The Pixel Array method for nonlinear systems, and applications to numerical PDEs

David I. Spivak

dspivak@math.mit.edu
Mathematics Department
Massachusetts Institute of Technology

Presented on 2016/10/26
in the Numerical Methods for PDE seminar

Outline

Outline

What to expect from the talk

The subject of this talk is a new method for solving nonlinear systems.

- It's very different: you get *all solutions* inside a bounding box.

What to expect from the talk

The subject of this talk is a new method for solving nonlinear systems.

- It's very different: you get *all solutions* inside a bounding box.
  - It arose from my work on *applied category theory*.
  - There will be no category theory (CT) in this talk.
  - The idea originated there, but all you need to know is matrix mult.

What to expect from the talk

The subject of this talk is a new method for solving nonlinear systems.

- It's very different: you get *all solutions* inside a bounding box.
  - It arose from my work on *applied category theory*.
  - There will be no category theory (CT) in this talk.
  - The idea originated there, but all you need to know is matrix mult.
- The talk is more about *solving systems of equations* than about PDE.
  - I'll discuss how the method applies to PDE near the end.
- I know next to nothing about PDEs and numerical methods.
  - I'm a foreigner here; please forgive my 'dialect' and 'cultural ignorance'.
  - I'd appreciate any feedback you may have.
  - I hope our communities—applied math and CT—can become friends.

An image, for your imagination

Separately plot the solutions to equations: $f(x, w) = 0$ and $g(w, y) = 0$.

- Plot each in its own bounding box, say in the range $[-1.5, 1.5]$.
- Consider the plots as matrices M, N whose entries are on/off pixels.
- That is, M and N are *boolean matrices* corresponding to f and g.

An image, for your imagination

Separately plot the solutions to equations: $f(x, w) = 0$ and $g(w, y) = 0$.

- Plot each in its own bounding box, say in the range $[-1.5, 1.5]$.
- Consider the plots as matrices M, N whose entries are on/off pixels.
- That is, M and N are *boolean matrices* corresponding to f and g.

Multiplying these two matrices MN yields...

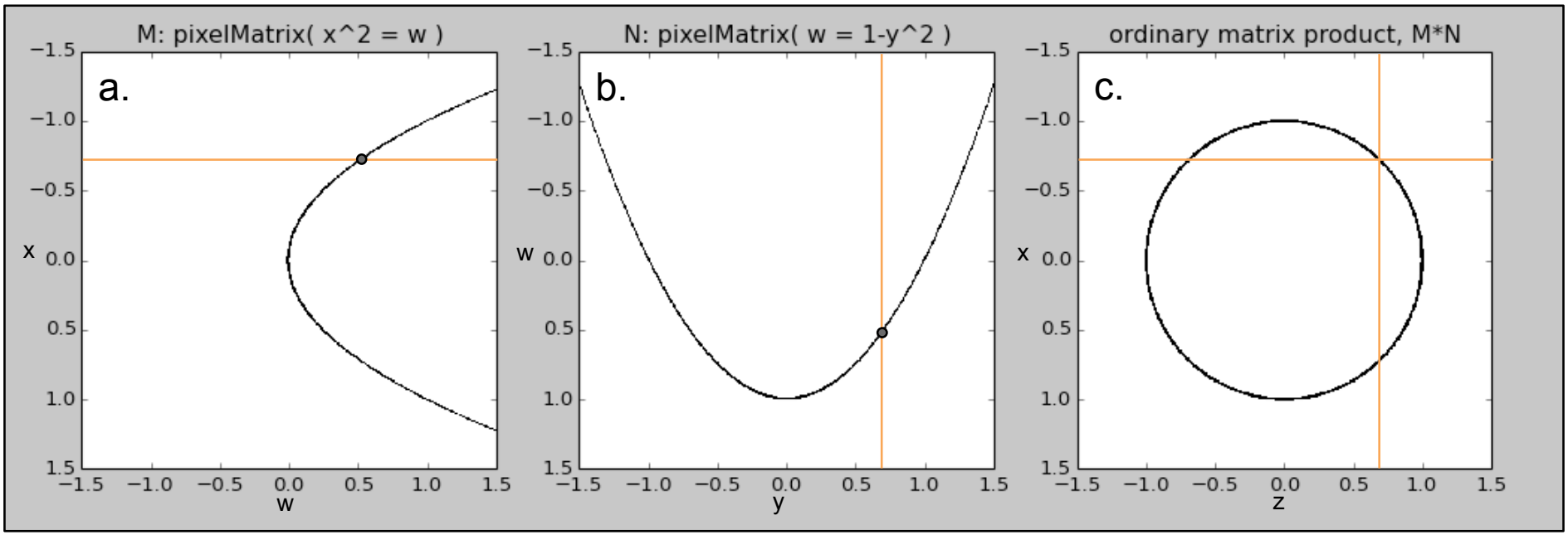
An image, for your imagination

Separately plot the solutions to equations: $f(x, w) = 0$ and $g(w, y) = 0$.

- Plot each in its own bounding box, say in the range $[-1.5, 1.5]$.
- Consider the plots as matrices M, N whose entries are on/off pixels.
- That is, M and N are *boolean matrices* corresponding to f and g.

Multiplying these two matrices MN yields the simultaneous solution.

- For example, plot equations $x^2 = w$ and $w = 1 - y^2$, and multiply.



A more complex example

The following eq's are not differentiable, nor even defined everywhere.

$$\cos\left(\ln(z^2 + 10^{-3}x)\right) - x + 10^{-5}z^{-1} = 0 \qquad \text{(Equation 1)}$$

$$\cosh(w + 10^{-3}y) + y + 10^{-4}w = 2 \qquad \text{(Equation 2)}$$

$$\tan(x + y)(x - 2)^{-1}(x + 3)^{-1}y^{-2} = 1 \qquad \text{(Equation 3)}$$

Q: For what values of w and z does a simultaneous solution exist? [1]

[1] Spivak, DI; Dobson, MRC; Kumari, S. (2016) "Pixel Arrays: A fast and elementary method for solving nonlinear systems". http://arxiv.org/pdf/1609.00061v1.pdf

A more complex example

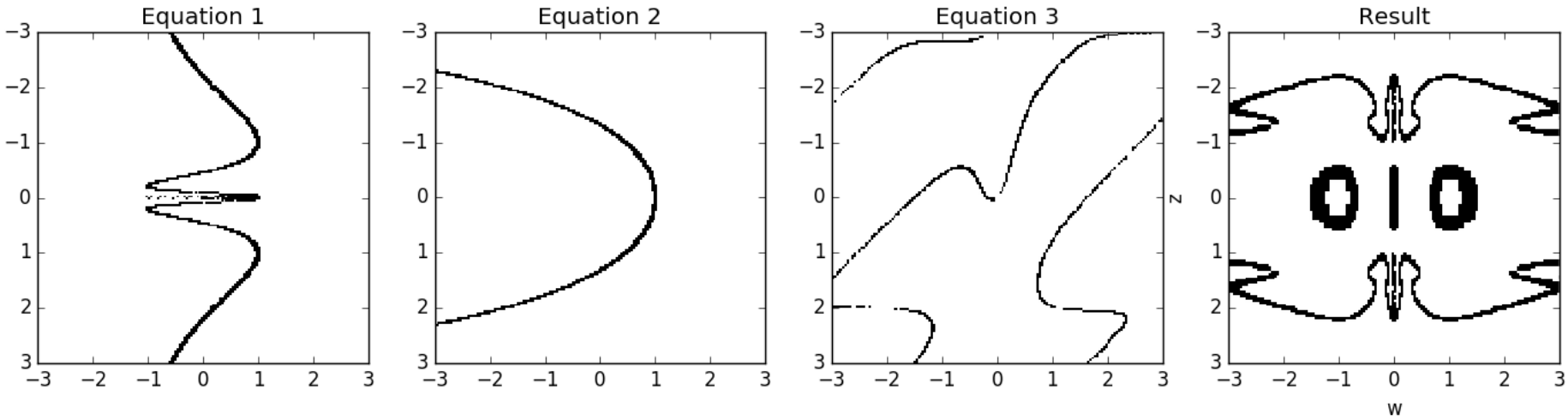
The following eq's are not differentiable, nor even defined everywhere.

$$\cos\left(\ln(z^2 + 10^{-3}x)\right) - x + 10^{-5}z^{-1} = 0 \qquad \text{(Equation 1)}$$

$$\cosh(w + 10^{-3}y) + y + 10^{-4}w = 2 \qquad \text{(Equation 2)}$$

$$\tan(x + y)(x - 2)^{-1}(x + 3)^{-1}y^{-2} = 1 \qquad \text{(Equation 3)}$$

Q: For what values of w and z does a simultaneous solution exist? [1]



[1] Spivak, DI; Dobson, MRC; Kumari, S. (2016) "Pixel Arrays: A fast and elementary method for solving nonlinear systems". http://arxiv.org/pdf/1609.00061v1.pdf

Another way to linearize

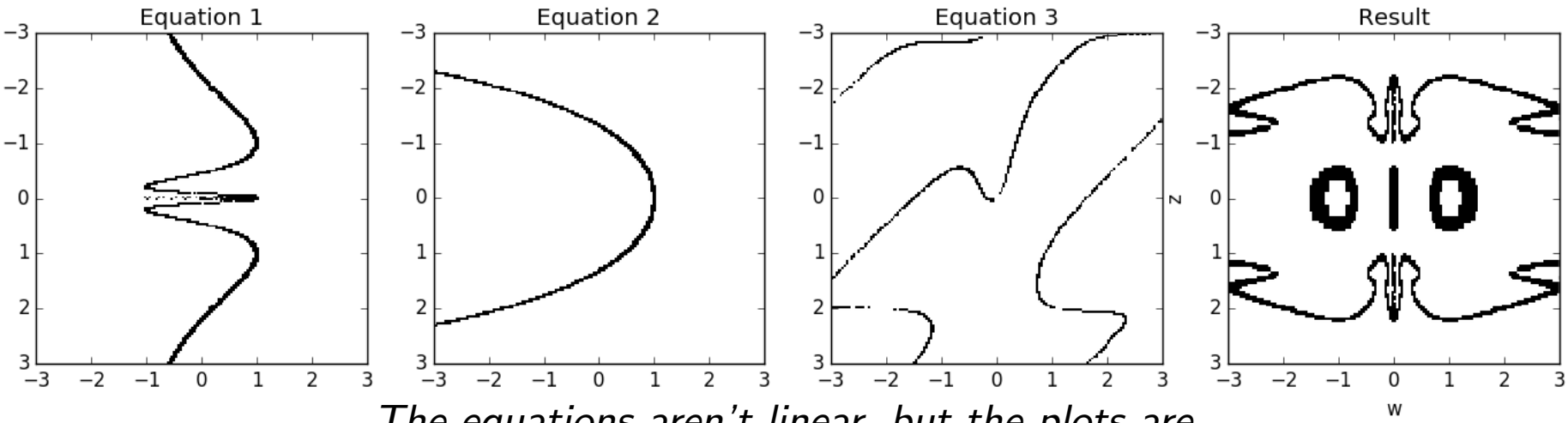
A popular refrain in mathematics:

- The only problems we know how to solve are linear ones.
- We approach any other problem by trying to linearize it.

