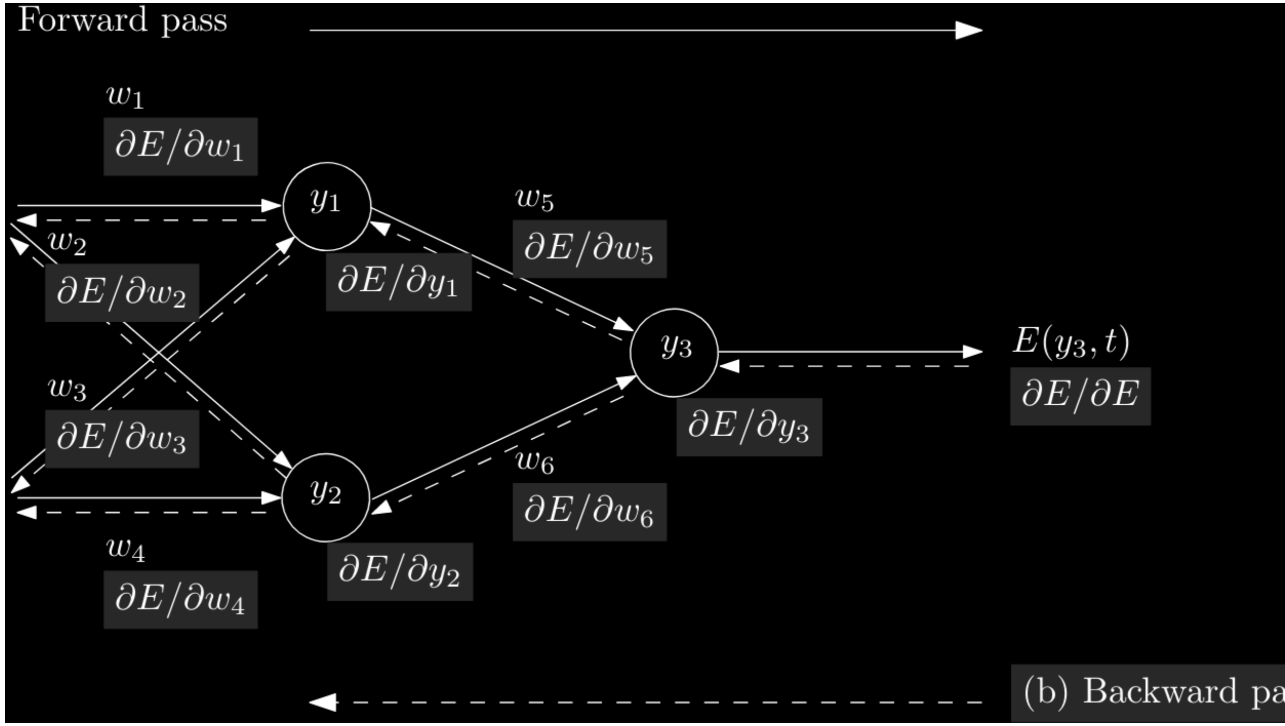
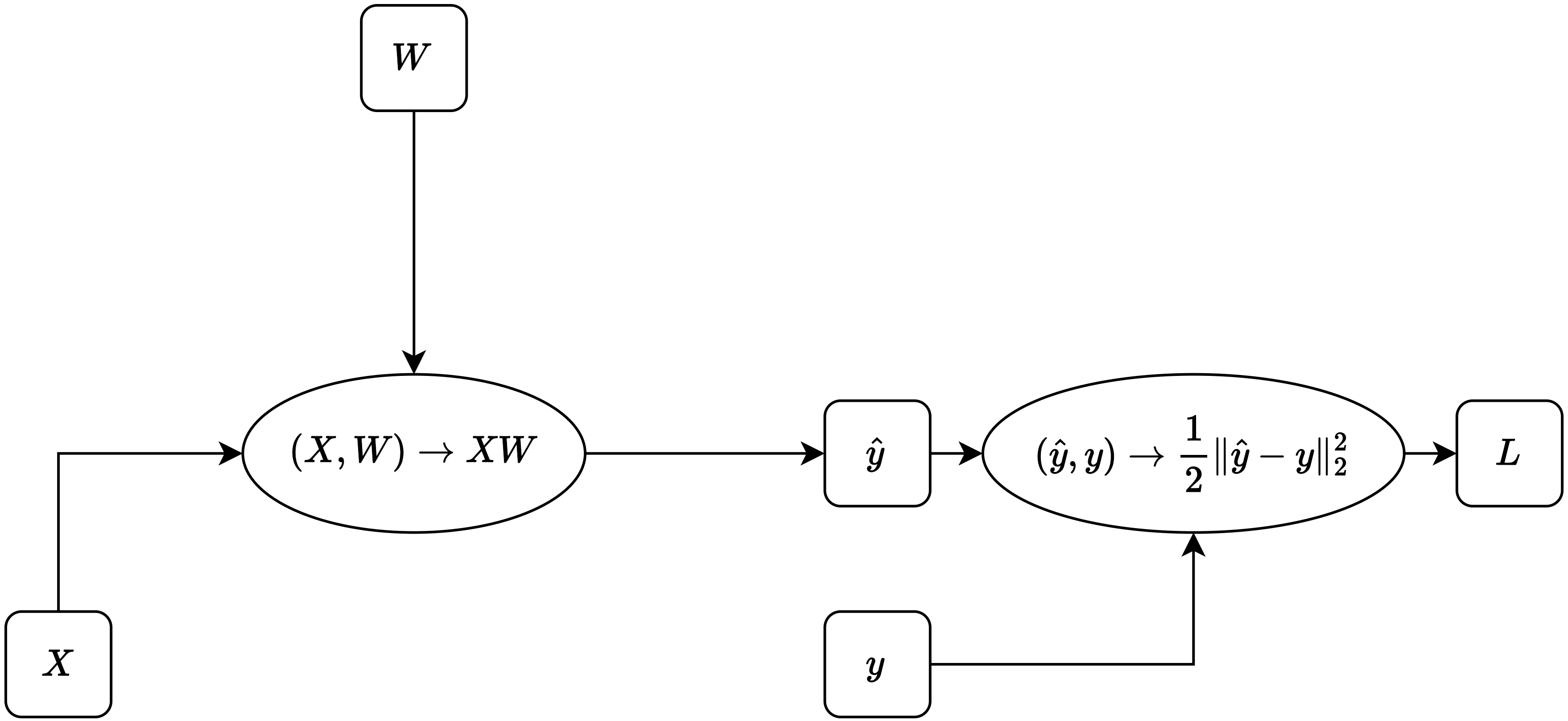
# Autodiff & Adjoints

