Discovering global Lyapunov functions

What does it take to solve a hard math problem?

François CHARTON, Meta AI

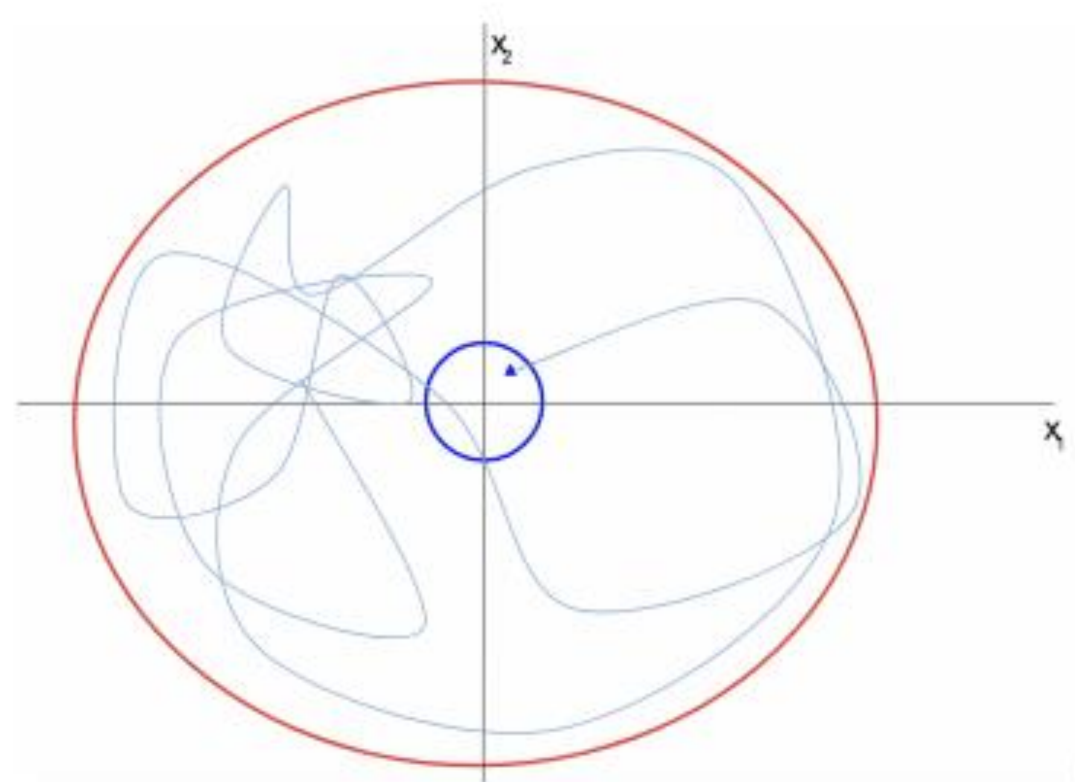
Local stability of dynamical systems

- A dynamical system is a system of n equations in n variables, such that $\dot{x} = f(x), \quad x \in \mathbb{R}^n, \quad f: \mathbb{R}^n \to \mathbb{R}^n$
- At x_0 such that $f(x_0) = 0$, the system is locally stable if all the (complex) eigenvalues of the Jacobian $\left(\frac{\partial f_i}{\partial x_j}\right)_{(i,j)\in\mathbb{N}_n^2}$ have negative real parts
- Transformers can learn this (Charton, Hayat, Lample, ICLR 2021)

