D.2.1 Educational plan: Cutting-edge scientific computing in the curriculum

The current computing landscape and the need for improvement. Numerical computing has been a revolutionary force in science and engineering over the last fifty to sixty years. It allows for simulation, exploration, and rapid prototyping of systems that are too complex for closed-form mathematical analysis or too expensive or time-consuming to iteratively construct, test, and revise in the laboratory. But numerical simulation is not easy. It requires in-depth knowledge of the relevant engineering or scientific domain, the applied mathematics of numerical approximation methods, and the programming languages and practical methodology of computer science (CS). CS is the newest of these areas, the one undergoing the most rapid current development, and the one that receives least attention in a typical undergraduate or graduate science/engineering education.

A typical science/math/engineering student might take one course in CS, likely programming in C, C++, or Java, usually with an emphasis on CS fundamentals, and notably not with emphasis on science, engineering, or numerical applications. Moreover, these languages are not designed with numerics in mind, and actually have substantial barriers to the implementation of numerical algorithms (such as lack of fundamental vector and matrix types, and complex, syntax-laden interfaces to the libraries that implement these). Further, for compiled languages like C, C++, and Fortran, the processes for software development, compilation, linking, debugging, and optimization can be extraordinarily complex.

Thus for undergraduates emerging from such CS courses and beginning to undertake numerical work for science and engineering, the typical response is to drop what was learned in the CS course and to pick up a more narrowly focused but easier-to-use language like Matlab. In fact, it is not uncommon for engineering and math departments to develop their own Matlab courses to meet this need, as an alternative to having a CS requirement in their majors.

Now, Matlab is a wonderful thing. The combination of an interactive command-line interface, quality graphics, a simple, powerful syntax for vectors and matrices, and a well-designed interface to well-tested linear algebra libraries, makes it a very productive system for interactive and rapid-prototyped numerical computation. And due to these features, Matlab has had an hugely beneficial role in education, research, and industry since its introduction in the late 1980s. But Matlab is an imperfect solution to the complete spectrum of needs of scientific computation. Specifically: (1) It was not designed for general-purpose computing, so the scope of what can be accomplished in it is limited, it becomes increasingly awkward when pushed towards that limit for large, complex problems, and it provides students with a only limited vision of what computing can be. (2) User-level Matlab code is interpreted and thus more than an order of magnitude slower than the corresponding compiled C or Fortran code, thus barring its use for large, high-performance computation (HPC). (3) Matlab is a licensed commercial product, which is fine for industry but problematic in education, in that private ownership of the language syntax and infrastructure precludes academic innovation and locks knowledge to private enterprise.

Due to these issues, primarily (1) and (2), the current typical practice in scientific computing is to use an interactive, interpreted, and graphics-capable language like Matlab or Python for exploration, prototyping, and small-scale simulation, to switch to an efficient, compiled, but low-level language like C, C++, or Fortran for high-performance code and production runs, and to return to the interactive graphical environment for data analysis. This practice is obviously inefficient, as it involves multiple languages and computational paradigms, and transfer of data and algorithms between them. Also, the complexity of the software development process for low-level languages prevents all but the most determined scientists and engineers from developing professional-level

skills in HPC. These language-centered impediments to research and education in scientific computing have been a thorn in the side of the PI for more than twenty years.

The Julia programming language for numerical computing. Recently, however, a group of visionary applied mathematicians and computer scientists at MIT have developed a modern language for numerical computing that promises to solve these problems, leapfrog current numerical languages, and open up a new era in scientific computing. The Julia programming language is a modern, dynamic, high-level language aimed squarely at scientific computation. It has the dynamic, high-level, and general-purpose scope of Python, but the linear algebra syntax, libraries, and graphics of Matlab. Its fundamental data types include integers, rationals, floating point, and complex numbers of fixed size and arbitrary precision, and vectors, matrices, and arbitrarily dimensioned arrays of each numeric type, as well as general-purpose CS types such as tuples, dictionaries, Unicode strings, hash tables, etc. It has an interactive command-line interface, but its just-in-time compilation system results in execution speed within a factor of two of C. Even further, it has a level of computer-science sophistication that leapfrogs by decades the capabilities of C, C++, and Fortran. For example, Julia is *homoiconic* –the language is able to represent and manipulate expressions in its own syntax as data structures within language itself. As a result, for example, anonymous functions in Julia that are defined at run time can be transformed at run time to optimized assembly language and thus run as fast as compiled C code. Further, Julia programs can generate algorithms at run time, serialize them, send them across clusters for execution on remote data, and return the results as structured objects –far beyond the capabilities of MPI (the current standard parallel Message-Passing Interface library). In fact, the high-level, dynamic nature of Julia's parallel programming paradigm is perhaps its most far-reaching feature. [3–5, 34, 36, 49].