## The machinery behind differentiable physics and deep learning

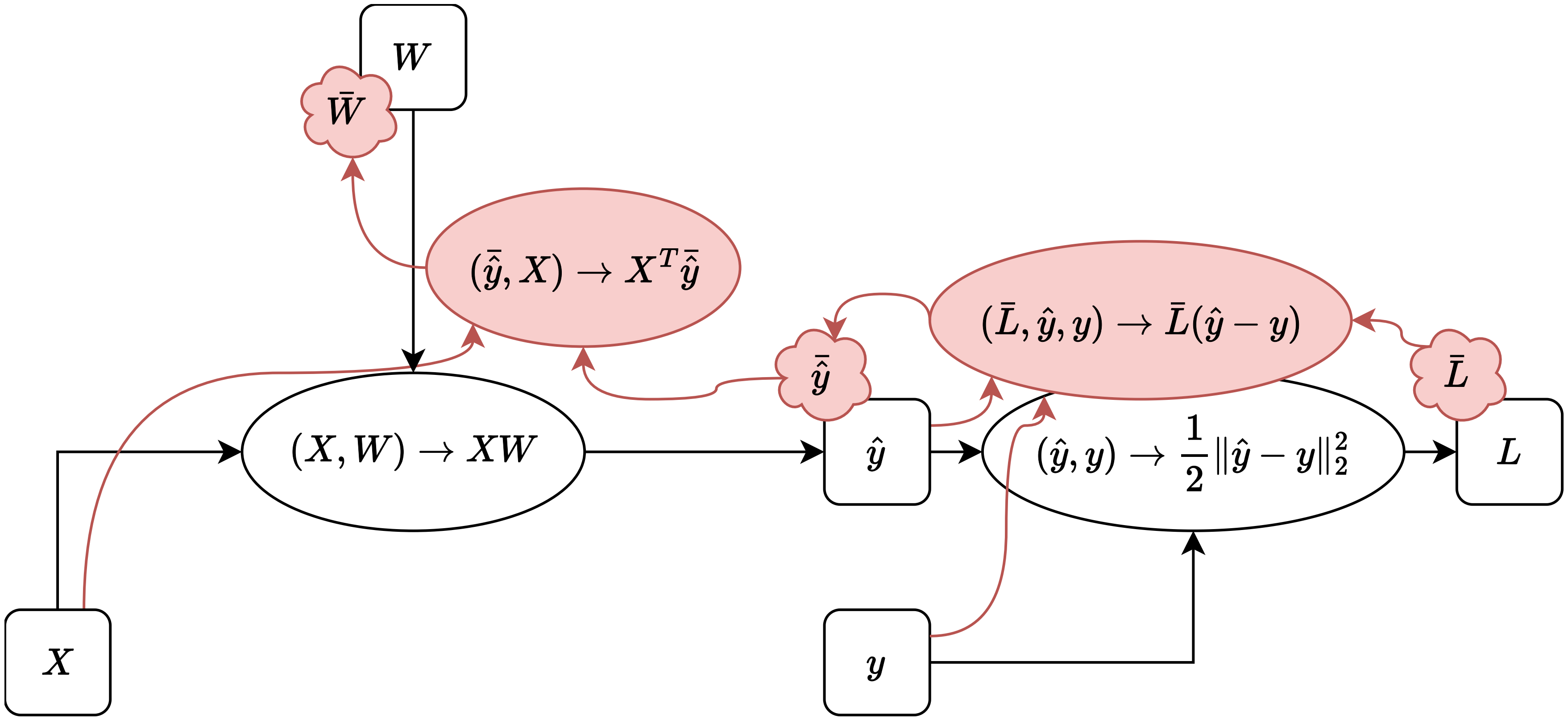
Felix Koehler



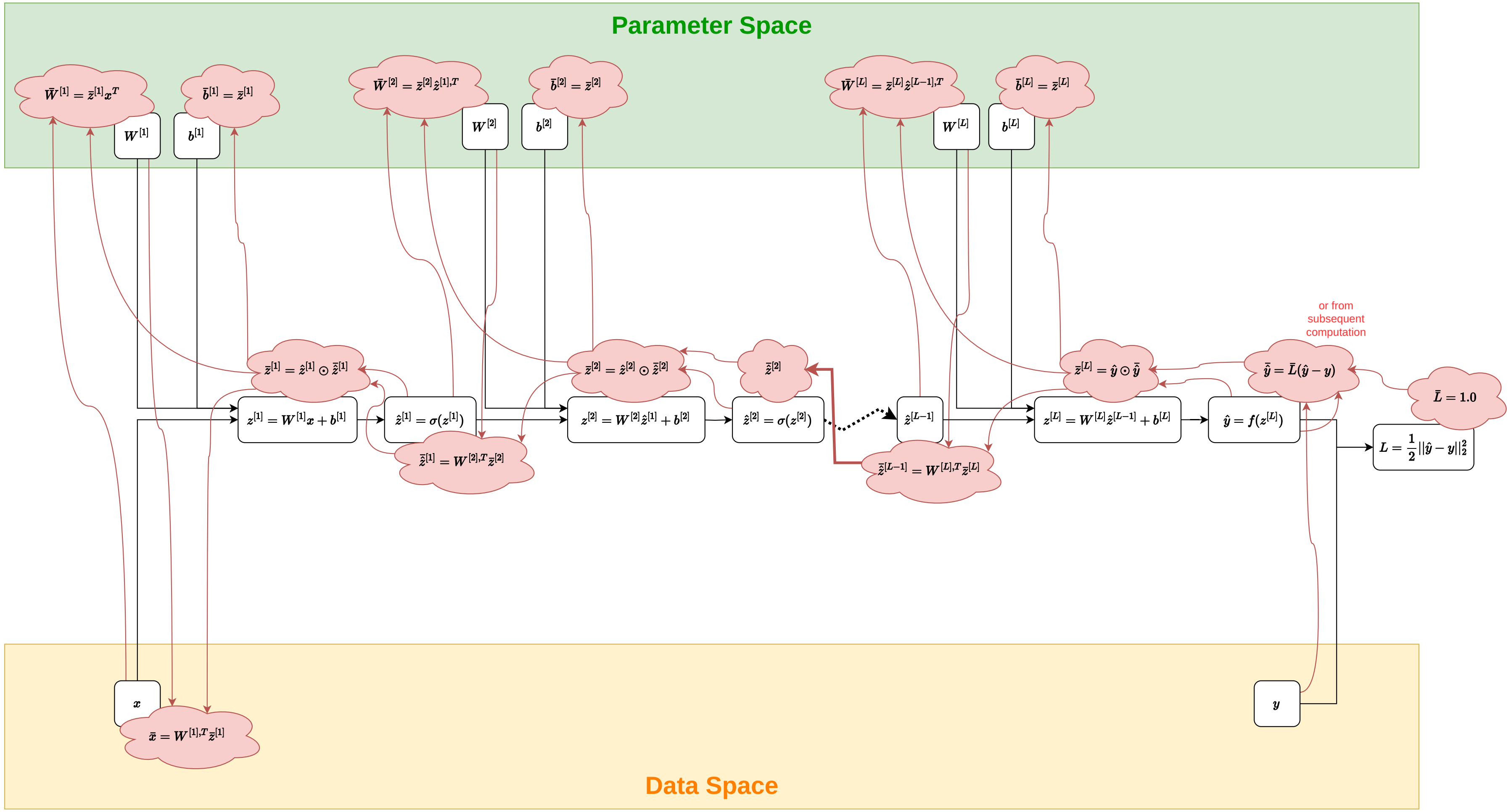
# Motivation - Linear Regression



# Linear Regression - Matrix Gradient

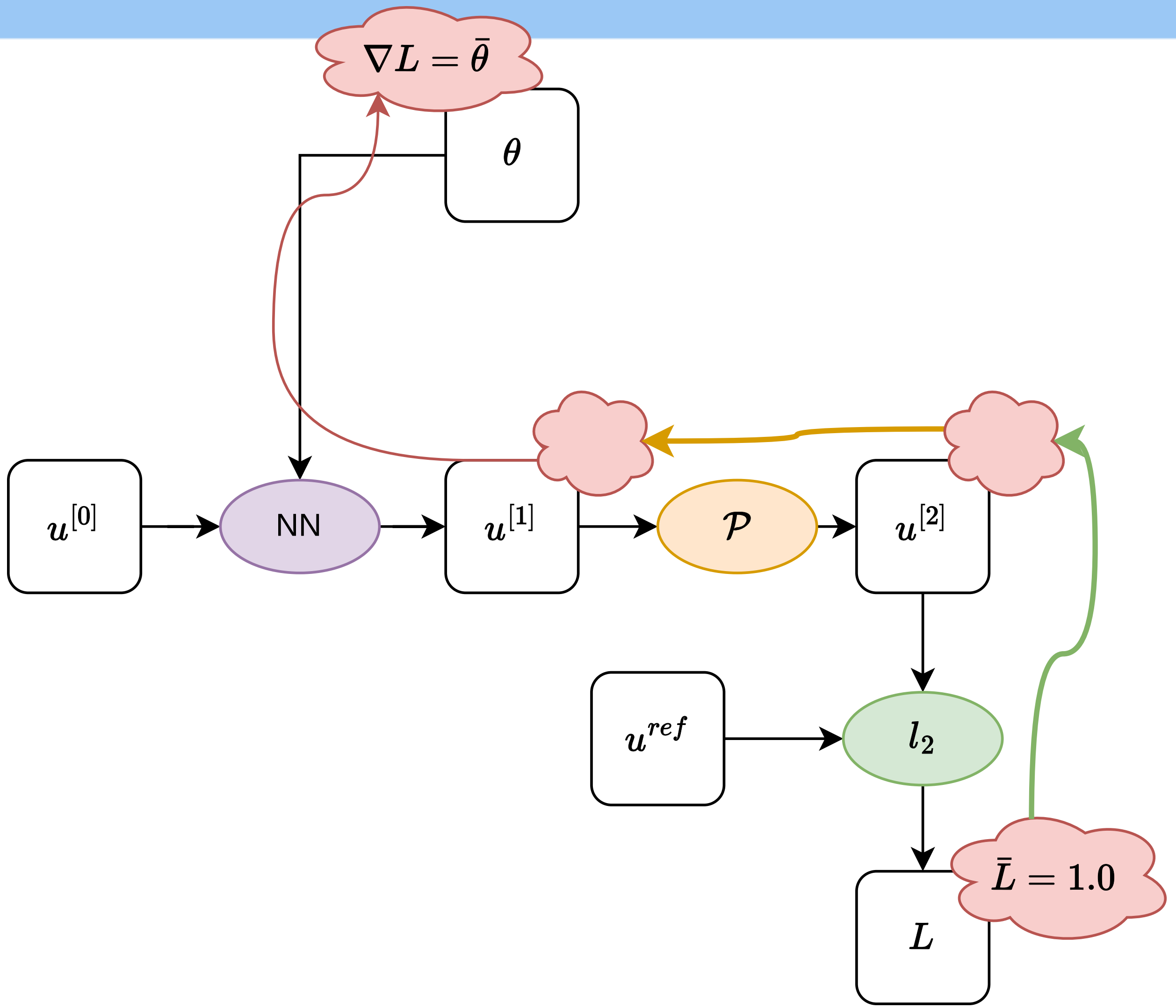


# Multi-Layer Perceptron



# Motivation

- Neural Networks are big nested compute graphs with many free parameters
- We fit these parameters using first-order optimizers
- Autodiff provides the gradients
- If physics $\mathcal{P}$ is part of the gradient flow, it has to be differentiated



# Outline

1. Autodiff from a more General Perspective
    i. A functional Viewpoint on Autodiff
    ii. Vector-mode Autodiff (BLAS-level)
    iii. Hierarchies in Autodiff
    iv. Adjoints/Continuous Sensitivities (PDE-level)
    v. History of Automatic Differentiation
2. Specialities of Differentiable Physics
3. Advanced topics

# A General Perspective on Autodiff

# Scalar Automatic Differentiation

$$y = f(x) = \sin(\exp(x^2)) = l(m(n(x)))$$

$$\begin{aligned}
z^{[0]} &= x \\
z^{[1]} &= n(z^{[0]}) = (z^{[0]})^2 \\
z^{[2]} &= m(z^{[1]}) = \exp(z^{[1]}) \\
z^{[3]} &= l(z^{[2]}) = \sin(z^{[2]}) \\
y &= z^{[3]}
\end{aligned}$$

# Two major ways of bracketing

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial z^{[3]}} \frac{\partial z^{[3]}}{\partial z^{[2]}} \frac{\partial z^{[2]}}{\partial z^{[1]}} \frac{\partial z^{[1]}}{\partial z^{[0]}} \frac{\partial z^{[0]}}{\partial x}$$

- $\frac{\partial y}{\partial x} = \underbrace{\frac{\partial y}{\partial z^{[3]}} \left( \frac{\partial z^{[3]}}{\partial z^{[2]}} \left( \frac{\partial z^{[2]}}{\partial z^{[1]}} \left( \frac{\partial z^{[1]}}{\partial z^{[0]}} \frac{\partial z^{[0]}}{\partial x} \right) \right) \right)}_{\text{forward-mode}}$
- $\frac{\partial y}{\partial x} = \underbrace{\left( \left( \left( \frac{\partial y}{\partial z^{[3]}} \frac{\partial z^{[3]}}{\partial z^{[2]}} \right) \frac{\partial z^{[2]}}{\partial z^{[1]}} \right) \frac{\partial z^{[1]}}{\partial z^{[0]}} \right) \frac{\partial z^{[0]}}{\partial x}}_{\text{reverse-mode}}$

# Pushforward = Jvp

$$\frac{\partial y}{\partial x}\dot{x} = \frac{\partial y}{\partial z^{[3]}} \left( \frac{\partial z^{[3]}}{\partial z^{[2]}} \left( \frac{\partial z^{[2]}}{\partial z^{[1]}} \left( \frac{\partial z^{[1]}}{\partial z^{[0]}} \frac{\partial z^{[0]}}{\partial x} \dot{x} \right) \right) \right)$$

```
In [1]: f = lambda x: jnp.sin(jnp.exp(x**2))

In [2]: jax.jvp(f, (0.3,), (1.0,))
(DeviceArray(0.88854975, dtype=float32, weak_type=True),
 DeviceArray(0.3011914, dtype=float32, weak_type=True))
```

