Partitioning Regression

Daniel Ashlock, Senior Member IEEE
Joint work with Joseph Brown

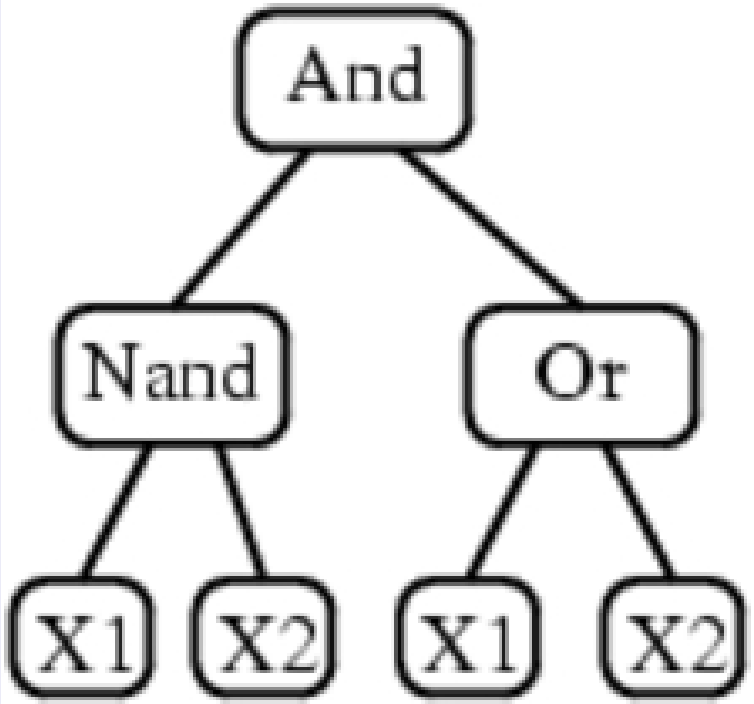
Motivation

Symbolic regression is the use of genetic programming to perform a very general sort of regression on data. Genetic programming permits the regression to be across a very large space of models. In this talk we combine this regression with a form of model selection that partitions the data. These partitioning regressors may be used to perform clustering of data.

A review of what I said last week with better pictures.

Representation for genetic programming

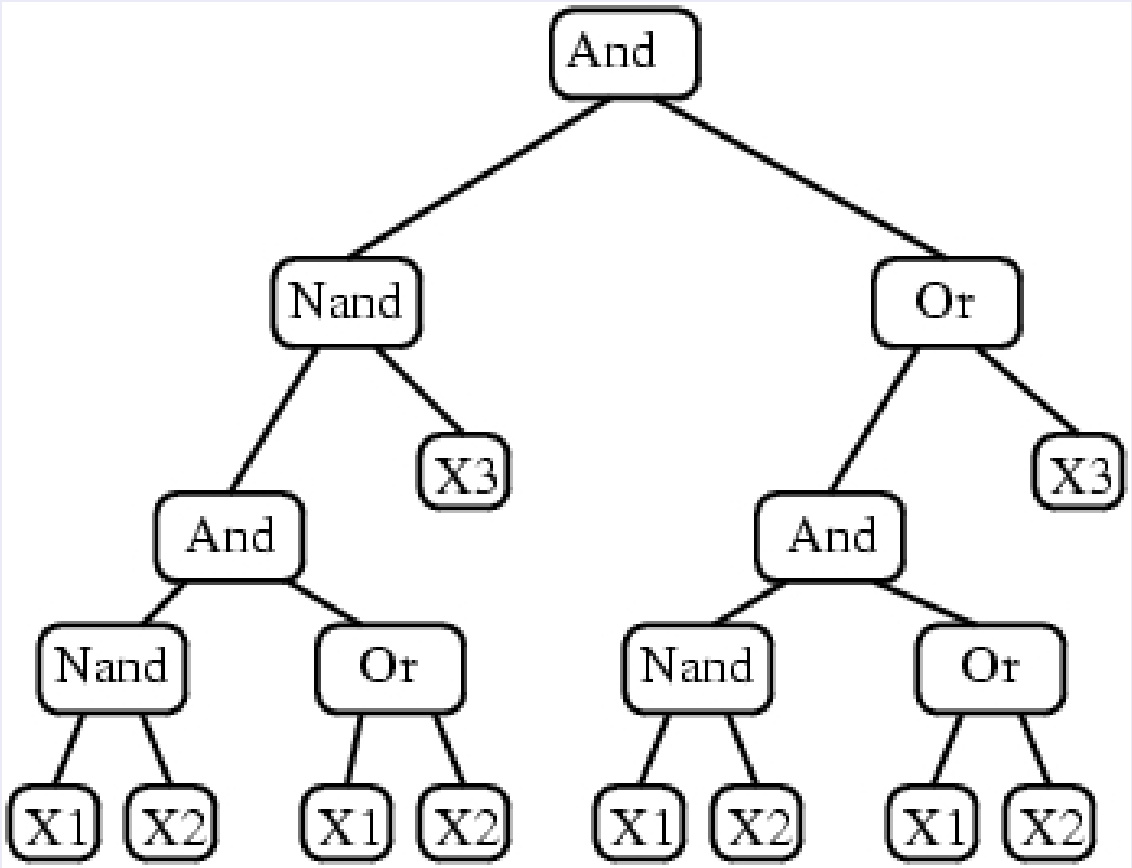
Genetic programming is a type of evolutionary computation that operates on a population of dynamically allocated, variable sized formulas. It was thought up by John Koza and John Rice at Stanford University. The formulas are stored as parse trees:



2-parity

The parity problem

The **parity problem** is a logic function induction problem. *Odd* parity is true if an odd number of its inputs are true. Parity is a standard Boolean function induction test problem, common in ANN research.