Global stability of dynamical systems

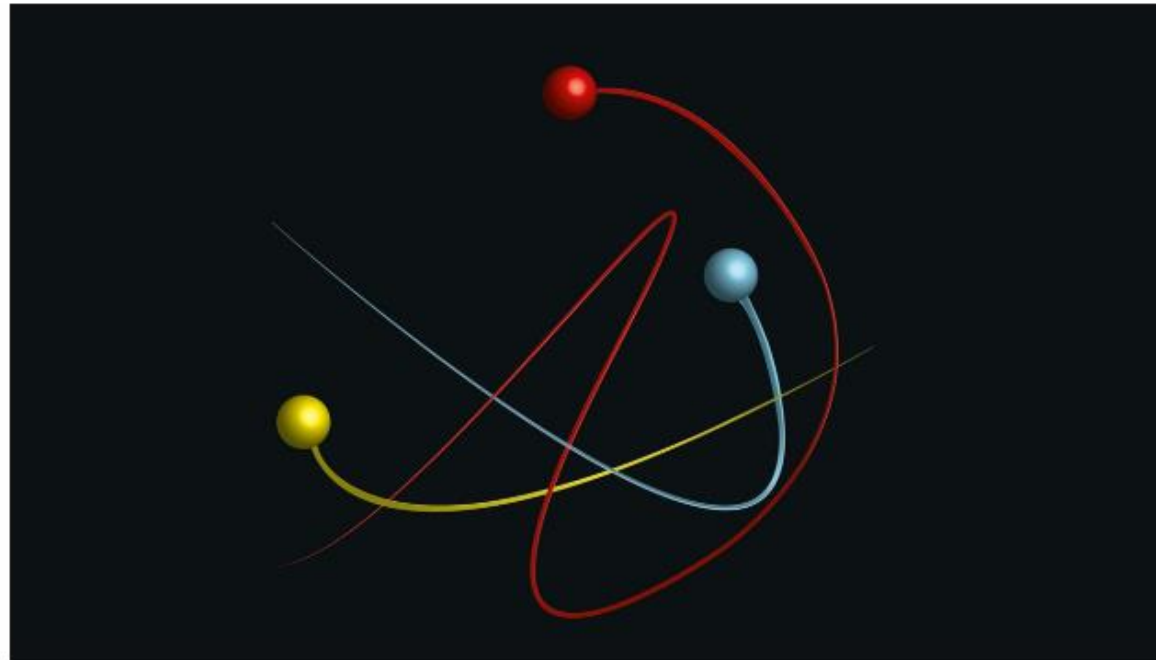
- Behavior around a stable equilibrium in $x_0 \doteq 0$
- If $x(t) < \delta$ for $t = 0$, do we have $x(t) < B$ for any $t > 0$?

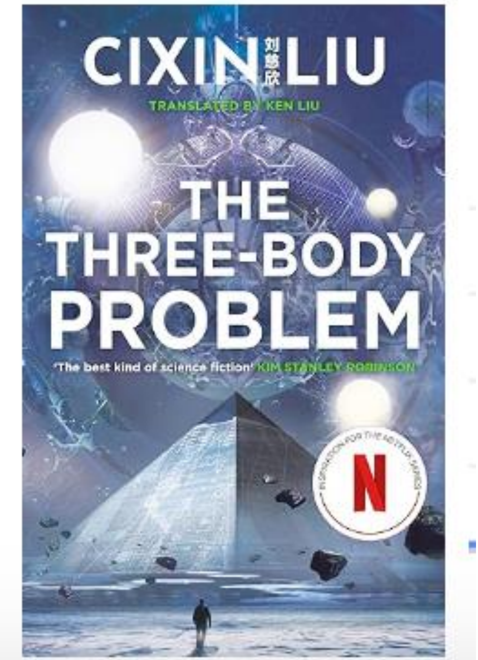
- If I start in the blue ball will I stay in the red ball?



Stability of dynamical systems

- A longstanding problem, that interested mathematicians for centuries, and Netflix for months





Lyapunov functions

- Lyapunov (1892): if there exists $V \in C^1(\mathbb{R}^n, \mathbb{R})$, and for all $x \in \mathbb{R}^n$,