- $\mathcal{F}(f, (x,), (\dot{x},)) = ((y,), (\dot{y}))$
- $OPS(\mathcal{F}(f, (x,), (\dot{x},))) \leq 2.5 \cdot OPS(f(x))$

# Pullback = vJp

$$\bar{y}\frac{\partial y}{\partial x} = \left(\left(\left(\left(\bar{y}\frac{\partial y}{\partial z^{[3]}}\right)\frac{\partial z^{[3]}}{\partial z^{[2]}}\right)\frac{\partial z^{[2]}}{\partial z^{[1]}}\right)\frac{\partial z^{[1]}}{\partial z^{[0]}}\right)\frac{\partial z^{[0]}}{\partial x}$$

```
In [3]: output, vjp_fun = jax.vjp(f, 0.3)

In [4]: vjp_fun(1.0)
Out[4]: (DeviceArray(0.3011914, dtype=float32, weak_type=True),)
```

- $\mathcal{B}(f, (x,), (\bar{y},)) = ((y,), (\bar{x},))$
- $OPS(\mathcal{B}(f, (x,), (\bar{y},))) \leq 4.0 \cdot OPS(f(x))$

# Automatic Differentiation

is a system to combine:

- Pushforward/Jvp rules for atomic operations into pushforward/Jvp
- Pullback/vJp rules for atomic operations into pullback/vJp

for larger computational graphs

- At some point, we have to implement symbolic derivatives for atomic operations

# Scalar Primitive Rules

| Primitive | Primal | Pushforward/Jvp | Pullback/vJp |
|---|---|---|---|
| **Explicit Scalar Rules** | | | |
| Scalar Addition | $z = x + y$ | $\dot{z} = \dot{x} + \dot{y}$ | $\bar{x} = \bar{z}$<br>$\bar{y} = \bar{z}$ |
| Scalar Multiplication | $z = x \cdot y$ | $\dot{z} = y \cdot \dot{x} + x \cdot \dot{y}$ | $\bar{x} = \bar{z} \cdot y$<br>$\bar{y} = \bar{z} \cdot x$ |
| Scalar Negation | $z = -x$ | $\dot{z} = -\dot{x}$ | $\bar{x} = -\bar{z}$ |
| Scalar Inversion | $z = \frac{1}{x}$ | $\dot{z} = -\frac{\dot{x}}{x^2}$ | $\bar{x} = -\frac{\bar{z}}{x^2}$ |
| Scalar Power | $z = x^l$ | $\dot{z} = l\, x^{l-1}\, \dot{x}$ | $\bar{x} = \bar{z}\, l\, x^{l-1}$ |

# Vector Automatic Differentiation

## via scalar operations is straightforward

each operation, e.g., matrix-vector multiplication, can be written in scalar operations (using loops, etc.)

- $y = f(x) = [x_0^3 \sin(x_1); x_2 x_1^2]$
- $x \in \mathbb{R}^3, y \in \mathbb{R}^2$ hence $\frac{\partial y}{\partial x} \in \mathbb{R}^{2\times 3}$

# Vector Pushforward / Vector Jvp

$$\mathcal{F}(f, (x,), (\dot{x})) = ((y,), (\dot{y} = \frac{\partial y}{\partial x}\dot{x}))$$

```
In [5]: f = lambda x: jnp.array([x[0]**3 * jnp.sin(x[1]), x[2]*x[1]**2])
In [6]: primal = jnp.array([1.0, 2.0, 3.0])
In [7]: tangent = jnp.array([1.0, 0.0, 0.0])

In [8]: jax.jvp(f, (primal,), (tangent,))
Out[8]:
(DeviceArray([ 0.9092974, 12.        ], dtype=float32),
 DeviceArray([2.7278922, 0.       ], dtype=float32))
```

# Vector Pullback / Vector vjp

$$\mathcal{B}(f, (x, ), (\bar{y})) = ((y, ), (\bar{x} = \left( \bar{y}^T \frac{\partial y}{\partial x} \right)^T, ))$$

```
In [9]: output, vjp_fun = jax.vjp(f, primal)

In [9]: cotangent = jnp.array([1.0, 0.0])

In [10]: vjp_fun(cotangent)
Out[10]: (DeviceArray([ 2.7278922 , -0.41614684,  0.        ], dtype=float32),)
```

# Detour: Mult. Matrices with Unit Vectors

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 4 \end{bmatrix}$$

$$\begin{bmatrix} 1 \\ 0 \end{bmatrix}^T \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}^T$$

# Obtaining Jacobians

- Now assume $f : \mathbb{R}^N \rightarrow \mathbb{R}^M$
  - $\mathcal{F}(f, (x,), (e_i))$ gives the $i$-th column of the Jacobian $J_f$
  - $\mathcal{B}(f, (x,), (e_i))$ gives the $i$-th row of the Jacobian $J_f$
- Hence, build full Jacobian $J \in \mathbb{R}^{M \times N}$ by:
  - batching $N$ pushforward evaluations
  - batching $M$ pullback evaluations

# Obtaining Jacobians II

- Consequentially:
  - $M > N$: forward-mode Jacobian more efficient
  - $M < N$: reverse-mode Jacobian more efficient (DL: $M = 1 \rightarrow \mathcal{O}(1)$)
  - $M \approx N$: forward-mode Jacobian more efficient due to smaller overhead

# Autodiff for BLAS-level operations

Example: `gemv` General Matrix-Vector multiplication

$$y = f(x, A, b) = Ax + b$$

- We could differentiate through the double for-loop, but we could also:
  - $\mathcal{F}(f, (x, A, b), (\dot{x}, \dot{A}, \dot{b})) = ((Ax + b, ), (A\dot{x} + \dot{A}x + \dot{b}, ))$
  - $\mathcal{B}(f, (x, A, b), (\bar{y}, )) = ((Ax + b, ), (W^T\bar{y}, \bar{y}x^T, \bar{y}, ))$
  - *JAX, TF, PyTorch, Zygote, etc. already do all that...*
- Express primitive rules again in terms of atomic operations

