Teaching numerical methods with IPython notebooks and inquiry-based learning

David I. Ketcheson*†

http://www.youtube.com/watch?v=OaP6LiZuaFM

Abstract—A course in numerical methods should teach both the mathematical theory of numerical analysis and the craft of implementing numerical algorithms. The IPython notebook provides a single medium in which mathematics, explanations, executable code, and visualizations can be combined, and with which the student can interact in order to learn both the theory and the craft of numerical methods. The use of notebooks also lends itself naturally to inquiry-based learning methods. I discuss the motivation and practice of teaching a course based on the use of IPython notebooks and inquiry-based learning, including some specific practical aspects. The discussion is based on my experience teaching a Masters-level course in numerical analysis at King Abdullah University of Science and Technology (KAUST), but is intended to be useful for those who teach at other levels or in industry.

Index Terms—IPython, IPython notebook, teaching, numerical methods, inquiry-based learning

Teaching numerical methods

Numerical analysis is the study of computational algorithms for solving mathematical models. It is used especially to refer to numerical methods for approximating the solution of continuous problems, such as those involving differential or algebraic equations. Solving such problems correctly and efficiently with available computational resources requires both a solid theoretical foundation and the ability to write and evaluate substantial computer programs.

Any course in numerical methods should enable students to:

1. **Understand** numerical algorithms and related mathematical concepts like complexity, stability, and convergence
2. **Select** an appropriate method for a given problem
3. **Implement** the selected numerical algorithm
4. **Test** and debug the numerical implementation

In other words, students should develop all the skills necessary to go from a mathematical model to reliably-computed solutions. These skills will allow them to select and use existing numerical software responsibly and efficiently, and to create or extend such software when necessary. Usually, only the first of the objectives above is actually mentioned in the course syllabus, and in some courses it is the only one taught. But the other three objectives are likely to be of just as much value to students in their careers. The last two skills are practical, and teaching them properly is in some ways akin to teaching a craft. Crafts are not generally taught through lectures and textbooks; rather, one learns a craft by *doing*.

Over the past few years, I have shifted the emphasis of my own numerical courses in favor of addressing all four of the objectives above. In doing so, I have drawn on ideas from inquiry-based learning and used both Sage worksheets and IPython notebooks as an instructional medium. I've found this approach to be very rewarding, and students have told me (often a year or more after completing the course) that the hands-on mode of learning was particularly helpful to them.

The notebooks used in my course for the past two years are available online:

- 2013 course: https://github.com/ketch/finite-difference-course
- 2013 course: https://github.com/ketch/AMCS252

Please note that these materials are not nearly as polished as a typical course textbook, and some of them are not self-contained (they may rely strongly on my unpublished course notes). Nevertheless, I've made them publicly available in case others find them useful. For more context, you may find it helpful to examine the course syllabus. You can also examine the notebooks for my short course on hyperbolic PDEs, which are more self-contained.

Inquiry-based learning

> *The best way to learn is to do; the worst way to teach is to talk.* --P. R. Halmos [Hal75]

Many great teachers of mathematics (most famously, R.L. Moore) have argued against lecture-style courses, in favor of an approach in which the students take more responsibility and there is more in-class interaction. The many related approaches that fit this description have come to be called *inquiry-based learning* (IBL). In an inquiry-based mathematics course, students are expected to find the proofs for themselves -- with limited assistance from the instructor. For a very recent review of what IBL is and the evidence for its effectiveness, see [Ern14a], [Ern14b] and references therein. If an active,

* *Corresponding author: david.ketcheson@kaust.edu.sa*
† *King Abdullah University of Science and Technology*

inquiry-based approach is appropriate for the teaching of theoretical mathematics, then it seems even more appropriate for teaching the practical craft of computational mathematics.

A related notion is that of the *flipped classroom*. It refers to a teaching approach in which students read and listen to recorded lectures outside of class. Class time is then used not for lectures but for more active, inquiry-based learning through things like discussions, exercises, and quizzes.

The value of practice in computational mathematics

Too often, implementation, testing, and debugging are viewed by computational mathematicians as mundane tasks that anyone should be able to pick up without instruction. In most courses, some programming is required in order to complete the homework assignments. But usually no class time is spent on programming, so students learn it on their own -- often poorly and with much difficulty, due to the lack of instruction. This evident disdain and lack of training seem to mutually reinforce one another. I believe that implementation, testing, and debugging are essential skills for anyone who uses or develops numerical methods, and they should be also taught in our courses.

