COMPUTATIONAL SCIENCE AND ENGINEERING SOFTWARE SUSTAINABILITY AND PRODUCTIVITY (CSESSP) CHALLENGES WORKSHOP REPORT

CSESSP

October 15–16, 2015

WASHINGTON, DC USA

CSESSP WORKSHOP REPORT GROUP

Michael A. Heroux (co-chair, Sandia National Laboratories)
Gabrielle Allen (co-chair, University of Illinois)

NITRD POINT OF CONTACT

Ernest Lucier
Email: lucier@nitrd.gov

SPONSORED BY

How to cite this document:

Heroux, A. M., Allen, G. (2016, September). Computational Science and Engineering Software Sustainability and Productivity (CSESSP) Challenges Workshop Report. Arlington, VA: Networking and Information Technology Research and Development (NITRD) Program. Retrieved from NITRD Website: https://www.nitrd.gov/PUBS/CSESSPWorkshopReport.pdf

Productive and Sustainable
Computational Science & Engineering Software Productivity and Sustainability (CSESSP) Challenges Workshop Report

Sponsored by
Networking and Information Technology Research and Development (NITRD) Program
with
U.S. Department of Energy,
U.S. National Science Foundation,
U.S. Department of Defense OSD OASD

October 15–16, 2015
Washington, DC USA

CSESSP Workshop Report Group
Michael A. Heroux (co-chair, Sandia National Laboratories)
Gabrielle Allen (co-chair, University of Illinois Urbana-Champaign)

NITRD Point of Contact
Ernie Lucier

Abstract

This report details the challenges and opportunities discussed at the NITRD sponsored multi-agency workshop on Computational Science and Engineering Software Productivity and Sustainability (CSESSP) Challenges, held in Washington, D.C. USA on October 15–16, 2015. The workshop brought together 85 attendees from all branches of the U.S. federal government, industry, academia, and U.S. and international research laboratories to discuss growing concerns over the sustainability of our Computational Science and Engineering (CSE) software foundation, and the productivity of scientists and engineers who develop and use this software. Discussions focused on characterization of the challenges, and opportunities for improved productivity and sustainability going forward. This report comprises workshop and subsequent discussions including a summary of key opportunities for the CSE community going forward.

Networking and Information Technology Research and Development (NITRD) Program

The Networking and Information Technology Research and Development (NITRD) Program is the Nation's primary source of federally funded work on advanced information technologies (IT) in computing, networking, and software. The Program is one of the oldest and largest of the formal Federal programs that engage multiple agencies in coordination activities. As required by the High-Performance Computing Act of 1991 (P.L. 102-194), the Next Generation Internet Research Act of 1998 (P.L. 105-305), and the America COMPETES (Creating Opportunities to Meaningfully Promote Excellence in Technology, Education, and Science) Act of 2007 (P.L. 110-69), the NITRD Program provides a framework and mechanisms for coordination among the Federal agencies that support advanced IT R&D and annually report IT research budgets in the NITRD crosscut of the President's Budget. Many other agencies with IT interests also participate in NITRD activities. Overall NITRD Program coordination is carried out by the Subcommittee on Networking and Information Technology Research and Development, under the aegis of the Committee on Technology (CoT) of the National Science and Technology Council (NSTC). The NITRD National Coordination Office (NITRD/NCO) provides technical, administrative, and logistical support for the activities of the NITRD Program. For further information about the NITRD Program, please see the NITRD website: www.nitrd.gov.

Software Productivity, Sustainability, and Quality (SPSQ) (Formerly SDP CG)

SPSQ is a NITRD Interagency Working Group (IWG) that coordinates R&D that spans both the science and the technology of software creation and sustainment (e.g., development methods and environments, verification and validation technologies, component technologies, languages, and tools) and software project management in diverse domains. The primary SPSQ IWG objective is to accelerate progress in the science and technology of software production to deliver orders of magnitude improvement in software defect rates and in the time and cost of creating and sustaining software to drive innovation to the strategic advantage of the United States and develop the next-generation software-intensive systems.

Copyright Information

This is a work of the U.S. Government and is in the public domain and may be freely distributed, copied, and translated; acknowledgment of publication by the National Coordination Office for Networking and Information Technology Research and Development is appreciated. Any translation should include a disclaimer that the accuracy of the translation is the responsibility of the translator and not the NITRD/NCO. It is requested that a copy of any translation be sent to the NITRD/NCO.

Publication of This Report

Electronic versions of NITRD documents are available at NITRD website: https://www.nitrd.gov.

Auspices Statement

This work has been supported by the National Science Foundation Award CCF - 1551592, managed by the University of Illinois Urbana-Champaign; by the Department of Energy, Office of Advanced Scientific Computing Research; and the Networking and Information Technology Research and Development Program.

Contents

Table of Contents i
Preface iv
Executive Summary v
Acknowledgments vii
Contributors viii

1 Introduction **1**
1.1 Dematerialization through digitization 1
1.2 Challenges, trends, and opportunities 1
1.3 Characterizing productivity and sustainability 2
1.4 Software productivity 3
1.5 Software sustainability 4
1.6 The shared productivity-sustainability concern for product quality 5

2 Improved CSE Software Sustainability and Developer Productivity: Opportunities and Incentives **5**
2.1 Opportunities 5
2.2 The CREATE Project: Emphasizing sustainability and productivity 8
2.3 Providing incentives: Funding agencies, publishers and employers 9

3 Role of Software Engineering Research **10**
3.1 Expanding software engineering practice 12
3.2 New software engineering research 13
3.3 Transition of research to practice in CSE software community 13
3.4 Collaboration opportunities 14
3.5 Specific needs of small user teams 15
3.6 Software engineering research roadmap 15
3.7 Productive and sustainable: Astrophysics community codes 16

4 Measuring Software Productivity and Sustainability **17**
4.1 Opportunities for investigation 18
4.2 Understanding the use of and experience with metrics in the software engineering (SWE) community 19
4.3 Exposing the current use and experience with metrics in the CSE community 20
4.4 CSE as a distinct software domain 20
4.5 Encouraging and increasing introspection into CSE software development . . 21
4.6 Measuring productivity and sustainability roadmap 22

5 New Approaches for Faster, More Affordable CSE Software **22**
5.1 Suggested research directions 23
5.1.1 Productivity through continued knowledge development, capture, and dissemination 24
5.1.2 Productivity through design capture, display, and revision 25
5.1.3 Productivity through design recovery and modernization 26

5.2 New approaches roadmap . . . 27

6 Economics of CSE Software Tools 29
6.1 Descriptive examples . . . 30
6.1.1 Addressing the current cultural economics of CSE tool development . . . 30
6.1.2 Establishing new academia-industry-government partnership models . . . 31
6.1.3 Economics of commercial software . . . 32
6.1.4 Shared public-private journey . . . 34
6.1.5 Identifying new incentives for retaining top software talent . . . 34
6.2 An urgent need for productive and sustainable tools . . . 35
6.3 Software tools roadmap . . . 35

7 Social Sciences Applied to CSE Software Systems 36
7.1 CSE software is an ecosystem . . . 38
7.2 CSE software development is a set of social communities . . . 39
7.2.1 Team dynamics and culture . . . 39
7.2.2 Individual and intra-team social skills . . . 39
7.2.3 Individual response to community culture . . . 40
7.2.4 Inter-team communities and practices . . . 40
7.2.5 The role of tools . . . 41
7.3 Social sciences roadmap . . . 41
7.4 Possible outcomes . . . 42

8 Workforce Needs for Sustainable Software for Sciences 43
8.1 Scale and scope of problem . . . 44
8.2 Gaps in current training processes . . . 45
8.3 Innovative and emerging solutions . . . 46
8.4 Role of universities, industry and funding agencies roadmap . . . 46

9 CSE Software in Industry & Manufacturing 47
9.1 Opportunities for CSE software to advance industry & manufacturing . . . 48
9.1.1 Financial impact . . . 48
9.1.2 Engines of productivity . . . 48
9.1.3 Instruments for insight . . . 50
9.1.4 Software ecosystem synergy . . . 50
9.1.5 Software scalability competitiveness . . . 50
9.1.6 A new paradigm for regulation . . . 51
9.1.7 Grand challenge: Scalable multiscale . . . 52
9.1.8 Stimulate workforce development . . . 52
9.2 Development and use challenges of CSE modeling & simulation (M&S) software by industry . . . 53
9.2.1 Software licensing terms . . . 53
9.3 CSE impact: Consumer products . . . 54

10 Summary and Conclusions 55

References **56**

Appendices **67**
White Papers 67
Workshop Participants 68
Workshop Agenda 72

Preface

The Computational Science & Engineering Software Sustainability and Productivity (CSESSP) Challenges workshop identified the unique issues around software productivity and sustainability faced by the NITRD computational science and engineering (CSE) communities by bringing together experts from academia, industry, government, and national laboratories. The CSESSP provided the international software engineering research community with a unique opportunity to develop, discuss, refine, and disseminate consequential new ideas about future investments in software sustainability and productivity research. The committee defined eight major discussion themes based on 39 position papers. This report summarizes the results of these discussions.

Executive Summary

This document contains a report from the Computational Science and Engineering Software Sustainability and Productivity (CSESSP) Challenges Workshop, held October 15–16, 2015 in Rockville, MD and sponsored by the Networking and Information Technology Research and Development (NITRD) Program, the US Department of Energy (DOE), the US Department of Defense (DOD) and the US National Science Foundation (NSF). Content comes from meeting discussions and subsequent exchanges between workshop participants. Highlights are summarized in this executive summary.

Computational Science and Engineering (CSE) is transforming scientific discovery and engineering design: Computational Science and Engineering (CSE)—using computer modeling & simulation of mathematical models applied to important problems in science and engineering—is firmly established as a legitimate discipline alongside theory, experimentation and (increasingly) data science for advancing human knowledge, understanding and technology by "substituting atoms with electrons." CSE plays a unique and indispensable role in our society today, growing in importance and impact with time.

The CSE software ecosystem and enterprise need transformation. At the same time, CSE software as the foundation for modeling & simulation activities is in urgent need of transformation. In order for us to take advantage of qualitative advances in software engineering in other domains, and maximize the impact of CSE, our software base will require significant investment in both modified and new code, and in the processes by which it is produced and sustained. A timely and significant investment in CSE software productivity and sustainability will have a magnified impact on science and engineering overall.

> CSE software as an enterprise has yet to emerge as a creative discipline in its own right. Both model complexity and hardware complexity are growing simultaneously, and they both make the other more difficult to manage. The time is upon us to address the growing challenge of software productivity, quality, and sustainability that imperils the whole endeavor of computation-enabled science and engineering.

CSE software productivity and sustainability improvements require fundamental research and development, metrics and new approaches. Software productivity and sustainability have improved dramatically in mainstream domains where user and developer communities are large and well funded. Many books, articles and community leaders in these domains have enabled steady quality improvements. While CSE software efforts can clearly benefit from this body of knowledge, our communities and problems are sufficiently different so as to often require stripping mainstream approaches to their essence and reconstructing them to meet our needs. Furthermore, our communities can contribute fundamental improvements to the overall body of knowledge since our efforts often push the limits of what software products can and must do. To make improvements we need some ability to measure productivity and sustainability changes, and define new approaches that result in measurable progress.

Focusing on CSE Software productivity and sustainability improvements can help address critical staffing shortages. Hardware, software and problem complexities are dramatically reducing the number of scientists and engineers who can effectively use CSE environments to address grand challenge problems. New models are needed to spur development of productive and sustainable tools that expand access to and usability of CSE capabilities. Improving CSE software developer productivity, and product usability, accessibility and sustainability will have a multiplicative impact on staffing challenges by enabling the use of CSE by more scientists and engineers, and increasing the productivity of each person. Therefore, even if staffing shortages continue we can experience the equivalent of increasing the number of skilled scientists and engineers who can use CSE.

CSE software efforts rely on communities and ecosystems. CSE software development is a social activity; the entire process from eliciting requirements to producing, integrating and sustaining a software product is typically a team activity involving dozens of people with diverse skills. Understanding and influencing group dynamics is key to improving software teams. Furthermore, CSE software productivity and sustainability improvements can be enhanced by fostering professional networks, and recognizing software contributions.

Education in CSE software productivity and sustainability is needed both as an academic discipline and a professional development service. The CSE software community is primarily composed of domain experts who understand the intended use of a CSE software product, but who are not as well prepared to design and develop the product. Traditionally, formal software education and training have had only limited attention from CSE community members. Presently it is becoming clearer that we need more attention paid to CSE software productivity and sustainability, both as an academic discipline and as a collection of accessible content delivered in creative ways to CSE software professionals who want to learn about the latest ideas in software efforts.

CSE software productivity and sustainability efforts need to address the research to production software transformation. CSE software has a tremendous impact on industry & manufacturing. Even so, leveraging research and laboratory CSE software in the industrial environment is extremely challenging. Many effective CSE software products never make it into the hands of the scientists and engineers who could benefit the most from their use, and could themselves provide innovative new uses to solve important problems. CSE software often starts as a small effort in a research environment, growing organically as a craft more than an engineering discipline. If we can effectively manage the research to production growth process we will dramatically increase the number of effective CSE products available to the broader industrial and manufacturing communities.

Any investment in CSE software productivity and sustainability will have a disproportionate positive impact on all of science and engineering. Science and engineering will benefit substantially by increasing the productivity and sustainability of CSE efforts. CSE has emerged as such an important element in the overall scientific and engineering endeavor that any substantial improvements in the quality of our software efforts will have multiple and large impacts on any endeavor where CSE plays a role.

Acknowledgments

There were many contributors to the process that lead to the composition of this report, and the CSESSP Challenges Workshop Report Group would like to express our gratitude to these individuals.

A large part of our information-gathering process was the workshop held in Washington, D.C. We would like to thank the workshop steering committee:

- Vivien Bonazzi, National Institutes of Health (NIH)/OD
- Steven Drager, Air Force Research Laboratory (AFRL)
- Sol Greenspan, National Science Foundation (NSF)
- Daniel S. Katz, National Science Foundation (NSF)
- Walid Keyrouz, National Institute of Standards and Technology (NIST)
- Dai Hyun Kim, Office of the Secretary of Defense (OSD)
- James Kirby, Naval Research Laboratory (NRL)
- T. Ndousse-Fetter, Office of Science, U.S. Department of Energy

and the Workshop Program Committee:

- Gabrielle Allen, University of Illinois Urbana-Champaign (co-chair)
- Michael A. Heroux, Sandia National Laboratories (co-chair)
- Jeff Carver, University of Alabama
- Tom Clune, National Aeronautics and Space Administration (NASA)
- Merle Giles, University of Illinois Urbana-Champaign
- Lois Curfman McInnes, Argonne National Laboratory
- Manish Parashar, Rutgers University
- Doug Post, Department of Defense (DOD)
- Roldan Pozo, National Institute of Standards and Technology (NIST)
- Ethan Coon, Los Alamos National Lab

and the staff at Oak Ridge Institute for Science and Education (ORISE) (Deneise Terry and Jody Crisp, in particular) and University of Illinois (Laura Owen) for their help with managing the logistics of the workshop.

Particular thanks are due to the breakout session chairs who led discussions at the workshop:

- Jack Dongarra, Jeffrey Vetter: Opportunities from Improved CSE SW Sustainability and Productivity
- Richard Arthur, CSE Software in Industry/Manufacturing
- Ray Idaszak, Economics of Software Tools
- Daniel S. Katz, Aleksandra Pawlik, Social Sciences Applied to CSE Software Systems
- Abani Patra, Workforce Development
- Anshu Dubey, Role of Software Engineering Research
- Lois Curfman McInnes, Measuring Software Productivity and Performance
- Sandy Landsberg, New Approaches for Faster, More Affordable CSE Software

We would finally like to thank the authors who submitted white papers as well as the workshop participants.

List of contributors

Lead Editors

- Michael A. Heroux (Sandia National Laboratories)
- Gabrielle Allen (University of Illinois Urbana-Champaign)

Sections[1]

- *Opportunities from Improved CSE Software Sustainability and Productivity*
  Lead: Jack Dongarra (University of Tennessee, Knoxville)
  Contributors: Michael A. Heroux (Sandia National Laboratories), Albert I. Reuther (MIT Lincoln Laboratory), Grady Campbell (domain-specific.com)
- *Role of Software Engineering Research*
  Lead: Anshu Dubey (Argonne National Laboratory)
  Contributors: Michael A. Heroux (Sandia National Laboratories), Daniel Ibanez (Rensselaer Polytechnic Institute), Grady Campbell (domain-specific.com), Vijay Mahadevan (Argonne National Laboratory), Ethan Coon (Los Alamos National Laboratory), Karl Rupp (TUW)
- *Measuring Software Productivity and Performance*
  Lead: David E. Bernholdt (ORNL)
  Contributors: Roscoe Bartlett (ORNL → SNL), Daniel S. Katz (NSF → University of Illinois Urbana-Champaign), Albert I. Reuther (MIT Lincoln Laboratory)
- *New Approaches for Faster, More Affordable CSE Software*
  Lead: Thomas Clune (NASA Global Modeling and Assimilation Office)
  Contributors: Albert I. Reuther (MIT Lincoln Laboratory), Grady Campbell (domain-specific.com)
- *Economics of Software Tools*
  Lead: Ray Idaszak (RENCI, University of North Carolina at Chapel Hill)
  Contributors: Richard Arthur (General Electric), Roscoe Bartlett (ORNL → SNL), Ira Baxter (Semantic Designs), David E. Bernholdt (Oak Ridge National Laboratory), Ronald Boisvert (NIST), Karamarie Fecho (Copperline Professional Solutions), Rob Fowler (RENCI, University of North Carolina at Chapel Hill), Sol Greenspan (NSF), Michael A. Heroux (Sandia National Laboratories), Costin Iancu (Lawrence Berkeley National Laboratory), Christos Kartsaklis (Oak Ridge National Laboratory), Daniel S. Katz (NSF → University of Illinois Urbana-Champaign), Quincey Koziol (The HDF Group), Sandy Landsberg (DoD HPC Modernization Program [HPCMP]), Ernie Lucier (NITRD), John McGregor (Clemson University), Thomas Ndousse-Fetter (DOE), Aleksandra Pawlik (Software Sustainability Institute → New Zealand eScience Infrastructure [NeSI]), Albert I. Reuther (MIT Lincoln Laboratory), Walter Scarborough (TACC, University of Texas), and Will Schroeder (Kitware, Inc.).
- *Social Sciences Applied to CSE Software Systems*
  Leads: Daniel S. Katz (NSF → University of Illinois Urbana-Champaign), Aleksandra

[1] "→" is used to indicate a change of affiliation of a contributor between the workshop and the completion of this report.

Pawlik (Software Sustainability Institute → New Zealand eScience Infrastructure [NeSI])
Contributors: Gabrielle Allen (University of Illinois Urbana-Champaign), William Barley (University of Illinois Urbana-Champaign), Ethan Coon (LANL), Kevin G. Crowston (Syracuse), Kosta Damevski (Virginia Commonwealth University), Mike Glass (Sandia), Timo Heister (Clemson), James Herbsleb (Carnegie Mellon), James Howison (U Texas), Ray Idaszak (RENCI), Paul Jones (FDA), David Lesmes (DOE/BER), Robert Nagler (RadiaSoft LLC), David Tarboton (Utah State University)

- *Workforce Development*
  Lead: Abani Patra (University at Buffalo)
  Contributors: Michael A. Heroux (Sandia National Laboratories), Daniel S. Katz (NSF → University of Illinois Urbana-Champaign)
- *CSE Software in Industry and Manufacturing*
  Lead: Richard Arthur (General Electric)
  Contributors: Joerg Gablonsky (The Boeing Company), Tom Lange (Procter & Gamble-Retired), Todd Simons (Rolls-Royce), Hai Zhu (DuPont)

1 Introduction

Computational science and engineering (CSE) is an essential and growing component of every technical field today. Advances in CSE provide competitive advantages: reducing costs and labor, shortening development cycles, reducing or eliminating experiments, and providing information for situations where theory and data are not sufficient, and experimentation is not possible.

1.1 Dematerialization through digitization

Over the past 20 years, CSE has transformed economics, science and engineering by progressively "substituting atoms with electrons" in the words of some futurists [93, 98]. We are poised to continue and accelerate this trend leveraging capabilities such as better planning, improved process yields, precision logistics and efficiencies in local custom manufacturing to reduce historic waste of materials (including those supplying energy). One can observe the concept of "dematerialization" in the macroeconomic reduction or leveling off of environmental and natural resource impact despite growth in populations and gross domestic products (GDPs) [8].

CSE has transformed economics, science and engineering by progressively "substituting atoms with electrons." We can sustain this progress by further improving software productivity, usability, capability and sustainability.

Within this broad market effect, CSE software enables the substitution of physical methods with computational models for experimentation, prototyping, and trial and error through ever-greater fidelity and confidence in simulated physics, chemistry, biology, engineering, and more. This reduced impact on the planet benefits industry not only in public favor for being responsible citizens, but also in direct reduction in the costs associated with obtaining, processing, shipping, and ultimately disposing of physical materials. CSE software also plays a critical role in national defense, for example eliminating the need for nuclear testing, and in discovering fundamental properties of physics and the universe, such as the nature of black holes [76]. We can sustain this progress by further improving software productivity, usability, capability and sustainability, as well as availability of validation data to drive confidence in broader adoption of digital methods [39].

1.2 Challenges, trends, and opportunities

While CSE capabilities offer tremendous value to society, the current broad collection of CSE software faces fundamental challenges (see for example [39]). Our software base is large and expensive to maintain. New capabilities are developed by science and engineering experts who often lack sufficient training in software engineering. Furthermore, while the broader software engineering community can clearly inform CSE software efforts, the rules of thumb and body of experience often have to be recalibrated for effective use in CSE software efforts.

Furthermore, the usefulness of CSE software tools often depends on efficient execution, and trends in computer design now require effective use of concurrency in order to realize performance improvements on new computing platforms. Introducing concurrency capabilities into CSE applications forces fundamental changes in data models and any dependent functionality. In many CSE software codes this means a complete refactoring, or even a replacement, of existing code.

> Timely, significant software productivity and sustainability investments will have a magnified impact on overall scientific and engineering productivity.

In addition to challenges from hardware changes, the success of CSE leads to a desire for higher fidelity models that are more accurate, include more diverse physics, and scale and assimilate more data, leading to a larger and more complex software eco-system with even more complex organizational demands [108]. This success has also spurred a desire to make CSE software tools more accessible, so that more segments of society can benefit from their use, meaning we want simpler ways to use a CSE software base that is growing in complexity.

1.3 Characterizing productivity and sustainability

The emergence of CSE software as a central and indispensable element of the research and development (R&D) enterprise implies that software infrastructure must be integrated into the core processes of science and engineering. Solid foundations of algorithms, software, computing system hardware, data and software repositories, and coupled infrastructure are the building blocks of CSE. While the desire to support the new "exploration of newly discovered phenomena, development of new theories and capabilities, and research into new ideas" is certainly essential, often the race to achieve the next result takes precedence over sustaining the infrastructure upon which most success rests. The result has been duplicated effort, fragile code bases, long ramp-up for new users and other costs that, when considered over a longer span of time, reduce overall progress.

CSE software is developed and maintained by a disparate group of universities, national laboratories, hardware vendors, and small, independent companies. Few of these groups have the human resources to support and sustain the software tools and infrastructure that enable CSE or to develop transforming technologies. Instead, academic and national laboratory researchers depend on an unpredictable stream of research grants and contracts, few of which contain explicit support for software development and maintenance for the many years that these software projects will be used.

