The Journal of Open Source Software

Example 3: Simplifying scientific software in Python

Cliff C. Kerr [1,2¶], **Paula Sanz-Leon** [1], **Romesh G. Abeysuriya** [1,3], **George L. Chadderdon** [3,4], **Vlad-Ştefan Harbuz**[5], **Parham Saidi**[5], **Maria del Mar Quiroga** [3,6], **Rowan Martin-Hughes** [3], **Sherrie L. Kelly** [3], **Jamie A. Cohen** [1], **Robyn M. Stuart** [1,7], **and Anna Nachesa**[8]

1 Institute for Disease Modeling, Global Health Division, Bill & Melinda Gates Foundation, Seattle, USA **2** School of Physics, University of Sydney, Sydney, Australia **3** Burnet Institute, Melbourne, Australia **4** CAE USA, Tampa, USA **5** Saffron Software, Bucharest, Romania **6** Melbourne Data Analytics Platform, The University of Melbourne, Melbourne, Australia **7** Department of Mathematical Sciences, University of Copenhagen, Copenhagen, Denmark **8** Google, Zürich, Switzerland ¶ Corresponding author

DOI: 10.21105/medportal.00032

Software

- Review
- Repository
- Archive

Editor: Max Proft

Reviewers:

- @max-anu

Submitted: 23 August 2023
Published: 23 August 2023

Summary

Sciris aims to streamline the development of scientific software by making it easier to perform common tasks. Sciris provides classes and functions that simplify access to frequently used low-level functionality in the core libraries of the scientific Python ecosystem (such as `numpy` for math and `matplotlib` for plotting), as well as in libraries of broader scope (such as `multiprocess` for parallelization and `pickle` for saving and loading objects). While low-level functionality is valuable for developing robust software applications, it can divert focus from the scientific problems being solved. Some of Sciris' key features include: ensuring consistent dictionary, list, and array types (e.g., enabling users to provide inputs as either lists or arrays); enabling ordered dictionary elements to be referenced by index; simplifying datetime arithmetic by allowing date input in multiple formats, including strings; simplifying the saving and loading of files and complex objects; and simplifying the parallel execution of code. With Sciris, users can often achieve the same functionality with fewer lines of code, avoid reinventing the wheel, and spend less time looking up recipes on Stack Overflow. This can make writing scientific code in Python faster, more pleasant, and more accessible, especially for people without extensive training in software development.

Statement of need

The landscape of scientific software

With the increasing availability of large volumes of data and computing resources, scientists across multiple fields of research have been able to tackle increasingly complex problems. But to harness these resources, the need for domain-specific software has become much greater. As the complexity of the questions being tackled has increased, so too has the amount of code used to answer them, creating a steep learning curve and significant burden of code review (Nature Editorial Board, 2018). For some scientists, this increasing reliance on software has created a barrier between themselves and the science they want to do. It is these people – people who want things to "just work" rather than worry about the implementation details – who are the primary audience for Sciris. (In contrast, people who care a lot about implementation details – such as those who love using type hints – will likely *not* find Sciris to be as helpful.)

Scientific code workflows (e.g., either a full cycle in the development of a new software library, or in the execution of a one-off analysis) typically rely on multiple codebases, including but not limited to: low-level libraries, domain-specific open-source software, and self-developed

and/or inherited Swiss-Army-knife toolboxes (whose original developer may or may not be around to pass on undocumented wisdom). Several scientific communities have adopted collaborative, community-driven, open-source software approaches due to the significant savings in development costs and increases in code quality that they afford, such as `astropy` (Robitaille et al., 2013), `fmriprep` (Esteban et al., 2019), and `nextstrain` (Hadfield et al., 2018). Despite this progress, a large fraction of scientific software development efforts remain a solo adventure (Kerr, 2019). This leads to proliferation of tools where resources are largely spent reinventing wheels of variable quality, which jeopardizes the code's minimum requirements of being "re-runnable, repeatable, reproducible, reusable, and replicable" (Benureau & Rougier, 2018).

