Formal/Symbolic/Numerical Computational Methods

Michael R. Douglas

CMSA, Harvard University *and* Simons Center, Stony Brook University

AITP, September 2020

Abstract

We discuss applications of ITP, AITP and related technologies:
Facilitate more readable formal proofs;
Support combining ML and symbolic computation to solve math problems and fit data;
Enable scientific computation which combines formal, symbolic and numerical methods.

Who am I ?

I'm a mathematical physicist and string theorist who has followed CS and AI for a long time, starting with a year working for Gerry Sussman and studying with John Hopfield in 1988. I have written computational papers, rigorous papers and several papers on the interface of physics and CS.

From 2012–2020 I was a researcher at Renaissance Technologies, one of the oldest and most successful quantitative hedge funds. We used a fair amount of machine learning, but I won't go into details.

In 2017 the success of AlphaGo convinced me that AI is making far faster progress than ever before, and is starting to achieve (super)human ability in tasks which we would think require reasoning. This led me to start thinking about how scientists like myself could use it to help us do research, to start coming to AITP and other meetings, and to learn from leaders in these fields.
Recently I left Rentec to pursue this full-time.

Who am I ?

I'm a mathematical physicist and string theorist who has followed CS and AI for a long time, starting with a year working for Gerry Sussman and studying with John Hopfield in 1988. I have written computational papers, rigorous papers and several papers on the interface of physics and CS.

From 2012–2020 I was a researcher at Renaissance Technologies, one of the oldest and most successful quantitative hedge funds. We used a fair amount of machine learning, but I won't go into details.

In 2017 the success of AlphaGo convinced me that AI is making far faster progress than ever before, and is starting to achieve (super)human ability in tasks which we would think require reasoning. This led me to start thinking about how scientists like myself could use it to help us do research, to start coming to AITP and other meetings, and to learn from leaders in these fields.
Recently I left Rentec to pursue this full-time.

Who am I ?

I'm a mathematical physicist and string theorist who has followed CS and AI for a long time, starting with a year working for Gerry Sussman and studying with John Hopfield in 1988. I have written computational papers, rigorous papers and several papers on the interface of physics and CS.

From 2012–2020 I was a researcher at Renaissance Technologies, one of the oldest and most successful quantitative hedge funds. We used a fair amount of machine learning, but I won't go into details.

In 2017 the success of AlphaGo convinced me that AI is making far faster progress than ever before, and is starting to achieve (super)human ability in tasks which we would think require reasoning. This led me to start thinking about how scientists like myself could use it to help us do research, to start coming to AITP and other meetings, and to learn from leaders in these fields.
Recently I left Rentec to pursue this full-time.

My other talks on this topic are at
https://cbmm.mit.edu/news-events/events/brains-minds-machines-seminar-series-how-will-we-do-mathematics-2030 and
https://av.tib.eu/media/47991?hl=icms+2020+douglas .

Goals of the present talk:
It seems to me (and many have said) that AITP could use some easier goals than "make an AI which achieves human level performance in mathematics." This may happen by 2030, or 2040 – I would certainly bet before 2050. But ambitious research goals should be balanced by other projects which are clearly doable, advance the state of the art, and which impact some user community. Of course you have many of these, but let me try to suggest more, challenging but easier than AGI or human level math.

I was also asked to comment on barriers encountered by researchers in my fields to using ITP, AITP and other technologies developed by the computer math community. Any criticisms are offered in that spirit.

My other talks on this topic are at
https://cbmm.mit.edu/news-events/events/brains-minds-machines-seminar-series-how-will-we-do-mathematics-2030 and
https://av.tib.eu/media/47991?hl=icms+2020+douglas .

Goals of the present talk:
It seems to me (and many have said) that AITP could use some easier goals than "make an AI which achieves human level performance in mathematics." This may happen by 2030, or 2040 – I would certainly bet before 2050. But ambitious research goals should be balanced by other projects which are clearly doable, advance the state of the art, and which impact some user community. Of course you have many of these, but let me try to suggest more, challenging but easier than AGI or human level math.

I was also asked to comment on barriers encountered by researchers in my fields to using ITP, AITP and other technologies developed by the computer math community. Any criticisms are offered in that spirit.

What is AITP ?

As discussed at this conference, it covers more or less any way to use ML and other modern AI methods to do computer math. Much of it draws analogies between mathematical problems and standard problems in ML:

- Extrapolating intricate mathematical functions (later we will discuss cohomology calculations) – supervised learning, try MLP's or similar models.
- Mathematics as language and linguistic transformation – try language models such as RNN's and transformers to learn the grammar and transformations.
- Mathematics as exploration of a space of concepts; proof as a game of solitaire whose moves are logical deductions – try RL.

All of these are good metaphors but the last one seems the best to me. But the space of concepts is far larger than a typical game state space, and its geometry and structure are as yet totally unknown.

Many of the successes of machine learning for mathematics I have heard about fall into these three categories:

1. Automation in theorem provers – ranging from ITP tools such as better tactics and library search for relevant lemmas, all the way to ATP.
2. Heuristics to solve well-posed problems, usually in NP so that proposed solutions can be verified. Even if it is known how to solve the problem, a solution which does not require time-consuming programming can be very attractive. Examples: Lample and Charton's symbolic integrator; work of many string theorists on ML learning Calabi-Yau structure (more about this below).
3. Search engines – for example MATHWEBSEARCH of Kohlhase *et al* and SEARCHONMATH of Gonzaga *et al*.

The outline of the rest of the talk:

- Describe ITP and obstacles to its use by mathematical scientists (starting with myself as a test case). I agree with Hales, Paulson, and others that readability and structure of ITP proofs is a central issue.
- Give examples of (2), heuristic ML solutions to problems. This is of growing interest in physics and other mathematical sciences, and perhaps AITP can help make it easier to do.
- Describe how (1) and (3) could be combined to aid in scientific programming, using code search, specifications and refinement, and “call by specification.”

When would mathematical scientists want to use a computational technology?

Axes along which to evaluate mathematical work:

1. exploratory – established – central, foundational
2. simple – intricate
3. low need for certainty – mission critical; human lives depend on it
4. used directly by humans – used to enable other computations

Much discussion of ITP focuses on (3), but there are not so many mission critical results in the basic sciences. We are very used to working with conjectures, uncertainty and unproven claims.
(But see Kevin Buzzard's colloquium for the opposite point of view.)

Axis (4), reuse in other computational work, brings many new opportunities. In a research project which uses many previous results from diverse sources, integrating the previous work into one's project can be very time-consuming. Later I will explain how I think ITP and search could help with this.

When would mathematical scientists want to use a computational technology?