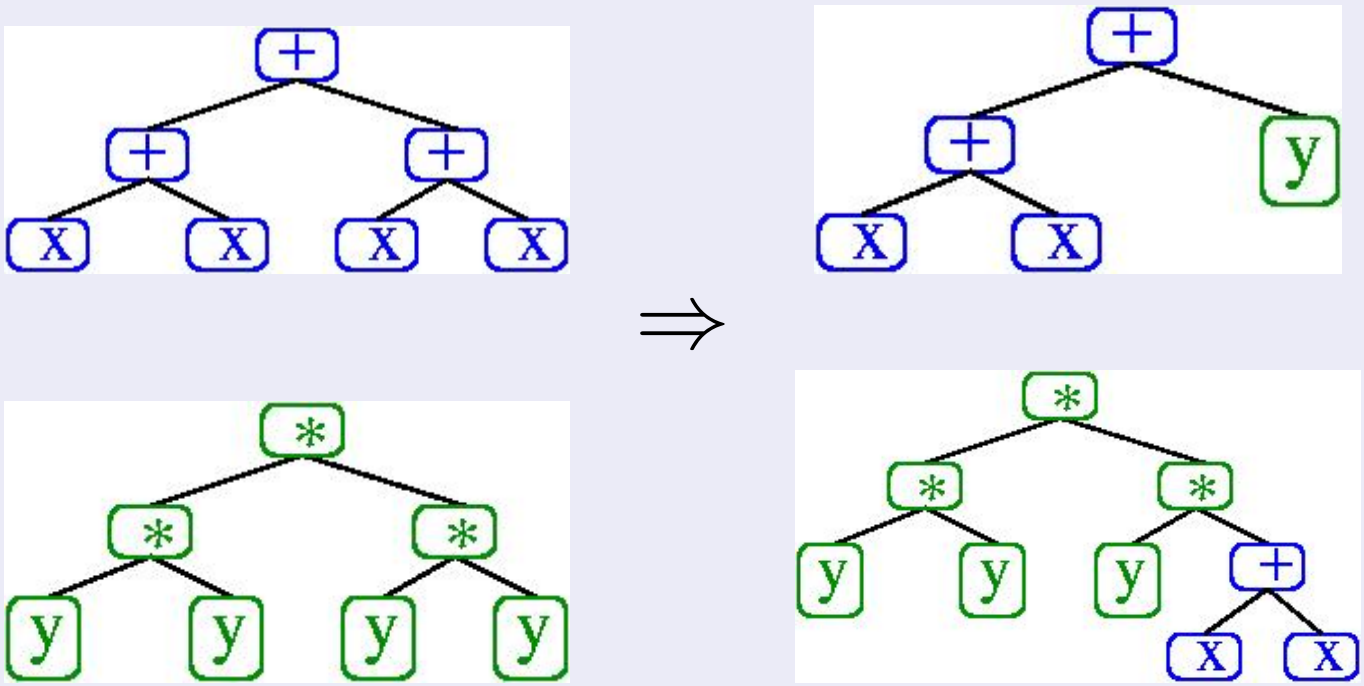


3-parity

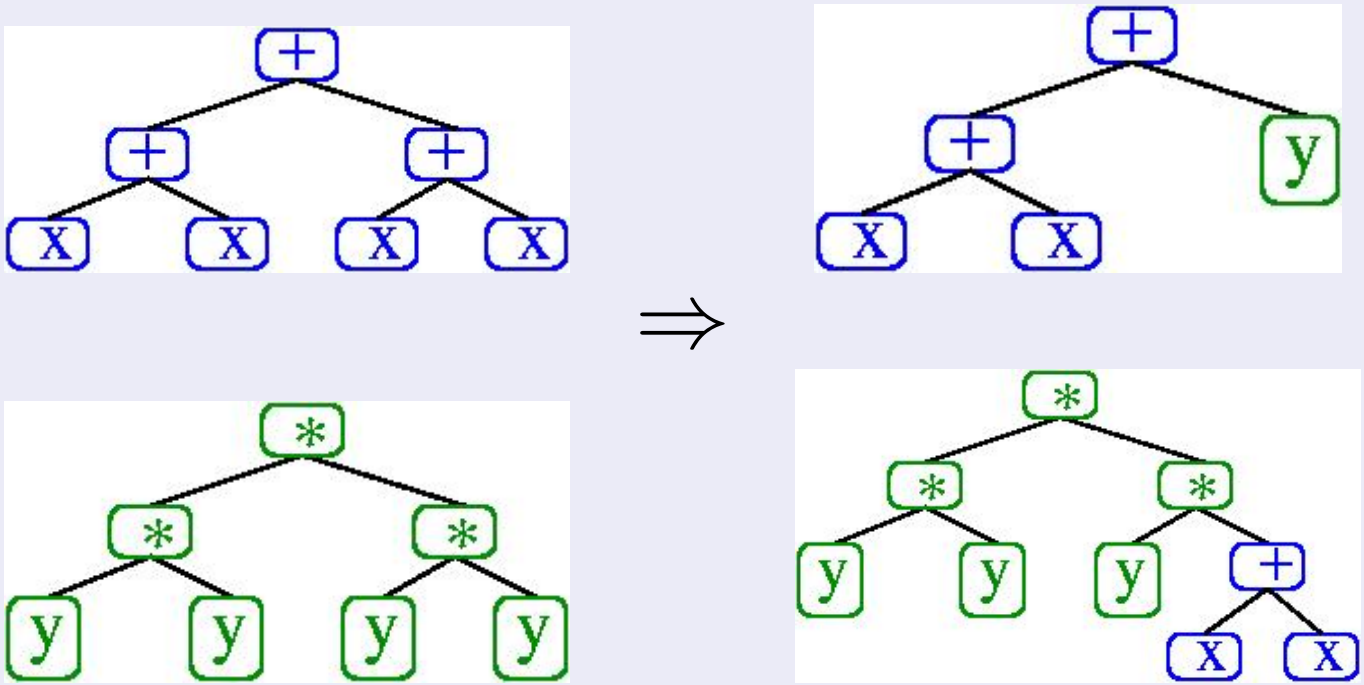
Some Feature of Genetic Programming

- The search space is exceedingly large and many-one.
- The standard crossover operator, subtree crossover (shown below), is highly disruptive.
- Genetic programming deals badly with ephemeral constants in real-valued problems, e.g. in symbolic regression.

Subtree crossover



Notice that subtree crossover is not size preserving.



Bloat - a critical feature

Larger trees can incorporate large amounts of material that does not affect their output (and hence fitness). If crossover or mutation take place inside unused "code" their current fitness becomes more heritable. The accumulation of such unused code is termed **bloat**.

Bloat and possible antidotes

- The maximum rate of bloat is proportional to 2^n after n generations.
- The actual rate is near $(\sqrt{2})^n$ after n generations.

