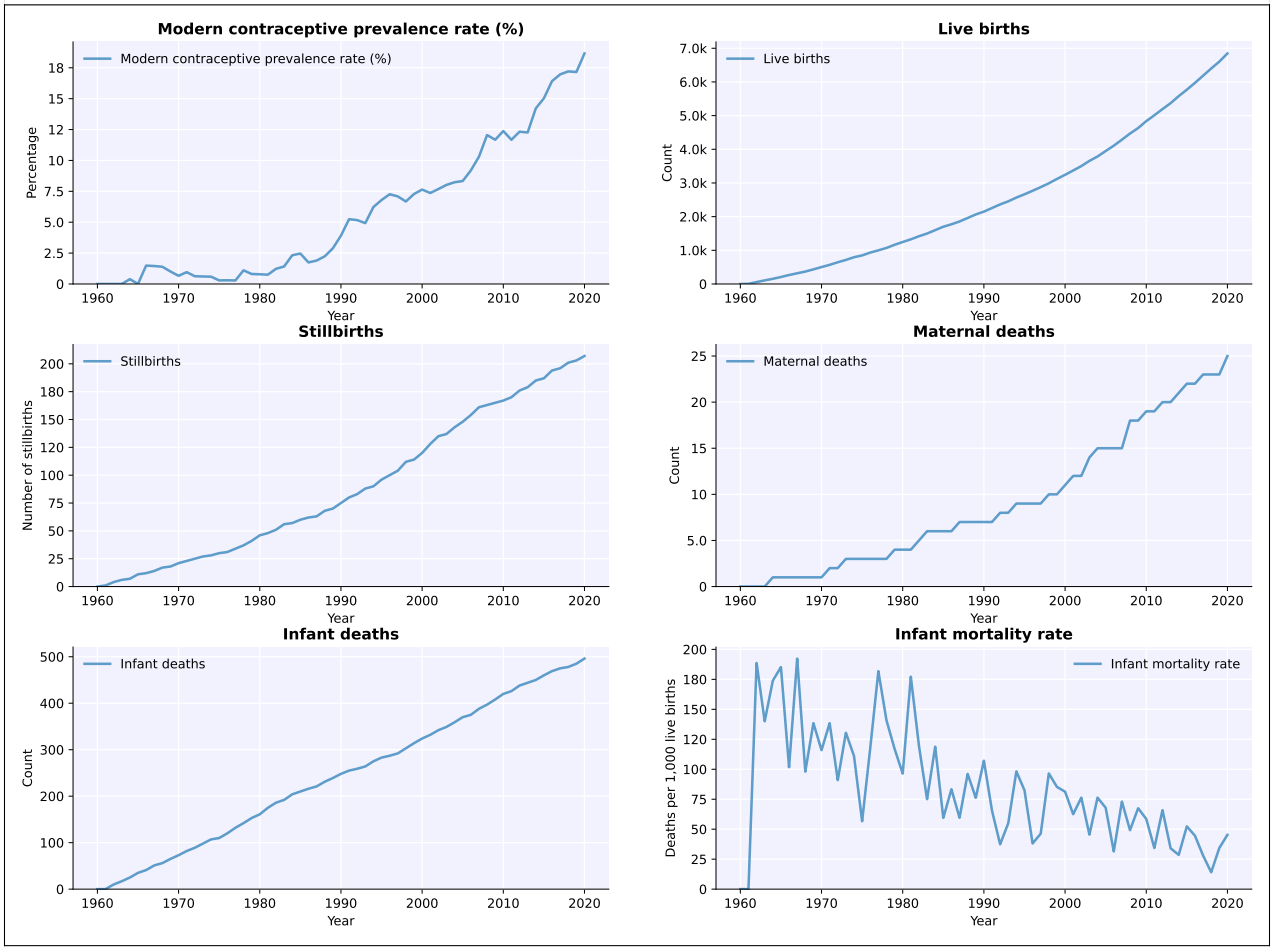
```
Running 1964 of 2020 (48/721) (0.61 s)  •------------------- 7%
Running 1965 of 2020 (60/721) (0.74 s)  •------------------- 8%
Running 1966 of 2020 (72/721) (0.87 s)  ••------------------ 10%
Running 1967 of 2020 (84/721) (1.00 s)  ••------------------ 12%
Running 1968 of 2020 (96/721) (1.13 s)  ••------------------ 13%
Running 1969 of 2020 (108/721) (1.27 s)  •••----------------- 15%
Running 1970 of 2020 (120/721) (1.40 s)  •••----------------- 17%
Running 1971 of 2020 (132/721) (1.53 s)  •••----------------- 18%
Running 1972 of 2020 (144/721) (1.67 s)  ••••---------------- 20%
Running 1973 of 2020 (156/721) (1.80 s)  ••••---------------- 22%
Running 1974 of 2020 (168/721) (1.94 s)  ••••---------------- 23%
Running 1975 of 2020 (180/721) (2.07 s)  •••••--------------- 25%
Running 1976 of 2020 (192/721) (2.21 s)  •••••--------------- 27%
Running 1977 of 2020 (204/721) (2.34 s)  •••••--------------- 28%
Running 1978 of 2020 (216/721) (2.48 s)  ••••••-------------- 30%
Running 1979 of 2020 (228/721) (2.61 s)  ••••••-------------- 32%
Running 1980 of 2020 (240/721) (2.75 s)  ••••••-------------- 33%
Running 1981 of 2020 (252/721) (2.88 s)  •••••••------------- 35%
Running 1982 of 2020 (264/721) (3.02 s)  •••••••------------- 37%
Running 1983 of 2020 (276/721) (3.16 s)  •••••••------------- 38%
Running 1984 of 2020 (288/721) (3.29 s)  ••••••••------------ 40%
Running 1985 of 2020 (300/721) (3.43 s)  ••••••••------------ 42%
Running 1986 of 2020 (312/721) (3.57 s)  ••••••••------------ 43%
Running 1987 of 2020 (324/721) (3.71 s)  •••••••••----------- 45%
Running 1988 of 2020 (336/721) (3.85 s)  •••••••••----------- 47%
Running 1989 of 2020 (348/721) (3.99 s)  •••••••••----------- 48%
Running 1990 of 2020 (360/721) (4.13 s)  ••••••••••---------- 50%
Running 1991 of 2020 (372/721) (4.27 s)  ••••••••••---------- 52%
Running 1992 of 2020 (384/721) (4.41 s)  ••••••••••---------- 53%
Running 1993 of 2020 (396/721) (4.55 s)  •••••••••••--------- 55%
Running 1994 of 2020 (408/721) (4.70 s)  •••••••••••--------- 57%
Running 1995 of 2020 (420/721) (4.84 s)  •••••••••••--------- 58%
Running 1996 of 2020 (432/721) (4.98 s)  ••••••••••••-------- 60%
Running 1997 of 2020 (444/721) (5.13 s)  ••••••••••••-------- 62%
Running 1998 of 2020 (456/721) (5.27 s)  ••••••••••••-------- 63%
Running 1999 of 2020 (468/721) (5.41 s)  •••••••••••••------- 65%
Running 2000 of 2020 (480/721) (5.56 s)  •••••••••••••------- 67%
Running 2001 of 2020 (492/721) (5.71 s)  •••••••••••••------- 68%
Running 2002 of 2020 (504/721) (5.86 s)  ••••••••••••••------ 70%
Running 2003 of 2020 (516/721) (6.01 s)  ••••••••••••••------ 72%
Running 2004 of 2020 (528/721) (6.28 s)  ••••••••••••••------ 73%
Running 2005 of 2020 (540/721) (6.43 s)  •••••••••••••••----- 75%
Running 2006 of 2020 (552/721) (6.58 s)  •••••••••••••••----- 77%
Running 2007 of 2020 (564/721) (6.73 s)  •••••••••••••••----- 78%
Running 2008 of 2020 (576/721) (6.89 s)  ••••••••••••••••---- 80%
Running 2009 of 2020 (588/721) (7.04 s)  ••••••••••••••••---- 82%
Running 2010 of 2020 (600/721) (7.20 s)  ••••••••••••••••---- 83%
Running 2011 of 2020 (612/721) (7.36 s)  •••••••••••••••••--- 85%
Running 2012 of 2020 (624/721) (7.53 s)  •••••••••••••••••--- 87%
Running 2013 of 2020 (636/721) (7.69 s)  •••••••••••••••••--- 88%
Running 2014 of 2020 (648/721) (7.85 s)  ••••••••••••••••••-- 90%
Running 2015 of 2020 (660/721) (8.00 s)  ••••••••••••••••••-- 92%
```

(continues on next page)

(continued from previous page)

```
  Running 2016 of 2020 (672/721) (8.16 s)  ••••••••••••••••••-- 93%
  Running 2017 of 2020 (684/721) (8.33 s)  ••••••••••••••••••- 95%
  Running 2018 of 2020 (696/721) (8.49 s)  ••••••••••••••••••- 97%
  Running 2019 of 2020 (708/721) (8.65 s)  ••••••••••••••••••- 98%
  Running 2020 of 2020 (720/721) (8.82 s)  •••••••••••••••••••• 100%

Final population size: 6139.
Run finished for "None" after 8.8 s
```



1.2.1.2 Defining parameters and running simulations

Parameters are defined as a dictionary. In FPsim, we categorize our parameters as:

Basic parameters Age limits Durations Pregnancy outcomes Fecundity and exposure MCPR

The most common category of parameters to change in FPsim is the basic category, which includes the location (i.e. Senegal, northern India), the starting population, the starting year, and the initial number of agents. We can define thiese as:

[2]:
```
pars = dict(
    n_agents   = 10_000,
    location   = 'senegal',
```

(continues on next page)

(continued from previous page)

```
    start_year = 1960,
    end_year   = 2020,
)
```

Running a simulation is pretty easy. In fact, running a sim with the parameters we defined above is just:

[3]:
```
sim = fp.Sim(pars)
sim.run()
```

```
  Running 1960 of 2020 ( 0/721) (0.20 s)  ---------- 0%
  Running 1961 of 2020 (12/721) (0.38 s)  ---------- 2%
  Running 1962 of 2020 (24/721) (0.57 s)  ---------- 3%
  Running 1963 of 2020 (36/721) (0.76 s)  •--------- 5%
  Running 1964 of 2020 (48/721) (0.96 s)  •--------- 7%
  Running 1965 of 2020 (60/721) (1.16 s)  •--------- 8%
  Running 1966 of 2020 (72/721) (1.36 s)  ••-------- 10%
  Running 1967 of 2020 (84/721) (1.56 s)  ••-------- 12%
  Running 1968 of 2020 (96/721) (1.76 s)  ••-------- 13%
  Running 1969 of 2020 (108/721) (1.97 s)  •••------- 15%
  Running 1970 of 2020 (120/721) (2.18 s)  •••------- 17%
  Running 1971 of 2020 (132/721) (2.40 s)  •••------- 18%
  Running 1972 of 2020 (144/721) (2.61 s)  ••••------ 20%
  Running 1973 of 2020 (156/721) (2.83 s)  ••••------ 22%
  Running 1974 of 2020 (168/721) (3.05 s)  ••••------ 23%
  Running 1975 of 2020 (180/721) (3.28 s)  •••••----- 25%
  Running 1976 of 2020 (192/721) (3.50 s)  •••••----- 27%
  Running 1977 of 2020 (204/721) (3.73 s)  •••••----- 28%
  Running 1978 of 2020 (216/721) (3.97 s)  ••••••---- 30%
  Running 1979 of 2020 (228/721) (4.21 s)  ••••••---- 32%
  Running 1980 of 2020 (240/721) (4.45 s)  ••••••---- 33%
  Running 1981 of 2020 (252/721) (4.69 s)  •••••••--- 35%
  Running 1982 of 2020 (264/721) (4.94 s)  •••••••--- 37%
  Running 1983 of 2020 (276/721) (5.19 s)  •••••••--- 38%
  Running 1984 of 2020 (288/721) (5.58 s)  ••••••••-- 40%
  Running 1985 of 2020 (300/721) (5.84 s)  ••••••••-- 42%
  Running 1986 of 2020 (312/721) (6.11 s)  ••••••••-- 43%
  Running 1987 of 2020 (324/721) (6.38 s)  •••••••••- 45%
  Running 1988 of 2020 (336/721) (6.65 s)  •••••••••- 47%
  Running 1989 of 2020 (348/721) (6.93 s)  •••••••••- 48%
  Running 1990 of 2020 (360/721) (7.21 s)  ••••••••••- 50%
  Running 1991 of 2020 (372/721) (7.50 s)  ••••••••••- 52%
  Running 1992 of 2020 (384/721) (7.79 s)  ••••••••••- 53%
  Running 1993 of 2020 (396/721) (8.09 s)  •••••••••••- 55%
  Running 1994 of 2020 (408/721) (8.40 s)  •••••••••••- 57%
  Running 1995 of 2020 (420/721) (8.71 s)  •••••••••••- 58%
  Running 1996 of 2020 (432/721) (9.02 s)  ••••••••••••- 60%
  Running 1997 of 2020 (444/721) (9.34 s)  ••••••••••••- 62%
  Running 1998 of 2020 (456/721) (9.67 s)  ••••••••••••- 63%
  Running 1999 of 2020 (468/721) (10.00 s)  •••••••••••••- 65%
  Running 2000 of 2020 (480/721) (10.34 s)  •••••••••••••- 67%
  Running 2001 of 2020 (492/721) (10.69 s)  •••••••••••••- 68%
  Running 2002 of 2020 (504/721) (11.04 s)  ••••••••••••••- 70%
  Running 2003 of 2020 (516/721) (11.40 s)  ••••••••••••••- 72%
```

(continues on next page)

(continued from previous page)

```
  Running 2004 of 2020 (528/721) (11.76 s)  •••••••••••••------ 73%
  Running 2005 of 2020 (540/721) (12.14 s)  •••••••••••••------ 75%
  Running 2006 of 2020 (552/721) (12.66 s)  •••••••••••••------ 77%
  Running 2007 of 2020 (564/721) (13.05 s)  •••••••••••••------ 78%
  Running 2008 of 2020 (576/721) (13.45 s)  ••••••••••••••---- 80%
  Running 2009 of 2020 (588/721) (13.86 s)  ••••••••••••••---- 82%
  Running 2010 of 2020 (600/721) (14.28 s)  ••••••••••••••---- 83%
  Running 2011 of 2020 (612/721) (14.70 s)  •••••••••••••••--- 85%
  Running 2012 of 2020 (624/721) (15.13 s)  •••••••••••••••--- 87%
  Running 2013 of 2020 (636/721) (15.58 s)  •••••••••••••••--- 88%
  Running 2014 of 2020 (648/721) (16.03 s)  ••••••••••••••••-- 90%
  Running 2015 of 2020 (660/721) (16.51 s)  ••••••••••••••••-- 92%
  Running 2016 of 2020 (672/721) (16.98 s)  ••••••••••••••••-- 93%
  Running 2017 of 2020 (684/721) (17.47 s)  •••••••••••••••••- 95%
  Running 2018 of 2020 (696/721) (17.97 s)  •••••••••••••••••- 97%
  Running 2019 of 2020 (708/721) (18.48 s)  •••••••••••••••••- 98%
  Running 2020 of 2020 (720/721) (19.00 s)  •••••••••••••••••• 100%

Final population size: 62232.
Run finished for "None" after 19.2 s
```

[3]:
```
Sim(<no label>; n=10000; 1960-2020; results: b=68914 =11805 pop=62232)
```

This will generate a results dictionary `sim.results`. For example, the number of pregnancies in the sim can be found using `sim.results['pregnancies']`.