Axes along which to evaluate mathematical work:

1. exploratory – established – central, foundational
2. simple – intricate
3. low need for certainty – mission critical; human lives depend on it
4. used directly by humans – used to enable other computations

Much discussion of ITP focuses on (3), but there are not so many mission critical results in the basic sciences. We are very used to working with conjectures, uncertainty and unproven claims.
(But see Kevin Buzzard's colloquium for the opposite point of view.)

Axis (4), reuse in other computational work, brings many new opportunities. In a research project which uses many previous results from diverse sources, integrating the previous work into one's project can be very time-consuming. Later I will explain how I think ITP and search could help with this.

When would mathematical scientists want to use a computational technology?

Axes along which to evaluate mathematical work:

1. exploratory – established – central, foundational
2. simple – intricate
3. low need for certainty – mission critical; human lives depend on it
4. used directly by humans – used to enable other computations

Much discussion of ITP focuses on (3), but there are not so many mission critical results in the basic sciences. We are very used to working with conjectures, uncertainty and unproven claims.
(But see Kevin Buzzard's colloquium for the opposite point of view.)

Axis (4), reuse in other computational work, brings many new opportunities. In a research project which uses many previous results from diverse sources, integrating the previous work into one's project can be very time-consuming. Later I will explain how I think ITP and search could help with this.

Comments on ITP

Automated and interactive theorem proving have a long and distinguished history, and automated SAT solvers, interactive provers such as Coq, Isabelle and Lean are standard tools in industry.

ITP languages are now able to express all of research level mathematics in a formally precise yet flexible way. Several large theorems (four-color, odd order, Hales-Kepler) have been verified. The Archive of Formal Proofs (Isabelle) has over 500 articles with over 150K lemmas. Lean Mathlib has over 36K lemmas and is growing fast. Still, ITP is much harder to learn to use than (loosely) comparable technologies such as computer algebra. Arguably more problematic is that ITP code is hard to read and does not convey concepts well. One description of the state of ITP I got from several experts is that formalizing a good math textbook (say 100 pages) can be done by an expert in a year. This seems pretty good to me as the textbook itself took a year to write. If the results were fully satisfactory, more mathematicians would learn ITP and formalize their textbooks.

Comments on ITP

Automated and interactive theorem proving have a long and distinguished history, and automated SAT solvers, interactive provers such as Coq, Isabelle and Lean are standard tools in industry.
ITP languages are now able to express all of research level mathematics in a formally precise yet flexible way. Several large theorems (four-color, odd order, Hales-Kepler) have been verified. The Archive of Formal Proofs (Isabelle) has over 500 articles with over 150K lemmas. Lean Mathlib has over 36K lemmas and is growing fast.
Still, ITP is much harder to learn to use than (loosely) comparable technologies such as computer algebra. Arguably more problematic is that ITP code is hard to read and does not convey concepts well.
One description of the state of ITP I got from several experts is that formalizing a good math textbook (say 100 pages) can be done by an expert in a year. This seems pretty good to me as the textbook itself took a year to write. If the results were fully satisfactory, more mathematicians would learn ITP and formalize their textbooks.

Comments on ITP

Automated and interactive theorem proving have a long and distinguished history, and automated SAT solvers, interactive provers such as Coq, Isabelle and Lean are standard tools in industry.
ITP languages are now able to express all of research level mathematics in a formally precise yet flexible way. Several large theorems (four-color, odd order, Hales-Kepler) have been verified. The Archive of Formal Proofs (Isabelle) has over 500 articles with over 150K lemmas. Lean Mathlib has over 36K lemmas and is growing fast.
Still, ITP is much harder to learn to use than (loosely) comparable technologies such as computer algebra. Arguably more problematic is that ITP code is hard to read and does not convey concepts well.
One description of the state of ITP I got from several experts is that formalizing a good math textbook (say 100 pages) can be done by an expert in a year. This seems pretty good to me as the textbook itself took a year to write. If the results were fully satisfactory, more mathematicians would learn ITP and formalize their textbooks.

Comments on ITP

Automated and interactive theorem proving have a long and distinguished history, and automated SAT solvers, interactive provers such as Coq, Isabelle and Lean are standard tools in industry.
ITP languages are now able to express all of research level mathematics in a formally precise yet flexible way. Several large theorems (four-color, odd order, Hales-Kepler) have been verified. The Archive of Formal Proofs (Isabelle) has over 500 articles with over 150K lemmas. Lean Mathlib has over 36K lemmas and is growing fast.
Still, ITP is much harder to learn to use than (loosely) comparable technologies such as computer algebra. Arguably more problematic is that ITP code is hard to read and does not convey concepts well.
One description of the state of ITP I got from several experts is that formalizing a good math textbook (say 100 pages) can be done by an expert in a year. This seems pretty good to me as the textbook itself took a year to write. If the results were fully satisfactory, more mathematicians would learn ITP and formalize their textbooks.

Let me say a few words about my recent experience with Lean.
First, many thanks to Floris van Doorn for tutoring and collaboration.

In March, Lean mathlib did not include any representation theory of finite groups, so we set out to formalize Maschke's theorem.
This is the first nontrivial theorem in the subject and states that linear representations are uniquely decomposable into a direct sum of irreducible representations.

The definition of linear group representation (a homomorphism into a subgroup of $GL(V)$) and all of its prerequisites were there:

```
/-- A representation of a group `G` on an `R`-module `M` is a group homomorphism from `G` to
  `GL(M)`. Normally `M` is a vector space, but we don't need that for the definition. -/
@[derive inhabited] def group_representation (G R M : Type*) [group G] [ring R] [add_comm_group M]
  [module R M] : Type* :=
G →* M →□[R] M
```

(Note: the arrow+box is "$\to \iota$" which denotes a linear transformation.)

Let me say a few words about my recent experience with Lean.
First, many thanks to Floris van Doorn for tutoring and collaboration.

In March, Lean mathlib did not include any representation theory of finite groups, so we set out to formalize Maschke's theorem.
This is the first nontrivial theorem in the subject and states that linear representations are uniquely decomposable into a direct sum of irreducible representations.

The definition of linear group representation (a homomorphism into a subgroup of $GL(V)$) and all of its prerequisites were there:

```
/-- A representation of a group `G` on an `R`-module `M` is a group homomorphism from `G` to
  `GL(M)`. Normally `M` is a vector space, but we don't need that for the definition. -/
@[derive inhabited] def group_representation (G R M : Type*) [group G] [ring R] [add_comm_group M]
  [module R M] : Type* :=
G →* M →□[R] M
```

(Note: the arrow+box is "$\longrightarrow \iota$" which denotes a linear transformation.)