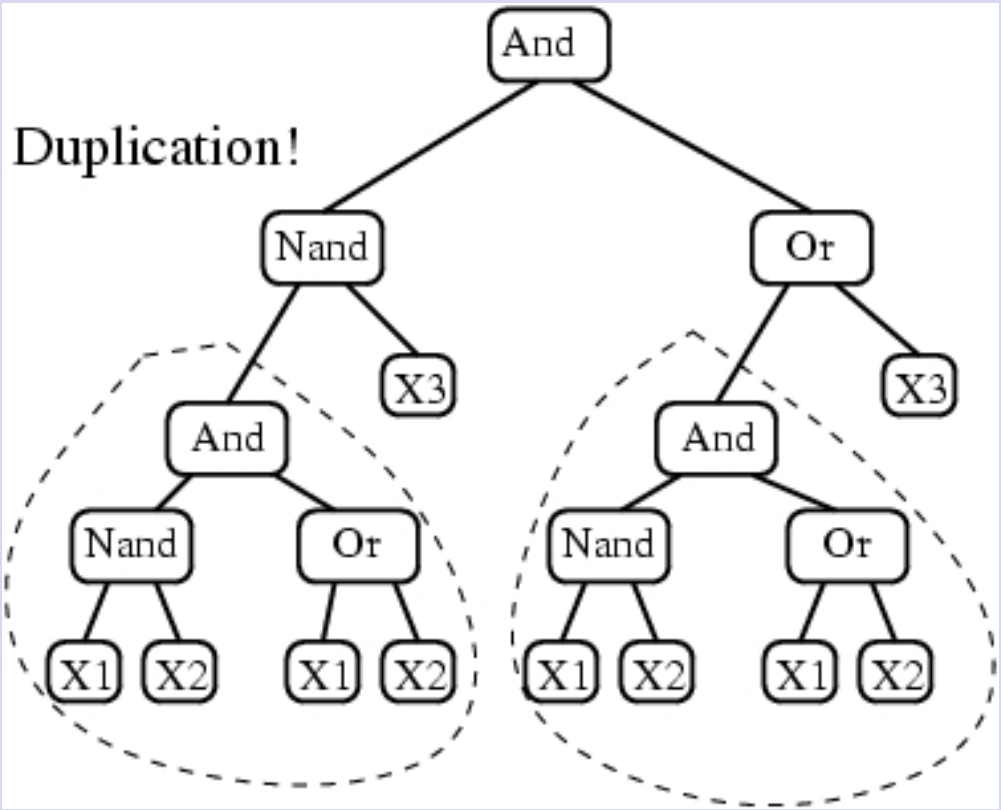
Possible antidotes:

- **Limit tree depth**. This is called burning and it the original method.
- Limit total nodes, promote a subtree when the limit is violated. This is called **chopping**. This also "turns over the pond", bringing up material from the depths.
- Impose tie-breaking on equal fitness for smaller trees. This is called a **bad idea** or possibly **parsimony pressure**. Soft parsimony pressure can help (unpublished: W. Ashlock).
- Single parent techniques, explained subsequently.
- Changing the representation, two examples follow.

Parity is *not* a natural tree-structured GP-problem

The picture below demonstrates an intrinsic inefficiency of genetic programming, at least for parity.

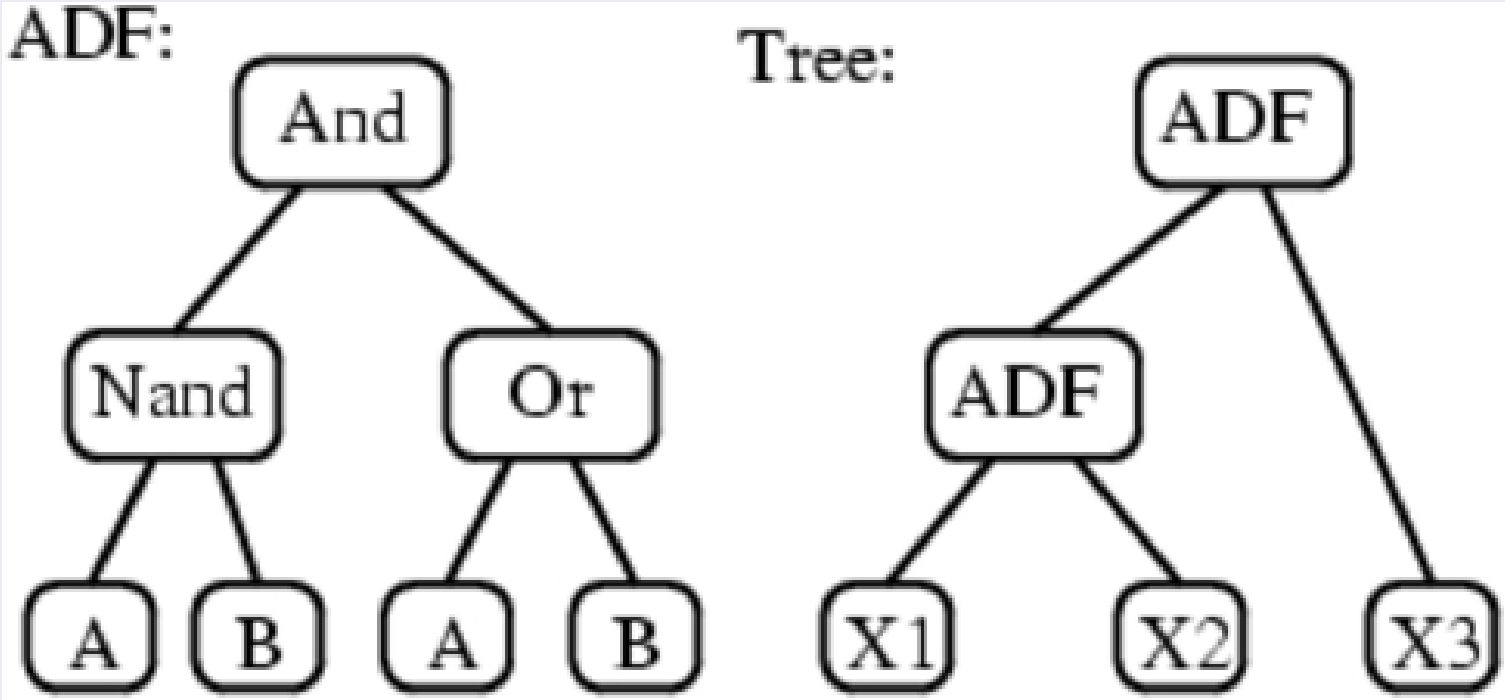
Three parity exposed!