Rather than creating a parameter dictionary, any valid parameter can also be passed to the sim directly. For example, exactly equivalent to the above is:

[4]:
```
sim = fp.Sim(n_agents=10e3, location='senegal', start_year=1960, end_year=2020)
sim.run()
```

```
  Running 1960 of 2020 ( 0/721) (0.25 s)  ---------- 0%
  Running 1961 of 2020 (12/721) (0.42 s)  ---------- 2%
  Running 1962 of 2020 (24/721) (0.61 s)  ---------- 3%
  Running 1963 of 2020 (36/721) (0.80 s)  •--------- 5%
  Running 1964 of 2020 (48/721) (1.00 s)  •--------- 7%
  Running 1965 of 2020 (60/721) (1.20 s)  •--------- 8%
  Running 1966 of 2020 (72/721) (1.40 s)  ••-------- 10%
  Running 1967 of 2020 (84/721) (1.59 s)  ••-------- 12%
  Running 1968 of 2020 (96/721) (1.80 s)  ••-------- 13%
  Running 1969 of 2020 (108/721) (2.01 s)  •••------- 15%
  Running 1970 of 2020 (120/721) (2.21 s)  •••------- 17%
  Running 1971 of 2020 (132/721) (2.42 s)  •••------- 18%
  Running 1972 of 2020 (144/721) (2.64 s)  ••••------ 20%
  Running 1973 of 2020 (156/721) (2.86 s)  ••••------ 22%
  Running 1974 of 2020 (168/721) (3.08 s)  ••••------ 23%
  Running 1975 of 2020 (180/721) (3.30 s)  •••••----- 25%
  Running 1976 of 2020 (192/721) (3.53 s)  •••••----- 27%
  Running 1977 of 2020 (204/721) (3.76 s)  •••••----- 28%
  Running 1978 of 2020 (216/721) (4.00 s)  ••••••---- 30%
  Running 1979 of 2020 (228/721) (4.24 s)  ••••••---- 32%
  Running 1980 of 2020 (240/721) (4.47 s)  ••••••---- 33%
  Running 1981 of 2020 (252/721) (4.72 s)  •••••••--- 35%
```

(continues on next page)

(continued from previous page)

```
  Running 1982 of 2020 (264/721) (4.97 s)  •••••••---------- 37%
  Running 1983 of 2020 (276/721) (5.22 s)  •••••••---------- 38%
  Running 1984 of 2020 (288/721) (5.47 s)  ••••••••--------- 40%
  Running 1985 of 2020 (300/721) (5.73 s)  ••••••••--------- 42%
  Running 1986 of 2020 (312/721) (6.00 s)  ••••••••--------- 43%
  Running 1987 of 2020 (324/721) (6.27 s)  •••••••••-------- 45%
  Running 1988 of 2020 (336/721) (6.54 s)  •••••••••-------- 47%
  Running 1989 of 2020 (348/721) (6.82 s)  •••••••••-------- 48%
  Running 1990 of 2020 (360/721) (7.10 s)  ••••••••••------- 50%
  Running 1991 of 2020 (372/721) (7.39 s)  ••••••••••------- 52%
  Running 1992 of 2020 (384/721) (7.68 s)  ••••••••••------- 53%
  Running 1993 of 2020 (396/721) (7.98 s)  ••••••••••••------ 55%
  Running 1994 of 2020 (408/721) (8.28 s)  ••••••••••••------ 57%
  Running 1995 of 2020 (420/721) (8.59 s)  ••••••••••••------ 58%
  Running 1996 of 2020 (432/721) (8.90 s)  •••••••••••••----- 60%
  Running 1997 of 2020 (444/721) (9.22 s)  •••••••••••••----- 62%
  Running 1998 of 2020 (456/721) (9.73 s)  •••••••••••••----- 63%
  Running 1999 of 2020 (468/721) (10.06 s)  ••••••••••••••---- 65%
  Running 2000 of 2020 (480/721) (10.40 s)  ••••••••••••••---- 67%
  Running 2001 of 2020 (492/721) (10.75 s)  ••••••••••••••---- 68%
  Running 2002 of 2020 (504/721) (11.10 s)  ••••••••••••••---- 70%
  Running 2003 of 2020 (516/721) (11.46 s)  ••••••••••••••---- 72%
  Running 2004 of 2020 (528/721) (11.82 s)  ••••••••••••••---- 73%
  Running 2005 of 2020 (540/721) (12.20 s)  ••••••••••••••••--- 75%
  Running 2006 of 2020 (552/721) (12.58 s)  ••••••••••••••••--- 77%
  Running 2007 of 2020 (564/721) (12.97 s)  ••••••••••••••••--- 78%
  Running 2008 of 2020 (576/721) (13.37 s)  ••••••••••••••••--- 80%
  Running 2009 of 2020 (588/721) (13.78 s)  ••••••••••••••••--- 82%
  Running 2010 of 2020 (600/721) (14.20 s)  ••••••••••••••••--- 83%
  Running 2011 of 2020 (612/721) (14.62 s)  •••••••••••••••••-- 85%
  Running 2012 of 2020 (624/721) (15.05 s)  •••••••••••••••••-- 87%
  Running 2013 of 2020 (636/721) (15.50 s)  •••••••••••••••••-- 88%
  Running 2014 of 2020 (648/721) (15.96 s)  •••••••••••••••••-- 90%
  Running 2015 of 2020 (660/721) (16.43 s)  •••••••••••••••••-- 92%
  Running 2016 of 2020 (672/721) (16.90 s)  •••••••••••••••••-- 93%
  Running 2017 of 2020 (684/721) (17.39 s)  ••••••••••••••••••- 95%
  Running 2018 of 2020 (696/721) (17.88 s)  ••••••••••••••••••- 97%
  Running 2019 of 2020 (708/721) (18.39 s)  ••••••••••••••••••- 98%
  Running 2020 of 2020 (720/721) (18.91 s)  •••••••••••••••••••• 100%

Final population size: 62232.
Run finished for "None" after 19.1 s
```

[4]:
```
Sim(<no label>; n=10000; 1960-2020; results: b=68914 =11805 pop=62232)
```

You can mix and match too – pass in a parameter dictionary with default options, and then include other parameters as keywords (including overrides; keyword arguments take precedence). For example:

[5]:
```
sim = fp.Sim(pars, n_agents=100) # Use parameters defined above, except start with 100␣
↪agents instead of 10,000
sim.run()
```

```
  Running 1960 of 2020 ( 0/721) (0.05 s)  ---------- 0%
```

(continues on next page)

(continued from previous page)

```
  Running 1961 of 2020 (12/721) (0.17 s)  ---------- 2%
  Running 1962 of 2020 (24/721) (0.28 s)  ---------- 3%
  Running 1963 of 2020 (36/721) (0.40 s)  •--------- 5%
  Running 1964 of 2020 (48/721) (0.52 s)  •--------- 7%
  Running 1965 of 2020 (60/721) (0.64 s)  •--------- 8%
  Running 1966 of 2020 (72/721) (0.76 s)  ••-------- 10%
  Running 1967 of 2020 (84/721) (0.88 s)  ••-------- 12%
  Running 1968 of 2020 (96/721) (1.00 s)  ••-------- 13%
  Running 1969 of 2020 (108/721) (1.12 s)  •••------- 15%
  Running 1970 of 2020 (120/721) (1.24 s)  •••------- 17%
  Running 1971 of 2020 (132/721) (1.36 s)  •••------- 18%
  Running 1972 of 2020 (144/721) (1.48 s)  ••••------ 20%
  Running 1973 of 2020 (156/721) (1.60 s)  ••••------ 22%
  Running 1974 of 2020 (168/721) (1.72 s)  ••••------ 23%
  Running 1975 of 2020 (180/721) (1.84 s)  •••••----- 25%
  Running 1976 of 2020 (192/721) (1.97 s)  •••••----- 27%
  Running 1977 of 2020 (204/721) (2.09 s)  •••••----- 28%
  Running 1978 of 2020 (216/721) (2.21 s)  ••••••---- 30%
  Running 1979 of 2020 (228/721) (2.33 s)  ••••••---- 32%
  Running 1980 of 2020 (240/721) (2.45 s)  ••••••---- 33%
  Running 1981 of 2020 (252/721) (2.58 s)  •••••••--- 35%
  Running 1982 of 2020 (264/721) (2.70 s)  •••••••--- 37%
  Running 1983 of 2020 (276/721) (2.82 s)  •••••••--- 38%
  Running 1984 of 2020 (288/721) (2.95 s)  ••••••••-- 40%
  Running 1985 of 2020 (300/721) (3.07 s)  ••••••••-- 42%
  Running 1986 of 2020 (312/721) (3.19 s)  ••••••••-- 43%
  Running 1987 of 2020 (324/721) (3.31 s)  •••••••--- 45%
  Running 1988 of 2020 (336/721) (3.43 s)  •••••••--- 47%
  Running 1989 of 2020 (348/721) (3.55 s)  •••••••--- 48%
  Running 1990 of 2020 (360/721) (3.68 s)  ••••••---- 50%
  Running 1991 of 2020 (372/721) (3.80 s)  ••••••---- 52%
  Running 1992 of 2020 (384/721) (3.92 s)  ••••••---- 53%
  Running 1993 of 2020 (396/721) (4.04 s)  •••••----- 55%
  Running 1994 of 2020 (408/721) (4.17 s)  •••••----- 57%
  Running 1995 of 2020 (420/721) (4.29 s)  •••••----- 58%
  Running 1996 of 2020 (432/721) (4.41 s)  ••••------ 60%
  Running 1997 of 2020 (444/721) (4.53 s)  ••••------ 62%
  Running 1998 of 2020 (456/721) (4.66 s)  ••••------ 63%
  Running 1999 of 2020 (468/721) (4.78 s)  •••------- 65%
  Running 2000 of 2020 (480/721) (4.90 s)  •••------- 67%
  Running 2001 of 2020 (492/721) (5.03 s)  •••------- 68%
  Running 2002 of 2020 (504/721) (5.15 s)  ••------- 70%
  Running 2003 of 2020 (516/721) (5.28 s)  ••------- 72%
  Running 2004 of 2020 (528/721) (5.41 s)  ••------- 73%
  Running 2005 of 2020 (540/721) (5.53 s)  •-------- 75%
  Running 2006 of 2020 (552/721) (5.66 s)  •-------- 77%
  Running 2007 of 2020 (564/721) (5.78 s)  •-------- 78%
  Running 2008 of 2020 (576/721) (5.91 s)  ---------- 80%
  Running 2009 of 2020 (588/721) (6.03 s)  ---------- 82%
  Running 2010 of 2020 (600/721) (6.16 s)  ---------- 83%
  Running 2011 of 2020 (612/721) (6.28 s)  •--------- 85%
  Running 2012 of 2020 (624/721) (6.41 s)  •--------- 87%
```

(continues on next page)

(continued from previous page)

```
  Running 2013 of 2020 (636/721) (6.53 s)  ••••••••••••••••••--- 88%
  Running 2014 of 2020 (648/721) (6.66 s)  ••••••••••••••••••-- 90%
  Running 2015 of 2020 (660/721) (6.78 s)  ••••••••••••••••••-- 92%
  Running 2016 of 2020 (672/721) (6.91 s)  ••••••••••••••••••-- 93%
  Running 2017 of 2020 (684/721) (7.03 s)  •••••••••••••••••••- 95%
  Running 2018 of 2020 (696/721) (7.16 s)  •••••••••••••••••••- 97%
  Running 2019 of 2020 (708/721) (7.28 s)  •••••••••••••••••••- 98%
  Running 2020 of 2020 (720/721) (7.41 s)  •••••••••••••••••••• 100%

Final population size: 597.
Run finished for "None" after 7.4 s
```

[5]:
```
Sim(<no label>; n=100; 1960-2020; results: b=672 =126 pop=597)
```

Now you know how to set up an initial sim using FPsim. In the next tutorial, we will explore plotting options.

1.2.2 T2 - Plotting with FPsim

Basic plotting with FPsim is pretty straightforward.

If you're new to FPsim, please start with tutorial T1 in order to install FPsim and learn how to run a basic simulation.

The basic design philosophy of FPsim is: **common tasks should be simple**. For example:

- Defining parameters
- Running a simulation
- Plotting results

In tutorial T1 we learned how to define parameters and run a simulation. In this tutorial, we will take on our third simple task and plot the results of a simulation.

In tutorial T1, we showed you that to create, run, and plot a sim with default options is just:

[1]:
```
import fpsim as fp

sim = fp.Sim()
sim.run()
fig = sim.plot()
```

```
/home/docs/checkouts/readthedocs.org/user_builds/institute-for-disease-modeling-fpsim/
↪envs/v0.19.2/lib/python3.9/site-packages/tqdm/auto.py:22: TqdmWarning: IProgress not␣
↪found. Please update jupyter and ipywidgets. See https://ipywidgets.readthedocs.io/en/
↪stable/user_install.html
  from .autonotebook import tqdm as notebook_tqdm
```

```
  Running 1960 of 2020 ( 0/721) (0.09 s)  ---------------------- 0%
  Running 1961 of 2020 (12/721) (0.21 s)  ---------------------- 2%
  Running 1962 of 2020 (24/721) (0.35 s)  ---------------------- 3%
  Running 1963 of 2020 (36/721) (0.48 s)  •--------------------- 5%
  Running 1964 of 2020 (48/721) (0.61 s)  •--------------------- 7%
  Running 1965 of 2020 (60/721) (0.74 s)  •--------------------- 8%
  Running 1966 of 2020 (72/721) (0.88 s)  ••-------------------- 10%
  Running 1967 of 2020 (84/721) (1.01 s)  ••-------------------- 12%
  Running 1968 of 2020 (96/721) (1.14 s)  ••-------------------- 13%
```

(continues on next page)