Let me say a few words about my recent experience with Lean.
First, many thanks to Floris van Doorn for tutoring and collaboration.

In March, Lean mathlib did not include any representation theory of finite groups, so we set out to formalize Maschke's theorem.
This is the first nontrivial theorem in the subject and states that linear representations are uniquely decomposable into a direct sum of irreducible representations.

The definition of linear group representation (a homomorphism into a subgroup of $GL(V)$) and all of its prerequisites were there:

```
/-- A representation of a group `G` on an `R`-module `M` is a group homomorphism from `G` to
  `GL(M)`. Normally `M` is a vector space, but we don't need that for the definition. -/
@[derive inhabited] def group_representation (G R M : Type*) [group G] [ring R] [add_comm_group M]
  [module R M] : Type* :=
G →* M →□[R] M
```

(Note: the arrow+box is "$\to \iota$" which denotes a linear transformation.)

Irreducible means that there are no nontrivial invariant subspaces. So, if V has an invariant subspace, we need to show that its complement is invariant; then use induction.

The informal proof is fairly short: given a subspace $W \subseteq V$ one can find a projection $\pi : V \to W$. Then one defines its average over the group,

$$\bar{\pi} \equiv \frac{1}{|G|} \sum_{g \in G} \rho(g^{-1}) \pi \rho(g),$$

and proves using algebra that this commutes with the group action. One can then show algebraically that this is also a projection $\bar{\pi}^2 = \bar{\pi}$, that $1 - \bar{\pi}$ is a projector onto the complement W', and that W' is invariant.

While the textbook and human intuition sufficed to fill in the intermediate steps and make all this precise, the project took about 3 man-weeks of our time (3 months of real time). The final statement was

```
theorem maschke [fintype G] (ρ : group_representation G R M) (N N' : submodule R M)
  (h : complementary N N') (hN : N.invariant_under ρ) (hG : is_unit (fintype.card G : R)) :
  ∃ N' : submodule R M, N'.invariant_under ρ ∧ complementary N N' :=
begin
  let π := invariant_projector hG ρ h.pr1, use ker π,
  use is_invariant_ker (is_equivariant_invariant_projector hG ρ h.pr1),
  suffices hR : (range (complementary.pr1 h)).invariant_under ρ,
  rw [complementary.comm, ← h.range_pr1, ← range_invariant_projector hG ρ h.pr1 h.pr1_pr1 hR],
  convert complementary_ker_range (is_projection_invariant_projector hG ρ h.pr1 h.pr1_pr1 hR),
  rw complementary.range_pr1, exact hN
end
```

This followed 150 lines of preliminaries, and some more general prerequisites (for example complementary subspaces and projectors).

Scott Morrison's solution:

```
/--
We construct our `k[G]`-linear retraction of `i` as
$$ \frac{1}{|G|} \sum_{g \in G} g⁻¹ • π(g • -). $$
-/
def equivariant_projection :
  W →◻[monoid_algebra k G] V :=
⅟(fintype.card G : k) • (sum_of_conjugates_equivariant π)
end

lemma equivariant_projection_condition (v : V) : (equivariant_projection π) (i v) = v :=
begin
  rw [equivariant_projection, linear_map_algebra_module.smul_apply, sum_of_conjugates_equivariant,
    equivariant_of_linear_of_comm_apply, sum_of_conjugates],
  rw [linear_map.sum_apply],
  simp only [conjugate_i π i h],
  rw [finset.sum_const, finset.card_univ,
    @semimodule.nsmul_eq_smul k _
      (restrict_scalars k (monoid_algebra k G) V) _ _ (fintype.card G) v,
    ←mul_smul, invertible.inv_of_mul_self, one_smul],
end
```

Plus about 100 additional lines, and a bit more to match our theorem.

Of course Maschke's theorem is very foundational and not in doubt. So what did we gain?

- I understand Maschke's theorem better, particularly the assumptions it uses, such as the condition that $|G|$ is invertible in the ground ring.
- Subsequent formal work can use Maschke's theorem.
- We have a certificate for Maschke's theorem and can convince anyone that it is true. Even better, they don't need to spend their own time reading and checking the proof, if they don't want to.
- The computer understands Maschke's theorem in some sense.

What does this last claim mean? How we could test it? It would be good to have more tasks to test – in NLP there are ~ 100 tasks and benchmark datasets.

On the negative side,

- Although three man-weeks is not much to prove a foundational result which will be used forever, it would be a lot if we were doing exploratory research and testing hypotheses. This is a typical use which researchers make of other mathematical software, and ITP suffers from the implicit comparison.
- The solutions are not very readable unless one invests the time to learn the definitions of a lot of prerequisites. Again, we can compare with computer algebra systems. To use them, one must also learn a programming language and read a manual chapter for each topic of interest. But the Mathematica or SageMath manuals for linear algebra and group theory (restricted to comparable coverage) are shorter than the Lean mathlib index, and stay closer to textbook definitions. This is because they provide only high level functionality, whereas mathlib is exposing many detailed lemmas which one might consider the "implementation."

To be fair, the implementation of these computer algebra packages would be far longer and much less clear. But most users don't need to understand the implementation, and one shouldn't have to learn about it to use high level functionality.

Even if developments in AITP and proof automation allowed us to leave out all of the tactics and their references to previous results, these proofs would still be hard to read. Solutions have been proposed to this problem:

- Structured proofs, as in Mizar and Isabelle/Isar.
- Controlled natural language, as in ForTheL and Colada.
- Note that proofs are easier to read in an interactive system that displays intermediate goals, references, *etc.*
- Could the computer answer questions and explain its proofs?

It would be very interesting to test these ideas on core textbook and research mathematics.

To be fair, the implementation of these computer algebra packages would be far longer and much less clear. But most users don't need to understand the implementation, and one shouldn't have to learn about it to use high level functionality.

Even if developments in AITP and proof automation allowed us to leave out all of the tactics and their references to previous results, these proofs would still be hard to read. Solutions have been proposed to this problem:

- Structured proofs, as in Mizar and Isabelle/Isar.
- Controlled natural language, as in ForTheL and Colada.
- Note that proofs are easier to read in an interactive system that displays intermediate goals, references, *etc.*
- Could the computer answer questions and explain its proofs?

It would be very interesting to test these ideas on core textbook and research mathematics.

I think the acid test would be to write an integrated textbook which both teaches a subject and contains machine readable proofs, which does not just interleave two independent presentations of the material, but in which the two presentations usefully refer to each other and support each other.
The interactivity of ITP should complement a traditional treatment, allowing the reader to ask questions and experiment.

