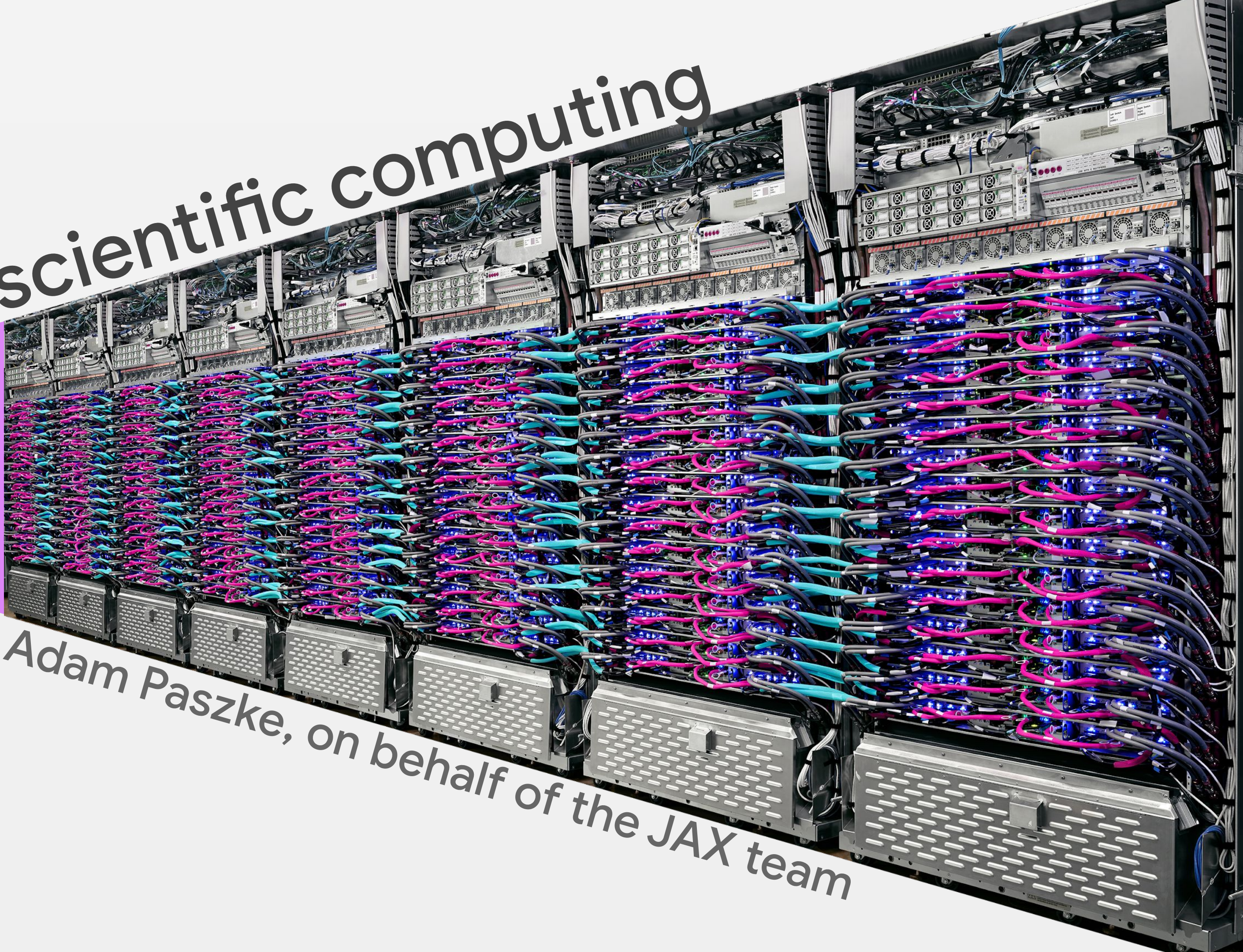
JAX for scientific computing



Adam Paszke, on behalf of the JAX team

The basics

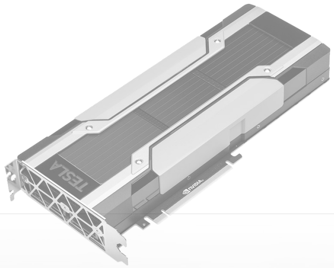
```python
import numpy as np


def predict(params, inputs):
  for W, b in params:
    outputs = np.dot(inputs, W) + b
    inputs = np.tanh(outputs)
  return outputs

def loss(params, batch):
  inputs, targets = batch
  preds = predict(params, inputs)
  return np.sum((preds - targets) ** 2)
```

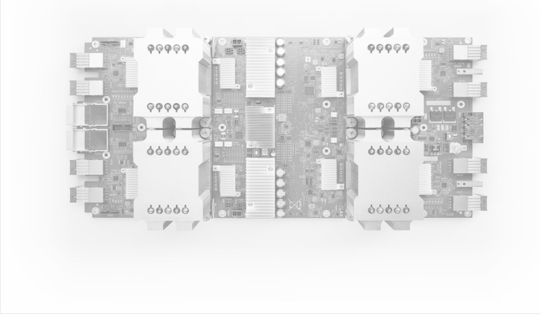
The NumPy EDSL:

- nd-array as the fundamental object
- implicit vectorization
- large library and ecosystem of scientific computing routines

