The pixel array method for solving non-linear systems of equations

David I. Spivak

Joint work with: Magdalen Dobson, Sapna Kumari, and Lawrence Wu

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

Presented on 2017/01/30
at the Mathematics IAP Lecture Series

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 from the usual Newton's method.
  - With Newton's method, you get very fine accuracy on one solution.
  - The new method approximates *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 from the usual Newton's method.
  - With Newton's method, you get very fine accuracy on one solution.
  - The new method approximates *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 we need is matrix multiplication.
- I'll also show how this method can help you solve PDEs.

Background on matrix multiplication (the how)

Before we begin, two slides about matrix multiplication: how and why.

- A matrix is just a 2-dimensional array of numbers.
  - "Numbers" here means anything you can multiply and add together.

Background on matrix multiplication (the how)

Before we begin, two slides about matrix multiplication: how and why.

- A matrix is just a 2-dimensional array of numbers.
  - "Numbers" here means anything you can multiply and add together.
  - An $(m \times n)$-matrix M has m rows and n columns.
  - For any i, j, where $i \in 1 \ldots m$ and $j \in 1 \ldots n$, there is a number $M_{i,j}$.

$$M = \begin{pmatrix} 1 & 3 & 0 & 9 \\ 3 & 5 & 6 & 1 \end{pmatrix}$$

  - This M has $m = 2$ rows, $n = 4$ cols, and the $(2,2)$-entry is $M_{2,2} = 5$.

Background on matrix multiplication (the how)

Before we begin, two slides about matrix multiplication: how and why.

- A matrix is just a 2-dimensional array of numbers.
  - "Numbers" here means anything you can multiply and add together.
  - An $(m \times n)$-matrix M has m rows and n columns.
  - For any i, j, where $i \in 1 \ldots m$ and $j \in 1 \ldots n$, there is a number $M_{i,j}$.

$$M = \begin{pmatrix} 1 & 3 & 0 & 9 \\ 3 & 5 & 6 & 1 \end{pmatrix}$$

  - This M has $m = 2$ rows, $n = 4$ cols, and the $(2,2)$-entry is $M_{2,2} = 5$.
- Two matrices can be multiplied if their sizes match appropriately.
  - Suppose M and N are $(m \times n)$ and $(n' \times p)$, respectively.
  - They can be multiplied iff $n = n'$; then their product MN is $(m \times p)$.
  - The (i,k)-entry of MN is given by the following formula:

Background on matrix multiplication (the how)

Before we begin, two slides about matrix multiplication: how and why.

- A matrix is just a 2-dimensional array of numbers.
  - "Numbers" here means anything you can multiply and add together.
  - An $(m \times n)$-matrix M has m rows and n columns.
  - For any i, j, where $i \in 1 \dots m$ and $j \in 1 \dots n$, there is a number $M_{i,j}$.

$$M = \begin{pmatrix} 1 & 3 & 0 & 9 \\ 3 & 5 & 6 & 1 \end{pmatrix}$$

  - This M has $m = 2$ rows, $n = 4$ cols, and the $(2,2)$-entry is $M_{2,2} = 5$.
- Two matrices can be multiplied if their sizes match appropriately.
  - Suppose M and N are $(m \times n)$ and $(n' \times p)$, respectively.
  - They can be multiplied iff $n = n'$; then their product MN is $(m \times p)$.
  - The (i,k)-entry of MN is given by the following formula:

$$(MN)_{i,k} = \sum_{j=1}^{n} M_{i,j} * N_{j,k}$$

Uses of matrices and matrix multiplication (the why)

- Why are matrices so important in science and engineering?

Uses of matrices and matrix multiplication (the why)

- Why are matrices so important in science and engineering?
- Because they efficiently represent, or model, many important things.
  1. Linear transformations: rotations, reflections, scalings of n-dim'l space.
  2. Finite graphs: systems of edges connecting n-vertices in arbitrary ways.
  3. Markov processes: probablilistic systems with n possible states.

Uses of matrices and matrix multiplication (the why)

- Why are matrices so important in science and engineering?
- Because they efficiently represent, or model, many important things.
  1. Linear transformations: rotations, reflections, scalings of n-dim'l space.
  2. Finite graphs: systems of edges connecting n-vertices in arbitrary ways.
  3. Markov processes: probablilistic systems with n possible states.
- What does (square) matrix mult. MN or M^n do in these cases?
  1. Compose linear transformations: do M then N; or do M, n times.
  2. Move along M-edges then N-edges; or along paths of length n in M.
  3. Chance of two-player M, N transition; chance of n-step transition in M.

Uses of matrices and matrix multiplication (the why)

- Why are matrices so important in science and engineering?
- Because they efficiently represent, or model, many important things.
  1. Linear transformations: rotations, reflections, scalings of n-dim'l space.
  2. Finite graphs: systems of edges connecting n-vertices in arbitrary ways.
  3. Markov processes: probablilistic systems with n possible states.
- What does (square) matrix mult. MN or M^n do in these cases?
  1. Compose linear transformations: do M then N; or do M, n times.
  2. Move along M-edges then N-edges; or along paths of length n in M.
  3. Chance of two-player M, N transition; chance of n-step transition in M.

In this talk, we will completely different use of matrix multiplication.

- We'll use them to solve arbitrary systems of equations.

Uses of matrices and matrix multiplication (the why)

- Why are matrices so important in science and engineering?
- Because they efficiently represent, or model, many important things.
  1. Linear transformations: rotations, reflections, scalings of n-dim'l space.
  2. Finite graphs: systems of edges connecting n-vertices in arbitrary ways.
  3. Markov processes: probablilistic systems with n possible states.
- What does (square) matrix mult. MN or M^n do in these cases?
  1. Compose linear transformations: do M then N; or do M, n times.
  2. Move along M-edges then N-edges; or along paths of length n in M.
  3. Chance of two-player M, N transition; chance of n-step transition in M.

In this talk, we will completely different use of matrix multiplication.

- We'll use them to solve arbitrary systems of equations.
- Two things to mention now:
  - The "matrices" will actually be higher-dim'l arrays (a.k.a. *tensors*).
  - The "numbers" in our arrays will be 0's and 1's (a.k.a. *booleans*).

A simple example

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.

A simple example

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...