In many subjects, and especially for mathematical scientists who apply the material, examples, algorithms and databases of results are as important as general theorems and rigorous proofs. Many of these have been incorporated into computer math platforms, for example SageMath has integrated the GAP group theory software. A really good integrated textbook would present this material as well.

This may be at or beyond the present state of the art – perhaps AITP could help make it possible.
(These ideas were developed in discussions with Jeremy Avigad and Kevin Buzzard).

I think the acid test would be to write an integrated textbook which both teaches a subject and contains machine readable proofs, which does not just interleave two independent presentations of the material, but in which the two presentations usefully refer to each other and support each other.
The interactivity of ITP should complement a traditional treatment, allowing the reader to ask questions and experiment.

In many subjects, and especially for mathematical scientists who apply the material, examples, algorithms and databases of results are as important as general theorems and rigorous proofs. Many of these have been incorporated into computer math platforms, for example SageMath has integrated the GAP group theory software. A really good integrated textbook would present this material as well.

This may be at or beyond the present state of the art – perhaps AITP could help make it possible.
(These ideas were developed in discussions with Jeremy Avigad and Kevin Buzzard).

I think the acid test would be to write an integrated textbook which both teaches a subject and contains machine readable proofs, which does not just interleave two independent presentations of the material, but in which the two presentations usefully refer to each other and support each other.
The interactivity of ITP should complement a traditional treatment, allowing the reader to ask questions and experiment.

In many subjects, and especially for mathematical scientists who apply the material, examples, algorithms and databases of results are as important as general theorems and rigorous proofs. Many of these have been incorporated into computer math platforms, for example SageMath has integrated the GAP group theory software. A really good integrated textbook would present this material as well.

This may be at or beyond the present state of the art – perhaps AITP could help make it possible.
(These ideas were developed in discussions with Jeremy Avigad and Kevin Buzzard).

Learning and heuristic algorithms

Over the years a number of systems have been developed to learn physical laws from data (Langley 1979, Falkenheimer and Michalski 1986, Schmidt and Lipson 2009, ...).

In doing physics, the difficulties of deciding what to measure, measuring it, choosing a subset of observations to try to model, *etc.* are far greater than those of fitting a model to data. Still, fitting models to data is important, and physics is distinctive in having simple models defined in terms of simple mathematical functions.

Symbolic regression: fit a dataset, not with a fixed linear model or a neural network, but with models from a large class of symbolic expressions. This task has been implemented using genetic algorithms and other techniques, and Schmidt and Lipson turned their work into the commercial system Eureqa.

Learning and heuristic algorithms

Over the years a number of systems have been developed to learn physical laws from data (Langley 1979, Falkenheimer and Michalski 1986, Schmidt and Lipson 2009, ...).

In doing physics, the difficulties of deciding what to measure, measuring it, choosing a subset of observations to try to model, *etc.* are far greater than those of fitting a model to data. Still, fitting models to data is important, and physics is distinctive in having simple models defined in terms of simple mathematical functions.

Symbolic regression: fit a dataset, not with a fixed linear model or a neural network, but with models from a large class of symbolic expressions. This task has been implemented using genetic algorithms and other techniques, and Schmidt and Lipson turned their work into the commercial system Eureqa.

Cranmer *et al*, "Discovering Symbolic Models from Deep Learning with Inductive Biases," `arxiv:2006.11287` treats the following problem:

- Input: Dynamical data for a system of interacting particles – their positions and velocities as functions of times: $\vec{x}_i(t_1)$, $\vec{v}_i(t_1)$, $\vec{x}_i(t_2)$, $\vec{v}_i(t_2)$, *etc.*
- Model class: mechanical systems with pairwise interactions. These could be described by Newton's laws,

$$m_i \frac{d^2 \vec{x}_i}{dt^2} = F_i = \sum_{j \neq i} F(\vec{x}_i, \vec{x}_j)$$

  where $F(\vec{x}, \vec{y})$ is an unknown force law. Or, by Hamiltonian dynamics or other reformulations.
- Output: fit $F(\vec{x}, \vec{y})$ and use symbolic regression to get an interpretable expression.

$$m_i \frac{d^2 \vec{x}_i}{dt^2} = \vec{F}_i = \sum_{j \neq i} F(\vec{x}_i, \vec{x}_j)$$

Approach: basically, directly fit the data with a neural model closely related to Newton's equations:

- Construct a graph whose nodes are particles and whose edges connect particles for whom the interaction is non-negligible.
- Model the interaction using a MLP on each edge to fit F.
- Model the dynamics by summing the F's and using a second MLP on each node.
- Finally, interpret the MLP fits using symbolic regression.

Two main results:

- In trial problems (e.g. simulate inverse square law), show that the method works without any prior knowledge about F. For example, F could take values in a high dimensional space, but the fit produces an F whose values are three dimensional vectors.
- Treat a real problem from cosmology: given a dataset generated by a dark matter simulation far more complicated than a simple interacting particle system, find an approximate model using pairwise interactions and a symbolic force law.