The basics

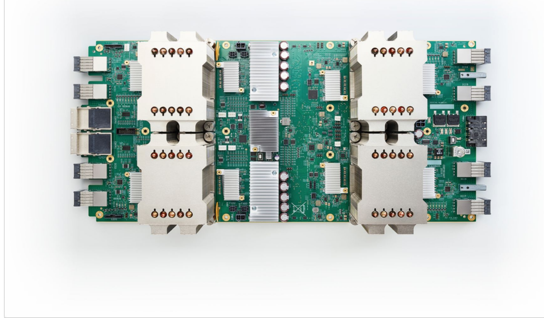
```python
import numpy as np


def predict(params, inputs):
  for W, b in params:
    outputs = np.dot(inputs, W) + b
    inputs = np.tanh(outputs)
  return outputs

def loss(params, batch):
  inputs, targets = batch
  preds = predict(params, inputs)
  return np.sum((preds - targets) ** 2)
```





The basics

```python
import jax.numpy as np


def predict(params, inputs):
  for W, b in params:
    outputs = np.dot(inputs, W) + b
    inputs = np.tanh(outputs)
  return outputs

def loss(params, batch):
  inputs, targets = batch
  preds = predict(params, inputs)
  return np.sum((preds - targets) ** 2)
```



Batching

Problem: *I have a function `simulate(initial_conditions)`, but I want to understand how the system evolves for a wide range of starting points.*

(Non-)Solution:

```
for init in initial_conditions:
  simulate(init)
```

🐢 Unvectorized execution! Poor accelerator utilization!

Solution:

```
jax.vmap(simulate)(initial_conditions)
```

🐇 Vectorized execution!

💡 Write a scalar version,
lift to array code automatically!

```
def expm_2x2(M):
  assert M.shape == (2, 2)
  [[a, b], [c, d]] = M
  ar, br, cr, dr = …  # Scalar math here
  return jnp.asarray([[ar, br], [cr, dr]])
```

Batching

Problem: *I have a function `simulate(position, momentum)`, but I want to understand how the system evolves for every pair of initial `position` and `momentum` values.*

(Non-)Solution:

```
for p in positions:
  for m in momenta:
    simulate(p, m)
```

Solution:

```
jax.vmap(jax.vmap(simulate, …), …)(positions, momenta)

# OR

jax.xmap(simulate,
         in_axes=(['position'], ['momentum']),
         out_axes=['position', 'momentum'])(positions, momenta)
```

First input provides a batch of positions (1D)

Second input provides a batch of momenta (1D)

Every combination of position and momentum yields a new output (2D)

Interlude: randomness

😵‍💫 Stateful PRNGs make reproducibility extremely difficult!

```
>>> from jax import random
>>> key = random.PRNGKey(0)
>>> key
DeviceArray([0, 0], dtype=uint32)

>>> random.uniform(key)
DeviceArray(0.41845703, dtype=float32)

>>> random.uniform(key)
DeviceArray(0.41845703, dtype=float32)

>>> key, subkey = random.split(key)
>>> random.uniform(subkey)
DeviceArray(0.10536897, dtype=float32)
```

Batching

Problem: *I have a function `simulate(prng)`, and I want to understand how the system evolves for a large number of random seeds.*

Solution:

```
prng_states = prng.split(1000)
jax.vmap(simulate)(prng_states)
```

Differentiation

Problem: *I have a function `simulate(initial_conditions)`, but I want to understand how sensitive the output is to the initial conditions.*

Solution:

```
jax.jvp(simulate)(init)
```

Problem: *I have a function `simulate(initial_conditions)`, and I want to optimize the `initial_conditions` according to some `metric`.*

```
jax.grad(lambda x: metric(simulate(x)))(init)
```

Also:

```
jax.jet, jax.jacfwd, jax.jacbwd, jax.hessian, jax.checkpoint
```

🔬 This is not numerical differentiation! It's all analytical.

Acceleration

Problem: *My simulations take way too long!*

Solution:

```
jax.jit(simulate)(init)
```

```
import jax.numpy as jnp

def log2(x):
  ln_x = jnp.log(x)
  ln_2 = jnp.log(2.0)
  return ln_x / ln_2
```

➡️

```
{ lambda  ; a.
  let b = log a
      c = log 2.0
      d = div b c
  in [d] }
```

➡️ ➡️ 10011101...

💸 This can be expensive, but there's caching!

Scaling — automatically

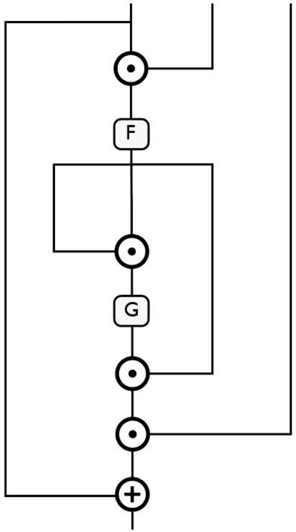
Problem: *I have lots of hardware and want to scale up/accelerate my experiments.*

Solution:

```
from jax.experimental.pjit import pjit, mesh, PartitionSpec as P

simulate(a, b)  # Runs locally, might be slow or OOM.

devices = np.array(
  [[d for d in jax.devices() if d.process_index == pidx]
   for pidx in range(jax.process_count())])

with jax.experimental.mesh(devices, ('hosts', 'local')):
  psimulate = pjit(simulate,
                   in_axis_resources=[P('local'), P('hosts')],
                   out_axis_resources=None)
  psimulate(a, b)  # Runs in parallel on all devices!
```