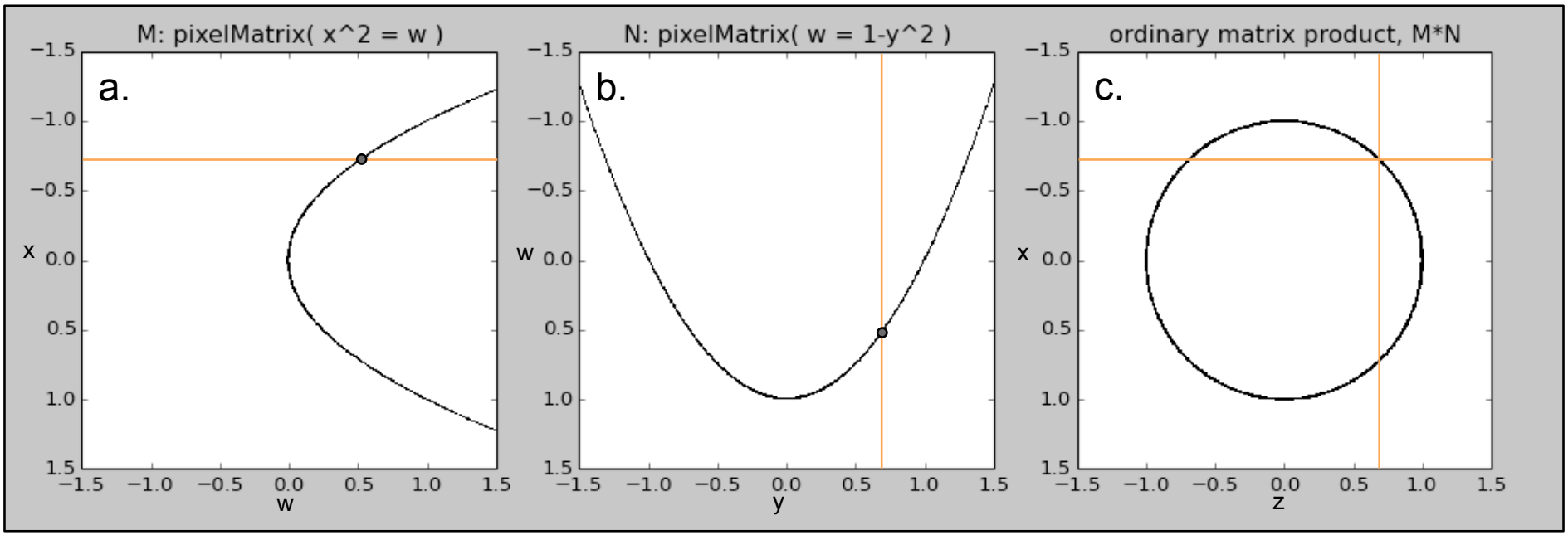
A simple example

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

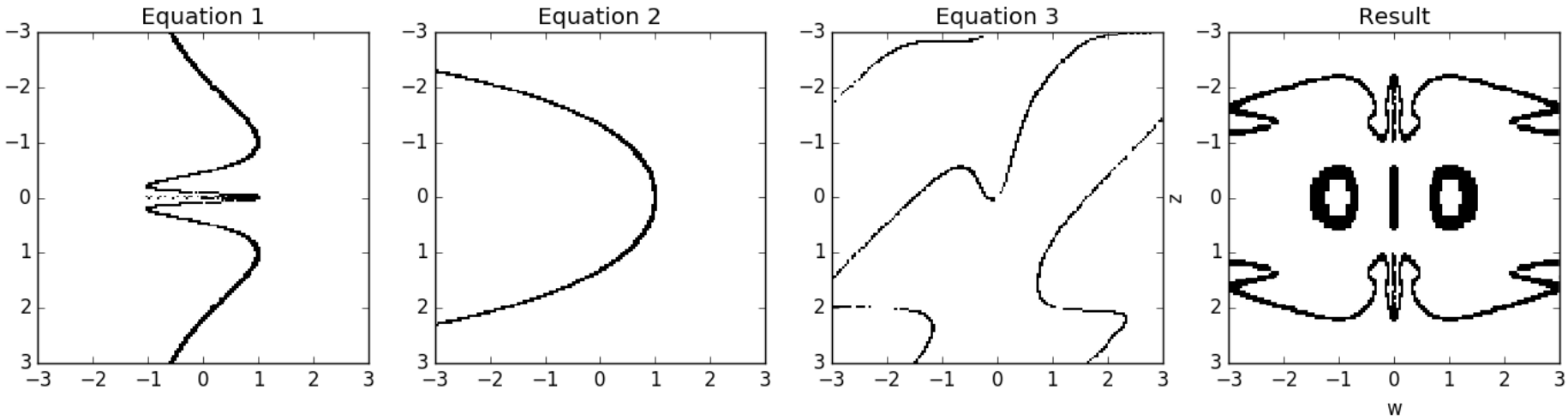
The following eq's are not differentiable, nor even defined everywhere.

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

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

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

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

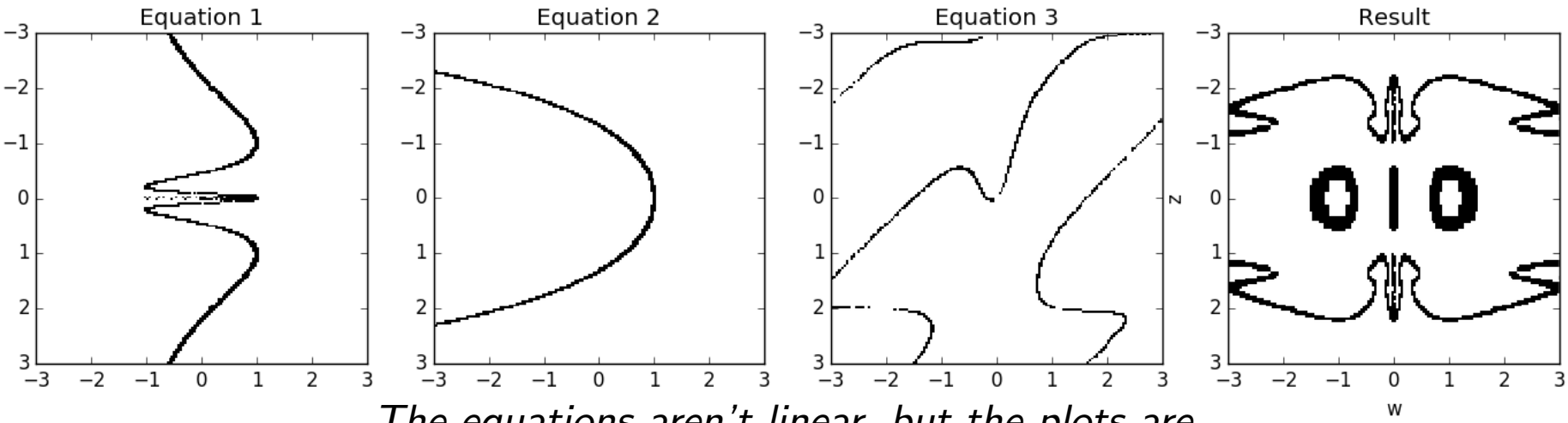
A popular refrain in mathematics:

- 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.

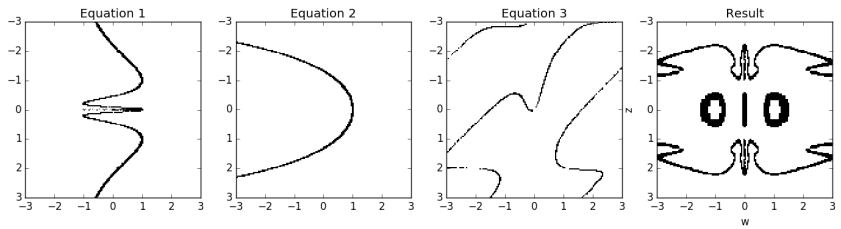
The Pixel Array slogan: *Plotting is a way of linearizing*.

- 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?

The Pixel Array method actually solves $|f(x)| < \epsilon$, not $f(x) = 0$.

- There is a relationship between ϵ and the pixel resolution.
- Refining the mesh always gives the correct (=0) answer in the limit.

So where are the limitations hiding?

The Pixel Array method actually solves $|f(x)| < \epsilon$, not $f(x) = 0$.

- There is a relationship between ϵ and the pixel resolution.
- Refining the mesh always gives the correct (=0) 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 methods is not straightforward.

Outline

Equations and wiring diagrams

Consider an arbitrary system of equations having the following form:

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

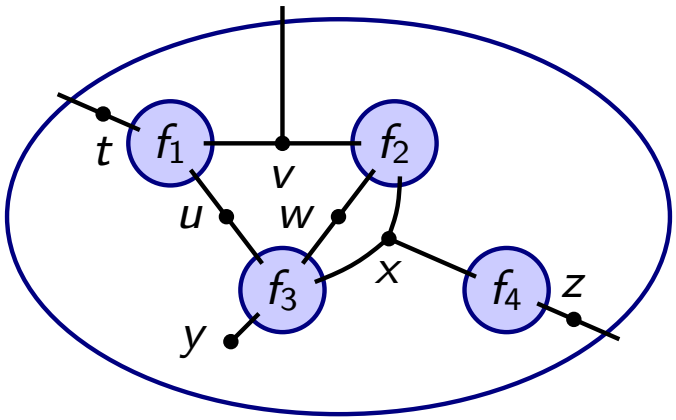
Bold variables are those we want to *expose*; others are latent or *unexposed*.