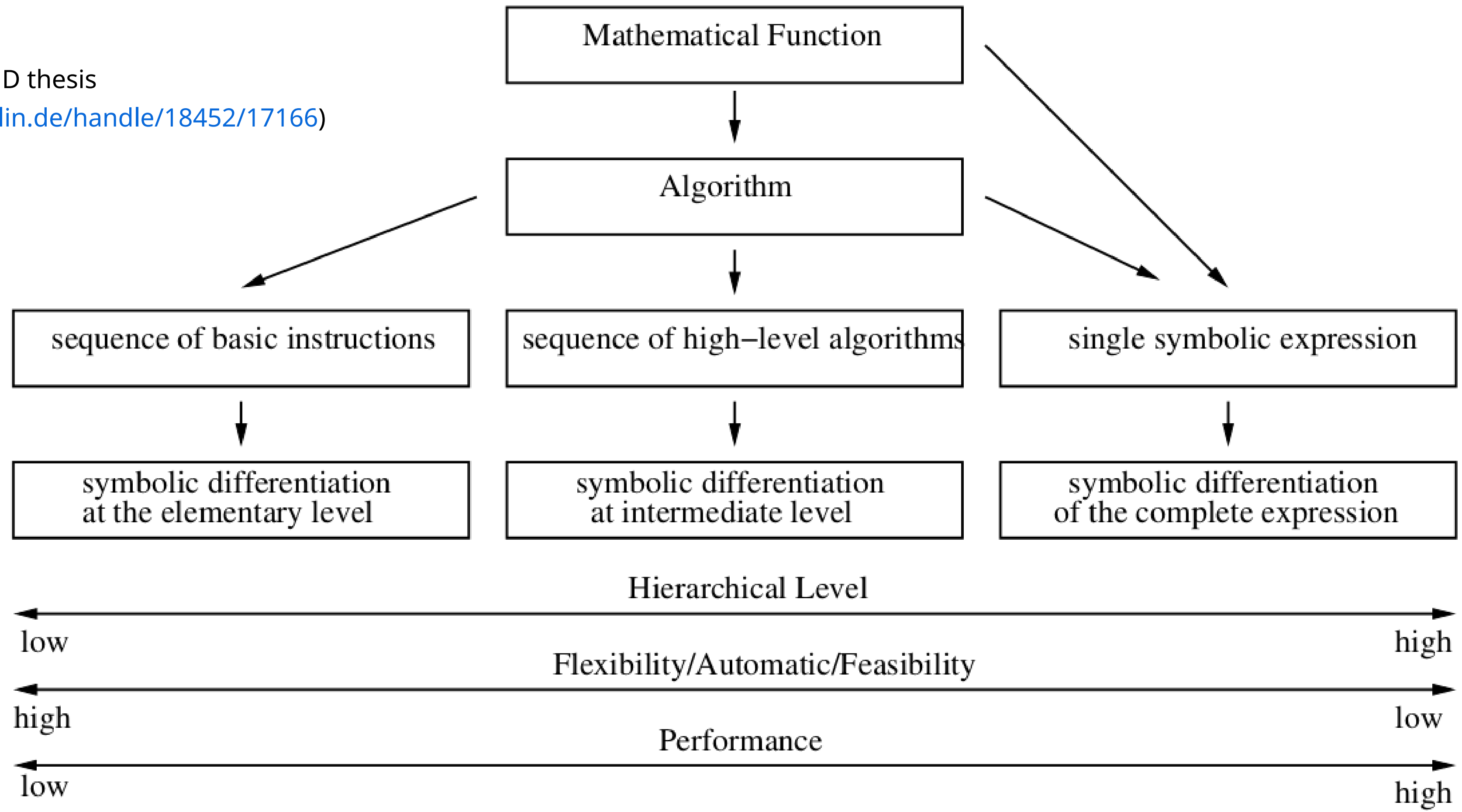
# BLAS-level Primitive Rules

| **Explicit Tensor Rules** | | | |
|---|---|---|---|
| Matrix-Vector Product | $\mathbf{z} = \mathbf{A}\mathbf{x}$ | $\dot{\mathbf{z}} = \dot{\mathbf{A}}\mathbf{x} + \mathbf{A}\dot{\mathbf{x}}$ | $\bar{\mathbf{x}} = \mathbf{A}^T\bar{\mathbf{z}}$<br>$\bar{\mathbf{A}} = \bar{\mathbf{z}}\mathbf{x}^T$ |
| Matrix-Matrix Product | $\mathbf{C} = \mathbf{A}\mathbf{B}$ | $\dot{\mathbf{C}} = \dot{\mathbf{A}}\mathbf{B} + \mathbf{A}\dot{\mathbf{B}}$ | $\bar{\mathbf{A}} = \bar{\mathbf{C}}\mathbf{B}^T$<br>$\bar{\mathbf{B}} = \mathbf{A}^T\bar{\mathbf{C}}$ |
| Scalar-Vector Product | $\mathbf{z} = \alpha\mathbf{x}$ | $\dot{\mathbf{z}} = \dot{\alpha}\mathbf{x} + \alpha\dot{\mathbf{x}}$ | $\bar{\mathbf{x}} = \bar{\mathbf{z}}\alpha$<br>$\bar{\alpha} = \bar{\mathbf{z}}^T\mathbf{x}$ |
| Scalar-Matrix Product | $\mathbf{C} = \alpha\mathbf{A}$ | $\dot{\mathbf{C}} = \dot{\alpha}\mathbf{A} + \alpha\dot{\mathbf{A}}$ | $\bar{\mathbf{A}} = \bar{\mathbf{C}}\alpha$<br>$\bar{\alpha} = \bar{\mathbf{C}} : \mathbf{A}$ |

https://fkoehler.site/autodiff-table/

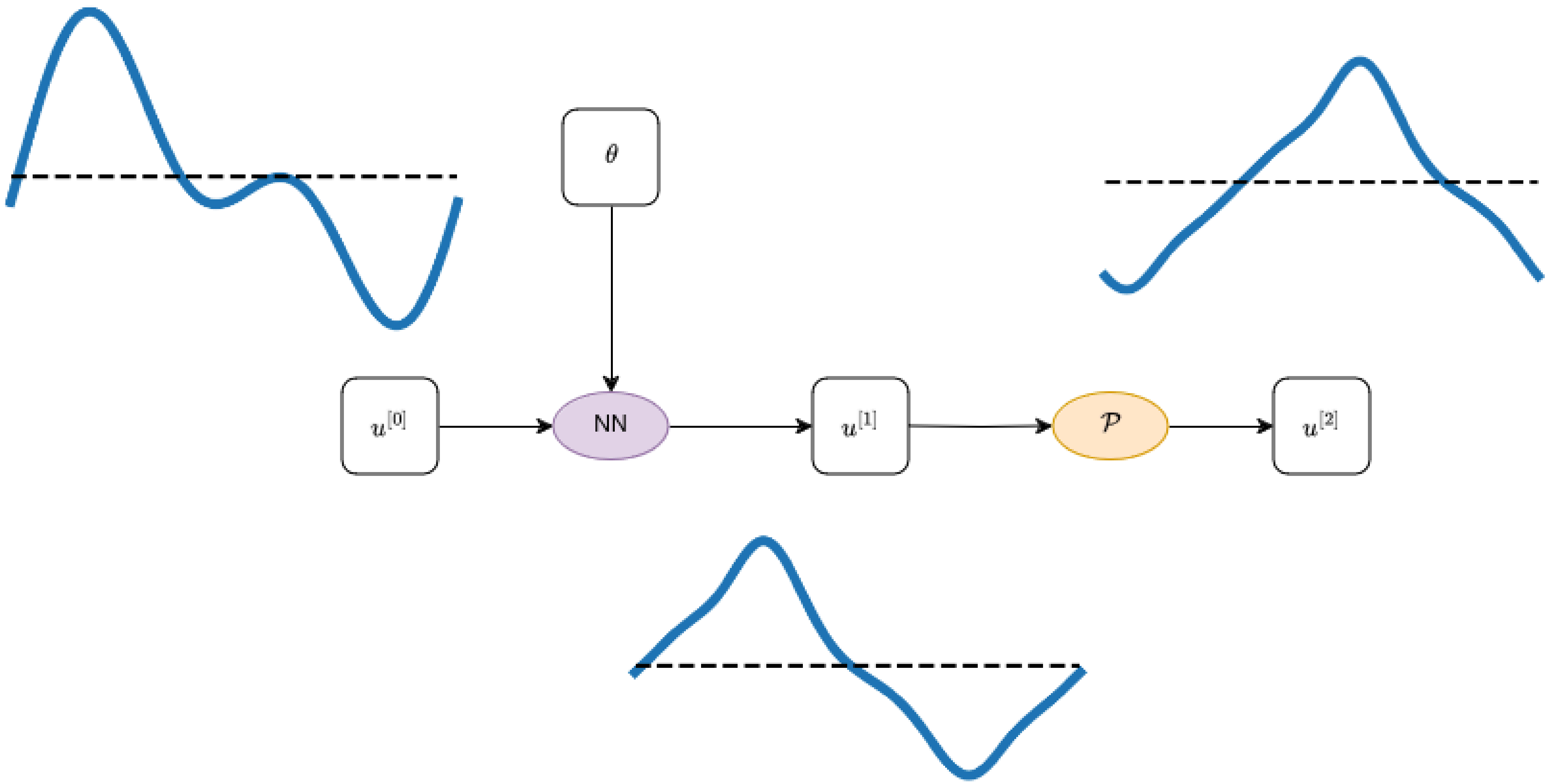
# Hierachies

* from S. Walther PhD thesis
(https://edoc.hu-berlin.de/handle/18452/17166)