This problem with genetic programming is that it is **intrinsically inefficient**. A useful value, once computed, must be recomputed to be used elsewhere. This problem is reduced by the use of **automatically defined functions** or **ADFs**. These are parse tree **subroutines**, implemented as additional operations that call their own parse trees.

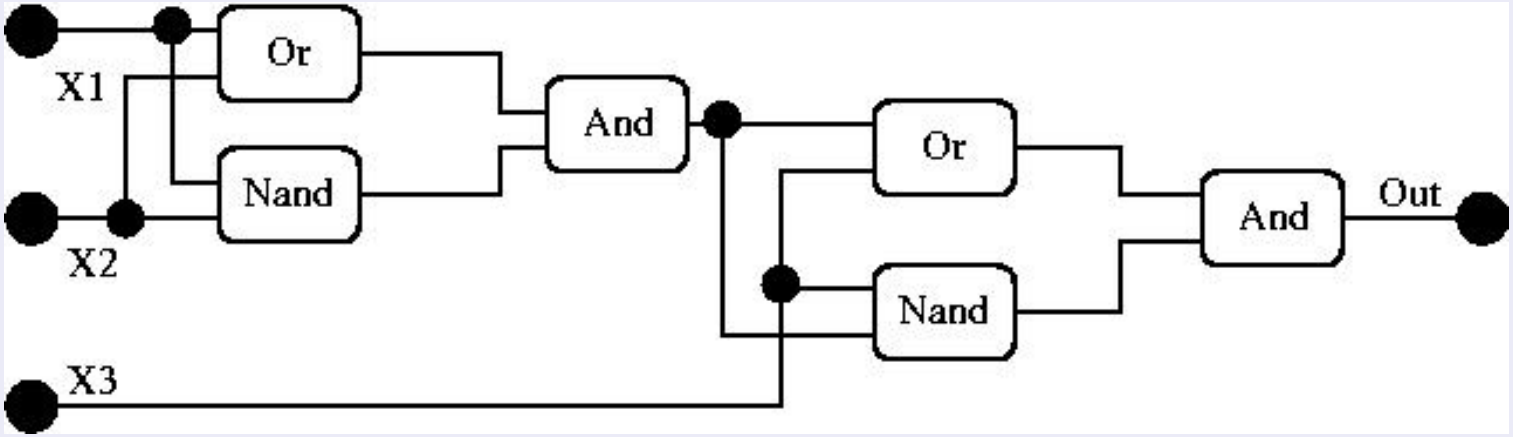
An ADF that Helps

Here is an example of how an ADF can make matters better:



BUT in Koza's original research on the problem this ADF did not appear! Other more complex ADFs appeared that were **more useful for particular instances of parity** but not friendly to generalization.

A circuit-like solution



This is a more efficient way of computing 3-parity; it is also closer to (though still less efficient than) the way a neural net would compute 3-parity. The big difference is the use of **fan-out greater than 1**: the first **And** gate's output is connected to two other gates.

A directed acyclic graphs representation: Function Stacks

A **function stack** is a representation derived from **Cartesian Genetic Programming**. Both representations store evolvable formulas as directed acyclic graphs (DAGs).

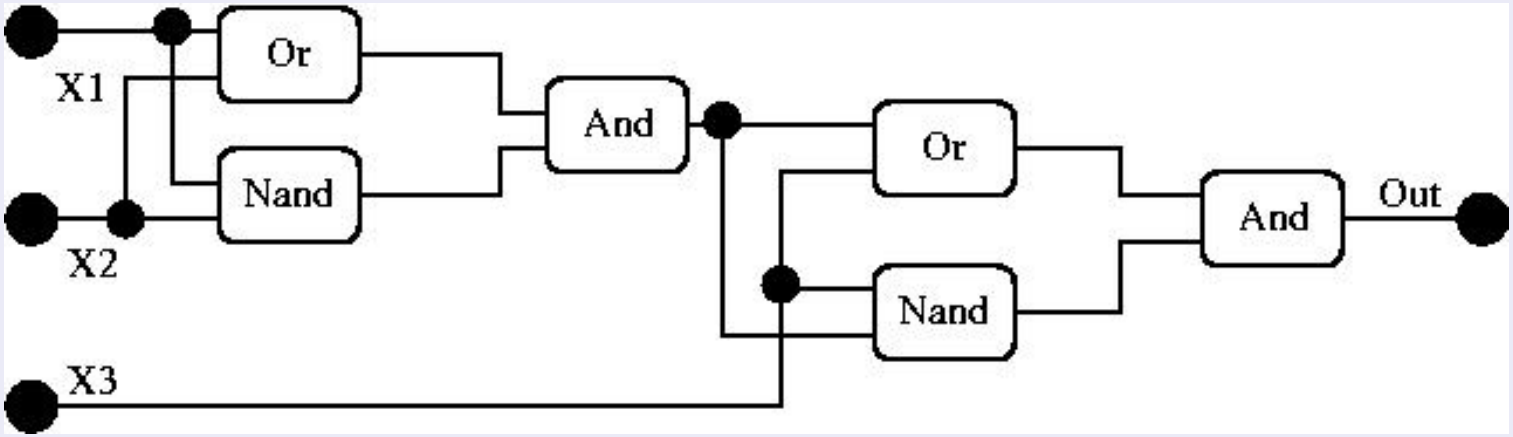
J. F. Miller, S. L. Smith Redundancy and computational efficiency in Cartesian genetic programming, IEEE Transactions on Evolutionary Computation, Volume: 10 , Issue: 2, PP: 167 - 174

Function Stacks

A function stack is implemented as an array of **nodes**. Each node has a (Boolean) function and specifies two arguments. The arguments can be input variables or the output of a node with a higher index number. Function stacks are thus a **directed acyclic graph** representation for genetic programming.

Node	Function	Arg1	Arg2
00	And	N01	N02
01	Nand	N03	X03
02	Or	N03	X03
03	And	N04	N05
04	Nand	X01	X02
05	Or	X01	X02

Tabular layout



Left-to-right Digraph layout.

A comment on search space size

Suppose that we are trying to find a parse tree that computes n-parity. Then the need to repeat subtrees means that if the size of a solution is $f(n)$ then

$$f(n) = 3 + 2\left(\min_{a+b=n}(f(a) + f(b))\right)$$

With $f(1) = 1$ we see the function grows:

$$1, 7, 19, 31, 55, 79, \ldots \geq 2^n$$

For function stacks the function is

$$1, 5, 9, 13, 17, \ldots = 4n - 3$$

Exponential versus linear: which space would you rather search?

