ARGONNE
ATPESC2023
EXTREME-SCALE COMPUTING

Putting it All Together: (the Sociology of) Using Numerical Packages In Practice

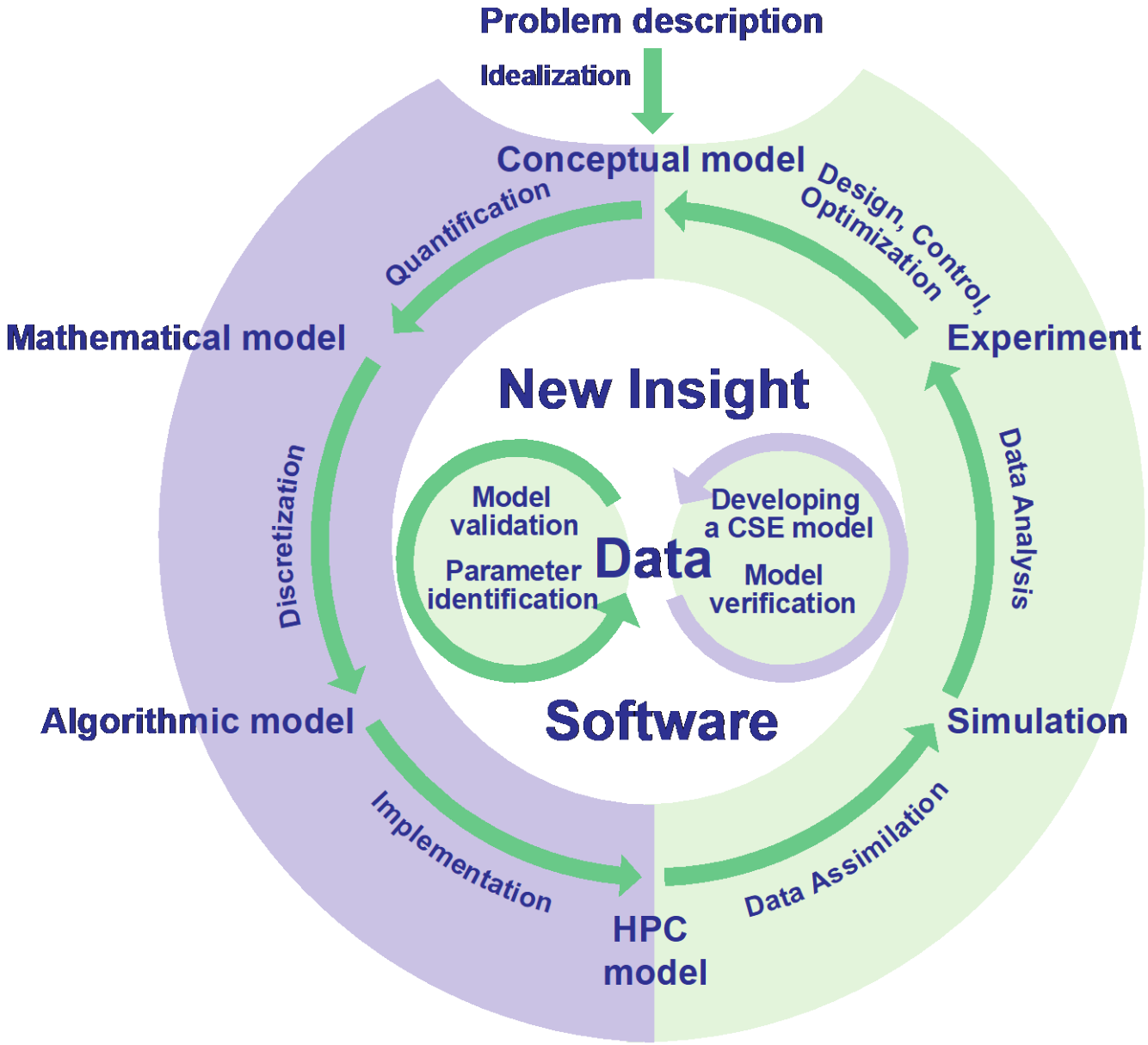
Ann Almgren

On behalf of your Track 5 Presenters

Here is one view of the “CSE cycle”:

None of us can actually do it all

So where do you want to spend your time?