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.

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

- I will discuss what plotting means and how to do it a bit later.

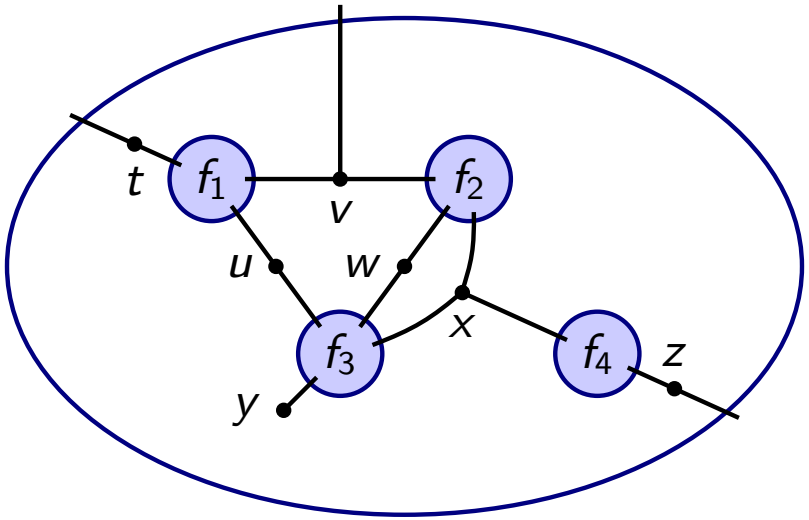
Using array multiplication to solve systems

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

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

- I will discuss what plotting means and how to do it a bit later.

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



Why and how does it work

There are now three important directions to go from here:

- Say what we mean by plotting an equation.
- Explain general array multiplication as dictated by a wiring diagram.
- Demystify relationship between multiplying arrays and solving systems.

Why and how does it work

There are now three important directions to go from here:

- Say what we mean by plotting an equation.
- 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, we'll need 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, we'll need 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:

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

The logic of matrix multiplication

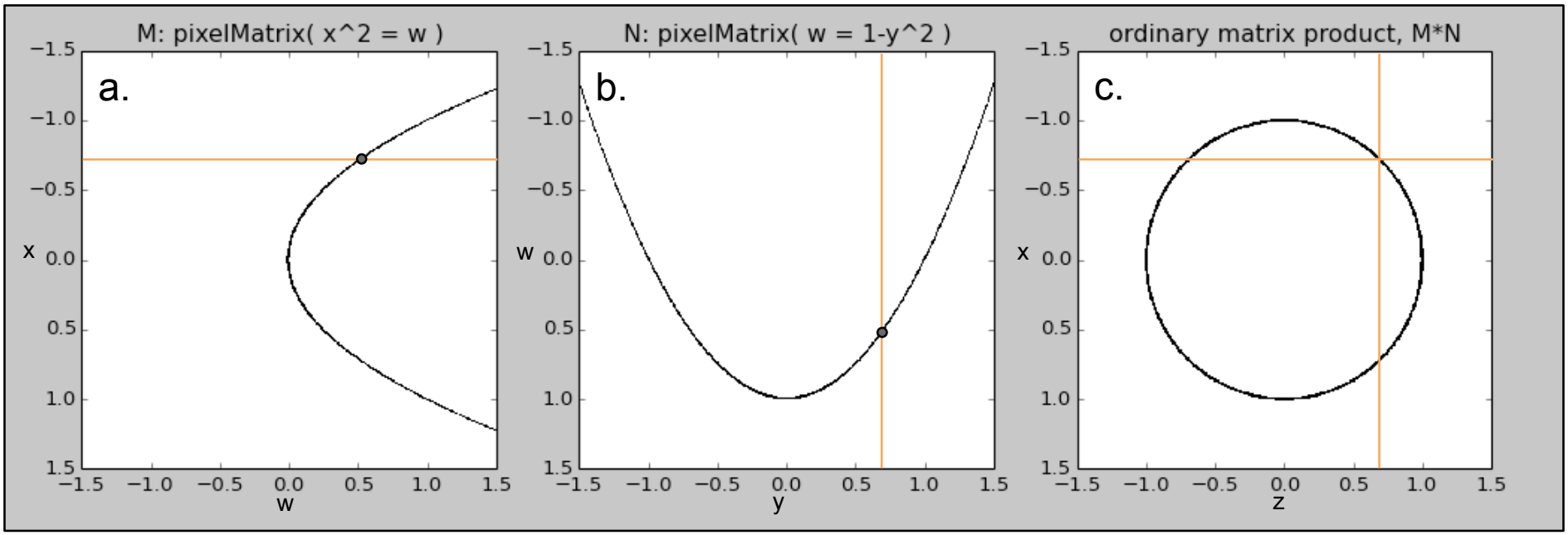
In general, we'll need to multiply higher-order arrays (tensors).

- 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.

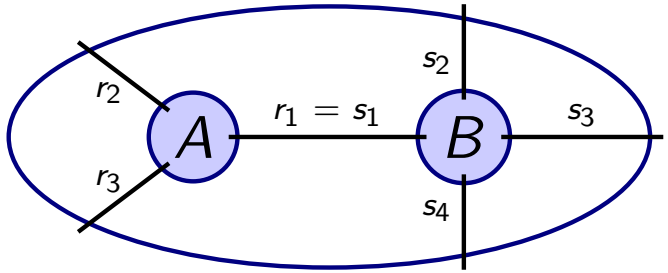
- For example, say that A is $r_1 \times \cdots \times r_m$ and that B is $s_1 \times \cdots \times s_n$.
- If $r_1 = s_1$, the product is an $r_2 \times \cdots \times r_m \times s_2 \times \cdots \times s_n$ array.

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.

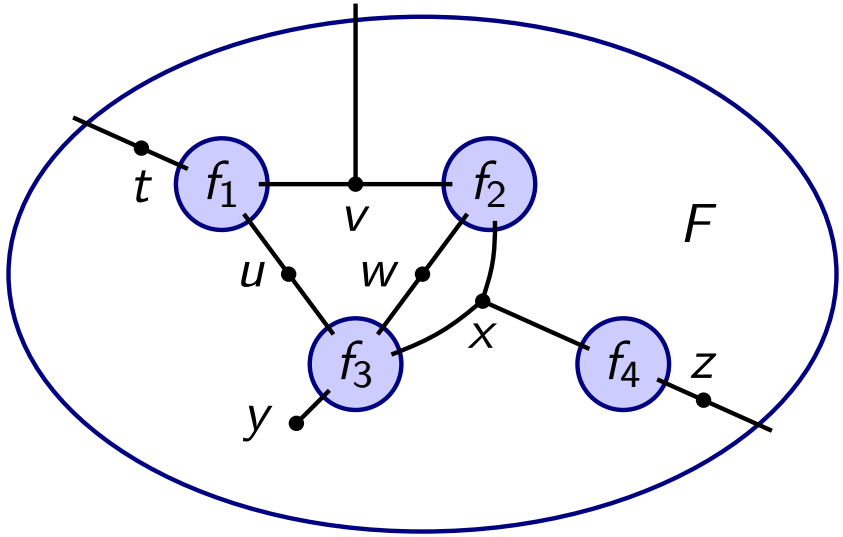


The general array multiplication algorithm

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

The general array multiplication algorithm

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



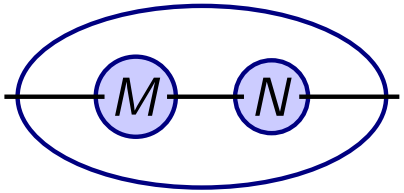
$$F(t,v,z) := \sum_{u,w,x,y} f_1(t,u,v) * f_2(v,w,x) * f_3(u,w,x,y) * f_4(x,z)$$

- This formula 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.