In addition, low-level programming abstractions can make it harder to clarify the science. For instance, one of the reasons PyTorch has become popular in academic and research environments is its success in making models easier to write compared to TensorFlow (Lorica, 2017). The need for libraries that provide "simplifying interfaces" for research applications is reflected in the development of multiple libraries in scientific Python ecosystems that have enabled researchers to focus their time and efforts on solving problems, prototyping solutions, deploying applications, and educating their communities. In addition to PyTorch (simplifying/extending Tensorflow), other examples include seaborn (simplifying/extending Matplotlib) (Waskom, 2021), pingouin (simplifying/extending pandas), and PyVista (simplifying/extending VTK) (Sullivan & Kaszynski, 2019), among many others. Sciris adds to this ecosystem as a "library of the gaps", addressing annoyances that are too small-scale to each need a dedicated library of their own, but common enough that together they add up to significant coding burden.

Sciris in practice

The name Sciris is a portmanteau of "scientific" and "iris" (a reference to seeing clearly, as well as the Greek word for "rainbow"). We began work on it in 2014, initially to support development of Optima HIV (Kerr et al., 2015, 2020). We repeatedly encountered the same inconveniences while building scientific webapps, and so we began collecting the tools we used to overcome them into a shared library. While Python is considered an easy-to-use language for beginners, the motivation that shaped Sciris' evolution was to further lower the barriers to accessing the numerous supporting libraries we were using.

Our investments in Sciris paid off when in early 2020 its combination of brevity and simplicity proved crucial in enabling the rapid development of the Covasim model of COVID-19 transmission (Kerr et al., 2021). Covasim's relative simplicity and readability, based in large part on its heavy use of Sciris, enabled it to become one of the most widely adopted models of COVID-19, used by students, researchers, and policymakers in over 30 countries (Kerr et al., 2022).

In addition to Optima HIV and Covasim, Sciris is currently used by many other scientific software tools, such as Optima Nutrition (Pearson et al., 2018), the Cascade Analysis Tool (Kedziora et al., 2019), Atomica (The Atomica Team, 2020), Optima TB (Goscé et al., 2021), the Health Interventions Prioritization Tool (Fraser-Hurt et al., 2021), SynthPops (Mistry et al., in preparation), and FPsim (O'Brien et al., 2022).

We believe using Sciris can lead to more efficient scientific code production for solo developers and teams alike, including increased longevity of new scientific libraries (Perkel, 2020). Some of the key functional aspects that Sciris provides are: (i) brevity through simple interfaces; (ii) "dejargonification"; (iii) fine-grained exception handling; and (iv) version management. We expand on each of these below, but first provide a vignette that illustrates many of Sciris' features.

Vignette

Compared with a domain-specific language like MATLAB, even relatively simple scientific code in Python can require significant boilerplate. This extra code can obscure the key logic of the scientific question being addressed.

For example, imagine that we wish to sample random numbers from a user-defined function with varying noise levels, save the intermediate calculations, and plot the results. In vanilla Python, each of these operations is somewhat cumbersome. Figure 1 presents two functionally identical scripts; the one written with Sciris is considerably more readable and succinct.

This vignette illustrates many of Sciris' most-used features, including timing, parallelization, feature-rich containers, file saving and loading, and plotting. For the lines of the script that differ, Sciris reduces the number of lines of code required from 33 to 7, a 79% decrease.

```
# Define random wave generator
import numpy as np

def randwave(std, xmin=0, xmax=10, npts=50):
    np.random.seed(int(100*std)) # Ensure differences between runs
    a = np.cos(np.linspace(xmin, xmax, npts))
    b = np.random.randn(npts)
    return a + b*std

# Other imports
- import time
- import multiprocessing as mp
- import pickle
- import gzip
- import matplotlib.pyplot as plt
- from mpl_toolkits.mplot3d import Axes3D # Unused but must be imported

# Start timing
- start = time.time()

# Calculate output in parallel
- multipool = mp.Pool(processes=mp.cpu_count())
- waves = multipool.map(randwave, np.linspace(0, 1, 11))
- multipool.close()
- multipool.join()

# Save to files
- filenames = []
- for i,wave in enumerate(waves):
-     filename = f'wave{i}.obj'
-     with gzip.GzipFile(filename, 'wb') as fileobj:
-         fileobj.write(pickle.dumps(wave))
-     filenames.append(filename)

# Create dict from files
- data_dict = {}
- for fname in filenames:
-     with gzip.GzipFile(fname) as fileobj:
-         filestring = fileobj.read()
-         data_dict[fname] = pickle.loads(filestring)

# Create 3D plot
- data = np.array([data_dict[fname] for fname in filenames])
- fig = plt.figure()
- ax = plt.axes(projection='3d')
- ny,nx = np.array(data).shape
- x = np.arange(nx)
- y = np.arange(ny)
- X, Y = np.meshgrid(x, y)
- surf = ax.plot_surface(X, Y, data, cmap='coolwarm')
- fig.colorbar(surf)

# Print elapsed time
- elapsed = time.time() - start
- print(f'Elapsed time: {elapsed:0.1f} s')
```

```
# Define random wave generator
import numpy as np

def randwave(std, xmin=0, xmax=10, npts=50):
    np.random.seed(int(100*std)) # Ensure differences between runs
    a = np.cos(np.linspace(xmin, xmax, npts))
    b = np.random.randn(npts)
    return a + b*std

# Other imports
+ import sciris as sc

# Start timing
+ T = sc.timer()

# Calculate output in parallel
+ waves = sc.parallelize(randwave, np.linspace(0, 1, 11))

# Save to files
+ filenames = [sc.save(f'wave{i}.obj', wave) for i,wave in enumerate(waves)]

# Create dict from files
+ data = sc.odict({fname:sc.load(fname) for fname in filenames})

# Create 3D plot
+ sc.surf3d(data[:],cmap='orangeblue')

# Print elapsed time
+ T.toc()
```

