Do Developers Understand IEEE Floating Point?

Peter Dinda  Conor Hetland
Prescience Lab  
Department of EECS
Northwestern University

pdinda.org
presciencelab.org

Survey Available Here, Please Participate!

Paper in IPDPS 2018
Paper in a Nutshell: **Not Really**

- **Targeted survey**
  - Aimed at practitioners likely to use FP
  - Quizzes for core, optimization, and suspicion of results
  - *First study of this kind*

- **Participants do only slightly better than chance on core concepts**
  - … and don’t know it
  - Some factors mitigate, but none particularly well

- **Participants do not understand optimization concepts**
  - … and do know it

- **Participants less suspicious than they should be**
  - … but similar to students in a sophomore course

- **Maybe systems software can do something about it**
Outline

• Motivation
• Study design
• Participant selection and factors
  – Important caveat!
• Core concepts
• Optimization concepts
• Suspicion of results
• What to do?
  – What are we doing?
For a Long Time...

<table>
<thead>
<tr>
<th>Developer</th>
<th>Focused on scientific and engineering uses, Some understanding of numerical methods Assumption/understanding of IEEE floating point</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compiler (Optimizations)</td>
<td>Small set of compilers used, slow change, difficult to break IEEE compliance</td>
</tr>
<tr>
<td>Hardware (Optimizations)</td>
<td>Small set of hardware, IEEE compliance universal, slow change</td>
</tr>
</tbody>
</table>
The Concerns Now

**Developer**

- **Dramatic expansion** in uses (e.g., machine learning, analytics, big data, and other expanding uses of FP) Less knowledge of numerical methods, and the standard

**IEEE 754(-2008) Standard**

- Stable, pretty much universal standard since early 1980s
- Considerable complexity

**Compiler (Optimizations)**

- **Fast evolution** (e.g., numerous compilers, automatic precision reduction, approximate computing, optimization flag choice, automatic optimization setting search, power/energy)

**Hardware (Optimizations)**

- **Fast evolution** (e.g., hardware diversity (GPUs, FPGAs, ARM), half-floats, different denorm handling, non-IEEE compliance, power/energy)
Do Developers Understand....

**Core Focus**

*Dramatic expansion* in uses (e.g., machine learning, analytics, big data, and other expanding uses of FP) Less knowledge of numerical methods, and standard

**Optimization Focus**

Fast evolution (e.g., numerous compilers, automatic precision reduction, approximate computing, optimization flag choice, automatic optimization setting search, power/energy)

Fast evolution (e.g., hardware diversity (GPUs, FPGAs, ARM), half-floats, different denorm handling, non-IEEE compliance, power/energy)

... and Suspicion
Study Design

• **Anonymity**
• **Factor identification**
• **Low time commitment**

• **Survey instrument (web-based)**
  – Participant background (for factor analysis)
  – Core quiz
  – Optimization quiz
  – Suspicion quiz

• **Closed for study reported here, but open again now**
  – http://presciencelab.org/float
Study Design

• Approximation of practice
  – Pose questions that might arise during software development

• Avoid prompting or anchoring
  – Don’t test if they remember terminology, test if they can see the concept
    • In a snippet of code...
    • In a choice of optimization option...
    • In an intern’s question...
Core Quiz

• Floating point arithmetic is not real number arithmetic, even though it looks like it
  – Commutativity, associativity, distributivity, ordering, identity, negative zero, overflow, NaN, operation precision, denormalized numbers, signaling...

• Floating point does not behave like computer integer arithmetic either...
  – Overflow (saturation), underflow, NaN, signaling
Floating Point Questions

The following questions ask about your understanding of floating point behavior in code. The syntax used is C/C++, and the syntax in Java or C# is identical or similar.

Assume that variables a, b, and c are floating point variables.

== is the equality operator in C/C++

if a and b are numbers, it is always the case that (a + b) == (b + a)

- True, it is always the case
- False, sometimes it is not the case
- I don't know
Optimization Quiz

• Hardware features change standard compliance
  – MADD, Flush-to-Zero

• Compiler optimizations change standard compliance
  – What’s the highest –O level that is standard compliant?
  – Is --fast-math standards compliant?

• Options and features can break compliance
Floating Point Optimizations

IEEE floating point behavior is standardized to help make analysis of floating point code tractable. Hardware implementations and compilers generally obey the standard, but it is possible for them to be configured not to. This is typically done in the context of compiler optimization, where some optimizations use non-standard behavior or non-standard elements of the hardware in order to compute faster.