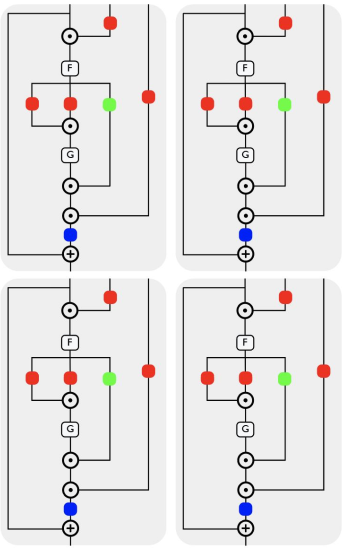
1. Set up a mesh of devices.
2. Specify how inputs and outputs are to be partitioned over the mesh.
3. Enjoy!

Scaling — automatically



Single device program

Input/output device assignment



Distributed program

Collective operations

Scaling — explicitly

Problem: *I have lots of hardware and want to scale up/accelerate my experiments.*

Solution:

```
jax.xmap(simulate,
         in_axes=(['position'], ['momentum']),
         out_axes=['position', 'momentum'])(positions, momenta)
```

⬇️

```
devices = …

with jax.experimental.mesh(devices, ('hosts', 'local')):
  psimulate = jax.xmap(simulate,
                       in_axes=(['position'], ['momentum']),
                       out_axes=['position', 'momentum'],
                       axis_resources={'position': 'hosts', 'momentum': 'local'})
  psimulate(positions, momenta)  # Runs in parallel on all devices!
```

Scaling

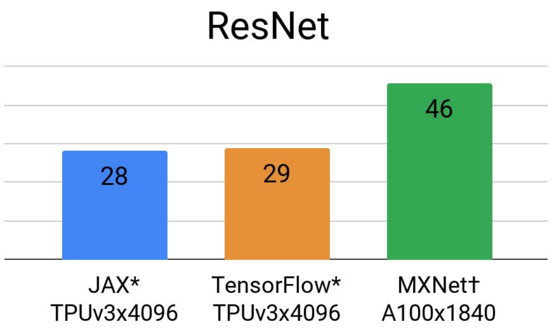
1️⃣Write code for a single device

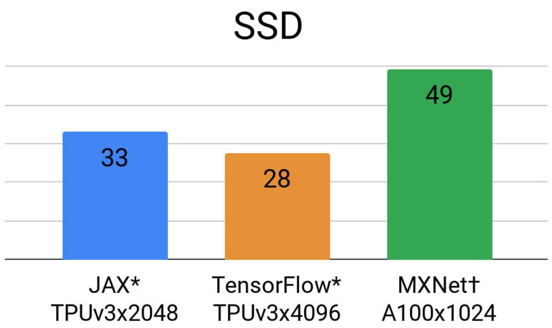
2️⃣Adapt to multiple devices (and even hosts)
without modifying the computational part

✅ Easy to transition to new hardware configurations

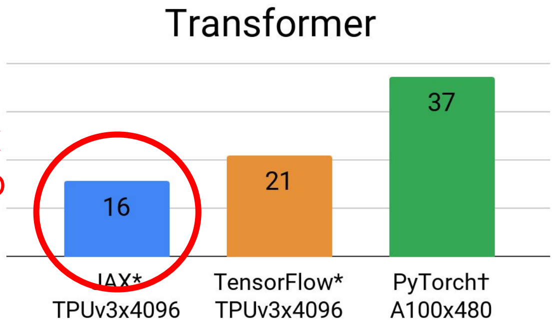
🐛 Easier debugging

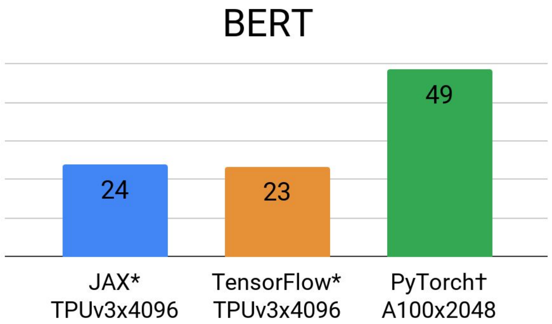
MLPerf Training v0.7 results (in seconds, lower is better)





50000x speedup over 5 years!





* Google, Research category
† NVIDIA, Available On-Premise category.

MLPerf v0.7 Training, closed division. Retrieved from www.mlperf.org 1 December 2020, entries 0.7-64, 0.7-65, 0.7-67, 0.7-30, 0.7-33, 0.7-37, 0.7-38. MLPerf name and logo are trademarks. See www.mlperf.org for more information.