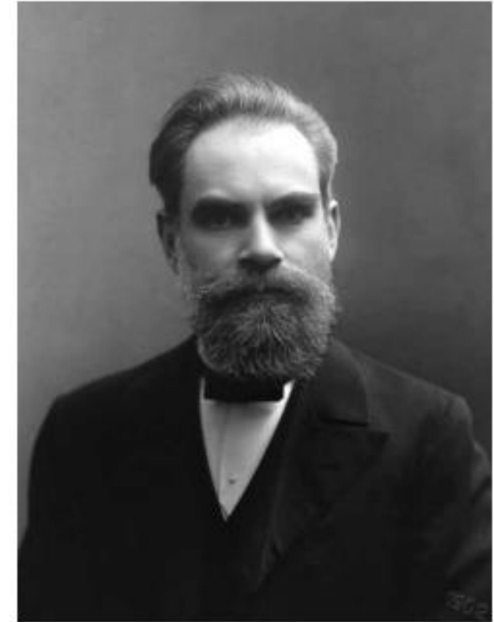
$$V(x) > V(0)$$

$$\lim_{|x| \to +\infty} V(x) = +\infty$$

$$\nabla V(x).\, f(x) \leq 0 \ ,$$

Then the system is globally stable

Start close to the origin => stay close to the origin



A. Lyapunov
(1857-1918)

Barrier Lyapunov functions

The strict minimum in condition $V(x) > V(0)$ can be relaxed.

If $V(x) \geq V(0)$

(keeping $\lim_{|x| \to +\infty} V(x) = +\infty$ and $\nabla V(x). f(x) \leq 0$),

V is a **barrier Lyapunov function**: it defines a subspace that a solution cannot leave

Start within the barrier => Stay within the barrier

Lyapunov functions

$$\begin{cases} \dot{x}_0 & = 2x_1^2 \\ \dot{x}_1 & = -10x_1 \end{cases}$$

Model inference: Our model recovers $V(x) = 10x_0^2 + 2x_0x_1^2 + 3x_1^4 + 6x_1^2$.

Clearly $V(0) = 0$ and $V(x) = 9(x_0)^2 + (x_0 + x_1^2)^2 + 2(x_1^2)^2 + 6x_1^2 > 0$ for all $x \neq 0$. Also $\nabla V \cdot f = -x_1^2(116x_1^2 + 120) \leq 0$.

Lyapunov functions

System	Lyapunov function
$\begin{cases} \dot{x_0} = -5x_0^3 - 2x_0x_1^2 \\ \dot{x_1} = -9x_0^4 + 3x_0^3x_1 - 4x_1^3 \end{cases}$	$V(x) = 6x_0^6 + 7x_0^4 + x_0^3 + 10x_0^2 + 8x_1^2$
$\begin{cases} \dot{x_0} = -x_0^5 - 4x_0^3 - 9x_0x_1^4 + 3x_0x_1^3 \\ \dot{x_1} = -3x_0^4x_1^2 - 10x_0^3x_1 + 3x_0x_1^2 - 7x_1^3 \end{cases}$	$V(x) = x_0^4 + 9x_0^2 + 3x_1^2$
$\begin{cases} \dot{x_0} = -3x_0^3 + 3x_0x_2 - 9x_0 \\ \dot{x_1} = -x_0^3 - 5x_1 + 5x_2^2 \\ \dot{x_2} = -9x_2^3 \end{cases}$	$V(x) = x_0^4 + 7x_0^2x_2^2 + 3x_0^2 + 4x_0x_2^2 + 3x_1^2 + 2x_2^4 + 10x_2^2$
$\begin{cases} \dot{x_0} = -8x_0x_1^2 - 10x_1^4 \\ \dot{x_1} = -8x_1^3 + 3x_1^2 - 8x_1 \\ \dot{x_2} = -x_2 \end{cases}$	$V(x) = 4x_0^2 - 2x_0x_1^2 + 6x_1^4 + 4x_1^2 + x_2^2$

Table 14: **Some additional examples generated from our models**.

Lyapunov functions

- No systematic way of finding V from f
- Or even to prove that V exists for a given f

Lyapunov functions: the state of the art

- No systematic method for finding Lyapunov functions
- Numerical methods exist for special cases
  - n small, f a polynomial of low degree, V a sum of squares (SOS) of monomials
  - SOSTOOLS: computationally intensive
  - BUT: there are stable polynomial systems with no SOS Lyapunov function (Ahmadi et al. 2011)
- Neural networks for a weaker problem: robustness with respect to small perturbations
  - FOSSIL, Neural Lyapunov,
  - Provide a neural approximation of V for small perturbations
  - Useful in robotics

Global Lyapunov functions

- The global case matters every time we want control over extreme situations:
  - Epidemiology
  - Flood control
  - Transportation

Global Lyapunov functions as an AI problem

(Alfarano, Charton, Hayat, 2024)

- Train a language model to predict V from f
- By presenting it with a lot of examples of f and V
- So that it learns to "translate" f into V
- And use it to suggest possible V for a given f

- We can do symbolic integration this way
  - Compute symbolic integrals as well as Mathematica (Lample Charton ICLR 2020)
- Why not Lyapunov functions?