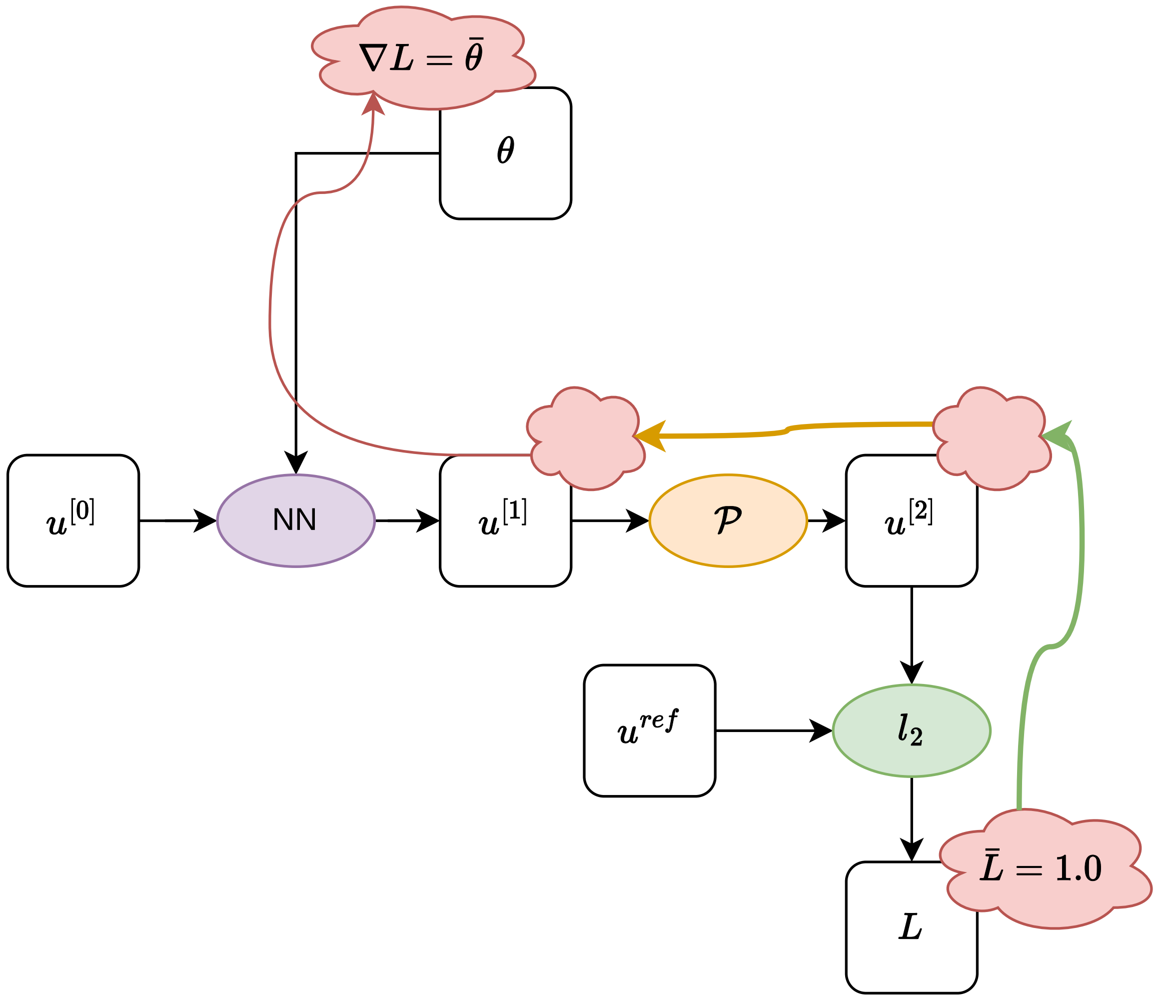



- **Float-Level** **BLAS-Level** **PDE-Level**

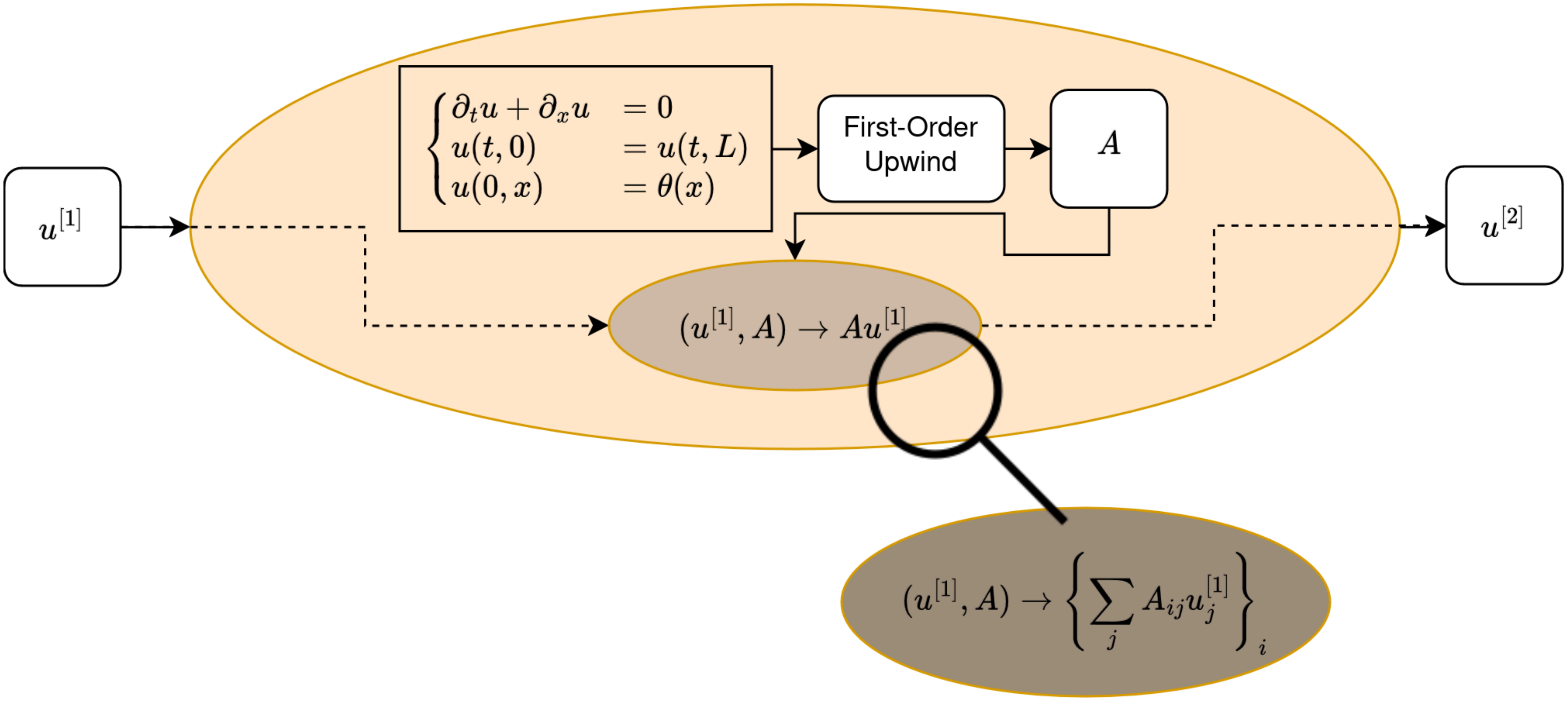
# Compute Graph with Diff. Physics



# Compute Graph with Diff. Physics II



# How to differentiate through PDEs?



# Continuous Adjoint Advection Equation

- Primal Physics
- $u = \mathcal{P}(\theta) = \{\text{integrate from } t = 0 \text{ to } t = \Delta t \begin{cases} \partial_t u + \partial_x u & = 0 \\ u(t, 0) & = u(t, L) \\ u(0, x) & = \theta(x) \end{cases}$
- Adjoint Physics
- $\bar{\theta} = \bar{\mathcal{P}}(\bar{u}) = \{\text{integrate from } t = \Delta t \text{ to } t = 0 \begin{cases} \partial_t \lambda - \partial_x \lambda & = 0 \\ \lambda(t, 0) & = \lambda(t, L) \\ \lambda(\Delta t, x) & = \bar{u}(x) \end{cases}$

# The Buzzwords

## Discretize-then-Optimize (DtO)

## Optimize-then-Discretize (OtD)

- But really ... it is a spectrum

# Levels of hierarchy

| Level | vJp-level | Memory | Tool |
|---|---|---|---|
| PDE | functional | result only | Dolfin/FEniCs-adjoint |
| BLAS | tensor | every algebra operation | PyTorch, TF, JAX, Zygote etc. |
| Scalar | scalar | every float | Scalar AD engines |

- BLAS-level rules are the OtD for scalar-mode AD
- PDE-level rules are the OtD for tensor-mode AD

# History

| 1960s | 1970s | 1980s |
|---|---|---|
| Precursors | **Linnainmaa, 1970, 1976** *Backpropagation* | **Speelpenning, 1980** *Automatic reverse mode* |
| Kelley, 1960<br>Bryson, 1961<br>Pontryagin et al., 1961<br>Dreyfus, 1962 | Dreyfus, 1973 *Control parameters* | Werbos, 1982 *First NN-specific backprop* |
| **Wengert, 1964** *Forward mode* | Werbos, 1974 *Reverse mode* | Parker, 1985 |
| | | LeCun, 1985 |
| | | **Rumelhart, Hinton, Williams, 1986** *Revived backprop* |
| | | **Griewank, 1989** *Revived reverse mode* |

29

* Unfortunately, I lost the source of the presentation where I took this slide from

# Comparison

| | OtD | DtO |
|---|---|---|
| Derivation | ❌ Requires manual derivation of adjoint code (including adjoint BC!) | ✅ automatic |
| Intrusiveness | ✅ Never open black box | ❌ Requires code to be written in a differentiable way |
| Performance | ✅ Can be faster | ❌ Can be slower |

- Loosely speaking: Manual code optimization vs. `gcc -O3`

# Comparison II

| | OtD | DtO |
|---|---|---|
| Memory | ✅ Might require to only save input and output | ❌ Tape all intermediary steps |
| Exactness | ☑️ Exact wrt continuous objective | ✅ Exact wrt discrete objective (better for discrete optimization like machine learning) |
| Debugging | ❌ hard | ☑️ medium |

- My advice: Use BLAS-level DTO, but be aware of its shortcomings.Switch to fully continuous OtD only for hardcore performance optimization.

# Specialities of Differentiable Physics

# Example: NS Pressure-Poisson Solve

```python
from phi.flow import *
velocity = StaggeredGrid(0, x=64, y=64, bounds=Box(x=100, y=100))
smoke = CenteredGrid(0, ZERO_GRADIENT, x=200, y=200, bounds=Box(x=100, y=100))
INFLOW = 0.2 * resample(Sphere(x=50, y=9.5, radius=5), to=smoke, soft=True)
pressure = None

def step(v, s, p, dt=1.):
    s = advect.mac_cormack(s, v, dt) + INFLOW
    buoyancy = resample(s * (0, 0.1), to=v)
    v = advect.semi_lagrangian(v, v, dt) + buoyancy * dt
    ### ---> Linsolve start <---
    v, p = fluid.make_incompressible(v, (), Solve(x0=p))
    ### ---> Linsolve end <---
    return v, s, p

for _ in range(10):
    velocity, smoke, pressure = step(velocity, smoke, pressure)
```

https://github.com/tum-pbs/PhiFlow/blob/c4cec7ba9e62209c7bcfefeba7d87a42fa8a8193/demos/smoke_plume.py

# Pressure-Poisson Solve

- Requirement on continuity: $\nabla \cdot \mathbf{v} = 0$
- Leads to a Poisson equation for the pressure: $\nabla^2 p = \nabla \cdot \mathbf{v}^*$
- To then correct the velocity field: $\mathbf{v}^{**} = \mathbf{v}^* - \nabla p$

- Discrete form: $A p_h = b_h$

# Conjugate Gradient Algorithm

$$\mathbf{r}_0 := \mathbf{b} - \mathbf{A}\mathbf{x}_0 \qquad \mathbf{p}_0 := \mathbf{r}_0 \qquad k := 0$$

repeat

$$\alpha_k := \frac{\mathbf{r}_k^\top \mathbf{r}_k}{\mathbf{p}_k^\top \mathbf{A}\mathbf{p}_k}$$

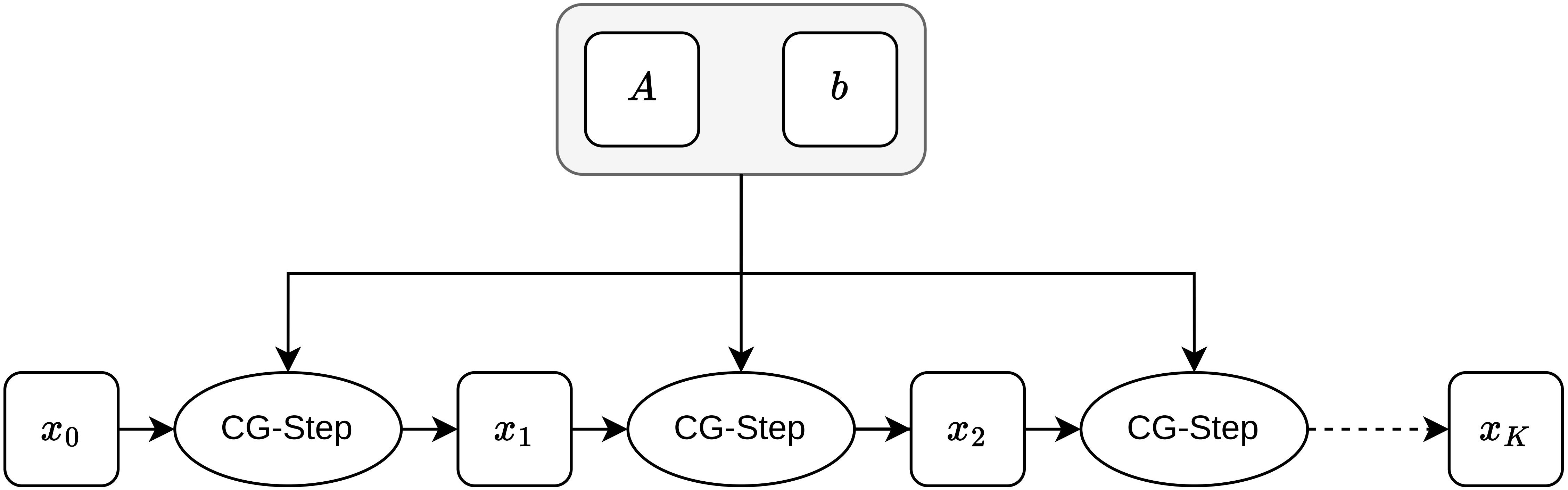
$$\mathbf{x}_{k+1} := \mathbf{x}_k + \alpha_k \mathbf{p}_k \qquad \mathbf{r}_{k+1} := \mathbf{r}_k - \alpha_k \mathbf{A}\mathbf{p}_k$$