In some situations, a lack of practical skills has the same effect as a lack of mathematical understanding. Students who cannot meaningfully test their code are like students who cannot read proofs: they have no way to know if the claimed results are correct or not. Students who cannot debug their code will never know whether the solution blows up due to an instability or due to an error in the code.

In many cases, it seems fair to say that the skills required to implement state-of-the-art numerical algorithms consists of equal parts of mathematical sophistication and software engineering. In some areas, the development of correct, modular, portable implementations of proposed algorithms is as significant a challenge as the development of the algorithms themselves. Furthermore, there are signs that numerical analysts need to move beyond traditional flop-counting complexity analysis and incorporate more intricate knowledge of modern computer hardware in order to design efficient algorithms for that hardware. As algorithms become increasingly adapted to hardware, the need for implementation skills will only increase.

Perhaps the most important reason for teaching implementation, testing, and debugging is that these skills can and should be used to reinforce the theory. The student who learns about numerical instability by reading in a textbook will forget it after the exam. The student who discovers numerical instability by implementing an apparently correct (but actually unstable) algorithm by himself and subsequently learns how to implement a stable algorithm will remember and understand it much better. Similarly, implementing an explicit solver for a stiff problem and then seeing the speedup obtained with an appropriate implicit solver makes a lasting impression.

It should be noted that many universities have courses (often called "laboratory" courses) that do focus on the implementation or application of numerical algorithms, generally using MATLAB, Mathematica, or Maple. Such courses may end up being those of most lasting usefulness to many students. The tools and techniques discussed in this article could very aptly be applied therein. Unfortunately, these courses are sometimes for less credit than a normal university course, with an attendant reduction in the amount of material that can be covered.

Hopefully the reader is convinced that there is some value in using the classroom to teach students more than just the theory of numerical methods. In the rest of this paper, I advocate the use of inquiry-based learning and IPython notebooks in full-credit university courses on numerical analysis or numerical methods. As we will see, the use of IPython notebooks and the teaching of the craft of numerical methods in general lends itself naturally to inquiry-based learning. While most of the paper is devoted to the advantages of this approach, there are some significant disadvantages, which I describe in the *Drawbacks* section near the end.

Teaching with the IPython notebook

Python and IPython

The teacher of numerical methods has several choices of programming language. These can broadly be categorized as

- specialized high-level interpreted languages (MATLAB, Mathematica, Maple)
- general-purpose compiled languages (C, C++, Fortran).

High-level languages, especially MATLAB, are used widely in numerical courses and have several advantages. Namely, the syntax is very similar to the mathematical formulas themselves, the learning curve is short, and debugging is relatively simple. The main drawback is that such languages do not provide the necessary performance to solve large research or industrial problems. This may be a handicap for students if they never gain experience with compiled languages.

Python strikes a middle ground between these options. It is a high-level language with intuitive syntax and high-level libraries for everything needed in a course on numerical methods. At the same time, it is a general-purpose language. Although (like MATLAB) it can be relatively slow [VdP14], Python makes it relatively easy to develop fast code by using tools such as Cython or f2py. For the kinds of exercises used in most courses, pure Python code is sufficiently fast. In recent years, with the advent of tools like numpy and matplotlib, Python has increasingly been adopted as a language of instruction for numerical courses.

IPython [Per07] is a tool for using Python interactively. One of its most useful components is the IPython notebook: a document format containing text, code, images, and more, that can be written, viewed, and executed in a web browser.

The IPython notebook as a textbook medium

Many print and electronic textbooks for numerical methods include code, either printed on the page or available online (or both). Some of my favorite examples are [Tre00] and [LeV07]. Such books have become more common, as the importance of exposing students to the craft of numerical methods -- and the value of experimentation in learning the theory -- has become more recognized. The IPython notebook can be viewed as the

next step in this evolution. As demonstrated in Figure 1, it combines in a single document

- Mathematics (using LaTeX)
- Text (using Markdown)
- Code (in Python or other languages)
- Figures and animations

Mathematica, Maple, and (more recently) Sage have document formats with similar capabilities. The Sage worksheet is very similar to the IPython notebook (indeed, the two projects have strongly influenced each other), so most of what I will say about the IPython notebook applies also to the Sage worksheet.

The notebook has some important advantages over Mathematica and Maple documents:

- It can be viewed, edited, and executed using only **free** software;
- It allows the use of multiple programming languages;
- It can be collaboratively edited by multiple users at the same time (currently only on SageMathCloud);
- It is open source, so users can modify and extend it.