Google Research

Scientific computing toolbox

🧰 Builtins

- ODE integrators
- FFTs
- Matrix factorizations
- Linear solvers
- Linear algebra routines (incl. matrix exponentials, …)
- Probability distributions
- Special functions

📚 Libraries

- Neural networks (Flax, Haiku, …)
- Optimization (optax, JAXopt, …)
- Physics (jax-md, …)
- Geometry (jaxlie, …)
- PPLs (Oryx, NumPyro, …)

Putting it all together

```
jit(vmap(grad(odeint(jet(model)))))
```

https://arxiv.org/abs/2007.04504
https://twitter.com/davidduvenaud/status/1284181673496776706

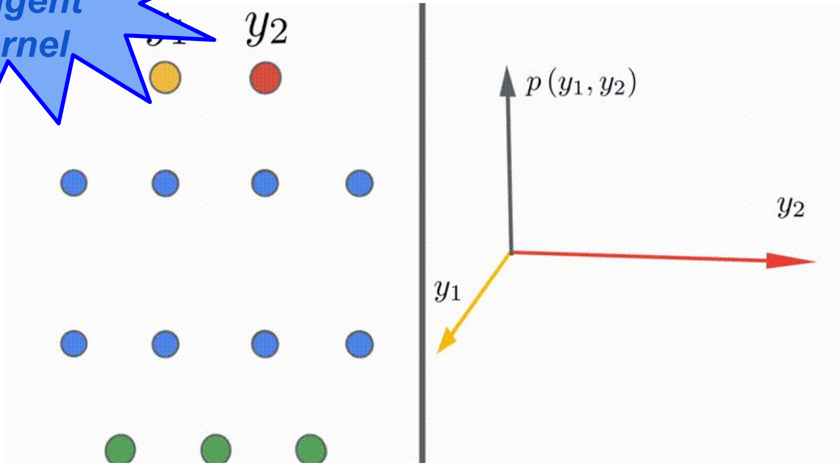
JAX is an extensible system for
composable function transformations
of Python + Numpy code.

Caveats

- Transformed functions need to be *side-effect free*
    - Modifying variables from outer scopes is not allowed (this includes globals!)
    - *Benign* side effects (print) might happen at surprising times (incl. many times)
    - Printing arrays might not display any real data
- Python control flow doesn't always work
    - ✅ `jax.vmap, jax.grad` ❌ `jax.jit, jax.pjit, jax.xmap`
    - Data-dependent branches disallowed
    - Have to use special combinators provided by JAX

```
jax.lax.cond(
    get_predicate_value(),
    lambda _: 23,
    lambda _: 42,
     operand=None)
```

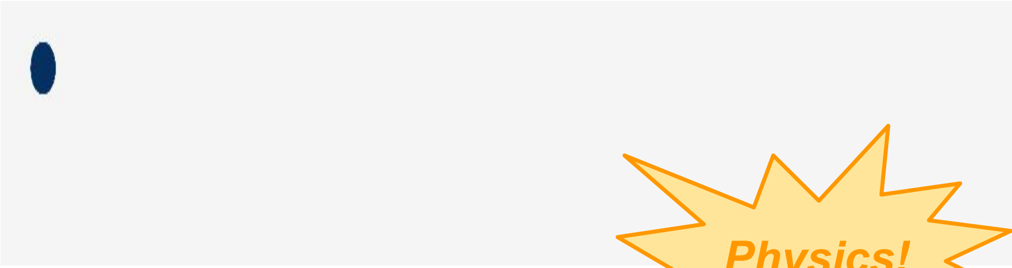
Our users



https://ai.googleblog.com/2020/03/fast-and-easy-infinitely-wide-networks.html

https://arxiv.org/abs/1907.03613

https://github.com/google/jax-md



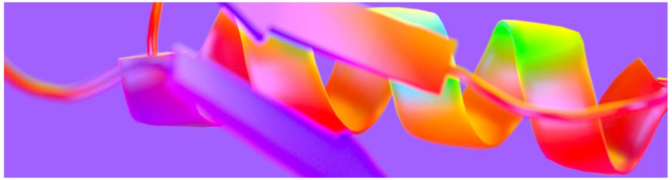
https://arxiv.org/abs/2003.04630
https://github.com/google/jax/blob/master/cloud_tpu_colabs/Wave_Equation.ipynb

deepmind/alphafold: Open sou

github.com/deepmind/alphafold

run_alphafold.py	Use pLDDT in the B-factor column of the outp...	23 days ago
run_alphafold_test.py	Use pLDDT in the B-factor column of the outp...	23 days ago
setup.py	Use tensorflow-cpu in setup.py as well.	23 days ago

README.md



AlphaFold

This package provides an implementation of the inference pipeline of AlphaFold v2.0. This is a completely new model that was entered in CASP14 and published in Nature. For simplicity, we refer to this model as AlphaFold throughout the rest of this document.

Any publication that discloses findings arising from using this source code or the model parameters should cite the AlphaFold paper. Please also refer to the Supplementary Information for a detailed description of the method.

@AlDante / **DTUFold**

Contributors 4

Augustin-Zidek Augusti...
tomwardio Tom Ward
saran-t Saran Tunyasuv...

Languages

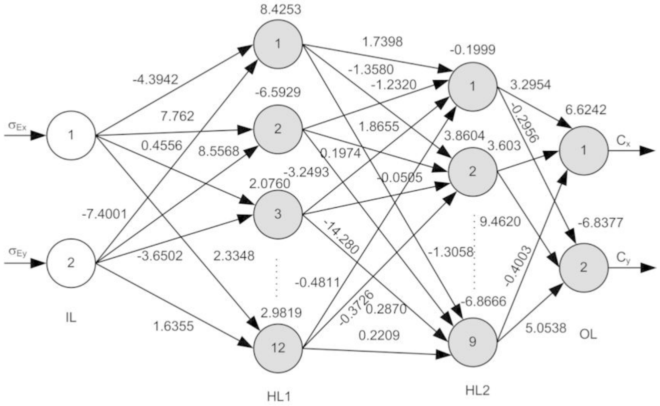
- **Python** 91.4%
- **Jupyter Notebook** 5.7%
- **Shell** 2.4%
- **Dockerfile** 0.5%