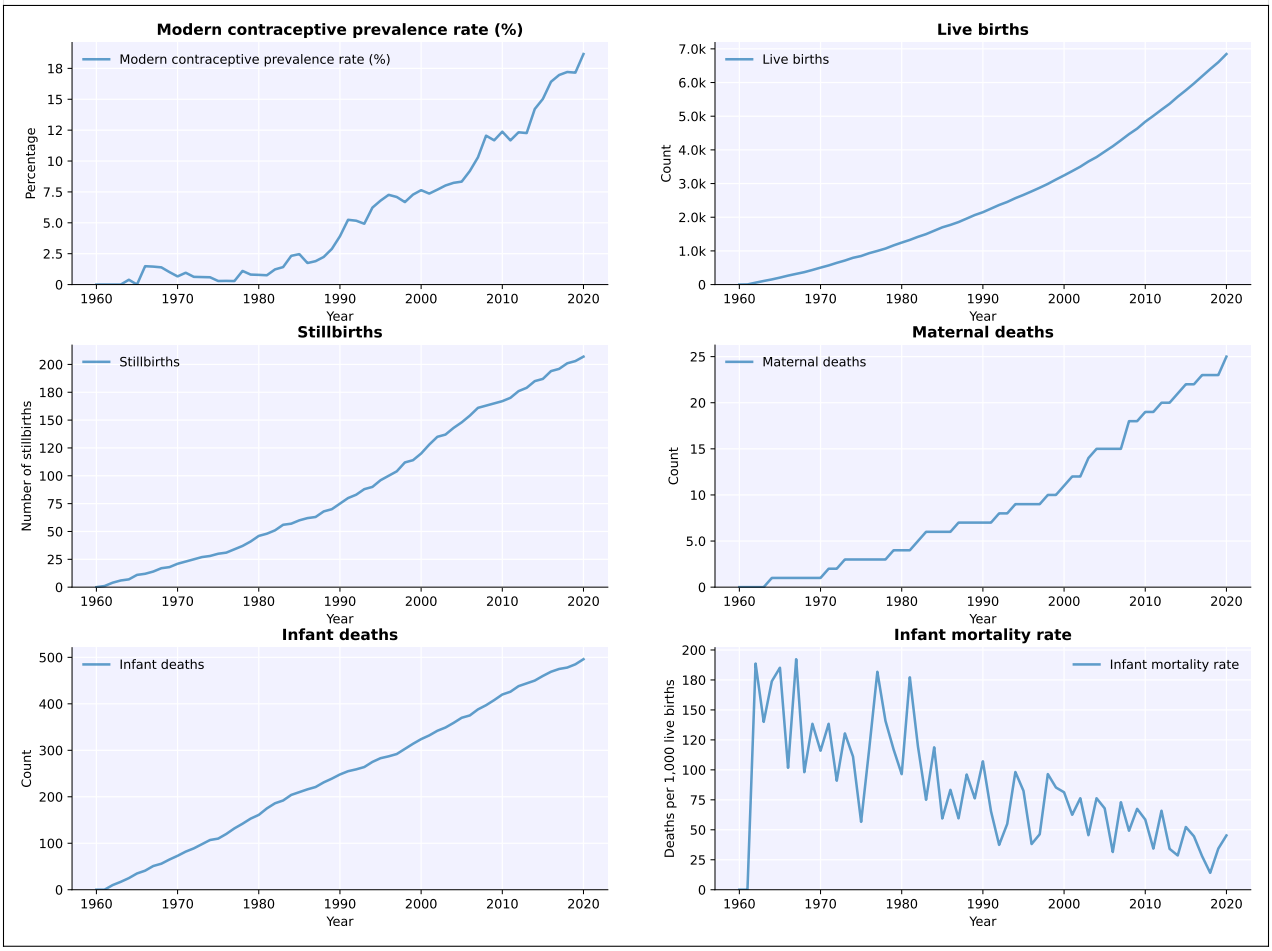
(continued from previous page)

```
  Running 1969 of 2020 (108/721) (1.27 s)  •••---------------- 15%
  Running 1970 of 2020 (120/721) (1.41 s)  •••---------------- 17%
  Running 1971 of 2020 (132/721) (1.54 s)  •••---------------- 18%
  Running 1972 of 2020 (144/721) (1.68 s)  ••••--------------- 20%
  Running 1973 of 2020 (156/721) (1.81 s)  ••••--------------- 22%
  Running 1974 of 2020 (168/721) (1.95 s)  ••••--------------- 23%
  Running 1975 of 2020 (180/721) (2.08 s)  •••••-------------- 25%
  Running 1976 of 2020 (192/721) (2.22 s)  •••••-------------- 27%
  Running 1977 of 2020 (204/721) (2.35 s)  •••••-------------- 28%
  Running 1978 of 2020 (216/721) (2.49 s)  ••••••------------- 30%
  Running 1979 of 2020 (228/721) (2.63 s)  ••••••------------- 32%
  Running 1980 of 2020 (240/721) (2.76 s)  ••••••------------- 33%
  Running 1981 of 2020 (252/721) (2.90 s)  •••••••------------ 35%
  Running 1982 of 2020 (264/721) (3.04 s)  •••••••------------ 37%
  Running 1983 of 2020 (276/721) (3.18 s)  •••••••------------ 38%
  Running 1984 of 2020 (288/721) (3.31 s)  ••••••••----------- 40%
  Running 1985 of 2020 (300/721) (3.46 s)  ••••••••----------- 42%
  Running 1986 of 2020 (312/721) (3.59 s)  ••••••••----------- 43%
  Running 1987 of 2020 (324/721) (3.73 s)  •••••••••---------- 45%
  Running 1988 of 2020 (336/721) (3.88 s)  •••••••••---------- 47%
  Running 1989 of 2020 (348/721) (4.02 s)  •••••••••---------- 48%
  Running 1990 of 2020 (360/721) (4.16 s)  ••••••••••--------- 50%
  Running 1991 of 2020 (372/721) (4.30 s)  ••••••••••--------- 52%
  Running 1992 of 2020 (384/721) (4.45 s)  ••••••••••--------- 53%
  Running 1993 of 2020 (396/721) (4.59 s)  •••••••••••-------- 55%
  Running 1994 of 2020 (408/721) (4.73 s)  •••••••••••-------- 57%
  Running 1995 of 2020 (420/721) (4.88 s)  •••••••••••-------- 58%
  Running 1996 of 2020 (432/721) (5.02 s)  ••••••••••••------- 60%
  Running 1997 of 2020 (444/721) (5.18 s)  ••••••••••••------- 62%
  Running 1998 of 2020 (456/721) (5.33 s)  ••••••••••••------- 63%
  Running 1999 of 2020 (468/721) (5.48 s)  •••••••••••••------ 65%
  Running 2000 of 2020 (480/721) (5.63 s)  •••••••••••••------ 67%
  Running 2001 of 2020 (492/721) (5.78 s)  •••••••••••••------ 68%
  Running 2002 of 2020 (504/721) (5.92 s)  ••••••••••••••----- 70%
  Running 2003 of 2020 (516/721) (6.07 s)  ••••••••••••••----- 72%
  Running 2004 of 2020 (528/721) (6.35 s)  ••••••••••••••----- 73%
  Running 2005 of 2020 (540/721) (6.50 s)  •••••••••••••••---- 75%
  Running 2006 of 2020 (552/721) (6.65 s)  •••••••••••••••---- 77%
  Running 2007 of 2020 (564/721) (6.81 s)  •••••••••••••••---- 78%
  Running 2008 of 2020 (576/721) (6.96 s)  ••••••••••••••••--- 80%
  Running 2009 of 2020 (588/721) (7.12 s)  ••••••••••••••••--- 82%
  Running 2010 of 2020 (600/721) (7.27 s)  ••••••••••••••••--- 83%
  Running 2011 of 2020 (612/721) (7.43 s)  •••••••••••••••••-- 85%
  Running 2012 of 2020 (624/721) (7.59 s)  •••••••••••••••••-- 87%
  Running 2013 of 2020 (636/721) (7.74 s)  •••••••••••••••••-- 88%
  Running 2014 of 2020 (648/721) (7.91 s)  ••••••••••••••••••- 90%
  Running 2015 of 2020 (660/721) (8.07 s)  ••••••••••••••••••- 92%
  Running 2016 of 2020 (672/721) (8.23 s)  ••••••••••••••••••- 93%
  Running 2017 of 2020 (684/721) (8.39 s)  ••••••••••••••••••• 95%
  Running 2018 of 2020 (696/721) (8.55 s)  ••••••••••••••••••• 97%
  Running 2019 of 2020 (708/721) (8.72 s)  ••••••••••••••••••• 98%
  Running 2020 of 2020 (720/721) (8.88 s)  ••••••••••••••••••• 100%
```

(continues on next page)

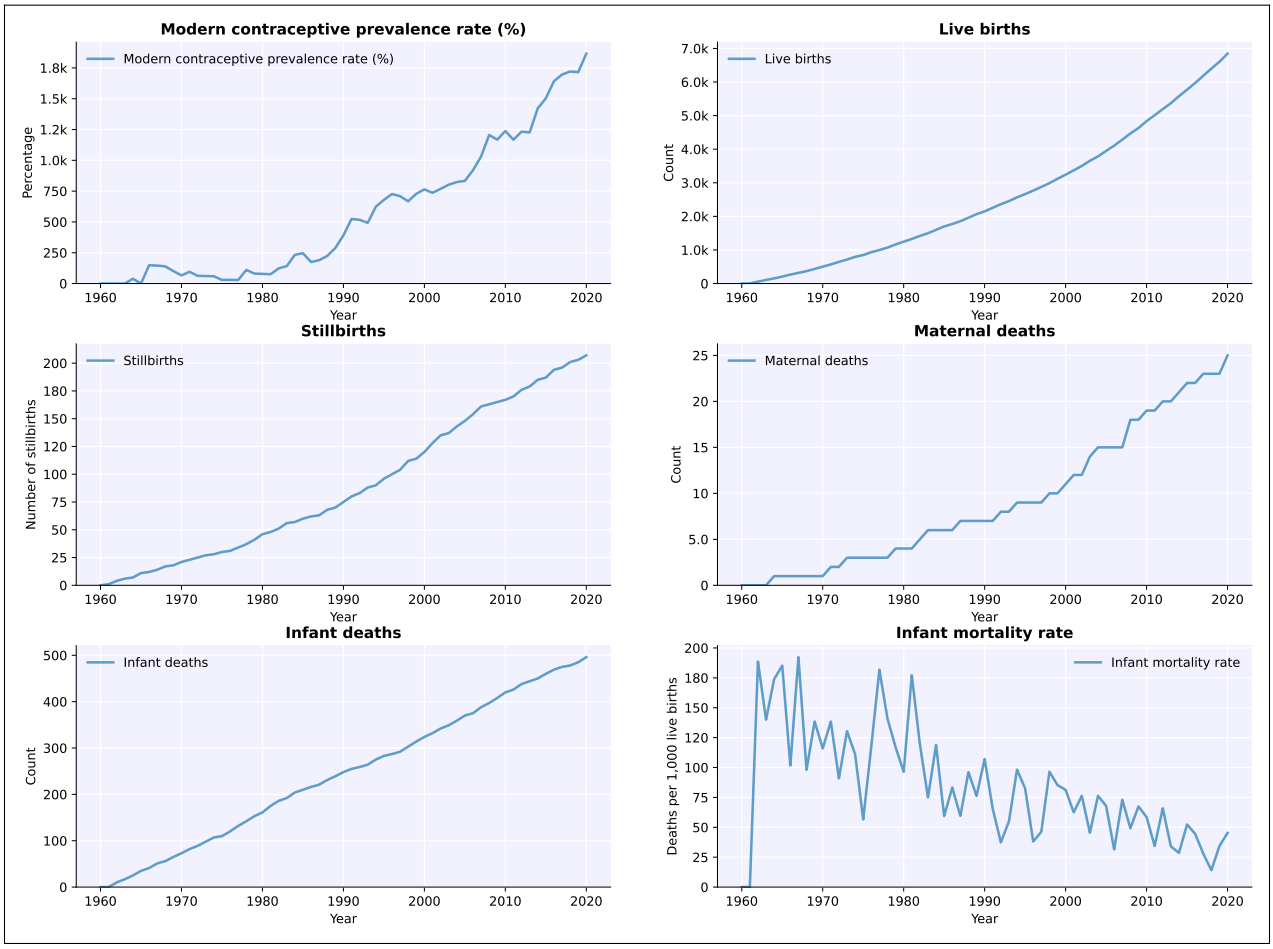
(continued from previous page)

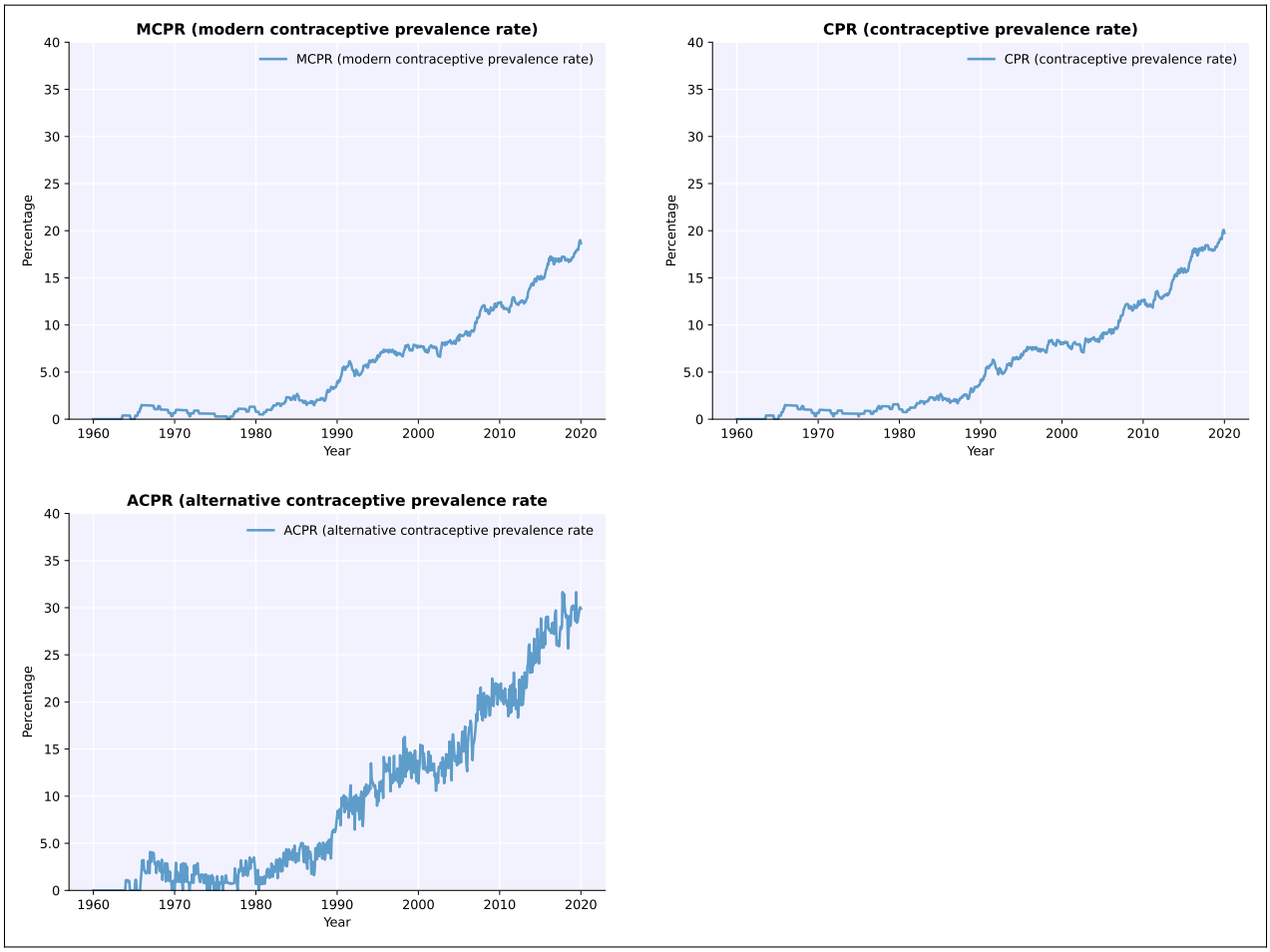
```
Final population size: 6139.
Run finished for "None" after 8.9 s
```