AI for Lyapunov functions

1. Generate a lot of training examples: pairs of f and V
2. Encode f and V as sequences of “words” that a language model can process
3. Train a language model to predict V from f

Then, once the model is trained

4. For a given f, generate guesses of V
5. Test the guesses, hope for the best

AI for Lyapunov functions – the easy stuff

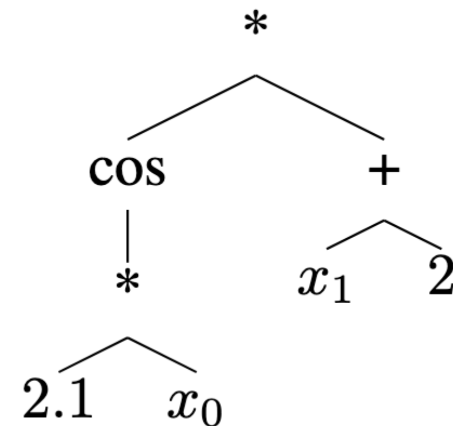
2. Encode f and V as sequences of "words" that a language model can process

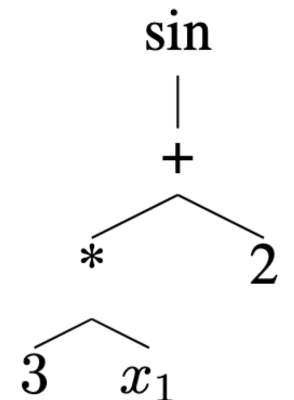
- f is a system of n symbolic functions:

$$\begin{cases} \dot{x}_0 = \cos(2.1x_0)(x_1 + 2) \\ \dot{x}_1 = \sin(3x_1 + 2) \end{cases}$$

- That can be represented as trees :
- and enumerated as sequences:

  *, cos, *, 2.1, x_0, +, x_1, 2
  sin, +, *, 3, x_1, 2





AI for Lyapunov functions – the easy stuff

3. Train a model to predict f from V

- A transformer model, with 8 layers and 640 dimensions
- Trained to predict the next word of V given f
  - From f predict v_0
  - From (f, v_0) predict v_1
  - Repeat until you predict and end-of-sequence token
- By minimizing the cross-entropy between the next token and its correct value in the training set
  - Start with random weights (model parameters), and adjust them when the next token is incorrectly predicted, to improve model prediction

AI for Lyapunov functions – the easy stuff

4. Generate guesses of V

- Models predict probabilities of the next token
- Instead of one, predict the n most likely first token v_0
- Predict the n most likely continuations v_1 You will have n^2 sequences v_0v_1, keep the n most likely
- Repeat until EOS

AI for Lyapunov functions – the easy stuff

5. Test the guesses

- Decode the sequence representing V into a function
- Use numerical methods (or SMT solvers) to prove that Lyapunov conditions hold :
  - $V(x) > V(0)$
  - $\lim_{|x| \to +} V(x) = +\infty$
  - $\nabla V(x).f(x) \leq 0$,

- This is the only moment we use mathematical software
- We need a verifier, and may have false negatives

Generating pairs of f and V

In an ideal world, we would

1. generate a random stable system f,
2. compute its Lyapunov function V with a solver,
3. add the pair (f,V) to the training set

Generate a stable system? We cannot even prove a system is stable

Compute the Lyapunov function? We only have solvers for the simplest cases

The Backward method

- Instead of finding the solutions of random problems, let us find the problems of random solutions

- Sample a function V
- Construct a system f which has V as its Lyapunov function