The second point above was especially important when I decided to switch from using Sage worksheets to IPython notebooks. Because both are stored as text, I was able to write a simple script to convert them. If I had been using a proprietary binary format, I would have lost a lot of time re-writing my materials in a new format.

Perhaps the most important advantage of the notebook is the community in which it has developed -- a community in which openness and collaboration are the norm. Because of this, those who develop teaching and research materials with IPython notebooks often make them freely available under permissive licenses; see for example Lorena Barba's AeroPython course [Bar14] or this huge list of books, tutorials, and lessons. Due to this culture, the volume and quality of available materials for teaching with the notebook is quickly surpassing what is available in proprietary formats. It should be mentioned that the notebook is also being used as a medium for publishing research, both as open notebook science and full articles.

Mechanics of an interactive, notebook-based course

I have successfully used IPython notebooks as a medium of instruction in both

- semester-length university courses; and
- short 1-3 day tutorials

I will focus on the mechanics of teaching a university course, but much of what I will say applies also to short tutorials. The notebook is especially advantageous in the context of a tutorial because one does not usually have the luxury of ensuring that students have a textbook. The notebooks for the course can comprise a complete, self-contained curriculum.

Typically I have used a partially-flipped approach, in which half of the class sessions are traditional lectures and the other half are *lab sessions* in which the students spend most of the time programming and discussing their programs. Others have used IPython notebooks with a fully-flipped approach; see for example [Bar13].

Getting students started with the notebook

One historical disadvantage of using Python for a course was the difficulty of ensuring that all students had properly installed the required packages. Indeed, when I began teaching with Python 5 years ago, this was still a major hassle even for a course with twenty students. If just a few percent of the students have installation problems, it can create an overwhelming amount of work for the instructor.

This situation has improved dramatically and is no longer a significant issue. I have successfully used two strategies: local installation and cloud platforms.

Local installation

It can be useful for students to have a local installation of all the software on their own computer or a laboratory machine. The simplest way to achieve this is to install either Anaconda or Canopy. Both are free and include Python, IPython, and all of the other Python packages likely to be used in any scientific course. Both can easily be installed on Linux, Mac, and Windows systems.

Cloud platforms

In order to avoid potential installation issues altogether, or as a secondary option, notebooks can be run using only cloud services. Two free services exist for running IPython notebooks:

- Sage Math Cloud
- Wakari

Both services are relatively new and are developing rapidly. Both include all relevant Python packages by default. I have used both of them successfully, though I have more experience with Sage Math Cloud (SMC). Each SMC project is a complete sandboxed Unix environment, so it is possible for the user to install additional software if necessary. On SMC, it is even possible for multiple users to collaboratively edit notebooks at the same time.

Teaching Python

Since students of numerical methods do not usually have much prior programming experience, and what they have is usually in another language, it is important to give students a solid foundation in Python at the beginning of the course. In the graduate courses I teach, I find that most students have previously programmed in MATLAB and are easily able to adapt to the similar syntax of Numpy. However, some aspects of Python syntax are much less intuitive. Fortunately, a number of excellent Python tutorials geared toward scientific users are available. I find that a 1-2 hour laboratory session at the beginning of the course is sufficient to acquaint students with the necessary basics; further details can be introduced as needed later in the course. Students should be strongly encouraged to work together in developing their programming skills. For examples of such an introduction, see this notebook or this one.