Figure 1: Comparison of functionally identical scripts without Sciris (left) and with Sciris (right), showing a nearly five-fold reduction in lines of code required (excluding whitespace, comments, and the shared "wave generator" code), from 33 lines to 7. The resulting plots are shown in Figure 2.

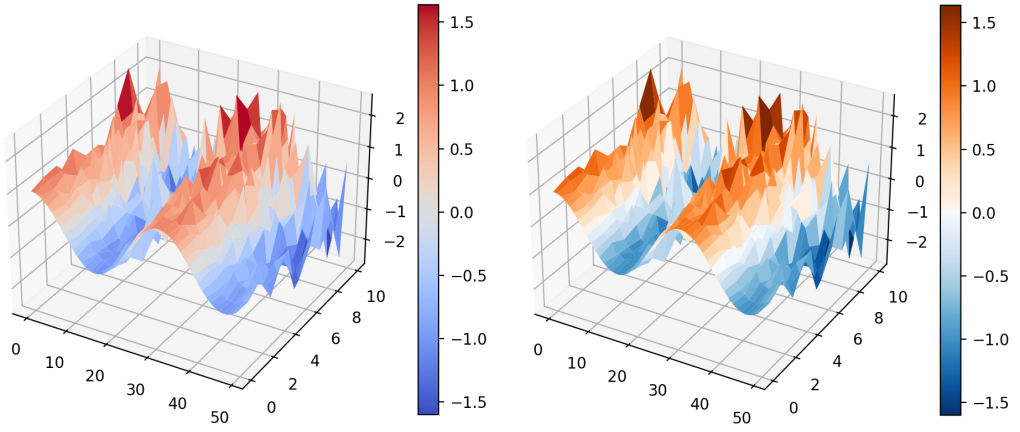

Figure 2: Output of the scripts shown in Figure 1, without Sciris (left) and with Sciris (right). The two plots are identical except for the new high-contrast colormap available in Sciris.

Design philosophy

The aim of Sciris is to make common tasks simpler. Sciris includes implementations of heavily used code patterns and abstractions that facilitate the development and deployment of complex domain-specific scientific applications, and helps non-specialist audiences interact with these applications. We note that Sciris "stands on the shoulders of giants", and as such is not intended as a replacement of these libraries, but rather as an interface that facilitates a more effective and sustainable development process through the following principles:

Brevity through simple interfaces. Sciris packages common patterns requiring multiple lines of code into single, simple functions. With these functions one can succinctly express and execute frequent plotting tasks (e.g., `sc.commaticks`, `sc.dateformatter`, `sc.plot3d`); ensure consistent types, including containers (e.g., `sc.toarray`, `sc.mergedicts`, `sc.mergelists`), or even perform line-by-line performance profiling (`sc.profile`). Brevity is also achieved by extending functionality of well established objects (e.g., `OrderedDict` via `sc.odict`) and methods (e.g., `isinstance` via `sc.checktype` that enables the comparison of objects against higher-level types like `arraylike`), as well as wrapping useful third-party libraries (e.g., `pyyaml` via `sc.loadyaml`). In providing a curated collection of common data science tools, Sciris has similarities to R's tidyverse.