The use of DAGs in genetic programming

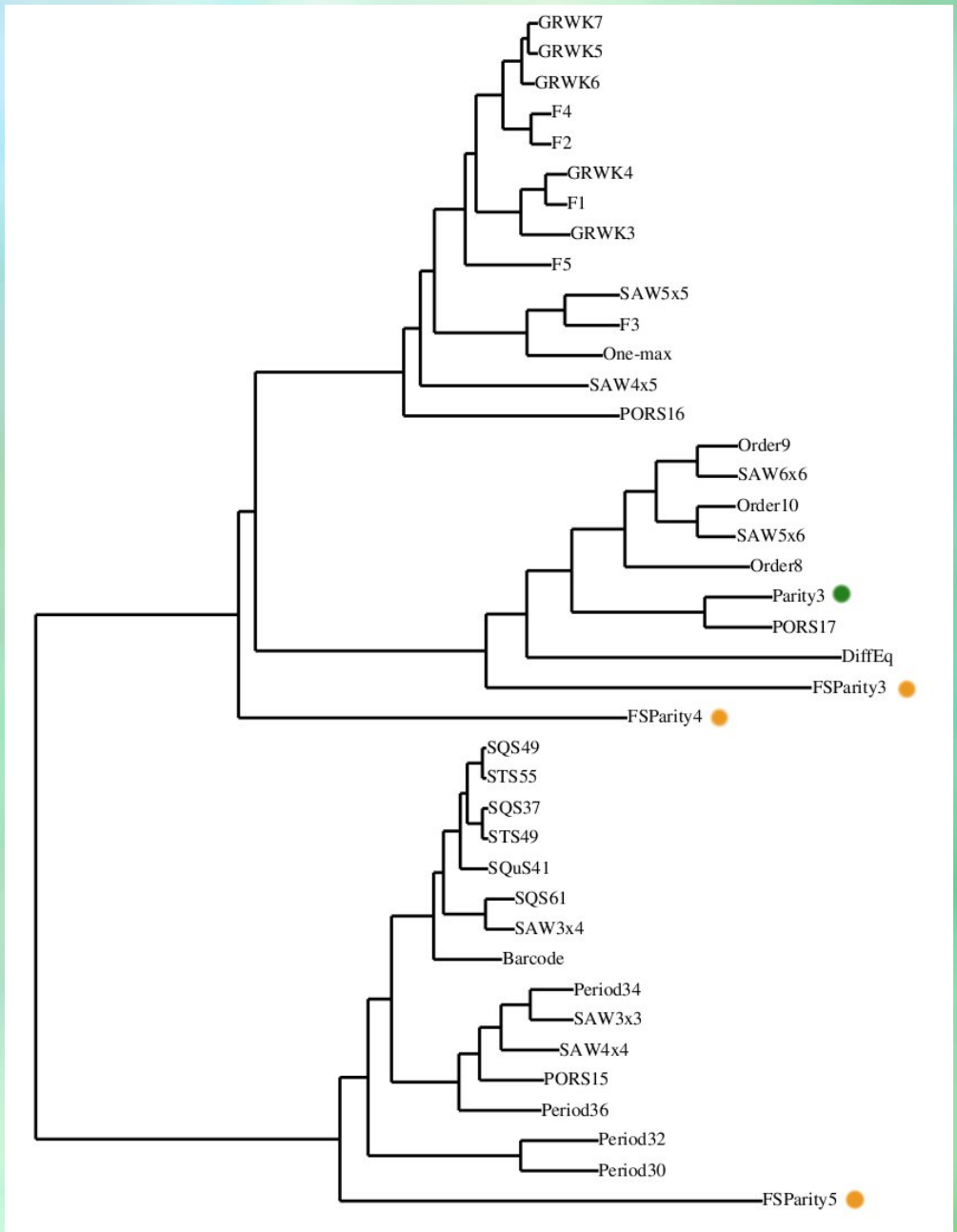
DAGs are **not new** in genetic programming. Permitting a **fan-out of more than one**, however, is an idea that seems to originate with Julian Miller. Why use DAGs to store a tree? **Standard genetic programming generates vast numbers of identical subtrees**; storing them once and using pointers can save a lot of storage and enormously speed evaluation. This idea dates to a workshop on the future of genetic programming at GP-96.

Aside: a reason why large trees are good

An ubiquitous phenomenon in genetic programming is **bloat**: uncontrolled growth of parse trees. It provides protection from the disruptive crossover operator.

Theorem:(Bill Langdon) **As tree size grows all Boolean functions appear with fixed, distinct probabilities**.

This theorem ensures honest sampling of the space of Boolean functions as the trees get larger. There may be analogs to this theorem for other data types.



Parity in the Taxonomy

Odd parity for 3, 4, and 5 inputs with **function stacks** and odd 3-parity for a **standard tree representation** with no ADFs were incorporated into the taxonomy. Recall: **horizontal distance within the taxonomy represents distance between the feature vectors** in Euclidean space.

Parity is another example of a diverse family of problems; while 3-parity with function stacks had, on average, a much shorter time-to-solution that the tree-based representation the two problem were close in the taxonomy.

DAG Variation Operators

- Since function stacks are stored as arrays of nodes, we can use crossover operators suitable for linear representations. Notice that these operators, unlike sub-tree crossover, are **conservative** (crossover of identical parents produces children that are copies of those parents).
- The mutation operator simply makes random modifications to a randomly selected node. In genetic programming some mutation operators **modify the tree topology**, making them even more difficult to understand.

Another property

Notice that multiple outputs can be read from the DAG at different positions. This means the representation can be used to evolve systems of linked equations in a natural way.