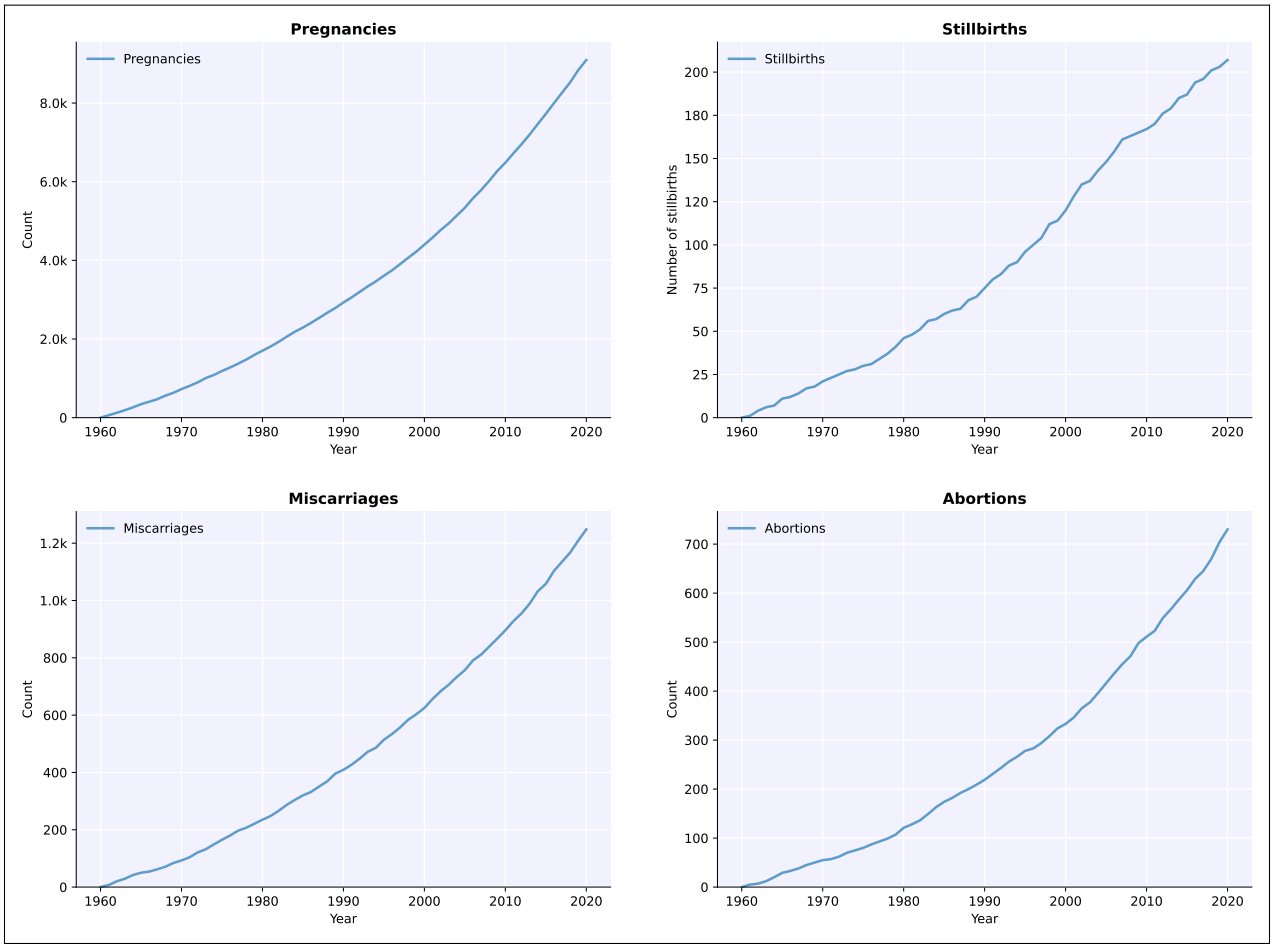


Let's take a look at the basic suite of plotting options, once we've run our initial simulation. The basic plot function will plot births, deaths, and mcpr over the entire simulation. There are also pre-defined options that combine similar types of output. For instance, 'cpr' will provide all definitions we use for contraceptive prevalence rates - mCPR, CPR, and aCPR. 'apo' stands for adverse pregnancy outcomes, and will plot infant deaths, stillbirths, and abortions.

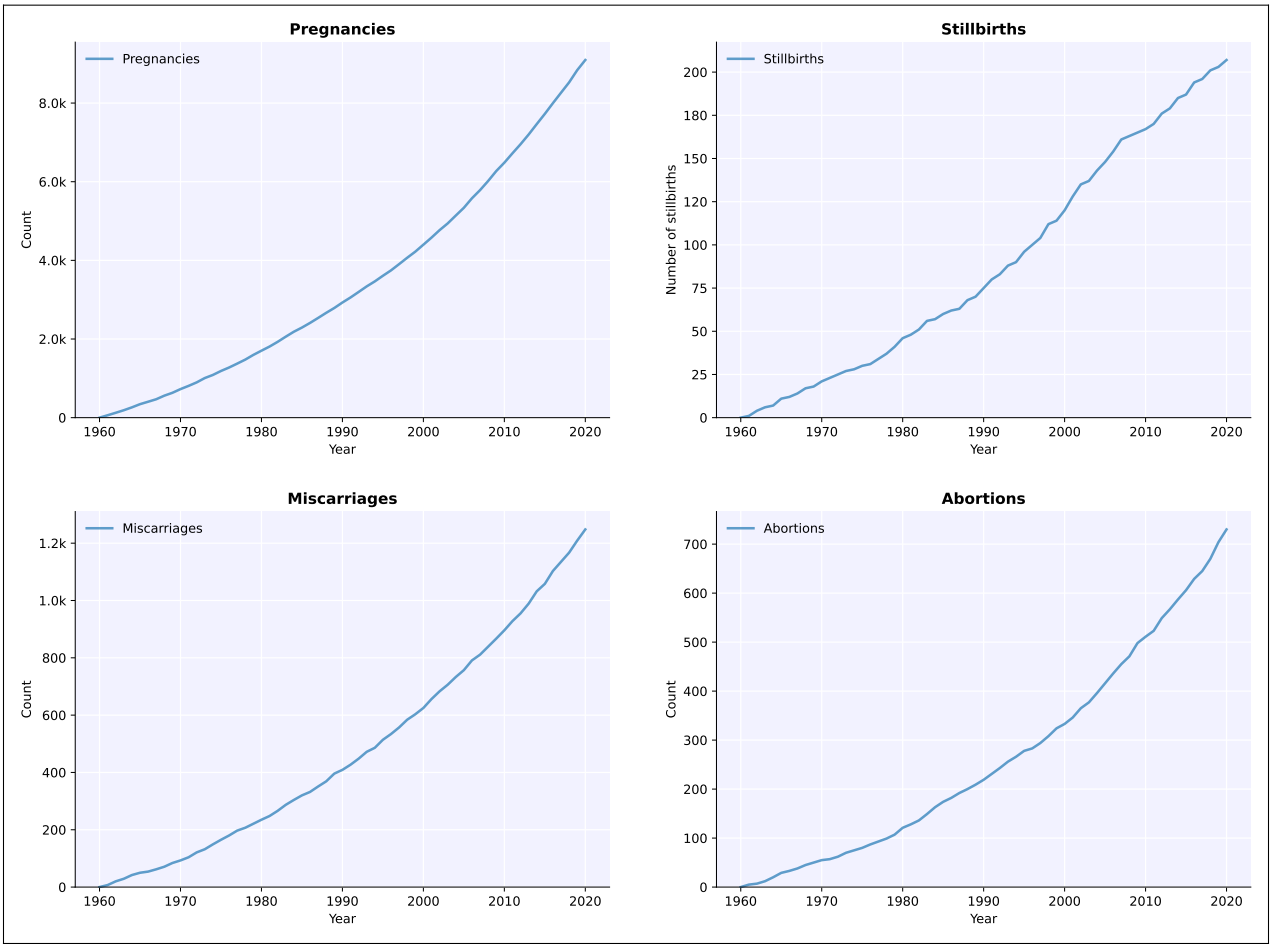
[2]:
```
sim.plot()
sim.plot('cpr')
sim.plot('apo')
```


Modern contraceptive prevalence rate (%)
Modern contraceptive prevalence rate (%)
Percentage
Year
Live births
Live births
Count
Year
Stillbirths
Stillbirths
Number of stillbirths
Year
Maternal deaths
Maternal deaths
Count
Year
Infant deaths
Infant deaths
Count
Year
Infant mortality rate
Infant mortality rate
Deaths per 1,000 live births
Year


MCPR (modern contraceptive prevalence rate)
MCPR (modern contraceptive prevalence rate)
Percentage
Year
CPR (contraceptive prevalence rate)
CPR (contraceptive prevalence rate)
Percentage
Year
ACPR (alternative contraceptive prevalence rate
ACPR (alternative contraceptive prevalence rate
Percentage
Year


Pregnancies
Pregnancies
Count
Year
Stillbirths
Stillbirths
Number of stillbirths
Year
Miscarriages
Miscarriages
Count
Year
Abortions
Abortions
Count
Year

[2]:



We can also plot our key output for specific (pre-defined) age groups, since we know that family planning and building changes over womens' lives. Here, we are plotting age-specific contraceptive prevalence.

[3]:
```
# CK: not sure if we want to enable this
# sim.plot('as_cpr')
```

This will generate a results dictionary `sim.results`. For example, the number of pregnancies in the sim can be found using `sim.results['pregnancies']`.

Rather than creating a parameter dictionary, any valid parameter can also be passed to the sim directly. For example, exactly equivalent to the above is:

1.3 What's new

All notable changes to the codebase are documented in this file. Changes that may result in differences in model output, or are required in order to run an old parameter set with the current version, are flagged with the term "Regression information".

Contents

1.3.1 Version 0.19.2 (2022-10-28)

- Added user guide
- *GitHub info*: PR 4

1.3.2 Version 0.19.1 (2022-10-26)

- Moved to new repository location (http://github.com/fpsim/fpsim)
- Updated documentation in README
- Created new tutorials in tutorials folder
- Ordered tutorials by complexity through T1, T2, T3… Tn numbering system
- *GitHub info*: PR 1

1.3.3 Version 0.19.0 (2022-09-01)

- Added age-specific plotting for tfr, pregnancies, imr, mmr, stillbirths, and births to Sim, MultiSim, and Scenarios
- Added ability to plot channels by age over the course of an interval of time (one year, for example)
- Added yearly age-specific plotting for pregnancies, imr and mmr
- *GitHub info*: PR 590

1.3.4 Version 0.18.2 (2022-08-12)

- Added age specific plotting for cpr, mcpr, and acpr to Sim, MultiSim, and Scenarios
- *GitHub info*: PR 584

1.3.5 Version 0.18.1 (2022-08-08)

- Added y-axis scaling to Sim.plot and MultiSim.plot()
- *GitHub info*: PR 583

1.3.6 Version 0.18.0 (2022-08-01)

- Adjusted stillbirth rates from Nori et al., which was conducted June 2022
- *GitHub info*: PR 560

1.3.7 Version 0.17.5 (2022-07-28)

- Refactored ExperimentVerbose and verbose_sim and related parts of test suite
- *GitHub info*: PR 471

1.3.8 Version 0.17.4 (2022-07-27)

- Added new test suite for the Scenarios API
- *GitHub info*: PR 527

1.3.9 Version 0.17.3 (2022-07-18)

- Added tutorial jupyter notebook to showcase Scenarios features
- *GitHub info*: PR 484

1.3.10 Version 0.17.2 (2022-07-15)

- Switched method mix plotting from line chart to stacked area chart for all classes
- *GitHub info*: PR 568

1.3.11 Version 0.17.1 (2022-07-14)

- Added example_scens.py for a quick debug of adding a novel method when developing new features
- Updated README with new debugging guidance
- GitHub info*: PR 570

1.3.12 Version 0.17.0 (2022-07-08)

- Added method mix timeseries plotting to Sim, MultiSim, and Scenarios through plot(to_plot='method')
- Added some test coverage for method mix plotting
- *GitHub info*: PR 554

1.3.13 Version 0.16.2 (2022-07-01)

- Refactors channel aggregation in Scenarios.analyze_sims()
- *GitHub info*: PR 561

1.3.14 Version 0.16.1 (2022-06-30)

- Add tracking of pregnancies
- Add cumulative sum of pregnancies to plotting functionality (see plot('apo'))
- *GitHub info*: PR 555

1.3.15 Version 0.16.0 (2022-06-28)

- Split matrix age category >25 into 26-35 and >35
- Baseline contraceptive behavior remains the same, but interventions can differentiate now
- *GitHub info*: PR 551

1.3.16 Version 0.15.0 (2022-06-13)

- Added new plotting functionality `Scenarios.plot('mortality')`
- Added new plotting functionality `Scenarios.plot('apo')` for adverse pregnancy outcomes
- Added `stillbirths_over_year` to keys, tracking, and plotting
- Added tracking of miscarriage, abortion, corresponding keys and plotting
- Temporarily commented out plot_interventions in `sim.py` to fix x-axis and vline issues in plotting

- *GitHub info*: PR 549

1.3.17 Version 0.14.2 (2022-06-06)

- Adding 3 new columns to the results dataframe in Scenarios

1.3.18 Version 0.14.2 (2022-05-31)

- Fixed bug in `fp.snapshot()` missing non-exact timesteps.
- Fixed bug with `fp.timeseries_recorder()` not being capable of being added as a kwarg.
- Tidied output of `SimVerbose.story()`.
- Added `sim.get_analyzer()` and `sim.get_intervention()` methods (along with the plural versions).
- Renamed `Experiment.dhs_data` to `Experiment.data`; likewise for `model_to_calib` → `model`.
- Fixed bug with MCPR year plotting in `Experiment`.
- Fixed bug with analyzers being applied only at the end of the sim instead of at every timestep.
- Fixed bug with interventions not plotting with simulations.
- Fixed bug with `finalize()` not being called for interventions.
- Increased code coverage of tests from 67% to 80%.
- *GitHub info*: PR 533

1.3.19 Version 0.14.1 (2022-05-27)

- Fixed bugs in how `copy_from` is implemented in scenarios.
- *GitHub info*: PR 526

1.3.20 Version 0.14.0 (2022-05-26)

- Adds an options module, allowing things like DPI to be set via `fp.options(dpi=150)`.
- Updates plotting options and allows more control over style.
- Adds more control to plots, including `start_year` and `end_year`.
- Adds a `copy_from` keyword to method probability update scenarios.
- Renames `years` to `par_years` in scenarios.
- Changes the logic of the `People` update step so that lactational amenorrhea is calculated after breastfeeding is updated.
- Changes the `Sim` representation to e.g. `Sim("My sim"; n=10,000; 1960-2020; results: b=69,541 =11,920 pop=62,630)`
- *GitHub info*: PR 522

1.3.21 Version 0.13.2 (2022-05-25)

- Added ASFR as an output of Experiments.
- `MultiSim.run()` now automatically labels un-labeled sims; this fixes bugs in MultiSim plotting functions.
- MultiSims also have additional error checking (e.g., they cannot be rerun).
- Refactored data files to be in "tall" instead of "wide" format.
- Removed years and age bins from summary statistics.
- *GitHub info*: PR 517

1.3.22 Version 0.13.1 (2022-05-25)

- Changed `MultiSim.plot_method_mix()` to be able to work with `Scenarios`
- *GitHub info*: PR 513

1.3.23 Version 0.13.0 (2022-05-23)

- Changed parameters from a dictionary to a class and added `parameters.py`. This class has additional validation, the ability to import from/export to JSON, etc.
- Restructured methods, including renaming `pars['method_efficacy']` to `pars['methods']['eff']`, plus a new entry, `pars['methods']['modern']`, to specify which are modern methods used for calculating MCPR.
- Methods have been reordered, grouping traditional and modern methods and sorting modern methods by longevity (e.g. condoms → pill → implants → IUDs).
- Added ability to add/remove contraceptive methods via `pars.add_method()` and `pars.rm_method()`.
- Added a method to run a single scenario.
- *GitHub info*: PR 503

1.3.24 Version 0.12.0 (2022-05-22)

- Split FPsim repository from analyses scripts.
- Refactors `experiment.py` to load files for a specific location rather than being hard-coded.
- *GitHub info*: PR 504

1.3.25 Version 0.11.5 (2022-05-21)

- Improvements to the scenarios, including more helpful docstrings and error messages.
- Improved error checking of sims.
- *GitHub info*: PR 502

1.3.26 Version 0.11.4 (2022-05-20)

- Renamed parameter `n` to `n_agents`, and adds parameter `scaled_pop`.
- Tracking of switch events is disabled by default; set `pars['track_switching'] = True` to re-enable.
- Update default end year from 2019 to 2020.
- *GitHub info*: PR 496

1.3.27 Version 0.11.3 (2022-05-20)

- Tidied `tests` folder.
- Removed the calibration database by default (to keep, use `fp.Calibration(keep_db=True)`.
- *GitHub info*: PR 495

1.3.28 Version 0.11.2 (2022-05-20)

- Added a `people.make_pregnant()` method.
- *GitHub info*: PR 494

1.3.29 Version 0.11.1 (2022-05-20)

- Replaced `high` and `low` breastfeeding duration parameters with Gumbel distribution parameters `mu` and `beta`.
- *GitHub info*: PR 493