if $\mathbf{r}_{k+1}$ is sufficiently small, then exit loop

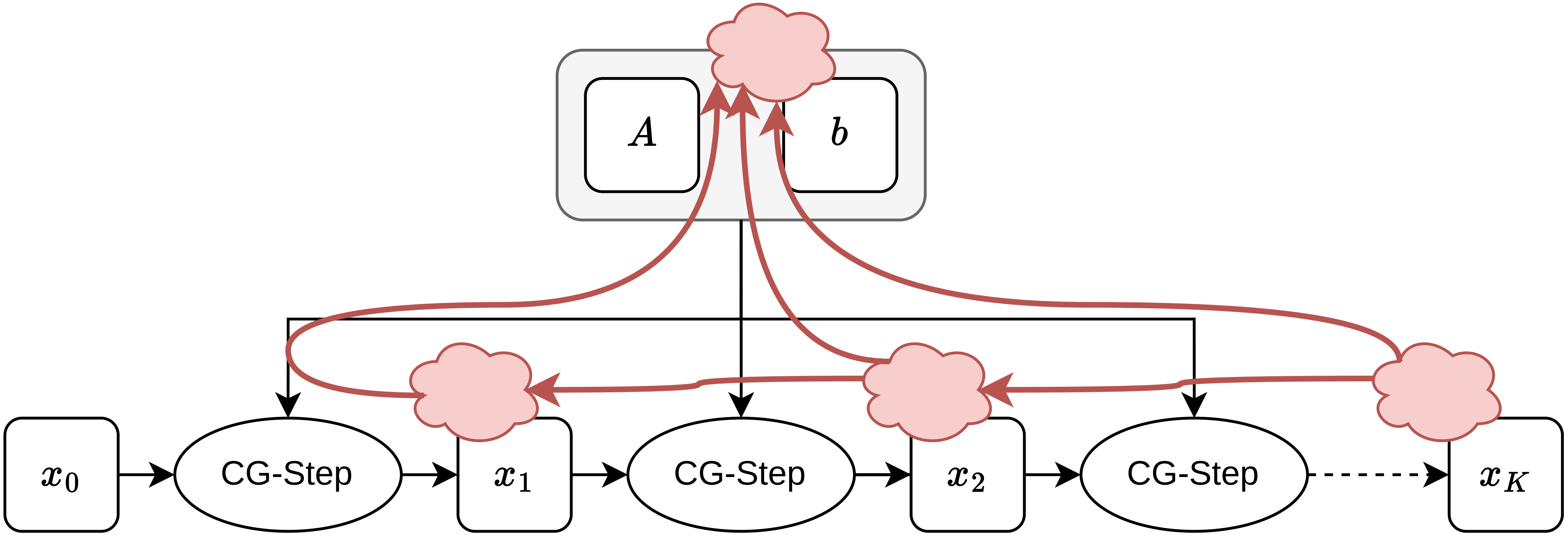
$$\beta_k := \frac{\mathbf{r}_{k+1}^\top \mathbf{r}_{k+1}}{\mathbf{r}_k^\top \mathbf{r}_k} \qquad \mathbf{p}_{k+1} := \mathbf{r}_{k+1} + \beta_k \mathbf{p}_k$$

return $\mathbf{x}_{k+1}$ as the result

# CG Compute Graph



# CG Compute Graph with Reverse Pass



# Differentiating through CG Solve

1. Unroll all iterations, build compute graph and transform (**Unrolled Diff**):
   - ✅ Exact derivative of all operations
   - ✅ Automatic, no modifications of code
   - ❌ Need to tape all iterations
   - ❌ Derivative Convergence might be different from primal convergence!
2. Find custom adjoint rule (what PhiFlow does) (**Implicit Diff**):
   - ✅ Can be faster
   - ✅ Less memory consumption in reverse mode
   - ❌ Manual effort, because requires fiddling with the autodiff engine
   - ❌ Might be hard to get right

# vJp rule for Linear System Solving

Primal

$$\mathbf{x} = \{\text{solve } \mathbf{A}\mathbf{x} = \mathbf{b} \text{ for } \mathbf{x}\}$$

Reverse Rule

$$\begin{aligned} \lambda &= \left\{\text{solve } \mathbf{A}^T \lambda = \bar{\mathbf{x}} \text{ for } \lambda\right\} \\ \bar{\mathbf{b}} &= \lambda \\ \bar{\mathbf{A}} &= -\lambda \mathbf{x}^T \end{aligned}$$

- Adjoint rules is again a linsolve but with $\mathbf{A}^T$

# Registering linsolve custom adjoint rule

```python
def _cg_solve(A, b):
  # Solve Ax = b

@jax.custom_vjp
def cg_solve(A, b):
  x = _cg_solve(A, b)
  return x

def cg_solve_fwd(A, b):
  x = _cg_solve(A, b)
  return x, (A, x)

def cg_solve_bwd(res, g):
  A, x = res
  lam = _cg_solve(A.T, g)
  return (-jnp.outer(lam, x), lam)
```

# Example: Nonlinear FEM Solve in FEniCs

$$-\nabla \cdot ((1+u^2)\nabla u) = f(\theta) \quad \text{in}\,\Omega, \quad u = 1 \quad \text{on}\,\Gamma_D, \quad \nabla u \cdot n = 0 \quad \text{on}\,\Gamma_N$$

```
from dolfin import *

class DirichletBoundary(SubDomain):
    def inside(self, x, on_boundary):
        return abs(x[0] - 1.0) < DOLFIN_EPS and on_boundary

mesh = UnitSquareMesh(32, 32); V = FunctionSpace(mesh, "CG", 1)
g = Constant(1.0); bc = DirichletBC(V, g, DirichletBoundary())
u = Function(V); v = TestFunction(V); f = Expression("x[0]*sin(x[1])")
F = inner((1 + u**2)*grad(u), grad(v))*dx - f*v*dx

solve(F == 0, u, bc, solver_parameters={"newton_solver":
                                        {"relative_tolerance": 1e-6}})
```

# Newton-Raphson Algorithm

$$
\begin{aligned}
&\mathbf{u}_0 \leftarrow \text{initial guess} \\
&\text{repeat} \\
&\qquad \mathbf{r}_k = \mathbf{F}(\mathbf{u}_k) \\
&\qquad \text{if } \mathbf{r}_k \text{ is sufficiently small, then exit loop} \\
&\qquad \text{linsolve} \quad \left.\frac{\partial \mathbf{F}}{\partial \mathbf{u}}\right|_{\mathbf{u}_k} \Delta \mathbf{u}_k = -\mathbf{r}_k \\
&\qquad \mathbf{u}_{k+1} = \mathbf{u}_k + \Delta \mathbf{u}_k
\end{aligned}
$$

# Unroll-Diff through Newton-Raphson

1. Unroll all iterations, build compute graph and transform (assume you do implicit diff to all linsolves):
   - ✅ Exact derivative of all operations (given exact linsolves)
   - ✅ Automatic, no modifications of code (given there is an implicit rule for the linsolve)
   - ❌ Need to tape all iterations
   - ❌ Derivative Convergence might be different from primal convergence!
   - ❌ **Reverse pass has to solve as many linear systems as primal pass**

# Implicit-Diff through Newton-Raphson

2. Find custom implicit rule:
    - ✅ Certainly be faster because needs only one linsolve
    - ✅ Less memory consumption in reverse-mode
    - ❌ Intrusive because requires fiddling with the autodiff engine
    - ❌ Might be hard to get right

# Nonlinear Solve Custom Adjoint Rule

Primal

$$\mathbf{x} = \{\text{solve } \mathbf{g}(\mathbf{x}, \theta) = \mathbf{0} \text{ for } \mathbf{x}\}$$

Reverse Rule

$$\lambda = \left\{ \text{solve } \left(\frac{\partial \mathbf{g}}{\partial \mathbf{x}}\right)^T \lambda = \bar{\mathbf{x}} \text{ for } \lambda \right\}$$

$$\bar{\theta} = -\left(\frac{\partial \mathbf{g}}{\partial \theta}\right)^T \lambda$$

# General Insights and Tips

- The Jvp/vJp propagation will always be linear!
- Especially if primal is a nonlinear solve, implicit propagation solve (for forward and reverse mode) will be linear solve and hence way cheaper
- Custom implicit rules require informing the autodiff engine:
    - JAX already comes with custom rules for `jax.numpy.linalg.solve` and `jax.scipy.sparse.linalg.XXX` with `XXX` $\in$ { `cg`, `bicgstab`, `gmres` }
    - If you have an algebra function calling into a third-party library, always custom rule (cannot open black box):
        - Promising tool: Enzyme

# Implicit Primitive Rules

| Primitive | Primal | Pushforward/Jvp | Pullback/vJp |
|---|---|---|---|
| **Discrete Problems** | | | |
| Scalar Root-Finding | $x = \{\text{solve } g(x, \theta) \text{ for } x\}$ | $\dot{x} = -\frac{\frac{\partial g}{\partial \theta}}{\frac{\partial g}{\partial x}} \dot{\theta}$ | $\bar{\theta} = -\bar{x} \frac{\frac{\partial g}{\partial \theta}}{\frac{\partial g}{\partial x}}$ |
| Linear System Solving | $\mathbf{x} = \{\text{solve } \mathbf{A}\mathbf{x} = \mathbf{b} \text{ for } \mathbf{x}\}$ | $\mathbf{d} = \dot{\mathbf{b}} - \dot{\mathbf{A}}\mathbf{x}$<br>$\dot{\mathbf{x}} = \{\text{solve } \mathbf{A}\dot{\mathbf{x}} = \mathbf{d} \text{ for } \dot{\mathbf{x}}\}$ | $\lambda = \{\text{solve } \mathbf{A}^T \lambda = \bar{\mathbf{x}} \text{ for } \lambda\}$<br>$\bar{\mathbf{b}} = \lambda$<br>$\bar{\mathbf{A}} = -\lambda \mathbf{x}^T$ |
| Nonlinear System Solving | $\mathbf{x} = \{\text{solve } \mathbf{g}(\mathbf{x}, \theta) = \mathbf{0} \text{ for } \mathbf{x}\}$ | $\mathbf{d} = -\frac{\partial \mathbf{g}}{\partial \theta} \dot{\theta}$<br>$\dot{\mathbf{x}} = \left\{\text{solve } \frac{\partial \mathbf{g}}{\partial \mathbf{x}} \dot{\mathbf{x}} = \mathbf{d} \text{ for } \dot{\mathbf{x}}\right\}$ | $\lambda = \left\{\text{solve } \left(\frac{\partial \mathbf{g}}{\partial \mathbf{x}}\right)^T \lambda = \bar{\mathbf{x}} \text{ for } \lambda\right\}$<br>$\bar{\theta} = -\left(\frac{\partial \mathbf{g}}{\partial \theta}\right)^T \lambda$ |