Dejargonification. Sciris aims to use plain function names (e.g., `sc.smooth`, `sc.findnearest`, `sc.safedivide`) so that the resulting code is as scientifically clear and human-readable as possible. Sciris also provides some MATLAB-like functionality, and uses the same names (e.g., `sc.tic` and `sc.toc`; `sc.boxoff`) to minimize the learning curve for scientists who have MATLAB experience.

Fine-grained exception handling. Across many classes and functions, Sciris uses the keyword `die`, enabling users to set a locally scoped level of strictness in the handling of exceptions. If `die=False`, Sciris is more forgiving and softly handles exceptions by using its default (opinionated) behavior, such as printing a warning and returning `None` so users can decide how to proceed. If `die=True`, it directly raises the corresponding exception and message. This flexibility reduces the need for try-catch blocks, which can distract from the code's scientific logic.

Version management. Keeping track of dates, authors, and code versions, plus additional notes or comments, is an essential part of scientific projects. Sciris provides methods to easily save and load metadata to/from figure files, including Git information (`sc.savefig`, `sc.gitinfo`,

`sc.loadmetadata`), as well as shortcuts for comparing module versions (`sc.compareversions`) or requiring them (`sc.require`).

Examples of key features

Here we illustrate a smattering of key features in greater detail; further information on installation and usage can be found at docs.sciris.org. Figure 3 illustrates the functional modules of Sciris. Sciris is available on pip (`pip install sciris`).

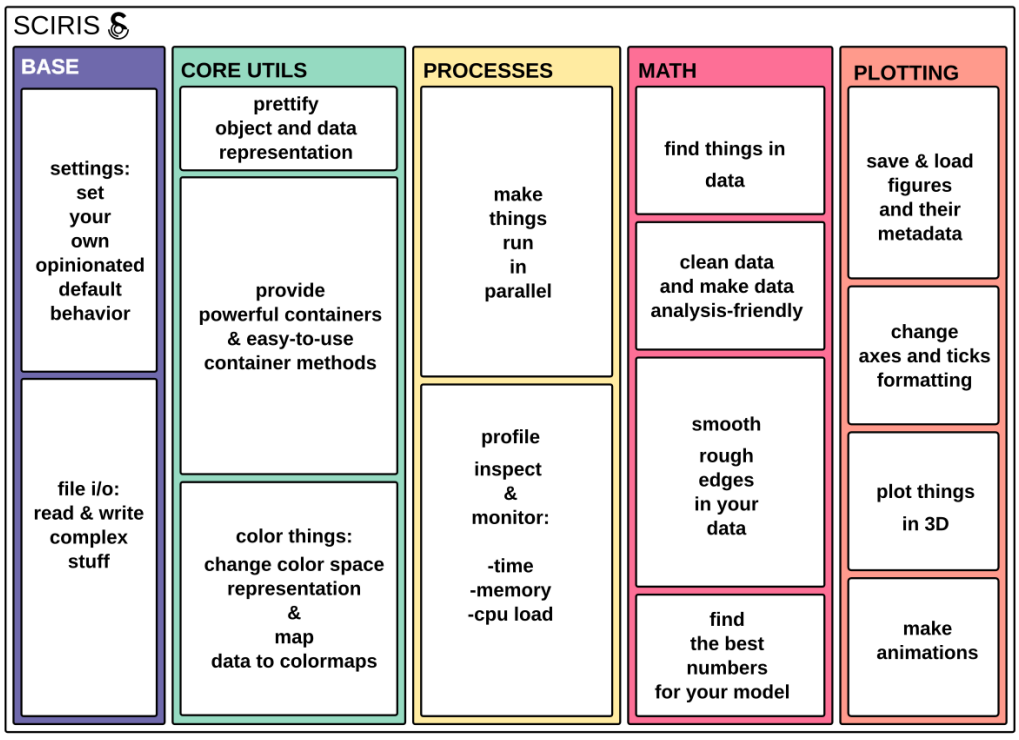

Figure 3: Block diagram of Sciris' functionality, grouped by high-level concepts and types of tasks that are commonly performed in scientific code.