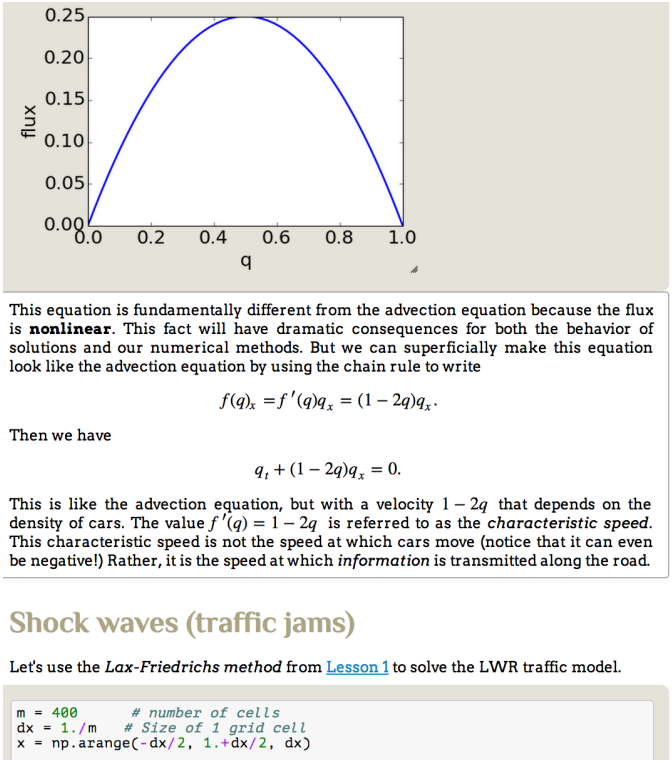

Fig. 1: *An excerpt from Notebook 2 of HyperPython, showing the use of text, mathematics, code, and a code-generated plot in the IPython notebook.*

Lab sessions

At the beginning of each lab session, the students open a new notebook that contains some explanations and exercises. Generally they have already been introduced to the algorithm in question, and the notebook simply provides a short review. Early in the course, most of the code is provided to the students already; the exercises consist mainly of extending or modifying the provided code. As the course progresses and students develop their programming skills, they are eventually asked to implement some algorithms or subroutines from scratch (or by starting from codes they have written previously). Furthermore, the specificity of the instructions is gradually decreased as students develop the ability to fill in the intermediate steps.

It is essential that students arrive to the lab session already prepared, through completing assigned readings or recordings. This doesn't mean that they already know everything contained in the notebook for that day's session; on the contrary, class time should be an opportunity for guided discovery. I have found it very useful to administer a quiz at the beginning of class to provide extra motivation. Quizzes can also be administered just before students begin a programming exercise, in order to check that they have a good plan for completing it, or just after, to see how successful they were.

The main advantage of having students program in class (rather than at home on their own) is that they can talk to the instructor and to other students as they go. Most students are extremely reluctant to do this at first, and it is helpful to require them to explain to one another what their code does (or is intended to do). This can be accomplished by having them program in pairs (alternating, with one programming while the other makes comments and suggestions). Another option is to have them compare and discuss their code after completing an exercise.

When assisting students during the lab sessions, it is important not to give too much help. When the code fails, don't immediately explain what is wrong or how to fix it. Ask questions. Help them learn to effectively read a traceback and diagnose their code. Let them struggle a bit to figure out why the solution blows up. Even if they seem to grasp things immediately, it's worthwhile to discuss their code and help them develop good programming style.

Typically, in an 80-minute class session the students spend 50-60 minutes working (thinking and programming) and 20-30 minutes listening to explanations, proposing ideas, discussing their solutions, and taking quizzes. During the working time, the instructor should assess and help students one-on-one as needed.

Designing effective notebooks

Prescribing how to structure the notebooks themselves is like stipulating the style of a textbook or lecture notes. Each instructor will have his or her own preferences. So I will share some principles I have found to be effective.

Make sure that they type code from the start

This goes without saying, but it's especially important early in the course. It's possible to write notebooks where all the code involved is already completely provided. That's fine if students

only need to understand the output of the code, but not if they need to understand the code itself (which they generally do). The plain truth is that nobody reads code provided to them unless they have to, and when they do they understand only a fraction of it. Typing code, like writing equations, dramatically increases the degree to which we internalize it. At the very beginning of the course, it may be helpful to have students work in an IPython session and type code from a notebook into the IPython prompt.

Help students to discover concepts on their own

This is the central principle of inquiry-based learning. Students are more motivated, gain more understanding, and retain knowledge better when they discover things through their own effort and after mentally engaging on a deep level. In a numerical methods course, the traditional approach is to lecture about instability or inaccuracy, perhaps showing an example of a method that behaves poorly. In the flipped approach, you can instead allow the students to implement and experiment in class with naive algorithms that seem reasonable but may be inaccurate or unstable. Have them discuss what they observe and what might be responsible for it. Ask them how they think the method might be improved.

Teaching is tricky because you want the students to come up to date on topics which have taken perhaps decades to develop. But they gain the knowledge quickly without the discipline of having struggled with issues. By letting them struggle and discover you simulate the same circumstances which produced the knowledge in the first place.

Tailor the difficulty to the students' level

Students will lose interest or become frustrated if they are not challenged or they find the first exercise insurmountable. It can be difficult to accommodate the varying levels of experience and skill presented by students in a course. For students who struggle with programming, peer interaction in class is extremely helpful. For students who advance quickly, the instructor can provide additional, optional, more challenging questions. For instance, in my HyperPython short course, some notebooks contain challenging "extra credit" questions that only the more advanced students attempt.