For science, differentiable programming makes it *possible* to combine the best of both worlds

Machine learning

for approximation

(soft constraints)

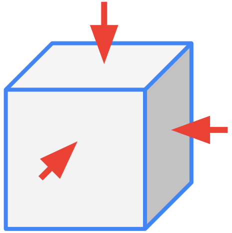


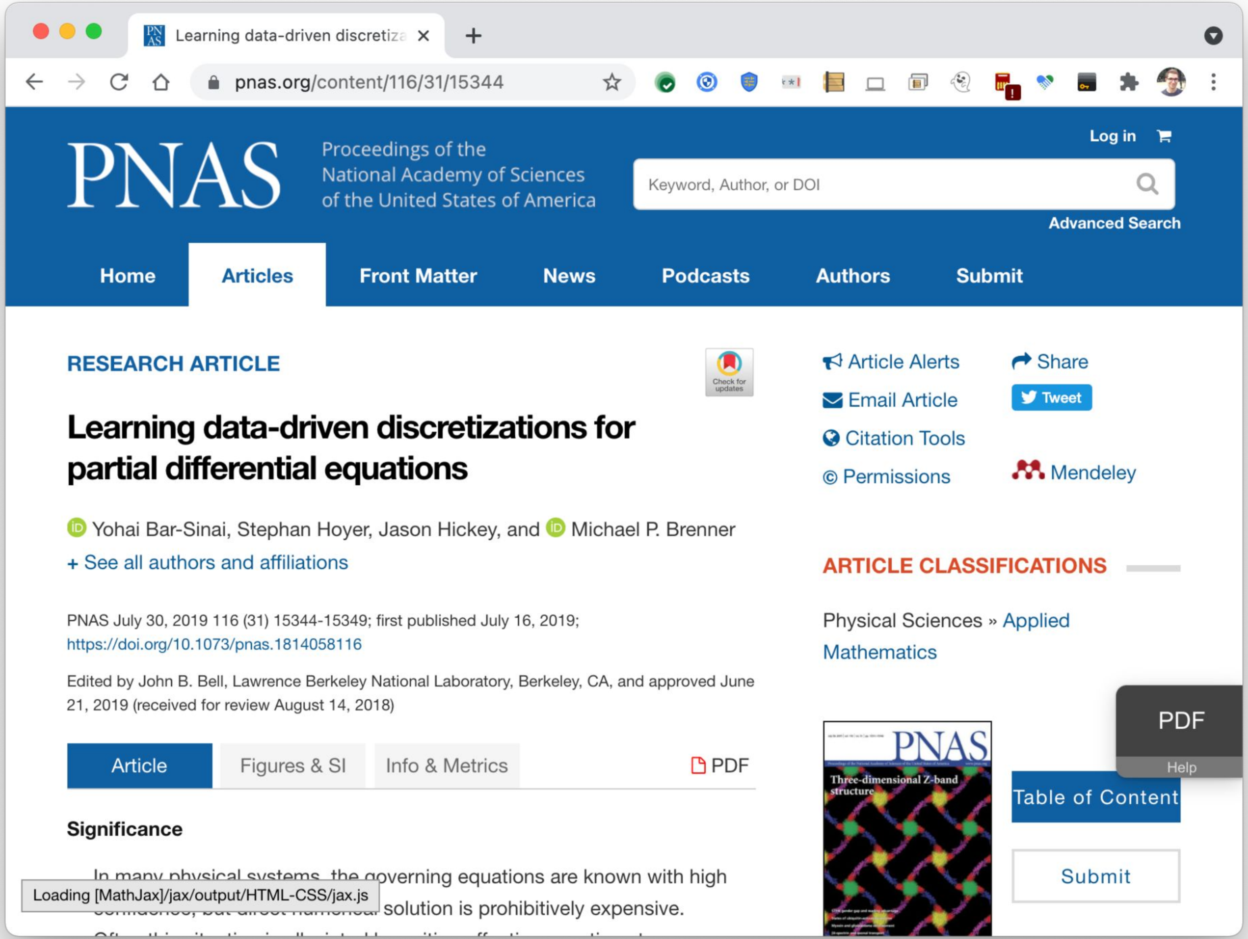
Numerical methods

for generalization

(hard constraints)