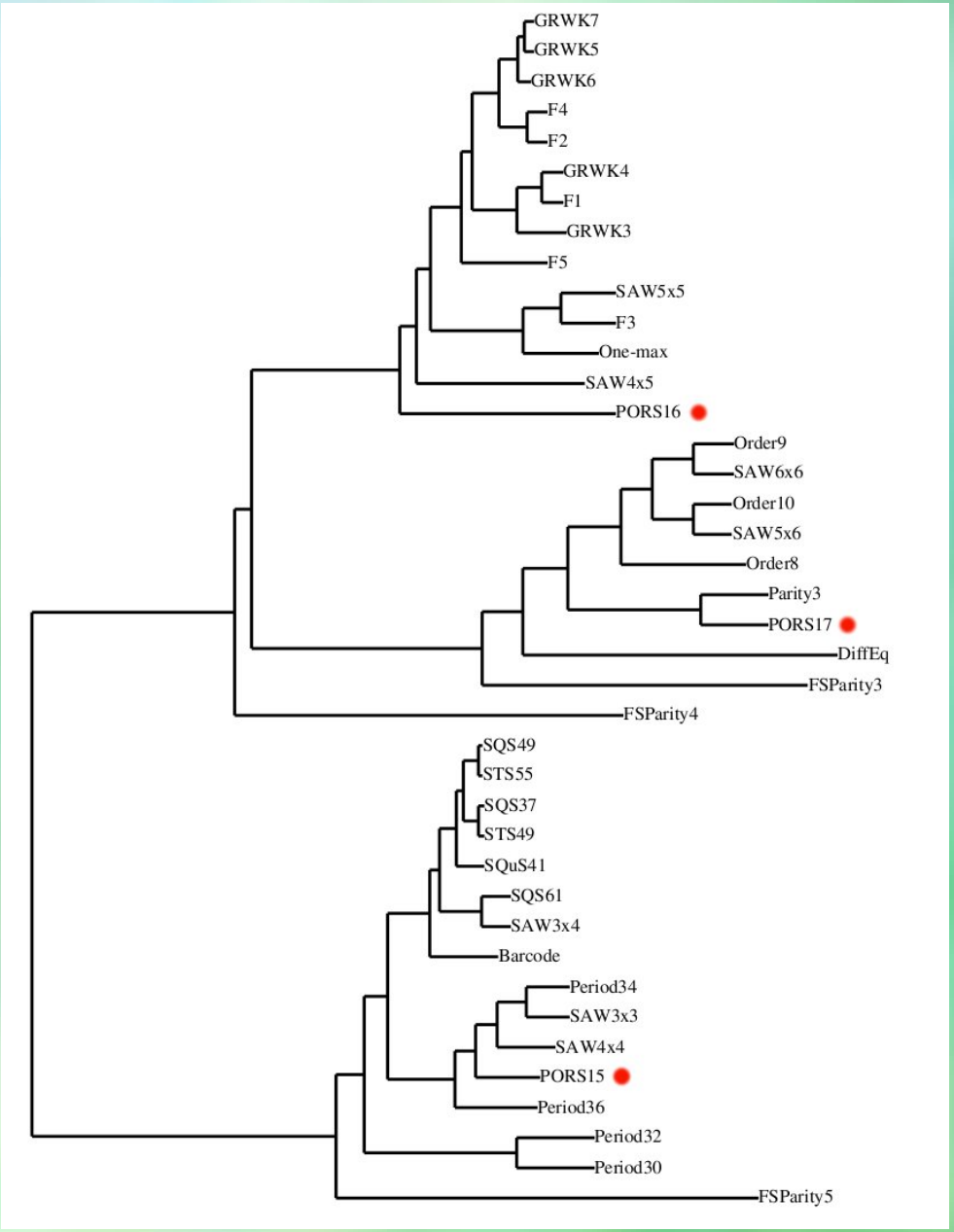
Operation Choice in GP

When working with any type of genetic programming, **you are permitted to choose which operations and terminals (inputs) you use**. This is an extraordinary point at which to embed domain knowledge and improve performance. Jason Moore's group has worked out techniques for **testing and comparing multiple sets of operations and terminals** and their work is well worth reading.

Idea source

R. J. Urbaniwicz, B. C. White, and J. H. Moore Mask Functions for the Symbolic Modeling of Epistasis Using Genetic Programming, in Proceedings of GECCO 2008, PP 339-346.

Another point of view on ADFs is that they are **evolvable operations** that permit the GP-system is permitted to augment itself on the fly.



GP using side effects

The **(P)lus (O)ne (R)ecall (S)tore (PORS)** is an example of a genetic programming problem that uses side effects. The problem has two operations **integer addition** and **store (Sto)** that returns its input and also writes it into a memory. There are also two terminals **1** and **recall (Rcl)** that returns the current contents of the memory.

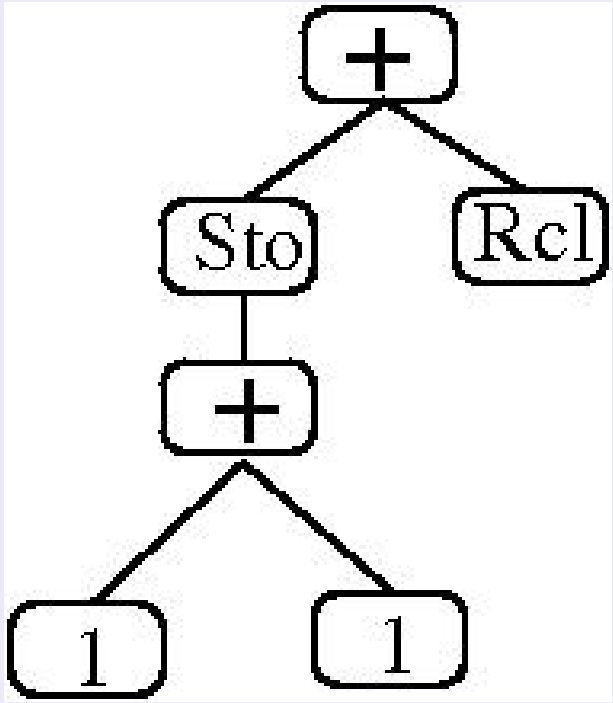
PORS is a **maximum problem** whose goal is to generate the largest possible number using at most some fixed number of nodes. Thus **PORS16** seeks to generate the largest possible number using a sixteen (or fewer) node tree.

The taxonomy at the left shows that PORS problems also contains diverse problem types, appearing in all major branches of the taxonomy. There are three major classes of PORS problem depending on the number of nodes (*mod* 3).

PORS properties

There are **three categories of problem** in the PORS problem.

- For $n = 3k$ nodes the answer is 2^k and the is a **unique optimal tree**.
- For $n = 3k + 1$ nodes the answer is $9 \cdot 2^{(k-3)}$ and there are $4 \cdot \binom{k-1}{2}$ **optimal trees** (quadratically many).
- For $n = 3k + 2$ nodes the answer is $3 \cdot 2^{k-1}$ and there are $2 \cdot k$ **optimal trees** (linearly many).