The open source model reflects the rise of collaborative projects that require the free exchange of software components as part of a shared infrastructure. Unlike hardware which becomes obsolete within a decade, the life of CSE software has consistently spanned two, often three, decades. Many national and international projects are predicated on the existence of a shared base of reusable and extensible software that can couple scientific instruments, data archives, distributed collaborators, and scientific codes, while also enabling research in software tools, techniques, and algorithms. In this shared, open source model, development is collaborative, with contributions from a diverse set of developers who are supported by

an equally diverse set of mechanisms.

The successful evolution and maintenance of such complex software systems depends on institutional memory: the continuous involvement of key developers who understand the software's design and participate in its support and development for multiple years. Unfortunately, such stability and continuity are rare. Research ideas can be explored by faculty and laboratory researchers with a small cadre of graduate students, but building and sustaining robust software requires experienced professionals and long-term commitments to hardening, porting, and enhancing that software infrastructure most valued by the research community.

Developing and supporting robust, user-friendly CSE software is expensive and intellectually challenging. Effective development requires many mundane activities not normally associated with academic research: software porting and testing, developing and testing intuitive user interfaces, and writing manuals and documentation. Thus one desirable concept is that of a software sustainability center that could work with academic researchers, application scientists, and vendors to evaluate, test, and extend community software. To ensure unbiased selection of those component technologies to be supported by the centers, independent oversight bodies could be appointed, ideally with membership drawn from academia, national laboratories, and industry.

Whatever model of support is used, its implementation should ensure that a stable organization, with a lifetime of decades, can maintain and evolve the software. At the same time, the government should not duplicate successful commercial software packages, but instead invest in new technology that does not yet exist. All too often, academic software tools fail to leverage commercial software capabilities and best practices. When new commercial providers emerge, the government should purchase their products, and redirect its own efforts toward technology that it cannot otherwise obtain.

The barriers to replacement of today's low-level application programming interfaces are also high, due to the large investments in application software. Significantly enhancing our ability to program very large systems will require radical, coordinated changes to many technologies. To make these changes, the government needs long-term, coordinated investments in a large number of interlocking technologies.

1.4 Software productivity

Scientific productivity can be considered an overall measure of quality of the complete process of achieving scientific results (Figure 1). Scientific productivity includes software productivity described in this section, along with execution-time productivity (efficiency, time, and cost for running scientific workloads), workflow and analysis productivity (effort, time, and cost for the overall cycle of simulation and analysis), and the value of computational output in terms of scientific exploration and discovery (more accurate physics, improved fidelity, coupled scales, etc.).

Software productivity expresses the effort, time, and cost of developing, deploying, and maintaining a product having needed software capabilities in a targeted scientific computing environment. Productivity has three primary elements: *feasibility* (the ability to build, integrate, and deploy a needed software product within given time and resource constraints), *sustainability* (the ability to maintain, configure, port, and evolve a software product as

needs and technology change), and *verifiability* (the ability to evaluate whether a software product satisfies specified criteria). A fourth element, *reusability* (encompassing modularity and adaptability), concerns the development of software capabilities that are anticipated to be usable in multiple similar and evolving products comprising a product family. Software productivity is further affected by two other concerns: *product value* (the anticipated utility and quality of the envisioned product) and *acquirer acuity* (the degree of insight and foresight as to what capabilities users need now and in the future).

1.5 Software sustainability

Software sustainability is the ability to maintain the scientifically-useful capability of a software product over its intended life span, i.e., "preserve the function of a system over a defined timespan" [74]. Whereas the overall concern of productivity is the (iterative) derivation of usable software capabilities, sustainability focuses more narrowly on the retention and evolution of those capabilities as scientific needs and enabling technology change. Sustainability encompasses the ability to understand and modify a software product's behavior to reflect new and changing needs and technology. This includes maintaining the software's scientific integrity consistent with underlying scientific concepts and computational assumptions. It also includes converting the software so as to operate correctly in differing computational environments.

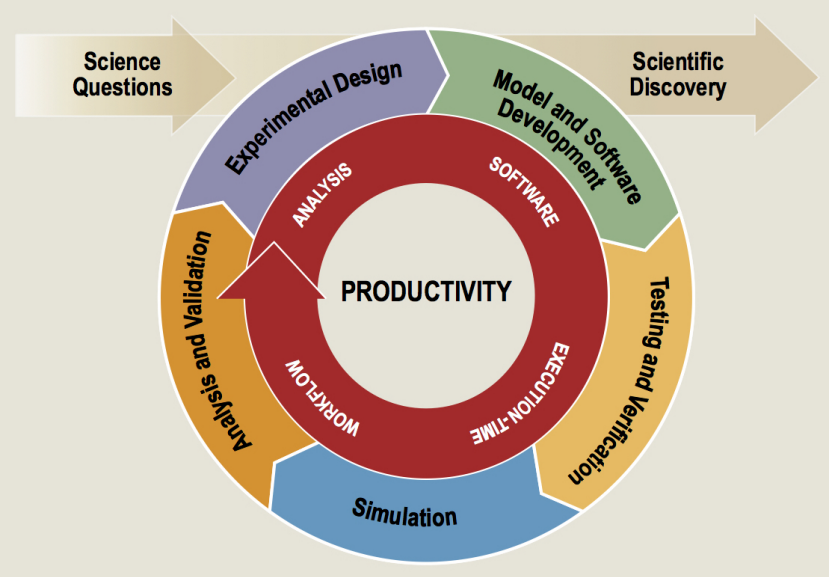


Figure 1: Scientific (and more generally CSE) productivity can be represented as a cycle of activities [39]. Software productivity is one component of overall CSE productivity. Because of the significant challenges our CSE software efforts face in adapting to new computing systems and meeting ever-increasing fidelity demands, investments in software sustainability and productivity will have a magnified impact on overall productivity. Furthermore, given the numerous advances in mainstream software engineering, the CSE community has a unique opportunity to dramatically improve the way we produce and sustain software.

Sustainability in particular concerns a variety of issues related to ensuring that software continues to meet its users' needs, including that it is affordable and maintainable over many decades, that it can be easily refactored and extended to respond to new programming models, hardware changes, or feature needs, and that it satisfies needs for reproducibility and verification.

1.6 The shared productivity-sustainability concern for product quality

While productivity concerns the ability to create needed software capabilities using chosen technology and sustainability concerns the ability to maintain those capabilities as needs and technology evolve, there is a shared concern for attainment and retention of product quality. Software quality encompasses deterministic functionality (the ability to perform its intended function, including predictability, interoperability, and reproducibility), performance (the ability to process its workload within available resource constraints, including utilization and throughput), dependability (the ability to operate effectively, including reliability, data integrity, and security), and usability (the ability to be used effectively and efficiently by users including accessibility and explainability). Quality is achieved through the use of an effective and efficient software development process that results in a software product that supports scientific productivity.

2 Improved CSE Software Sustainability and Developer Productivity: Opportunities and Incentives

Opportunities from CSE software advances:

- Rapidly prototype, deploy and test new ideas
- Algorithms and software for V&V of complex SW
- Reliable code with reproducibility constraints
- Performance portability on modern architectures
- High level productive domain-specific abstractions
- Improved workforce development and education
- Interoperable quality reusable software components
- Industry, academic and lab interactions around SW
- Usable software for beginners through better tools

From space science to genomics, disaster planning to cybersecurity, designing new materials to modeling the origins of the Universe, exploiting big data to designing energy efficient aircraft, the rate and extent of progress in academic and industrial research and development depends on the availability of scalable and optimized software which is also reliable, trustable, usable and accessible. The impact of good software is tremendous, providing the potential to shorten the design cycle between physical modeling and exploration to real CSE solutions, contributing to economic development, national security and scientific understanding. Good software enables the solution of more difficult problems, with less effort, more quickly (both in time to initially run the code and making the code complete runs faster). Also it would allow modifying and changing the code very rapidly based on solution feedback.

2.1 Opportunities

This report discusses and characterizes the challenges and opportunities for programmer productivity and software sustainability from several perspectives. Here we summarize the

key opportunities we believe could most benefit the CSE community.

- **Explore how the CSE high performance computing (HPC) workflow could be revised to improve productivity.** Software activities are currently a prominent aspect of the notional HPC workflow (e.g., Figure 2). These activities distract from the scientific endeavor itself, being concerned with primarily technical aspects of providing software capabilities needed in that endeavor. A more streamlined workflow could be envisioned that would reduce software-related bottlenecks. For example, abstraction and automation are key to improving productivity by reducing the dependence on multidisciplinary experts:
  - Providing computational abstractions to reflect the science and math of the problem domain would reduce the program complexity, improve understandability and ease maintainability and verification
  - Providing hardware-independent abstractions would make it possible to express algorithmic parallelization for optimization and tuning for performance and automating mapping to hardware for data layout, latency, etc.

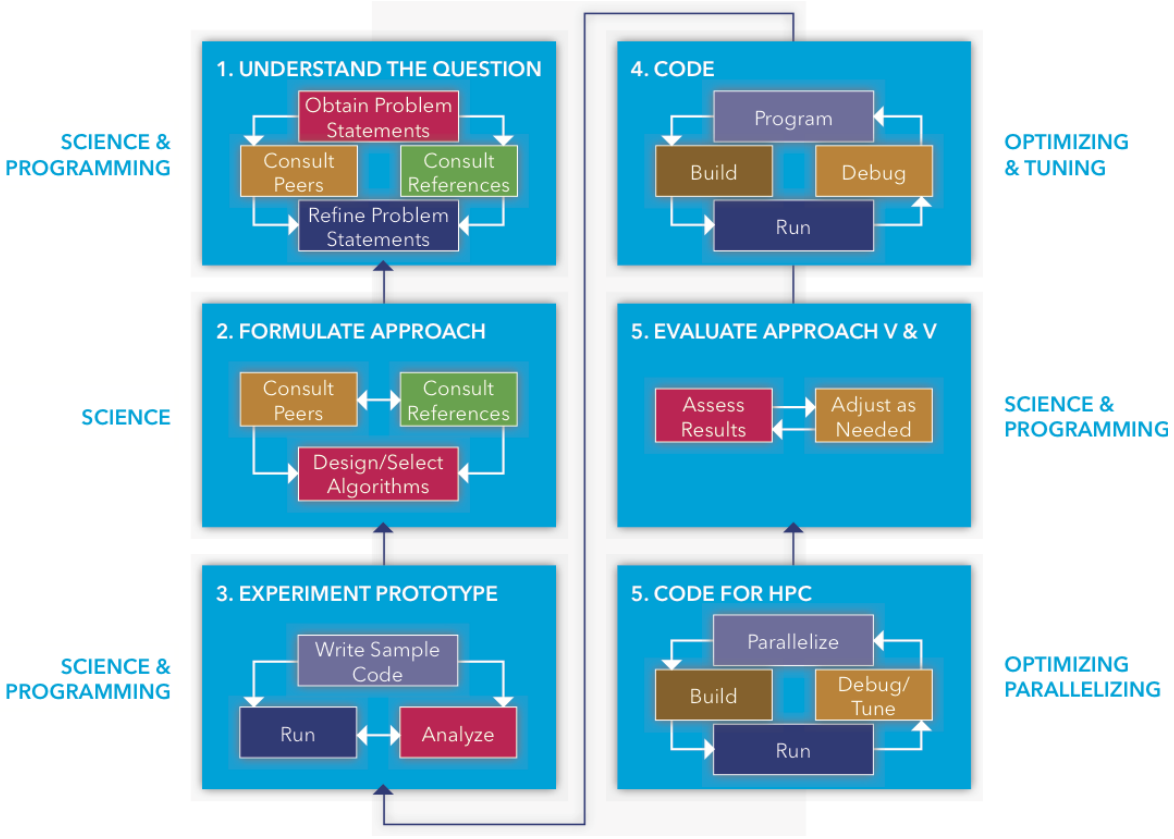


Figure 2: A canonical HPC workflow (Stuart Faulk). Software activities are currently a prominent aspect of the notional HPC workflow. These activities distract from the scientific endeavor itself, being concerned with primarily technical aspects of providing software capabilities needed in that endeavor. A more streamlined workflow could be envisioned that would reduce software-related bottlenecks.

- **Explore how the CSE HPC experimental environment could be improved with more advanced software capabilities.** Improvements could come from better integration of tools with interfaces tailored to the terminology, experimental methods, and subject matter of a scientific discipline. Instrumented domain-specific environments (see for example the breakout session discussed in section 2.2) could enable more effective management and performance of experiments and the use of community-sanctioned sharing of configurable models of physical processes.
- **Explore how CSE software practices could be revised to improve software productivity-sustainability and quality.** The software engineering discipline entails an investment of effort that provides a long-term payoff in terms of the software being better understood, being more easily evolved, and being a higher quality product. Such discipline involves practices such as documenting assumptions and rationale for maintainability, engineering to accommodate both current and potential future needs for evolvability, techniques for verifiability of models of physical processes, configurability to differing needs and computational technology (e.g., scaling), and improved techniques for all aspects of product quality (functionality, performance, dependability, usability).
- **Provide coordinated focus on advanced research in methods and algorithms for the community.** The challenges presented by exascale computing to scientific computing are daunting by themselves, and the aspirations of domain scientists to increase the size and complexity of their simulations by several orders of magnitude compound them enormously. What is needed is a plan to catalyze a multi-threaded research effort into numerical methods, algorithms, and software libraries that are most important to the rapid progress of simulations in the fields of interest to the community.
- **Improve application orientation of advanced work in methods, algorithms, and libraries.** The development of simulation software, like all mathematical software, is ultimately driven by the needs of applications, and therefore careful review and analysis of application requirements is critical. The activities will spread this understanding to collaborating researchers, using it to fuel and drive research in methods and algorithms with crucial cases from leading applications while at the same time fostering the use of the most advanced algorithmic techniques and optimized libraries among domain scientists.
- **Develop software technologies better adapted to the domain scientists.** Because of the complexity of the system layers that lie between the performance of the raw hardware and the usable or sustainable performance available to applications, the tradeoff between the usability of software for domain researchers and their ability to wring performance out of it tends to be very steep. This challenge is already formidable in today's high-performance computing environments, and should current trends continue, looks nearly insurmountable at the exaflop levels for all but an extremely specialized range of applications. A sponsored collaborative project to create software technology that dramatically lessens the trade-off gradient, achieving much better performance with only moderately greater programming effort, would be of great help.
- **Sponsor and coordinate expert consulting services for domain researchers.** Faster development of better-quality software would mean that domain scientists could spend less time writing and debugging programs and more time on research problems. Coordinating the consulting activities of the methods, algorithms, and libraries community for the domain scientists would reduce duplication of software development effort

by promoting the sharing of software modules, reduce time and effort spent in locating relevant software and information through the use of appropriate indexing and search mechanisms, facilitate the adoption of useful standards, promote the use of superior performing libraries, and so on.

- **Identify metrics for improving software productivity-sustainability and quality.** Improvements in software productivity, sustainability, and quality should be based on objective measures of the effectiveness of current practices. These measures provide a baseline against which the effectiveness of alternative practices can be evaluated as beneficial. Measures to collect should be chosen to answer specific questions concerning process and product improvement objectives with the intent of informing the weighing of alternatives and making decisions on how to achieve those objectives. The cost of collecting process and product data should be weighed in determining what questions can be quantified.
- **Identify and pursue opportunities for international collaboration on software as infrastructure for scientific research.** The objectives and challenges of CSE experimentation are international. Solutions are similarly shared through conferences, publications, and shared models of physical phenomena. These are subsequently replicated and validated by other researchers around the world. Opportunities for shared community-wide advances in using software capabilities as an infrastructure for scientific research should be pursued at an international level when feasible.

The following sections explore several perspectives on productivity and sustainability. The topics match the workshop breakout sessions, and serve as a useful factoring of the broad content scope. Workshop attendees and other members of the scientific, engineering, and software communities have collaborated in producing the content with the intent of increasing understanding and providing motivation and direction for future efforts to improve programmer productivity and software sustainability.

2.2 The CREATE Project: Emphasizing sustainability and productivity

The Computational Research and Engineering Acquisition Tools and Environments (CREATE) program is a multiphase Department of Defense (DOD) effort that started in 2008 with the plan to develop and deploy computational engineering tool sets for acquisition engineers. The physics-based software resulting from the program has been used in performance predictions using virtual prototypes that augment physical testing to identify design defects throughout the acquisition process, to substantially reducing acquisition time and cost overruns. CREATE

The productivity and sustainability driven CREATE Project provides tools for antenna, aircraft, ship and ground vehicle design. CREATE tools provide virtual protypes that augment physical testing, identifying design defects and substantially reducing acquisition time and cost overruns.

developed toolsets for:

- Meshing and geometry support: the geometry and meshing project improves the ease, speed, flexibility, and quality of geometry and mesh generation, and enables the generation of general digital representations and product models of weapons systems and platforms and operational terrains and environments.
- Radio frequency antenna design and integration tools: conceptual design and detailed analysis tools relevant to virtually all DOD platforms.
- Aircraft design tools: fixed-wing aircraft, rotocraft, conceptual design, trade-space exploration, and operational testing and translation.
- Ship design tools: shock/damage, hydrodynamics, early-stage design and trade-space exploration, and operation testing and translation.
- Ground vehicle tools: end-to-end mobility solver, provide rapid, physics-based data for design and trade-space analysis.

2.3 Providing incentives: Funding agencies, publishers and employers

Funding agencies, publishers and employers play an essential role in establishing CSE software quality expectations. Any sustained improvement in quality requires increased expectations from these entities.

Funding agencies, publishers and employers have an opportunity to directly influence the incentive structure of for improved quality in CSE software, and thus raise awareness of best practices in software engineering and provide motivation for steady improvement in CSE productivity. Many CSE software teams express a strong desire to improve sustainability and productivity, but the pressures of acquiring funding and meeting project deadlines, publishing papers, and satisfying employer requirements for rank and tenure or company promotion leave little time for focusing on these improvements. However, this situation can improve:

- **Funding agencies:** Funding agencies and project leaders can directly increase incentives for investing in productivity and sustainability by requesting that work proposals and plans include descriptions of how software will be produced and sustained, how developers will be trained, and more. By asking for this information a baseline can be established for subsequent proposals that require a minimum competency in these areas.
- **Publishers:** Publishers can provide incentive by increasing expectations of independent reproducibility in computational results. When authors know that a reviewer will attempt to replicated or otherwise repeat their computations in order to produce the same results, there is a natural incentive to improve numerous productivity and sustainability attributes of their software base [7, 60]. Similarly, conference proceedings can provide incentive by evaluating computational artifacts as part of the conference paper review process [82].
- **Employers:** Universities, laboratories and business can elevate the importance of productivity and sustainability by placing high value on publications, projects and products

that have proven software quality. When coupled with the above funding agency and publisher initiatives, metrics will be easier to determine as part of the promotion assessment process [31].

Funding agencies, publishers and employers can play an important role in encouraging improved quality of CSE software development processes by more explicitly recognizing the role that software plays in the CSE research they fund, and setting general expectations and providing incentives related to the software itself. On the other hand, care must be used to avoid unintended consequences. We believe that the value derives from thoughtful examination and improvement of software development processes rather than from the achievement of any particular value for any particular metric. Any metric can be gamed, and if the stakes are perceived to be high enough, it will be. Rather than mandating specific procedures or metrics, it seems preferable to set high-level expectations, and let development teams respond as to how they propose to meet those expectations in their project. This is similar to the approach many funding agencies have taken in response to the Holdren memo [63] to require that proposals include a project-specific data management plan, which is then evaluated for responsiveness and adequacy as part of the peer review process.

Funding agencies can play an additional role in fostering improved economics of CSE software, acknowledging through their funding models that investments in good SWE during software development by academic researchers will save money and increase research productivity in the long term, ultimately producing consistently superior software. The National Science Foundation (NSF) Software Institutes [91] are responsive to this charge and are on the path to achieve sustainable, high-quality software for several communities of users. The Science Gateways Software Institute (SGSI), for instance, will serve as a hub and a software ecosystem for the development of science gateways [103].

Economic improvements can also include funding long-term software maintenance equally with software development. Specifically, dedicated staff members are required for long-term software maintenance. As with laboratory equipment, developed software products require maintenance costs over time. For example, software like Geant 4 [53] generates more data than the hardware it models; as such, software maintenance benefits from having a budget commensurate with initial software development. Another example is the instrumental role that the Pegasus Workflow Management Engine played in the recent LIGO announcement of the discovery of gravitational waves [97]. This important discovery required years of underlying research that relied on critical software tools, thus highlighting the importance of long-term funding for software maintenance. Funding models such as those invoked by the NSF typically limit project support to five years, which is insufficient to meet long-term software maintenance requirements. Finally, funding research on how best to apply SWE best practices for academic CSE research [12, 32, 59, 113] is important, addressing questions such as how to balance SWE and infrastructure needs with the needs of a given research project, within the scope of that project's overall budget.

3 Role of Software Engineering Research

In the last couple of decades, modeling and simulation have firmly established their importance in the scientific discovery process. With advances in modeling, numerical techniques,

and the ever-increasing power of computing platforms, many highly-complex problems have become tractable. The role and importance of simulations is obvious with respect to those phenomena that are inaccessible to theory, experiment, or observations. CSE is also making its usefulness clear in many fields where experiments are possible but simulations provide insight at much lower cost, i.e., industrial applications where production design increasingly relies on simulations rather than large experimental facilities (e.g., [33, 67, 105]). Similarly, in several instances simulations are instrumental in the design of experiments, a symbiotic process that accelerates scientific discovery through better designed and understood experimental results (e.g., [123]). While the role and importance of computational science has been largely accepted by various scientific research communities, similar attention has not been given to the primary tool of CSE, namely software [39, 62].

CSE software as an enterprise has yet to emerge as a creative discipline in its own right. Some modest- to large-sized groups have understood the need for reliable, robust, and reproducible results, and give due importance to their software. Several such code bases have become community codes for their respective communities either by design or through adoption and evolution. (For a partial set of such examples see [50].) Adoption of similar practices more broadly is not just desirable, it is necessary to consolidate the role of science through computation. A strong scientific discovery process requires reproducibility of results, and simulations should strive to meet this requirement. Although the constantly changing landscape of system software and hardware makes this difficult, documenting computational experiments appropriately can mitigate the challenges. This documentation would require a verified and validated code base with a strong software engineering and auditing process in place that would allow reconfiguration to repeat a computational experiment. The minimal requirement to meet the standard of believable scientific discovery is to treat the code base as infrastructure, similar to the way the experimental facilities are treated.

CSE software as an enterprise has yet to emerge as a creative discipline in its own right. Both model complexity and hardware complexity are growing simultaneously, and they both make the other more difficult to manage. The time is upon us to address the growing challenge of software productivity, quality, and sustainability that imperils the whole endeavor of computation-enabled science and engineering.

The lack of support for computational software as infrastructure has been an ongoing challenge, and with the change in hardware paradigm right down to the commodity processors it has become a more complex and ubiquitous issue. Both model complexity and hardware complexity are growing simultaneously, and they both make the other more difficult to manage. The time is upon us to address the growing challenge of software productivity, quality, and sustainability that imperils the whole endeavor of computation-enabled scientific discovery. Analogously, opportunities abound to improve overall CSE productivity and quality through modernization and collaborative advancement of software engineering practices.