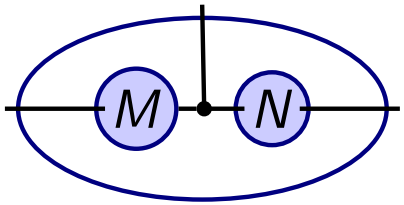
Example wiring diagrams for named operations

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

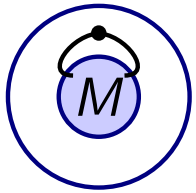
Multiplication: MN



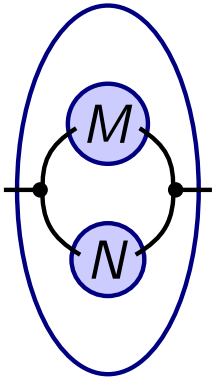
Khatri-Rao: $M \odot N$



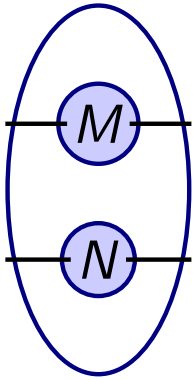
Trace: $\mathrm{Tr}(M)$



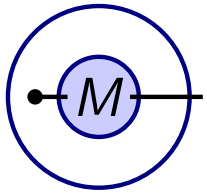
Hadamard: $M \circ N$



Kronecker: $M \otimes N$

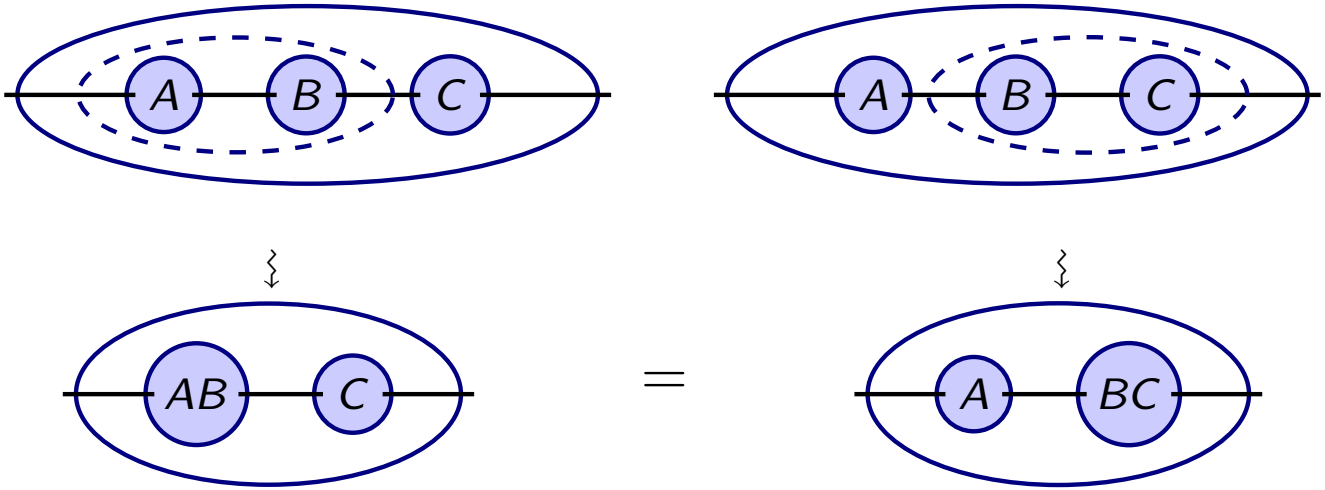


Marginalize: $\sum_i M_{i,j}$



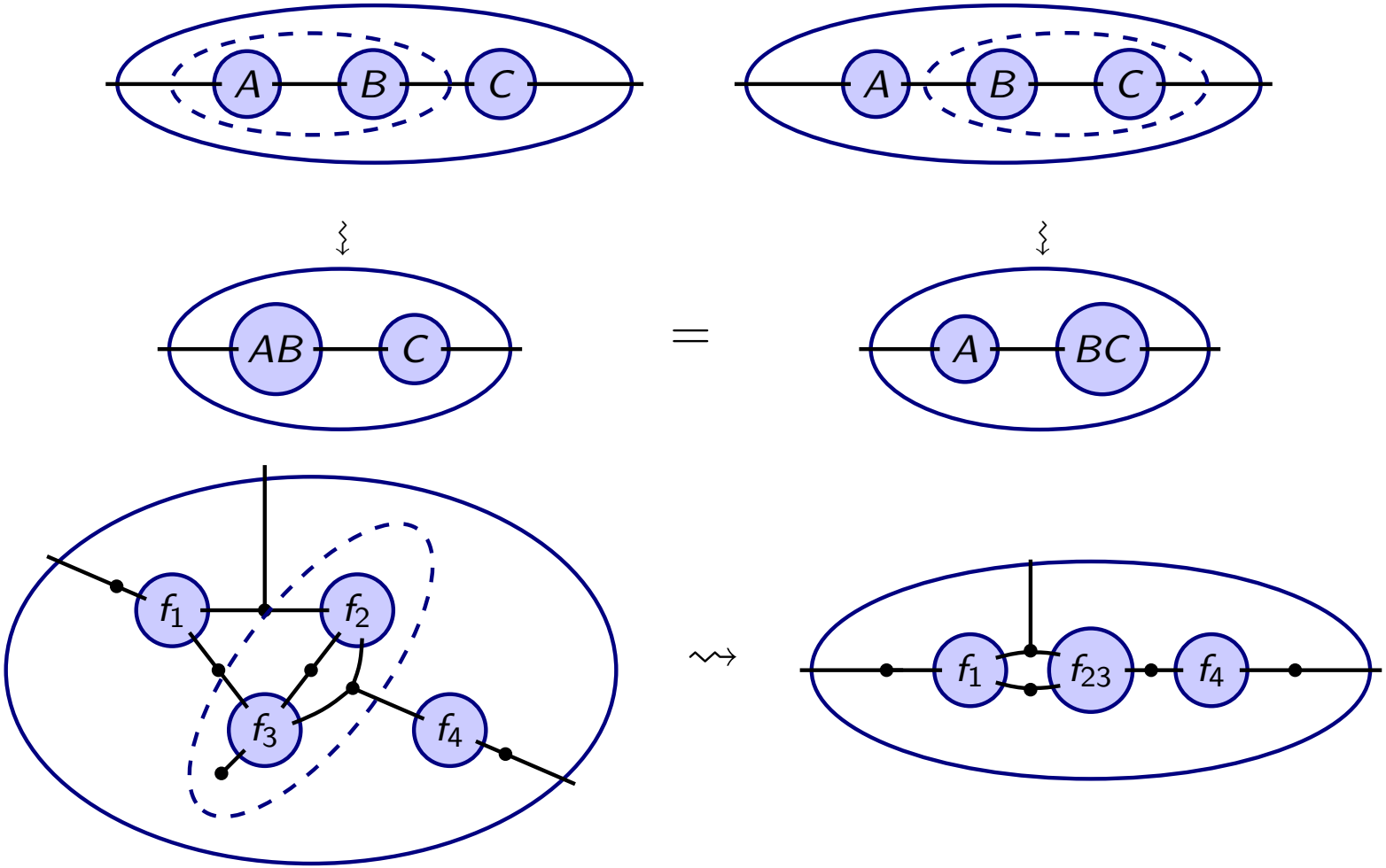
Clustering as associative law

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



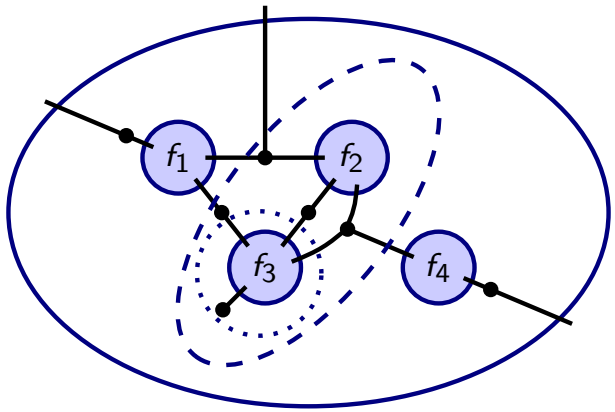
Clustering as associative law

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



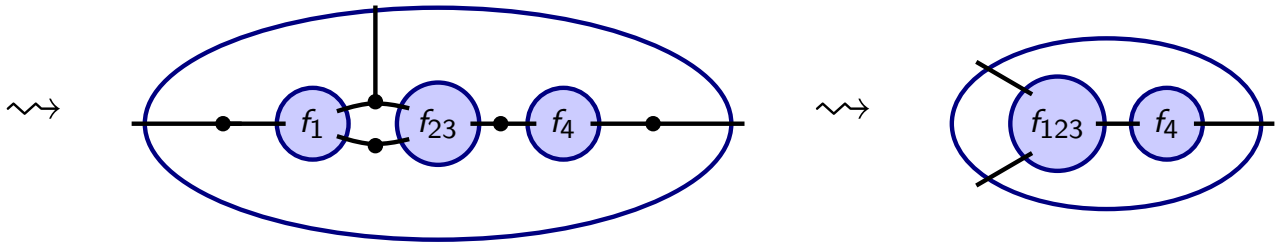
Clustering for speed

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



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

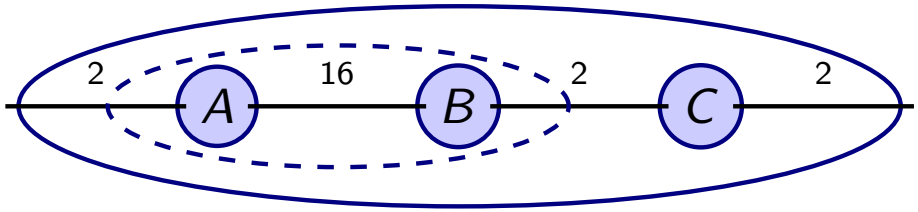
- Even if we can use sparsity, etc. solving big systems is expensive.
- Instead, we should use the associative law: cluster.



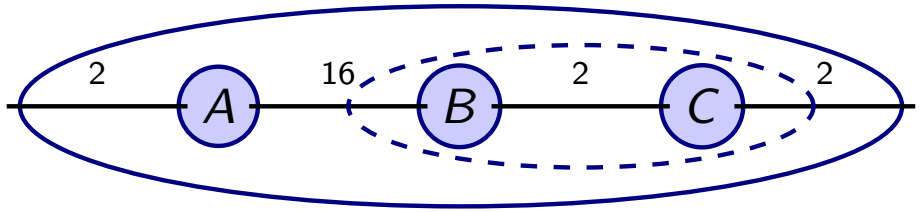
- Naive cost goes from $O(n^7)$ to $O(n^4)$.

Cluster trees

Different cluster strategies lead to different speeds.

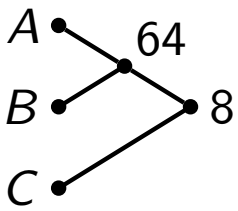


$$\text{cost} = (2 * 16 * 2) + (2 * 2 * 2) = 72$$

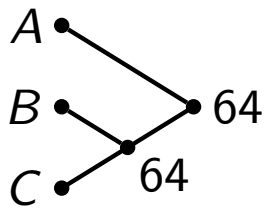


$$\text{cost} = (16 * 2 * 2) + (2 * 16 * 2) = 128$$

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



(Total: 72)



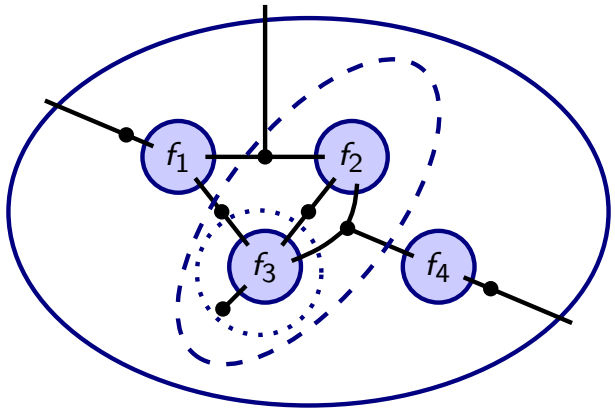
(Total: 128)

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.

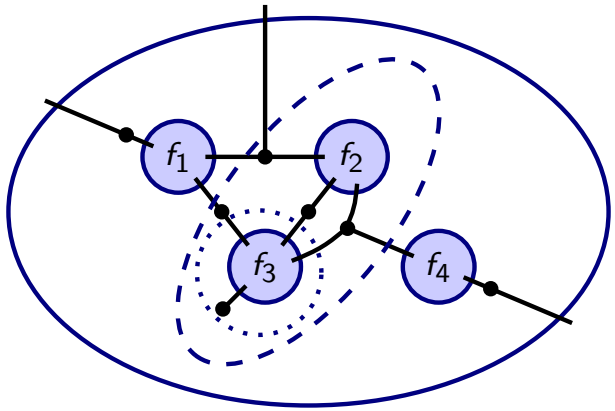
Some clustering algorithms

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.

How to plot equations

One thing we've skipped until now: how to plot each function?