Another way to linearize

A popular refrain in mathematics:

- The only problems we know how to solve are linear ones.
- We approach any other problem by trying to linearize it.



The equations aren't linear, but the plots are.

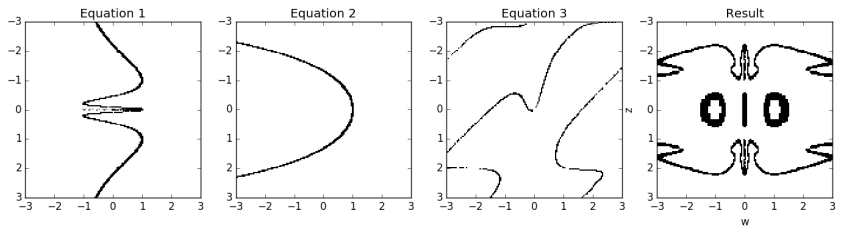
The Pixel Array slogan: *Plotting is a way of linearizing*.

- Once we plot, we have matrices (more generally, arrays / tensors).
- We can use *linear algebra* to put plots together and solve big systems.

Selling points

The Pixel Array method has the following features:

- it returns *all solutions in a given bounding box*;
- it's *much faster* than quasi-Newton methods for finding "all solutions";
- it introduces *no false negatives*;
- it works for *non-differentiable* or even *discontinuous* functions;
- it's *not iterative* and requires no initial guess, in contrast with quasi-Newton methods;
- it's *elementary*—relying only on generalized matrix arithmetic—hence has low barrier to entry; and
- it *provides insights*, by showing the whole solution set at once.



So where are the limitations hiding?

Takes original plots as input.

- The Pixel Array (PA) method requires a plot for each equation.
- The plot of $f(x_1, \ldots, x_n) = 0$ is n-dimensional; can be costly.
  - But there are methods: sampling, zero-crossing, "interval arithmetic".
  - Nice aside: you can even use raw data as your plots.
- PA method is fairly robust, as we saw in butterfly picture.

So where are the limitations hiding?

Takes original plots as input.

- The Pixel Array (PA) method requires a plot for each equation.
- The plot of $f(x_1, \ldots, x_n) = 0$ is n-dimensional; can be costly.
  - But there are methods: sampling, zero-crossing, "interval arithmetic".
  - Nice aside: you can even use raw data as your plots.
- PA method is fairly robust, as we saw in butterfly picture.

Introduces false positives.

- False positives can occur arbitrarily far from real solutions.
- Accuracy often degrades when a local derivative approaches 0 or ∞.
- However, refining the mesh gives the correct answer in the limit.

So where are the limitations hiding?

Takes original plots as input.

- The Pixel Array (PA) method requires a plot for each equation.
- The plot of $f(x_1, \ldots, x_n) = 0$ is n-dimensional; can be costly.
  - But there are methods: sampling, zero-crossing, "interval arithmetic".
  - Nice aside: you can even use raw data as your plots.
- PA method is fairly robust, as we saw in butterfly picture.

Introduces false positives.

- False positives can occur arbitrarily far from real solutions.
- Accuracy often degrades when a local derivative approaches 0 or ∞.
- However, refining the mesh gives the correct answer in the limit.

Speed comes from unexposed variables.

- I will discuss exposed/unexposed variables in depth.
- For now, recall butterfly picture: we exposed w and z, but not x or y.
- Comparison with quasi-Newton is not straightforward.

Outline

Equations and wiring diagrams

Consider an arbitrary system of equations having the following form:

$$f_1(\mathbf{t}, u, \mathbf{v}) = 0$$
$$f_2(\mathbf{v}, w, x) = 0$$
$$f_3(u, w, x, y) = 0$$
$$f_4(x, \mathbf{z}) = 0$$

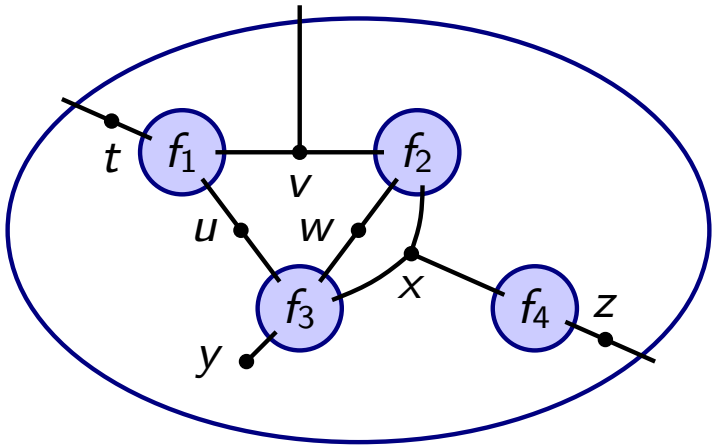
Bold variables are those we want to *expose*; others are latent or *unexposed*.

Equations and wiring diagrams

Consider an arbitrary system of equations having the following form:

$$f_1(\mathbf{t}, u, \mathbf{v}) = 0$$
$$f_2(\mathbf{v}, w, x) = 0$$
$$f_3(u, w, x, y) = 0$$
$$f_4(x, \mathbf{z}) = 0$$

Bold variables are those we want to *expose*; others are latent or *unexposed*.



Said another way, we want $\{(t, v, z) \mid \exists u, w, x, y : f_1 = f_2 = f_3 = f_4 = 0\}$.

Using array multiplication to solve systems

To solve the system for t, v, z, we plot each equation as an array.

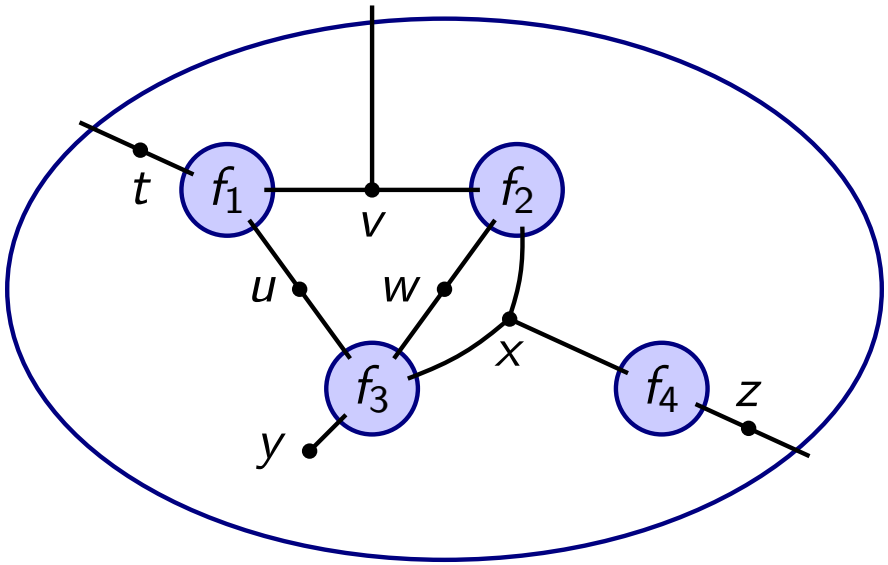
$$f_1(\mathbf{t}, u, \mathbf{v}) = 0$$
$$f_2(\mathbf{v}, w, x) = 0$$
$$f_3(u, w, x, y) = 0$$
$$f_4(x, \mathbf{z}) = 0$$

Using array multiplication to solve systems

To solve the system for t, v, z, we plot each equation as an array.

$$f_1(\mathbf{t}, u, \mathbf{v}) = 0$$
$$f_2(\mathbf{v}, w, x) = 0$$
$$f_3(u, w, x, y) = 0$$
$$f_4(x, \mathbf{z}) = 0$$

PA says: multiply the arrays according to "variable sharing" diagram:



Why and how does it work

There are now two important directions to go from here:

- Explain general array multiplication as dictated by a wiring diagram.
- Demystify relationship between multiplying arrays and solving systems.

Let's start by demystifying why the Pixel Array method works.

- As we'll see, it just comes down to logic: AND, OR, TRUE, FALSE.

To keep things simple, let's restrict our attention to matrix multiplication.

Multiplying Boolean matrices

The matrix multiplication formula works well in any semiring.

$$(MN)_{i,k} = \sum_j M_{i,j} * N_{j,k}.$$

- Roughly, a semiring is a set with $0, 1, +, *$ that act reasonably.
- It's like a ring, but you don't need negatives (e.g. $\mathbb{N}$).