1.3.30 Version 0.11.0 (2022-05-20)

- Major refactor of `senegal.py`, organizing parameters into groups and renaming.
- Parameter names made more consistent, e.g. `exposure_correction` → `exposure_factor`, `maternal_mortality_multiplier` → `maternal_mortality_factor`.
- Added comprehensive parameter checking.
- Updates to the default representation: `print(sim)` is now a very brief representation; use `sim.disp()` to get the old behavior.
- *GitHub info*: PR 492

1.3.31 Version 0.10.7 (2022-05-19)

- Updated `fp.Scenarios()` API.
- Added a new `fp.Scenario()` class, with a convenience function `fp.make_scen()` for creating new scenarios, for later use with `fp.Scenarios()`.
- *GitHub info*: PR 488

1.3.32 Version 0.10.6 (2022-05-19)

- Adds `fp.parallel()` to quickly run multiple sims in parallel and return a `MultiSim` object.
- Adds an `fp.change_par()` intervention.
- *GitHub info*: PR 487

1.3.33 Version 0.10.5 (2022-05-18)

- Changes how the matrices are implemented. For example, `sim['methods']['probs']['18-25']` has been renamed `sim['methods']['raw']['annual']['18-25']`; `sim['methods']['probs']['18-25']` has been renamed `sim['methods']['adjusted']['annual']['18-25']`; `sim['methods_postpartum']['probs1to6']['18-25']` has been renamed `sim['methods']['adjusted']['pp1to6']['18-25']`; etc.
- Various other parameters were renamed for consistency (e.g. `years` → `year`).
- Various other methods were renamed for clarity (e.g. `maternal_mortality()` → `check_maternal_mortality()`; `check_mcpr()` → `track_mcpr()`).
- Input validation has been added to the `Scenarios` class.
- Fixed `fp.update_methods()` so it can no longer produce probabilities >1.
- Removed a circular import in `scenarios.py`.
- *GitHub info*: PR 482

1.3.34 Version 0.10.4 (2022-05-17)

- Fixes bugs with the MCPR growth implementation, as well as the wrong matrix being used.
- Added three new parameters: `mcpr_growth_rate`, `mcpr_max`, and `mcpr_norm_year`, to control how MCPR growth is projected into the future.
- Updated `sim.run()` to return `self` rather than `self.results`.
- *GitHub info*: PR 480

1.3.35 Version 0.10.3 (2022-05-12)

- Move country-specific parameters from `fpsim.data` to `fpsim.locations`.
- *GitHub info*: PR 464

1.3.36 Version 0.10.2 (2022-05-10)

- Refactored `People.get_method()` to use more efficient looping.
- Numbafied `n_multinomial()` to get a ~20% speed increase.
- Added a `method_timestep` parameter to allow skipping contraceptive matrix updates (saves significant time for small sims).
- Added `fp.pars(location='test')` to use defaults for testing (e.g. small population size).
- Fixed divide-by-zero bug for small population sizes in total fertility rate.

- Refactored tests; they should now run locally in ~15 s.
- *GitHub info*: PR 448

1.3.37 Version 0.10.1 (2022-05-09)

- Fix `Scenarios` class.
- *GitHub info*: PR 433

1.3.38 Version 0.10.0 (2022-05-08)

- Moved Senegal parameters into FPsim.
- Added age of sexual debut.
- *GitHub info*: PR 427

1.3.39 Version 0.9.0 (2022-05-05)

- Added a new `Scenarios` class.
- *GitHub info*: PR 416

1.3.40 Version 0.8.0 (2021-08-28)

- Refactored the `People` object to use a new filtering-based approach.
- *GitHub info*: PR 219

1.3.41 Version 0.7.3 (2021-07-15)

- Fix bug to ensure that at least one process runs on each worker.
- *GitHub info*: PR 163

1.3.42 Version 0.7.2 (2021-07-14)

- Allow `total_trials` to be passed to an `fp.Calibration` object.
- *GitHub info*: PR 162

1.3.43 Version 0.7.1 (2021-07-14)

- Allow `weights` to be passed to an `fp.Calibration` object.
- *GitHub info*: PR 161

1.3.44 Version 0.7.0 (2021-06-29)

- Added new calibration plotting methods.
- Separated Experiment and Calibration into separate files, and renamed `model.py` to `sim.py`.
- Fixed a bug where the age pyramid was being unintentionally modified in-place.
- *GitHub info*: PR 144

1.3.45 Version 0.6.5 (2021-06-11)

- Added R support; see `examples/example_sim.R`.
- Fixed a bug where the age pyramid was being unintentionally modified in-place.
- *GitHub info*: PR 128

1.3.46 Version 0.6.4 (2021-06-10)

- Added a `MultiSim` class, which can handle parallel runs and uncertainty bounds.
- *GitHub info*: PR 124

1.3.47 Version 0.6.3 (2021-06-10)

- Fixed a bug where exposure correction by age was accidentally being clipped to the range [0,1], restoring behavior of the array-based model to match the object-based model (notwithstanding stochastic effects and other bugfixes).
- *GitHub info*: PR 119

1.3.48 Version 0.6.2 (2021-05-10)

- Added `fp.Intervention` and `fp.Analyzer` classes, which are much more flexible ways to modify and record the state of the simulation, respectively.
- Fixed a bug with only females being born.
- *GitHub info*: PR 100

1.3.49 Version 0.6.1 (2021-05-02)

- Renamed `fp.Calibration` to `fp.Experiment`, and added a new `fp.Calibration` class, using Optuna.
- This allows the user to do e.g. `calib = fp.Calibration(pars); calib.calibrate(calib_pars)`
- Calibrating a single parameter takes about 20 seconds for a single parameter and a small population size (500 people). Realistic calibrations should take roughly 10 - 60 minutes.
- *GitHub info*: PR 93

1.3.50 Version 0.6.0 (2021-05-01)

- Refactored the model to use an array-based implementation, instead of a loop over individual people.
- This results in a performance increase of roughly 20-100x, depending on the size of the simulation. In practice, this means that 50,000 people can be run in roughly the same amount of time as 500 could be previously.
- *GitHub info*: PR 92

1.3.51 Version 0.5.2 (2021-04-30)

- Added a new script, `preprocess_data.py`, that takes large raw data files and preprocesses them down to only the essentials used in the model.
- This increases the performance of `calib.run()` (**not** counting model runtime) by a factor of 1000.
- *GitHub info*: PR 91

1.3.52 Version 0.5.1 (2021-04-29)

- Added `summarize()` and `to_json()` methods to `Calibration`. Also added an `fp.diff_summaries()` method for comparing them.
- Added regression and benchmarking tests (current total time: 24 s).
- Added a code coverage script (current code coverage: 59%).
- Added default flags for which quantities to compute.
- Split the logic of `Calibration` out into more detail: e.g., initialization, running, and post-processing.
- *GitHub info*: PR 90

1.4 API reference

1.4.1 Subpackages

1.4.1.1 fpsim.locations package

Submodules

fpsim.locations.senegal module

Set the parameters for FPsim, specifically for Senegal.

scalar_pars()

data2interp(*data*, *ages*, *normalize=False*)

Convert unevenly spaced data into an even spline interpolation

filenames()

Data files for use with calibration, etc – not needed for running a sim

age_pyramid()

Starting age bin, male population, female population

age_mortality()

Age-dependent mortality rates, Senegal specific from 1990-1995 – see age_dependent_mortality.py in the fp_analyses repository Mortality rate trend from crude mortality rate per 1000 people: https://data.worldbank.org/indicator/SP.DYN.CDRT.IN?locations=SN

maternal_mortality()

Risk of maternal death assessed at each pregnancy. Data from Huchon et al. (2013) prospective study on risk of maternal death in Senegal and Mali. Maternal deaths: The annual number of female deaths from any cause related to or aggravated by pregnancy or its management (excluding accidental or incidental causes) during pregnancy and childbirth or within 42 days of termination of pregnancy, irrespective of the duration and site of the pregnancy, expressed per 100,000 live births, for a specified time period.

infant_mortality()

From World Bank indicators for infant morality (< 1 year) for Senegal, per 1000 live births From API_SP.DYN.IMRT.IN_DS2_en_excel_v2_1495452.numbers Adolescent increased risk of infant mortality gradient taken from Noori et al for Sub-Saharan African from 2014-2018. Odds ratios with age 23-25 as reference group: https://www.medrxiv.org/content/10.1101/2021.06.10.21258227v1

miscarriage()

Returns a linear interpolation of the likelihood of a miscarriage by age, taken from data from Magnus et al BMJ 2019: https://pubmed.ncbi.nlm.nih.gov/30894356/ Data to be fed into likelihood of continuing a pregnancy once initialized in model Age 0 and 5 set at 100% likelihood. Age 10 imputed to be symmetrical with probability at age 45 for a parabolic curve

stillbirth()

From Report of the UN Inter-agency Group for Child Mortality Estimation, 2020 https://childmortality.org/wp-content/uploads/2020/10/UN-IGME-2020-Stillbirth-Report.pdf

Age adjustments come from an extension of Noori et al., which were conducted June 2022.

female_age_fecundity()

Use fecundity rates from PRESTO study: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5712257/ Fecundity rate assumed to be approximately linear from onset of fecundity around age 10 (average age of menses 12.5) to first data point at age 20 45-50 age bin estimated at 0.10 of fecundity of 25-27 yr olds, based on fertility rates from Senegal

fecundity_ratio_nullip()

Returns an array of fecundity ratios for a nulliparous woman vs a gravid woman from PRESTO study: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5712257/ Approximates primary infertility and its increasing likelihood if a woman has never conceived by age

lactational_amenorrhea()

Returns an array of the percent of breastfeeding women by month postpartum 0-11 months who meet criteria for LAM: Exclusively breastfeeding (bf + water alone), menses have not returned. Extended out 5-11 months to better match data as those women continue to be postpartum insusceptible. From DHS Senegal calendar data

sexual_activity()

Returns a linear interpolation of rates of female sexual activity, defined as percentage women who have had sex within the last four weeks. From STAT Compiler DHS https://www.statcompiler.com/en/ Using indicator "Timing of sexual intercourse" Includes women who have had sex "within the last four weeks" Excludes women who answer "never had sex", probabilities are only applied to agents who have sexually debuted Data taken from 2018 DHS, no trend over years for now Onset of sexual activity probabilities assumed to be linear from age 10 to first data point at age 15

sexual_activity_pp()

Returns an array of monthly likelihood of having resumed sexual activity within 0-35 months postpartum Uses DHS Senegal 2018 individual recode (postpartum (v222), months since last birth, and sexual activity within 30 days. Limited to 35 months postpartum (can use any limit you want 0-35 max) Postpartum month 0 refers to the first month after delivery

debut_age()

Returns an array of weighted probabilities of sexual debut by a certain age 10-45. Data taken from DHS variable v531 (imputed age of sexual debut, imputed with data from age at first union) Use sexual_debut_age_probs.py under fp_analyses/data to output for other DHS countries

exposure_age()

Returns an array of experimental factors to be applied to account for residual exposure to either pregnancy or live birth by age. Exposure to pregnancy will increase factor number and residual likelihood of avoiding live birth (mostly abortion, also miscarriage), will decrease factor number

exposure_parity()

Returns an array of experimental factors to be applied to account for residual exposure to either pregnancy or live birth by parity.

birth_spacing_pref()

Returns an array of birth spacing preferences by closest postpartum month. Applied to postpartum pregnancy likelihoods.

NOTE: spacing bins must be uniform!

methods()

Names, indices, modern/traditional flag, and efficacies of contraceptive methods – see also parameters.py Efficacy from Guttmacher, fp_prerelease/docs/gates_review/contraceptive-failure-rates-in-developing-world_1.pdf BTL failure rate from general published data Pooled efficacy rates for all women in this study: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4970461/

method_probs()

Define "raw" (un-normalized, un-trended) matrices to give transitional probabilities from 2018 DHS Senegal contraceptive calendar data.

Probabilities in this function are annual probabilities of initiating (top row), discontinuing (first column), continuing (diagonal), or switching methods (all other entries).

Probabilities at postpartum month 1 are 1 month transitional probabilities for starting a method after delivery.

Probabilities at postpartum month 6 are 5 month transitional probabilities for starting or changing methods over the first 6 months postpartum.

Data from Senegal DHS contraceptive calendars, 2017 and 2018 combined

barriers()

Reasons for nonuse – taken from DHS

make_pars()

Take all parameters and construct into a dictionary

1.4.2 Submodules

1.4.2.1 fpsim.analyzers module

Specify the core analyzers available in FPsim. Other analyzers can be defined by the user by inheriting from these classes.

class Analyzer(*label=None*)

Bases: prettyobj

Base class for analyzers. Based on the Intervention class. Analyzers are used to provide more detailed information about a simulation than is available by default – for example, pulling states out of sim.people on a particular timestep before it gets updated in the next timestep.

To retrieve a particular analyzer from a sim, use sim.get_analyzer().

Parameters
label (*str*) – a label for the Analyzer (used for ease of identification)

initialize(*sim=None*)

Initialize the analyzer, e.g. convert date strings to integers.

finalize(*sim=None*)

Finalize analyzer

This method is run once as part of *sim.finalize()* enabling the analyzer to perform any final operations after the simulation is complete (e.g. rescaling)

apply(*sim*)

Apply analyzer at each time point. The analyzer has full access to the sim object, and typically stores data/results in itself. This is the core method which each analyzer object needs to implement.

Parameters
sim – the Sim instance

to_json()

Return JSON-compatible representation

Custom classes can't be directly represented in JSON. This method is a one-way export to produce a JSON-compatible representation of the intervention. This method will attempt to JSONify each attribute of the intervention, skipping any that fail.

Returns
JSON-serializable representation

class snapshot(*timesteps*, **args*, *die=True*, ***kwargs*)

Bases: *Analyzer*

Analyzer that takes a "snapshot" of the sim.people array at specified points in time, and saves them to itself.

Parameters

- **timesteps** (*list*) – list of timesteps on which to take the snapshot
- **args** (*list*) – additional timestep(s)
- **die** (*bool*) – whether or not to raise an exception if a date is not found (default true)
- **kwargs** (*dict*) – passed to Analyzer()

Example:

```
sim = fp.Sim(analyzers=fps.snapshot('2020-04-04', '2020-04-14'))
sim.run()
snapshot = sim.pars['analyzers'][0]
people = snapshot.snapshots[0]
```

apply(*sim*)

Apply snapshot at each timestep listed in timesteps and save result at snapshot[str(timestep)]

class timeseries_recorder

Bases: *Analyzer*

Record every attribute in people as a timeseries.

self.**i**

The list of timesteps (ie, 0 to 261 steps).

self.**t**

The time elapsed in years given how many timesteps have passed (ie, 25.75 years).

self.**y**

The calendar year of timestep (ie, 1975.75).

self.**keys**

A list of people states excluding 'dobs'.

self.**data**

A dictionary where self.data[state][timestep] is the mean of the state at that timestep.

Initializes self.i/t/y as empty lists and self.data as empty dictionary

initialize(*sim*)

Initializes self.keys from sim.people

apply(*sim*)

Applies recorder at each timestep

plot(*x='y'*, *fig_args=None*, *pl_args=None*)

Plots time series of each state as a line graph

class age_pyramids(*bins=None*)

Bases: *Analyzer*

Records age pyramids for each timestep.

self.**bins**

A list of ages, default is a sequence from 0 to max_age + 1.

self.**data**

A matrix of shape (number of timesteps, number of bins - 1) containing age pyramid data.