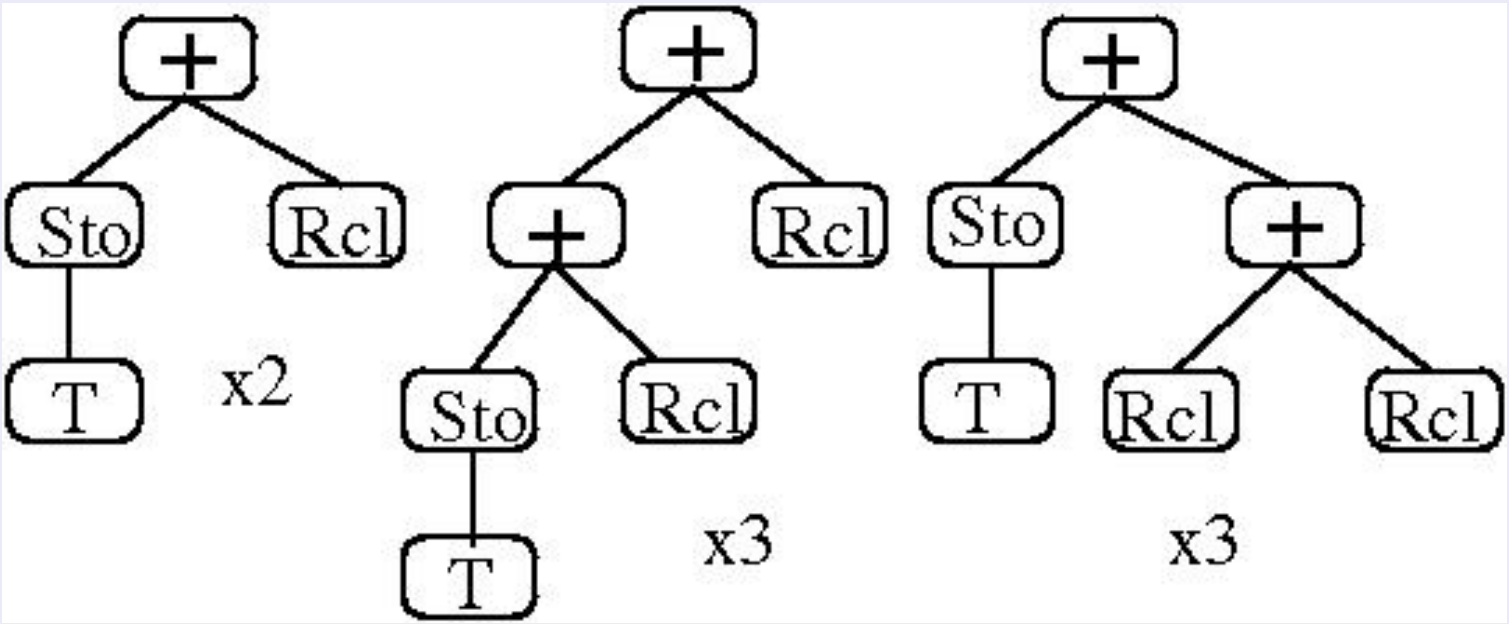
The optimal tree for PORS 6

The $n = 3k$ problems are quite hard, the $n = 3k + 1$ are not too difficult, and the $n = 3k + 2$ are of intermediate difficulty. The optimal answers are known for all n.

The side effect in PORS

The unique feature of PORS is that it requires the system to learn to correctly use the memory. The **Sto** operation and **Rcl** terminal affect the contents of the memory. Unambiguous use of the memory requires the following semantics:

- Left subtrees of an addition operation, including all their memory operations, are executed before right subtrees.
- The memory is loaded with zero before any parse tree is executed.



The three tree fragments above are "building blocks" that make up most of any optimal PORS tree.

Single parent genetic programming

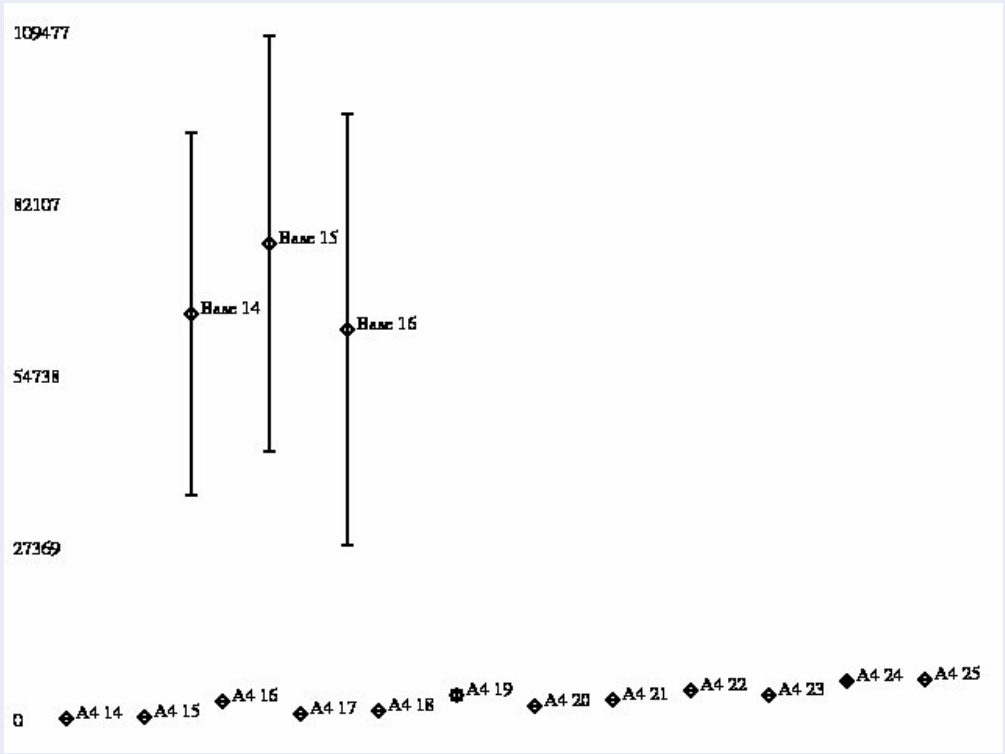
Mutation operators for parse trees can be designed not to contribute to bloat, but subtree crossover, since subtree sizes may mismatch, opens the door to uncontrolled growth of tree size.