Multiplying Boolean matrices

The matrix multiplication formula works well in any semiring.

$$(MN)_{i,k} = \sum_j M_{i,j} * N_{j,k}.$$

- Roughly, a semiring is a set with $0, 1, +, *$ that act reasonably.
- It's like a ring, but you don't need negatives (e.g. $\mathbb{N}$).

Today we'll focus on the Boolean semiring, $\mathbb{B}$.

- It has two elements $\mathbb{B} = \{0, 1\}$.
- 0 means FALSE, and 1 means TRUE.
- Multiplication is given by boolean AND (denoted $\wedge$).
- Addition is given by boolean OR (denoted $\vee$).
- The only slightly unexpected thing is that $1 + 1 = 1$.

The logic of matrix multiplication

In general, the PA method is to multiply higher-order arrays (tensors).

- But for now, let's suppose A, B are boolean matrices.
- Say that A and B are plots of $f(x, y) = 0$ and $g(y, z) = 0$, resp.
- So $A_{i,j} = 1$ means $f(x, y) = 0$ in corresponding pixel (else $A_{i,j} = 0$).

The logic of matrix multiplication

In general, the PA method is to multiply higher-order arrays (tensors).

- But for now, let's suppose A, B are boolean matrices.
- Say that A and B are plots of $f(x, y) = 0$ and $g(y, z) = 0$, resp.
- So $A_{i,j} = 1$ means $f(x, y) = 0$ in corresponding pixel (else $A_{i,j} = 0$).
- The (i, k)-entry of their product AB is given by the formula:

$$\begin{aligned}(AB)_{i,k} &= \sum_j A_{i,j} * B_{j,k} \\ &= \bigvee_j A_{i,j} \wedge B_{j,k} \\ &= \exists_j (A_{i,j} \wedge B_{j,k})\end{aligned}$$

The logic of matrix multiplication

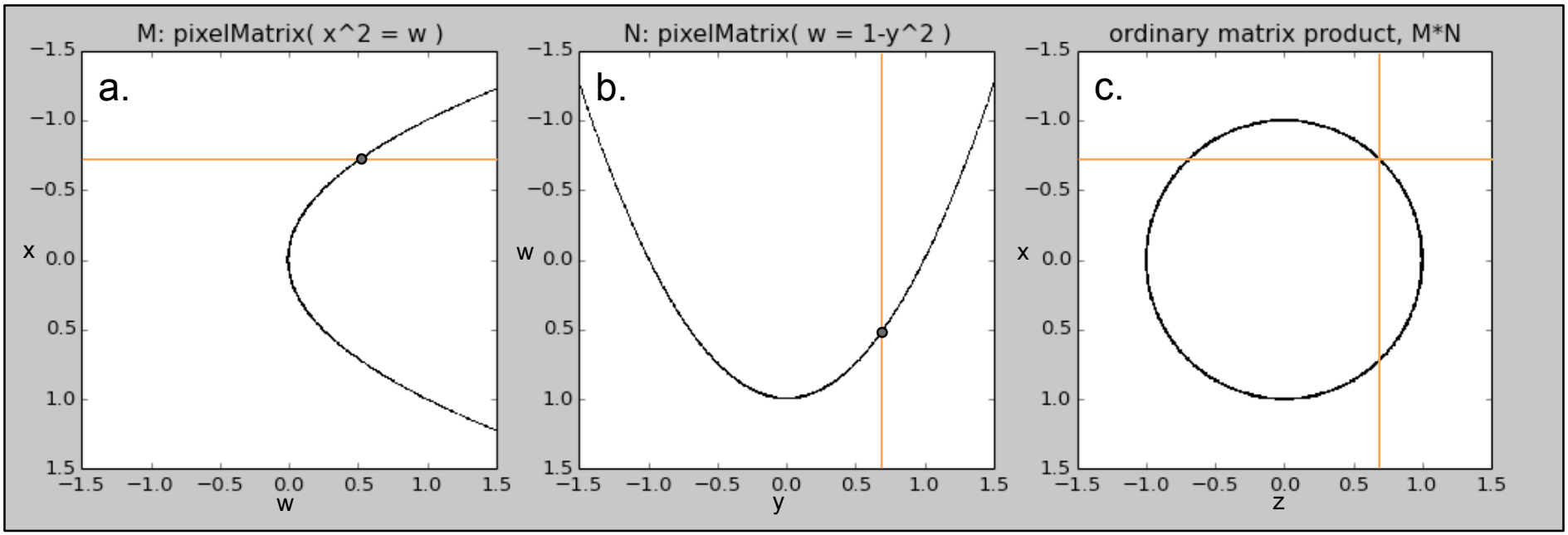
In general, the PA method is to multiply higher-order arrays (tensors).

- But for now, let's suppose A, B are boolean matrices.
- Say that A and B are plots of $f(x, y) = 0$ and $g(y, z) = 0$, resp.
- So $A_{i,j} = 1$ means $f(x, y) = 0$ in corresponding pixel (else $A_{i,j} = 0$).
- The (i, k)-entry of their product AB is given by the formula:

$$\begin{aligned}(AB)_{i,k} &= \sum_j A_{i,j} * B_{j,k} \\ &= \bigvee_j A_{i,j} \wedge B_{j,k} \\ &= \exists_j \left(A_{i,j} \wedge B_{j,k}\right)\end{aligned}$$

- " There exists some y such that $f(x, y) = 0$ and $g(y, z) = 0$. "

Visual example again



Demo

Here is a little demo.[2]

Terminal:

```
> cd /Users/davidspivak/Dropbox\ \(MIT\)/Code/
> notebook
```

[2] Thanks to David Sanders and Andreas Noack for lots of help with Julia.

Multiplying larger-order arrays

When two arrays share a common dimension, they can be multiplied.

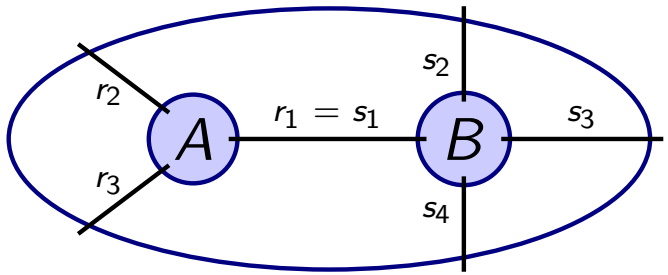
- For example, say that A is $r_1 \times \cdots \times r_m$ and that B is $s_1 \times \cdots \times s_n$.
- If $r_1 = s_1$, the product is an $r_2 \times \cdots \times r_m \times s_2 \times \cdots \times s_n$ array.

Multiplying larger-order arrays

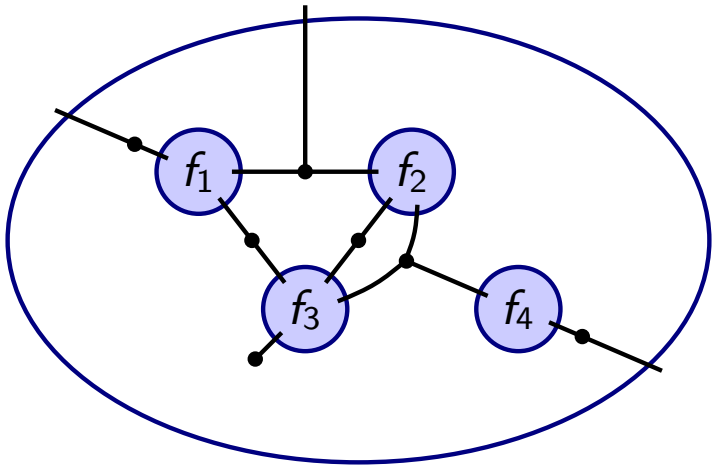
When two arrays share a common dimension, they can be multiplied.

- For example, say that A is $r_1 \times \cdots \times r_m$ and that B is $s_1 \times \cdots \times s_n$.
- If $r_1 = s_1$, the product is an $r_2 \times \cdots \times r_m \times s_2 \times \cdots \times s_n$ array.

We will be drawing these situations using wiring diagrams.



The general array multiplication algorithm



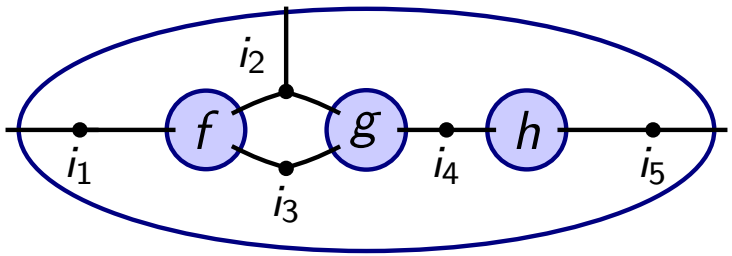
A single formula exists to multiply arrays according to any wiring diagram.

- Basically: iterate over all the links, multiply array entries, and sum.
- But this is very naive: $O(n^{\#\text{links}})$.
  - Modern techniques are much faster, especially given parallel processors.
  - Plots of equations are sparse matrices.
  - Boolean matrices are very special, bit arithmetic.
  - So we can do much better than our current "naive" implementation.

Skip: a naive general array multiplication algorithm

Precondition: Wiring diagram $\Phi\colon P_1, \ldots, P_n \to P'$ and arrays $A_i \in \mathrm{Arr}(P_i)$.