This set of questions tests your understanding of what is standard behavior and what is not.

The floating point fused multiply-add operation integrates multiplication and addition into a single operation for faster operation and higher precision.

- It is part of the standard
- It is not part of the standard
- I don't know
Suspicion Quiz

• Floating point condition codes can point to numeric problems

• How suspicious should you be of your results when your code produces a...
  – Overflow, underflow, precision (rounding), invalid (NaN), or denormalized result

• Lack of suspicion may mean bad results get through
Floating point condition code interpretation

IEEE floating point hardware records exceptional conditions as they occur. Using this capability, it is possible to determine whether any of a set of possible exceptional conditions occurred one or more times during the execution of a function.

Suppose you are given the following code:

```c
int scientific_simulation()
{
    clear_floating_point_exceptional_conditions();

    do_scientific_simulation();

    conditions = get_floating_point_exceptional_conditions();
}
```

For each of the following exceptional conditions, indicate how suspicious you would be of the function `do_scientific_simulation()`'s results if the condition occurred. This is not a ranking.

There are no "right" answers for these questions.

Overflow - the result of some floating point instruction in `do_scientific_simulation()` was an infinity.

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Low Suspicion</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Participant Recruitment Goals

• PhD student or above
• Actively involved in software development or management for science and engineering
  – Both as main and secondary roles
• Universities, national labs, and industry

Biggest Caveat:  Not a random sample
Participant Recruitment Process

• **Standardized email sent to seed recipients**
  – Relevant department chairs, center directors, faculty, postdocs, and Ph.D. students at NU
  – Highest-level personal contacts at national labs
  – Faculty contacts at >20 universities

• **Request to take survey and forward email only to people relevant to our recruitment goals**
Participant Background / Factors

• **Anonymity**

• **199 Participants**
  – Plus additional 52 undergrads for suspicion quiz

• **11 factors (self-reported)**
  – 2 pages of details in paper

• **Factors matter much less than expected**
  – Will highlight a few as we go on
Prepare to be Scared
We believe that the combination of our reported codebases in which floating point was not involved. Over 2/3 have been involved with a codebase of at least 10,000 lines of code, and floating point was intrinsic to almost 2/3 of those codebases. Over 2/3 have experience with arbitrary precision languages/libraries, with Mathematica being by far the most common (over 1/3).

Over 2/3 have experience with arbitrary precision floating point, most commonly one or more lectures in a course. The participants have experience with floating point in 55 different languages, with Python, C, C++ or mentor. The participants (within the codebase they built (Figure 8.) illustrate here suggest that our sample is a good representation of software developers who write code for, and in support of, science and engineering applications. We now summarize the main results of our analysis.
Experience With Code Matters (slightly)

<table>
<thead>
<tr>
<th>Experience With Code</th>
<th># Correct</th>
<th># Incorrect</th>
<th># Don't Know</th>
<th># Unanswered</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;1M</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100K-1M</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10K-100K</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1K-10K</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100-1K</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

# Chance
### Question Area Analysis

<table>
<thead>
<tr>
<th>Question</th>
<th>% Correct</th>
<th>% Incorrect</th>
<th>% Don't Know</th>
<th>% Unanswered</th>
</tr>
</thead>
<tbody>
<tr>
<td>Commutativity</td>
<td>53.3</td>
<td>27.6</td>
<td>18.6</td>
<td>0.5</td>
</tr>
<tr>
<td>Associativity</td>
<td>69.3</td>
<td>14.1</td>
<td>15.6</td>
<td>1.0</td>
</tr>
<tr>
<td>Distributivity</td>
<td>81.9</td>
<td>6.0</td>
<td>10.6</td>
<td>1.5</td>
</tr>
<tr>
<td>Ordering</td>
<td>80.4</td>
<td>6.0</td>
<td>12.6</td>
<td>1.0</td>
</tr>
<tr>
<td>Identity</td>
<td>16.6</td>
<td>76.9</td>
<td>5.5</td>
<td>1.5</td>
</tr>
<tr>
<td>Negative Zero</td>
<td>58.8</td>
<td>28.1</td>
<td>11.6</td>
<td>1.5</td>
</tr>
<tr>
<td>Square</td>
<td>47.2</td>
<td>35.2</td>
<td>16.6</td>
<td>1.0</td>
</tr>
<tr>
<td>Overflow</td>
<td>60.8</td>
<td>24.1</td>
<td>11.1</td>
<td>4.0</td>
</tr>
<tr>
<td>Divide by Zero</td>
<td>11.6</td>
<td>76.4</td>
<td>11.1</td>
<td>1.0</td>
</tr>
<tr>
<td>Zero Divide By Zero</td>
<td>70.4</td>
<td>9.0</td>
<td>19.6</td>
<td>1.0</td>
</tr>
<tr>
<td>Saturation Plus</td>
<td>54.8</td>
<td>26.1</td>
<td>17.6</td>
<td>1.5</td>
</tr>
<tr>
<td>Saturation Minus</td>
<td>53.3</td>
<td>25.6</td>
<td>19.6</td>
<td>1.5</td>
</tr>
<tr>
<td>Denormal Precision</td>
<td>52.3</td>
<td>24.6</td>
<td>22.1</td>
<td>1.0</td>
</tr>
<tr>
<td>Operation Precision</td>
<td>73.4</td>
<td>9.0</td>
<td>16.6</td>
<td>1.0</td>
</tr>
<tr>
<td>Exception Signal</td>
<td>69.3</td>
<td>10.1</td>
<td>19.6</td>
<td>1.0</td>
</tr>
</tbody>
</table>