- For any function $f(x_1, \ldots, x_n)$, its *plot* is an n-dimensional array P_f.
- We need some information to make the plot P_f:
  - Need a range $[a_i, b_i]$ and resolution r_i for each variable $1 \leq i \leq n$.
  - Need also a *tolerance*, $\epsilon > 0$.
- Given that, we'll fill each of the $\prod_{i=1}^{n} r_i$ cells in P_f.
  - Choose a cell $c = (c_1, \ldots, c_n)$ where $1 \leq c_i \leq r_i$ for each i.
  - c refers to a tiny sub-cube of the cube $C := [a_1, b_1] \times \cdots \times [a_n, b_n]$.
  - Let x be the center of cell c. We sample at x:
  - If $f(x) < \epsilon$ then put $P_f(c) = 1$; otherwise put $P_f(c) = 0$.
- A pixel is "on" if—at its center—the function f is near 0.

Uniform continuity

To get good properties, we need a connection between f, ϵ, and the r_i.

- Suppose f is continuous throughout the cube C. This means:

$$\forall \epsilon > 0, \forall x_0 \in C, \exists \delta > 0, \forall x \in C, \text{ if } |x - x_0| < \delta \text{ then } |f(x) - f(x_0)| < \epsilon$$

Uniform continuity

To get good properties, we need a connection between f, ϵ, and the r_i.

- Suppose f is continuous throughout the cube C. This means:

$$\forall \epsilon > 0, \forall x_0 \in C, \exists \delta > 0, \forall x \in C, \text{ if } |x - x_0| < \delta \text{ then } |f(x) - f(x_0)| < \epsilon$$

- Think of δ as a function of both x_0 and ϵ; lets write $\delta(x_0, \epsilon)$.
- Since C is closed and bounded, f is *uniformly continuous* on C.
  - Reason: we can take the max ("sup") δ over all $x_0 \in C$.
  - That is, let $\delta(\epsilon) := \sup_{x_0 \in C} \delta(x_0, \epsilon)$.
- Uniform continuity: δ depends only on ϵ.

$$\forall \epsilon > 0, \exists \delta > 0, \forall x_0 \in C, \forall x \in C, \text{ if } |x - x_0| < \delta \text{ then } |f(x) - f(x_0)| < \epsilon$$

Accuracy of plotting

We use the uniform continuity of $f(x)$ on C to choose the pixel size.