https://fkoehler.site/implicit-autodiff-table/

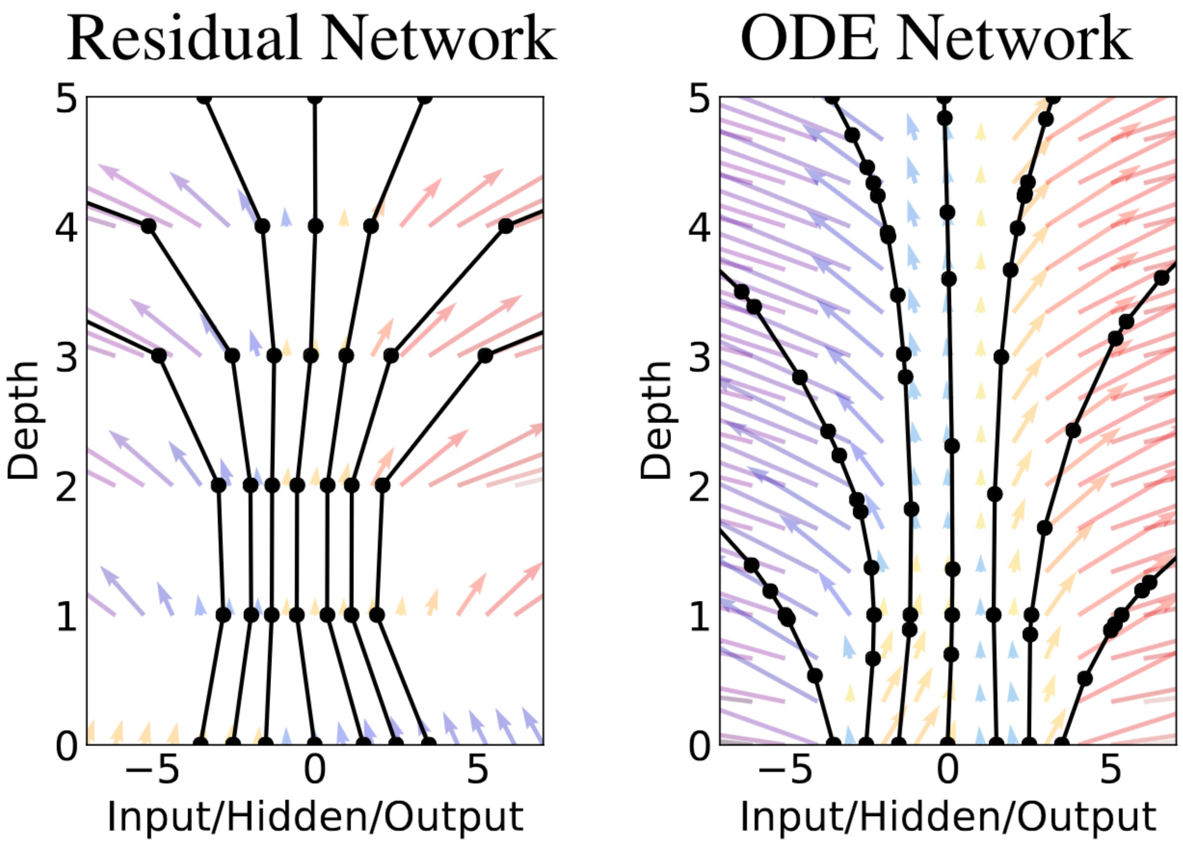
# Levels of hierarchy (revisited)

| Level | vJp-level | memory | Tool |
|---|---|---|---|
| PDE | functional | result only | Dolfin/FEniCs-adjoint |
| Algebra | tensor+custom rules | each algebra operation and implicit function | *Need to be made aware* |
| BLAS | tensor | each algebra operation | PyTorch, TF, JAX, Zygote etc. |
| Scalar | scalar | every float | Scalar AD engines |

# Advanced Topics

# Cont. Repr. for NNs

- Neural ODEs (Chen et al. 2018):
  - Inference via integration of an ODE (with continuous adjoint)
- Deep Equilibrium Networks (Bai et al. 2019):
  - Inference via solution to a root-finding problem (with adjoint linear solve)



# Auto-Implicit Diff

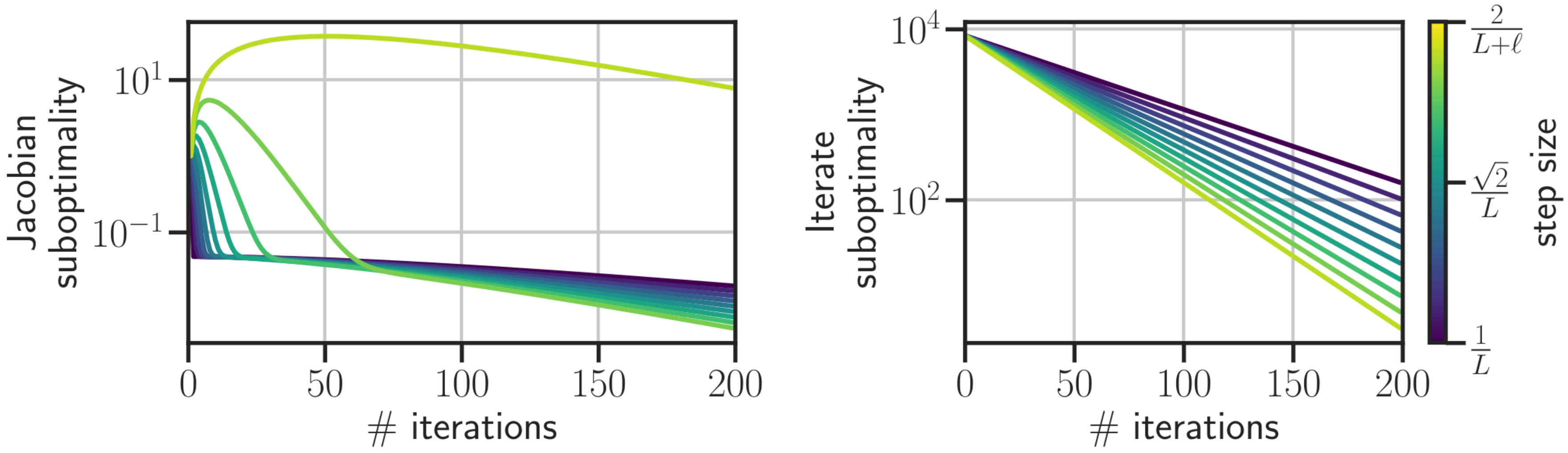
- Blondel et al. 2022 "Efficient and Modular Implicit Differentiation"
- Given an optimality condition, automatically register (co-)tanget propagation rules within JAX
  - Internally performs matrix-free linear solves with linearizing the optimality condition

```python
X_train, y_train = load_data()  # Load features and labels

def f(x, theta):  # Objective function
  residual = jnp.dot(X_train, x) - y_train
  return (jnp.sum(residual ** 2) + theta * jnp.sum(x ** 2)) / 2

# Since f is differentiable and unconstrained, the optimality
# condition F is simply the gradient of f in the 1st argument
F = jax.grad(f, argnums=0)

@custom_root(F)
def ridge_solver(init_x, theta):
  del init_x  # Initialization not used in this solver
  XX = jnp.dot(X_train.T, X_train)
  Xy = jnp.dot(X_train.T, y_train)
  I = jnp.eye(X_train.shape[1])  # Identity matrix
  # Finds the ridge reg solution by solving a linear system
  return jnp.linalg.solve(XX + theta * I, Xy)

init_x = None
print(jax.jacobian(ridge_solver, argnums=1)(init_x, 10.0))
```