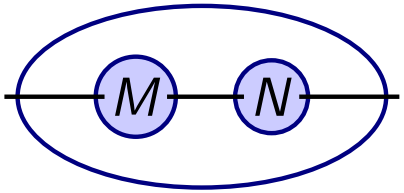
```
1  function Arr(Φ)(A_1, ..., A_n)
2      A' := 0                                          ▷ A' ∈ Arr(P')
3      for i ∈ multi_index(Φ) do                        ▷ One index per link
4          a_i := 1
5          for j ∈ {0, ..., n} do
6              i_j := multi_index^j_Φ(i)                ▷ Indices for j^th pack
7              a_i := a_i * A_j(i_j)
8          i' := multi_index'_Φ(i)                      ▷ Indices for outer pack
9          A'(i') := A'(i') + a_i
10     return A'
```



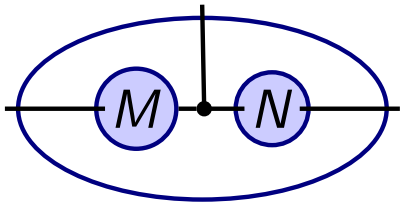
Example wiring diagrams for named operations

The same general array multiplication formula returns famous matrix products for the following wiring diagrams:

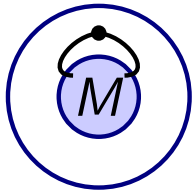
Multiplication: MN



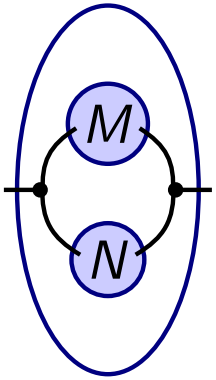
Khatri-Rao: $M \odot N$



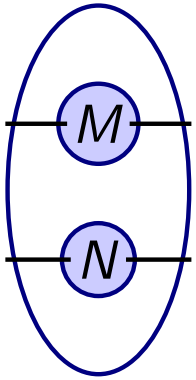
Trace: $\mathrm{Tr}(M)$



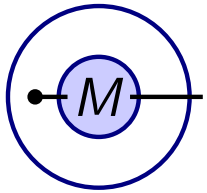
Hadamard: $M \circ N$



Kronecker: $M \otimes N$

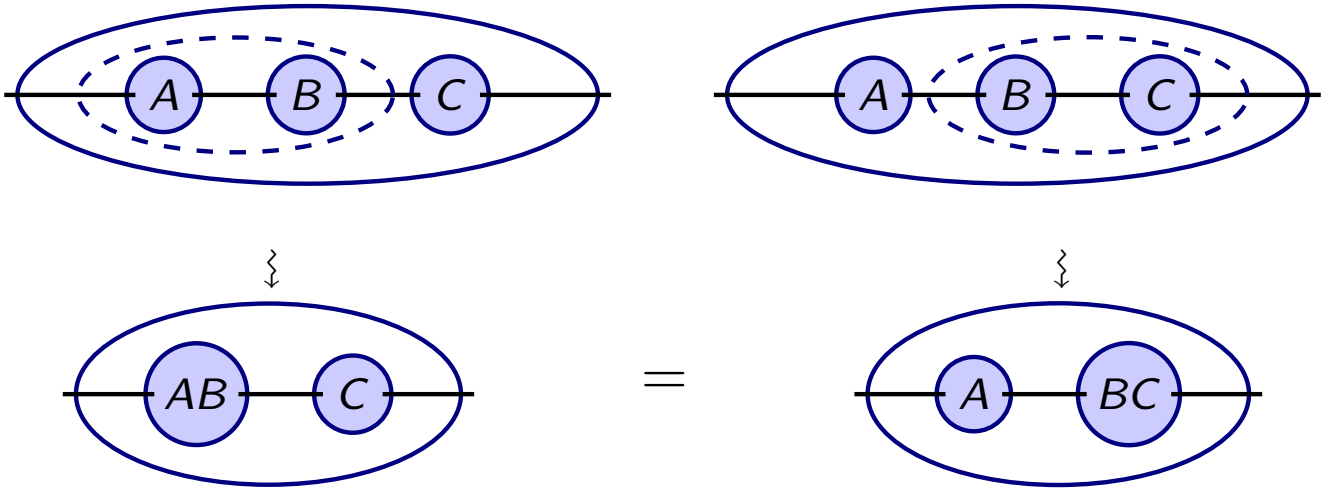


Marginalize: $\sum_i M_{i,j}$



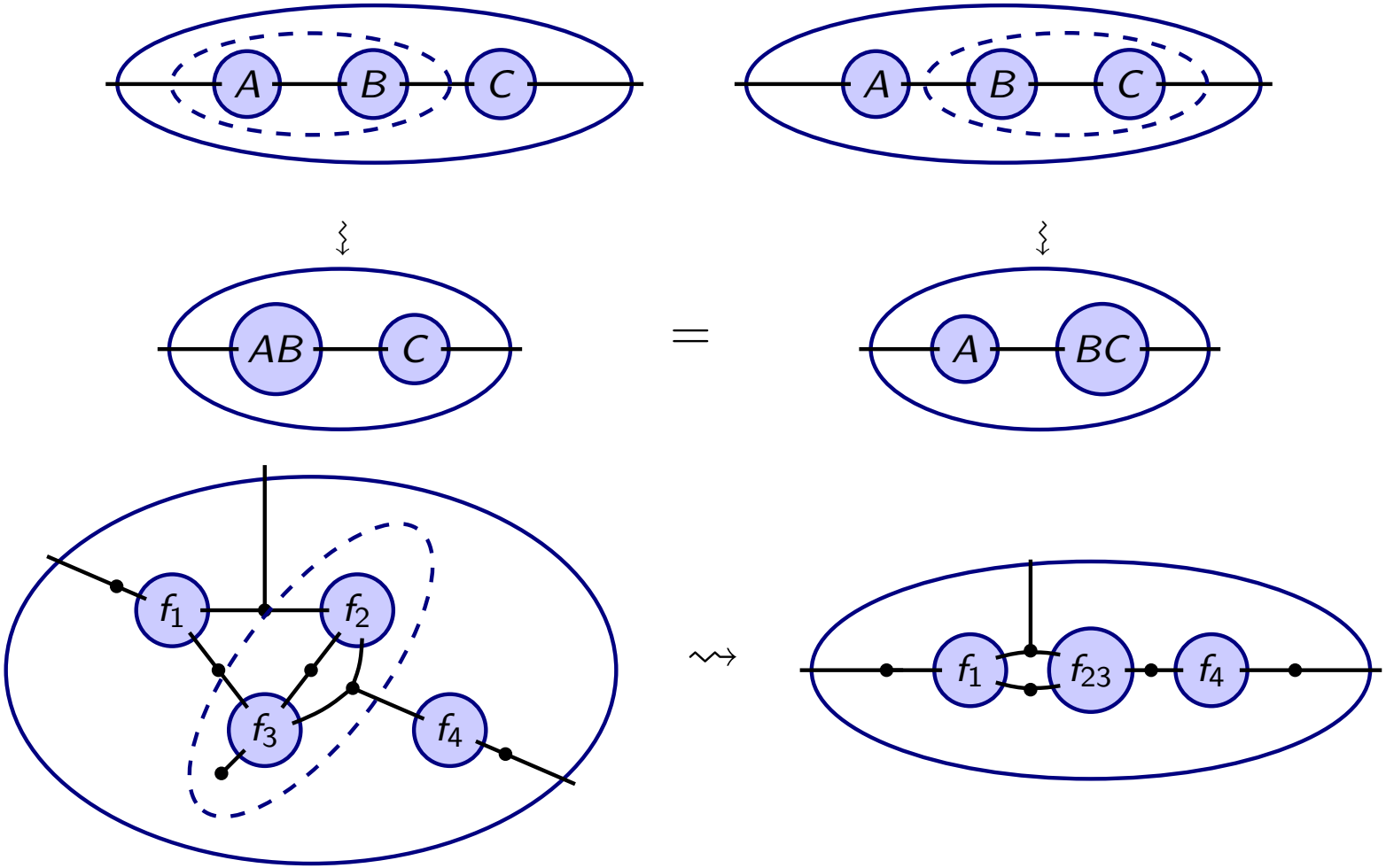
Clustering as associative law

The associative law for matrix multiplication can be seen as clustering.



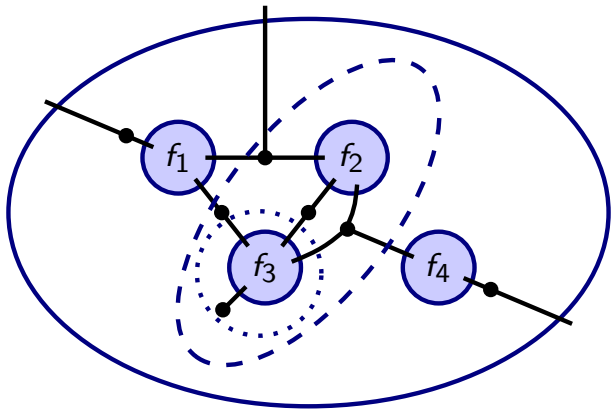
Clustering as associative law

The associative law for matrix multiplication can be seen as clustering.



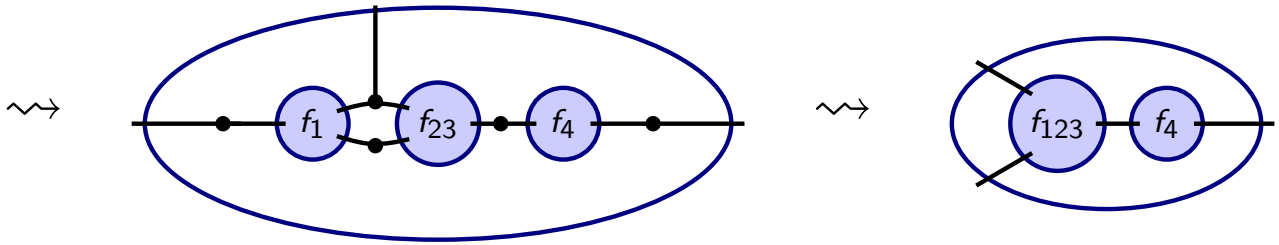
Clustering for speed

Order of array multiplication doesn't affect solution, but does affect speed.