Key steps of simulation science application development

- Physical model
  - Expertise may be very domain-specific
- Mathematical model
  - Expertise may require detailed mathematical knowledge
- Discretization and algorithm development
  - Expertise includes knowing regimes of applicability, stability, approximation, error bounds
- Parallel GPU-aware implementation
  - Expertise in hardware, software stack and parallel programming models

That’s a lot of expertise!

Very few of us are experts in all of these areas. So how do we optimize the insight/impact of our computational science?

- Team science – in an ideal world we could work in teams that have all the relevant expertise within one team

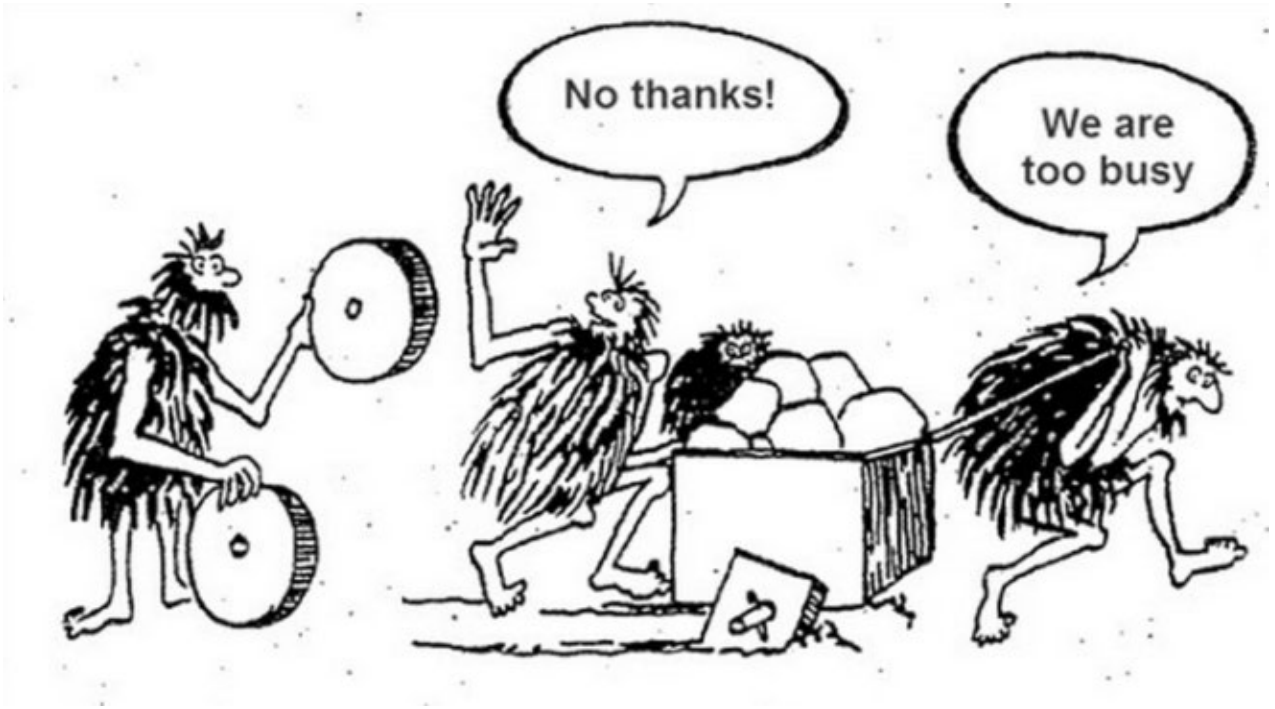
- That’s not always possible –so one way to broadly share expertise is through software libraries
  - Expertise in discretization and algorithm development
  - Expertise in hardware, software stack and parallel programming models

In the short-term we often prefer to do things ourselves

$$\nabla^2 T = 0 \in \Omega$$

$$T(0) = 180^o$$

$$T(1) = 0^o$$



For the 1-D heat equation why bother learning a software package?

Sometimes simple is good

We can prototype in matlab, build simple serial implementations, and demonstrate proof-of-concept.

This can be good:

- New algorithms are often designed and validated in this mode.
- Sometimes writing your own version of a known technology (e.g. multigrid solver) is worth it -- “learning by doing”

This can be bad:

- Our own implementations are more likely to lack generality, be inefficient or even buggy.
- How much time do we spend “reinventing the wheel?”