3.1 Expanding software engineering practice

The broader software engineering community has examined and addressed several of the issues that the computational science community is now experiencing. However, their solutions have found limited acceptance in the CSE community, partly because of lack of exposure and partly because they are not always applicable, at least without adaptation, in the CSE community.

Studying those CSE teams that have had success in using software engineering could lead to a better understanding of where software engineering research and/or adaptation of prevailing practices can help the CSE community. For example there are areas such as performance portability and tool support for Fortran that are important to CSE but get little attention in the wider software engineering community.

CSE researchers need easy-to-use, efficient, and precise software engineering tools that present clear and immediate benefits to their project while imposing minimal overhead. Not only do these tools improve productivity and enable development teams to grow, they provide critical risk management with respect to software errors (bugs). The adoption of these tools and practices would be accelerated by the dissemination of "learning experience" reports by CSE peers. Such reports could also describe subjective and objective changes in productivity and technical debt during adoption. Of key importance is the method of dissemination; an online collection alone is not quite sufficient. It should be augmented with conference presentations, on-site seminars, etc.

The longer-term CSE goal for software engineering practice should be to institute improvements that will reduce bottlenecks in the canonical HPC workflow [49]. These include computational abstractions that reflect the science and math of the problem domain, hardware-independent abstractions that support problem-specific parallelization and optimization/tuning, automated performance-preserving mapping of software to differing computing configurations, and systematic configurable reuse of verified domain-specific software capabilities. This goal is anticipated in software engineering efforts to introduce product line approaches for creating domain-specific families of software. These approaches improve productivity, sustainability, and quality by leveraging common effort needed to build and maintain similar software (e.g. for a field of CSE research), providing decision-guided mechanisms for deriving software customized to the purpose and technology of each supported project.

The intent of implementing these recommendations is: (1) to reduce the need for individuals with multidisciplinary expertise, instead allow individual with diverse expertise to work in more effective collaboration; (2) to reduce manual labor through automation of software activities; and (3) to provide scientists with a virtual experimental framework where computational capabilities are tailored to the needs of their scientific domain.

The purpose of an experimental framework (e.g., as the realization of product line engineering, providing the means to derive customized computational software) is to enable performing experiments for understanding and predicting behavior and effects of physical processes. Such an experimental framework should be conceived as a coherently integrated scientific instrument which links empirical (physical) data with (numerical) models in a manner that allows continual recalibration/correction similar to the figure. This instrument takes the form of a computationally-enabled scientific environment that: (1) provides scien-

tists with an operable interface in terms of terminology, methods, and models appropriate to their field of research; (2) encapsulates access to configurable models of relevant physical processes and/or data from real-world sensors; (3) provides capabilities for the management and iterative performance of an instrumented experimental process (configure models, formulate experiments, perform experiments to collect data, analyze and evaluate results, and report conclusions); (4) abstracts the underlying computational environment used for experiments, with solution software scaled (up or down) and optimized as needed to make best use of allocated resources; and (5) maintains and improves its capabilities as scientists' needs and enabling computational technology evolve.

3.2 New software engineering research

CSE software scale is comparable to the scale of products at large software companies, while development funding is driven more by domain research needs of new capabilities than large volumes of product sales. This affects the organization dynamics in a way that may necessitate different approaches to software engineering. Because funding is driven by novel capabilities, user requirements change at an accelerated pace, which can be addressed in part by composable, component-based designs. Some CSE software targets a wide range of hardware, including unusual and experimental architectures. Combined with the fact that a significant portion of the value of CSE software is in its performance, this results in a strong focus on performance portability. CSE software provides vital contributions to scientific understanding and helps to inform critical decisions in the design of physical systems. Combined with the high rate of change in the source code induced by shifting requirements, this puts a strong focus on validation and verification in all its forms. The results of CSE software should ideally be invariant across changes in both the implementation source code and the executing hardware architecture. In practice such changes can induce acceptable variance, requiring the design of mathematical methods and tools to correctly interpret the differences.

3.3 Transition of research to practice in CSE software community

The CSE community faces a number of challenges in software lifecycles. A great deal of CSE software begins life as a research project whose only purpose is to generate research results. If it is found useful, it often grows in both use and size through accretion. Typical *ad-hoc* software lifecycles by which CSE software matures often occur slowly and at great expense (demonstrating some large failures along the way [99]). Typically, by the time software reaches production-level maturity, the state of the software is such that major development largely stops and only minimal maintenance efforts can be afforded. Therefore, new software is created for new research, and the cycle begins again.

This typically slow and ill-defined lifecycle process for research-birthed CSE software can likely be significantly improved by creating better-defined lifecycle models and processes based on lessons from the broader software development community. In particular, software engineering's agile and product line practices may be attractive for the creation of CSE-targeted lifecycle models. All practices should be expected to contribute to the engineering

discipline necessary to support CSE goals of productivity and sustainability. However, practices to be adopted should be scalable so that efforts supporting early exploratory research remain productive while building toward needed quality, reproducibility of results, and sustainability as research needs become more demanding and enabling technology evolves. One example of an effort to define such a lifecycle model for research-based CSE software is described in [13]. Research and more case studies are needed to better define such lifecycle models and improve their adoption by the CSE community.

Another way to raise awareness for the need of better software engineering in CSE is to formalize the dissemination of software engineering practices. An excellent example is the Software Sustainability Institute in the U.K., whose mission is to cultivate better, more sustainable research software to enable world-class research (better software, better research). In such an institution, research can be focused on a better understanding of the particular software engineering requirements of the CSE community along with extraction and dissemination best practices tailored to the unique needs of the CSE community through a unified platform. The U.S. can start by organizing a working group to propose a scope and a charter with objectives, with a final aim of establishing such an institute.

The diffusion of good software engineering practices into the CSE community can also be stimulated by defining a canonical set of practices appropriate to CSE research projects. However, any such characterization should be descriptive rather than prescriptive as to specific techniques or representations to be used. Consensus as to preferred techniques and representations should be allowed to arise organically through experience and collaboration within the CSE community.

3.4 Collaboration opportunities

Sustainable software practices in several successful CSE projects [9] have common optimal workflow patterns that support long-term extensibility and collaborative development with their user community. But often, adoption of the process goes through a series of iterations to converge on minimal developer productivity impedances through tailor-made customizations in the process. This intrinsic variability leads to a spectrum of standard practices that may not conform to recommended definitions. For example, there are different recommended workflows when using distributed source configuration management like Git (workflow comparison) that scales well depending on the size of the team. But, the suitable choice ***evolves*** as both teams and software make their transition in terms of capability, complexity, and maturity [68]. Due to this dynamic nature in process evolution, research targeted at scientific software is needed to carefully design metrics and to standardize process definitions that help to continually evaluate progress towards sustainability.

The march towards mature scientific software development requires key infrastructure and process optimizations in software configuration and automated test management. This is especially imperative when increase in code and feature reuse correlates with increase in complexities due to external dependencies (growing software ecosystem). Robust and scalable configuration and build system implementations along with rigorous automated testing do serve as the first level of abstraction to enable portability on existing and emerging hardware systems. Additionally, the configuration process needs to be flexible to accommodate interoperability issues in multi-model (multi-physics/multi-scale) simulation workflows that

often involve mixed-language, individually hosted code repositories. Even though the time investment in development of these practices can be relatively high during the initial stages of the project, the productivity payoff can be significant as the scientific software matures.

3.5 Specific needs of small user teams

Processes designed in the software industry for large-scale development needs may not yield increased productivity in small CSE teams. However, several of them, including automated testing infrastructure (unit, integration) and test suites, should be followed to catch regressions and to maintain short cycles to feature completion.

Development of continuous integration and deployment strategies through containerization techniques (Docker, Vagrant) can provide a powerful platform for both jumpstarting the learning curve for new users and to ensure repeatability of experiments and reproducibility of scientific results. Such methods can also provide standard frameworks to disseminate good software engineering practices, either through software carpentry [112] or canned tutorials, thereby reducing the barriers for small research teams to adapt to the growing scales in successful software development.

3.6 Software engineering research roadmap

The understanding of how software engineering research can help the CSE community begins with profiling how software is used in the community. Profiling should characterize a notional CSE workflow, noting any differences among scientific disciplines and also how software is used within the workflow, including packaged and custom-built tools. The tools should be categorized in terms of capabilities provided. It would also be useful to profile how software is supplied for use in CSE, including a characterization of how custom-built software is developed and sustained and which practices are used for its management, process, requirements, design, implementation, evaluation, and delivery. Characterization of how software packages are selected, configured for use, and sustained, and any experiences with shared use of software within or across CSE research efforts would also be very informative.

Once the understanding of the above has been reached, a roadmap (e.g., [30]) can be developed for research and development for improved CSE software practices. This roadmap should identify means to improve the computational quality of existing tools for CSE, reduce dependence of solutions on physical computing resource configuration (e.g. consistent behavior from standard math functions like sin/cos), and improve practices for better software productivity, quality, and sustainability. This will need more effective tools, configurable to differing needs and with their own sustainability guaranteed, that would enhance CSE productivity, quality, sustainability, and means of organizing and integrating tools into coherent environments for domain-specific scientific experimentation

The final step in the roadmap is to develop the means to transition improved CSE software practices into use through the establishment of a resource for proposing and discussing needed improvements and for sharing experiences, productizing, validating the effectiveness, and sustaining improved practices and tools, and assisting CSE organizations in identifying and instituting improved practices and tools.

3.7 Productive and sustainable: Astrophysics community codes

The astrophysics community has been ahead of many other science communities in making research codes publicly available, and has been a leader in the development and adoption of community codes [42]. ZEUS-2D [117, 118] was one of the earliest codes to become public, and it has been followed by several others such as FLASH [52], Gadget [114], Enzo [95], Cactus [56]/Einstein Toolkit [84] and Athena [119] (see ascl.net for a complete list).

> The astrophysics community is a leader in fostering community codes. From the early 1990s codes such as ZEUS-2D, FLASH, Gadget, Enzo, Cactus/Einstein Toolkit and Athena are part of a long legacy. Community codes advance science in numerous ways and provide incentive for junior community members to contribute and receive recognition.

Widespread use of open-source tools has helped to hasten a change in attitude in the theoretical astrophysics community. Over the past decade the concept of a "community code" has become gradually more accepted, and young scientists have seen how contribution to these open-source projects can have a positive impact on their scientific reputation and career prospects.

Having completely open tool chains (e.g., Enzo, FLASH of the Einstein Toolkit used for simulations, and yt [121] for data analysis and figure production) also helps with the promotion of open, reproducible research, with some authors releasing simulation parameter files and yt analysis scripts along with their papers. This makes it relatively straightforward to reproduce scientific results, supports federally-mandated goals regarding the sharing of scientific data, and assists with making "apples to apples" comparisons between different simulation tools [4]. Additionally, investment in a well-architected community code framework can sometimes have unexpected beneficial side effects. For example, because many needed capabilities overlap between astrophysics and high energy density physics (HEDP), FLASH was adopted as the primary open code for the academic HEDP community [107, 122]. The impact was huge because the academic HEDP community has historically had very little access to codes capable of simulating experiments for design and analysis. The presence of an adaptive mesh refinement (AMR) framework [87] with built-in IO, runtime management, many needed solvers, and plugin-capable architecture has enabled a community code for an entirely new community in a short time (less than two years) with very modest investment.

4 Measuring Software Productivity and Sustainability

One of the fundamental goals of this workshop was to gain an understanding of the current state of software productivity and sustainability in CSE and discuss strategies to improve it. But how can we, as a community, share, critically discuss, and build upon the work and experience of others to improve the state of practice? How can we as individual software developers and teams become more productive and create more sustainable software? Measurements, metrics, and indicators are important tools in the objective understanding, assessment, and improvement of any process, including software development. They allow us to apply a systematic, engineering methodology to the software development process, identifying techniques that are useful and those that aren't. Broad collection of metrics can also help identify unintended consequences of changes made to the development process. Objective metrics can be meaningfully shared and applied to different software development projects and compared. Metrics may be useful in incentivizing behavior, defining reward structures, and in influencing the economics of CSE software development.

Measurements, metrics, and indicators are important tools in the objective understanding, assessment, and improvement of any process, including software development. They allow us to apply a systematic, engineering methodology to the software development process, identifying techniques that are useful and those that aren't.

Although, from its title, the nominal focus of the workshop was on software sustainability and productivity, there was significant discussion of scientific productivity as well. In many cases in CSE the ultimate goal of the work is to advance scientific knowledge and understanding. As such, we might wish to directly measure scientific productivity and make adjustments to given CSE software development processes to improve scientific productivity. Publication counts are widely used as a basis for measuring scientific productivity. However, publication counts alone cannot gauge the impact of a publication or collection of publications. Nor do measures of publication counts or impacts (however they might be defined) provide any insight into the correctness or reproducibility of the work, or, for computationally-based research, the quality of the software used in the work. All of which are issues currently drawing scrutiny in scientific research, and serious mistakes can actually retard scientific progress and impact [89]. Finally, measures of scientific productivity tend to lag significantly behind the software development process. Papers often do not appear in print until months, or even years, after the software on which they are based was written. Measures of the impact of these papers lags further. As an extreme example, Nobel Prizes are typically awarded 20–30 years after the initial publication [86].

This illustrates two of the challenges of attempts to measure the software development process (or any other). First, there are many different metrics which can be used. How do we know which is the right one? Second, if we want to use the measurements to improve our processes, we need a reasonably tight feedback loop, measured in days or weeks—perhaps a few months—rather than years. Although the connection between software development

and scientific productivity remains of interest, as a practical matter, it seems that focusing more directly on the software development process itself is likely to provide a tighter feedback loop. So we return to the concepts of software sustainability and productivity. Unfortunately, just as with scientific productivity, we can envision many different metrics for software productivity or software sustainability.

The SWE community, however, has found a way to address this issue. According to Rombach and Ulrey [106], "the primary measurement question is not 'What metrics should be used?' Instead, the primary question is 'What do I want to learn?' or 'What is the measurement goal?'". The goal/question/metric (GQM) approach they advocate [14–16, 96, 106] provides a structured approach to specifying measurement goals, refining them into a set of questions, and identifying metrics that can answer the question. Following this model, we recognize software productivity and software sustainability as goals, rather than things we expect to measure directly. Rather than spending time trying to define productivity and sustainability in ways that can be quantified, we can develop a set of questions that address the goals in specific ways, tailored to the situation at hand. Metrics can then be defined to answer the questions. For example, we may ask the question, "What is the largest bottleneck or the largest overhead to developers on my project hindering them from delivering new features?" Metrics that might shed some light on that question include the average time to rebuild the code after changes, the average time to re-run the test suite, the average lines of code that are changed in order to add a new feature, the average time needed to get code reviews performed, the average time spent relearning code before it gets changed, etc.

Despite the fact that the GQM method has been around for nearly three decades, and the fact that the SWE community has even more experience in developing and applying metrics to software and software development, there remains a significant gap: these approaches, which are well known in the SWE community are neither known nor used to any significant extent in modern CSE software development. Where they are used, the experience is often not considered part of the "science" product of interest and is therefore often not reported in the scientific literature, nor even discussed extensively in less formal settings. There may be many reasons for this gap but we believe two points are at the core of the problem. First, most developers of CSE software are trained as domain scientists and have little or no actual training in the software development aspect of their work, instead picking it up informally from others who are likewise trained in the scientific domain but not in software development. This fact has many consequences, including that the CSE "culture" is generally not very introspective as to the software development processes it uses. Second, the CSE community and sponsors of CSE research typically do not place a high value on the software produced in the course of research when compared with the "science" output. A corollary to this is the fact that discussion and sharing of software development practices and experiences is also not highly valued.

4.1 Opportunities for investigation

We believe, therefore, that in order to make the measurement of software productivity and sustainability a useful and widely used approach within the CSE community we must take steps aimed at understanding how both the CSE and SWE communities use metrics, and

understanding how the metrics they have already identified can inform CSE software development. We also need to examine CSE as a distinct type of software from many others, and look for questions and metrics that may be unique to CSE software. And, finally, we need to look for ways to increase the level of introspection applied to software development activities within the CSE community. Many of these activities can be connected closely to the issues and opportunities identified in the previous chapter for software engineering research in a broader context in the field of CSE.

4.2 Understanding the use of and experience with metrics in the SWE community

As mentioned above, the SWE community has extensive experience with attempts to quantify software, software development, and software developers in order to address a wide range of questions and goals. By way of example we describe but a tiny sampling in the following paragraphs.

Agile software development processes tend to be very lightweight and define few metrics. But the metrics they do define are generally taken very seriously. For SCRUM processes, the main metric is function points delivered per sprint [79]. This metric is used to measure process improvement efforts (implemented as part of a retrospective process) but is primarily used to estimate how many stories can be completed is future sprints. Kanban processes tend to define the cycle time, or lead time, which is the time between when a story begins work and when it is completed and delivered to the customer [79]. The cycle time then becomes both a metric used in iterative process improvement and to identify bottlenecks in the overall process.

The most common metric in software development processes in SLOC (source lines of code). It is easy to measure and there are well known correlations between SLOC, the number of developers, and the amount of features that can be implemented in a given amount of time. Such data and correlations are very useful to know when doing software estimation. But as a metric to use directly in process improvement (or to optimize for), SLOC metrics are widely criticized as being counterproductive.

The standard software productivity metrics mentioned above are fairly coarse-grained in that they can't identify what is causing productivity problems; they can only give some indication if things are getting better or worse. Therefore, mature and skilled teams tend to also define and use several finer-grained metrics to help pinpoint bottlenecks and areas that need improvement. For example, Google adds automated monitoring into many of their build and testing processes to see where time is wasted (i.e., waiting for builds to complete and tests to run).

But, arguably, the most useful metrics are obtained from investigating where developers and users spend their time. For example, many traditional software development projects require issue tracking and require developers to log their time in a fairly fine-grained way. Metrics such as the number of defects reported, defects fixed, time spent fixing defects, when a defect was introduced (i.e., the requirements, design, or implementation stage), how long the defect took to discover, and how expensive it was to address the defect are all recorded in many projects. These types of metrics are some of the most useful in development and

process improvement efforts.

In absence of objective finer-grained metrics, it may be useful to investigate if qualitative surveys of developers, users, and stakeholders can be used to provide a means to determine if things are improving or getting worse. Such an approach is advocated in [51].

It behooves the CSE community to survey the SWE community's experience with these many and varied metrics and the questions they have been used to address. While we cannot necessarily expect everything to translate directly into the CSE domain, understanding this prior experience will accelerate the learning process.

4.3 Exposing the current use and experience with metrics in the CSE community

It is not as if there is no awareness of SWE in the CSE community, and there certainly are software teams who thoughtfully try to examine and improve their software development processes. However, as mentioned above, these activities and experiences are often not well known because the CSE community and research sponsors do not recognize them as part of the scientific output and thus the researchers carrying out this work are not incentivized to spend the time and money required to capture and convey their experiences to the broader community.

As part of the larger opportunity to expand the practice of SWE in the CSE community noted in the prior chapter, it would be extremely valuable to identify situations where SWE practices already in use can, with relatively little additional effort, be exposed for the benefit of the community. Over the longer term, it may be possible to modify the incentive system, and eventually the culture to place a higher value on the SWE research that can arise from such work. However, in the near term, it may be useful to encourage the exposure of this work by lowering the barriers to such exposure. While it may, at present, be hard for CSE researchers to justify putting in the time and effort necessary to bring their experience to the level expected for a paper in the SWE literature, we can provide venues that value the "experience" paper or presentation with a somewhat lower bar.

4.4 CSE as a distinct software domain

Although at some level CSE software is software like any other, there are also aspects of CSE that result in the software products in this community having characteristics that are distinctive or different than many other types of software [17]. As interest in the understanding and use of SWE to improve software development grows in the CSE community, there is a significant opportunity for the two communities to work together to obtain a deeper understanding how CSE software is both similar to and different from other types of software. Of course this is a general software engineering research question (see previous chapter), but it clearly extends to unique ways to measure CSE software and development processes. In our discussions at the workshop, there was even speculation that different domains within the broad field of computational science and engineering might justifiably define specialized metrics, for instance as the climate modeling community today uses simulated years per day of computer time as a metric of the performance of their simulations.

4.5 Encouraging and increasing introspection into CSE software development

The work described above would play a crucial role in helping to understand the role and potential benefits of measurement in CSE software development. Having evidence of past successes would also be very valuable in helping to persuade CSE software developers that their own work might benefit from a closer examination. However, we believe that there are additional steps which would further enhance the willingness and ability of CSE developers to use metrics to understand and improve their software and their development processes.

One important aspect is to improve the available tools, and awareness of them, to help increase the set of metrics which can be obtained more or less automatically just from the usage of software development and workflow tools. For example, one can automatically determine the number of comments made on a GitHub pull request (as a measure of the toughness of a code review) and then try to correlate this with a reduction in software defects related to that pull request [37]. Another example might be a scientific workflow management tool that directs (and records) the process for setting up a new simulation or analysis. Such a tool could record the amount of wallclock time needed to perform each step as part of recording the intermediate files in a content-management or version control system. Code metrics, for example line counts (SLOC), complexity metrics, static analysis results, and others, can be obtained by applying appropriate tools to the code base in a version control repository and perhaps updated with each commit. Test coverage and related software quality metrics can be obtained as part of the testing process, which can also be automated.

Compliance requirements and recommendations for software provide another mechanism to improve and measure the quality of software. Basic items such error handling policies, bug reporting mechanisms, minimal portability requirements, and test coverage are easily checked via tools and automatic scripts. While universal standards may not be achievable, within communities there may be enough consensus to develop meaningful standards. One approach is the compliance requirements established by the xSDK suite of libraries [111].

Automatically gathered metrics could be made available to developers via a "dashboard", to facilitate both awareness and use of them. By working across the community, and informed by the work of the previously described opportunities, it should be possible to identify a set of "baseline" metrics, which are easily and automatically gathered, and are generally recognized as providing useful, actionable input to the CSE software development process.

A second important aspect would be to directly encourage increased introspection of CSE software development practices by funding such activities. In a whitepaper submitted to this workshop, Ahalt, et al. propose the establishment of an open, community-governed framework by which to develop "metrics of success" for CSE software, particularly with respect to software sustainability [2]. They envision the effort beginning as a funded activity involving a group of researchers and stakeholders tasked with reviewing their own and other participants' software with an eye to identifying metrics which correlate with success of the software in satisfying stakeholder needs and goals. Such an activity would serve as a pilot for a future, larger effort to establish and refine a framework for sustainable software metrics.

4.6 Measuring productivity and sustainability roadmap

In order to foster understanding, development and deployment of productivity and sustainability metrics, we suggest the following roadmap:

- Create a pilot "peer-review group" of grant recipients funded to develop sustainable software [2]. The primary goal would be to define and refine CSE software success metrics. This group could also be involved with any of the efforts listed below.
- Survey existing CSE projects looking for existing metrics, gaps, and opportunities.
- Perform case studies on the GQM approach applied to CSE projects.
- Investigate how to improve automatic gathering of metrics from software development and workflow tools.
- Encourage the usage of issue tracking for CSE developers and users to capture finer-grained metrics (but only if other benefits are derived from the usage of such tools and processes).
- Invest in the usage of dashboards to collect, display, and analyze metrics.
- Build a community of individuals interested in process improvement and metrics (e.g., the application of the GQM method).