Lyapunov functions

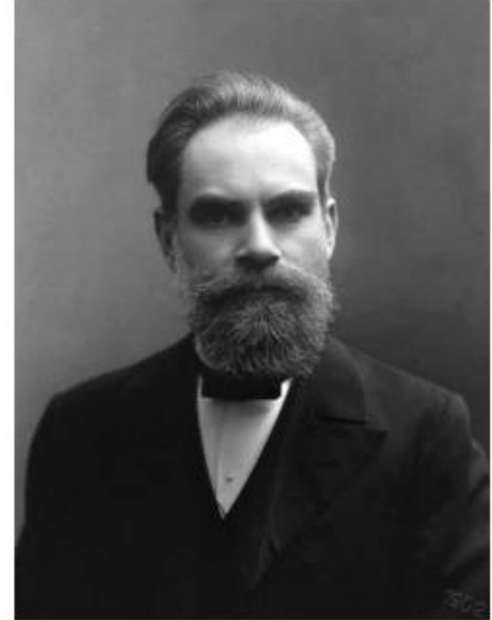
- Lyapunov (1892): if there exists $V \in C^1(\mathbb{R}^n, \mathbb{R})$, and for all $x \in \mathbb{R}^n$,

$$V(x) > V(0)$$

$$\lim_{|x| \to +} V(x) = +\infty$$

$$\nabla V(x).\, f(x) \leq 0\ ,$$

Then the system is globally stable



A. Lyapunov
(1857-1918)

The Backward method

- Select a random function V
  - A C^1 function
  - With a strict minimum in 0
  - Infinite at infinity

- Then find a random system f such that

$$\nabla V(x).f(x) \leq 0$$

Constraints on backward methods

- We want V, and f, as generic as possible
  - We should sample V from the class of continuous function with a strict minimum at 0, and infinite at infinity
  - For a given V, f should be sampled from all functions such that $\nabla V(x).f(x) \leq 0$

- We want to select V and f from the largest possible class, with as little bias as possible

Constraints on backward methods

- We want the model to learn to solve the Lyapunov problem not to reverse the generating procedure

- Suppose we compute (exactly) the real roots of polynomial with integer coefficients
- Backward generation is tempting: you can easily generate a polynomial from its roots
  - e.g. $P(x)=(x-2)(x-5)(x-7)$, for roots 2,5,and 7

- If the model is presented with the factorized form of polynomial P during training, it will learn to "read" the roots in the polynomial, instead of solving it
- If the model is provided with the developed form $P(x) = x^3-14x^2+59x-70$, we are solving the hard problem.

Constraints on backward methods

- We need to prevent the generation from “leaking” information about the solution into the problem
- Naive methods will not work
- We need to obsfuscate our generation method

- We cannot guarantee the absence of such leaks (Yehudah 2020)

Generating V

- V infinite at infinity, with a strict minimum in zero
  - No systematic way to sample functions with a strict minimum
- We rewrite $V = V_{proper} + V_{cross}$,
  - V_{proper} has a strict minimum in zero and is infinite at infinity,
  - V_{cross} is non-negative and bounded

Generating V

- Specifically, for V_{proper}, we
  - sample a positive polynomial P(x) = $\sum_{i,j=1}^{n} a_{i,j} x_i^{b_i} x_j^{b_j}$ with $a_{i,j}$ the entries of a random positive definite matrix,
  - apply a generic increasing function I (sampled in a large class)
  - multiply by random positive functions
- V_{cross} is a sum of squares of bounded functions

$$V(x) = \left[I\left(\sum_{i=1}^{n} \alpha_{i,j} x_i^{\beta_i} x_j^{\beta_j} \right) - f(0) \right] g\left(\sum_{i=1}^{q} \alpha_{\sigma(i),\sigma(j)} x_{\sigma(i)}^{\beta_{\sigma(i)}} x_{\sigma(j)}^{\beta_{\sigma(j)}} \right) + \sum_{i=1}^{m} b_k(\xi_k + p_k(x)),$$

Generating f from V

We want $\nabla V(x).f(x) \leq 0,$
$f(x) = -\nabla V(x)$ is an obvious solution

But a very bad one: finding V from f, now amounts to integrating f, we are solving a different (and easier) problem!