Sometimes simple is good

We can prototype in matlab, build simple serial implementations, and demonstrate proof-of-concept.

This can be good:

- New algorithms are often designed and validated in this mode.
- Sometimes writing your own version of a known technology (e.g. multigrid solver) is worth it -- “learning by doing”

This can be bad:

- Our own implementations are more likely to lack generality, be inefficient or even buggy.
- How much time do we spend “reinventing the wheel?”
- **Do we impact anyone/anything beyond our own immediate application?**

The “supply” side of software libraries

Software libraries/frameworks/tools are made by real people.

The people aspect matters

- Software developers know a lot about their product
- But they don’t necessarily know exactly what you need

Communication / Collaboration is an important part of the process it’s good for the developer as well as the user!

Why don’t people “just” use software libraries

Lack of knowledge – how do you know whether the right tool even exists?

And if it exists: Where do you find it? How do you use it? Will it work with your other tools?

AKA: “package fatigue”

Why don’t people “just” use software libraries

Frustration! It can be really frustrating to not have the tool do what you want as well as you want. And how do you tell whether it’s you or the tool?

So how can you find the right tool – if it exists - and how do you learn how to use it correctly?

Ideal solution: a “toolbox” of compatible (interoperable) tools that “just work”

- This is exactly what the software developers are working towards
- But it takes time and resources
- The developer/user interaction can be a win-win

On a practical level, there are trade-offs

Advantages

- Key challenges addressed well
  - Portable, Performant, Scalable, Interoperable
- Numerics are well tested/vetted
- Functionality is often more general than you would have made yourself
- More science, more impact; less time writing/debugging software
- Become part of a community – for collaboration and help

On a practical level, there are trade-offs

Challenges

- Something new to learn
- Hard to predict show-stoppers
- Not always plug-n-play
- Trusting the work of others
- Overhead of collaborating
- Funding priorities

How do we tip the balance?

Challenge		Mitigation
Something new to learn		Many examples and documentation
Hard to predict show-stoppers		Engage package developers early
Not always plug-n-play		Submit build issues
Trusting others		Identify or develop tests
Overhead of collaborating		Builds relationships
Funding priorities		Add to the package yourself

How do we tip the balance?

Challenge		Mitigation
Something new to learn	⟷	Many examples and documentation
Hard to predict show-stoppers	⟷	Engage package developers early
Not always plug-n-play	⟷	Submit build issues
Trusting others	⟷	Identify or develop tests
Overhead of collaborating	⟷	Builds relationships
Funding priorities	⟷	Add to the package yourself

The point of open source is to encourage use

Package teams want users to make progress.

If package is missing a crucial feature, ask.

Critical factors that determine whether you will use an open-source library or tool

- **Licensing:** The library or tool must have acceptable license constraints and restrictions. It should be compatible with the project's or organization's licensing terms.
- **Functionality:** The tool or library should be able to fulfill specific needs and improve productivity.
- **Developer and Community Support:** The tool or library should have ongoing support from its developers, including regular updates and bug fixes. A robust user and developer community is also crucial for problem-solving and help.
- **Maturity and Documentation:** The tool or library should have reached a level of maturity, proven stability and should have comprehensive documentation to assist its users.
- **Compatibility and Portability:** The library or tool should support the programming languages used in the project and be portable across different platforms.
- **Quality:** The library or tool should have a high standard of development quality, including extensive testing and active maintenance on bug reports and pull requests.
- **Ease of Use and Learning Curve:** The tool or library should be easy to use, have a short learning curve, and be capable of integration with other tools and libraries.
- **Active and Welcoming Development Team:** The team behind the library or tool should value user feedback and provide a supportive environment for users.
- **Performance:** The library or tool should deliver high performance and low latency.
- **Sustainability:** There should be recent development activity and a long-term support model, indicating a high likelihood that development will continue.
- **Vendor Support:** For some users, particularly in High Performance Computing (HPC) environments, vendor support may be a significant factor.
- **Part of a Larger Ecosystem:** The tool or library should preferably be part of a larger, friendly user base, like StackOverflow or other forums, which can offer additional support and resources.