From Cranmer *et al*, arxiv:2006.11287

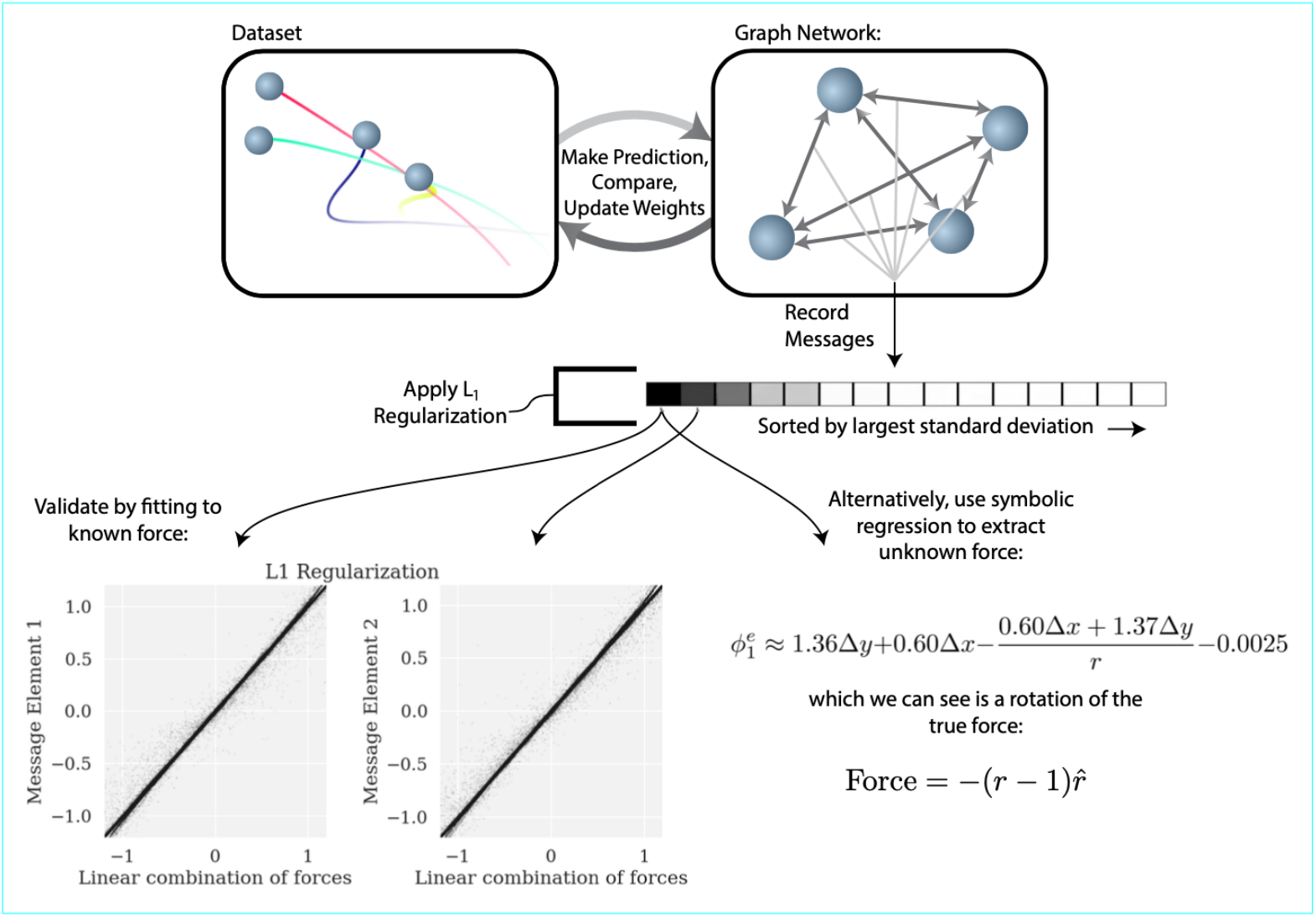

Figure 3: A diagram showing how we implement and exploit our inductive bias on GNs. A video of this figure during training can be seen by going to the URL `https://github.com/MilesCranmer/symbolic_deep_learning/blob/master/video_link.txt`.

Examples from string theory

String theory provides models of our universe, in which the fundamental laws of physics (the Standard Model and its hypothetical extensions) can be derived by assuming that there are six or seven extra dimensions of space which form a small manifold M, and doing a lot of physical analysis based on the geometry of M. I gave a short introduction to this at AITP 2018, and there are many reviews, for example Anderson and Karkheiran `arxiv:1804.08792` (but they all assume some math/physics background).

Because we don't know what manifold to use for M, we want to do some analysis of every candidate, at least enough to rule it out or get some simple predictions. There are millions of known candidates, so the number and scale of the mathematical problems we need to solve is vast.

The best understood class of M are the Calabi-Yau (CY) manifolds, six (or eight) dimensional Ricci flat Kähler manifolds. These have been intensively studied and we have combinatorial algorithms to answer many geometric questions. But these are often too inefficient to run on all the examples, so we want to extrapolate the results. In addition they can be intricate to program. On the other hand for this sort of large scale scan, we do not need 100% accuracy. Thus these problems are good candidates for heuristic ML algorithms.

I will briefly describe two of them:

- Compute Hodge numbers of complete intersection CY's, following Y-H He `arXiv:1706.02714` and many subsequent works, and Erbin and Finotello `arxiv:2007.15706` .
- Compute cohomology of line bundles, following Constantin and Lukas `arxiv:1808.09992`, Klaewer and Schechter, `arxiv.:1809.02547`.

Fit Hodge numbers of CICY's.
Input:

$$X = \left[\begin{array}{c|cccc} \mathbb{P}^{n_1} & q_1^1 & q_2^1 & \cdots & q_K^1 \\ \mathbb{P}^{n_2} & q_1^2 & q_2^2 & \cdots & q_K^2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \mathbb{P}^{n_m} & q_1^m & q_2^m & \cdots & q_K^m \end{array}\right]_{m\times K} , \qquad \begin{aligned} & q_j^r \in \mathbb{Z}_{\geq 0}; \\ & K = \sum_{r=1}^{m} n_r - 3 \ , \\ & \sum_{j=1}^{K} q_j^r = n_r + 1 \ , \ \forall\, r = 1, \ldots, m \ . \end{aligned}$$

Figure: Configuration matrix of a CICY – from He 1706.02714, eq. (3.5).

These were classified for dimension 3 (Candelas *et al* 1988, 7890 cases) and 4 (Gray *et al* 1303.1832, 921,497 cases).
Output: $h^{1,1} \in \mathbb{Z}_+$, a topological invariant (the rank of a basis of homology two-cycles). For dim 4 there is also $h^{3,1}$.
Physically these translate into the numbers of some type of particle (which could be produced at an ultra-high energy accelerator).

It is known how to compute these numbers directly (it can be done from line bundle cohomology), and complete databases were made. But the computation is somewhat intricate, so these authors looked at the ML approach. To summarize, He *et al* used MLP's, and got $\sim$ 95% accuracy. Erbin and Finotello tried a wide variety of approaches and neural architectures, and got 100% for $h^{1,1}$ of threefolds using Inception networks. Both found that feature engineering (mostly, adding well chosen monomials in the q's) makes a big difference.

A mathematically simpler and arguably more fundamental problem is to compute cohomology of line bundles. One can get much better constraints – in many (all?) cases one can show that the exact result is a piecewise polynomial function. So, a data fitting approach is far more controlled, and one can leverage ML fits into exact formulas.

It is known how to compute these numbers directly (it can be done from line bundle cohomology), and complete databases were made. But the computation is somewhat intricate, so these authors looked at the ML approach. To summarize, He *et al* used MLP's, and got $\sim$ 95% accuracy. Erbin and Finotello tried a wide variety of approaches and neural architectures, and got 100% for $h^{1,1}$ of threefolds using Inception networks. Both found that feature engineering (mostly, adding well chosen monomials in the q's) makes a big difference.

A mathematically simpler and arguably more fundamental problem is to compute cohomology of line bundles. One can get much better constraints – in many (all?) cases one can show that the exact result is a piecewise polynomial function. So, a data fitting approach is far more controlled, and one can leverage ML fits into exact formulas.