- For any tolerance ϵ, there is some δ that "works" throughout C.
  - Suppose each pixel c has radius δ, and let x_c be its center point.
  - Then for all $x \in c$, we know that $f(x)$ is within ϵ of $f(x_c)$.

Accuracy of plotting

We use the uniform continuity of $f(x)$ on C to choose the pixel size.

- For any tolerance ϵ, there is some δ that "works" throughout C.
  - Suppose each pixel c has radius δ, and let x_c be its center point.
  - Then for all $x \in c$, we know that $f(x)$ is within ϵ of $f(x_c)$.
- So we take our mesh to be such that every pixel has radius δ.
- We turn on pixel c if $f(x_c) < \epsilon$.

Accuracy of plotting

We use the uniform continuity of $f(x)$ on C to choose the pixel size.

- For any tolerance ϵ, there is some δ that "works" throughout C.
  - Suppose each pixel c has radius δ, and let x_c be its center point.
  - Then for all $x \in c$, we know that $f(x)$ is within ϵ of $f(x_c)$.
- So we take our mesh to be such that every pixel has radius δ.
- We turn on pixel c if $f(x_c) < \epsilon$.
- What guarantees do we have?
  - If $f(x) = 0$ anywhere in c, then $f(x_c) < \epsilon$, so the pixel is *on*.
  - If the pixel is *on* then $f(x_c) < \epsilon$, hence $f(x) < 2\epsilon$ for all $x \in c$.

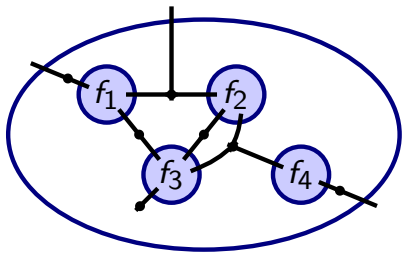
Accuracy of the whole method

We are ready to combine our plots as discussed above. What happens?

- Suppose the plot of each function $f(x)$ has the above two properties:
  - If $f(x) = 0$ anywhere in c, then $f(x_c) < \epsilon$, so the pixel is *on*.
  - If the pixel is *on* then $f(x_c) < \epsilon$, hence $f(x) < 2\epsilon$ for all $x \in c$.
- We then multiply the arrays according to the wiring diagram:

Plot as arrays by sampling at the center point:

$$|f_1(t, u, v)| < 2\epsilon$$
$$|f_2(v, w, x)| < 2\epsilon$$
$$|f_3(u, w, x, y)| < 2\epsilon$$
$$|f_4(x, z)| < 2\epsilon$$



- What does it mean if a pixel is on in the final result?
  - It means that there exists a simultaneous 2ϵ-solution to the system.
  - There exist values of all unexposed variables making $|f_i| < 2\epsilon$ for each i.
- Thus we have complete control on the accuracy in this sense.

Drawing 3D arrays may be better than one thinks

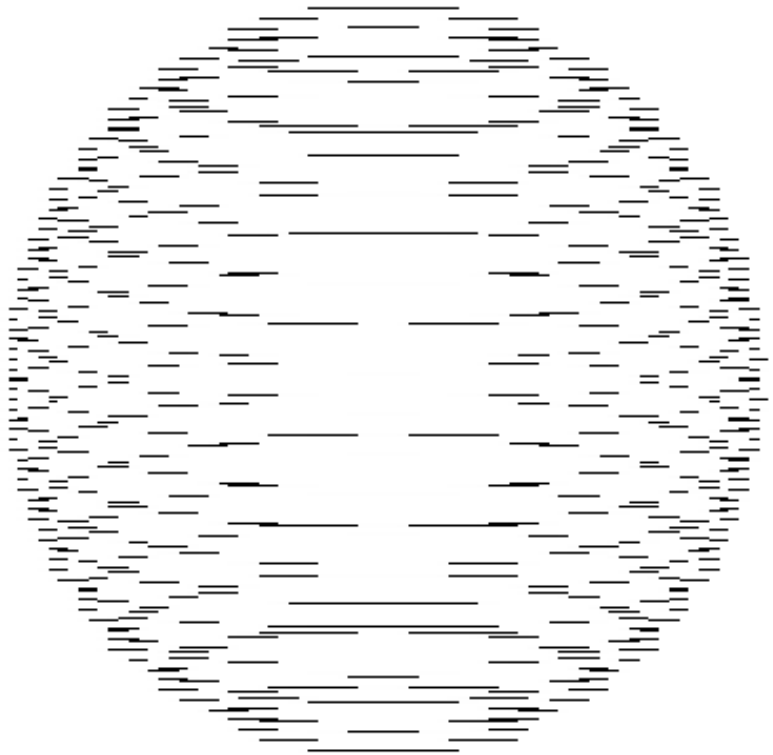
Before moving on to PDEs, here's something I found interesting.

- 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 dynamical systems and PDEs

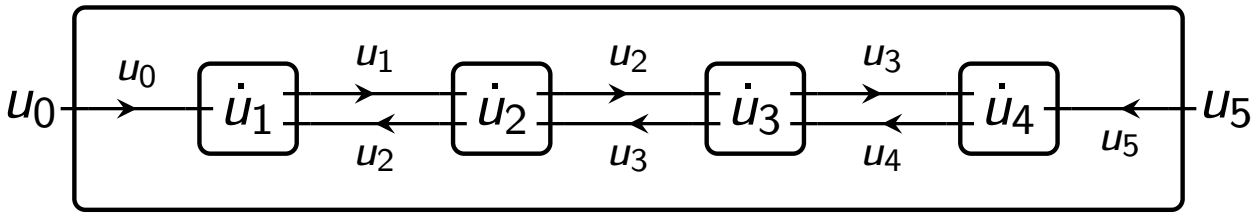
In this section I'll discuss an application of the Pixel Array method.

- The whole PA idea came from category theory.
- It is part of a bigger story of *compositionality*.
- In many domains, you want to analyze a system by considering simpler components.

Let's see how this applies in PDEs, for example the heat equation.

The heat equation

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



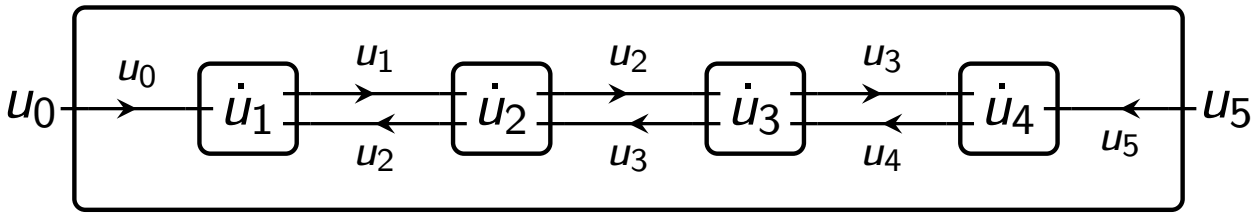
(1)

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.

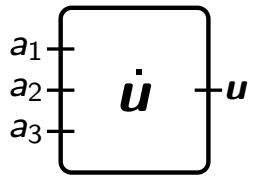
$$\dot{u}_i = u_{i-1} - 2u_i + u_{i+1} \qquad (i = 1, \ldots, 4)$$

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.