We can modify this solutions by multiplying each coordinate by a random positive function

$$f(x) = -(h_i^2(x)(\nabla V(x))_i),$$

we still verify $\nabla V(x).f(x) \leq 0$, and integration of f no longer allows to recover V

Generating f from V

$f(x) = -h^2(x)\nabla V(x)$ verifies $\nabla V(x). f(x) \leq 0$, but it is not generic enough

We can improve it by adding a random vector $w(x)$, satisfying

$$w(x). \nabla V(x) = 0$$

a vector w from the normal hyperplane $\mathcal{H}_x = \{w | w. \nabla V = 0\}$, that can be generated as

$$w(x) = \sum_{i=0}^{n-1} g_i(x) e_i(x),$$

with g_i random functions and e_i base vectors of $\mathcal{H}_x$

$$f(x) = \left(h_i^2(x). (-\nabla V(x))_i\right) + w(x)$$

Generating f from V

$f(x) = -h^2(x)\nabla V(x)$ verifies $\nabla V(x).f(x) \leq 0$, but it is not generic enough

A better candidate is $f(x) = -h^2(x)\nabla V(x) + w(x)$, with $w(x).\nabla V(x) = 0$

i.e. adding a vector w from the normal hyperplane $\mathcal{H}_x = \{w | w.\nabla V = 0\}$, generated as

$$w(x) = \sum_{i=0}^{n-1} g_i(x) e_i(x),$$

with g_i random functions and e_i base vectors of $\mathcal{H}_x$

Generating f from V

But do not use Gram Schmidt to calculate e, And do not make the base orthonormal!

If you do so, denominators of the form $1/||\nabla V(x)||$ will appear in the expression of $e_i(x)$, and therefore in w(x).

And it is unlikely that they will simplify away.

This leaks information about V(x).

Go for a non orthonormal base, and make the $e_i(x)$ as simple as you can.

Backward generation

Summarizing: we generate a function

$$V(x) = V_{proper}(x) + V_{cross}(x)$$

and a system

$$f(x) = \left(h_i^2(x)(-\nabla V(x))_i\right) + w(x)$$

that verifies the Lyapunov conditions

Generation depends on a number of random functions, by sampling them in different families, we can constrain V and f to belong to specific classes: e.g. polynomials

Lots of steps were taken to ensure that the generating procedure does not leak information about V in f. But we have no guarantee.

The training sets

- We generate two backward datasets:
  - BPoly: 1M backward generated polynomial systems of 2 to 5 equations
  - BNonPoly: 1M backward generated non-polynomial systems: polynomials of general functions (e.g. trigonometric polynomials)

- And use SOSTOOLS to generate two polynomial forward sets
  - FLyap: 100,000 polynomial systems that SOSTOOLS can solve
  - FBarr: 300,000 polynomial systems for which SOSTOOLS can find a barrier Lyapunov function (V(x) is no longer strictly positive).

In-domain performance

- Models perform well on the distributions they were trained on
- For forward datasets, the model achieves SOTA performance
- But for backward sets, this proves nothing… The model might be exploiting a gap in the generating procedure

Backward datasets	Accuracy bs=1	Accuracy bs=50	Forward datasets	Accuracy bs=1	Accuracy bs=50
BPoly (polynomial)	99	100	FBarr (barrier)	93	98
BNonPoly (non-poly)	77	87	FLyap (Lyapunov)	81	88

Table 2: **In-domain accuracy of models**. Beam size (bs) 1 and 50.

Out-of-domain performance

- Test backward on forward, and forward on backward
- The model cannot cheat
- Forward models do not generalize
- Backward models generalize to polynomial systems, and even to barrier systems (a slightly different problem)

Backward datasets	FLyap	FBarr	Forward datasets	BPoly
BPoly (polynomial)	**73**	35	FBarr (barrier)	15
BNonPoly (non-poly)	**75**	24	FLyap (Lyapunov)	10

Table 3: **Out-of-domain accuracy of models**. Beam size 50. Columns are the test sets.

Priming for improved performance

- Models trained on Bpoly achieve 73% accuracy on Forward generated polynomial test sets
- Part of the failures are due to the fact that the model was trained on a different distribution of systems:
  - OOD generalization is hard
- A similar problem happens when models trained on small problems are tested on large problems
  - addition of small integers, generalizing to large integers
- Priming (Jelassi 2023) was proposed as a solution to length generalization: adding a tiny number of large integers to the training set, greatly improves model performance.