Naive cost for multiplying arrays: $O(n^{\#\text{Links}})$

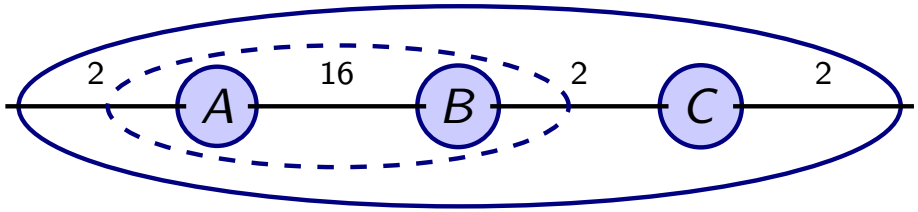
- Even if we can use sparsity, etc. solving big systems is expensive.
- Instead, we should use the associative law: cluster.



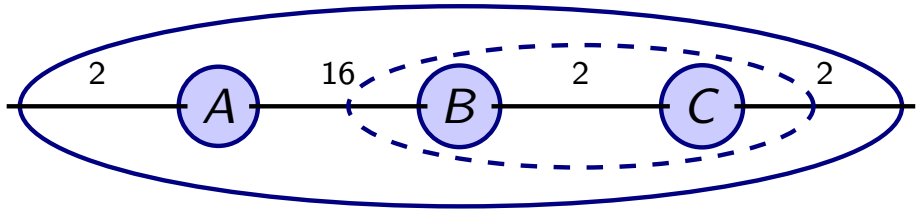
- Naive cost goes from $O(n^7)$ to $O(n^4)$.

Cluster trees

Different cluster strategies lead to different speeds.

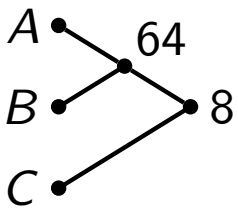


$$\text{cost} = (2 * 16 * 2) + (2 * 2 * 2) = 72$$

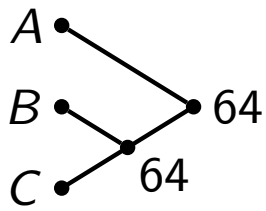


$$\text{cost} = (16 * 2 * 2) + (2 * 16 * 2) = 128$$

Each strategy can be drawn as a "cluster tree":



(Total: 72)



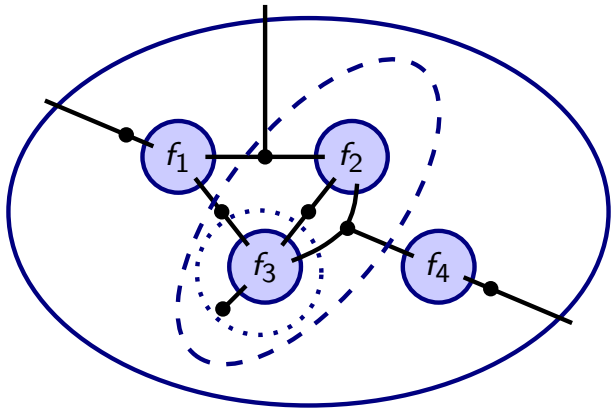
(Total: 128)

Tree tells you how much the computation will cost, using the strategy.

- Given serial processors, the cost is the sum of node values.
- Given parallel processors, the cost is the length of longest path.

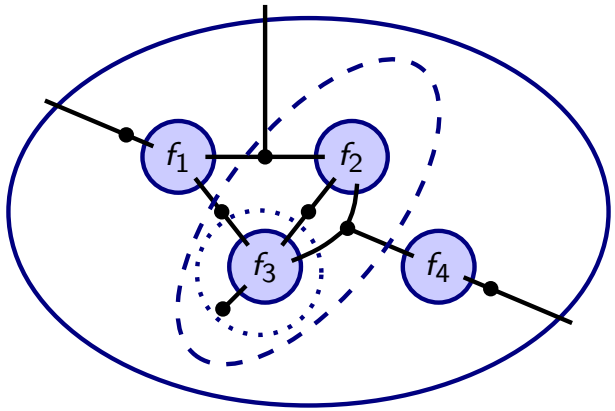
Some clustering algorithms

We need to hierarchically cluster our wiring diagrams to minimize cost.



Some clustering algorithms

We need to hierarchically cluster our wiring diagrams to minimize cost.



Most hypergraph clustering algorithms don't apply "out of the box":

- Our wiring diagrams are "pointed" hypergraphs (outside circle).
- Our clustering should be hierarchical: all the way down.

Some existing hypergraph clustering algorithms:

- Greedy algorithm: look for pairwise cluster of minimize cost.
- Try 1000 random samples and pick the best. Or exhaustively search.
- Try to use min cut algorithms or spectral graph theory.

Where false positives come from

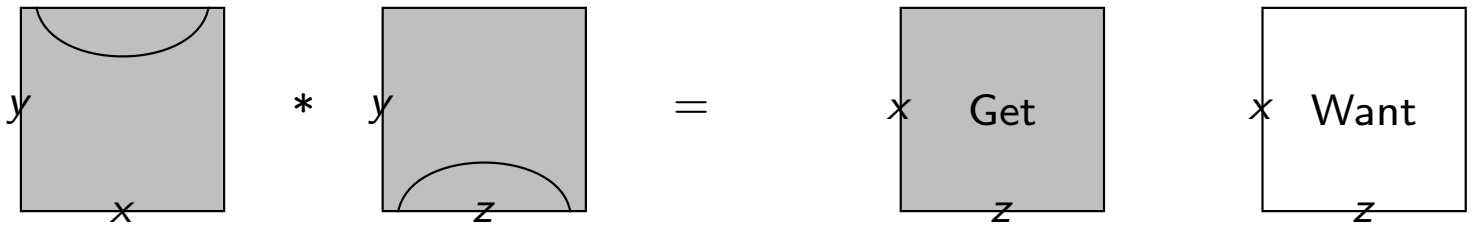
Discretization and matrix multiplication together introduce false positives.

- Discretization corresponds to an equivalence relation, denoted $\sim$.
  - All numbers in the same bin are equivalent.
- This causes discrepancy between what we want and what we get:
  - Want: $\{(x, z) \mid \exists y : f(x, y) = 0 \text{ and } g(y, z) = 0\}$.
  - Get: $\{(x, z) \mid \exists y_1 \sim y_2 : f(x, y_1) = 0 \text{ and } f(y_2, z) = 0\}$
- This provably approaches the correct solution as mesh is refined.

Where false positives come from

Discretization and matrix multiplication together introduce false positives.

- Discretization corresponds to an equivalence relation, denoted $\sim$.
  - All numbers in the same bin are equivalent.
- This causes discrepancy between what we want and what we get:
  - Want: $\{(x, z) \mid \exists y : f(x, y) = 0 \text{ and } g(y, z) = 0\}$.
  - Get: $\{(x, z) \mid \exists y_1 \sim y_2 : f(x, y_1) = 0 \text{ and } f(y_2, z) = 0\}$
- This provably approaches the correct solution as mesh is refined.



- True negatives: "Get" $\geq$ "Want". Further refine the "on" pixels.

Drawing 3D arrays may be better than one thinks

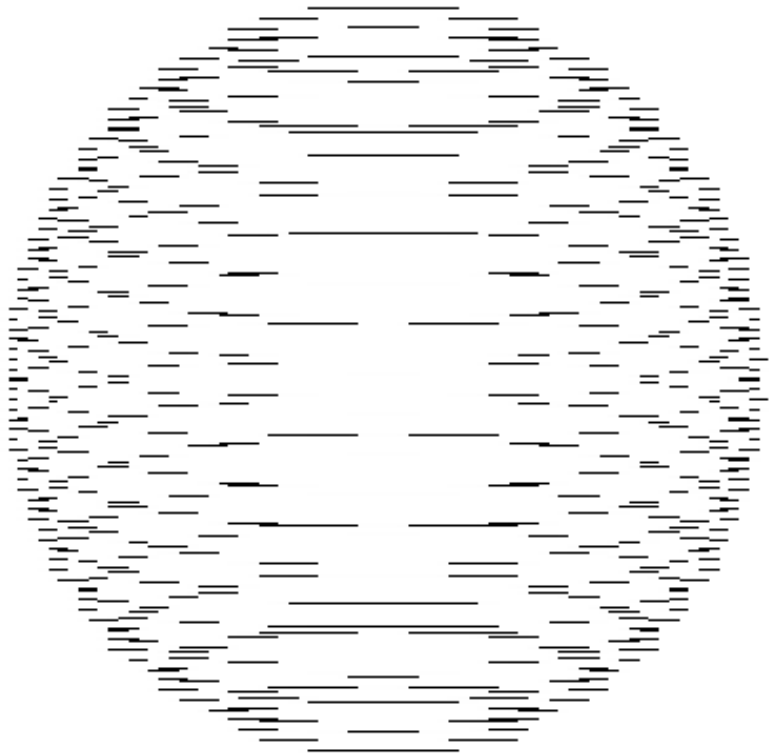
Before moving on to PDEs, here's something I found interesting.

- Start with a sphere $x^2 + y^2 + z^2 = 1$, and plot it as a pixel array.
- The plot is 3D, say $100 \times 100 \times 100$.
- Now reshape it to a 100×10000 array.
- Plot it in a square box. What do you get?

Drawing 3D arrays may be better than one thinks

Before moving on to PDEs, here's something I found interesting.

- Start with a sphere $x^2 + y^2 + z^2 = 1$, and plot it as a pixel array.
- The plot is 3D, say $100 \times 100 \times 100$.
- Now reshape it to a 100×10000 array.
- Plot it in a square box. What do you get?



Outline

Relation to Numerical methods for PDE

In this section I'll discuss how the Pixel Array method may apply to PDE.

- The whole PA idea came from category theory.
- What do pixel arrays, PDEs, and category theory have in common?