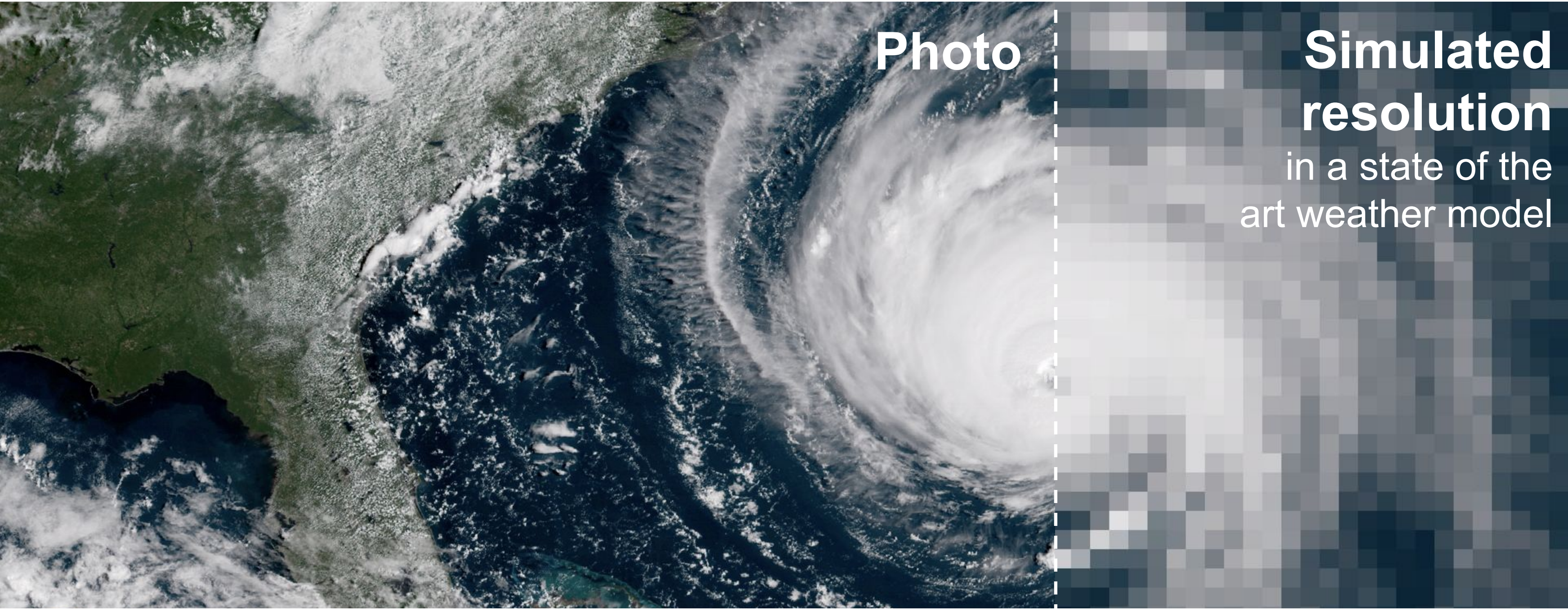
$$\frac{\partial \rho}{\partial t} + \nabla \cdot \mathbf{j} = \sigma$$




Learning data-driven discretiza
pnas.org/content/116/31/15344
PNAS
Proceedings of the National Academy of Sciences of the United States of America
Log in
Keyword, Author, or DOI
Advanced Search
Home
Articles
Front Matter
News
Podcasts
Authors
Submit
RESEARCH ARTICLE
Check for updates
Learning data-driven discretizations for partial differential equations
Yohai Bar-Sinai, Stephan Hoyer, Jason Hickey, and Michael P. Brenner
+ See all authors and affiliations
PNAS July 30, 2019 116 (31) 15344-15349; first published July 16, 2019;
https://doi.org/10.1073/pnas.1814058116
Edited by John B. Bell, Lawrence Berkeley National Laboratory, Berkeley, CA, and approved June 21, 2019 (received for review August 14, 2018)
Article
Figures & SI
Info & Metrics
PDF
Significance
In many physical systems, the governing equations are known with high
solution is prohibitively expensive.
Loading [MathJax]/jax/output/HTML-CSS/jax.js
Article Alerts
Email Article
Citation Tools
Permissions
Share
Tweet
Mendeley
ARTICLE CLASSIFICATIONS
Physical Sciences » Applied Mathematics
PNAS
Three-dimensional Z-band structure
PDF
Help
Table of Content
Submit

Google

How can we solve PDEs accurately on coarser grids?



The Challenge: Need $\Delta x \rightarrow 0$ for accuracy, but runtime is $O(1/\Delta x^{d+1})$

“Super-resolution” with machine learning

Input | Bicubic | Neural net | Original (4x resolution)

Every standard numerical method!