5 New Approaches for Faster, More Affordable CSE Software

Present day methods for building CSE software are remarkably traditional and fail to leverage knowledge or tools well; one gets the uneven results of individual craftsmanship rather than engineering discipline. We believe that raising the level of abstractions used to develop programs can make them easier to write, easier to maintain and enhance, and, crucially, easier to (re)target to the rapidly changing landscape of high performance computing systems. We also believe that automation can be provided to enhance the development process, by offering complex predefined mappings of abstractions to targeted hardware, providing timely analyses of issues in code as it is modified, enabling the construction of more complex models, and providing support to reverse-engineer legacy models into more manageable abstractions.

Present day methods for building CSE software are remarkably traditional and fail to leverage knowledge or tools well; one gets the uneven results of individual craftsmanship rather than engineering discipline. Increasing automation in workflows and raising the level of abstractions used to develop programs can make software easier to write, maintain, enhance and adapt.

As a community we continue to develop better, more extensible abstractions (with explicit representations) for expressing both our computational intentions and means of achieving them. In an ideal world this knowledge would be widely disseminated to practitioners via education, easy web discovery, advanced programming languages, recommender-style au-

tomation, and strong automated implementation. These abstractions would then enable us to specify computations in terms of the physics and geometry instead of solvers pre-biased towards particular architectures on awkward data structures. Abstractions of implementation advice will enable translation from the specifications to the target architectures, allowing automation to carry out the bulk of the work under the guidance of the modeler, allowing her to control the final result in as little or as much detail as is required to achieve desired performance. Abstractions for mathematics used for various approximations will enable the modeler to make appropriate tradeoffs and allow automation to implement most of the details reliably. Algorithms over various data abstractions can be used to realize effective results in known circumstances, especially in defining and realizing data distributions for complex distributed hardware systems. Hardware abstractions can provide the essentials for targeting actual machine properties, such as multicore, glssimd, data parallel, GPU-like, and FPGA style systems and the communication fabrics that join them, and for automated reasoning about the performance of various proposed implementations.

Automation can be used to enhance and shorten the program construction and testing process. Programs at high levels of abstraction can be refined automatically or with advice down to low-level target machine code, using abstractions and associated implementation knowledge at all levels from physics down to FPGA configuration. This will produce high-performance codes with high confidence in correctness. One could associate abstractions with corresponding tests; then one might also be able to generate tests for the generated program. The availability of formal descriptions of the specification artifacts and the mechanical implementation steps may help the theorem proving community to even prove these implementations are correct.

An intriguing effect of such automation would be the explicit capture of all the implementation decisions made by the modeler or the tools in realizing a particular implementation. Failures in functionality, accuracy, or performance might then be tracked back to specific implementation or specification choices, to be replaced by more effective choices. This will shorten development times by enabling fast experimentation cycles. All this implementation knowledge could be used by mixed-initiative reverse engineering tools to abstract legacy programs, removing design decisions focused on legacy architectures or less than ideal implementation algorithms. This would allow applications to be migrated more quickly to new architectures, allow models to have longer useful lives, and reduce software maintenance costs by helping the modeler understand the design choices in such legacy code.

5.1 Suggested research directions

Many CSE software teams work to incrementally improve productivity and sustainability while developing new software capabilities. However, there is very little effort in the CSE community dedicated to research in methodologies as a first-order concern. Below we describe specific research directions that can benefit from explicit and direct funding.

5.1.1 Productivity through continued knowledge development, capture, and dissemination

As individuals we have the skills and knowledge we acquire by education and exposure. One significant way to advance our individual productivity is to acquire useful knowledge more easily.

This initially suggests the need to create additional useful knowledge. To support CSE, there should be continued effort to develop abstractions of various elements of working systems, at all levels of design and understanding of those systems, e.g. science, mathematics, algorithms, data, computer science, and hardware.

Deeper understanding and clearer descriptions should enable better engineering. Development of these ideas by the community facilitates expert contributions and increases buy-in. Some additional emphasis on making this information known through demonstrations, lectures, teaching, and technical papers is required to help spread the results.

Simply capturing such concepts in traditional ways leaves us limited to our individual learning efforts; as the library of good ideas grows, it in fact becomes increasingly hard to learn enough to be effective. We must find ways to package that knowledge for easy access and easy delivery to practitioners. One approach includes building software libraries that realize the concepts to enable direct reuse. Ideally, such libraries can be used and mixed at multiple levels; this requires flexible interfaces to enable easier integration into targeted software [64]. The difficulty here is that the engineer must be proactive in discovery and use of such libraries to gain benefit. Another more proactive approach is to create automated tools that contain this knowledge and metaknowledge about when and and how to apply it effectively, and make such tools available to practitioners. A third, intermediate approach, is to create a domain-specific environment that streamlines building of software for a specific field of CSE research. Using a product line engineering approach, engineers can create an environment that encompasses the terminology, knowledge, and components appropriate to that field. Scientists can then use that environment to derive software for performing specific experiments based on their decisions concerning models, data sources, analytics, and computing infrastructure to be applied.

Many tools that can provide automated analyses of errors in code have been proposed (e.g, physical units checking) or even implemented commercially (static analysis of pointer errors), yet are not widely used. A variety of tools that capture knowledge about how to generate code from abstractions have been built over the last 20 years (e.g., [3, 20, 71]). A variety of custom code generators for special architectures (CUDA, OpenMP, OpenACC) have been implemented. We need to determine what these tools have in common to make them easier to engineer and apply to application codes. There has been research in general-purpose foundations for such tools [78, 92, 101] and even production tools [23]. Most of these tools are non-interactive; the problem is posed up front and the tool produces an answer with little interaction from the the engineer. Ideal versions of such tools would provide interaction with the developer as they produce code, providing better alternatives and means to analyze tradeoffs between such choices.

Virtually all code generation tools (including our "trusted" compilers) have informal foundations: there is no explicit specification of the semantics of the abstractions or realizations of those abstractions, let alone a formal proof that the realization actually models

the abstraction correctly. While we may test such generators to gain empirical evidence of reliability, we have no theoretical basis for that confidence.

Yet there has been considerable work on formal development processes for software construction, and automation, including theorem proving, leading to applications that can be correct by construction. There have even been some examples of formal development for scientific codes [48, 85]. Correct automated generation could lead to faster development times by virtue of avoiding time spent debugging. And provably correct optimizations could lead to more efficient implementations. Multiple implementation methods could enable portable programs to be correct on arrival. A key problem here is the ability to specify approximations in a formal way, to compute reliable estimates of numerical error produced by an approximation, and to enforce the achievement of error bounds in generated programs.

5.1.2 Productivity through design capture, display, and revision

When working with existing software, we invariably lose the knowledge that went into its construction. Mostly all that remains after a development cycle is a mountain of code, some out-of-date technical documents, and some implicit understanding on the part of the developers, which rapidly fades as they move on to other tasks or career changes. We then pay a very high price in rediscovering that design knowledge when trying to understand, enhance, evolve or tune the software.

We should find ways to capture construction knowledge as the software is being constructed, expose this knowledge to developers and tools that are working with the code, and update this knowledge as the application is modified, so it is available for the next developer.

In particular, we must find ways to associate design knowledge permanently and reliably with code. Stale design knowledge is often worse than none by providing time-wasting red herrings. Of particular importance is the ability of the user to easily inspect the knowledge used in the design, and understand how that knowledge has been used by the code. New user interface techniques to make this practical may be needed. Overwhelming scale may be a particular problem; informal measurements indicate that large applications can contain millions of design choices.

Ensuring this state of affairs permanently is difficult. If an engineer can skip recording such design knowledge it will likely be skipped in the interest of short-term gains, leaving the present situation of decayed design knowledge unchanged. It may be necessary to force the user to provide the design information to get code. One may be able to alleviate this by having such design information provided by smart tools as they make decisions for the engineer. (Kant/Scicomp accomplishes this by recording a series out design-choice outcomes.)

With a product line approach, decisions are an integral element of deriving software as an instance of a preconceived family. Many decisions concerning software capabilities and computing infrastructure are common to the family and can be made by the software engineer but others can only be determined according to each scientist's needs. These decisions are deferred to be made by the scientist and then used to guide mechanical selection and customization of prebuilt components that provide the specific capabilities that the scientist needs for a planned investigation. Changes to decisions are realized as derivation of a new version of the software.

The construction knowledge to capture should include descriptive abstractions from the

problem and solution domains, such as those suggested above and an explicit relationship of those abstractions with realized code.

For example, the ideal design would capture the physics problem to solve in its most abstract form, and follow the reification of that most abstract form via algorithms into code for programming languages designed for special classes of hardware. Some of this knowledge must describe desired performance properties of the target application (e.g., target hardware, throughput, error rates). A rationale for individual design choices should be present to justify the legitimacy of that design choice, and ideally, the design knowledge will include alternative possible implementations and tradeoffs among alternatives to allow easy exploration of alternative realizations.

Somewhat informal approaches have been tried (Conklin/GIBIS, Desclaux/MACS) to capture rationales and qualitative tradeoffs between named design properties. It is unclear how one ensures design capture with these tools or how effective they are. More automated approaches have been tried as "CASE-based reasoning" in the AI-in-design research common in the 1980s. Baxter [18] has provided theory for how this might work for software and has demonstrated the concept in the small scale. We need more research into how to "refactor a design" to achieve this vision.

5.1.3 Productivity through design recovery and modernization

In an ideal world, existing software would come equipped with accurate design knowledge. Our world is not ideal, and the existing successful application code base is so large that it seems impractical to rewrite all that code from scratch to capture the design. What might be possible is to recover design information relevant to the task at hand, justified by the observation that most changes to programs, do not affect most of the program.

What is needed is an approach to rediscover design information from legacy code (C, C++, Fortran, build scripts) focused only on the part of an application where understanding or change is needed in a form ideally compatible with design knowledge capture as outlined above.

Stewart [115, 116] has investigated reverse engineering of scientific codes. Baxter [21] suggests theory for recovering specifications and implementation steps by applying concept-refining transforms backwards in the implementation space. Rich [104] suggests how to rediscover abstractions in conventional computer code from idiomatic patterns with implied dataflows. Baxter [19] has implemented production tools to accomplish the program analysis (PA) vision in factory automation domains, using programmer-supplied explicit idiom descriptions written in source-code terms, including specific design choices where multiple choices are possible and the ideas appear to transfer to scientific computing. [23, 101] provide tools that can accomplish the difficult task of parsing C, C++, and Fortran codes completely and compute the necessary dataflows for abstraction discovery by the PA method. [22] shows how to build practical tools to find potential code abstractions by locating cloned code; such code is cloned because the users had a simple conceptual model of what it did. We note that these techniques may be very useful but are not likely to cover the entire set of abstraction recovery techniques necessary.

With design recovery, programs will become easier to understand and therefore easier to modify/extend/tune for performance.

One might wish to modernize an application by removing legacy constraints (old software architectures, targeted hardware architectural assumptions, legacy algorithms for computing boundary conditions, etc.). Design recovery coupled with design revision should enable one to focus on removing the troublesome design aspect. What we need is "refactoring program designs" to remove undesired constraints, leaving the implementer free to revise the program to the new functionality or performance desired.

5.2 New approaches roadmap

The community needs to continue investigating the abstractions and relationships between the abstractions, that comprise the various design levels used to support the construction of scientific code, including abstractions of the design levels themselves. We assume that development of such ideas will continue as conventionally funded research.

Specific research topics useful in the scientific space would include but not be limited to:

- Defining approximations and automated computation of error bounds
- Automated estimation with integrated empirical measurement of performance, with identification of performance bottlenecks
- Optimization techniques for various optimization goals: design effort, time, space, power, target-hardware, accuracy, or understanding
- Means to specify various kinds of communication (e.g, async, broadcast, reduction) across different architectures (tightly coupled, hierarchical, distributed memory, heterogeneous execution units) and how to partition abstract programs into elements communicating between and operating on those architectural parts

But the results of this research should be explicitly encoded into tools to bring that knowledge in a usable form to the software engineer rather than just a prototype implementation or technical documentation. However, one should not build separate tools for each bit of knowledge and each target programming language, or this knowledge will not be composable for individual applications or across communities. One should choose at best a small number of platforms for encoding this knowledge to aid if not enforce composability.

One choice might be to cast such knowledge as traditional Fortran or even C++ template-based libraries, to the extent that all future applications will be coded in these specific languages. As this is unlikely, a better choice is some framework that can mechanically manipulate arbitrary sets of abstractions/specifications (which happen to include programming languages), in extensible ways. We see the notion of program transformation of abstractions at multiple levels as the unifying vision here, as the Neighbors/Draco framework initially demonstrated; much work in program construction is cast in this form of domain-specific languages and corresponding transformations (e.g., [3, 83]). One element of a product line approach is the definition of a domain-specific notation by which scientists can specify the deferred decisions that determine the software capabilities needed for planned experiments.

It may be that such tools are used to generate libraries to support conventional language, subsuming the traditional approach. We also observe that such frameworks can provide a solid basis for better ("refactoring") IDEs supporting direct programming of applications. We view these as near-term payoffs, which would be accelerated by funding specific IDE support.

We need deeper understanding of the ideas and mechanisms behind such frameworks to ensure an easier capture and use of abstraction and refinement knowledge. Broader experimentation with existing systems such as [23, 71, 78, 101], can be tried to obtain experience. Such experimentation will be necessary to introduce the scientific community to the ideas of automated implementation and the change of perspective from manually coding to controlling the development process with the tools doing the bulk of the coding work. We remark that these systems require big investments (Baxter, Kant, Quinlan, Klint all have decades of investment in each of their respective systems); do we need to build on these platforms or invest in building better replacements? Integration of formal models and proofs should be considered; the program transformation frameworks have a natural place for these elements.

To be effective longer term, these systems must be interactive so that knowledge can be exposed, inspected, applied, and removed with impunity by the engineer; the Rascal system is experimenting with interactive interfaces. Visual modeling tools can aid effective understanding. Such interactive systems will likely make training easier for novices because of the ability to personally experiment with direct feedback.

By unifying abstractions and refinements to lower levels, one should be able to build and maintain product lines of scientific software, with different variants solving different problems on different target architectures. In a product line approach, software engineers create an environment tailored to build software relevant to a specific field of scientific research (reflecting the terminology, computational and experimental techniques, and technology used in that field). That environment is then used by scientific teams to produce software customized to the specific needs of their planned investigations. Many aspects of such environments may be common across multiple fields of research, allowing leveraging of the effort needed to build such environments.

To ensure that these tools lead to long-term sharing, the abstractions they use must be separable from the tool and sharable with at least other instances of the tool, if not other tools. The latter is very difficult; environments for shared tool infrastructures have been tried many times in the past and failed, largely due to disagreement about the precise nature of the data exchanged between the tools. Being able to export the concepts to other users will help ensure uptake. One might consider establishing a central repository for such knowledge.

Long term success in building and maintaining complex software requires we avoid losing our hard-won knowledge about how the applications are designed and built and the assumptions made. While some ideas about design capture and revision have been proposed, none has been seriously tried. Automated tools such program transformation engines provide the promise that some design choices can be easily represented, applied, and stored, but we have no experience actually doing this. It may be that even a different direction is actually required; the informal decision capture of [38] may be useful and it would take less effort to achieve. As the work to try this on realistic code is necessary to prove its value, and that work is likely to be significant, support for building such experiments must be found. Success here would be valuable not only for scientific computing, but for software engineering in every other application area. While this may be high risk, it is a high payoff gamble.

6 Economics of CSE Software Tools

The economics of software tools have proven challenging to understand for users and stakeholders in CSE. In the past, many funding agencies have supported academic and governmental research that produced high-value (but not necessarily high-quality) software as a byproduct of the proposed research, not as a direct aim of the proposal or line item in the budget. In other words, funders did not allocate money specifically for the development and maintenance of software, although software was often developed to support the funded research. In CSE research, existing software is not always a good fit for a particular project or, in the case of commercial software, is prohibitively expensive. The result is that scientists frequently develop their own software, spending as much as 30 percent of their time doing so [57]. The time (usually unfunded) that scientists spend developing software represents a significant portion of overall CSE software economics. There are many dimensions to improve the production, distribution, and consumption of CSE software. This includes approaches to improve how researchers spend their time developing software and methods to provision an infrastructure, and to encourage a culture that supports high-quality software development. These factors are often intertwined.

For example, how does one convince or incentivize academic CSE researchers to adopt best practices in SWE when the main goal of the researcher is to quickly produce research results, not the software used to get those results? A common perception among researchers is that if one-off, custom-developed, poorly architected code can yield research findings in a timely manner, then why spend time creating well-engineered and sustainable code? However, from a CSE software economics perspective, this short-sightedness could haunt a researcher downstream. The researcher often does not appreciate the value of high-quality code developed using best practices in SWE. For example, in the absence of a verification and validation testing framework there is increased risk for code errors that can result in erroneous research findings. Inaccurate research findings may then lead to scientific paper retractions, damage to one's scientific credibility, and potential societal implications with real economic costs (e.g., faulty commercial products). Moreover, if the code is not well engineered and documented new developers will have a difficult time contributing to the code and will introduce inefficiencies through time spent learning the code and refactoring it, if that is even possible. If a graduate student develops the code additional inefficiencies may be introduced, as the amount of time a graduate student is in school is relatively short; thus, new students will require training on a fairly frequent basis. As time goes on, these inefficiencies will amplify, making it increasingly more difficult and time-consuming for developers to determine what the code should accomplish, identify errors in poorly architected code, and refactor the code.

Yet another factor that impacts CSE software economics is the fact that academic budgets are meager when compared to industry budgets. Academic researchers cannot compete with their industry colleagues in attracting and retaining good software talent, creating better infrastructure for development of CSE software, and learning from and leveraging industrial models of software development.

All of these CSE software economic factors, while significant, are not insurmountable. Tremendous opportunity exists for funding agencies, CSE researchers, and stakeholders in academia, industry, and government to work together to identify and implement solutions

to the challenges. In the next section, we will explore some of the proposed solutions that CSESSP workshop participants identified to improve CSE software economics.

6.1 Descriptive examples

The economics of SWE and development is a well-studied field, with seminal work dating back several decades [25, 26]. While these early efforts and subsequent ones have moved the field at-large forward, unique challenges have emerged in the economics of academic and governmental CSE software generation. Among the topics and opportunities unique to CSE software economics include: addressing the current cultural economics of CSE tool development; establishing new academia-industry-government partnership models; identifying new incentives for retaining top software talent; and enhancing the role of funding agencies in each of these efforts.

6.1.1 Addressing the current cultural economics of CSE tool development

As previously stated, in academic CSE research, existing software often is not a good fit for a particular project or, in the case of commercial software, is prohibitively expensive. The result is that scientists often spend significant amounts of time developing their own software, even though few incentives exist for software development in traditional tenure and promotion decision-making processes [66]. In other words, the time that an academic scientist spends developing software is not rewarded or recognized as a significant, independent accomplishment. Instead, tenure and promotion are based primarily on influential research, a successful publication record, the acquisition of grants, and teaching, not on whether one can author good software. Various groups have put forth ideas and approaches to change the academic tenure and promotion process, including the ability to cite software and methods to assign software credit [54, 69, 72, 100, 129]. However, the academic and government entities who support CSE research must also encourage the creation of well-engineered, high-value software and tie the recognition and credit for such efforts directly to the tenure and promotion process.

Another issue that affects the economics of CSE software development is the lack of sustainable code. As discussed in previous chapters, unsustainable software may result from poor documentation, poor code architecture, and/or general lack of SWE best practices. Unsustainable code, by its very definition, is difficult for subsequent developers to contribute to.

The opportunity exists to convince and incentivize CSE researchers to apply SWE best practices from the very start of any research project that entails the development of new software code. The perception must be overcome that the use of one-off, custom-developed, poorly architected code to rapidly produce new research findings may be less cost-effective than an initial investment in time to create well-engineered and sustainable code.

From a CSE software economics perspective, the absence of SWE best practices in academic and governmental CSE software development is short-sighted and could have adverse consequences, especially if the developed software is intended for widespread community adoption. The consequences also may have long-lasting impact on a researcher's career if, for example, poorly architected code yields erroneous research findings. Moreover, while lit-

tle time may need to be invested in the development of one-off, custom-developed CSE code, downstream time devoted to the maintenance of poorly written CSE code may be significant and detract from research time resulting in poor CSE software economics. Some estimates put software maintenance at 60 percent on average of software costs [55]. Also, if the CSE code is not sufficiently and continually tested over the course of its lifespan, errors are more likely to be introduced into the code and subsequently, the research findings. Erroneous research findings may result not just in paper retractions, but in a loss of scientific credibility and potential societal impact with real economic costs (e.g., faulty commercial products) [125]. Efforts such as Software Carpentry [112] are initially needed to train CSE researchers in the early application of SWE best practices to any effort to develop community software code, and subsequent more advanced training is also needed. Such efforts will yield positive effects on both CSE software economics and CSE research.

A more forward-looking approach to address the current cultural economics of CSE tool development is to shift the focus from developing individual tools or applications to building an ecosystem for developing tools and applications [11]. Indeed, CSE tool development infrastructures are not pervasive, but they need to be. For example, consider the R statistical framework [102]. An R software package can serve a group as small as two users or one as large as thousands of users. R provides an excellent model for how a single tool development infrastructure, coupled with strong community buy-in, can efficiently support tool development with any number of users.

6.1.2 Establishing new academia-industry-government partnership models

The economics of CSE software tools can be improved upon by exploring successful partnership models that can inform the broader academic and governmental CSE community. There is a spectrum of successful models in academia, industry, non-profit organizations, and national laboratories. Government projects that know concept-to-supported product lifecycle costs well (not just software) can better inform CSE science projects. The success of these models can be defined to include dimensions of sustainability, open source community development, efficient production of well-architected code, and more. The opportunity exists to focus much more study on how to translate these partnership models to better serve the needs of the CSE community.

Industry, for instance, is well suited to partner with academia on the development of code as a model for scientific software sustainability. The Hierarchical Data Format (HDF) [58] and the Visualization Toolkit (VTK) [126] provide examples of academic software products that were successfully transferred to industry. Industry, in turn, can collaborate with universities to acquire federal funding to support scientific software development. An example is Kitware Inc., which has received funding from the National Institutes of Health (NIH) in collaboration with the University of North Carolina at Chapel Hill to develop scientific software for application in the prediction of stroke outcomes, using 3D models of brain blood vessels [77]. Finally, academia-industry consortiums can be very effective for CSE software development and maintenance. Two successful examples include the iRODS consortium [70] and the Kerberos consortium [75].

Industry supported open source software development models may well inform academic CSE software development. For instance, the estimated value of software contributed to the

open source Fedora Linux project is over $10B [88], even though Red Hat, the company that oversees Fedora's development, hasn't spent anywhere near that much. In addition, an impressive 65 percent of Fedora's code is maintained by over 20,000 volunteers [88], suggesting that open source software economics can work well. While Linux open source software development models are especially large examples, smaller, but successful, CSE open source software examples exist using a variety of partnering models, including OpenFOAM [94] and NAMD [90]. Open source software development requires community building from the onset, and successful open source projects continuously gain new community support over the software lifecycle. Community building must take place concurrently with software development. This is of paramount importance, particularly if the software is intended or anticipated to have a broad community of contributing end users. However, concurrent community building is inconsistently implemented. Workshop participants acknowledged that a formalized approach for implementing concurrent community building would be beneficial to the CSE community. Moreover, if a specific CSE software product is initially developed for the developer's own use, but then later finds a community of interested users, a decision point needs to be identified whereby the developer engages the relevant community concurrently with further software development. Recommendations and support for developers who find themselves in this situation would be useful. A community construct can also be useful in the more efficient and thorough evaluation of software [2].