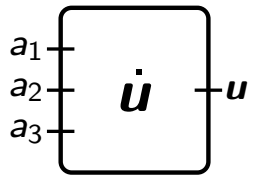


A box of the above shape represents an ODE $\dot{\boldsymbol{u}} = f(\boldsymbol{u}, a_1, \ldots, a_n)$.

- 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.

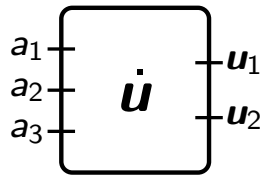


A box of the above shape represents an ODE $\dot{\boldsymbol{u}} = f(\boldsymbol{u}, a_1, \ldots, a_n)$.

- 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: 

Open dynamical systems

Let's look at the boxes in isolation.

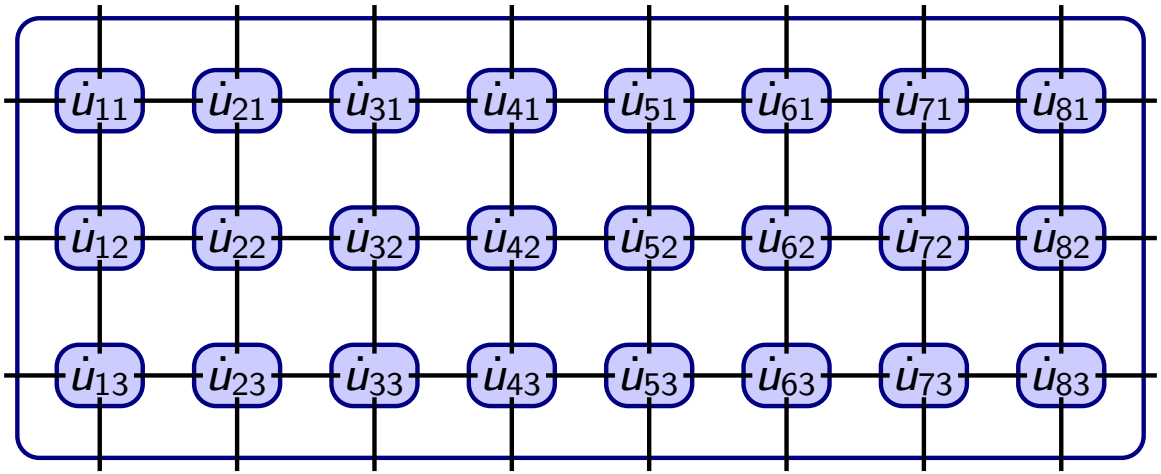


A box of the above shape represents an ODE $\dot{\boldsymbol{u}} = f(\boldsymbol{u}, a_1, \ldots, a_n)$.

- 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: 

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

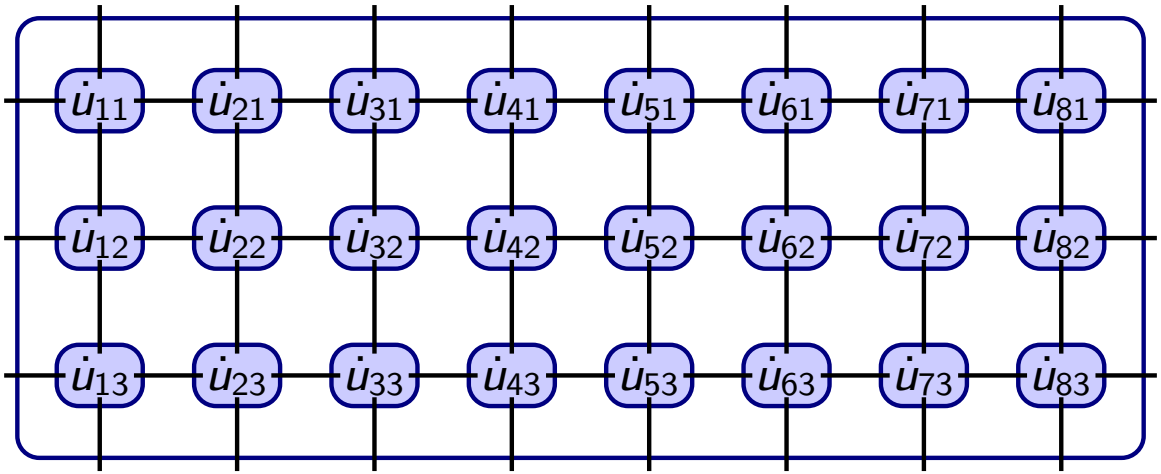
Interconnected open dynamical systems (ODS's)



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.

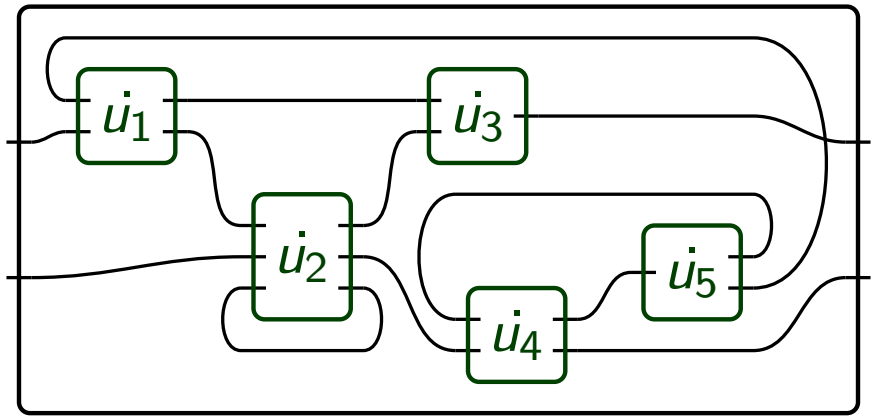
Interconnected open dynamical systems (ODS's)



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.

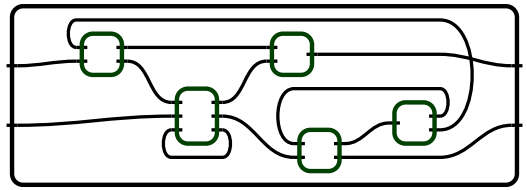
Interconnected open dynamical systems (ODS's)



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).

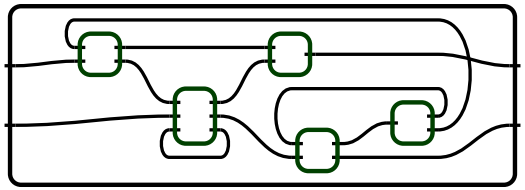
Pixel Array analysis



The Pixel Array method lets you assemble local results.

- Suppose you want the steady state solutions to the whole system.
- To do so, first plot the steady states of each component ODS.
- Each plot is an array, indexed by the input and output wires.
  - The rows are indexed by the steady input values;
  - The columns are indexed by the steady output values;
  - Given a steady input and steady output, you get a cell in the matrix.
  - Put a 1 there if there's a corresponding steady state; else 0.

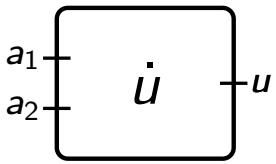
Pixel Array analysis

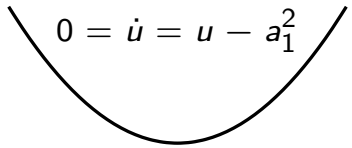


The Pixel Array method lets you assemble local results.

- Suppose you want the steady state solutions to the whole system.
- To do so, first plot the steady states of each component ODS.
- Each plot is an array, indexed by the input and output wires.
  - The rows are indexed by the steady input values;
  - The columns are indexed by the steady output values;
  - Given a steady input and steady output, you get a cell in the matrix.
  - Put a 1 there if there's a corresponding steady state; else 0.
- Then multiply these arrays according to the wiring diagram.
- The result is the steady states of the whole system (PDE).