Ledig *et al* (Twitter), arXiv:1609.04802

Google

Taking a step back

The MATLAB* model of array programming

First-order array ops called from an interpreted host language

The good

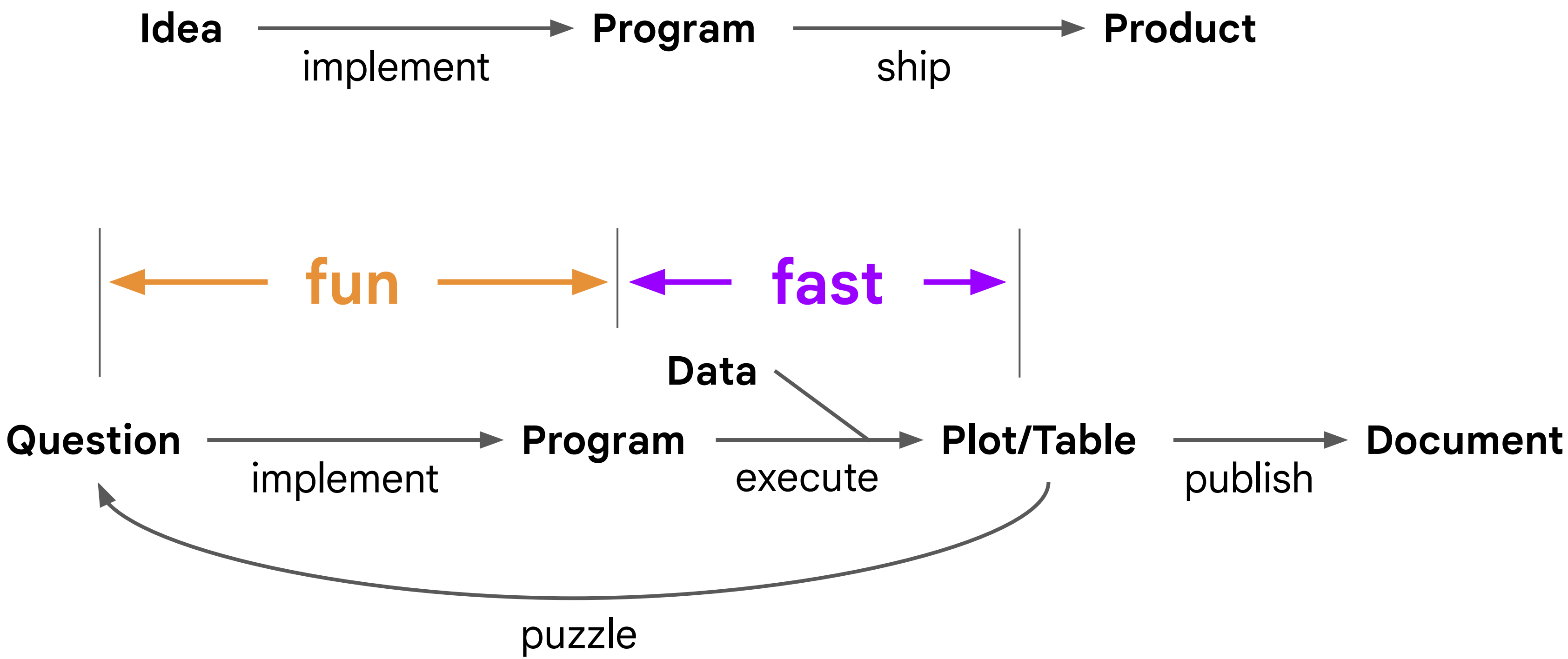
- Access to data parallelism (GPUs! TPUs!)
- Primitive set closed automatic differentiation
- Naturally embeddable (no need for a new language and compiler)

The bad

- **Expressiveness**
  - Fixed set of reductions
  - Limited data types
- **Clarity**
  - Constrains program organization (e.g. loops forced inward)
  - Shape and indexing errors

* a.k.a. APL model, MATLAB model, TensorFlow model, PyTorch model, JAX model

Scientific computing should be fun and fast



Getting to the Point.
Index Sets and Parallelism-Preserving Autodiff for Pointful Array Programming

ADAM PASZKE, Google Research, Poland
DANIEL JOHNSON, Google Research, Canada
DAVID DUVENAUD, University of Toronto, Canada
DIMITRIOS VYTINIOTIS, DeepMind, United Kingdom
ALEXEY RADUL, Google Research, USA
MATTHEW JOHNSON, Google Research, USA
JONATHAN RAGAN-KELLEY, Massachusetts Institute of Technology, USA
DOUGAL MACLAURIN, Google Research, USA

Presented at ICFP 2021

Dex by example — matrix multiplication

```
def matmul (l : n=>k=>Float) (r : k=>m=>Float) : n=>m=>Float =
  for i j. sum for u. l.i.u * r.u.j
```

No need to spell out loop bounds (but you can if you'd like)!

```
def matmul (l : n=>k=>Float) (r : k=>m=>Float) : n=>m=>Float =
  for i j. sum for u. l.u.i * r.u.j
> Type error:
> Expected: k
>   Actual: n
>
>   for i j. sum for u. l.u.i * r.u.j
>                             ^^
```

Expressive array types prevent errors and make code more accessible to readers

```
def matmul [Semiring a] (l : n=>k=>a) (r : k=>m=>a) : n=>m=>a =
  for i j. sum for u. l.i.u * r.u.j
```

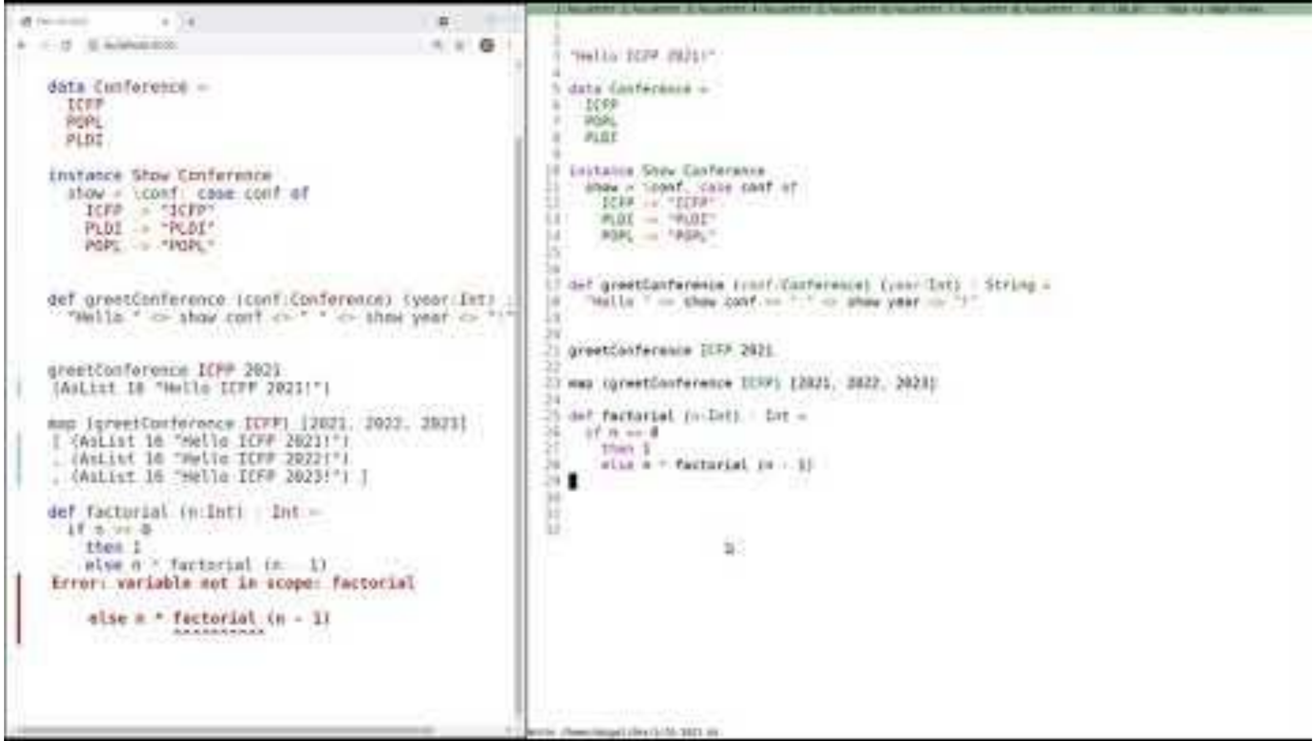
Zero-cost generics/type-classes/traits make it easy to write reusable libraries