6.1.3 Economics of commercial software

As mentioned earlier, to be considered successful in delivering financial impact, M&S itself must be performed with productivity in mind. The productivity of M&S depends on people and computing, where computing includes both hardware and software. Analyst or people costs have not been increasing due to the world supply of talent. Dramatic cost reductions in hardware, attributable to Moore's Law, have greatly expanded computation and storage capacities affordable at consistent budgets.

Intuitively, more hardware produces more value when more processors can be tasked with ensembles, executing more (independent) problems concurrently, larger problems than possible with fewer processors, or faster solution turnaround – less "wall clock time" than computing with fewer processors.

Naturally, a user would like to use their software on whatever size computer s/he was able to afford, and the digital replication of the software on 10, 100, 1,000, or even 1,000,000 such processors imparts no material cost on the vendor (if there were, open source would not be possible). The implications of software licensing alternatives are examined in Table 1.

For most of the challenges identified above, commercial vendors are the first choice to supply industry and manufacturing with CSE software. Historically, pricing of commercial software licenses were tied to "seats" (desktop user) or processor counts (on a server). And to the extent such calculations involved small numbers on the hardware side, the software license costs roughly matched changes in hardware even though purchasing was independent in most cases.

But as affordable hardware enabled greatly expanding the size of computing facilities on steady budgets, the software costs aligned with hardware steadily increased, if not acceler-

Software License	Result
$ / (core or node)	Penalizes user's investment in more hardware via "artificial" software limit, impacting uses such as turnaround-time reduction, acceleration of design space exploration and parametric sensitivity analysis.
Single-node license	Problems with small central processing unit (CPU) advantage but bound by memory unable to leverage multiple distributed physical nodes for larger memory pool.
$ / accelerator	Accelerators (graphics processing unit (GPU), Xeon Phi, field-programmable gate array (FPGA)) can result in dramatic energy use and performance advantages. Recognized costs to vendors should not be adversely transferred to users per (core/node) above.
$ / concurrent user	Ideal for predictable, large user base allows full system use. Detrimental when overlapped use "spikes."
$ / registered user	Ideal for small user base continually using software allows full system use. Vendor must rely on users not to exploit.
Unlimited on-demand	Allows full system use, given predictable runtime scheduling. Highly customizable to wide range of user need, costs may be hard to predict.

Table 1: Sample implications of licensing models on application in Industry/Manufacturing

ated, for ardent end-users. These software license costs imposed a pragmatic limit on the potential value of M&S, notwithstanding potentially greatly expanded technical capability of the hardware and software. While a few commercial vendors have introduced innovative pricing structures to mitigate scale-penalizing of their users, the industry has proven averse to change and uncertainty in revenue, particularly for vendors with broad consolidated portfolios.

Such cost limitations have led to the expansion of open source and internally written, curated, and maintained codes in industries such as biology, finance, and oil and gas. While many companies resist noncommercial options, they are compelled to adopt open source or internal development strategies when the cost-scale penalties for requisite functionality become exorbitant. Such software has no usage restrictions, there is in-house expertise on both the algorithms implemented in the software as well as how best to use it, and novel ideas can be implemented to provide potential competitive advantage.

Countering such possible benefits are the need to have a team of developers with the expertise in the physics modeled and the models themselves, as well as how to implement those models efficiently on HPC systems. If the end users are not part of the development team, the software will require special attention to usability and documentation, attributes that are often overlooked. As HPC systems become increasingly complicated, developing efficient software for future architectures requires ever-increasing commitment to such knowledge and capable resources.

6.1.4 Shared public-private journey

Government, academia, and industry all share the ever-increasing difficulty to discover algorithms and create efficient implementations of those algorithms on the rapidly evolving hardware. Developing domain specific languages, libraries and reusable components that can be incorporated into all types of software (provided the right license model) can help contain complexity and improve developer productivity, enabling software developers to port applications to modern platforms while protecting the significant prior investments.

For application developers to adapt such abstractions, we need to develop a sustainable ecosystem which needs to include longer-term commitments to funding and publishing roadmaps for software toolchains (e.g., compilers, runtime containers, services, profilers, debuggers), components (e.g., libraries, microservices, translators, interoperability), documentation and training, reference implementations showcasing the use in real life applications, commitment for optimized implementations from system providers, and most importantly, "critical mass" participation and buy-in from the software developer community in all of the points above.

CSE software paced by the momentum in advanced hardware architectures can be a valuable tool to industry and manufacturing, harnessing the power of computation as both an engine for productivity and efficiency, and as an instrument for insight and discovery. In considering the perspectives highlighted throughout this document, prudent appreciation of commercial subtleties such as legal terms, cost structures, and regulatory obligations should be applied.

6.1.5 Identifying new incentives for retaining top software talent

The retention of top CSE software talent can be challenging for academic and government laboratories that have meager funding when compared to industry colleagues and thus cannot match industry salaries. Alternative, non-monetary incentives must be identified to help academic researchers attract and retain skilled CSE software personnel. CSESSP workshop participants identified the following examples of non-monetary incentives:

- Prestige, knowing that you are contributing to something that is important at a local, national, and/or global scale and public recognition for your contribution;
  - e.g., all U.S. national weather data stored in HDF.
- Greater sense of community involvement, ability to work collaboratively with a community of users.
- Recognition of the greater freedom and creativity available in academic jobs compared to most industry jobs.
- A sense of service, doing something for the broader community, nation, world.
- Job descriptions or course descriptions that use the right language to attract top talent.
  - For example, at the 2014 NSF Cyber-Physical Systems Principal Investigators' Meeting (CPS), it was reported that a massive open online course (MOOC) attracted more interest when advertised as relevant to the "Internet of Things" or "Applied Robotics" than when advertised as "Embedded Software Design" [47].
- A clearer career path, especially for long-term career paths, and titles such as "research

software engineer" [61, 73].

This last point on career paths and titles is critical because the current lack of recognition of software activities in tenure and promotion decision making strongly influences recruitment and retention of top talent in software development. This is true at every career level, including graduate students, postdoctoral fellows, faculty, and staff. As stated previously, academic and government entities who support academic CSE research must also encourage the creation of well-engineered, high-value software and tie the recognition and credit for such efforts directly to the tenure and promotion process in order to attract and retain top CSE software talent.

6.2 An urgent need for productive and sustainable tools

Increasing CSE hardware speed and complexity, software environments and problem sophistication are driving an increase in required developer knowledge and SWE best practices, decreasing the number of developers able to work on CSE grand challenge problems, and increasing the gap between SWE and domain developers doing coding. The CSE user and funding community needs urgent recognition that these economics neither work nor are sustainable, and new models are needed. Furthermore, the CSE community needs to identify and develop improved models of CSE software tool economics that drive hardware architectures and not vice-versa. CSESSP workshop participants, through the discussion topics documented herein, identified the critical activities in the Section 6.3 roadmap to address key software tool economic challenges facing the CSE community.

> Hardware, software and problem complexities are dramatically reducing the number of developers who can effectively use CSE environments to address grand challenge problems. New models are needed to spur development of productive and sustainable tools that expand access to and usability of CSE capabilities.

6.3 Software tools roadmap

The following roadmap presents a strategy to realize opportunities and overcome challenges of CSE software tool economics discussed herein.

- Fund research on how academic and governmental CSE researchers spend their time developing and maintaining software throughout their careers (see Section 4 for additional discussion on this). Correlate this to CSE software tool economics demonstrating a range of approaches and where and how improved efficiencies are achieved.
- Fund research on how to improve upon academic and governmental CSE SWE. There is need for a holistic view of what constitutes "optimal" economics for software tools, e.g. hardware, software, partnerships, metrics, funding, social, community, incentives, and more.

- Create many more SWE training and education opportunities for CSE career-oriented students and CSE researchers, especially early career CSE researchers. In these courses, highlight how CSE software tool economics are impacted by the SWE choices made throughout their careers.
- Position the CSE community to require federal funding agencies acknowledge the importance of funding software maintenance concomitant with how hardware maintenance is funded.
- Position the CSE community to require federal funding agencies to support long-term (e.g. more than 5 years) software development and maintenance of high-value CSE community software and infrastructure.
- Position the CSE community to require academic and governmental entities recognize the creation of high-value CSE software with good SWE as a direct factor in the tenure and promotion process concomitant with how one's publications and research record are considered.
- Create more career opportunities and tangible non-monetary incentives to attract and retain CSE research software developers and research software engineers.
- Create a body-of-knowledge on how to instantiate, grow, and sustain successful industry, academia, and government partnering model (e.g. consortiums) to sustain high-value CSE community codes.
- Create a body-of-knowledge on how to instantiate, grow, and sustain successful CSE communities (e.g. open source communities) with a vibrant community of contributors. An improved understanding is needed on how to best build CSE community while developing CSE software.
- Encourage funding agencies to fund the creation of more CSE infrastructures for creating CSE tools and help make CSE researchers more efficient in tool generation. For example, in R&D software tool concept-to-supported-product lifecycle models, don't build a tool at a time, rather build an infrastructure for building tools to improve CSE software tool development economics and efficiencies.

7 Social Sciences Applied to CSE Software Systems

Software development is a social challenge, as well as a technical one. The creation and maintenance of software involves constant interaction between people across domains, roles, and communities. Since most projects involve more than one person, and even single-person projects usually create or use software dependencies, social science research can be beneficial in addressing issues across the whole software lifecycle, from requirements gathering to development and maintenance to end-of-life.

CSE software development involves particular social groups and dynamics: the main stakeholders are researchers for whom research outcomes are of highest priority, and software is often viewed as a means to an end rather than the goal itself; these researchers are not professional software developers. There is a body of existing research studying CSE social dynamics (e.g., [44, 109, 110]), and discussion of that work has impacted the thinking of code developers and project managers in CSE, leading to social practices widely thought to support more effective software development. For example, studies which looked at the

problems in CSE projects emerging from differences in priorities and values ascribed to software development show that making the stakeholders aware of the reasons for these issues can help resolve them.

From this kind of research it's clear that understanding social issues is important and studying the whole ecosystem rather than only focusing on the technical side of CSE is needed. Such studies may involve looking into various resources related to software development and its lifecycle. For example, the Agile Manifesto (http://www.agilemanifesto.org) is really a social document. Furthermore, a number of artifacts (including software) can be viewed and studied as social documents that tell a story about the authors (project, process) and the purpose for which they were created.

The social science community has developed methodologies and approaches that can be applied to effectively research these issues. Bringing social scientists who are aware of these methodologies and approaches into CSE work can help researchers and other stakeholders work together better and do better work. Eventually this leads to better (more robust, more maintainable, better fitting user needs, etc.) software in CSE.

Social scientists can make contributions by studying the differences between the developers, the domain scientists, and the general population. For example, the approaches and methodologies used for testing by professional software developers (with background in generic, rather than domain-specific software development) and domain scientists developing software can vary significantly. There is a rationale behind both approaches and understanding it rather than simply judging it can improve communication and exchange of knowledge in the area. Social scientists can help us understand how to achieve it and implement it.

While many social scientists have studied the general area of SWE, the lessons from this work are not well known in the general software development community or the narrower CSE software development community. In addition, it would be beneficial to understand what the differences are between CSE vs the broader SWE, some of which are technical but many of which are social. SWE can be understood as a broad set of (generic) guidelines, methodologies, tools and their applications necessary to build robust and maintainable software. CSE covers a range of scientific domains strongly grounded in computational background and typically requiring at least some software development for advancing research. Studying the difference between the two spurs questions like, Which methodologies, tools, etc. work best in each area (SWE vs CSE)? What adaptations of SWE tools to CSE are working and why? What social lessons from SWE apply to CSE and what needs to be changed?

Understanding and influencing group norms is key to improving software teams. Software development is often a team activity, and understanding the social aspects of how teams work best is important. Expected behaviors are important, such as "making sure teams had clear goals and creating a culture of dependability." But the most important norm is psychological safety, "a shared belief held by members of a team that the team is safe for interpersonal risk taking."

Understanding and influencing group norms is key to improving software teams. Software development is often a team activity, and understanding the social aspects of how teams work best is important to today's biggest companies, as Charles Duhigg wrote in The New York Times, "In Silicon Valley, software engineers are encouraged to work together, in part because studies show that groups tend to innovate faster, see mistakes more quickly and find better solutions to problems. Studies also show that people working in teams tend to achieve better results and report higher job satisfaction. Within companies and conglomerates, as well as in government agencies and schools, teams are now the fundamental unit of organization. If a company wants to outstrip its competitors, it needs to influence not only how people work but also how they work together." [43]

Duhigg further described how Google, believing that better teams would be more productive, studied how to build better teams, "The company's top executives long believed that building the best teams meant combining the best people. [But] after looking at over a hundred groups for more than a year, ... researchers concluded that understanding and influencing group norms were the keys to improving Google's teams." [43]

They found that some expected behaviors were important, such as "making sure teams had clear goals and creating a culture of dependability." [43] But they found that the most important norm was psychological safety, "a shared belief held by members of a team that the team is safe for interpersonal risk taking." [45]

7.1 CSE software is an ecosystem

Software development constitutes an ecosystem of interacting people and projects, particularly in CSE. This overall ecosystem has no overarching management structure. However, many individual projects do have specific developer roles of integrator, reviewer, and tester. While this may be informal in small projects, there is a somewhat standard management framework and set of practices that are often followed in a CSE software development project, since many people are members of more than one project, and practices that work well can be informally communicated between people and thus across projects. However, there is a need to more formally understand how communities act and interact [35] as well as to identify what makes communities successful [80].

Another way of saying this is that CSE software is built to solve problems at a variety of physical and temporal scales, across a variety of domains, and across a range of hardware types. Even for solving simple CSE problems, multiple elements of software interact either through software dependencies or software reuse. For more complex problems, multiple elements of CSE software are needed to work across scales, domains, and hardware systems.

One example of this is changes in a commonly used library. The library developers (producers) may want to concentrate their development effort on a new version, but may face resistance if there is a need for changes in user (consumer) applications, such as an API or dependency change, for example, moving from Python 2 to Python 3. The activities of the producers and consumers need to be clearly communicated and, ideally, they should also be coordinated.

More generally, different groups or people in different roles within the same group can have differing understandings of data, metadata, and coding structures. These differing understandings, if not resolved, can create what Edwards et al. [46] call "friction," and this

friction can hinder interdisciplinary activities.

Today, the coordination of activities and the resolution of differing understandings across the CSE ecosystem is mostly done in an ad hoc manner, though there are both bottom-up exceptions where communities come together at conferences and through virtual organizations for the purpose of coordination, and top-down exceptions where funding agencies or companies coordinate development of software with the goals of solving specific problems or classes of problems. We need to understand how to do this more systematically to build an overall ecosystem that's accessible to domain scientists and sustainable. Social sciences can help with this.

7.2 CSE software development is a set of social communities

CSE software development involves groups. Many single projects are developed by more than one person, and CSE software as an larger ecosystem involves multiple development teams. Thus social interactions and practices of these groups and communities are important to understand and improve.

7.2.1 Team dynamics and culture

In the context of collaboration and team management, culture is understood as a set of norms, rules, practices, shared priorities, and goals. Each working environment has its own culture. In generic software development (as opposed to CSE), the culture is such that the full focus and priority is given to the software. For enterprise software, the culture is influenced by the size of the team and the pressures around meeting customer deadlines and shipping the product.

In CSE software development, the culture influencing the process and the product is determined by a set of characteristics that are significantly different from that of a larger software enterprise. Typically CSE software is developed by the small team of a faculty PI (or two) and a relatively steady stream of grad students and postdocs who contribute to the code. CSE certainly has large code efforts, but there are hundreds of these small teams. Understanding the social dynamics and culture of these teams is important.

CSE codebases tend to live a very long time, even as they are regularly modified to prepare for the next experiment. The tension between long-lived codes and the flexibility and speed required by computational research to produce results can mean that sustainability investments in the code are compromised, even though a team could be more productive in the long run if it spent time on sustainability. Social sciences could be helpful in studying the reward system in CSE software efforts and help the community build incentives that encourage better productivity and sustainability.

7.2.2 Individual and intra-team social skills

Formal efforts to teach social skills, such as classes on social styles, can help people understand each other's background, motivations, and goals, and help them better work together. Communication can be thus improved, which leads to better collaboration. We want to reduce social friction but retain the desired intellectual friction, to allow people to better

work together while still considering many possible ideas and implementations. By creating an inclusive environment we can increase diversity. And having a better understanding of the motivations and goals of the team members can help overcome resistance to change.

At both the individual and team levels (see Figure 6 in [35]), better social skills help in listening and in recognizing the knowledge of experts, which improves goal orientation creating a safe environment in which to contribute. These skills empower team members and the community at-large to make significant contributions and to evangelize broader adoption. Teams become better functioning, with better decision making, stronger and more effective leadership, improved coordination and collaboration, and better long-term knowledge management.

In particular, as software projects get larger it's increasingly likely that the teams will include software professionals in addition to scientists who have software development skills. These software professionals will often have more experience with general technical and social methodologies and social lessons because they recognize that there is more to software development than just writing code.

CSE teams, particularly large ones, have the tendency to be highly diverse in terms of the disciplinary communities involved within them. It has been shown that individuals have a very hard time talking about technical ideas in a manner that is understandable to people who come from different domains of expertise [10, 24, 27, 28, 41]. Understanding how to effectively construct a shared language to negotiate the direction of the code (between the developers and the users) is likely to be a central component in CSE success.

7.2.3 Individual response to community culture

If developers are rewarded for others using or modifying their code, it will change the behavior in how they work with others and how they share their code; just making it open source is not enough. However, it should be noted that in academia and even in many labs, published peer reviewed papers are the primary metric for success, while software artifacts developed are seen as by-products. This change in perspective will not happen unless the social behavior of management teams changes to incentivize software development in CSE by rewarding researchers for contributing to software development.

7.2.4 Inter-team communities and practices

Code is developed within a particular context, which is created by people and culture. It is thus important to understand and recognize the (social) context when studying the process and code. This can be done effectively by social scientists. To make a project sustainable, we need the people who work on it to continue to want to do so, need to grow community, and make it easy and desirable for people to join and contribute. Joseph Porcelli's (GovDelivery) model of motivations (for people) and friction (for projects) is related: his principle is that when the motivations outweigh the frictions, people contribute to the projects, and successful community project managers work at both increasing the motivations and reducing the frictions.

7.2.5 The role of tools

Development environments can enhance collaboration and coordination. For example, "social coding" sites like GitHub have become wildly popular and successful perhaps because of the transparency of activities and communications these sites afford, which allows for successful self-coordination [36]. A surprising amount of improvement can be accomplished with minimal interpersonal interdependencies [65]. It should be noted that software development within the scientific community increasingly happens remotely (fewer face-to-face interactions), including the use of text, video, voice, and other collaboration tools to manage distributed teams.

7.3 Social sciences roadmap

The CSE community should put effort into examining sociotechnical aspects in two parallel streams:

1. Within ecosystem collaborations
2. With regards to individuals and team interactions

Key social relationships need to be understood in the context of the technical dependencies they help manage. For example, applications that use particular frameworks place the developers of those applications and frameworks in a producer/consumer relationship that requires specific kinds of communication and accommodation. Managing these relationships is quite important, since success on the social side enables sharing of resources (such as libraries, frameworks, and other technical and social contributions by the user community), greatly enhancing productivity. Note that ideally these relationship could transition from the producer/consumer model to become collaborations, where the consumers become contributors in what were formerly the producers' products, by documenting, fixing bugs, supporting, etc.

This work should comprise the following steps:

- Identifying the various kinds of technical dependencies in CSE software that is developed by different groups
- Describing what kinds of social interactions, communication channels, etc., are needed to manage these relationships
- Identifying common needs that could be met by enlarging pools of shared resources
- Studying, identifying, and recommending tools to support collaboration and communities of practice
- Defining how to build communities around shared resources to enhance and maintain them.

We need to better understand and define/structure the role of social scientists in software (CSE) projects. A challenge that may possibly occur when introducing social scientists into CSE environments is that if the social scientists aren't respected by the other members of the CSE teams, they can't contribute effectively. It should be noted that engaging social scientists can be beneficial for both parties: social researchers and the groups studied by them. We need to build, document, and publicize arguments and evidence that support the inclusion of social scientists.

Problems in science, technology, and software are very interesting to social scientists and provide them with a lot of valuable research material and data. If CSE groups are resistant to having social scientists around, it perhaps would be possible to emphasize the fact that social scientists are there to do their research and not to solve potential problems. It is likely that over time the social scientists will develop understanding, which will indeed be very useful.

We also need to understand and define when it would be better to engage a social science researcher and when it would be more appropriate to include a professional social scientist, for example, a process consultant, who can advise on project organization, management, and planning. It is important to note that while social science is likely quite helpful in understanding teams and planning interventions, understanding teams and planning interventions is not necessarily (and not always) social science research. The CSE community and social science professionals can work together in mutually beneficial ways. The CSE community would benefit from practical, concrete advice and professional social scientists could further their careers by focusing on CSE teams.

All the steps listed above will contribute to addressing a challenge related to the fact that some of the aspects of improving CSE though social science studies and research are difficult to measure. For example, there is no consent as to how programmer's productivity can be measured. In fact, as in many cases of researching human interaction, it is possible to make approximations and include a number of factors, but we need to accept that many outcomes and solutions will be very much context dependent. We may need to plan separate steps to address this issue.

7.4 Possible outcomes

There are at least two possible outcomes from applying social sciences to CSE systems. The first is knowledge and better understanding of the context, the culture, the priorities and practices. In particular, we seek:

1. Better understanding of the human activity that is CSE software development
2. Models that describe how CSE software teams interact and that can be used to reason about how CSE software efforts can be more productivity and sustainable

Second, we want to apply this knowledge. In particular, we would like to:

1. Propose strategies for making CSE software teams more aware of the progress in the broader SWE community
2. Propose strategies for naturally increasing the desire to know more about better SWE in the CSE community

The latter could be done through improved training for both professional and non-professional developers in best practices for software development and digital data skills, using the Software and Data Carpentry activities.

Finally, it is important to show results of the change and the impact that we are making. The challenge is to collect sufficient and convincing evidence, in part by developing metrics to gauge the success of applying the gained knowledge in practice. For instance, we could measure commits by external contributors to CSE projects, the number of issues/tickets filed

by users and answers provided by the developers. Having more documentation, better test coverage, and deployment automation is another indicator as well as propagation of improved workflows which include pull requests, git-flow and release staging (dev, alpha, beta, prod). Increased use of external tools, e.g. continuous integration (travis, snap, etc.), static analysis (codeclimate), documentation generation (readthedocs), and package repositories (docker hub, vagrant cloud, pypi) also allows to measure the impact. Another example of the metrics is related more to social aspects such as increased interaction through local and international meetups, conferences, and BOFs as well as local and international forums. Finally there are installation counts: science-as-a-service sites, behind the firewall installs, etc.