Initializes bins and data variables

initialize(*sim*)

Initializes bins and data with proper shapes

apply(*sim*)

Records histogram of ages of all alive individuals at a timestep such that self.data[timestep] = list of proportions where index signifies age

`plot`()

Plots self.data as 2D pyramid plot

`plot3d`()

Plots self.data as 3D pyramid plot

`class verbose_sim`(*to_csv=False*, *custom_csv_tables=None*, *to_file=False*)

Bases: *Analyzer*

Initializes a verbose_sim analyzer which extends the logging functionality of the sim with calculated channels, total state results of a sim run, the story() feature, and configurable file formatting for results

`apply`(*sim*)

Logs data for total_results and events at each timestep.

Output:

self.total_results::dict
Dictionary of all individual results formatted as {timestep: attribute: [values]} keys correspond to fpsim.defaults debug_states

self.events::dict
Dictionary of events correponding to self.channels formatted as {timestep: channel: [indices]}.

`save`(*to_csv=True*, *to_json=False*, *custom_csv_tables=None*)

At the end of sim run, stores total_results as either a json or feather file.

Inputs

self.to_csv::bool
If True, writes results to csv files in /sim_output where each state's history is a separate file

self.to_json::bool
If True, writes results to json file

custom_csv_tables::list
List of states that the user wants to write to csv, default is all

Outputs:
Either a json file at "sim_output/total_results.json" or a csv file for each state at "sim_output/{state}_state.csv"

`story`(*index*, *output=False*, *debug=False*)

Prints a story of all major events in an individual's life based on calculated verbose_sim channels, base Sim channels, and statistics calculated within the function such as year of birth of individual.

Parameters

- **index** (*int*) – index of the individual, must be less than population
- **output** (*bool*) – return as output string rather than print
- **debug** (*bool*) – print additional information

Outputs:
printed display of each major event in the individual's life

1.4.2.2 fpsim.base module

Base classes for loading parameters and for running simulations with FP model

class ParsObj(*pars*, ***kwargs*)

Bases: FlexPretty

A class based around performing operations on a self.pars dict.

update_pars(*pars=None*, *create=False*, ***kwargs*)

Update internal dict with new pars.

Parameters

- **pars** (*dict*) – the parameters to update (if None, do nothing)
- **create** (*bool*) – if create is False, then raise a KeyNotFoundError if the key does not already exist
- **kwargs** (*dict*) – additional parameters

class BasePeople

Bases: prettyobj

Class for all the people in the simulation.

Initialize essential attributes used for filtering

keys()

Returns keys for all properties of the people object

property is_female

Boolean array of everyone female

property is_male

Boolean array of everyone male

property int_age

Return ages as an integer

property ceil_age

Rounds age up to the next highest integer

property int_age_clip

Return ages as integers, clipped to maximum allowable age for pregnancy

property n

Number of people alive

property inds

Alias to self._inds to prevent accidental overwrite & increase speed

property len_inds

Alias to len(self)

property len_people

Full length of People array, ignoring filtering

plot(*fig_args=None*, *hist_args=None*)

Plot histograms of each quantity

filter(*criteria=None*, *inds=None*)

Store indices to allow for easy filtering of the People object.

Parameters

- **criteria** (*array*) – a boolean array for the filtering critria
- **inds** (*array*) – alternatively, explicitly filter by these indices

Returns

A filtered People object, which works just like a normal People object except only operates on a subset of indices.

unfilter()

An easy way of unfiltering the People object, returning the original.

binomial(*prob*, *as_inds=False*, *as_filter=False*)

Return indices either by a single probability or by an array of probabilities. By default just return the boolean array, but can also return the indices, or the filtered People object.

Parameters

- **prob** (*float/array*) – either a scalar probability, or an array of probabilities of the same length as People
- **as_inds** (*bool*) – return as list of indices instead of a boolean array
- **as_filter** (*bool*) – return as filter instead than boolean array

class BaseSim(*pars*, ***kwargs*)

Bases: *ParsObj*

The BaseSim class handles the dynamics of the simulation.

year2ind(*year*)

ind2year(*ind*)

ind2calendar(*ind*)

property npts

Count the number of points in timesteps between the starting year and the ending year.

property tvec

Create a time vector array at intervals of the timestep in years

property n

get_interventions(*label=None*, *partial=False*, *as_inds=False*)

Find the matching intervention(s) by label, index, or type. If None, return all interventions. If the label provided is "summary", then print a summary of the interventions (index, label, type).

Parameters

- **label** (*str, int, Intervention, list*) – the label, index, or type of intervention to get; if a list, iterate over one of those types
- **partial** (*bool*) – if true, return partial matches (e.g. 'beta' will match all beta interventions)
- **as_inds** (*bool*) – if true, return matching indices instead of the actual interventions

get_intervention(*label=None*, *partial=False*, *first=False*, *die=True*)

Like get_interventions(), find the matching intervention(s) by label, index, or type. If more than one intervention matches, return the last by default. If no label is provided, return the last intervention in the list.

Parameters

- **label** (*str, int, Intervention, list*) – the label, index, or type of intervention to get; if a list, iterate over one of those types
- **partial** (*bool*) – if true, return partial matches (e.g. 'beta' will match all beta interventions)
- **first** (*bool*) – if true, return first matching intervention (otherwise, return last)
- **die** (*bool*) – whether to raise an exception if no intervention is found

get_analyzers(*label=None*, *partial=False*, *as_inds=False*)

Same as get_interventions(), but for analyzers.

get_analyzer(*label=None*, *partial=False*, *first=False*, *die=True*)

Same as get_intervention(), but for analyzers.

1.4.2.3 fpsim.calibration module

Define the Calibration class

class Calibration(*pars*, *calib_pars=None*, *weights=None*, *verbose=True*, *keep_db=False*, ***kwargs*)

Bases: prettyobj

A class to handle calibration of FPsim objects. Uses the Optuna hyperparameter optimization library (optuna.org).

Note: running a calibration does not guarantee a good fit! You must ensure that you run for a sufficient number of iterations, have enough free parameters, and that the parameters have wide enough bounds. Please see the tutorial on calibration for more information.

Parameters

- **sim** (Sim) – the simulation to calibrate
- **calib_pars** (*dict*) – a dictionary of the parameters to calibrate of the format dict(key1=[best, low, high])
- **weights** (*dict*) – a custom dictionary of weights for each output
- **n_trials** (*int*) – the number of trials per worker
- **n_workers** (*int*) – the number of parallel workers (default: maximum)
- **total_trials** (*int*) – if n_trials is not supplied, calculate by dividing this number by n_workers
- **name** (*str*) – the name of the database (default: 'fpsim_calibration')
- **db_name** (*str*) – the name of the database file (default: 'fpsim_calibration.db')
- **keep_db** (*bool*) – whether to keep the database after calibration (default: false)
- **storage** (*str*) – the location of the database (default: sqlite)
- **label** (*str*) – a label for this calibration object
- **verbose** (*bool*) – whether to print details of the calibration

- **kwargs** (*dict*) – passed to cv.Calibration()

Returns
A Calibration object

set_optuna_defaults()
Create a (mutable) dictionary with default global settings

configure_optuna(***kwargs*)
Update Optuna configuration, if required

validate_pars()
Ensure parameters are in the correct format. Two formats are permitted: either a dict of arrays or lists in order best-low-high, e.g.:

```
calib_pars = dict(
    exposure_factor           = [1.0, 0.5,  1.5],
    maternal_mortality_factor = [1,   0.75, 3.0],
)
```

Or the same thing, as a dict of dicts:

```
calib_pars = dict(
    exposure_factor           = dict(best=1.0, low=0.5,  high=1.5),
    maternal_mortality_factor = dict(best=1,   low=0.75, high=3.0),
)
```

run_exp(*pars*, *return_exp=False*, ***kwargs*)
Create and run an experiment

run_trial(*trial*)
Define the objective for Optuna

worker()
Run a single worker

run_workers()
Run multiple workers in parallel

remove_db()
Remove the database file if keep_db is false and the path exists.

make_study()
Make a study, deleting one if it already exists

calibrate(*calib_pars=None*, *weights=None*, *verbose=None*, ***kwargs*)
Actually perform calibration

summarize()

parse_study()
Parse the study into a data frame

to_json(*filename=None*)
Convert the data to JSON

plot_trend(*best_thresh=2*)
Plot the trend in best mismatch over time

plot_all()

Plot every point: warning, very slow!

plot_best(*best_thresh=2*)

Plot only the points with lowest mismatch

plot_stride(*npts=200*)

Plot a fixed number of points in order across the results

1.4.2.4 fpsim.defaults module

Define defaults for use throughout FPsim

1.4.2.5 fpsim.experiment module

Define classes and functions for the Experiment class (running sims and comparing them to data)

class Experiment(*pars=None*, *flags=None*, *label=None*, ***kwargs*)

Bases: prettyobj

Class for running calibration to data. Effectively, it runs a single sim and compares it to data.

Parameters

- **pars** (*dict*) – dictionary of parameters
- **flags** (*dict*) – which analyses to run; see fp.experiment.default_flags for options
- **label** (*str*) – label of experiment
- **kwargs** (*dict*) – passed into pars

load_data(*key*, ***kwargs*)

Load data from various formats

extract_data()

Load data

pop_growth_rate(*years*, *population*)

run_model(*pars=None*, ***kwargs*)

Create the sim and run the model

post_process_sim()

extract_model()

model_pop_size()

model_mcpr()

model_mmr()

Calculate maternal mortality in model over most recent 3 years

model_infant_mortality_rate()

model_crude_death_rate()

model_crude_birth_rate()

model_data_tfr()

model_data_asfr(*ind=-1*)

extract_skyscrapers()

extract_birth_spacing()

extract_methods()

extract_age_pregnancy()

compute_fit(**args*, ***kwargs*)

Compute how good the fit is

post_process_results(*keep_people=False*, *compute_fit=True*, ***kwargs*)

Compare the model and the data

run(*pars=None*, *keep_people=False*, *compute_fit=True*, ***kwargs*)

Run the model and post-process the results

compare()

Create and print a comparison between model and data

summarize(*as_df=False*)

Convert results to a one-number-per-key summary format. Returns summary, also saves to self.summary.

Parameters

as_df (*bool*) – if True, return a dataframe instead of a dict.

to_json(*filename=None*, *tostring=False*, *indent=2*, *verbose=False*, ***kwargs*)

Export results as JSON.

Parameters

- **filename** (*str*) – if None, return string; else, write to file
- **tostring** (*bool*) – if not writing to file, whether to write to string (alternative is sanitized dictionary)
- **indent** (*int*) – if writing to file, how many indents to use per nested level
- **verbose** (*bool*) – detail to print
- **kwargs** (*dict*) – passed to savejson()

Returns

A unicode string containing a JSON representation of the results, or writes the JSON file to disk

Examples:

```
json = exp.to_json()
exp.to_json('results.json')
```

plot(*do_show=None*, *do_save=None*, *filename='fp_experiment.png'*, *axis_args=None*, *do_maximize=True*)

Plot the model against the data

class Fit(*data*, *sim*, *weights=None*, *keys=None*, *custom=None*, *compute=True*, *verbose=False*, ***kwargs*)

Bases: prettyobj

A class for calculating the fit between the model and the data. Note the following terminology is used here:

- fit: nonspecific term for how well the model matches the data
- difference: the absolute numerical differences between the model and the data (one time series per result)
- goodness-of-fit: the result of passing the difference through a statistical function, such as mean squared error
- loss: the goodness-of-fit for each result multiplied by user-specified weights (one time series per result)
- mismatches: the sum of all the losses (a single scalar value per time series)
- mismatch: the sum of the mismatches – this is the value to be minimized during calibration

Parameters

- **sim** (Sim) – the sim object
- **weights** (*dict*) – the relative weight to place on each result (by default: 10 for deaths, 5 for diagnoses, 1 for everything else)
- **keys** (*list*) – the keys to use in the calculation
- **custom** (*dict*) – a custom dictionary of additional data to fit; format is e.g. {'my_output':{'data':[1,2,3], 'sim':[1,2,4], 'weights':2.0}}
- **compute** (*bool*) – whether to compute the mismatch immediately
- **verbose** (*bool*) – detail to print
- **kwargs** (*dict*) – passed to cv.compute_gof() – see this function for more detail on goodness-of-fit calculation options

Example:

```
sim = cv.Sim()
sim.run()
fit = sim.compute_fit()
fit.plot()
```

compute()

Perform all required computations

reconcile_inputs(*verbose=False*)

Find matching keys and indices between the model and the data

compute_diffs(*absolute=False*)

Find the differences between the sim and the data

compute_gofs(***kwargs*)

Compute the goodness-of-fit

compute_losses()

Compute the weighted goodness-of-fit

compute_mismatch(*use_median=False*)

Compute the final mismatch

plot(*keys=None*, *width=0.8*, *font_size=18*, *fig_args=None*, *axis_args=None*, *plot_args=None*, *do_show=True*)

Plot the fit of the model to the data. For each result, plot the data and the model; the difference; and the loss (weighted difference). Also plots the loss as a function of time.

Parameters

- **keys** (*list*) – which keys to plot (default, all)
- **width** (*float*) – bar width
- **font_size** (*float*) – size of font
- **fig_args** (*dict*) – passed to pl.figure()
- **axis_args** (*dict*) – passed to pl.subplots_adjust()
- **plot_args** (*dict*) – passed to pl.plot()
- **do_show** (*bool*) – whether to show the plot

compute_gof(*actual*, *predicted*, *normalize=True*, *use_frac=False*, *use_squared=False*, *as_scalar='none'*, *eps=1e-09*, *skestimator=None*, ***kwargs*)

Calculate the goodness of fit. By default use normalized absolute error, but highly customizable. For example, mean squared error is equivalent to setting normalize=False, use_squared=True, as_scalar='mean'.

Parameters

- **actual** (*arr*) – array of actual (data) points
- **predicted** (*arr*) – corresponding array of predicted (model) points
- **normalize** (*bool*) – whether to divide the values by the largest value in either series
- **use_frac** (*bool*) – convert to fractional mismatches rather than absolute
- **use_squared** (*bool*) – square the mismatches
- **as_scalar** (*str*) – return as a scalar instead of a time series: choices are sum, mean, median
- **eps** (*float*) – to avoid divide-by-zero
- **skestimator** (*str*) – if provided, use this scikit-learn estimator instead
- **kwargs** (*dict*) – passed to the scikit-learn estimator

Returns
array of goodness-of-fit values, or a single value if as_scalar is True

Return type
gofs (arr)

Examples:

```
x1 = np.cumsum(np.random.random(100))
x2 = np.cumsum(np.random.random(100))

e1 = compute_gof(x1, x2) # Default, normalized absolute error
e2 = compute_gof(x1, x2, normalize=False, use_frac=False) # Fractional error
e3 = compute_gof(x1, x2, normalize=False, use_squared=True, as_scalar='mean') #␣
↪Mean squared error
e4 = compute_gof(x1, x2, skestimator='mean_squared_error') # Scikit-learn's MSE␣
↪method
```

(continues on next page)

(continued from previous page)

```
e5 = compute_gof(x1, x2, as_scalar='median') # Normalized median absolute error --␣
↪highly robust
```

diff_summaries(*sim1*, *sim2*, *skip_key_diffs=False*, *output=False*, *die=False*)

Compute the difference of the summaries of two FPsim calibration objects, and print any values which differ.

Parameters

- **sim1** (*sim/dict*) – the calib.summary dictionary, representing a single sim
- **sim2** (*sim/dict*) – ditto
- **skip_key_diffs** (*bool*) – whether to skip keys that don't match between sims
- **output** (*bool*) – whether to return the output as a string (otherwise print)
- **die** (*bool*) – whether to raise an exception if the sims don't match
- **require_run** (*bool*) – require that the simulations have been run

Example:

```
c1 = fp.Calibration()
c2 = fp.Calibration()
c1.run()
c2.run()
fp.diff_summaries(c1.summarize(), c2.summarize())
```

1.4.2.6 fpsim.interventions module

Specify the core interventions available in FPsim. Other interventions can be defined by the user by inheriting from these classes.

class Intervention(*label=None*, *show_label=False*, *do_plot=None*, *line_args=None*)

Bases: `object`

Base class for interventions. By default, interventions are printed using a dict format, which they can be recreated from. To display all the attributes of the intervention, use disp() instead.