Julia notebooks for interactive, online course content. Additionally, the open-source, dynamic character of the Julia language has enabled the development of an extremely powerful *Julia notebook* system which forms an ideal framework for developing and delivering interactive, online, mathematical course content. A Julia notebook mixes editable and in-browser-executable Julia code, the code's live graphical output, and mathematical annotation in a simple LaTeX-aware markup language. As such, an educator can produce and place on the Internet a Julia notebook that discusses a scientific computation in high-quality mathematical notation, provides Julia code that implements the algorithm, and shows the graphics that visualize the results. A learner on a distant computer can then open the notebook in a browser, click on the code, modify it by editing, press "return" to execute the revised code, and view the revised graphical results, all within his or her local browser. This system is ideal for presenting numerically-oriented lecture notes, examples, homeworks, and labs, and an automatic homework feedback/grading system is available. *This is a game-changing improvement on commercial course content management systems,* which are typically very poor for even representing mathematical symbols, let alone executing code, generating graphics, or assessing student work. Moreover, the open-source framework of Julia facilitates and encourages free distribution of course content developed as Julia notebooks.

Design of Julia began in 2009 at MIT, and the first public release occurred in 2012. The language is still somewhat young (it is currently at release 0.40), but it is sufficiently solid in syntax and libraries for teaching and research (the Julia website lists courses taught at seventeen universities, including MIT, Brown, Stanford, and Cornell). The development of the Julia language is supported by a Julia-focused start-up company, Julia Computing LLC, and the NumFocus Foundation, with funding from Intel, Microsoft, J.P. Morgan, and Continuum Analytics.

In summary, Julia is a vastly superior scientific programming language with the ease of use and numerical libraries of Matlab, the general-purpose power of Python, the speed of compiled C, a

revolutionary new approach to parallel computing, and an ideal system for interactive online course content.

D.2.2 Graduate and undergraduate curriculum enhancements

The PI will transform the computational aspects of the undergraduate and graduate applied math curriculum at UNH by teaching Julia within these courses and developing course content as interactive Julia notebooks. The benefit to students will be (1) the acquisition of single, powerful modern language that meets all their computing needs, from interactive graphical exploration to high-performance parallel computation, and (2) an increased level of interest and engagement with the course content, due to its presentation in interactive Julia notebooks.

Graduate level. The PI currently teaches two graduate courses: IAM 961 *Numerical Linear Algebra*, and IAM 950 *Spatiotemporal and Turbulence Dynamics*. Both these courses are highly computational, and in the past the PI taught them with Matlab and assumed basic Matlab familiarity on the part of the students. IAM 961 emphasizes matrix factorization algorithms, solution of $Ax = b$, the eigenvalue problem, Krylov subspace methods, and the relation between stability, conditioning, and accuracy of numerical computations in finite precision. IAM 950 analyzes the formation and dynamics of coherent structures in nonlinear PDEs ranging from Swift-Hohenberg to Navier-Stokes, and as such involves the numerical approximation methods for nonlinear PDEs. The PI will revise these courses, placing programming and the Julia language more at the forefront and delivering lecture notes and homework assignments as Julia notebooks. The benefit will be that students learn a cutting-edge computational paradigm that is as easy to learn and rapid to tinker with as Matlab, but is also appropriate for high-performance computation. The PI will leverage pre-existing Julia notebooks from MIT's equivalent courses.

Undergraduate level. The PI currently teaches two undergraduate courses: freshman-level Math 445 *Introduction to Mathematics Applications in Matlab*, and sophomore-level Math 527 *Differential Equations with Linear Algebra*. The PI will teach Math 445 as a Julia course in the spring of 2016. As before, students will benefit by training in a more future-oriented, powerful, flexible, and free numerical platform. But more importantly at this introductory stage, they will be more engaged, due to the ease with which they can directly tinker with the course materials that are presented as interactive Julia notebooks. The PI will continue to teach Math 527 most of *Differential Equations* via whiteboard with homeworks and lecture notes delivered via LaTeX-aware Wiki, since symbolic mathematics in Julia is still under development, but the linear algebra will be taught via Julia. The PI is scheduled to teach Math 645 *Applied Linear Algebra* and Math 753 *Introduction to Numerical Methods* in 2016-17; the teaching of Julia will be seamlessly incorporated into the content and delivery of both.

D.2.3 Outreach to high-school students via UNH Techventures

The PI will engage high-school students in scientific computation at the UNH Techventures summer camp. Two hands-on, week-long projects will help students understand the connections between real-world practical problems and their abstract mathematical representations and numerical solution on computers.

UNH Tech Camp is a summer STEM camp for girls and boys in the 7th through 10th grades hosted by the UNH College of Engineering and Physical Sciences, developed in and running since 2008. In each week the campers choose to focus on one of a variety of STEM related projects.