8 Workforce Needs for Sustainable Software for Sciences

> "For America's true competitiveness rests in the efficiency with which the nation reallocates productive resources, including the adaptability of its workforce."
> – Work, Council of Competitiveness, 2016

The ability of any human activity to sustain itself depends on developing and maintaining a process to train a workforce capable of supporting that activity. The relevant skills that are required need careful specification. For sustainable CSE software, the entry level is the skillset of Computational Literacy, defined as the ability to harness the power of computing by both the composition of relevant problems into forms amenable to computation and the comprehension of the computed results (including error factors, limitations, etc.). Further, Computational Proficiency is defined as the additional ability to employ computational assets as a strategic asset, encompassing an understanding of high performance hardware, systems modeling and integration, model verification, validation and uncertainty quantification, data analysis and synthesis, scalable software architectures and algorithms, and communicating through digital media. These skills are essential for data and model driven scientific discovery and engineering innovation.[2]

The practical embodiment of these skills is software tools, the development of which is "as important to modern scientific research as telescopes and test tubes" [112]. Software transcends these traditional tools of science in that it can be both the tool by which the science is done and also a reusable and self-contained means of dissemination. However, the workforce that creates this vital infrastructure comprises largely of "self-taught" scientists. While there are some ad hoc attempts at providing training and characterizing the needed skill, [128] professional high quality training and education resources are few. A survey of physics doctorates and post-doctoral fellows subsequently employed in the public sector

[2]While the notion of "computational thinking" advocated by J. Wing and others (e.g. http://www.cs4fn.org/computationalthinking) has some overlaps with these ideas, the focus here is on a set of core professional skills rather than on pedagogy and ontology.

indicates that more than three quarters of them list programming and modeling as core skills in their current employment [5].

It is clear that scientific software is a critical part of the infrastructure for science whether it facilitates the discovery of gravitational waves or new materials for energy storage. It follows then that the development of a well-trained and professional talent base of workers with skills in developing this software is likely to have a large impact on the infrastructure that enables science. This workforce can dramatically enhance both the quality of discovery science and also provide competitive advantage to innovation driven U.S. industry.

In this chapter, we will outline the many opportunities and challenges in developing a workforce capable of creating and sustaining the scientific software infrastructure needed for discovery and innovation.

8.1 Scale and scope of problem

There appears to be only sparse quantitative data on actual need of personnel required to develop and sustain the software infrastructure needed for discovery science, defense and security related research, and industrial innovation infrastructure. The reasons for this are several but the lack of data is compounded due to ad hoc development processes and the lack of formal planning and monitoring of such work in a manner that has been more common in commercial software development. There exists plenty of anecdotal data on a deficit of skilled personnel with the required combination of disciplinary knowledge and software skills.

We attempt to collate some of the readily available data but highlight that most data we cite is of much coarser granularity and usually gathered with different goals.

(a) Bureau of Labor Statistics (BLS) data: The employment category of "Software Developers, Applications" is listed as having 686,470 employed at an average salary of $99,530 [29]. A cross-cut of this with the "Science and research development services" industry sector reveals that 14,980 of these at an average salary of $103,410 are possibly engaged in developing software for the sciences. An additional 4,570 are listed as computer programmers engaged in this industry sector. Further confounding any inference is the listing of 570 mathematicians (average salary $119,830), 4,100 statisticians (average salary $95,140) who are likely part-time or full time contributors. A significant fraction of a long list of science and engineering professionals (physics, chemistry, engineering) employed in this industry are also likely engaged in software development. The BLS goes on to project that the employment of application software developers is likely "to grow 17 percent from 2014 to 2024, much faster than the average for all occupations." [29] The training pipeline for producing such professionals is unlikely to be able to match this need if this indeed is the true demand growth. Moreover, the slow response timelines of much of the science research establishment that is government funded (through grants and contract mechanisms that cannot pivot and change rapidly) points to an impending crisis. As a counterpoint, one must also note that the category of "Computer Programmer" is projected for an 8 percent decline due to increased offshore outsourcing, an option that is impractical for many of these needs.

(b) Taulbee Report from CRA: The annual Taulbee report [120] from the Computing Research Association (CRA) points to the strong sustained growth of undergraduate enrollment

while doctoral and masters production rates are relatively flat. The doctoral degrees (the primary source of a large part of the talent pool from which computational scientists are recruited) awarded actually declined by 4.4 percent over 2013 in 2014 but this was offset by increased enrollment of 3.6 percent. The undergraduate degrees awarded went up 12 percent while the enrollment went up 17.4 percent. Such strong enrollment growth in undergraduate computer science increases the potential talent pool. Data from the American Institute of Physics also indicates a healthy growth in physics BS degrees (many with minors in mathematics) averaging over 5 percent per year over the last 15 years [6]. However, specific strategies are needed to attract sufficient numbers of these graduates into the computational sciences and scientific software. Existing programs like the Computational Sciences Graduate Fellowship (CSGF) [81] are successful, but the scale of the need is much larger than the small supply from these.

There is a need for additional data gathering efforts on specific requirements. Such efforts should focus on information from hiring and operational managers at the different national laboratories, industry research agencies, independent vendors of major science tools, and Department of Defense facilities. Principal investigators of major academic research groups that develop and provide software tools also need to be surveyed on these aspects. A forthcoming study by the Council of Competitiveness on workforce readiness [34] provides some data on the level of computational literacy and proficiency.

The upcoming need for major restructuring of most software tools for science to take advantage of two computing transformations: the ubiquitous availability and use of "Big Data", and the complex architecture of the next generation of high-end computing and data analysis hardware will make the workforce needs more acute. It is necessary to examine the anticipated complexity of scientific software for this environment, where the algorithmic (complex computational schemes, data management, and computing for exascale) and software infrastructure are both in flux.

8.2 Gaps in current training processes

Two hypotheses have been proposed for the current sources and training modalities for this workforce. The first one proposes that most of the current workforce has not been formally trained in software but has migrated from the domain sciences. The second one postulates that most computer science majors are not interested in computational science and therefore, traditional computer science curricula will not produce graduates capable of developing/supporting CSE applications. Anecdotal evidence supports both hypotheses. A general observation made by several hiring managers is that in the current training processes there is a 2-year gap between hiring and a productive engineer. These hypotheses need verification and validation with specific data where possible.

A careful delineation of the skills needed for the next generation of CSE developers is needed [6, 40, 112]. The SIAM CSE (draft) report[108] emphasizes cross-disciplinary graduate program learning outcomes for CSE. Clearly these comprise the traditional skills of programming and software engineering, modeling, applied mathematics, and numerical analysis, but we need to add other skills to this mix. The increasing role of data-driven sciences indicates that we need to add data sciences to this mix. Software project management

is increasingly essential for large scale and distributed development by collaborating teams.

While higher learning programs can improve the supply of entry-level technical skills, industry adoption of advanced computational methods will stall until executive-level decision-makers are also computationally literate. Thus, supplemental training programs (perhaps broadly localized through community colleges) should also be assessed for retraining the existing workforce and aim not only at technical positions but also at leadership.

8.3 Innovative and emerging solutions

These workforce needs are being addressed to a certain extent by "multiscale" training responses. These include focused training like those in the software and data carpentry activities [127]. Others like "Google Summer of Code", and the XSEDE and Argonne Leadership Computing Facility annual workshops also take direct aim at this problem. However, much more needs to be done. Curriculum delivery platforms like MOOCs are potentially transformative here, but serious attempts are not yet in place (see for e.g. [124] on programming linear algebra applications).

A significant positive development in contributing to closing this gap is the number of new graduate programs in computational science, and more recently in data science, that have begun in the last two decades. Graduates from these programs are cross-trained in domain and computational/data sciences and are usually capable of and appreciative of the need for developing good software tools. The 2012 survey [40] of 59 such programs shows an average enrollment of 26.7 students in MS programs and 21.7 in PhD programs, which is a reasonable, if not abundant, supply of talent. These graduates are as defined computationally proficient and appear to be highly sought after by national laboratories, academia, and private industry.

8.4 Role of universities, industry and funding agencies roadmap

Once the dimensions of the problem have been established, what actions can different stakeholders—universities, industry, and funding agencies—take? Let us first consider the role of funding agencies. While mission agencies and laboratories are among the primary beneficiaries of this workforce, they are not traditionally the primary agencies investing in workforce development. The NSF traditionally bears some of that role, but its efforts are diffuse and more focused on developing a general STEM (science, technology, engineering and math) workforce for academic research and industry needs. Will it serve the needs of other agencies to create centers focused on sustainable software development at universities who have a primary mandate to educate? The three-pronged scientific software elements, scientific software integration and software institutes program started by NSF has seeded a number of efforts in the last few years that provide or develop a set of robust tools and, more importantly, as a byproduct, train a set of researchers/professionals that can support some of this workforce development need. However, for this to really have an impact, this role needs to be formalized and sustained.

The traditional training venues of universities and academics are not currently incentivized to practice and hence "teach" software skills, since there is meager reward in the academic system for producing high-quality and reusable software tools. Furthermore, there

exist no metrics or common understanding of how such contributions may be valued. A career path for any faculty who do provide such training has traditionally been unclear. Recent CAREER awards, notably from the NSF Advanced Computing Infrastructure Division, support the intent of these academics to develop reusable and robust software for the sciences and increase the acceptability of such activity in academic circles. The importance of this training and this career path needs to be further reinforced and, to really have an impact, these awards need to be scaled. As of this writing, only a very small fraction of young investigator awards in computational or data science come with an incentive to develop robust and sustainable software. A significant scale-up of these young investigator awards in ACI and incentivizing all computational and data science awards to produce robust, reusable and sustainable software tools will have the impact of both greatly increasing the workforce and the long term creation of a digital infrastructure for science.

To stimulate focus on such skills, perhaps academic institution ranking agencies such as Barron's or U.S. News and World Report might be guided by a national call and clarification of skillsets to introduce a category assessing and ranking universities on the preparedness of graduates for job needs in computational literacy. The listing by most PhDs of these skills as of use and value in their current jobs provides graduate curriculum developers incentive to do so. Accreditation processes provide another important venue for incentivizing the provision of such training. ABET, the accreditation organization for most undergraduate engineering and computing related programs in the U.S. uses an accreditation process that ensures that universities are taking input from employers and training their graduates in the skills that are required in the workplace [1]. Recent ABET focus on workplace skills can be logically expanded to include robust software development. Systematic *engagement with the ABET and other accreditation and ranking agencies to highlight the importance of these skills* is necessary. However, graduate education does not provide such a standards based approach since each program is highly individualized and driven by tradition.

9 CSE Software in Industry & Manufacturing

Advances in computing technology propelling affordable collection, storage and analysis of data at massive scale are transforming industry from how products are designed and manufactured to the connectivity with customers. The data-driven enterprise will seek to capitalize on increasingly sophisticated software in both the operation of businesses and the creation of competitive goods, such as employing machine intelligence as a collaborator to tame voluminous data or embedding sensing and connectivity in products themselves. Ultimately, however, value vs. cost drives adoption within the commercial enterprise.

It will become competitively necessary for industrial/manufacturing[3] companies to employ a variety of CSE software across the lifecycle of their products and services, most heavily leveraging numerical (digital) methods where economically advantageous over traditional, empirical (physical) methods. Simulation, modeling and analytics will drive competitive and innovative designs while subsequently reducing time, costs and risks in supply chain

[3]While commercial users of CSE software includes manufacturing entities as found in finance, insurance, education, information services, entertainment, etc., we focus the industry & manufacturing section on enterprises producing commercial and consumer physical goods and related services.

logistics, manufacturing, assembly, packaging, distribution, operation, maintenance and servicing. Our discussion will reference M&S as the predominant application of CSE software under consideration.

9.1 Opportunities for CSE software to advance industry & manufacturing

Many of the opportunities described in this section rely upon overcoming significant challenges addressed more directly in the section to follow. But in preface, ***three factors merit emphasis as typical prerequisites for CSE software adoption by commercial industry & manufacturing:***

1. *Legal clearance* (including intellectual property protection, indemnity, and licensing terms)
2. *Validation* (more so than scientific studies, products have liability consequences)
3. *Affordable professional support* (responsiveness to business needs but also not prohibitively expensive in context of use)

9.1.1 Financial impact

It should go without saying that success in industry & manufacturing requires careful attention to costs and returns. A Department of Energy-commissioned study[4] of the return on investment (ROI) examined macroeconomic and innovation impact of HPC-driven CSE, but to date an industry-acceptable microeconomic ROI has proven challenging for industrial advocates to develop. This is because by strict financial accounting definition, ROI requires the return be exclusively attributable to the investment in the M&S activities themselves. But by its very nature, modeling, simulation and analytics are methods employed by more broadly-scoped teams, tasked with product improvement, manufacturing, servicing or new product development. While M&S does not itself generate revenue as such, companies at least qualitatively value the benefits listed in Table 2 and, as commercial enterprises, seek a quantifiable and accepted measure of that value to justify and size further investment in the people, software, and hardware necessary for its practice.

There is thus an opportunity to explore how public policies or standards may encourage adoption of CSE software in industry & manufacturing through clarification of financial impact. For example, Proctor & Gamble adopted a practice of calculating a financial impact of returns divided by costs as follows: returns were counted if M&S was necessary for the financial gain (in new product launch, cost savings, productivity improvement, capital avoidance for example, using an existing asset to generate new revenue) and all costs (including M&S people, software, hardware and external services) are included.

9.1.2 Engines of productivity

The most straightforward application of M&S is in modeling a product or process that already exists, but then leveraging computational horsepower toward super-human outcomes.

[4]Economic Model For a Return on Investment Analysis of United States Government High Performance Computing (HPC) Research and Development (R&D) Investment, IDC Research, (Sep, 2013)

	Engines of Productivity	Instruments for Insight
Increase	Return on labor Profit margins Supply chain & distribution efficiency Design exploration Agility to seize opportunities	Product diversity & novelty Yields & production capacity Data-driven decisions Trade-off analysis Perception of previously unseen
Decrease	Costs of overhead & rework Time to market Equipment downtime Response time to fix problems	Operational exposure Uncertainty & risk Contradictions Noise obscuring main effect
Methods of Practice	Automation of repetitive tasks/tests Faster than real time simulation or analysis Digitally replicate studied resources Concurrent studies on parallel system	Model unmeasurable effects Isolate effects in complex interactions Observe without physical interference "Big Data" analysis and synthesis

Table 2: Potential CSE software modeling and simulation benefits to industry and manufacturing

Unfortunately, the productivity results stemming from such analyses are sometimes attached to significant prior investments, and the M&S impact gets an honorable mention but is not espoused as "critical."

Financial impact becomes more clear when these capabilities are engaged earlier in a commercial process (before a prototype, before a process change is implemented) to perform virtual trial and error of options. This yields "innovation productivity" by avoiding capital expenses and investment of time in physical tests, prototypes, clinical trials, etc. While these will likely be needed at a later stage, the confidence in the result of said tests passing will be much improved. The downside of virtual trial & error is you have to already have thought of the idea to test it, and it is not advantageous compared to traditional methods if computational methods cost more time and/or money. So the very productivity of the M&S process plays into the equation.

> At Procter & Gamble, this "Virtual Trial & Error" is the bulk of our Financial Impact. A typical example can be in something as simple as a plastic bottle. The plastic bottle needs to meet certain strength requirements for manufacturing, shipping, and home use. When you make billions of them, you cannot "overdesign" for strength, but if they fall off the shelf "cleanup on aisle 7" can be something a grocer or a home owner does not want, especially if they feel that you make the product "weak". You can take 12 weeks and build $50,000 molds

for every iteration, or you can do all the testing virtually. Every year, P&G tests over 100,000 virtual tests on different containers. Some you see everyday in the store and some you will never see because they would not work in the real world. – Tom Lange (P&G - Retired)

9.1.3 Instruments for insight

The ultimate way M&S can impact a business is by enabling discovery. Analysis-led discovery is often associated with products, formulas, manufacturing systems, supply configurations that you would have never thought to test. M&S can enable exploration of a high dimensional search space to seek an optimization or configuration you would not likely have considered. As many products and processes push the cutting edge of technology, the ability of traditional methods to observe and measure critical effects has reached its limit. From machine perception augmenting our human senses—filtering complexity or making vast data tractable by focusing attention to simply measuring physics at scales of time or ranges (e.g., temperature) infeasible through empirical methods, M&S become essential "instruments for insight". Unlike productivity impacts, where credit must be shared—these synthetic discoveries can often be wholly attributed to computational modeling.

9.1.4 Software ecosystem synergy

The discussion throughout this report clearly illustrates many aspects of how the government will hold a position of leadership in CSE Software. The learnings, discoveries, developments and data flourishing from that advancement likewise will endow industry with the fruits of those capabilities and experts' knowledge, providing critical software components such as:

- application development tools (e.g., compilers, profilers, frameworks)
- execution environments (e.g., operating systems, kernels, containers)
- component implementations (e.g., microservices, libraries, algorithms & data structures)
- secure elastic deployment infrastructure (e.g., hardened cloud, cyber security protocols)
- "gold standard" reference data (e.g., codified physical test data repositories for VVUQ)
- ultra-high fidelity reference models at leadership computing facilities (e.g., human liver)

These are not even limited to end-users of CSE Software, as commercial software vendors can leverage such assets to modernize their own codes—improving scalability, performance, interoperability and reliability while reducing associated costs and risks. In fact, an additional topic for further consideration should be acceptable pathways to commercialization and professional support of these codes themselves to better secure their sustainability.

9.1.5 Software scalability competitiveness

The topics of hardware access and software costs will be addressed in more detail as challenges in the following section. Considering opportunities, however, one market dynamic of commercial software that merits examination is the reluctance of established software vendors to keep pace with advances in the underlying hardware or application of novel algorithms enabled by emerging architectures. Industry's reliance upon commercial codes is well-established and defines that very market. Many commercial software vendors of CSE

codes have been consolidated under larger entities over the past decade and in doing so are compelled by market economics to avoid risks and costs associated with porting to cutting edge hardware or serving the niche of lead users demanding extreme scalability.

Additionally, some commercial licenses expressly forbid publication of performance benchmarking or even in some cases the benchmarking itself. Such protections impede due diligence in fairly evaluating the merits of alternatives, and would hamper the prospect for periodically publishing competitive capability assessments by a neutral party.

Therein lies an opportunity to challenge these commercial vendors in more aggressively adopting the components of the software ecosystem described previously. While the Top500.org's ranking of hardware has been the source of controversy[5], it has undeniably motivated commercial behaviors. What sort of scalability ranking of various representative problems for CSE software (e.g., fluid-solid interaction, fracture mechanics, turbulence, etc.) might be devised to demonstrate comparative capabilities in handling extreme problems? Such a list published with the periodicity of Top500 would both advise end-users seeking to expand the scale of their problems and offer software vendors a potential tool for marketing improvements to their solver's capabilities (and thus justify the investment to do so). How might such an evaluation be governed? Hardware vendors would also participate insofar as they could provide the underlying system access to prove out the scalability and performance for the target benchmarks.

The compelling aspects of the *cloud computing* business model that has evolved from on-demand hosted compute infrastructures include reduced costs due to multi-tenant driven higher utilization, simplification of IT operations, global accessibility and of course flexible capacity leveraging host elasticity. While many high-end industry & manufacturing companies will sensibly host internal resources to match reliably predictable internal demand, enable performance-sensitive or confidential computations, etc., all will likely employ a non-trivial cloud footprint. The advantages of cloud computing grow larger as aggregate corporate computational workload decreases—so to reach small to medium-sized commercial enterprises, CSE software should be functional (support cloud deployment and on-demand licensing) and execute efficiently on modest problems in an elastic cloud infrastructure.

9.1.6 A new paradigm for regulation

The most stark public-private opportunity for further investigation lies in the formal intersection of government and private industry: *regulation, standardization and certification.* Social media-driven transparency and expectation of corporate citizenship and accountability now align private enterprise with the regulatory agencies' goals of product safety, efficacy and security. Industry and its regulators[6] could elevate the state of the art in achieving these common goals, leveraging CSE software and related technology to both improve the veracity of certification and reducing the burden in cost and time for both parties. For example, to what extent do advances in high fidelity modeling and analytics, pervasive networked sensors and methodologies for verification, validation and uncertainty quantification (VVUQ)[7]

[5]Kramer, William, NCSA blog: http://www.ncsa.illinois.edu/news/stories/TOP500problem/

[6]FDA, FAA, NHTSA, EPA, etc.

[7]Note April, 2013 symposium by The National Academies: "Validation, Verification, and Uncertainty Quantification in Regulation" as well as ASME established (March, 2016) Vol. 1 Issue 1 "Journal of Verifi-

enable virtualized certification methodologies? Correspondingly, how might more standardized guidelines[8] for CSE software employed in critical infrastructure and other public systems improve safety, interoperability, and maintainability—similarly reducing costs and risks to both private and public interests?

This opportunity represents a potential pattern for both productivity and sustainability. Both the regulator and the regulated party have the potential to reduce risks and costs associated with regulatory process and compliance. Commercial entities should be able to justify payment toward sustainable software, reference data and oversight when those costs result in a net financial benefit relative to reporting costs, risk, and time uncertainty as the regulator-accepted software enables virtualization, automation, transparency, and predictability.

9.1.7 Grand challenge: Scalable multiscale

Beyond the scaling of solved problems, there is an opportunity to investigate strategies toward unraveling a prominent barrier problem common to numerous physics-based modeling Grand Challenges (biological systems, materials, climate, weather, etc.): performance-scalable multiscale modeling. Just like the scientific problems mentioned, Industry would benefit from the ability to efficiently couple computations performed at widely different phenomenological resolutions in pursuit of multi-disciplinary product design. Unifying analyses of structural, fluid, electromagnetic, thermal, chemical, mechanical, etc. factors results in higher fidelity simulation of the physical world, which in turn improves understanding and drives convergence in exploring design options. Parallel hardware/software architectures to date have made only modest progress in multiscale solutions in niche problem spaces, leaving an opportunity to develop more general approaches to tame the intractability of integrating time scales or the tyranny imposed on problems exhibiting obstinately long causal chains.

9.1.8 Stimulate workforce development

All of these opportunities, however, rely upon people with the knowledge, skills, and experience to leverage the CSE software advances and fully exploit the potential of emerging computing hardware. In Section 8 we discussed Computational Literacy and Proficiency in detail. In a 2015 survey commissioned by the Council on Competitiveness[9] to understand the workforce's level of readiness in developing and employing computational modeling methods, employers reported requiring "more than one year" of internal (sometimes ad-hoc) training before reaching the desired knowledge and proficiency from recent university graduates. The responses additionally indicated "more than one year" investment in re-training experienced hires and executive decision-makers.

The Council on Competitiveness (March, 2015) publication "*NDEMC Final Report: Modeling, Simulation and Analysis, and High Performance Computing: Force Multiplier for American Innovation*"[10] describes in detail a public-private pilot project enabling small

cation, Validation and Uncertainty Quantification"

[8] "We Need a Building Code for Building Code", Carl Landwehr, Communications of the ACM, Vol. 58 No. 2, pp 24-26

[9] Perceive. Finding the Future through an Exascale Economy, Council on Competitiveness (April, 2016)

[10] http://www.compete.org/publications/all/2938

businesses to leverage CSE software on HPC through support and expertise of several large manufacturers, commercial software vendors, and modeling experts in academia and government. The exercise features not only the challenge to extend expertise, but also pragmatics in accessing hardware facilities and requisite software licensing.