Gradually build up complexity

In mathematics, one learns to reason about highly abstract objects by building up intuition with one layer of abstraction at a time. Numerical algorithms should be developed and understood in the same way, with the building blocks first coded and then encapsulated as subroutines for later use. Let's consider the multigrid algorithm as an example. Multigrid is a method for solving systems of linear equations that arise in modeling things like the distribution of heat in a solid. The basic building block of multigrid is some way of smoothing the solution; the key idea is to apply that smoother successively on copmutational grids with different levels of resolution.

I have students code things in the following sequence:

1. Jacobi's method (a smoother that doesn't quite work)
2. Under-relaxed Jacobi (a smoother that does work for high frequencies)
3. A two-grid method (applying the smoother on two different grids in succession)
4. The V-cycle (applying the smoother on a sequence of grid)
5. Full multigrid (performing a sequence of V-cycles with successively finer grids)

In each step, the code from the previous step becomes a subroutine. In addition to being an aid to learning, this approach teaches students how to design programs well. The multigrid notebook from my course can be found (with some exercises completed) here.

Use animations liberally

Solutions of time-dependent problems are naturally depicted as animations. Printed texts must restrict themselves to waterfall plots or snapshots, but electronic media can show solutions in the natural way. Students learn more -- and have more fun -- when they can visualize the results of their work in this way. I have used Jake Vanderplas' JSAnimation package [VdP13] to easily create such animations. The latest release of IPython (version 2.1.0) natively includes interactive widgets that can be used to animate simulation results.

Time-dependent solutions are not the only things you can animate. For iterative solvers, how does the solution change after each algorithmic iteration? What effect does a given parameter have on the results? Such questions can be answered most effectively through the use of animation. One simple example of teaching a concept with such an animation, shown in Figure 2, can be found in this notebook on aliasing.

Drawbacks

The approach proposed here differs dramatically from a traditional course in numerical methods. I have tried to highlight the advantages of this approach, but of course there are also some potential disadvantages.

Material covered

The most substantial drawback I have found relates to the course coverage. Programming even simple algorithms takes a lot of time, especially for students. Therefore, the amount of material that can be covered in a semester-length course on numerical methods is substantially less under the interactive or flipped model. This is true for inquiry-based learning techniques in general, but even more so for courses that involve programming. I believe that it is better to show less material and have it fully absorbed and loved than to quickly dispense knowledge that falls on deaf ears.

Scalability

While some people do advocate IBL even for larger classes, I have found that this approach works best if there are no more than twenty students in the course. With more students, it can be difficult to fit everyone in a computer lab and nearly impossible for the instructor to have meaningful interaction with individual students.

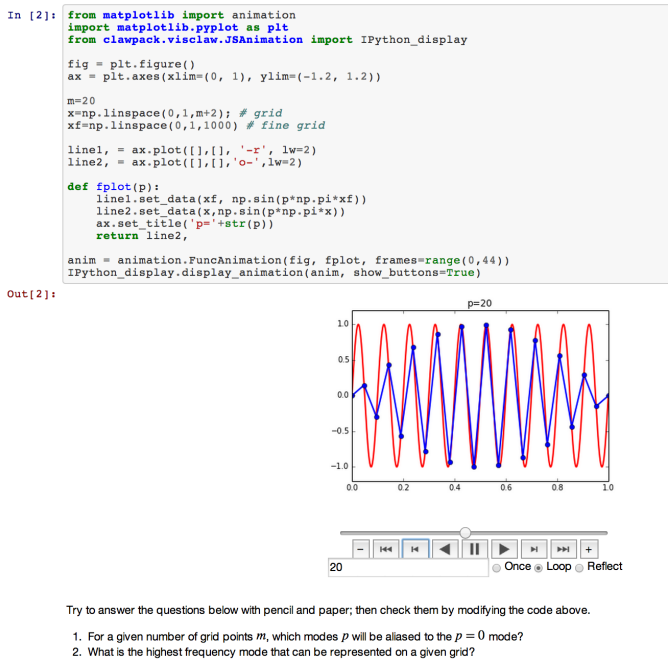

Fig. 2: *A short notebook on grid aliasing, including code, animation, and exercises.*

Nonlinear notebook execution

Code cells in the notebook can be executed (and re-executed) in any order, any number of times. This can lead to different results than just executing all the cells in order, which can be confusing to students. I haven't found this to be a major problem, but students should be aware of it.