**Figure 14:** Core quiz questions. Boldfaced questions were answered correctly at the level of chance. Italicized questions were answered incorrectly or reported as unknown more often than answered correctly.

### Optimization Quiz

<table>
<thead>
<tr>
<th>Question</th>
<th>% Correct</th>
<th>% Incorrect</th>
<th>% Don't Know</th>
<th>% Unanswered</th>
</tr>
</thead>
<tbody>
<tr>
<td>MADD</td>
<td>15.6</td>
<td>10.0</td>
<td>72.4</td>
<td>2.0</td>
</tr>
<tr>
<td>Flush to Zero</td>
<td>13.6</td>
<td>7.5</td>
<td>76.9</td>
<td>2.0</td>
</tr>
<tr>
<td>Standard-compliant Level</td>
<td>8.5</td>
<td>20.7</td>
<td>68.8</td>
<td>2.0</td>
</tr>
<tr>
<td>Fast-math</td>
<td>29.1</td>
<td>3.0</td>
<td>65.8</td>
<td>2.0</td>
</tr>
</tbody>
</table>

**Figure 15:** Optimization quiz questions. All questions were reported as unknown by more than half the participants.

### Area Effect on Core Quiz Scores

<table>
<thead>
<tr>
<th>Area</th>
<th># Correct</th>
<th># Incorrect</th>
<th># Don't Know</th>
<th># Unanswered</th>
</tr>
</thead>
<tbody>
<tr>
<td>EE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CS</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Math</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PhysSci</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Eng</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 17:** Effect of Area on core quiz scores.

### Software Development Role Effect on Core Quiz Scores

<table>
<thead>
<tr>
<th>Role</th>
<th># Correct</th>
<th># Incorrect</th>
<th># Don't Know</th>
<th># Unanswered</th>
</tr>
</thead>
<tbody>
<tr>
<td>SWEng</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SWEngMgr</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SupMain</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SupMainMgr</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unspecified</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 18:** Effect of Software Development Role on core quiz scores.

### Formal Training (in Floating Point) Effect on Core Quiz Scores

<table>
<thead>
<tr>
<th>Training</th>
<th># Correct</th>
<th># Incorrect</th>
<th># Don't Know</th>
<th># Unanswered</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt;=week</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt;=course</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt;=lecture</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 19:** Effect of Formal Training (in floating point) on core quiz scores.

One might hope that Formal Training (in floating point) would have a considerable effect. While it does have an effect (as illustrated in Figure 19, it is not a large one. The maximum gain over the baseline is only about 1/15, and the variation is about 2/15.

The factors we have not discussed thus far, Position, Informal Training (in Floating Point), Floating Point Language Experience, and Arbitrary Precision Language Experience, have minimal or ambiguous effects on core quiz scores. Note that the participant supplies lists for some of these factors. What seems to be true about them is that very short lists predict bad scores. That is, if the participant has reported some kind of Informal Training, etc, it does not seem to matter what it is. A possible explanation is that these are driven by the two Codebase Size factors. If a participant wrote or was involved with a codebase of any significant size, they probably experienced a range of languages and floating point experiences.
Now Some Good News for Correctness, but Bad News for Innovation
Participants Aware of Not Understanding Optimizations (HW/ SW)

![Bar chart showing the number of questions correct and incorrect for different disciplines.](chart.png)

**Observation:**
Many developers do not understand core floating point behavior particularly well, yet believe they do. This suggests that some existing and future codebases may have hidden numeric correctness issues. This is probably more likely to be the case in smaller and newer projects where there is no specialist whose role is in part to mitigate these issues. As use of floating point rapidly expands outside of the traditional domains of science and engineering, the problem is likely becoming widespread.