Dex by example — Mandelbrot set

```
def update (c:Complex) (z:Complex) : Complex = c + (z * z)

def inBounds (z:Complex) : Bool = complex_abs z < 2.0

def escapeTime (c:Complex) : Int =
  fst $ yieldState (0, zero) \(n, z).
    for i:(Fin 1000).
      z := update c $ get z
      n := (get n) + (BToF $ inBounds $ get z)

xs = linspace (Fin 300) (-2.0) 1.0
ys = linspace (Fin 200) (-1.0) 1.0
mandelbrot : (Fin 200)=>(Fin 300)=>Int =
  for j i. escapeTime (MkComplex xs.i ys.j)
```

In-place modifications are allowed through effects.

Batching achieved using explicit for loops.


data Conference =
ICFP
POPL
PLDI
instance Show Conference
show = \conf. case conf of
ICFP -> "ICFP"
PLDI -> "PLDI"
POPL -> "POPL"
def greetConference (conf:Conference) (year:Int)
"Hello " <> show conf <> " " <> show year <> "!"
greetConference ICFP 2021
(AsList 16 "Hello ICFP 2021!")
map (greetConference ICFP) [2021, 2022, 2023]
[(AsList 16 "Hello ICFP 2021!")
, (AsList 16 "Hello ICFP 2022!")
, (AsList 16 "Hello ICFP 2023!")]
def factorial (n:Int) : Int =
if n == 0
then 1
else n * factorial (n - 1)
Error: variable not in scope: factorial
else n * factorial (n - 1)
"Hello ICFP 2021!"
data Conference =
ICFP
POPL
PLDI
instance Show Conference
show = \conf. case conf of
ICFP -> "ICFP"
PLDI -> "PLDI"
POPL -> "POPL"
def greetConference (conf:Conference) (year:Int) : String =
"Hello " <> show conf <> " " <> show year <> "!"
greetConference ICFP 2021
map (greetConference ICFP) [2021, 2022, 2023]
def factorial (n:Int) : Int =
if n == 0
then 1
else n * factorial (n - 1)

Scientific computing's future is **typed** and **functional**

But *we need to build it*!

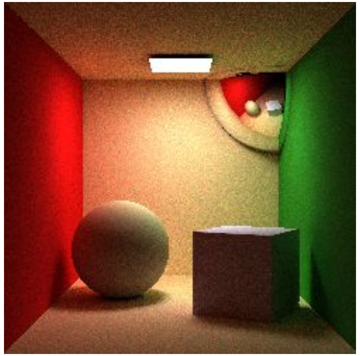
Should you consider Dex?

1️⃣ Your problem is difficult to express in array DSLs

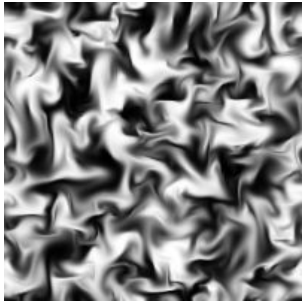
2️⃣ You are comfortable working with research software (but with support)

📞 Let us know if it sounds interesting! We're looking for a small group of pilot users.

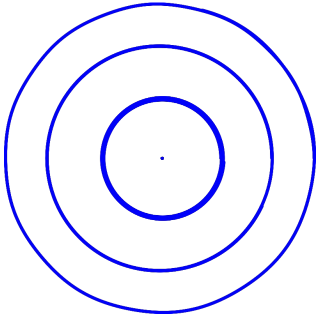
Ray tracing



Fluid simulations



n-body simulations



Recap

JAX	Dex
NumPy	Explicit loops
Acceleration	Acceleration
Differentiation	Differentiation
Batching	Batching
Scaling	🚧 Scaling
Scientific computing helpers	🆘 Scientific computing helpers 🆘
💪 Battle tested	🧑‍🔬 Research software

Thank you!

apaszke@google.com