9.2 Development and use challenges of CSE M&S software by industry

Results of a study at General Electric referenced in Industrial Applications of High-Performance Computing[11] outline barriers to adoption that characterize the challenges in development and use of CSE software for industrial users. To start, the economics always must endure scrutiny as decision-makers consider trade-offs and alternative investments—whether between commercial vendors, open source, in-house development or even returning to traditional non-digital methods. Factors considered in such a decision include:

- **Assurance of Rights:** Can I use the software on my proprietary data? Will I retain ownership of the resulting insights? Am I indemnified if the software has infringed on intellectual property or leads to other liabilities?
- **Confidence:** Can I trust the software's results? Will those results justify the cost? Can it overcome internal cultural barriers to change or adoption, including established confidence in a legacy methodology?
- **Productivity:** How difficult is it to learn to use for the types of problems I wish to solve? Can I leverage regression data from legacy software or processes? What are the costs in training, ramp-up time, consulting support? How tedious is it to provide input to the software? How readily will its output answer the questions we ask?
- **Flexibility:** Can the software input from and output to other software I use (including transition from software this is replacing)? Can it be used by multiple users? In other countries? Embedded within a product or service?
- **Portability:** Will the software run efficiently on the targeted hardware (servers/workstations/mobile devices)? Can I migrate use between alternatives? Can it make use of modern hardware (accelerators, visualization technology)?
- **Scalability:** Can I run the software on the full hardware system to which I have access? Can the software compute problems of the size in which I am interested? Is there a prohibitively scale-penalizing licensing fee for my intended use/relative to the size of my hardware investments? (See Table 1).

These factors may be in mutual conflict with one another, resulting in reluctant acquiescence by decision-makers and investigation of further alternatives.

9.2.1 Software licensing terms

Software can be used independent of a product or service or in varying degrees of dependency, such as: exclusively employed a priori in its design; called as a distinct process; linked at runtime or embedded within a product or service itself.

[11] Osseyran, Anwar, Giles, Merle, "Industrial Applications of High-Performance Computing", CRC Press, 2015, pp. 254-261 (GE Research Industrial Applications' Journey to Supercomputing)

A number of open source licensing models[12] exist to provide a legal framework for use of software. Many licenses have terms that are fairly benign for public and academic use, but impose problematic constraints on commercial users. One challenge of public-private partnering is the inconsistency of license selection and the subsequent complexity of evaluating and managing numerous subtly different models within the ecosystem.

Further, regulated industries or commercial entities conforming to certifications such as ISO-9000 require formal documentation of processes, which could include origins of decision-supporting information and workflows embedded within software otherwise opaque.

Where the software created through government agencies or programs is intended to be shared with industry, an examination of the impact of software license choice merits further investigation, including a standardization and simplification strategy.

Sections 6.1.3 and 6.1.4 discuss in depth the economics of commercial software and impact of different licensing choices.

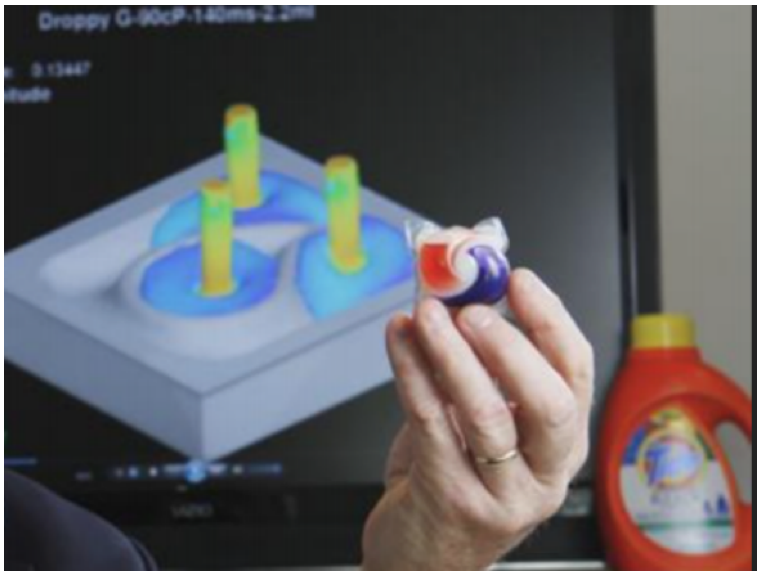


Figure 3: "'Affordable manufacture' of the Tide Pod pouch means doing it FAST, and if there is one thing that is hard to do, it is make liquids do things faster than they want to (especially non-Newtonian liquids). So, we did significant amounts of CFD (Computational Fluid Dynamics) to understand that phenomenon, as well as structural efforts to understand pouch dynamics and strengths." Tom Lange (Procter & Gamble, retired)

9.3 CSE impact: Consumer products

Procter & Gamble have received wide publicity for their innovative use of CSE in product development and enhancement. Tom Lange (retired) has presented many compelling stories of the positive impact CSE has had over the years. There is much more that can be done to improve productivity and sustainability challenges of our CSE tools. Even so, successes are notable, and understandable to non-experts, as seen from the Tide Pod story below.

In 2011, Procter & Gamble's highly successful Tide detergent brand launched an easy-to-use per-wash product called "Tide Pods" (see picture). This product faced an engineering challenge: To put liquid detergent into three separate liquids in a "no leftover" water-dissolvable film pouch that could be affordably manufactured with parts per million quality.

The engineering challenge was also a product acceptance challenge: one leaky pouch and the whole package is ruined, one ruined container and the whole shelf set is ruined, and if

[12] https://en.wikipedia.org/wiki/Comparison_of_free_and_open-source_software_licenses

you create ANY negative consumer experiences, it does not succeed. P&G had been striving for a successful unit dose product since SALVO (not successful) in the late 1950's. But over the last few years, this product form has revolutionized laundry.

10 Summary and Conclusions

Scientific and engineering productivity can be roughly measured as the number of high-quality results produced over a space of time by a particular team. CSE plays a partial but growing role in the productivity equation. It can have a strong positive impact in all aspects of productivity, but at the same time increases the complexity and required skill set for a team.

> Science and engineering will benefit substantially by increasing the productivity and sustainability of CSE efforts. CSE has emerged as such an important element in the overall scientific and engineering endeavor that any substantial improvements in the quality of our software efforts will have multiple and large impacts on any endeavor where CSE plays a role.

At this point in time, there is strong consensus that the scientific and engineering community can benefit substantially by increasing the quality of CSE efforts. CSE has emerged as such an important element in the overall scientific and engineering endeavor that any substantial improvements in our CSE efforts will have a broad impact on science, engineering, industry, economics and society. Furthermore, the biggest opportunities for improving CSE will come from programmer productivity and software sustainability, and lack of progress will significantly hinder our overall success.

This report of the Computational Science and Engineering Software Sustainability and Productivity Challenges Workshop lays out the characterization of our challenges, impediments, and opportunities. We hope that its contents will provide a foundation for progress.

References

[1] Accreditation Board for Engineering and Technology. Accreditation board for engineering and technology main page. http://www.abet.org. [Accessed: 2016-02-13].

[2] S. Ahalt, B. Berriman, M. Brown, J. Carver, N. Chue Hong, A. Fish, R. Idaszak, G. Newman, D. Panda, A. Patra, E. G. Puckett, C. Roland, D. Thain, S. Uluagac, and B. Zhang. Toward a framework for evaluating software success a proposed first step. In *Computational Science & Engineering Software Sustainability and Productivity Challenges (CSESSP) Workshop*, October 15 2015. [Poster] https://www.orau.gov/csessp2015/posters/Ahalt_Stan.pdf [Accessed: 2016-04-11].

[3] B. Akin, F. Franchetti, and J. C. Hoe. FFTs with near-optimal memory access through block data layouts: Algorithm, architecture and design automation. *Journal of Signal Processing Systems*, pages 1–16, 2015. ISSN 1939-8115. doi:10.1007/s11265-015-1018-0.

[4] M. Alcubierre, G. Allen, C. Bona, D. Fiske, T. Goodale, F. S. Guzman, I. Hawke, S. H. Hawley, S. Husa, M. Koppitz, C. Lechner, D. Pollney, D. Rideout, M. Salgado, E. Schnetter, E. Seidel, H. aki Shinkai, D. Shoemaker, B. Szilgyi, R. Takahashi, and J. Winicour. Towards standard testbeds for numerical relativity. *Classical and Quantum Gravity*, 21(2):589, 2004. URL http://stacks.iop.org/0264-9381/21/i=2/a=019.

[5] American Institute of Physics. Recent physics doctorates: Skills used and satisfaction with employment. https://www.aip.org/sites/default/files/statistics/employment/phds-skillsused-p10.pdf, . [Accessed: 2016-04-16].

[6] American Institute of Physics. American institute of physics statistics. https://www.aip.org/sites/default/files/statistics/undergrad/bachdegrees-p-14.pdf, . [Accessed: 2016-02-13].

[7] Association for Computing Machinery. ACM Result and Artifact Review and Badging. http://www.acm.org/publications/policies/artifact-review-badging. [Accessed: 2016-07-01].

[8] J. H. Ausubel. The return to nature: How technology liberates the environment. *Breakthrough Journal*, (5), Spring 2015. URL http://thebreakthrough.org/index.php/journal/issue-5/the-return-of-nature.

[9] W. Bangerth and T. Heister. What makes computational open source software libraries successful? *Computational Science & Discovery*, 6(1):015010, 2013. doi:10.1088/1749-4699/6/1/015010.

[10] W. C. Barley. Anticipatory work: How the need to represent knowledge across boundaries shapes work practices within them. *Organization Science*, 26(6):1612–1628, 2015. doi:10.1287/orsc.2015.1012. URL http://dx.doi.org/10.1287/orsc.2015.1012.

[11] R. Bartlett. A Roadmap for Sustainable Ecosystems of CSE Software. Technical report, Oak Ridge National Laboratory, June 26 2015. URL http://web.ornl.gov/~8vt/CSEEcosystemSustainability_CSESSP2015.pdf.

[12] R. Bartlett, M. Heroux, and J. Willenbring. Estimating the Total Development Cost of a Linux Distribution. Technical report, SAND2012-0561, SANDIA, February 2012. URL http://web.ornl.gov/~8vt/TribitsLifecycleModel_v1.0.pdf.

[13] R. A. Bartlett, M. A. Heroux, and J. M. Willenbring. Overview of the TriBITS lifecycle model: A lean/agile software lifecycle model for research-based computational science and engineering software. In *8th IEEE International Conference on E-Science (e-Science 2012)*, pages 1–8, October 2012. doi:10.1109/eScience.2012.6404448.

[14] V. Basili, A. Trendowicz, M. Kowalczyk, J. Heidrich, C. Seaman, J. Münch, and D. Rombach. *Aligning Organizations Through Measurement: The GQM+Strategies Approach*. The Fraunhofer IESE Series on Software and Systems Engineering. Springer, 2014.

[15] V. R. Basili. Using measurement for quality control and process improvement. In *Second Annual SEPG Workshop, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA*, June 21-22 1989.

[16] V. R. Basili and H. D. Rombach. The TAME project: towards improvement-oriented software environments. *IEEE Transactions on Software Engineering*, 14(6):758–773, Jun 1988. ISSN 0098-5589. doi:10.1109/32.6156.

[17] V. R. Basili, J. C. Carver, D. Cruzes, L. M. Hochstein, J. K. Hollingsworth, F. Shull, and M. V. Zelkowitz. Understanding the high-performance-computing community: A software engineer's perspective. *IEEE Software*, 25(4):29–36, 2008. ISSN 0740-7459. doi:10.1109/MS.2008.103.

[18] I. Baxter. Design maintenance systems. *Communications of the ACM*, 35(4), April 1992.

[19] I. Baxter. Program analysis methods, 2016. Personal Communication.

[20] I. Baxter and E. Kant. Using domain-specific, abstract parallelism. In I. Foster and E. Tick, editors, *Proceedings of the Workshop on Compilation of (Symbolic) Languages for Parallel Computers*, October 1991.

[21] I. Baxter and M. Mehlich. Reverse engineering is reverse forward engineering. In *Working Conference on Reverse Engineering*. IEEE, 1997.

[22] I. Baxter, A. Yahin, L. Moura, M. SantAnna, , and L. Bier. Clone detection using abstract syntax trees. In *Proceedings of the International Conference on Software Maintenance*. IEEE Press, 1998.

[23] I. Baxter, C. Pidgeon, and M. Mehlich. Program transformations for practical scalable software evolution. In *Proceedings, 26th International Conference on Software Engineering*. IEEE Computer Society, 2004.

[24] B. A. Bechky. Sharing meaning across occupational communities: The transformation of understanding on a production floor. *Organization Science*, 14(3):312–330, 2003. doi:10.1287/orsc.14.3.312.15162.

[25] B. Boehm. *Software Engineering Economics (1st Edition)*. Prentice Hall, 1981.

[26] B. Boehm. Software Engineering Economics. *IEEE Transactions on Software Engineering*, 10(1):4–21, 1984.

[27] J. S. Brown and P. Duguid. Knowledge and organization: A social-practice perspective. *Organization Science*, 12(2):198–213, 2001. doi:10.1287/orsc.12.2.198.10116.

[28] H. C. Bruns. Working alone together: Coordination in collaboration across domains of expertise. *Academy of Management Journal*, 56(1):62–83, 2013. doi:10.5465/amj.2010.0756.

[29] Bureau of Labor Statistics. Bureau of Labor Statistics Data. http://www.bls.gov/oes/current/oes151132.htm. [Accessed: 2016-03-18].

[30] G. Campbell. Software-intensive systems producibility: A vision and roadmap (v. 0.1). Technical Report CMU/SEI-2007-TN-017, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, December 2007. URL http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=8243.

[31] C. Collberg and T. A. Proebsting. Repeatability in computer systems research. *Commun. ACM*, 59(3):62–69, Feb. 2016. ISSN 0001-0782. doi:10.1145/2812803.

[32] Computational Infrastructure for Geodynamics (CIG). Software development Best Practices. https://geodynamics.org/cig/dev/best-practices/. [Accessed: 2016-03-18].

[33] Council on Competitiveness. Case study: Boeing catches a lift with high performance computing, 2009. [Report] http://hpc4energy.org/wp-content/uploads/HPC_Boeing_072809_A-1.pdf [Accessed 2016-04-09].

[34] Council on Competitiveness. Perceive. Finding the future through an exascale economy. Technical report, 2016. [Report] http://www.compete.org [Accessed 2016-07-01].

[35] K. Crowston, K. Wei, J. Howison, and A. Wiggins. Free/libre open-source software development: What we know and what we do not know. *ACM Comput. Surv.*, 44(2):7:1–7:35, Mar. 2008. ISSN 0360-0300. doi:10.1145/2089125.2089127.

[36] L. Dabbish, C. Stuart, J. Tsay, and J. Herbsleb. Social coding in GitHub: Transparency and collaboration in an open software repository. In *Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work*, CSCW '12, pages 1277–1286, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1086-4. doi:10.1145/2145204.2145396.

[37] C. W. H. Davis. Case study: Moving to the pull request workflow & integrating quality engineers on the team for better software quality. *DZone/Agile Zone*, March 28 2015. https://dzone.com/articles/case-study-moving-pull-request [Accessed: 2016-04-12].

[38] C. Desclaux. Capturing design and maintenance decisions with macs. *Journal of Software: Evolution and Process*, 4(4), October 2006.

[39] DOE report. Software productivity for extreme-scale science, January 2014. URL http://www.orau.gov/swproductivity2014/reference.htm.

[40] J. Dongarra, L. Petzold, and V. Voevodin. Summary of CSE Survey. https://www.siam.org/students/resources/pdf/Summary-of-CSE-Survey.pdf. [Accessed: 2016-02-13].

[41] D. Dougherty. Interpretive barriers to successful product innovation in large firms. *Organization Science*, 3(2):179–202, 1992. doi:10.1287/orsc.3.2.179.

[42] A. Dubey, M. Turk, and B. O'Shea. The impact of community software in astrophysics. In *Proceedings of WCCM-ECCM-ECFD 2014*, 2014. Available at http://www.wccm-eccm-ecfd2014.org/admin/files/filePaper/p1174.pdf. [Accessed: 2016-04-05].

[43] C. Duhigg. What google learned from its quest to build the perfect team. New York Times, February 28 2016. URL http://nyti.ms/20Vn3sz. [Accessed: 2016-04-10].

[44] S. M. Easterbrook and T. C. Johns. Engineering the software for understanding climate change. *Computing in Science & Engineering*, 11(6):65–74, Nov 2009. ISSN 1521-9615. doi:10.1109/MCSE.2009.193.

[45] A. Edmondson. Psychological safety and learning behavior in work teams. *Administrative Science Quarterly*, 44(2):350–383, 1999. doi:10.2307/2666999.

[46] P. N. Edwards, M. S. Mayernik, A. L. Batcheller, G. C. Bowker, and C. L. Borgman. Science friction: Data, metadata, and collaboration. *Social Studies of Science*, 41(5): 667–690, 2011. doi:10.1177/0306312711413314.

[47] M. Egerstedt. The Mechanics of a CPS(-ish) MOOC: The Good, The Bad, The Ugly. In *2014 NSF Cyber-Physical Systems Principal Investigators Meeting (CPS)*, November 11 2014. [Presentation] http://cps-vo.org/node/15865 [Accessed: 2016-03-18].

[48] T. Ellman and T. Murato. Deductive synthesis of numerical simulation programs from networks of algebraic and ordinary differential equations. In *Automated Software Engineering*. Kluwer Academic Publishers, 1998.

[49] S. Faulk, E. Loh, M. L. V. de Vanter, S. Squires, and L. Votta. Scientific computing's productivity gridlock: How software engineering can help. *Computing in Science & Engineering*, 11(6):30–39, November 2009. ISSN 1521-9615. doi:10.1109/MCSE.2009.205.

[50] Flash Center for Computational Science. Building community codes for effective scientific research on HPC platforms, 2012. [Workshop] http://flash.uchicago.edu/cc2012/ [Accessed 2016-04-09].

[51] M. Fowler. Improvement ravine, October 18 2006. [Blog] http://martinfowler.com/bliki/ImprovementRavine.html [Accessed: 2016-04-11].

[52] B. Fryxell, K. Olson, P. Ricker, F. X. Timmes, M. Zingale, D. Q. Lamb, P. MacNeice, R. Rosner, J. W. Truran, and H. Tufo. FLASH: Adaptive mesh hydrodynamics code for modeling astrophysical thermonuclear flashes. Astrophysics Source Code Library, Oct. 2010. URL http://adsabs.harvard.edu/abs/2010ascl.soft10082F. [Accessed: 2016-04-11].

[53] Geant4. Geant4 web page. http://www.geant4.org. [Accessed: 2016-03-18].

[54] GitHub. Github guides: Making your code citable. https://guides.github.com/activities/citable-code/. [Accessed: 2016-03-18].

[55] R. Glass. Frequently Forgotten Fundamental Facts about Software Engineering. *IEEE Software*, May/June:110, 2001.

[56] T. Goodale, G. Allen, G. Lanfermann, J. Massó, T. Radke, E. Seidel, and J. Shalf. The Cactus framework and toolkit: Design and applications. In *Vector and Parallel Processing – VECPAR'2002, 5th International Conference, Lecture Notes in Computer Science*, Berlin, 2003. Springer. URL http://edoc.mpg.de/3341.

[57] J. E. Hannay, C. MacLeod, and J. Singer. How do scientists develop and use scientific software? In *ICSE Workshop on Software Engineering for Computational Science and Engineering*, pages 1–8. Vancouver, Canada: IEEE, 2009.

[58] HDF. The HDF Group. https://www.hdfgroup.org/. [Accessed: 2016-03-18].

[59] M. Heroux, R. Bartlett, and J. Willenbring. Software Engineering Principles: The TriBITS Lifecycle Model. SANDIA National Laboratories. February 2012. [Presentation] http://www.sandia.gov/~maherou/docs/HerouxTribitsOverview.pdf [Accessed: 2016-03-18].

[60] M. A. Heroux. Editorial: ACM TOMS replicated computational results initiative. *ACM Trans. Math. Softw.*, 41(3):13:1–13:5, June 2015. ISSN 0098-3500. doi:10.1145/2743015.

[61] S. Hettrick. Why we need to create careers for research software engineers. In *Scientific Computing World*, November 11 2015. http://www.scientific-computing.com/news/news_story.php?news_id=2737 [Accessed: 2016-03-18].

[62] S. Hettrick. Research software sustainabilty. Report on a Knowledge Exchange Workshop, 2016. http://repository.jisc.ac.uk/6332/1/Research_Software_Sustainability_Report_on_KE_Workshop_Feb_2016_FINAL.pdf.

[63] J. P. Holdren. Increasing access to the results of federally funded scientific research, February 22 2013. Memorandum for the heads of executive departments and agencies, Office of Science and Technology Policy.

[64] E. Houstis, J. Rice, E. Gallopoullos, and R. Bramley. *Enabling Technologies for Computational Science: Frameworks, Middleware, and Environments*. Springer, 2012.

[65] J. Howison and K. Crowston. Collaboration through open superposition: A theory of the open source way. *MIS Quarterly*, 38(1):29–50, 2014.

[66] J. Howison and J. Herbsleb. Scientific software production: Incentives and collaboration. In *Proceedings of the ACM 2011 conference on Computer Supported Cooperative Work*, pages 513–522. Hangzhou, China: ACM, 2011.

[67] HPC4Energy. Success stories: Boeing. http://hpc4energy.org/hpc-road-map/success-stories/boeing/, 2009. Last access: 2016-04-05.

[68] W. Humphrey. Characterizing the software process: a maturity framework. *IEEE Software*, 5(2):73–79, March 1988. ISSN 0740-7459. doi:10.1109/52.2014.

[69] Impactstory. Let's value the software that powers science: Introducing Depsy. [Blog] http://blog.impactstory.org/introducing-depsy/ [Accessed: 2016-03-18].

[70] iRODS Consortium. About the iRODS Consortium. http://irods.org/consortium/. [Accessed: 2016-03-18].

[71] E. Kant. Synthesis of mathematical-modeling software. *IEEE Software*, 10, May 1993.

[72] D. S. Katz. Transitive credit as a means to address social and technological concerns stemming from citation and attribution of digital products. *Journal of Open Research Software*, 2(1):e20, 2014. doi:10.5334/jors.be.

[73] D. S. Katz. Sustainable software needs a change in the culture of science. In *Scientific Computing World*, January 15 2016. http://www.scientific-computing.com/news/news_story.php?news_id=2759 [Accessed: 2016-03-18].

[74] D. S. Katz, S.-C. T. Choi, H. Lapp, K. Maheshwari, F. Löffler, M. Turk, M. Hanwell, N. Wilkins-Diehr, J. Hetherington, J. Howison, S. Swenson, G. Allen, A. Elster, B. Berriman, and C. Venters. Summary of the first workshop on sustainable software for science: Practice and experiences (WSSSPE1). *Journal of Open Research Software*, 2(1):e6, 2014. ISSN 2049-9647. doi:10.5334/jors.an.

[75] Kerberos Consortium. MIT Kerberos Consortium. http://www.kerberos.org/. [Accessed: 2016-03-18].

[76] D. Keyes and V. Taylor (Task Force Co-Chairs). National Science Foundation Advisory Committee on CyberInfrastructure, Task Force on Software for Science and Engineering, Final Report, 2011. http://www.nsf.gov/cise/aci/taskforces/TaskForceReport_Software.pdf.

[77] Kitware. Kitware and UNC Receive NIH Award to Predict Stroke Outcomes Using 3D Models of Brain Blood Vessels. https://blog.kitware.com/kitware-and-unc-receive-nih-award-to-predict-stroke-outcomes-using-3d-models-of-brain-blood-vessels/. [Accessed: 2016-03-18].

[78] P. Klint, T. van der Storm, and J. Vinju. Rascal: A domain specific language for source code analysis and manipulation. In *Proceedings 2009 Conference of Source Code Analysis and Manipulation*. IEEE, 2009.