Feature-rich containers

One of the key features in Sciris is `sc.odict`, a flexible container representing an associative array with the best-of-all-worlds features of lists, dictionaries, and numeric arrays. This is based on `OrderedDict` from `collections`, but supports list methods like integer indexing, key slicing, and item insertion:

```
data = sc.odict(a=[1,2,3], b=[4,5,6])
assert data['a'] == data[0]
assert data[:].sum() == 21
for i, key, value in data.enumitems():
  print(f'Item {i} is named "{key}" and has value {value}')
# Output:
# Item 0 is named "a" and has value [1, 2, 3]
# Item 1 is named "b" and has value [4, 5, 6]
```

Numerical utilities

Indexing arrays is a common task in NumPy, but can be difficult due to incompatibilities of object type. `sc.findinds` will find matches even if two things are not exactly equal due to differences in type (e.g., floats vs. integers, lists vs. arrays). The code shown below produces the same result as calling `np.nonzero(np.isclose(arr, val))[0]`.

```
sc.findinds([2,3,6,3], 3.0)
# Output:
# array([1,3])
```

Parallelization

A frequent hurdle scientists face is parallelization. Sciris provides `sc.parallelize`, which acts as a shortcut for using `multiprocess.Pool()`. By default it adjusts the pool size based on the CPUs available, but can also use either a fixed number of CPUs or allocate them dynamically based on load (`sc.loadbalancer`). This example shows three equivalent ways to iterate over multiple complex arguments:

```
def f(x, y):
   return x*y

out1 = sc.parallelize(func=f, iterarg=[(1,2),(2,3),(3,4)])
out2 = sc.parallelize(func=f, iterkwargs={'x':[1,2,3], 'y':[2,3,4]})
out3 = sc.parallelize(func=f, iterkwargs=[{'x':1, 'y':2},
                                          {'x':2, 'y':3},
                                          {'x':3, 'y':4}])
```

Plotting

Numerous shortcuts for customizing and prettifying plots are available in Sciris. Several commonly used features are illustrated below, with the results shown in Figure 4:

```
sc.options(font='Garamond') # Set custom font
x = sc.daterange('2022-06-01', '2022-12-31', as_date=True) # Create dates
y = sc.smooth(np.random.randn(len(x))**2)*1000 # Create smoothed random numbers
c = sc.vectocolor(y, cmap='turbo') # Set colors proportional to y values
plt.scatter(x, y, c=c) # Plot the data
sc.dateformatter() # Automatic x-axis date formatter
sc.commaticks() # Add commas to y-axis tick labels
sc.setylim() # Automatically set the y-axis limits
sc.boxoff() # Remove the top and right axis spines
```

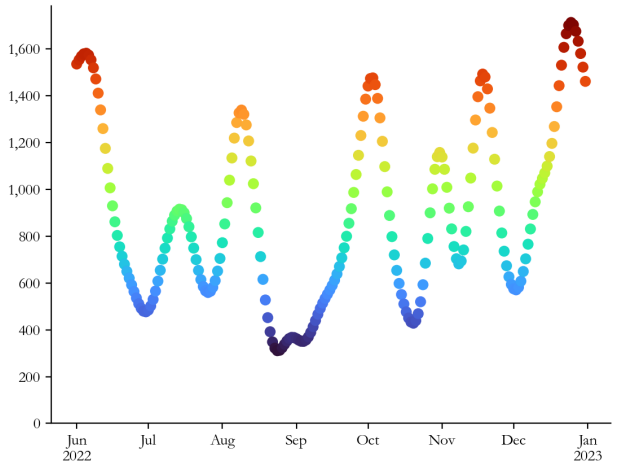

Figure 4: Example of plot customizations via Sciris, including x- and y-axis tick labels and the font.

ScirisWeb

While a full description of ScirisWeb is beyond the scope of this paper, briefly, it builds on Sciris to enable the rapid development of Python-based webapps, including those powering Covasim and Optima Nutrition. By default, ScirisWeb uses Vuejs and sciris-js for the frontend, Flask as the web framework, Redis for the (optional) database, and Matplotlib/mpld3 for plotting. However, ScirisWeb is completely modular, which means that it could also be used to (for example) link a React frontend to a MySQL database with Plotly figures. This modularity is in contrast to full-stack solutions such as Shiny for Python, Plotly Dash, Streamlit, and Voilà. While these libraries are even easier to use than ScirisWeb (since they do not require any knowledge of JavaScript), they provide limited options for customization or switching between technology stacks. In contrast, ScirisWeb provides the flexibility of a custom-written webapp within the context of an "it just works" framework.