Relation to Numerical methods for PDE

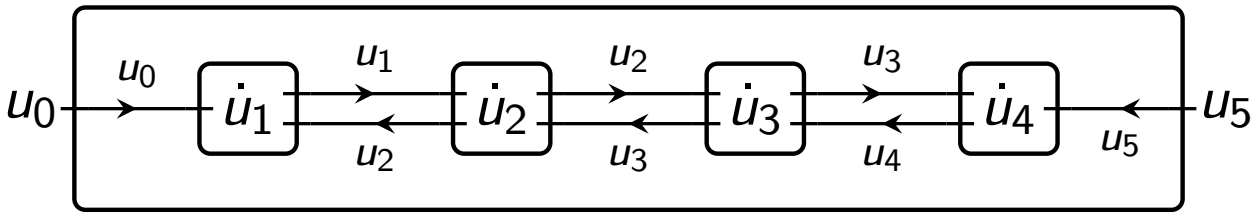
In this section I'll discuss how the Pixel Array method may apply to PDE.

- The whole PA idea came from category theory.
- What do pixel arrays, PDEs, and category theory have in common?
- Compositionality: analyze a system by assembling simple components.

Let's go a bit deeper and look at the heat equation.

The heat equation

Spacial discretization of the 1-d heat equation $u_t = u_{xx}$ over $\Omega = [0, 5]$.



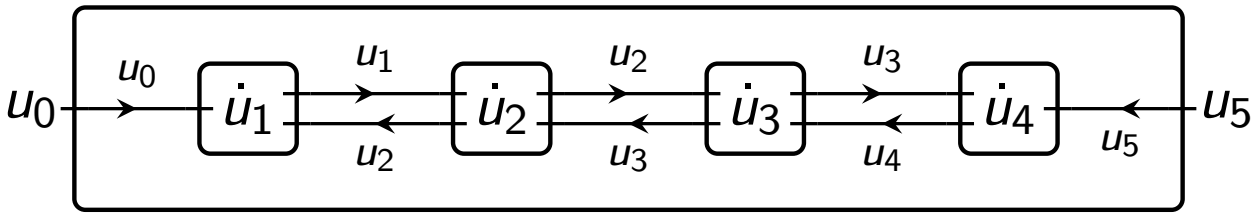
(1)

Discretize x with distance $h = 1$, we use Taylor's thm to obtain four ODEs.

$$\dot{u}_i = u_{i-1} - 2u_i + u_{i+1} \qquad (i = 1, \ldots, 4)$$

The heat equation

Spacial discretization of the 1-d heat equation $u_t = u_{xx}$ over $\Omega = [0, 5]$.



(1)

Discretize x with distance $h = 1$, we use Taylor's thm to obtain four ODEs.

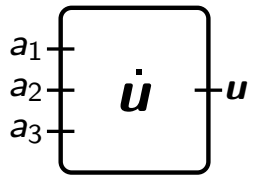
$$\dot{u}_i = u_{i-1} - 2u_i + u_{i+1} \qquad (i = 1, \ldots, 4)$$

Wiring diagram (1) is more general than the heat equation. It represents:

- Four interconnected dynamical systems, each defined by an ODE.
- Each ODE is a function of the neighbors' time-varying states.
- Some ODEs, u_1 and u_4, involve parameters from the "outside world".

Open dynamical systems

Let's look at the boxes in isolation.

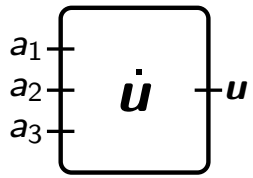


A box of the above shape represents an ODE $\dot{\boldsymbol{u}} = f(\boldsymbol{u}, a_1, \ldots, a_n)$.

- The left-hand ports represent input parameters $a_1(t), \ldots, a_n(t)$.
- The right-hand port represents the state $\boldsymbol{u}(t)$, being exported.
- Called an *open dynamical system* (ODS), because it interacts with the outside world.

Open dynamical systems

Let's look at the boxes in isolation.

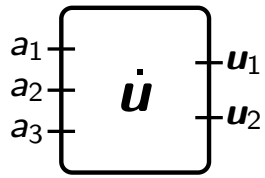


A box of the above shape represents an ODE $\dot{\boldsymbol{u}} = f(\boldsymbol{u}, a_1, \ldots, a_n)$.

- The left-hand ports represent input parameters $a_1(t), \ldots, a_n(t)$.
- The right-hand port represents the state $\boldsymbol{u}(t)$, being exported.
- Called an *open dynamical system* (ODS), because it interacts with the outside world.
- A closed dynamical system would look like this: $\boxed{\dot{u}}$

Open dynamical systems

Let's look at the boxes in isolation.

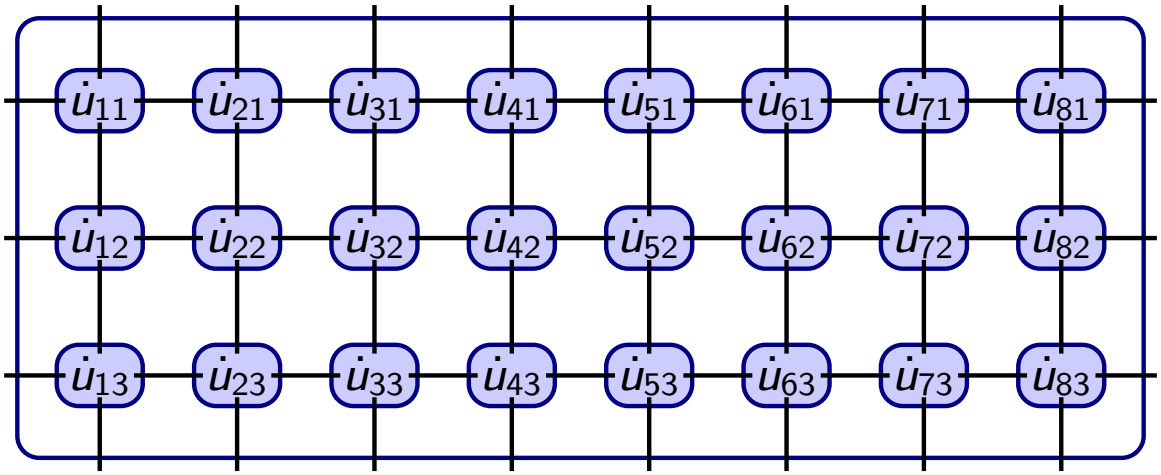


A box of the above shape represents an ODE $\dot{\boldsymbol{u}} = f(\boldsymbol{u}, a_1, \ldots, a_n)$.

- The left-hand ports represent input parameters $a_1(t), \ldots, a_n(t)$.
- The right-hand port represents the state $\boldsymbol{u}(t)$, being exported.
- Called an *open dynamical system* (ODS), because it interacts with the outside world.
- A closed dynamical system would look like this: $\boxed{\dot{u}}$

You could also have multiple outputs, $\boldsymbol{u}_1, \ldots, \boldsymbol{u}_m$ representing functions, possibly projections, of the state: $\boldsymbol{u}_i = \pi_i(\boldsymbol{u})$.

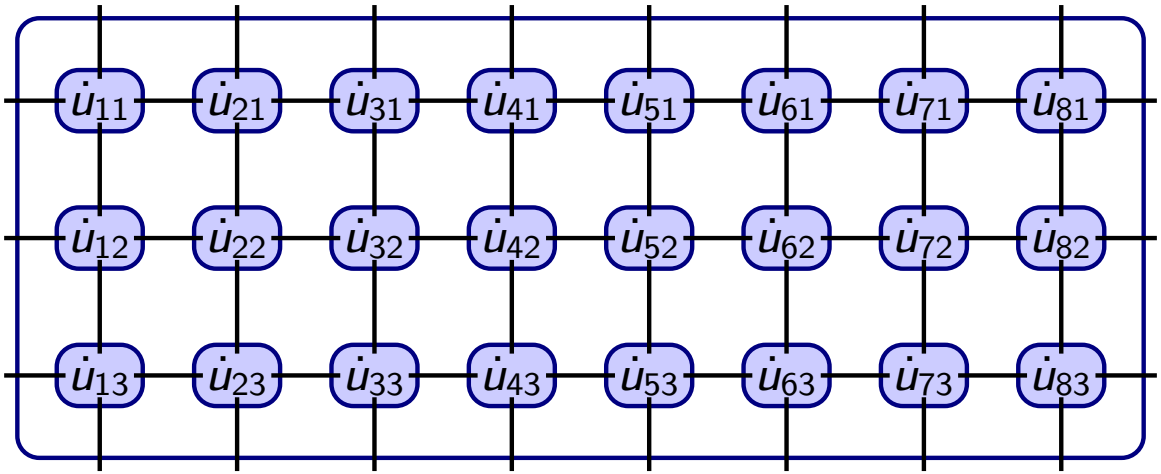
Interconnected open dynamical systems (ODS's)



Setup for the PA method: interconnected 'open dynamical systems'.

- In each box (i,j), there is an ODS $\dot{\boldsymbol{u}}_{ij} = f(\boldsymbol{u}_{ij}, \boldsymbol{u}_{\text{neighbors}})$.
  - These open dynamical systems are sharing variables with neighbors.
  - Together these form a more complicated ODS on the outer box.

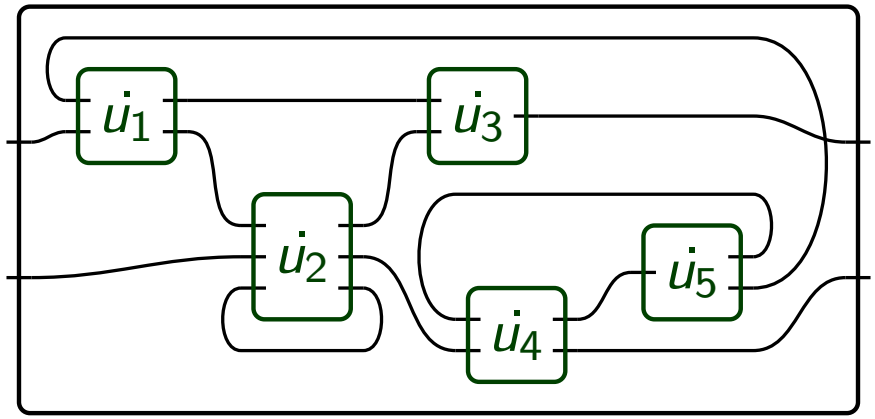
Interconnected open dynamical systems (ODS's)



Setup for the PA method: interconnected 'open dynamical systems'.