To retrieve a particular intervention from a sim, use sim.get_intervention().

Parameters

- **label** (*str*) – a label for the intervention (used for plotting, and for ease of identification)
- **show_label** (*bool*) – whether or not to include the label in the legend
- **do_plot** (*bool*) – whether or not to plot the intervention
- **line_args** (*dict*) – arguments passed to pl.axvline() when plotting

disp()

Print a detailed representation of the intervention

initialize(*sim=None*)

Initialize intervention – this is used to make modifications to the intervention that can't be done until after the sim is created.

finalize(*sim=None*)

Finalize intervention

This method is run once as part of *sim.finalize()* enabling the intervention to perform any final operations after the simulation is complete (e.g. rescaling)

apply(*sim*)

Apply the intervention. This is the core method which each derived intervention class must implement. This method gets called at each timestep and can make arbitrary changes to the Sim object, as well as storing or modifying the state of the intervention.

Parameters
sim – the Sim instance

Returns
None

plot_intervention(*sim*, *ax=None*, ***kwargs*)

Plot the intervention

This can be used to do things like add vertical lines on days when interventions take place. Can be disabled by setting self.do_plot=False.

Note 1: you can modify the plotting style via the `line_args` argument when creating the intervention.

Note 2: By default, the intervention is plotted at the days stored in self.days. However, if there is a self.plot_days attribute, this will be used instead.

Parameters

- **sim** – the Sim instance
- **ax** – the axis instance
- **kwargs** – passed to ax.axvline()

Returns
None

to_json()

Return JSON-compatible representation

Custom classes can't be directly represented in JSON. This method is a one-way export to produce a JSON-compatible representation of the intervention. In the first instance, the object dict will be returned. However, if an intervention itself contains non-standard variables as attributes, then its *to_json* method will need to handle those.

Note that simply printing an intervention will usually return a representation that can be used to recreate it.

Returns
JSON-serializable representation (typically a dict, but could be anything else)

class change_par(*par*, *years=None*, *vals=None*, *verbose=False*)

Bases: *Intervention*

Change a parameter at a specified point in time.

Parameters

- **par** (*str*) – the parameter to change
- **years** (*float/arr*) – the year(s) at which to apply the change

- **vals** (*any*) – a value or list of values to change to (if a list, must have the same length as years); or a dict of year:value entries

If any value is `'reset'`, reset to the original value.

Example:

```
ec0 = fp.change_par(par='exposure_factor', years=[2000, 2010], vals=[0.0, 2.0]) #␣
↪Reduce exposure factor
ec0 = fp.change_par(par='exposure_factor', vals={2000:0.0, 2010:2.0}) # Equivalent␣
↪way of writing
sim = fp.Sim(interventions=ec0).run()
```

initialize(*sim*)

apply(*sim*)

finalize(*sim=None*)

class update_methods(*year*, *eff=None*, *probs=None*, *matrix=None*, *verbose=False*)

Bases: *Intervention*

Intervention to modify method efficacy and/or switching matrix.

Parameters

- **year** (*float*) – The year we want to change the method.
- **eff** (*dict*) – An optional key for changing efficacy; its value is a dictionary with the following schema:

  {method: efficacy}
  Where method is the method to be changed, and efficacy is the new efficacy (can include multiple keys).

- **probs** (*list*) – A list of dictionaries where each dictionary has the following keys:

  source (str): the source method to be changed. dest (str): the destination method to be changed. factor (float): the factor by which to multiply existing probability; OR value (float): the value to replace the switching probability value. keys (list): a list of strings representing age groups to affect. matrix (str): one of [‘probs’, ‘probs1’, ‘probs1to6’] where:

  probs: Changes the specified uptake at the corresponding year regardless of state. probs1: Changes the specified uptake for all individuals in their first month postpartum. probs1to6: Changes the specified uptake for all individuals that are in the first 6 months postpartum.

apply(*sim*)

Applies the efficacy or contraceptive uptake changes if it is the specified year based on scenario specifications.

1.4.2.7 fpsim.parameters module

Handle sim parameters

class Pars(*pars=None*, **args*, ***kwargs*)

Bases: dict

Class to hold a dictionary of parameters, and associated methods.

Usually not called by the user directly – use `fp.pars()` instead.

Parameters
pars (*dict*) – dictionary of parameters

copy()

Shortcut for deep copying

to_dict()

Return parameters as a new dictionary

to_json(*filename*, ***kwargs*)

Export parameters to a JSON file.

Parameters

- **filename** (*str*) – filename to save to
- **kwargs** (*dict*) – passed to `sc.savejson`

Example::
sim.pars.to_json('my_pars.json')

from_json(*filename*, ***kwargs*)

Import parameters from a JSON file.

Parameters

- **filename** (*str*) – filename to load from
- **kwargs** (*dict*) – passed to `sc.loadjson`

Example::
sim.pars.from_json('my_pars.json')

validate(*die=True*, *update=True*)

Perform validation on the parameters

Parameters

- **die** (*bool*) – whether to raise an exception if an error is encountered
- **update** (*bool*) – whether to update the method and age maps

update_method_eff(*method*, *eff=None*, *verbose=False*)

Update efficacy of one or more contraceptive methods.

Parameters

- **method** (*str/dict*) – method to update, or dict of method:value pairs
- **eff** (*float*) – new value of contraceptive efficacy (not required if method is a dict)

Examples::
pars.update_method_eff('Injectables', 0.99) pars.update_method_eff({'Injectables':0.99, 'Condoms':0.50})

update_method_prob(*source=None*, *dest=None*, *factor=None*, *value=None*, *ages=None*, *matrix=None*, *copy_from=None*, *verbose=False*)

Updates the probability matrices with a new value. Usually used via the intervention `fp.update_methods()`.

Parameters

- **source** (*str/int*) – the method to switch from
- **dest** (*str/int*) – the method to switch to
- **factor** (*float*) – if supplied, multiply the probability by this factor
- **value** (*float*) – if supplied, change the probability to this value
- **ages** (*str/list*) – the ages to modify (default: all)
- **matrix** (*str*) – which switching matrix to modify (default: annual)
- **copy_from** (*str*) – the existing method to copy the probability vectors from (optional)
- **verbose** (*bool*) – how much detail to print

reset_methods_map()

Refresh the methods map to be self-consistent

add_method(*name*, *eff*, *modern=True*)

Add a new contraceptive method to the switching matrices.

A new method should only be added before the sim is run, not during.

Note: the matrices are stored in `pars['methods']['raw']`; this method is a helper function for modifying those. For more flexibility, modify them directly. The `fp.update_methods()` intervention can be used to modify the switching probabilities later.

Parameters

- **name** (*str*) – the name of the new method
- **eff** (*float*) – the efficacy of the new method
- **modern** (*bool*) – whether it's a modern method (default: yes)

Examples::
pars = fp.pars() pars.add_method('New method', 0.90) pars.add_method(name='Male pill', eff=0.98, modern=True)

rm_method(*name*)

Removes a contraceptive method from the switching matrices.

A method should only be removed before the sim is run, not during, since the method associated with each person in the sim will point to the wrong index.

Parameters
name (*str/ind*) – the name or index of the method to remove

Example::
pars = fp.pars() pars.rm_method('Other modern')

reorder_methods(*order*)

Reorder the contraceptive method matrices.

Method reordering should be done before the sim is created (or at least before it's run).

Parameters

- **order** (*arr*) – the new order of methods, either ints or strings
- **sim** (Sim) – if supplied, also reorder

Exampls::
pars = fp.pars() pars.reorder_methods([2, 6, 4, 7, 0, 8, 5, 1, 3])

pars(*location=None*, *validate=True*, *die=True*, *update=True*, ***kwargs*)

Function for getting default parameters.

Parameters

- **location** (*str*) – the location to use for the parameters; use 'test' for a simple test set of parameters
- **validate** (*bool*) – whether to perform validation on the parameters
- **die** (*bool*) – whether to raise an exception if validation fails
- **update** (*bool*) – whether to update values during validation
- **kwargs** (*dict*) – custom parameter values

Example::
pars = fp.pars(location='senegal')

1.4.2.8 fpsim.scenarios module

Class to define and run scenarios

make_scen(**args*, ***kwargs*)

Alias for `fp.Scenario()`.

Store the specification for a single scenario (which may consist of multiple interventions).

This function is intended to be as flexible as possible; as a result, it may be somewhat confusing. There are five different ways to call it – method efficacy, method probability, method initiation/discontinuation, parameter, and custom intervention.

Args (shared):
spec (dict): a pre-made specification of a scenario; see keyword explanations below (optional) args (list): additional specifications (optional) label (str): the sim label to use for this scenario pars (dict): optionally supply additional sim parameters to use with this scenario (that take effect at the beginning of the sim, not at the point of intervention) year (float): the year at which to activate efficacy and probability scenarios matrix (str): which set of probabilities to modify for probability scenarios (e.g. annual or postpartum) ages (str/list): the age groups to modify the probabilities for

Args (efficacy):
year (float): as above eff (dict): a dictionary of method names and new efficacy values

Args (probablity):
year (float): as above matrix (str): as above ages (str): as above source (str): the method to switch from dest (str): the method to switch to factor (float): if supplied, multiply the [source, dest] probability by

this amount value (float): if supplied, instead of factor, replace the [source, dest] probability by this value copy_from (str): if supplied, copy probabilities from a different method

Args (initiation/discontinuation):

year (float): as above matrix (str): as above ages (str): as above method (str): the method for initiation/discontinuation init_factor (float): as with "factor" above, for initiation (None → method) discont_factor (float): as with "factor" above, for discontinuation (method → None) init_value (float): as with "value" above, for initiation (None → method) discont_value (float): as with "value" above, for discontinuation (method → None)

Args (parameter):

par (str): the parameter to modify par_years (float/list): the year(s) at which to apply the modifications par_vals (float/list): the value(s) of the parameter for each year

Args (custom):

interventions (Intervention/list): any custom intervention(s) to be applied to the scenario

Congratulations on making it this far.

Examples:

```
# Basic efficacy scenario
s1 = fp.make_scen(eff={'Injectables':0.99}, year=2020)

# Double rate of injectables initiation
s2 = fp.make_scen(source='None', dest='Injectables', factor=2)

# Double rate of injectables initiation -- alternate approach
s3 = fp.make_scen(method='Injectables', init_factor=2)

# More complex example: change condoms to injectables transition probability for 18-
↪25 postpartum women
s4 = fp.make_scen(source='Condoms', dest='Injectables', value=0.5, ages='18-25',␣
↪matrix='pp1to6')

# Parameter scenario: halve exposure
s5 = fp.make_scen(par='exposure_factor', years=2010, vals=0.5)

# Custom scenario
def update_sim(sim): sim.updated = True
s6 = fp.make_scen(interventions=update_sim)

# Combining multiple scenarios: change probabilities and exposure factor
s7 = fp.make_scen(
    dict(method='Injectables', init_value=0.1, discont_value=0.02, create=True),
    dict(par='exposure_factor', years=2010, vals=0.5)
)

# Scenarios can be combined
s8 = s1 + s2
```

class Scenario(*spec=None*, *label=None*, *pars=None*, *year=None*, *matrix=None*, *ages=None*, *eff=None*, *probs=None*, *source=None*, *dest=None*, *factor=None*, *value=None*, *copy_from=None*, *method=None*, *init_factor=None*, *discont_factor=None*, *init_value=None*, *discont_value=None*, *par=None*, *par_years=None*, *par_vals=None*, *interventions=None*)

Bases: `prettyobj`, `dictobj`

Store the specification for a single scenario (which may consist of multiple interventions).

This function is intended to be as flexible as possible; as a result, it may be somewhat confusing. There are five different ways to call it – method efficacy, method probability, method initiation/discontinuation, parameter, and custom intervention.

Args (shared):
spec (dict): a pre-made specification of a scenario; see keyword explanations below (optional) args (list): additional specifications (optional) label (str): the sim label to use for this scenario pars (dict): optionally supply additional sim parameters to use with this scenario (that take effect at the beginning of the sim, not at the point of intervention) year (float): the year at which to activate efficacy and probability scenarios matrix (str): which set of probabilities to modify for probability scenarios (e.g. annual or postpartum) ages (str/list): the age groups to modify the probabilities for

Args (efficacy):
year (float): as above eff (dict): a dictionary of method names and new efficacy values

Args (probablity):
year (float): as above matrix (str): as above ages (str): as above source (str): the method to switch from dest (str): the method to switch to factor (float): if supplied, multiply the [source, dest] probability by this amount value (float): if supplied, instead of factor, replace the [source, dest] probability by this value copy_from (str): if supplied, copy probabilities from a different method

Args (initiation/discontinuation):
year (float): as above matrix (str): as above ages (str): as above method (str): the method for initiation/discontinuation init_factor (float): as with "factor" above, for initiation (None → method) discont_factor (float): as with "factor" above, for discontinuation (method → None) init_value (float): as with "value" above, for initiation (None → method) discont_value (float): as with "value" above, for discontinuation (method → None)

Args (parameter):
par (str): the parameter to modify par_years (float/list): the year(s) at which to apply the modifications par_vals (float/list): the value(s) of the parameter for each year

Args (custom):
interventions (Intervention/list): any custom intervention(s) to be applied to the scenario

Congratulations on making it this far.

Examples:

```
# Basic efficacy scenario
s1 = fp.make_scen(eff={'Injectables':0.99}, year=2020)

# Double rate of injectables initiation
s2 = fp.make_scen(source='None', dest='Injectables', factor=2)

# Double rate of injectables initiation -- alternate approach
s3 = fp.make_scen(method='Injectables', init_factor=2)

# More complex example: change condoms to injectables transition probability for 18-
↪25 postpartum women
s4 = fp.make_scen(source='Condoms', dest='Injectables', value=0.5, ages='18-25',␣
↪matrix='pp1to6')

# Parameter scenario: halve exposure
s5 = fp.make_scen(par='exposure_factor', years=2010, vals=0.5)
```

(continues on next page)

(continued from previous page)

```
# Custom scenario
def update_sim(sim): sim.updated = True
s6 = fp.make_scen(interventions=update_sim)

# Combining multiple scenarios: change probabilities and exposure factor
s7 = fp.make_scen(
    dict(method='Injectables', init_value=0.1, discont_value=0.02, create=True),
    dict(par='exposure_factor', years=2010, vals=0.5)
)

# Scenarios can be combined
s8 = s1 + s2
```

update_label(*label=None*)

Ensure all specs have the correct label

run(*run_args=None*, ***kwargs*)

Shortcut for creating and running a Scenarios object based on the current scenario.

Parameters

- **run_args** (*dict*) – passed to scens.run()
- **kwargs** (*dict*) – passed to Scenarios()

class Scenarios(*pars=None*, *repeats=None*, *scens=None*, ***kwargs*)

Bases: `prettyobj`

Run different intervention scenarios.

A "scenario" can be thought of as a list of sims, all with the same parameters except for the random seed. Usually, scenarios differ from each other only in terms of the interventions run (to compare other differences between sims, it's preferable to use a MultiSim object).