Beyond Sciris

Like seaborn, Sciris aims to "facilitate rapid exploration and prototyping through named functions and opinionated defaults" (Waskom, 2021). Eventually, a time may come when the user's opinions diverge from Sciris' defaults. Since most Sciris functions are standalone, individual functions can be replaced on as as-needed basis. For example, in situations where strictness is an asset (e.g., low-level libraries where an unexpected type is indicative of an error), the added flexibility that Sciris provides (e.g., the type-agnostic `sc.toarray`) can be a liability. As another example, `sc.odict` adds small but nonzero overhead to the `dict` built-in. While in most cases this performance difference is negligible (<500 ms per million set/get operations), for innermost loops of compute-intensive applications, `dict` should be used instead. Finally, since Sciris aims for breadth rather than depth, Sciris functions may eventually need to be supplanted by more powerful alternatives. For example, while `sc.parallelize` provides one-line parallelization on a local machine or single virtual machine, parallelizing across multiple machines requires more powerful libraries such as Dask (Rocklin, 2015), Ray, or Celery.

Acknowledgements

The Sciris Development Team (info@sciris.org) wishes to thank David J. Kedziora, Dominic Delport, Kevin M. Jablonka, Meikang Wu, and Dina Mistry for providing helpful feedback on the Sciris library. David P. Wilson, William B. Lytton, and Daniel J. Klein provided in-kind support of Sciris development. Financial and personnel support has been provided by the United States Defense Advanced Research Projects Agency (DARPA) Contract N66001-10-C-2008 (2010–2014), World Bank Assignment 1045478 (2011–2015), the Australian National Health and Medical Research Council (NHMRC) Project Grant APP1086540 (2015–2017), the Australian Research Council (ARC) Discovery Early Career Research Award (DECRA) Fellowship Grant DE140101375 (2014–2019), Intellectual Ventures (2019–2021), and the Bill & Melinda Gates Foundation (2021–present).

References

Benureau, F. C., & Rougier, N. P. (2018). Re-run, repeat, reproduce, reuse, replicate: Transforming code into scientific contributions. *Frontiers in Neuroinformatics*, *11*, 69. https://doi.org/10.3389/fninf.2017.00069

Esteban, O., Markiewicz, C. J., Blair, R. W., Moodie, C. A., Isik, A. I., Erramuzpe, A., Kent, J. D., Goncalves, M., DuPre, E., Snyder, M., & others. (2019). fMRIPrep: A robust preprocessing pipeline for functional MRI. *Nature Methods*, *16*(1), 111–116. https://doi.org/10.1038/s41592-018-0235-4

Fraser-Hurt, N., Hou, X., Wilkinson, T., Duran, D., Abou Jaoude, G. J., Skordis, J., Chukwuma, A., Lao Pena, C., Tshivuila Matala, O. O., Gorgens, M., & others. (2021). Using allocative efficiency analysis to inform health benefits package design for progressing towards universal health coverage: Proof-of-concept studies in countries seeking decision support. *PLOS One*, *16*(11), e0260247. https://doi.org/10.1371/journal.pone.0260247

Goscé, L., Abou Jaoude, G. J., Kedziora, D. J., Benedikt, C., Hussain, A., Jarvis, S., Skrahina, A., Klimuk, D., Hurevich, H., Zhao, F., & others. (2021). Optima TB: A tool to help optimally allocate tuberculosis spending. *PLOS Computational Biology*, *17*(9), e1009255. https://doi.org/10.1371/journal.pcbi.1009255

Hadfield, J., Megill, C., Bell, S. M., Huddleston, J., Potter, B., Callender, C., Sagulenko, P., Bedford, T., & Neher, R. A. (2018). Nextstrain: Real-time tracking of pathogen evolution. *Bioinformatics*, *34*(23), 4121–4123. https://doi.org/10.1093/bioinformatics/bty407

Kedziora, D. J., Abeysuriya, R., Kerr, C. C., Chadderdon, G. L., Harbuz, V.-Ș., Metzger, S., Wilson, D. P., & Stuart, R. M. (2019). The Cascade Analysis Tool: Software to analyze and optimize care cascades. *Gates Open Research*, *3*. https://doi.org/10.12688/gatesopenres.13031.2

Kerr, C. C. (2019). Is epidemiology ready for big software? *Pathogens and Disease*, *77*(1), ftz006. https://doi.org/10.1093/femspd/ftz006

Kerr, C. C., Stuart, R. M., Gray, R. T., Shattock, A. J., Fraser-Hurt, N., Benedikt, C., Haacker, M., Berdnikov, M., Mahmood, A. M., Jaber, S. A., & others. (2015). Optima: A model for HIV epidemic analysis, program prioritization, and resource optimization. *Journal of Acquired Immune Deficiency Syndromes*, *69*(3), 365–376.