**Action:**
The HPC community should make an effort to make developers in general more suspicious about floating point behavior. The analogy might be how the programming languages and operating systems communities have raised awareness about C's undefined behavior and its interaction with modern compilers [12], [14].

**Action:**
Although our study found that formal training in floating point has only a small effect on understanding, we believe the issue is not that training does not work per se, but rather that the community has just not found the right training approach yet. A rigorous process to develop effective training for a broad range of developers is an action that the HPC community, for example via SIGHPC, could undertake. We would then also need to convince the broader (and ever expanding) non-CS community of developers that such training is necessary.

**Action:**
Static and dynamic analysis tools that can examine existing codebases and point developers to potentially suspicious code would likely have significant impact. Several such tools exist [1], [11], [8], but the tools would also need...
Participants Aware of Not Understanding Optimizations (HW/SW)

"don’t know"

<table>
<thead>
<tr>
<th></th>
<th># Correct</th>
<th># Incorrect</th>
<th># Don't Know</th>
<th># Unanswered</th>
</tr>
</thead>
<tbody>
<tr>
<td>EE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CS</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Math</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PhysSci</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Eng</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 20: Effect of Area on optimization quiz scores.

Figure 21: Effect of Software Development Role on optimization quiz scores.

C. Factor analysis for optimization quiz

The main story about the optimization quiz is the dominance of the response "Don't Know" regardless of how the data is sliced by the factors.

Only the factors Software Development Role (Figure 21) and Area (Figure 20) appear in our data to have an effect on optimization quiz scores. Even there, the effects cap quickly (0.7/3 above chance for Role and 0.5 above chance for Area), although the variation is considerable (1.4/3 for Role and 0.8/3 for Area). The effect of Informal Training (in floating point) is ambiguous in our data, although it could be interpreted as producing considerable variation, albeit the maximum effect we see is only slightly better than chance.

It is surprising that factors like the codebase sizes and floating point experience within codebases have no effect here. On the other hand, the limited impact of Formal Training (in floating point) could be explained by the fact that most of those who have received training in our sample have received a level similar to that of an introductory computer systems or machine organization course. Such an introduction will not touch on optimizations at all.

D. Suspicion analysis

There is of course no ground truth for the suspicion quiz component of our survey, and it really depends on the application. However, as we described earlier, some exceptional conditions are generally more suspicious than others—an arguably reasonable ranking is that generating a NaN (Invalid) is by far more suspicious than generating an infinity (Overflow), which is in turn much more suspicious than generating any of the other three conditions.

Figure 22(a) shows the distribution of reported suspicion for the five exceptional conditions within our 199 participant main group, while Figure 22(b) shows the corresponding distribution for our separate 52 participant student group. The groups behave quite similarly, although the student group is overall less suspicious about Underflow and De-norm, possibly because the topic is fresh in their minds given the course. The student group is also less suspicious of Overflow.

As we might hope, both groups do tend to be more suspicious of Invalid and Overflow than the other conditions. However, consider Invalid more carefully: About 1/3 of both groups reported a suspicion level less than the maximum for a computation that somewhere encountered a NaN!

V. CONCLUSIONS

Stepping back from the data and analysis, we believe that some generalizations can be made, along with actions to address them.

Observation: Many developers do not understand core floating point behavior particularly well, yet believe they do. This suggests that some existing and future codebases may have hidden numeric correctness issues. This is probably more likely to be the case in smaller and newer projects where there is no specialist whose role is in part to mitigate these issues. As use of floating point rapidly expands outside of the traditional domains of science and engineering, the problem is likely becoming widespread.

Action: The HPC community should make an effort to make developers in general more suspicious about floating point behavior. The analogy might be how the programming languages and operating systems communities have raised awareness about C's undefined behavior and its interaction with modern compilers [12], [14].

Action: Although our study found that formal training in floating point has only a small effect on understanding, we believe the issue is not that training does not work per se, but rather that the community has just not found the right training approach yet. A rigorous process to develop effective training for a broad range of developers is an action that the HPC community, for example via SIGHPC, could undertake. We would then also need to convince the broader (and ever expanding) non-CS community of developers that such training is necessary.

Action: Static and dynamic analysis tools that can examine existing codebases and point developers to potentially suspicious code would likely have significant impact. Several such tools exist [1], [11], [8], but the tools would also need...
Now Some News that is Hard to Characterize
Can you tell these graphs apart?