- In each box (i,j), there is an ODS $\dot{\boldsymbol{u}}_{ij} = f(\boldsymbol{u}_{ij}, \boldsymbol{u}_{\text{neighbors}})$.
  - These open dynamical systems are sharing variables with neighbors.
  - Together these form a more complicated ODS on the outer box.
- Examples of interconnected ODS's:
  - Finite differences, in any dimension, discretized in space.

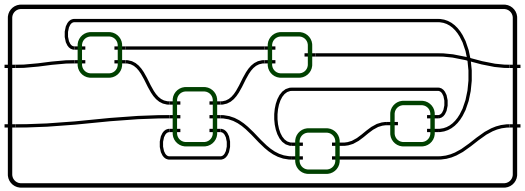
Interconnected open dynamical systems (ODS's)



Setup for the PA method: interconnected 'open dynamical systems'.

- In each box (i, j), there is an ODS $\dot{\boldsymbol{u}}_{ij} = f(\boldsymbol{u}_{ij}, \boldsymbol{u}_{\text{neighbors}})$.
  - These open dynamical systems are sharing variables with neighbors.
  - Together these form a more complicated ODS on the outer box.
- Examples of interconnected ODS's:
  - Finite differences, in any dimension, discretized in space.
  - Differential equation on a network (compartment model).
  - Systems of systems (smart grid, National Airspace System).

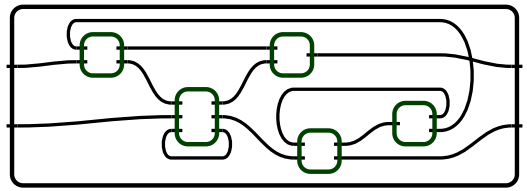
Pixel Array analysis



The Pixel Array method lets you assemble local results.

- First choose some sort of data to plot for each ODS.
  - Most obvious: plot the steady states $\boldsymbol{u} = 0$.
  - But you could also try plotting solutions over time $[0, \tau]$.
- The plot is an array, indexed by the input and output wires.
  - For steady state data: index by constant input/output functions.
  - For $[0, \tau]$ solutions: index by classifying $[0, \tau]$ input/output functions.

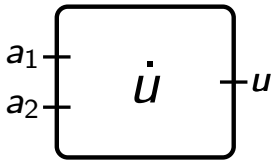
Pixel Array analysis

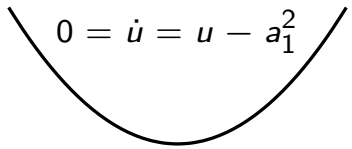


The Pixel Array method lets you assemble local results.

- First choose some sort of data to plot for each ODS.
  - Most obvious: plot the steady states $\boldsymbol{u} = 0$.
  - But you could also try plotting solutions over time $[0, \tau]$.
- The plot is an array, indexed by the input and output wires.
  - For steady state data: index by constant input/output functions.
  - For $[0, \tau]$ solutions: index by classifying $[0, \tau]$ input/output functions.
- Then put the plots together using the Pixel Array method.
- The result is a plot of the chosen sort for the whole system (PDE).

Plotting



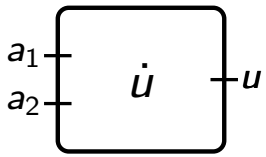


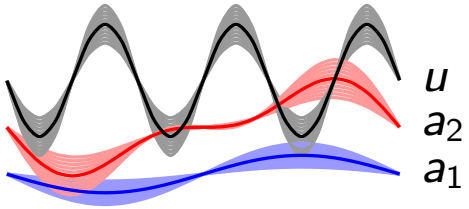
1	0	0	0	1
1	0	0	0	1
0	1	0	1	0
0	1	1	1	0
0	1	1	1	0

In the box we have an open dynamical system $\dot{u} = f(u, a_1, \ldots, a_n)$.

- A plot for this ODS will be an order-$(n+1)$ array.
- For example, you can plot steady-state solutions.
  - Set $\dot{u} = 0$ and get an equation $f(u, a_i) = 0$.
  - This is just the bifurcation diagram of the dynamical system.
  - Plot it as a pixel array: PA method combines bifurcation diagrams.

Plotting





turn pixel on if solution exists for (a_1, a_2, u) in this range

In the box we have an open dynamical system $\dot{u} = f(u, a_1, \ldots, a_n)$.

- A plot for this ODS will be an order-$(n+1)$ array.
- For example, you can plot steady-state solutions.
  - Set $\dot{u} = 0$ and get an equation $f(u, a_i) = 0$.
  - This is just the bifurcation diagram of the dynamical system.
  - Plot it as a pixel array: PA method combines bifurcation diagrams.
- More exotic: plot bounded-time solutions, e.g. on $[0, \tau]$.
  - Bin the Fourier coefficients for each a_i and u: a bin is a range.
  - A pixel corresponds to a class of periodic functions for each a_i and u.
  - Turn the pixel on if there is a solution for that sequence of classes.

How to obtain these plots?

How do we obtain these plots? "Good luck!"

How to obtain these plots?

How do we obtain these plots? Develop numerical techniques!

The Pixel Array method says: "plotting is a way of linearizing".

- Nonlinear solvers traditionally linearize the functions using Jacobians.
- The PA method goes around this, "linearizing" in a different way.
- For PA the challenge is plotting, rather than approximating Jacobians.

How to obtain these plots?

How do we obtain these plots? Develop numerical techniques!

The Pixel Array method says: "plotting is a way of linearizing".

- Nonlinear solvers traditionally linearize the functions using Jacobians.
- The PA method goes around this, "linearizing" in a different way.
- For PA the challenge is plotting, rather than approximating Jacobians.

Some methods for plotting $f(\boldsymbol{x}) = 0$:

- Sample points at corner of each cell; turn pixel on if $\operatorname{sign}(f)$ changes.
- Sample one point in each cell and put $2^{-|f(\boldsymbol{x})|}$ (not Boolean).
- Use "Interval arithmetic" (e.g. $[1,2] * [2,3] = [2,6]$).
- If your function is Lipschitz, you can obtain a perfect plot.
- Use your favorite n-dimensional contour plotter (with 0-contour).

How to obtain these plots?

How do we obtain these plots? Develop numerical techniques!

The Pixel Array method says: "plotting is a way of linearizing".

- Nonlinear solvers traditionally linearize the functions using Jacobians.
- The PA method goes around this, "linearizing" in a different way.
- For PA the challenge is plotting, rather than approximating Jacobians.

Some methods for plotting $f(\boldsymbol{x}) = 0$:

- Sample points at corner of each cell; turn pixel on if $\operatorname{sign}(f)$ changes.
- Sample one point in each cell and put $2^{-|f(\boldsymbol{x})|}$ (not Boolean).
- Use "Interval arithmetic" (e.g. $[1,2] * [2,3] = [2,6]$).
- If your function is Lipschitz, you can obtain a perfect plot.
- Use your favorite n-dimensional contour plotter (with 0-contour).

Once you have the plots, the PA method lets you combine them.

Outline

Apples and oranges

The inputs and outputs of the PA method are different than Newton's.

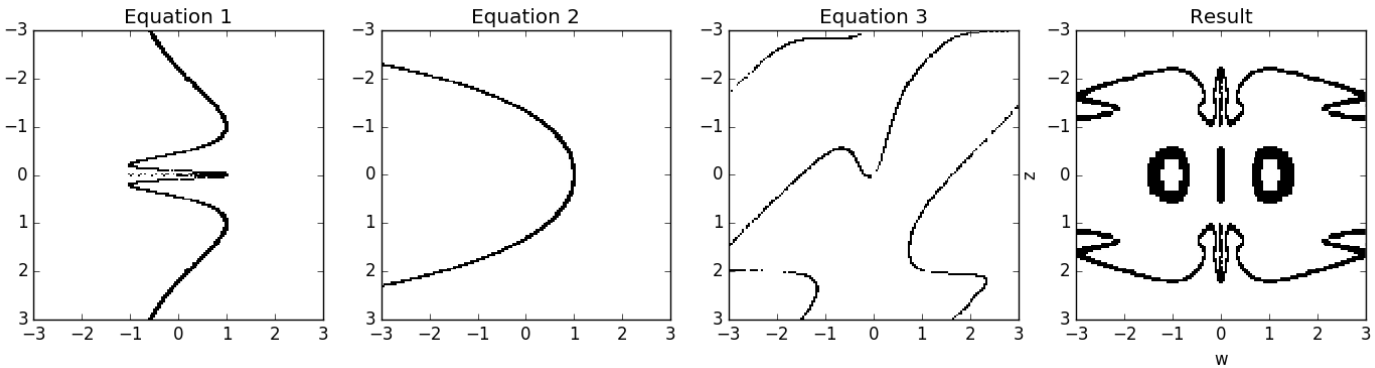
	Pixel Array method	**Newton's method**
Inputs:	a range for each variable	a good initial guess
Outputs:	all solutions for some variables	one solution in all variables.

Apples and oranges

The inputs and outputs of the PA method are different than Newton's.