Kerr, C. C., Stuart, R. M., Kedziora, D. J., Brown, A., Abeysuriya, R., Chadderdon, G. L., Nachesa, A., & Wilson, D. P. (2020). Optima HIV methodology and approach. In F. Zhao, C. Benedikt, & D. Wilson (Eds.), *Tackling the world's fastest-growing HIV epidemic* (p. 291). The World Bank. https://doi.org/10.1596/978-1-4648-1523-2_ch13

Kerr, C. C., Stuart, R. M., Mistry, D., Abeysuriya, R. G., Cohen, J. A., George, L., Jastrzebski, M., Famulare, M., Wenger, E., & Klein, D. J. (2022). Python vs. the pandemic: A case study in high-stakes software development. *Proceedings of the 21st Python in Science Conference (SciPy 2022)*. https://doi.org/10.25080/majora-212e5952-00e

Kerr, C. C., Stuart, R. M., Mistry, D., Abeysuriya, R. G., Rosenfeld, K., Hart, G. R., Núñez, R. C., Cohen, J. A., Selvaraj, P., Hagedorn, B., & others. (2021). Covasim: An agent-based model of COVID-19 dynamics and interventions. *PLOS Computational Biology*, *17*(7), e1009149. https://doi.org/10.1371/journal.pcbi.1009149

Lorica, B. (2017). *Why AI and machine learning researchers are beginning to embrace PyTorch*. oreilly.com/radar/podcast/why-ai-and-machine-learning-researchers-are-beginning-to-embrace-pytorch

Mistry, D., Kerr, C. C., Abeysuriya, R. G., Wu, M., Fisher, M., Thompson, A., Skrip, L., Cohen, J. A., & Klein, D. J. (in preparation). *SynthPops: A generative model of human contact networks.*

Nature Editorial Board. (2018). Easing the burden of code review. *Nature Methods*, *15*(9), 641. https://doi.org/10.1038/s41592-018-0137-5

O'Brien, M. L., Valente, A., Chabot-Couture, G., Proctor, J., Klein, D., Kerr, C., & Zimmermann, M. (2022). FPSim: An agent-based model of family planning for informed policy decision-making. *PAA 2022 Annual Meeting*.

Pearson, R., Killedar, M., Petravic, J., Kakietek, J. J., Scott, N., Grantham, K. L., Stuart, R. M., Kedziora, D. J., Kerr, C. C., Skordis-Worrall, J., & others. (2018). Optima nutrition: An allocative efficiency tool to reduce childhood stunting by better targeting of nutrition-related interventions. *BMC Public Health*, *18*(1), 1–12. https://doi.org/10.1186/s12889-018-5294-z

Perkel, J. M. (2020). Challenge to scientists: Does your ten-year-old code still run? *Nature*, *584*(7822), 656–659. https://doi.org/10.1038/d41586-020-02462-7

Robitaille, T. P., Tollerud, E. J., Greenfield, P., Droettboom, M., Bray, E., Aldcroft, T., Davis, M., Ginsburg, A., Price-Whelan, A. M., Kerzendorf, W. E., & others. (2013). Astropy: A community Python package for astronomy. *Astronomy & Astrophysics*, *558*, A33.

Rocklin, M. (2015). Dask: Parallel computation with blocked algorithms and task scheduling. In K. Huff & J. Bergstra (Eds.), *Proceedings of the 14th Python in science conference (SciPy 2015)* (pp. 130–136). https://doi.org/10.25080/Majora-7b98e3ed-013

Sullivan, C., & Kaszynski, A. (2019). PyVista: 3D plotting and mesh analysis through a streamlined interface for the visualization toolkit (VTK). *Journal of Open Source Software*, *4*(37), 1450. https://doi.org/10.21105/joss.01450

The Atomica Team. (2020). Atomica: A simulation engine for compartmental models. In *GitHub repository*. GitHub. https://github.com/atomicateam/atomica

Waskom, M. L. (2021). Seaborn: Statistical data visualization. *Journal of Open Source Software*, *6*(60), 3021. https://doi.org/10.21105/joss.03021