Opening notebooks

Perhaps the biggest inconvenience of the notebook is that opening one is not as simple as clicking on the file. Instead, one must open a terminal, go to the appropriate directory, and launch the ipython notebook. This is fine for users who are used to UNIX, but is non-intuitive for some students. With IPython 2.0, one can also launch the notebook from any higher-level directory and then navigate to a notebook file within the browser.

It's worth noting that on SMC one can simply click on a notebook file to open it.

Lengthy programs and code reuse

Programming in the browser means you don't have all the niceties of your favorite text editor. This is no big deal for small bits of code, but can impede development for larger programs. I also worry that using the notebook too much may keep students from learning to use a good text editor. Finally, running long programs from the browser is problematic since you can't detach the process.

Usually, Python programs for a numerical methods course can be broken up into fairly short functions that each fit on a single screen and run in a reasonable amount of time.

Placing code in notebooks also makes it harder to reuse code, since functions from one notebook cannot be used in another with copying. Furthermore, for the reasons already given, the notebook is poorly suited to development of library code. Exclusive use of the notebook for coding may thus encourage poor software development practices. This can be partially countered by teaching students to place reusable functions in files and then importing them in the notebook.

Interactive plotting

In my teaching notebooks, I use Python's most popular plotting package, Matplotlib [Hun07]. It's an extremely useful package, whose interface is immediately familiar to MATLAB users, but it has a major drawback when used in the IPython notebook. Specifically, plots that appear inline in the notebook are not interactive -- for instance, they cannot be zoomed or panned. There are a number of efforts to bring interactive plots to the notebook (such as Bokeh and Plotly) and I expect this weakness will soon be an area of strength for the IPython ecosystem. I plan to incorporate one of these tools for plotting in the next course that I teach.

More resources

Many people are advocating and using the IPython notebook as a teaching tool, for many subjects. For instance, see:

- Teaching with the IPython Notebook by Matt Davis
- How IPython Notebook and Github have changed the way I teach Python by Eric Matthes
- Using the IPython Notebook as a Teaching Tool by Greg Wilson
- Teaching with ipython notebooks -- a progress report by C. Titus Brown

To find course actual course materials (in many subjects!), the best place to start is this curated list: A gallery of interesting IPython Notebooks.

Acknowledgments

I am grateful to Lorena Barba for helpful discussions (both online and offline) of some of the ideas presented here. I thank Nathaniel Collier, David Folch, and Pieter Holtzhausen for their comments that significantly improved this paper. This work was supported by the King Abdullah University of Science and Technology (KAUST).

REFERENCES

[LeV07] R. J. LeVeque. *Finite Difference Methods for Ordinary and Partial Differential Equations*, Society for Industrial and Applied Mathematics, 2007.

[Tre00] L. N. Trefethen. *Spectral Methods in MATLAB*, Society for Industrial and Applied Mathematics, 2000.

[Bar14] L. A. Barba, O. Mesnard. *AeroPython*, 10.6084/m9.figshare.1004727. Code repository, Set of 11 lessons in classical Aerodynamics on IPython Notebooks. April 2014.

[Bar13] L. A. Barba. *CFD Python: 12 steps to Navier-Stokes*, http://lorenabarba.com/blog/cfd-python-12-steps-to-navier-stokes/, 2013.

[Hal75] P. R. Halmos, E. E. Moise, and G. Piranian. *The problem of learning how to teach*, The American Mathematical Monthly, 82(5):466--476, 1975.

[Ern14a] D. Ernst. *What the heck is IBL?*, Math Ed Matters blog, http://maamathedmatters.blogspot.com/2013/05/what-heck-is-ibl.html, May 2014

[Ern14b] D. Ernst. *What's So Good about IBL Anyway?*, Math Ed Matters blog, http://maamathedmatters.blogspot.com/2014/01/whats-so-good-about-ibl-anyway.html, May 2014.

[VdP14] J. VanderPlas. *Why Python is Slow: Looking Under the Hood*, Pythonic Perambulations blog, http://jakevdp.github.io/blog/2014/05/09/why-python-is-slow/, May 2014.

[VdP13] J. VanderPlas. *JSAnimation*, https://github.com/jakevdp/JSAnimation, 2013.

[Per07] F. Pérez, B. E. Granger. *IPython: A System for Interactive Scientific Computing*, Computing in Science and Engineering, 9(3):21-29, 2007. http://ipython.org/

[Hun07] J. D. Hunter. *Matplotlib: A 2D graphics environment*, Computing in Science and Engineering, 9(3):90--95, 2007. http://matplotlib.org/