One is undergrads in an introductory systems course
Can you tell these graphs apart?

One is undergrads in an introductory systems course
for this that would not be gameable or too onerous to use?

they understand it. How can we create an effective interface not be able to use an optimization without demonstrating that developer knowledge into account—ideally, a developer would results behind it.

of optimizations may be leaving a hidden trail of incorrect what they think they know? If not, then the introduction as conservative about what they use as they are about mizations without knowing their consequences. Are they wittingly or unwittingly they break working kernel code or make it insecure.

leverage C’s undefined behavior are carefully avoided lest with the OS developer community, where optimizations that could simply avoid them out of fear of incorrect results, them without understanding the consequences, or developers them without knowing the consequences, or developers a particularly paranoid developer arbitrary precision arithmetic is too thick. A system that would to easily sanity check the behavior of their code (and any compiled to use arbitrary precision would enable developers.

optimizations they chose). A particularly paranoid developer to have interfaces suitable for a non-CS community and covered by implementing an exception handler that runs a monitoring tool to spy on unmodified binaries and track exceptional conditions using floating point condition codes, as covered by implementing an exception handler that runs a monitoring tool to spy on unmodified binaries and track.

Coverity [2] will expand their purview to include floating point. We ourselves have been developing a simple runtime...

Figure 22: Distribution of suspicion for different exceptional conditions.

Percent Reporting

100

80

60

40

20

0

Suspicion Level

1

2

3

4

5

1/3 do not find NaN Maximally Suspicious

1/3 do not find NaN

Maximally Suspicious

Overflow

Underflow

Precision

Invalid

Denorm

1/3 do not find NaN

Maximally Suspicious

MAXIMALLY SUSPICIOUS
Caveats

• Participants are not a random sample
• Anonymity and self-reporting
  – We cannot be sure we have hit our recruitment goals
• Confusion/lack of time for participant
  – Survey design was iterated based on feedback
• Only 199+52 data points
  – But these are users
• ...

29
Potential Actions

• HPC community should *sow suspicion*
  – Much like PL and compilers community did with undefined behavior in C

• HPC community should *develop better training*
Potential Actions

• Better static/dynamic analysis tools
  – Work in progress

• Blurring the boundary between FP and arbitrary precision arithmetic
  – Work in progress

• Developer knowledge-limited access to software and hardware optimizations
  – “Achievement Unlocked”
  – Work in progress
A Work in Progress: FPSpy

• User-level shim that *slides underneath existing, unmodified application binary*
  – Gets out of the way on conflict with application

• Uses FP hardware features, Linux FP interfaces, and debugger-style techniques to track issues
  – Aggregate mode:
    • FP condition codes set at any point during execution
    • Fast – zero overhead
  – Individual mode:
    • Instruction-level tracking of FP condition codes
    • Slower

• Current: applying FPSpy and other tools to *study existing, unmodified applications*
  – Does developer confusion as measured in present study manifest in codes in current use?
A Work In Progress: FPKernel

• Floating point exceptions have much lower latency and overhead in a kernel-only model
  – Like our Hybrid Run-Time (HRT) scheme and the Nautilus Kernel Framework that supports it
• Combine fixed precision hardware FP and arbitrary precision software FP to create simple arithmetic model for programmer
  – FP exceptions trigger transition to software FP
  – NaN boxing / signaling NaN for values
  – Made more practical by fast FP exceptions
Minimum Time to Floating Point Exception Handler
Linux User-level Versus Nautilus Kernel-level

- Min: kernel-level is 6.5-30x faster
- Median: kernel-level is 3-30x faster
- Variance: kernel-level is 17-95x better
Paper in a Nutshell: Not Really

• Targeted survey
  – Aimed at practitioners likely to use FP
  – Quizzes for core, optimization, and suspicion of results
    – *First study of this kind*

• Participants do *only slightly better than chance* on core concepts
  – ... and don’t know it
  – Some factors mitigate, but none particularly well

• Participants *do not understand* optimization concepts
  – ... and do know it

• Participants *less suspicious* than they should be
  – ... but similar to students in a sophomore course

• Maybe systems software can do something about it
For More Information

• Peter Dinda
  – pdinda@northwestern.edu
  – http://pdinda.org

• Conor Hetland
  – ConorHetland2015@u.northwestern.edu

• Take the survey
  – http://presciencelab.org/float

• Prescience Lab
  – http://presciencelab.org

• Acknowledgements
  – NSF, DOE