Plotting

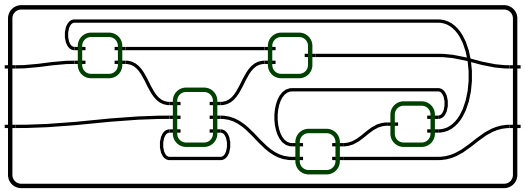




1	0	0	0	1
1	0	0	0	1
0	1	0	1	0
0	1	1	1	0
0	1	1	1	0

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

- The steady-state plot for this ODS will be an order-$(n+1)$ array.
  - Set $\dot{u} = 0$ and get an equation $f(u, a_i) = 0$.
  - This is called the *bifurcation diagram* of the dynamical system.



- Multiply the component bifurcation diagrams together.
- The result is the bifurcation diagram of the whole system.

Outline

Apples and oranges

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

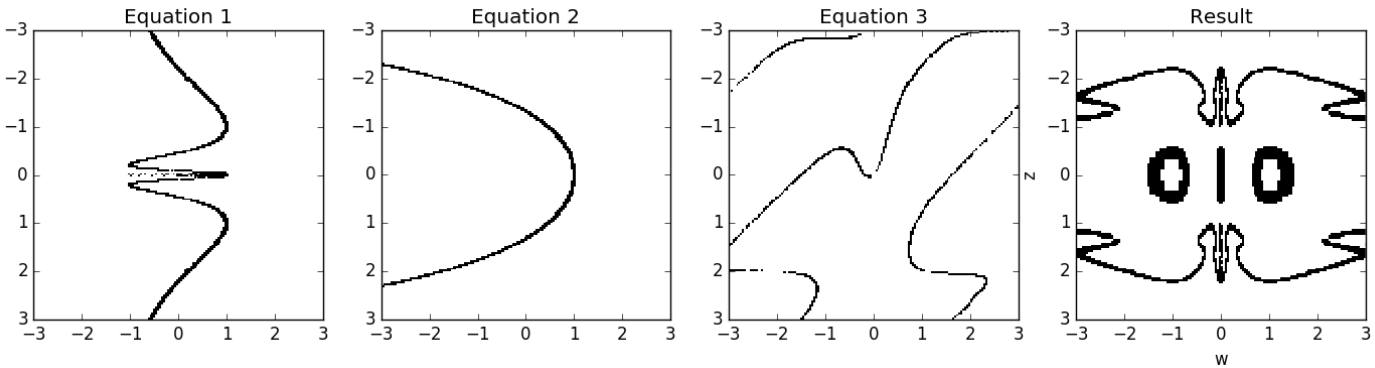
	Pixel Array method	**Newton's method**
Inputs:	a range for each variable	a good initial guess
Outputs:	all solutions for some variables	one solution in all variables.

Apples and oranges

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

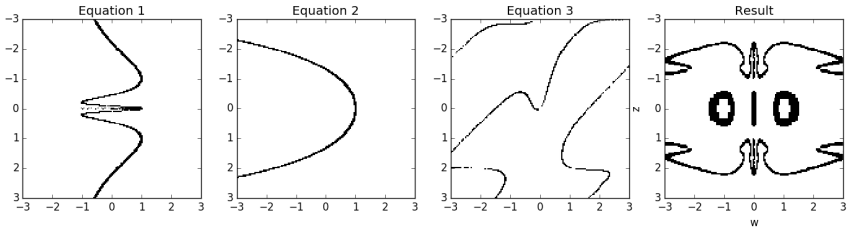
	Pixel Array method	**Newton's method**
Inputs:	a range for each variable	a good initial guess
Outputs:	all solutions for some variables	one solution in all variables.

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.

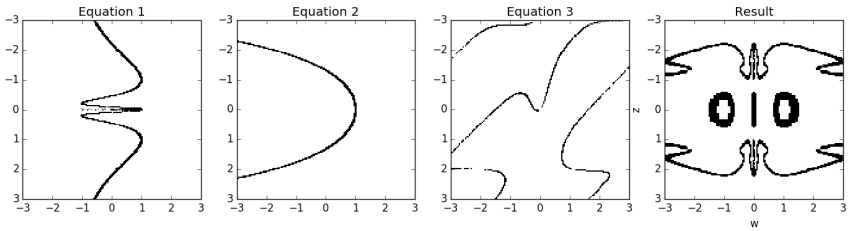
- We iterated over all boxes in the grid, each as an initial guess.

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

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

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

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.

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

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

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

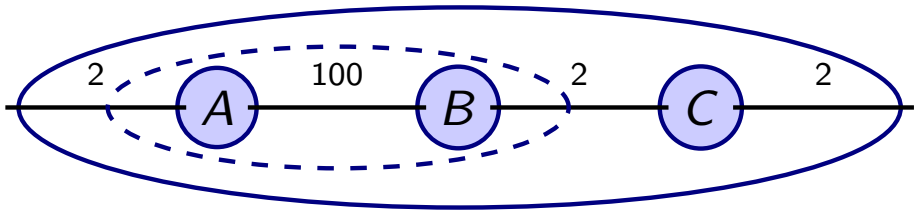
- This feels unfair, but we haven't found a better approach against which to benchmark. Thoughts?

Under that comparison, for the above case, PA was over 7200x faster.

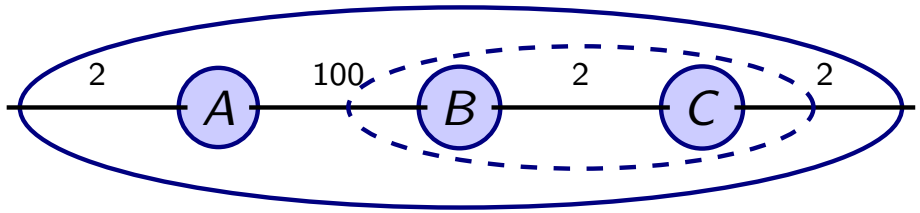
- PA: 1.5 seconds; Newton: we stopped Julia's NLsolve after 3 hours.

How to cluster?

- There exist methods for deciding what order to multiply a chain of matrices.



cost ≈ 400



cost $= \approx 800$

- Our question is how to cluster any wiring diagram, not just a chain.
- There exist clustering algorithms for graphs and hypergraphs.
  - A wiring diagram can be considered a "pointed, weighted hypergraph".
  - Existing algorithms are not optimized for our particular problem.
- We have some heuristic algorithms, but it'd be nice to have a really good one!

Outline

Summary

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

- We have an array for each equation, obtained by plotting $|f| < \epsilon$.
- We know how variables are shared (captured by a wiring diagram).
- We cluster the arrays to minimize the cost of array multiplication.
- Multiplying the arrays gives the simultaneous solution set to $|f| < 2\epsilon$.

Summary

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

- We have an array for each equation, obtained by plotting $|f| < \epsilon$.
- We know how variables are shared (captured by a wiring diagram).
- We cluster the arrays to minimize the cost of array multiplication.
- Multiplying the arrays gives the simultaneous solution set to $|f| < 2\epsilon$.

The PA method has some good properties.

- It seems quite fast for finding all solutions in a bounding box.
- (Question: how should we benchmark it to be sure?)
- It has good accuracy guarantees.
- It can be used to find steady states of interconnected dynamical systems and PDEs.

Summary

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

- We have an array for each equation, obtained by plotting $|f| < \epsilon$.
- We know how variables are shared (captured by a wiring diagram).
- We cluster the arrays to minimize the cost of array multiplication.
- Multiplying the arrays gives the simultaneous solution set to $|f| < 2\epsilon$.

The PA method has some good properties.

- It seems quite fast for finding all solutions in a bounding box.
- (Question: how should we benchmark it to be sure?)
- It has good accuracy guarantees.
- It can be used to find steady states of interconnected dynamical systems and PDEs.

Questions and comments are welcome!