[79] H. Kniberg and M. Skarin. *Kanban and Scrum - Making the Most of Both*. InfoQ, 2009.

[80] R. E. Kraut and P. Resnick. *Building Successful Online Communities: Evidence-Based Social Design*. MIT Press, 2011.

[81] Krell Institute. Computational science graduate fellowships. https://www.krellinst.org/csgf/. [Accessed: 2016-02-13].

[82] S. Krishnamurthi. Artifact evaluation for software conferences. *SIGSOFT Softw. Eng. Notes*, 38(3):7–10, May 2013. ISSN 0163-5948. doi:10.1145/2464526.2464530. URL http://doi.acm.org/10.1145/2464526.2464530.

[83] E. Lagercrantz. Stencil computation auto-tuning via dataflow graph transformations. Master's thesis, Uppsala University, Department of Information Technology, 2015.

[84] F. Loeffler, J. Faber, E. Bentivegna, T. Bode, P. Diener, R. Haas, I. Hinder, B. C. Mundim, C. D. Ott, E. Schnetter, G. Allen, M. Campanelli, and P. Laguna. The einstein toolkit: a community computational infrastructure for relativistic astrophysics. *Classical and Quantum Gravity*, 29(11):115001, 2012. URL http://stacks.iop.org/0264-9381/29/i=11/a=115001.

[85] M. Lowry, A. Philpot, T. Pressburger, and I. Underwood. Amphion: Automatic programming for scientific subroutine libraries. In *Proceedings 8th International Symposium on Methodologies for Intelligent Systems*, LNCS 869. Springer Verlag, 1994.

[86] D. F. Maron. 12 surprising facts about Nobel prizes. *Scientific American*, October 5 2015. http://www.scientificamerican.com/article/12-surprising-facts-about-nobel-prizes/ [Accessed: 2016-04-10].

[87] M. Matsumoto, F. Mori, S. Ohshima, H. Jitsumoto, T. Katagiri, and K. Nakajima. Implementation and evaluation of an AMR framework for FDM applications. *Procedia Computer Science*, 29:936 – 946, 2014. ISSN 1877-0509. doi:10.1016/j.procs.2014.05.084. 2014 International Conference on Computational Science.

[88] A. McPherson, B. Proffitt, and R. Hale-Evans. Estimating the Total Development Cost of a Linux Distribution. Technical report, The Linux Foundation, October 2008. URL http://www.linuxfoundation.org/sites/main/files/publications/estimatinglinux.html.

[89] G. Miller. A scientist's nightmare: Software problem leads to five retractions. *Science*, 314(5807):1856–1857, 2006. ISSN 0036-8075. doi:10.1126/science.314.5807.1856.

[90] NAMD. NAMD - Scalable Molecular Dynamics. http://www.ks.uiuc.edu/Research/namd/. [Accessed: 2016-03-18].

[91] National Science Foundation. Implementation of NSF CIF21 Software Vision (SW-Vision). http://www.nsf.gov/si2. [Accessed: 2016-03-18].

[92] J. Neighbors. The Draco approach to constructing software from reusable components. *Transactions on Software Engineering*, 10(5), 1984.

[93] T. Oden et al. Final Report of the Advisory Committee for Cyberinfrastructure Task Force on Grand Challenges. Technical report, National Science Foundation, 2011. http://www.nsf.gov/cise/aci/taskforces/TaskForceReport_GrandChallenges.pdf.

[94] OpenFOAM. OpenFOAM - The open source CFD toolbox. http://www.openfoam.com/.

[95] B. W. O'Shea, G. Bryan, J. Bordner, M. L. Norman, T. Abel, R. Harkness, and A. Kritsuk. Enzo: AMR cosmology application. Astrophysics Source Code Library, Oct. 2010. URL http://adsabs.harvard.edu/abs/2010ascl.soft10072O. [Accessed: 2016-04-11].

[96] R. E. Park, W. B. Goethert, and W. A. Florac. Goal-driven software measurement—a guidebook. Technical Report CMU/CEI-96-HB-002, Software Engineering Institute Handbook, 1996. URL http://resources.sei.cmu.edu/library/asset-view.cfm?assetID=12453.

[97] Pegasus. Pegasus powers LIGO gravitational wave detection analysis. https://pegasus.isi.edu/2016/02/11/pegasus-powers-ligo-gravitational-waves-detection-analysis/. [Accessed: 2016-03-18].

[98] L. Petzold et al. Graduate education in computational science and engineering. *SIAM Review*, 43(1):163–177, 2001.

[99] D. Post and L. Votta. Computational science demands and new paradigm. *Physics Today*, 58(1):35–41, 2005.

[100] Project CRediT. Contributor Roles Taxonomy. http://casrai.org/CRediT/. [Accessed: 2016-03-18].

[101] D. Quinlan. Rose tutorial - rose compiler infrastructure, 2015.

[102] R Project. The R Project for Statistical Computing. https://www.r-project.org/. [Accessed: 2016-03-18].

[103] R. Ramnath and D. S. Katz. Software Infrastructure for Sustained Innovation (SI2). In *2016 NSF SI2 PI Workshop*, February 16 2016. [Presentation, Slide 25] http://cococubed.asu.edu/si2_pi_workshop_2016/ewExternalFiles/2016-PI-Meeting-PD-Presentation-2016-02-15-20-49-Compressed.pdf [Accessed: 2016-03-18].

[104] C. Rich and R. Waters. The programmer's apprentice project: A research overview. Technical Report Memo 1004, MIT AI Lab, 1987.

[105] Rolls Royce. Design systems and tools. http://www.rolls-royce.com/about/our-technology/enabling-technologies/design-systems-tools.aspx, 2016. [Accessed: 2016-04-09].

[106] H. D. Rombach and B. T. Ulery. Improving software maintenance through measurement. *Proceedings of the IEEE*, 77(4):581–595, Apr 1989. ISSN 0018-9219. doi:10.1109/5.24144.

[107] R. Ross, P. Beckman, R. Latham, K. Iskra, et al. HEDP software introductions: ESC, operating systems, and i/o. http://flash.uchicago.edu/~dubey/Feb28_2011/ross_io_os.pdf, February 2011. Last access: April 16, 2016.

[108] U. Rüde, K. Willcox, L. C. McInnes, H. D. Sterck, et al. Future Directions in CSE Education and Research, 2016. Report from a Workshop Sponsored by SIAM and EESI-2, available via http://wiki.siam.org/siag-cse/index.php/Main_Page.

[109] J. Segal. When software engineers met research scientists: A case study. *Empirical Software Engineering*, 10(4):517–536, 2005. ISSN 1573-7616. doi:10.1007/s10664-005-3865-y.

[110] J. Segal. Some challenges facing software engineers developing software for scientists. In *Software Engineering for Computational Science and Engineering, 2009. SECSE '09. ICSE Workshop on*, pages 9–14, May 2009. doi:10.1109/SECSE.2009.5069156.

[111] B. F. Smith and R. A. Bartlett. xSDK package compliance standards, March 2016. [draft document] https://ideas-productivity.org/resources/xsdk-docs/.

[112] Software Carpentry. Software carpentry. http://software-carpentry.org/, 1998. [Accessed: 2016-04-09].

[113] Software Engineering for Science. Call for Book Chapter Proposals. http://se4science.org/. [Accessed: 2016-03-18].

[114] V. Springel. GADGET-2: A code for cosmological simulations of structure formation. Astrophysics Source Code Library, Mar. 2000. URL http://adsabs.harvard.edu/abs/2000ascl.soft03001S. [Accessed: 2016-04-11].

[115] M. Stewart. An experiment in scientific program understanding. In *Automated Software Engineering*. Kluwer Academic Publishers, 2000.

[116] M. Stewart. Report on automated semantic analysis of scientific and engineering codes. Technical Report CR-2001-211078, NASA, 2001.

[117] J. M. Stone and M. L. Norman. ZEUS-2D: A radiation magnetohydrodynamics code for astrophysical flows in two space dimensions. i - the hydrodynamic algorithms and tests. *Astrophysical Journal Supplement Series*, 80:753–790, June 1992. doi:10.1086/191680. URL http://adsabs.harvard.edu/abs/1992ApJS...80..753S. [Accessed: 2016-04-11].

[118] J. M. Stone and M. L. Norman. ZEUS-2D: Simulation of fluid dynamical flows. Astrophysics Source Code Library, June 2013. URL http://adsabs.harvard.edu/abs/2013ascl.soft06014S. [Accessed: 2016-04-11].

[119] J. M. Stone, T. A. Gardiner, P. Teuben, J. F. Hawley, and J. B. Simon. Athena: Grid-based code for astrophysical magnetohydrodynamics (MHD). Astrophysics Source Code Library, Oct. 2010. URL http://adsabs.harvard.edu/abs/2010ascl.soft10014S. [Accessed: 2016-04-11].

[120] Taulbee Survey. The Taulbee Survey. http://cra.org/crn/2015/05/2014_taulbee_survey. [Accessed: 2016-02-13].

[121] The yt project. yt: A multi-code analysis toolkit for astrophysical simulation data. Astrophysics Source Code Library, Nov. 2010. URL http://adsabs.harvard.edu/abs/2010ascl.soft11022T. [Accessed: 2016-04-11].

[122] P. Tzeferacos. FLASH code, development and applications: an open source tool for HEDP. https://www.alcf.anl.gov/events/flash-code-development-and-applications-open-source-tool-hedp, December 19 2013. [Accessed: 2016-04-11].

[123] P. Tzeferacos, M. Fatenejad, N. Flocke, C. Graziani, G. Gregori, D. Lamb, D. Lee, J. Meinecke, A. Scopatz, and K. Weide. FLASH MHD simulations of experiments that study shock-generated magnetic fields. *High Energy Density Physics*, 17, Part A:24–31, 2015. ISSN 1574-1818. doi:10.1016/j.hedp.2014.11.003. Special Issue: 10th International Conference on High Energy Density Laboratory Astrophysics.

[124] R. van de Geijn. Linear algebra foundations to frontiers. http://www.ulaff.net/index.html.

[125] R. Van Noorden. Science publishing: The trouble with retractions. *Nature*, 478:26–28, 2011.

[126] VTK. The Visualization Toolkit. http://www.vtk.org/. [Accessed: 2016-03-18].

[127] G. Wilson. Software carpentry: lessons learned. *arXiv:1307.5448*. URL http://arxiv.org/abs/1307.5448. [Accessed: 2016-04-16].

[128] G. Wilson, D. A. Aruliah, C. T. Brown, N. P. C. Hong, M. Davis, R. T. Guy, S. H. Haddock, K. D. Huff, I. M. Mitchell, M. D. Plumbley, B. Waugh, E. P. White, , and P. Wilson. Best practices for scientific computing. 2014. doi:10.1371/journal.pbio.1001745.

[129] Zenodo. Research shared. http://zenodo.org/. [Accessed: 2016-03-18].

Submitted White Papers

Electronic version of these papers are available for download from the CSESSP Challenges Workshop website: https://www.orau.gov/csessp2015/whitepapers.htm.

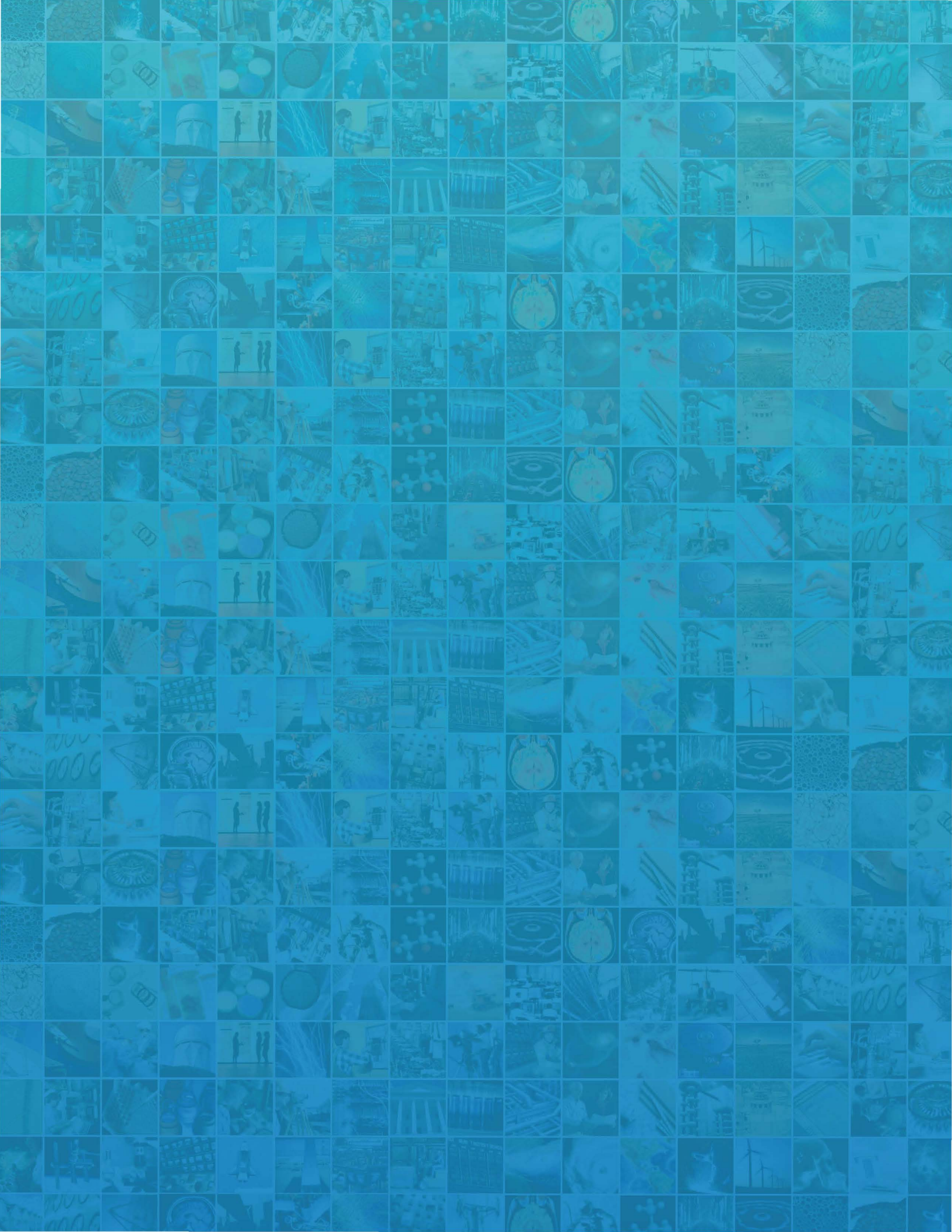
Workshop Participants and Report Contributors

Name	*Organization*
Gabrielle Allen	University of Illinois Urbana-Champaign
Richard Arthur	General Electric
Roscoe Bartlett	Oak Ridge National Laboratory, Sandia National Laboratories
Ira Baxter	Semantic Designs
David E. Bernholdt	Oak Ridge National Laboratory
Ronald Boisvert	National Institute of Standards and Technology
Grady Campbell	domain-specific.com
Lali Chatterjee	DOE Office of Science, High Energy
Thomas Clune	NASA Global Modeling and Assimilation Office
Ethan Coon	Los Alamos National Laboratory
Kosta Damevski	Virginia Commonwealth University
Jack Dongarra	University of Tennessee, Knoxville
Anshu Dubey	Argonne National Laboratory
Karamarie Fecho	Copperline Professional Solutions
Rob Fowler	RENCI, University of North Carolina at Chapel Hill
Joerg Gablonski	The Boeing Company
Mike Glass	Sandia National Laboratories
Sol Greenspan	National Science Foundation
Timo Heister	Clemson University
Michael Heroux	Sandia National Laboratories
Costin Iancu	Lawrence Berkeley National Laboratory
Daniel Ibanez	Renselaer Polytechnical Institute
Ray Idaszak	RENCI, University of North Carolina at Chapel Hill
Paul Jones	FDA
Christos Kartsaklis	Oak Ridge National Laboratory
Daniel S. Katz	National Science Foundation
Walid Keyrouz	National Institute of Standards and Technology
Quincey Koziol	The HDF Group
Sandy Landsberg	DoD HPC Modernization Program (HPCMP)
Tom Lange	Procter & Gamble-Retired
Steven Lee	DOE ASCR
David Lesmes	Department of Energy/BER
Frank Löffler	Louisiana State University
Ernie Lucier	NITRD
Vijay Mahadevan	Argonne National Laboratory
John McGregor	Clemson University
Lois Curfman McInnes	Argonne National Laboratory
Robert Nagler	RadiaSoft LLC

Name	*Organization*
Thomas Ndousse-Fetter	U.S. Department of Energy
Tien Nguyen	Iowa State University
Sudhakar Pamidighantam	Indiana University
Abani Patra	University at Buffalo
Aleksandra Pawlik	Software Sustainability Institute, New Zealand eScience Infrastructure (NeSI)
Douglass Post	Software Engineering Institute, DoD High Performance Modernization Program (HPCMP)
Hridesh Rajan	Iowa State University
Albert Reuther	MIT Lincoln Laboratory
Rob Roser	Fermi National Accelerator Laboratory
Karl Rupp	Freelance Computational Scientist
Walter Scarborough	Texas Advanced Computing Center, University of Texas
Will Schroeder	Kitware, Inc.
Todd Simons	Rolls-Royce
Faulk Stuart	University of Oregon
David Tarboton	Utah State University
Karen Tomko	Ohio Supercomputer Center
Colin Venters	University of Huddersfield
Jeffrey Vetter	Oak Ridge National Laboratory, Georgia Tech University
Hai Zhu	DuPont

Computational Science and Engineering Software Sustainability and Productivity Challenges (CSESSP) Workshop
October 15 - 16, 2015

	Thursday, October 15, 2015	
7:30am - 8:30am	Breakfast and registration	
8:30am - 9:00am	Program Workshop Welcome: Networking and Information Technology Research and Development Program	Sol Greenspan, NITRD SDP Keith A. Marzullo, NITRD – Director Erwin Gianchandani, NSF - Activing Deputy Director, CISE Thomas Ndousse-Fetter, DOE
9:00am - 9:30am	High Energy Physics and The Issues of Software Sustainability	Robert Roser, Fermi Lab
9:30am - 10:00am	Invited lightning talks (5 minutes each) 4 in this session: 1. Addressing sustainability and performance portability challenges in Albany 2. Analysis and Optimization of High Level Languages 3. Scientific Software Sustainability The Numerical Reproducibility Challenge 4. The Need for a Common Concurrency Language	1. **Irina Demeshko**, Andrew Salinger and Michael A Heroux 2. **Costin Iancu**, Wim Lavrijsen and Koushik Sen 3. **Walid Keyrouz** and Michael Mascagni 4. Stephen F. Siegel, Manchun Zheng, Paul Hovland and **Matthew Dwyer**
10:00am - 10:30am	*Break*	
10:30am - 11:00am	Review of the agenda and breakout sessions	
11:00am - 11:45am	Breakout session 1: Explore: A, B. C, D, E, F	A. Jack Dongarra, University of Tennessee B. Richard Arthur, GE C. Ray Idaszak, RENCI D. Daniel Katz, NSF E. Abani Patra, SUNY Buffalo F. Anshu Dubey, LBNL
11:45am - 12:15pm	Breakout report out	
12:15pm - 1:15pm	*Lunch*	
1:15pm - 2:00pm	Invited lightning talks (5 minutes each) 8 in this session: 1. Looking at Software Sustainability and Productivity Challenges from NSF 2. Lack of Software Specifications: A Sustained Sustainability and Productivity Crisis 3. Adapting Legacy Monolithic Applications into API Services for Web, Mobile and Other Clients 4. Sustainability and Reproducibility via Containerized Computing 5. A Risk-based, Practice-centered Approach to Project Management for HPCMP CREATE 6. User-Extensible Compiler Toolchains for Refactoring CSE Software 7. The curse of growing scales: from inception to successful community-driven software development 8. A Platform Strategy for Economically Sustainable Software	1. **Daniel Katz** and Rajiv Ramnath 2. **Hridesh Rajan**, Tien Nguyen, Gary T. Leavens, Robert Dyer and Vasant Honavar 3. **Walter Scarborough**, Carrie Arnold, Rion Dooley, Ajit Gauli, Maytal Dahan and Matthew Hanlon 4. **Robert Nagler**, David Bruhwiler, Paul Moeller and Stephen Webb 5. **Douglass Post**, Chris Atwood and Richard Kendall 6. **Christos Kartsaklis**, David E. Bernholdt and Dali Wang 7. **Vijay Mahadevan** and Andrew Siegel 8. **William Schroeder**
2:00pm - 2:45pm	Breakout session 2: Explore G,H Focus: A, B, C	Lois McInnes, ANL Sandy Landsberg, DOD
2:45pm - 3:15pm	Breakout report out	
3:15pm - 3:45pm	*Break*	
3:45pm - 5:00pm	Federal Panel on CSE Software Investments: Michael Heroux, SNL Invited panelists: DOD, DOE, NASA, SEI, NIST, NSF	Guna Seetharaman, NRL Thomas Ndousse-Fetter, DOE Tom Clune, NASA Doug Post, SEI Ron Boisvert, NIST Rajiv Ramnath, NSF
5:00pm - 5:30pm	Breakout focus group	
5:30pm - 7:00pm	Poster Session with reception	Salon I

Computational Science and Engineering Software Sustainability and Productivity Challenges (CSESSP) Workshop
October 15 - 16, 2015

	Friday, October 16, 2015	
8:00am - 9:00am	*Breakfast*	
9:00am - 9:30am	Productivity and Sustainability in Disruptive Times	Bob Lucas, USC, LSTC
9:30am - 10:00am	Scientific Computing's Productivity Gridlock and How Software Engineering Can Help	Stuart Faulk, University of Oregon
10:00am - 10:15am	Review of day one and plan for day two	
10:15am - 10:45am	*Break*	
10:45am - 11:30am	Breakout sessions 3: Focus: D, E, F, G, H	
11:30am - 12:00pm	Breakout report out	
12:15pm - 1:15pm	*Lunch*	
1:15pm - 1:35pm	Invited lightning talks (5 minutes each) 3 in this session: 1. A Roadmap for Sustainable Ecosystems of CSE Software 2. Better Software, Better Research: Providing Scalable Support for Scientific Software Development 3. Opportunities in Computational Science to Advance Software Engineering	1. **Roscoe Bartlett** 2. **Aleksandra Pawlik**, Neil Chue Hong 3. **Devin Matthews**, Don Batory, Bryan Marker and Robert van de Geijn
1:35pm - 2:00pm	Breakout sessions 4: Write up: A, B, C, D, E, F, G, H A. Opportunities from Improved CSE SW Sustainability and Productivity B. CSE Software in Industry_Manufacturing C. Economics of Software Tools D. Social Sciences Applied to CSE Software Systems E. Workforce Development F. Role of Software Engineering Research G. Measuring Software Productivity and Performance H. New Approaches for Faster, More Affordable CSE Software	A. Jack Dongarra, University of Tennessee B. Richard Arthur, GE C. Ray Idaszak, RENCI D. Daniel Katz, NSF E. Abani Patra, SUNY Buffalo F. Anshu Dubey, LBNL G. Lois Curfman McInnes H. Sandy Landsberg
2:00pm - 3:15pm	Breakout report out	
3:15pm - 3:45pm	*Break*	
3:45pm - 5:00pm	Report structure and planning	
5:00pm - 5:10pm	Conclusion