Priming Lyapunov

- Adding a small number of problems with know solutions to the backward training set (Bpoly) improves performance
- Adding 0.01% new examples brings accuracy from 73 to 82%
- Adding 0.03% examples from FBarr brings accuracy on FLyap to 83%, and accuracy on FBarr to 89%.
- Few shot learning of a related problem (barrier Lyapunov functions)

Forward datasets	Examples added (1M in training set)	FLyap	FBarr
No mixture	0	73	35
FBarr	30	75	61
	300	83	89
	3,000	85	93
	30,000	89	95
FLyap	10	75	25
	100	82	29
	1,000	83	37
	10,000	86	38

Beating the state of the art

Test sets	SOSTOOL	Existing AI methods			Models			
	`findlyap`	Fossil 2	ANLC	LyzNet	PolyMixture	FBarr	FLyap	BPoly
FSOSTOOLS	-	32	30	46	**84**	80	53	54
FBarr	-	12	18	28	**93**	-	28	35
FLyap	-	42	32	66	84	**93**	-	73
BPoly	15	10	6	24	**99**	15	10	-

Table 5: **Performance comparison on different test sets**. Beam size 50. PolyMixture is BPoly + 500 FBarr.

What have we learned so far?

- Models trained on backward datasets can generalize to generic polynomial systems (that SOSTOOLS can solve)
- Priming improves performance, and allows generalization to related tasks
- We do better than existing AI methods

- But we are only solving polynomial systems, that are already known to be stable (because SOSTOOLS can solve them)

- Tout ça pour ça?

Into the wild

- Generate three test sets of random systems with a local minimum in zero, but not guaranteed to be stable
  - Poly3: polynomial systems of degree 2 and 3
  - Poly5: polynomial systems of degree 2 to 5
  - NonPoly: non polynomial systems
- We expect bad performance: most of those systems will be unstable (we don’t know how many)

Into the wild

- SOS Methods fail badly,
- some recent AI tools do better

Test set	Sample size	SOSTOOL `findlyap`	Existing AI methods Fossil 2	ANLC	LyzNet
Poly3	65,215	1.1	0.9	0.6	4.3
Poly5	60,412	0.7	0.3	0.2	2.1
NonPoly	19,746	-	1.0	0.6	3.5

Into the wild

- Backward primed models achiece 10 to 12% accuracy on random systems (even non polynomials)
- We have no way of knowing how good this is, because we do not know how many systems have Lyapunov functions
- But this is encouraging

Test set	Sample size	SOSTOOL `findlyap`	Existing AI methods Fossil 2	ANLC	LyzNet	Forward model FBarr	Backward model PolyMixture	NonPolyMixture
Poly3	65,215	1.1	0.9	0.6	4.3	11.7	**11.8**	11.2
Poly5	60,412	0.7	0.3	0.2	2.1	8.0	**10.1**	9.9
NonPoly	19,746	-	1.0	0.6	3.5	-	-	**12.7**

Table 6: **Discovering Lyapunov comparison for random systems**. Beam size 50. PolyMixture is BPoly + 500 FBarr. NonPolyMixture is BNonPoly + BPoly + 500 FBarr.

Priming in the wild

- We can bootstrap: use “wild examples” that the transformer can solve to prime the training set.
- Adding 1000 wild examples to Bpoly, and fine-tuning the model on this dataset (regenerating the test set to prevent contamination) brings performance to 13.5% (vs 11.7) on Poly3 and 11.9 (vs 9.6) on Poly5.

What about human mathematicians?

While our models exceed the algorithmic state of the art, one might wonder **how they compare to human mathematicians**. To this effect, we proposed 75 problems from the FSOSTOOLS datasets (polynomial systems with 2 or 3 equations) as a examination for 25 first year Masters students in mathematics, following a course on the subject. Each student was given 3 systems chosen at random from FSOSTOOLS and had a total of 30 min. Their performance was 9.33%, significantly lower than our models (84%).

Wondering out loud

- We managed to solve random problems in the wild
- We don’t know yet how to solve interesting problems

- Generation is the hard part, we need better, systematic techniques for backward generation
- Backward generation is Out Of Distribution by design, but what are those “distributions of problems”? We need a theory of this.
- Priming and bootstrapping definitely helps, we need to understand this phenomenon