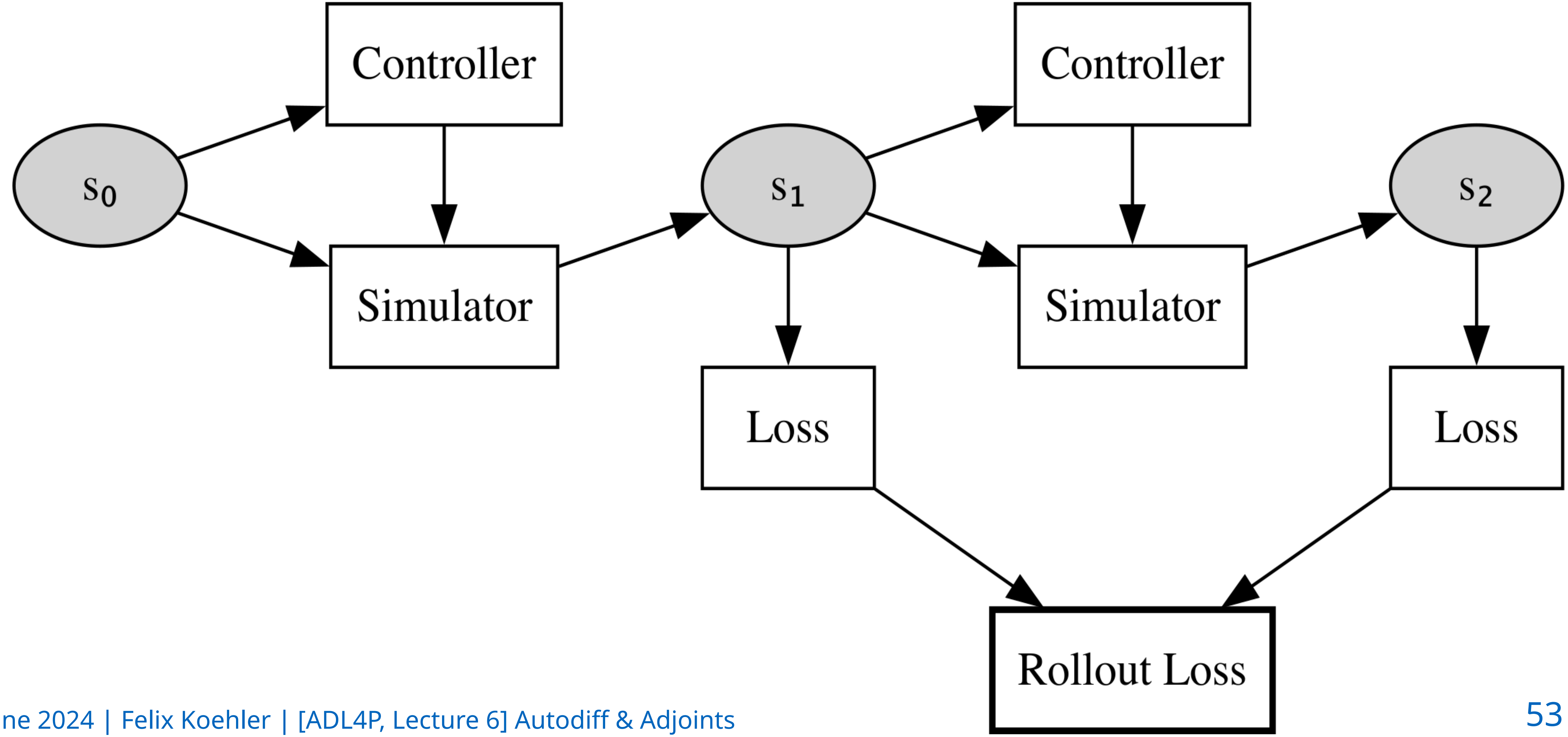
# Curse of Unrolling

- Even if your primal converges (exponentially) linear, the derivative (=Jacobian) might not initially (Scieur et al. "The Curse of Unrolling: ...")

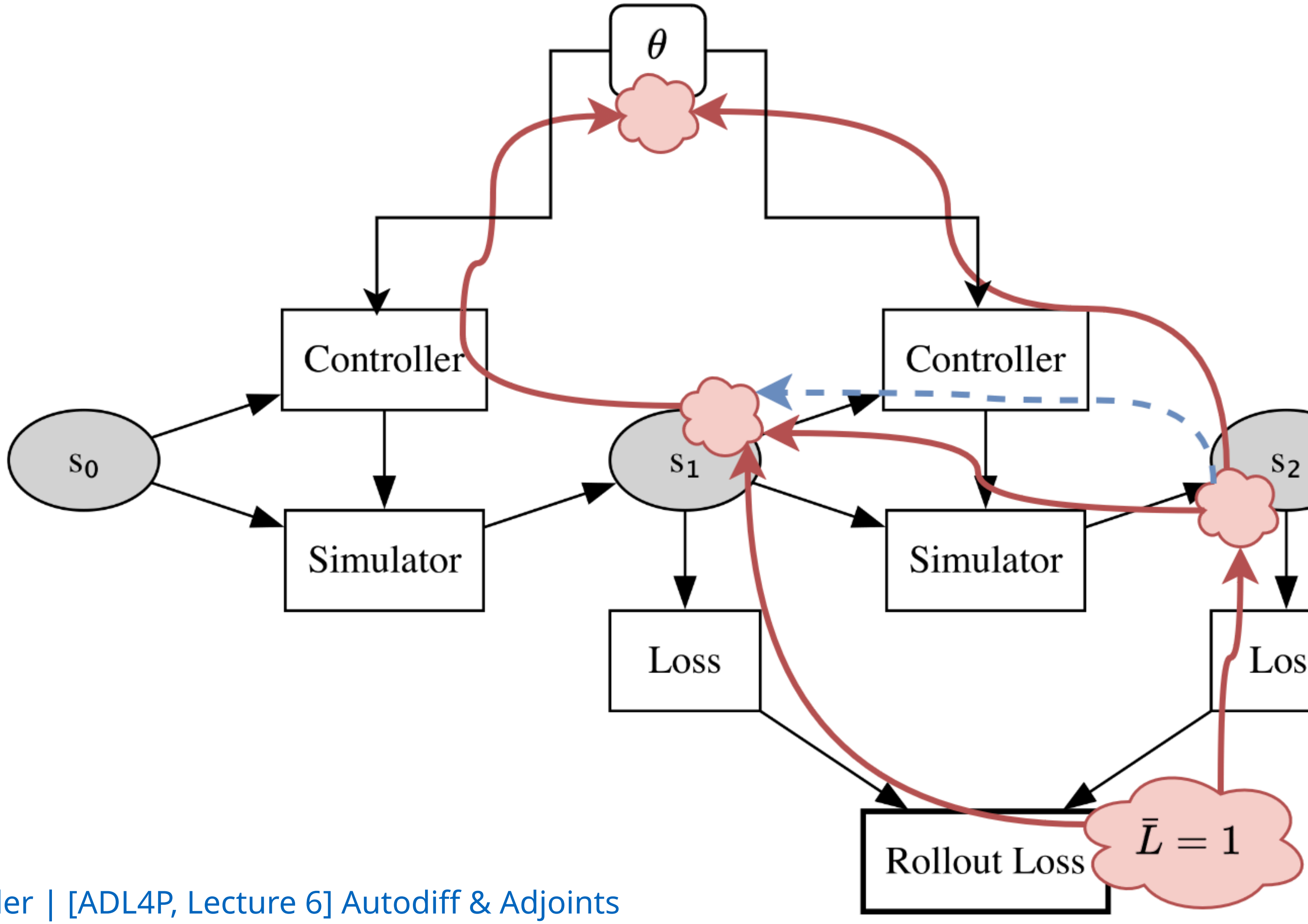
```
To ensure convergence of the Jacobian with gradient descent, we must either 1) accept that
the algorithm has a burn-in period proportional to the condition number 1/κ, or 2) choose
a small step size that will slow down the algorithm’s asymptotic convergence
```



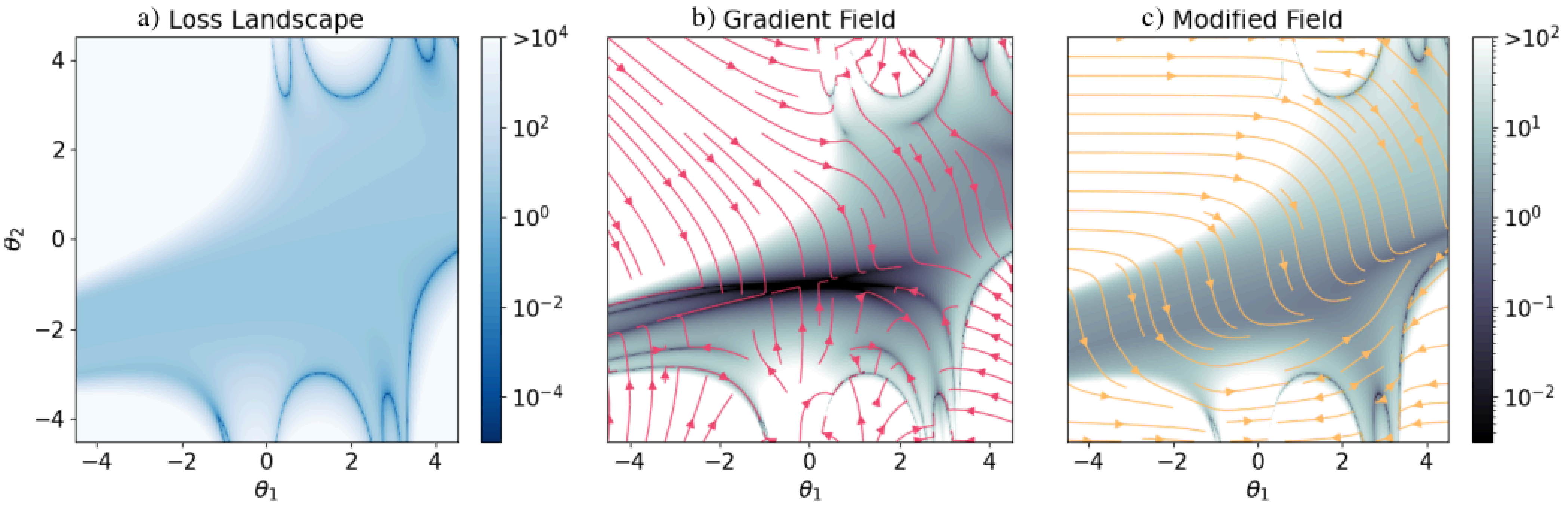
# Strategic Gradient Cuts



# Strategic Gradient Cuts II



# Strategic Gradient Cuts III




"Stabilizing Backpropagation Through Time ..." by Schnell & Thuerey 2024

# Additional Topics

- Approximate Gradients (not fully execute iterative processes):
  - "Hyperparameter optimization with approximate gradient" (Pedregosa 2016)
  - "One-step differentiation of iterative algorithms" (Bolte et al. 2023)
- Automated Continuous Adjoint Derivation:
  - Dolfin-Adjoint for FEniCs
- Avoiding Differentiable Physics:
  - "How Temporal Unrolling Supports Neural Physics Simulators" (List et al. 2023)

# Interested?

**There are so many cool topics and open questions!**

Feel free to contact me if you want to discuss any of these topics or have any questions!

# Conclusion

# Summary

- Autodiff is a system to combine pushforward/jvp and pullback/vjp rules for atomic operations
  - We need to define atomic operations with symbolic derivatives
  - Atomic operations can be on scalar-level, BLAS-level or continuous PDE-level (with a spectrum in-between)
  - Taking gradients is syntactic sugar for pushforward and pullback
- Always think input-output: Even continuous adjoints will eventually have discrete inputs and outputs
- Machine Learning often works well with slightly inaccurate gradients (stochastic anyway); just get the gradients flowing 😉

# Additional Resources

- The definitive book on the mathematical perspective of Autodiff: "Evaluating Derivatives: ..." by Griewank and Walther
- A more digestible read for machine learning: "Automatic Differentiation in Machine Learning: ..." by Baydin et al.
- Refresher on Backpropagation from the modern "Understanding Deep Learning" Book by Prince (Chapter 7 "Gradients and Initialization")
- JAX tutorial on Autodiff and custom primtive rules
- Matthew Johnson's talk on Autograd
- Chapter 8 and Chapter 10 of Chris Rackauckas' SciML Book
- ChainRules.jl ecosystem in Julia