Parameters

- **pars** (*dict*) – parameters to pass to the sim
- **repeats** (*int*) – how many repeats of each scenario to run (default: 1)
- **scens** (*list*) – the list of scenarios to run; see also `fp.make_scen()` and `Scenarios.add_scen()`
- **kwargs** (*dict*) – optional additional parameters to pass to the sim

Example:

```
scen1 = fp.make_scen(label='Baseline')
scen2 = fp.make_scen(year=2002, eff={'Injectables':0.99}) # Basic efficacy scenario
scens = fp.Scenarios(location='test', repeats=2, scens=[scen1, scen2])
scens.run()
```

add_scen(*scen=None*, *label=None*)

Add a scenario or scenarios to the Scenarios object

make_sims(*scenlabel*, ***kwargs*)

Create a list of sims that are all identical except for the random seed

make_scens()

Convert a scenario specification into a list of sims

run(*recompute=True*, **args*, ***kwargs*)

Actually run a list of sims

check_run()

Give a meaningful error message if the scenarios haven't been run

plot(*to_plot=None*, *plot_sims=True*, ***kwargs*)

Plot the scenarios with bands – see `sim.plot()` for args

plot_sims(*to_plot=None*, *plot_sims=True*, ***kwargs*)

Plot each sim as a separate line across all senarios – see `sim.plot()` for args

analyze_sims(*start=None*, *end=None*)

Take a list of sims that have different labels and extrapolate statistics from each

1.4.2.9 fpsim.settings module

Define options for FPsim, mostly plotting and Numba options. All options should be set using set() or directly, e.g.:

```
fp.options(font_size=18)
```

To reset default options, use:

```
fp.options('default')
```

Note: "options" is used to refer to the choices available (e.g., DPI), while "settings" is used to refer to the choices made (e.g., DPI=150).

1.4.2.10 fpsim.sim module

Defines the Sim class, the core class of the FP model (FPsim).

class People(*pars*, *n=None*, ***kwargs*)

Bases: *BasePeople*

Class for all the people in the simulation.

update_method()

Uses a switching matrix from DHS data to decide based on a person's original method their probability of changing to a new method and assigns them the new method. Currently allows switching on whole calendar years to enter function. Matrix serves as an initiation, discontinuation, continuation, and switching matrix. Transition probabilities are for 1 year and only for women who have not given birth within the last 6 months.

update_method_pp()

Utilizes data from birth to allow agent to initiate a method postpartum coming from birth by 3 months postpartum and then initiate, continue, or discontinue a method by 6 months postpartum. Next opportunity to switch methods will be on whole calendar years, whenever that falls.

update_methods()

If eligible (age 15-49 and not pregnant), choose new method or stay with current one

check_mortality()

Decide if person dies at a timestep

check_sexually_active()

Decide if agent is sexually active based either on month postpartum or age if not postpartum. Postpartum and general age-based data from DHS.

check_conception()

Decide if person (female) becomes pregnant at a timestep.

make_pregnant()

Update the selected agents to be pregnant

check_lam()

Check to see if postpartum agent meets criteria for LAM in this time step

update_breastfeeding()

Track breastfeeding, and update time of breastfeeding for individual pregnancy. Agents are randomly assigned a duration value based on a gumbel distribution drawn from the 2018 DHS variable for breastfeeding months. The mean (mu) and the std dev (beta) are both drawn from that distribution in the DHS data.

update_postpartum()

Track duration of extended postpartum period (0-24 months after birth). Only enter this function if agent is postpartum

update_pregnancy()

Advance pregnancy in time and check for miscarriage

reset_breastfeeding()

Stop breastfeeding, calculate total lifetime duration so far, and reset lactation episode to zero

check_maternal_mortality()

Check for probability of maternal mortality

check_infant_mortality()

Check for probability of infant mortality (death < 1 year of age)

check_delivery()

Decide if pregnant woman gives birth and explore maternal mortality and child mortality

update_age()

Advance age in the simulation

update_age_bin_totals()

Count how many total live women in each 5-year age bin 10-50, for tabulating ASFR

log_age_split(*binned_ages_t*, *channel*, *numerators*, *denominators=None*)

track_mcpr()

Track for purposes of calculating mCPR at the end of the timestep after all people are updated Not including LAM users in mCPR as this model counts all women passively using LAM but DHS data records only women who self-report LAM which is much lower. Follows the DHS definition of mCPR

track_cpr()

Track for purposes of calculating newer ways to conceptualize contraceptive prevalence at the end of the timestep after all people are updated Includes women using any method of contraception, including LAM Denominator of possible users includes all women aged 15-49

track_acpr()

Track for purposes of calculating newer ways to conceptualize contraceptive prevalence at the end of the timestep after all people are updated Denominator of possible users excludes pregnant women and those not sexually active in the last 4 weeks Used to compare new metrics of contraceptive prevalence and eventually unmet need to traditional mCPR definitions

init_step_results()

update()

Update the person's state for the given timestep. t is the time in the simulation in years (ie, 0-60), y is years of simulation (ie, 1960-2010)

class Sim(*pars=None*, *location=None*, *label=None*, *track_children=False*, ***kwargs*)

Bases: *BaseSim*

The Sim class handles the running of the simulation: the creation of the population and the dynamics of the epidemic. This class handles the mechanics of the actual simulation, while BaseSim takes care of housekeeping (saving, loading, exporting, etc.). Please see the BaseSim class for additional methods.

Parameters

- **pars** (*dict*) – parameters to modify from their default values
- **location** (*str*) – name of the location (country) to look for data file to load
- **label** (*str*) – the name of the simulation (useful to distinguish in batch runs)
- **track_children** (*bool*) – whether to track links between mothers and their children (slow, so disabled by default)
- **kwargs** (*dict*) – additional parameters; passed to `fp.make_pars()`

Examples:

```
sim = fp.Sim()
sim = fp.Sim(n_agents=10e3, location='senegal', label='My small Seneagl sim')
```

initialize(*force=False*)

init_results()

get_age_sex(*n*)

For an ex nihilo person, figure out if they are male and female, and how old

make_people(*n=1*, *age=None*, *sex=None*, *method=None*, *debut_age=None*)

Set up each person

init_people(*output=False*, ***kwargs*)

Create the people

update_methods()

Update all contraceptive method matrices to have probabilities that follow a trend closest to the year the sim is on based on mCPR in that year

update_mortality()

Update infant and maternal mortality for the sim's current year. Update general mortality trend as this uses a spline interpolation instead of an array

update_mothers()

Add link between newly added individuals and their mothers

`apply_interventions()`

Apply each intervention in the model

`apply_analyzers()`

Apply each analyzer in the model

`finalize_interventions()`

Make any final updates to interventions (e.g. to shrink)

`finalize_analyzers()`

Make any final updates to analyzers (e.g. to shrink)

`run`(*verbose=None*)

Run the simulation

`store_postpartum()`

Stores snapshot of who is currently pregnant, their parity, and various postpartum states in final step of model for use in calibration

`to_df`(*include_range=False*)

Export all sim results to a dataframe

Parameters

include_range (*bool*) – if True, and if the sim results have best, high, and low, then export all of them; else just best

`conform_y_axes`(*figure*, *bottom=0*, *top=100*)

`plot`(*to_plot=None*, *xlims=None*, *ylims=None*, *do_save=None*, *do_show=True*, *filename='fpsim.png'*, *style=None*, *fig_args=None*, *plot_args=None*, *axis_args=None*, *fill_args=None*, *label=None*, *new_fig=True*, *colors=None*)

Plot the results – can supply arguments for both the figure and the plots.

Parameters

- **to_plot** (*str/dict*) – What to plot (e.g. 'default' or 'cpr'), or a dictionary of result:label pairs
- **xlims** (*list/dict*) – passed to pl.xlim() (use `[None, None]` for default)
- **ylims** (*list/dict*) – passed to pl.ylim()
- **do_save** (*bool*) – Whether or not to save the figure. If a string, save to that filename.
- **do_show** (*bool*) – Whether to show the plots at the end
- **filename** (*str*) – If a figure is saved, use this filename
- **style** (*bool*) – Custom style arguments
- **fig_args** (*dict*) – Passed to pl.figure() (plus `nrows` and `ncols` for overriding defaults)
- **plot_args** (*dict*) – Passed to pl.plot()
- **axis_args** (*dict*) – Passed to pl.subplots_adjust()
- **fill_args** (*dict*) – Passed to pl.fill_between())
- **label** (*str*) – Label to override default
- **new_fig** (*bool*) – Whether to create a new figure (true unless part of a multisim)
- **colors** (*list/dict*) – Colors for plots with multiple lines

plot_age_first_birth(*do_show=None*, *do_save=None*, *fig_args=None*, *filename='first_birth_age.png'*)

Plot age at first birth

Parameters

- **fig_args** (*dict*) – arguments to pass to `pl.figure()`
- **do_show** (*bool*) – whether or not the user wants to show the output plot (default: true)
- **do_save** (*bool*) – whether or not the user wants to save the plot to filepath (default: false)
- **filename** (*str*) – the name of the path to output the plot

compute_method_usage()

Computes method mix proportions from a sim object

Returns

list of lists where list[years_after_start][method_index] == proportion of fecundity aged women using that method on that year

format_method_df(*method_list=None*, *timeseries=False*)

Outputs a dataframe for method mix plotting for either a single year or a timeseries

Parameters

- **method_list** (*list*) – list of proportions where each index is equal to the integer value of the corresponding method
- **timeseries** (*boolean*) – if true, provides a dataframe with data from every year, otherwise a method_list is required for the year

Returns

pandas.DataFrame with columns ["Percentage", "Method", "Sim", "Seed"] and optionally "Year" if timeseries

class MultiSim(*sims=None*, *base_sim=None*, *label=None*, *n=None*, ***kwargs*)

Bases: `prettyobj`

The MultiSim class handles the running of multiple simulations

run(*compute_stats=True*, ***kwargs*)

Run all simulations in the MultiSim

compute_stats(*return_raw=False*, *quantiles=None*, *use_mean=False*, *bounds=None*)

Compute statistics across multiple sims

static merge(**args*, *base=False*)

Convenience method for merging two MultiSim objects.

Parameters

- **args** (`MultiSim`) – the MultiSims to merge (either a list, or separate)
- **base** (*bool*) – if True, make a new list of sims from the multisim's two base sims; otherwise, merge the multisim's lists of sims

Returns

a new MultiSim object

Return type

msim (*MultiSim*)

Examples:

mm1 = fp.MultiSim.merge(msim1, msim2, base=True) mm2 = fp.MultiSim.merge([m1, m2, m3, m4], base=False)

split(*inds=None*, *chunks=None*)

Convenience method for splitting one MultiSim into several. You can specify either individual indices of simulations to extract, via inds, or consecutive chunks of indices, via chunks. If this function is called on a merged MultiSim, the chunks can be retrieved automatically and no arguments are necessary.

Parameters

- **inds** (*list*) – a list of lists of indices, with each list turned into a MultiSim
- **chunks** (*int or list*) – if an int, split the MultiSim into that many chunks; if a list return chunks of that many sims

Returns

A list of MultiSim objects

Examples:

```
m1 = fp.MultiSim(fp.Sim(label='sim1'))
m2 = fp.MultiSim(fp.Sim(label='sim2'))
m3 = fp.MultiSim.merge(m1, m2)
m3.run()
m1b, m2b = m3.split()

msim = fp.MultiSim(fp.Sim(), n_runs=6)
msim.run()
m1, m2 = msim.split(inds=[[0,2,4], [1,3,5]])
mlist1 = msim.split(chunks=[2,4]) # Equivalent to inds=[[0,1], [2,3,4,5]]
mlist2 = msim.split(chunks=2) # Equivalent to inds=[[0,1,2], [3,4,5]]
```

remerge(*base=True*, *recompute=True*, ***kwargs*)

Split a sim, compute stats, and re-merge.

Parameters

- **base** (*bool*) – whether to use the base sim (otherwise, has no effect)
- **kwargs** (*dict*) – passed to msim.split()

Note: returns a new MultiSim object (if that concerns you).

to_df(*yearly=False*, *mean=False*)

Export all individual sim results to a dataframe

plot(*to_plot=None*, *plot_sims=True*, *do_show=None*, *do_save=None*, *filename='fp_multisim.png'*, *fig_args=None*, *axis_args=None*, *plot_args=None*, *style=None*, *colors=None*, ***kwargs*)

Plot the MultiSim

Parameters

plot_sims (*bool*) – whether to plot individual sims (else, plot with uncertainty bands)

See `sim.plot()` for additional args.

plot_age_first_birth(*do_show=False*, *do_save=True*, *output_file='age_first_birth_multi.png'*)

parallel(**args*, ***kwargs*)

A shortcut to `fp.MultiSim()`, allowing the quick running of multiple simulations at once.

Parameters

- **args** (*list*) – The simulations to run
- **kwargs** (*dict*) – passed to multi_run()

Returns
A run MultiSim object.

Examples:

```
s1 = fp.Sim(exposure_factor=0.5, label='Low')
s2 = fp.Sim(exposure_factor=2.0, label='High')
fp.parallel(s1, s2).plot()
msim = fp.parallel(s1, s2)
```

1.4.2.11 fpsim.utils module

File for storing utilities and probability calculators needed to run FP model

set_seed(*seed=None*)
Reset the random seed – complicated because of Numba

bt(*prob*)
A simple Bernoulli (binomial) trial

bc(*prob*, *repeats*)
A binomial count

rbt(*prob*, *repeats*)
A repeated Bernoulli (binomial) trial

mt(*probs*)
A multinomial trial

sample(*dist='uniform'*, *par1=0*, *par2=1*, *size=1*, ***kwargs*)
Draw a sample from the distribution specified by the input. The available distributions are:

- 'uniform' : uniform distribution from low=par1 to high=par2; mean is equal to (par1+par2)/2
- 'normal' : normal distribution with mean=par1 and std=par2
- 'lognormal' : lognormal distribution with mean=par1 and std=par2 (parameters are for the lognormal distribution, *not* the underlying normal distribution)
- 'normal_pos' : right-sided normal distribution (i.e. only positive values), with mean=par1 and std=par2 *of the underlying normal distribution*
- 'normal_int' : normal distribution with mean=par1 and std=par2, returns only integer values
- 'lognormal_int' : lognormal distribution with mean=par1 and std=par2, returns only integer values
- 'poisson' : Poisson distribution with rate=par1 (par2 is not used); mean and variance are equal to par1
- 'neg_binomial' : negative binomial distribution with mean=par1 and k=par2; converges to Poisson with k=∞

Parameters

- **dist** (*str*) – the distribution to sample from
- **par1** (*float*) – the "main" distribution parameter (e.g. mean)
- **par2** (*float*) – the "secondary" distribution parameter (e.g. std)

- **size** (*int*) – the number of samples (default=1)
- **kwargs** (*dict*) – passed to individual sampling functions

Returns
A length N array of samples

Examples:

```
fp.sample() # returns Unif(0,1)
fp.sample(dist='normal', par1=3, par2=0.5) # returns Normal(=3, =0.5)
fp.sample(dist='lognormal_int', par1=5, par2=3) # returns a lognormally distributed␣
↪set of values with mean 5 and std 3
```

Notes

Lognormal distributions are parameterized with reference to the underlying normal distribution (see: https://docs.scipy.org/doc/numpy-1.14.0/reference/generated/numpy.random.lognormal.html), but this function assumes the user wants to specify the mean and std of the lognormal distribution.

Negative binomial distributions are parameterized with reference to the mean and dispersion parameter k (see: https://en.wikipedia.org/wiki/Negative_binomial_distribution). The r parameter of the underlying distribution is then calculated from the desired mean and k. For a small mean (~1), a dispersion parameter of ∞ corresponds to the variance and standard deviation being equal to the mean (i.e., Poisson). For a large mean (e.g. >100), a dispersion parameter of 1 corresponds to the standard deviation being equal to the mean.

1.4.2.12 fpsim.version module

PYTHON MODULE INDEX

f

INDEX

G

I

K

L

M

N

P

R

S

T

U

V

W

Y