Comparable attempts have been made to compute knot invariants (Hughes `arxiv:1610.05744`). Also there is a huge field of ML-inspired numerical methods – in general this is a different topic, but there is overlap.

General remarks:

- Naive ML fits don't work well.
- By adding some domain knowledge – the class of function being fit, and feature engineering – one can get useful approximations without too much effort.
- Could one make a system which takes a problem specification (types of input and output, domain knowledge about the relation) and produces an appropriate ML model ?
- Use of probabilistic/approximate methods and results in mathematics $\Rightarrow$ mathematical reasoning becomes more like other types of reasoning in AI.

Formal/Symbolic/Numerical computational methods

ITP languages have advanced to where any mathematical structure can be formalized. This includes definitions in analysis such as "smooth manifold" and "ordinary differential equation." One could state and formally prove claims such as,

- Data structure DS can represent any manifold M and any vector field v on M to a specified precision ϵ, requiring storage $f(M, \epsilon)$.
- Algorithm A can compute the flow map for time t associated to a vector field v (the map $M \to M$ obtained by integrating the ODE $\dot{x} = v(x)$) with precision ϵ, using computer time $g(\epsilon)$.
- Algorithm S can find a equilibrium (stationary) distribution ρ on M which is preserved by the stochastic ODE $\dot{x} = v(x) + \xi$, again to specified precision, and giving information on fluctuations and (non-)uniqueness.

An example of the second, verified computation of a flow map, was done by Fabian Immler in his 2017 thesis, as part of his verification of Warwick Tucker's proof that the Lorenz system is a strange attractor. (See Ordinary Differential Equations in the Isabelle AFP).

Immler's thesis points the way towards a systematic method to prove existence theorems in ODE and perhaps PDE – break the problem down into parts which can be solved numerically, solve each part using rigorous interval arithmetic, and combine the parts into a rigorous proof. Perhaps this could be another goal for AITP.

Staying with scientific computation, these are specifications for some of the workhorse problems, covering a nontrivial fraction of computational biology and parts of computational physics, economics, *etc.*. One could extend the list to cover many more topics.

An example of the second, verified computation of a flow map, was done by Fabian Immler in his 2017 thesis, as part of his verification of Warwick Tucker's proof that the Lorenz system is a strange attractor. (See Ordinary Differential Equations in the Isabelle AFP).

Immler's thesis points the way towards a systematic method to prove existence theorems in ODE and perhaps PDE – break the problem down into parts which can be solved numerically, solve each part using rigorous interval arithmetic, and combine the parts into a rigorous proof. Perhaps this could be another goal for AITP.

Staying with scientific computation, these are specifications for some of the workhorse problems, covering a nontrivial fraction of computational biology and parts of computational physics, economics, *etc.*. One could extend the list to cover many more topics.

An example of the second, verified computation of a flow map, was done by Fabian Immler in his 2017 thesis, as part of his verification of Warwick Tucker's proof that the Lorenz system is a strange attractor. (See Ordinary Differential Equations in the Isabelle AFP).

Immler's thesis points the way towards a systematic method to prove existence theorems in ODE and perhaps PDE – break the problem down into parts which can be solved numerically, solve each part using rigorous interval arithmetic, and combine the parts into a rigorous proof. Perhaps this could be another goal for AITP.

Staying with scientific computation, these are specifications for some of the workhorse problems, covering a nontrivial fraction of computational biology and parts of computational physics, economics, *etc.*. One could extend the list to cover many more topics.

Applications to scientific computation

What could we do with this capability?

Most work on ITP focuses on using it for proofs of correctness – and this is certainly a valuable capability in all software development, including scientific computation.

One can imagine axiomatically defining the more mathematical parts of any of the sciences, and verifying scientific computations and claims. This might have caught some difficult to find mistakes.

An example from particle physics which I discussed at length at Big Proof 2 (ICMS, May 2019) is the computation of $g-2$ for the muon, which required combining theoretical calculations in the Standard Model with experimental data. From 1996-2001, the accepted result was in error, due to a mistake in conventions. See

http://www.icms.org.uk/downloads/bigproof/Douglas.pdf

Applications to scientific computation

What could we do with this capability?

Most work on ITP focuses on using it for proofs of correctness – and this is certainly a valuable capability in all software development, including scientific computation.

One can imagine axiomatically defining the more mathematical parts of any of the sciences, and verifying scientific computations and claims. This might have caught some difficult to find mistakes.

An example from particle physics which I discussed at length at Big Proof 2 (ICMS, May 2019) is the computation of $g-2$ for the muon, which required combining theoretical calculations in the Standard Model with experimental data. From 1996-2001, the accepted result was in error, due to a mistake in conventions. See

`http://www.icms.org.uk/downloads/bigproof/Douglas.pdf`

Most exploratory research does not reach the level of interest that would justify proving it correct. On the other hand one has to combine a lot of parts and get a lot of details right to get any sort of sensible results. The physics problems I described earlier – fit the force laws in a cosmological simulation, compute the properties of Calabi-Yau manifolds in order to derive their physical predictions – are very intricate and require painstaking attention to detail. Standard practice in academia is to give this part of the job to graduate students and postdocs. But could we automate more of it?
Thus, let me ask,

- How can we search the literature for relevant results and software, and
- How can we correctly integrate this into our projects?

Mathematical search engines

Familiar search engines such as Google and Bing are based on indexing by keywords and metadata. One can imagine many other types of search. Even very unusual search capabilities have been successfully implemented and commercialized, for example music recognition by matching spectrograms (Shazam).

Two domains where existing search technologies are limiting are code search and mathematical formula search. Representative systems for code search are Google Code Search and SourceGraph, while active research projects include Deep Code Search (ICSE 2018) and CodeBERT `arXiv:2002.08155`.
Mathematical formula search engines include `https://www.searchonmath.com` (Gonzaga *et al*) and MathWebSearch (Kohlhase *et al*), `http://mathweb.org/projects/mws/pubs/mkm08.pdf` which is used by zbMATH mathematical reviews.

In my opinion, code search and mathematical search are similar and the main difficulties are similar.

One problem is that there are many different ways to say the same thing. People rarely remember the details of a piece of code, or a particular syntactic formulation of a mathematical theorem. Rather, they remember the meaning, whatever that is. A good search engine would be "semantic," finding all of the responses relevant to the meaning of the query. See the CodeSearchNet challenge (Husain *et al*, `arXiv:1909.09436`) for a discussion of semantic code search, and a benchmark dataset and leaderboard.

A second (related) problem is that almost all mathematics and software refers to a vast body of previous work – the standard mathematics and CS we learn in university, broadly accepted research results, software libraries, *etc.*. To refer to it in a query, we need to know precisely how the computer refers to it.