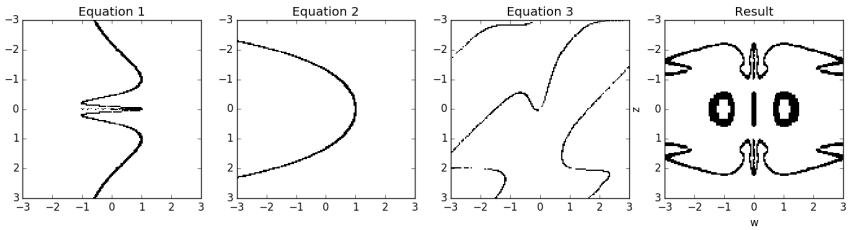
	Pixel Array method	**Newton's method**
Inputs:	a range for each variable	a good initial guess
Outputs:	all solutions for some variables	one solution in all variables.

Our speed test assumes you want to produce "all solutions" in range.



How might we use Newton to find all solutions??

How to compare speed?



Our comparison with Newton assumes you want "all solutions" in range.

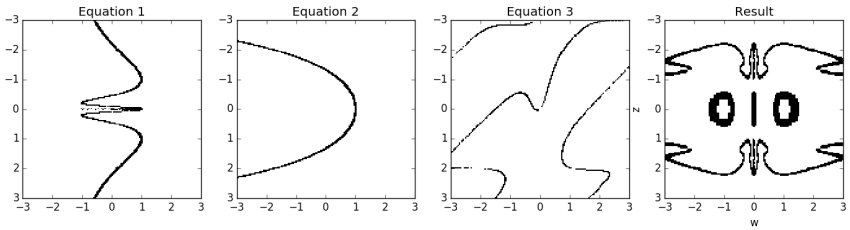
- We iterated over all boxes in the grid, each as an initial guess.

$$\cos\left(\ln(z^2 + 10^{-3}x)\right) - x + 10^{-5}z^{-1} = 0 \qquad \text{(Equation 1)}$$

$$\cosh(w + 10^{-3}y) + y + 10^{-4}w = 2 \qquad \text{(Equation 2)}$$

$$\tan(x + y)(x - 2)^{-1}(x + 3)^{-1}y^{-2} = 1 \qquad \text{(Equation 3)}$$

How to compare speed?



Our comparison with Newton assumes you want "all solutions" in range.

- We iterated over all boxes in the grid, each as an initial guess.

$$\cos\left(\ln(z^2 + 10^{-3}x)\right) - x + 10^{-5}z^{-1} = 0 \qquad \text{(Equation 1)}$$

$$\cosh(w + 10^{-3}y) + y + 10^{-4}w = 2 \qquad \text{(Equation 2)}$$

$$\tan(x + y)(x - 2)^{-1}(x + 3)^{-1}y^{-2} = 1 \qquad \text{(Equation 3)}$$

- Is this unfair? (Ruben compares to QR vs. inverse iteration.)

With that assumption, for the above case, PA was over 7200x faster.

- PA: 1.5 seconds; Newton: we stopped Julia's NLsolve after 3 hours.

Determining accuracy

The PA method can introduce false positives, but no false negatives.

- The plotting process can easily introduce false negatives.
- And the Pixel Array method will of course propagate them.
- But assuming plots have true negatives, so do results.

[3] Probably either the ℓ_1- or the ℓ_∞- norm is best suited for defining distance here.

Determining accuracy

The PA method can introduce false positives, but no false negatives.

- The plotting process can easily introduce false negatives.
- And the Pixel Array method will of course propagate them.
- But assuming plots have true negatives, so do results.

We thus define the *error* to be:[3]

$$\text{error} := \max_{p \in \text{calculated solution}} \left(\min_{q \in \text{real solution}} \text{dist}(p, q) \right)$$

[3] Probably either the ℓ_1- or the ℓ_∞- norm is best suited for defining distance here.

Determining accuracy

The PA method can introduce false positives, but no false negatives.

- The plotting process can easily introduce false negatives.
- And the Pixel Array method will of course propagate them.
- But assuming plots have true negatives, so do results.

We thus define the *error* to be:[3]

$$\text{error} := \max_{p \in \text{calculated solution}} \left(\min_{q \in \text{real solution}} \text{dist}(p, q) \right)$$

Fact: for a given resolution, the error is "unbounded":

[3] Probably either the ℓ_1- or the ℓ_∞- norm is best suited for defining distance here.

Determining accuracy

The PA method can introduce false positives, but no false negatives.

- The plotting process can easily introduce false negatives.
- And the Pixel Array method will of course propagate them.
- But assuming plots have true negatives, so do results.

We thus define the *error* to be:[3]

$$\text{error} := \max_{p \in \text{calculated solution}} \left(\min_{q \in \text{real solution}} \text{dist}(p, q) \right)$$

Fact: for a given resolution, the error is "unbounded":

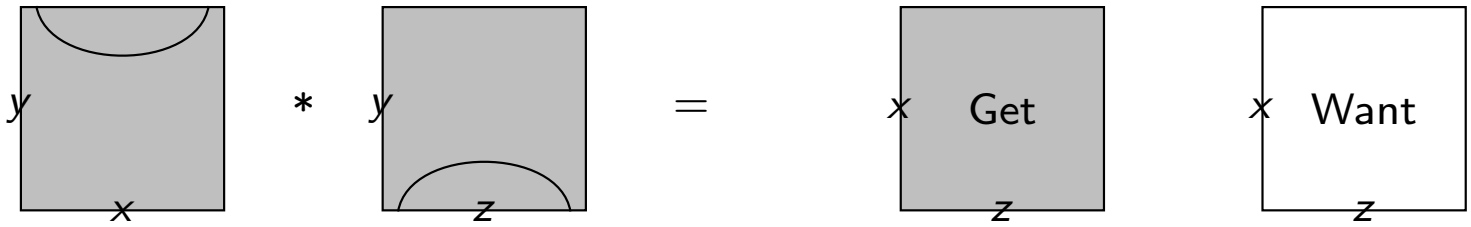
- For example, take $f(x, y) = y$, $g(y, z) = y - \frac{\text{step size}}{2}$.
- PA returns the "all 1's" pixel matrix; correct solution is "all 0's"

Again, refinement solves this, but at a cost.

[3] Probably either the ℓ_1- or the ℓ_∞- norm is best suited for defining distance here.

The source of error

When multiplying arrays, one should think of lining up matching indices.

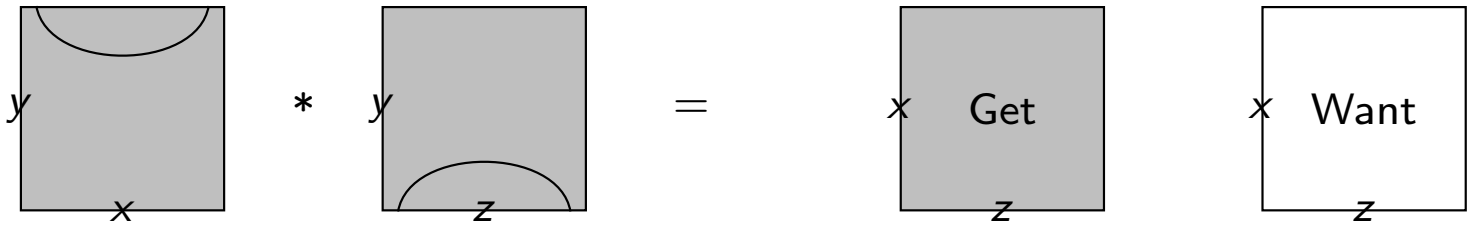


Note that both of these have small y-coverage.

- This is related to having a small value of $\frac{\partial y}{\partial x}$.
- Interestingly, a similar condition comes up in Newton's method.

The source of error

When multiplying arrays, one should think of lining up matching indices.



Note that both of these have small y-coverage.

- This is related to having a small value of $\frac{\partial y}{\partial x}$.
- Interestingly, a similar condition comes up in Newton's method.

But errors (false positives) also propagate as we combine arrays.

- A bound for the error in solving the whole system would be useful.
- I've got UROPs on the case, but we are looking for a good idea.

Clustering cost function

Suppose given a cost function for multiplying two arrays.

- For example, my naive cost function was $r^{\#\text{variables}}$.
- But if matrices are sufficiently sparse, this goes way down.

Then for any cluster tree, you get the parallel and serial costs.

- Finding a good one can save you a lot of time in multiplying arrays.
- But finding a good one is probably NP.

Clustering cost function

Suppose given a cost function for multiplying two arrays.

- For example, my naive cost function was $r^{\#\text{variables}}$.
- But if matrices are sufficiently sparse, this goes way down.

Then for any cluster tree, you get the parallel and serial costs.

- Finding a good one can save you a lot of time in multiplying arrays.
- But finding a good one is probably NP.

It remains to find a good algorithm for minimizing this cost function.

Outline

Summary

In this talk, I discussed the Pixel Array method for solving systems.

- Such systems come up in PDE when discretizing in space.
- For example, we can plot a pixel array for steady states of each ODE.
- Any networked system of ODS's can be analyzed using pixel arrays.

Summary

In this talk, I discussed the Pixel Array method for solving systems.

- Such systems come up in PDE when discretizing in space.
- For example, we can plot a pixel array for steady states of each ODE.
- Any networked system of ODS's can be analyzed using pixel arrays.

I showed how the PA method works: it's pretty elementary.

- We have an array for each equation.
- We know which variables are shared (captured by a wiring diagram).
- We then cluster the arrays to minimize the cost of multiplying.
- Multiplying the arrays gives the simultaneous solution set.

The PA method is quite fast for finding all solutions.

Summary

In this talk, I discussed the Pixel Array method for solving systems.

- Such systems come up in PDE when discretizing in space.
- For example, we can plot a pixel array for steady states of each ODE.
- Any networked system of ODS's can be analyzed using pixel arrays.

I showed how the PA method works: it's pretty elementary.

- We have an array for each equation.
- We know which variables are shared (captured by a wiring diagram).
- We then cluster the arrays to minimize the cost of multiplying.
- Multiplying the arrays gives the simultaneous solution set.

The PA method is quite fast for finding all solutions.

Thanks for inviting me to speak!