In my opinion, code search and mathematical search are similar and the main difficulties are similar.

One problem is that there are many different ways to say the same thing. People rarely remember the details of a piece of code, or a particular syntactic formulation of a mathematical theorem. Rather, they remember the meaning, whatever that is. A good search engine would be "semantic," finding all of the responses relevant to the meaning of the query. See the CodeSearchNet challenge (Husain *et al*, `arXiv:1909.09436`) for a discussion of semantic code search, and a benchmark dataset and leaderboard.

A second (related) problem is that almost all mathematics and software refers to a vast body of previous work – the standard mathematics and CS we learn in university, broadly accepted research results, software libraries, *etc.*. To refer to it in a query, we need to know precisely how the computer refers to it.

Let us give a simple illustration from group theory. Suppose we know the concept of a normal subgroup, and we want to ask, "Is there a known concept of a group without normal subgroups?" Our search engine should be able to answer queries like

- `Define:`
  $\text{normal}(H, G) := \forall H \in \text{subgroups}(G), \forall g \in G, gHg^{-1} = H$
- `FIND:`
  $\forall H \in \text{subgroups}(G) : \text{normal}(H, G) \Rightarrow H \cong \text{trivial} \vee H \cong G$

While the definition of "normal" is fairly short, it is a well known concept, so we would prefer to refer to the standard definition. But is it denoted normal(G, H), or normal(H, G), or $H \triangleleft G$, or ?

People tend to remember mathematical and CS concepts, the functionality of library routines, *etc.*. But they have a hard time remembering all of their exact names, precise notations and calling conventions, *etc.*. A good search engine would somehow help the user through these difficulties.

Let us give a simple illustration from group theory. Suppose we know the concept of a normal subgroup, and we want to ask, "Is there a known concept of a group without normal subgroups?" Our search engine should be able to answer queries like

- `Define:`
  $\text{normal}(H, G) := \forall H \in \text{subgroups}(G), \forall g \in G, gHg^{-1} = H$
- `FIND:`
  $\forall H \in \text{subgroups}(G) : \text{normal}(H, G) \Rightarrow H \cong \text{trivial} \vee H \cong G$

While the definition of "normal" is fairly short, it is a well known concept, so we would prefer to refer to the standard definition. But is it denoted normal(G, H), or normal(H, G), or $H \triangleleft G$, or ?

People tend to remember mathematical and CS concepts, the functionality of library routines, *etc.*. But they have a hard time remembering all of their exact names, precise notations and calling conventions, *etc.*. A good search engine would somehow help the user through these difficulties.

This problem is far greater for software. The Python Package Index registers 244,695 projects at this writing. For any task you need to solve, unless it is really very novel, there is a good chance that existing software can solve a significant part of it.

But, how can we find the software relevant for our tasks? How can we even describe our tasks? A long standing problem with software verification is that specifications are generally no easier to write than programs.

However, mathematical and scientific software is easier to specify. The whole point of mathematics is to develop concepts which can be stated precisely and concisely, and which generalize beyond the context in which they were invented (or discovered).

The examples we gave earlier in the talk – representing a vector field on a manifold, or integrating the corresponding ODE to get the flow – are certainly examples.

This problem is far greater for software. The Python Package Index registers 244,695 projects at this writing. For any task you need to solve, unless it is really very novel, there is a good chance that existing software can solve a significant part of it.

But, how can we find the software relevant for our tasks? How can we even describe our tasks? A long standing problem with software verification is that specifications are generally no easier to write than programs.

However, mathematical and scientific software is easier to specify. The whole point of mathematics is to develop concepts which can be stated precisely and concisely, and which generalize beyond the context in which they were invented (or discovered).

The examples we gave earlier in the talk – representing a vector field on a manifold, or integrating the corresponding ODE to get the flow – are certainly examples.

One might object that lumping together all instances of integrating an ODE is too limiting from a computational point of view. Standard computer algebra systems such as Mathematica and Matlab have functions such as `NSolve` for numerically solving ODE's. In this sense the problem is already solved for the cases they can handle – say 10's of variables and 100's of predetermined initial conditions.

Of course, many applications in engineering, computational biology, economic modeling, *etc.*, involve far larger systems, require far more integrations and far more analysis of the results. Such applications often use parallel and high performance computers, and special algorithms.

As examples of special algorithms with simple mathematical definitions, one might do a symmetry reduction, or one might integrate out (approximate the behavior of) fast variables to get an effective ODE describing slow variables. These are standard techniques in theoretical physics and throughout the mathematical sciences.

At present one performs such analyses by hand and then hands the resulting ODE to the computer. But they are well defined and systematic, and one could develop packages to do them.
So, let us add to the previous list of standard problems,

- Given a manifold M with the action of a finite group G which acts equivariantly on a vector field v, data structure DS1 represents v (to specified accuracy *etc.*) with no redundancy.
- Similarly DS2 handles Euclidean groups, DS3 other Lie groups.
- DS4 is like DS2 but can handle 1000's of variables efficiently (and is only effective for large problems). And so on...
- Algorithm A computes the flow map for vector field v represented using DS. Resp. A1 uses DS1 for quotients, *etc.*
- A3 is like A but is implemented in interval arithmetic and provides rigorous bounds on solutions.
- A4 implements a fast-slow separation of variables technique.

And so on. All of these subcases and variations on the problem could be specified, and a good search engine would look through the available solutions for the ones best matching our specification.

Now, suppose we search for a package, say to integrate an ODE with certain symmetries, and we find it. The next steps are to check that it actually does what we want, that we can run it on our computer, to learn the API and integrate it with our software, to test it and so on. All of this takes time, and this often makes software reuse impractical.

Still, suppose we do this, and are able to use a previously developed package P to solve our problem. Suppose the result is really useful and we publish our new package. Then, suppose package P changes its API (or worse some unspecified aspect of its behavior). Of course we specified the version of P we used, so our package doesn't break immediately, but clearly there will be a future need to deal with this.

These are two examples of the "formal friction" which software users and developers deal with every day. As long as computers have no ability to reason about their software, every piece has to fit precisely, and humans have to deal with all of the mismatches. Conversely, any ability we can give the computer to reason about software, should be applied to smooth away these problems.

Call by specification

The long term solution to many of these problems is to get away from syntactic API's, which assume particular names, calling conventions and functionality. Instead, the calling program should completely specify the functionality it expects a library routine to supply, and the computer should match this up with the specification of the library routine. Then, if the library routine changes in some way, either its specification will change and the mismatch will be detected (and ideally automatically fixed), or else the library will fail to verify its own specification (perhaps noticed by failing tests).

This would be very laborious and probably impractical for most software. But if the specifications are already written and standardized, it might be possible. Say, for an ODE solver, the specification is taken from a textbook on numerical methods with a formal specification of ODEs and approximate solutions. The library writer might use a different textbook, but that is OK if we can match up the definitions.

An example from cosmology

There is a three-dimensional compact positively curved space, different from S^3 but a quotient of S^3, which has been proposed to describe our universe. We would like to simulate cosmology on this space and look at what happens if the dark matter is bound systems of black holes. (Sounds hard – and it is – but we want to test our ideas.)

Suppose I didn't know the name of the space, if I had a mathematical search engine I could type

```
#find (Q : manifold) (G : group) := Q ≅ S^3/G∧
    finite π_1(M)
```

and learn that it is the Poincaré dodecahedral space. Furthermore I would find various realizations of Q suitable for numerical work.

Matter in general relativity is described by its equation of motion and stress-energy tensor. In many (not all) cases one can model it as a large collection of N particles with pairwise interactions.

An example from cosmology

There is a three-dimensional compact positively curved space, different from S^3 but a quotient of S^3, which has been proposed to describe our universe. We would like to simulate cosmology on this space and look at what happens if the dark matter is bound systems of black holes. (Sounds hard – and it is – but we want to test our ideas.)

Suppose I didn't know the name of the space, if I had a mathematical search engine I could type

$$\texttt{\#find } (Q \texttt{ : manifold}) \ (G \texttt{ : group}) \ := \ Q \cong S^3/G \wedge \texttt{ finite } \pi_1(M)$$

and learn that it is the Poincaré dodecahedral space. Furthermore I would find various realizations of Q suitable for numerical work.

Matter in general relativity is described by its equation of motion and stress-energy tensor. In many (not all) cases one can model it as a large collection of N particles with pairwise interactions.

An example from cosmology

There is a three-dimensional compact positively curved space, different from S^3 but a quotient of S^3, which has been proposed to describe our universe. We would like to simulate cosmology on this space and look at what happens if the dark matter is bound systems of black holes. (Sounds hard – and it is – but we want to test our ideas.)

Suppose I didn't know the name of the space, if I had a mathematical search engine I could type

$$\texttt{\#find } (Q \texttt{ : manifold}) \ (G \texttt{ : group}) \ := \ Q \cong S^3/G \wedge \texttt{finite } \pi_1(M)$$

and learn that it is the Poincaré dodecahedral space. Furthermore I would find various realizations of Q suitable for numerical work.

Matter in general relativity is described by its equation of motion and stress-energy tensor. In many (not all) cases one can model it as a large collection of N particles with pairwise interactions.

To simulate cosmology, one needs to model the equations of general relativity coupled to matter (regular and dark). Writing or even specifying such a simulator is nontrivial and very likely I will instead use one from a textbook or research paper on the subject. So, how easy would it be to change it to use Q as three-dimensional space, and to incorporate the dark matter model?

Let us assume a particle model for the matter – this is an ODE and we can change it on that level. So, we look for a simulator which makes the ODE description explicit.

Now, the problem we are discussing is related to standard cosmology by two modifications:

- Space Q is a quotient by the 120 element binary icosahedral group $\mathcal{I}$.
- Dark matter consists of particle systems with fast dynamics.

These are the two cases handled by algorithms A1 and A4 of our previous discussion. So, one would hope that the textbook simulator could be easily modified to use them.

So how would this look in terms of code? Let's imagine the textbook simulator can work with space M as any one of the three standard choices ($\mathbb{R}^3$, S^3 or H^3), and defines a configuration space $\mathcal{M} \equiv M^N \times \mathcal{G}$ where $\mathcal{G}$ are the metric variables (scale factor etc.). It then defines a "numerical manifold" $\mathcal{DM}$ which can represent geometric objects on $\mathcal{M}$. It then constructs a vector field v on $\mathcal{M}$ describing GR, particle dynamics in curved space and pairwise interactions (all standard stuff). It then defines a space-time history using the integrator,

```
def space_time (DM : nmanifold) (v : vec DM) :=
  integrator DM v init precision_and_other_specs
```

The integrator is defined by its specification, namely that it defines a function $\mathbb{R} \to \mathcal{DM}$ satisfying the ODE with given initial conditions. The AS4 integrator satisfies the same spec, if we tell it that the internal dark matter degrees of freedom are "fast."

As for the quotient, the idea is that it can be done by defining another numerical manifold $\mathcal{DM}'$ with the same specifications as $\mathcal{DM}$ but describing functions and geometric objects on $Q \equiv M/\mathcal{I}$. We need to find out how the simulator defines $\mathcal{DM}$ in terms of M, say

```
def make_DM (M : manifold) (N : nat) :
    nmanifold := (ndata FRWmetric M) \times vec
    (npoint M) N
```

and modify this to define $\mathcal{DM}'$.

$\mathcal{DM}'$ is not literally $\mathcal{DM}/\mathcal{I}$ – for example the metric degrees of freedom don't work this way. What we do know is how everything described by $\mathcal{DM}$ is related to the geometry of M, as a point, a vector field, a metric tensor *etc.*. The action of $\mathcal{I}$ is entirely determined if we know this (up to questions about how we treat singularities).

So what we need to do is specify this quotient action – but this came from our original search where we found out about M. We then specify

```
def nquotient_axiom1 (𝒟𝓜 : nmanifold) (M :
    manifold) (G : group) :=
    (quotient (make_𝒟𝓜 M) G) ≅
    make_𝒟𝓜 (quotient M) G
```

and whatever other conditions we believe the data for particles moving on a quotient manifold should satisfy.

This specification will require nontrivial work to get right. But hopefully, after defining one or two such axioms, the computer will be able to find that this concept or something similar has already been defined. We can then copy it and say, we are looking for an implementation of that. Perhaps the definition of quotient comes with an implementation, perhaps there is a better implementation out there somewhere else. The point is to be able to work with the entire body of prior art in a way similar to what is presently done in developing a proof using an ITP.

Summary and conclusions

Machine learning is being tried and adopted by practitioners in many different areas, and mathematical scientists are very active at this. In this talk I focused on questions and developments which might be interesting for the AITP community – because they make use of ITP and program verification, because they use ML to obtain symbolic results, and because they are probably too difficult to do without advances in ML and ITP automation. These included

- Systems which learn intricate mathematical relations
- Systems which discover symbolic physical models
- Observations on ITP as a tool and medium for mathematical scientists, and
- A proposal for scientific programming by specification and refinement, replacing software library search and manual integration with call by specification.

I look forward to discussion, continued progress, and more and better tools for doing science.