Single parent genetic programming modifies standard genetic programming as follows. Population members do not undergo crossover with one another. A population of **ancestors** is designated. When a population member would normally undergo crossover with another population member, it instead does so with one of the ancestors. The ancestor does not change.

Reference

Wendy Ashlock and Daniel Ashlock, **Single Parent Genetic Programming** in Proceedings of the 2005 Congress on Evolutionary Computation, Vol 2, pages 1172-1179.

Single parent genetic programming for PORS



n	mean base	mean SP	ratio
14	65,000	667	**97.5**
15	76,300	880	**86.7**
16	62,600	3330	**18.8**
17	1,420,000	1400	**1010**
18	8,758,000	1850	**4730**
19	241,000	4320	**55.8**

The improvement in time to solution is large . Note that for $n = 18$ a just under **5000-fold** improvement occurs.

The above left shows **95% confidence intervals** for mean time-to-solution for the baseline PORS problem for $n = 14, 15$, and 16 as well as the results of using single parent genetic programming with the ancestor set

$$A = \{\textbf{(+ (Sto (+ (Sto (+ (Sto (+ 1 1)) Rcl)) Rcl)) Rcl)}\}$$

for $n = 14, 15, \ldots, 25$. Baseline runs are labels "Base N", single parent runs are denoted "A4 N".

Single parent results for odd for parity

n	mean base	mean SP	ratio	base Fail	SP Fail
3	29292	14945	1.96	0.5%	0.5%
4	305710	116583	2.62	48.5%	17.5%
5	477197	210780	2.26	96.0%	31.0%

The **improvement in time to solution here is near 2-fold**, far less impressive than PORS, but the system's rate of failing to find a solution at all is much better for single parent techniques. Notice that the rate at which the algorithm **failed to find a solution** also dropped substantially.

The ancestor set used consisted of **copies of evolved solutions** for $n-1$ variables with the variable names randomly changed.

Comments on Single Parent Techniques

- The choice of the ancestor set permits you to embed **domain knowledge** into the system.
- The information (building blocks, whatever) in the ancestor set **cannot** be lost, solving an extant problem in standard genetic programming.
- The technique could be used in **any** evolutionary computation system to prevent loss of information.
- Mean tree size in the population is directly proportional to ancestor tree size: single parent techniques **probabilistically discourage bloat**. The new trees swapped in have bounded size - the size of the ancestor.
- There is a potential for an **apotheosis operator**. Exceptionally fit population members of reasonable size could be promoted to ancestor status.

Linear genetic programming representations

Context free grammar genetic programming(CFG-GP) uses a string of production rules over a context free grammar (CFG) as a linear representation to specify a parse tree. Some additional semantics are required:

- When traversing a string of rules, **always expand the leftmost appropriate nonterminal**.
- When the string of rules is finished, **resolve all remaining nonterminals**, e.g. by applying rule 4.

A simple CFG for the PORS language:

Nonterminals: S

Terminals: +, 1, Rcl, Sto

Rules:

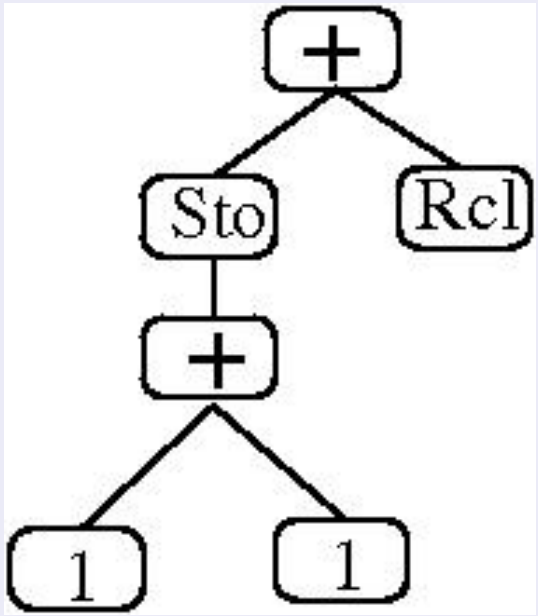
1) S$\rightarrow$ (+ S S)
2) S$\rightarrow$ (Sto S)
3) S$\rightarrow$ Rcl
4) S$\rightarrow$ 1

Reference

J. J. Freeman A Linear Representation for GP using Context Free Grammars, in *Genetic Programming 1998: Proceedings of the Third Annual Conference*, PP 72-77, 1998.

A CFG-GP example

If we start with **S** and apply rules 121443 we get the optimal PORS solution for six nodes.



Recall:
1) S→ (+ S S)
2) S→ (Sto S)
3) S→ Rcl
4) S→ 1

Notice that by using fixed-size strings, CFG-GP permits us to **use a standard string based evolutionary algorithm to do genetic programming**. The system is intrinsically bloat immune. Because of the "expand the leftmost appropriate non-terminal" rule, string crossover operators act like "multi-point" subtree crossovers. Not better, but very different.

Comments on CFG-GP

- Adding additional nonterminal symbols and production rules to accommodate them allows evolution to change the order of expansion in spite of the "leftmost first" rule. In essence such added rules leave room for self organization of the developmental biology induced by the rules.
- More complex grammars can be designed that permit the encoding of domain knowledge, e.g. a double Sto is useless or every terminal before the first Sto should be a 1, every terminal after should be a Rcl.
- It is possible to evolve the grammar itself to gain enormous expressive power.
- Several different researchers and groups have independently discovered some version of this idea.

Linear genetic programming representations

This form of genetic programming uses a linear collection of imperative statements that manipulate the contents of variables or have side effects. The structure evolved is very similar to a traditional procedural program. The languages can involve **goto** and **if** statements and so have flow of control similar to standard programs. Some of the linear GP representations are demonstrably **Turing complete**, which may not be a good thing.

Some examples

D. Ashlock and M. Joenks, **ISAc Lists: A Different Program Induction Method**, in *Genetic Programming 1998, Proceedings of the Second Annual Conference on Genetic Programming*, PP 18-26, 1998.

M . F. Brameier and W. Banzhaf, **Linear Genetic Programming**, Springer, 2007.

Now, this week's paper.

Partitioning Regression with Function Stacks

0) sub	L1	L4
1) add	L6	5.0033
4) add	L11	L6
6) sub	L8	L7
7) wav	L11	L8 wt 0.5003
8) mlt	4.9974	X1
11) add	-1.9944	X1

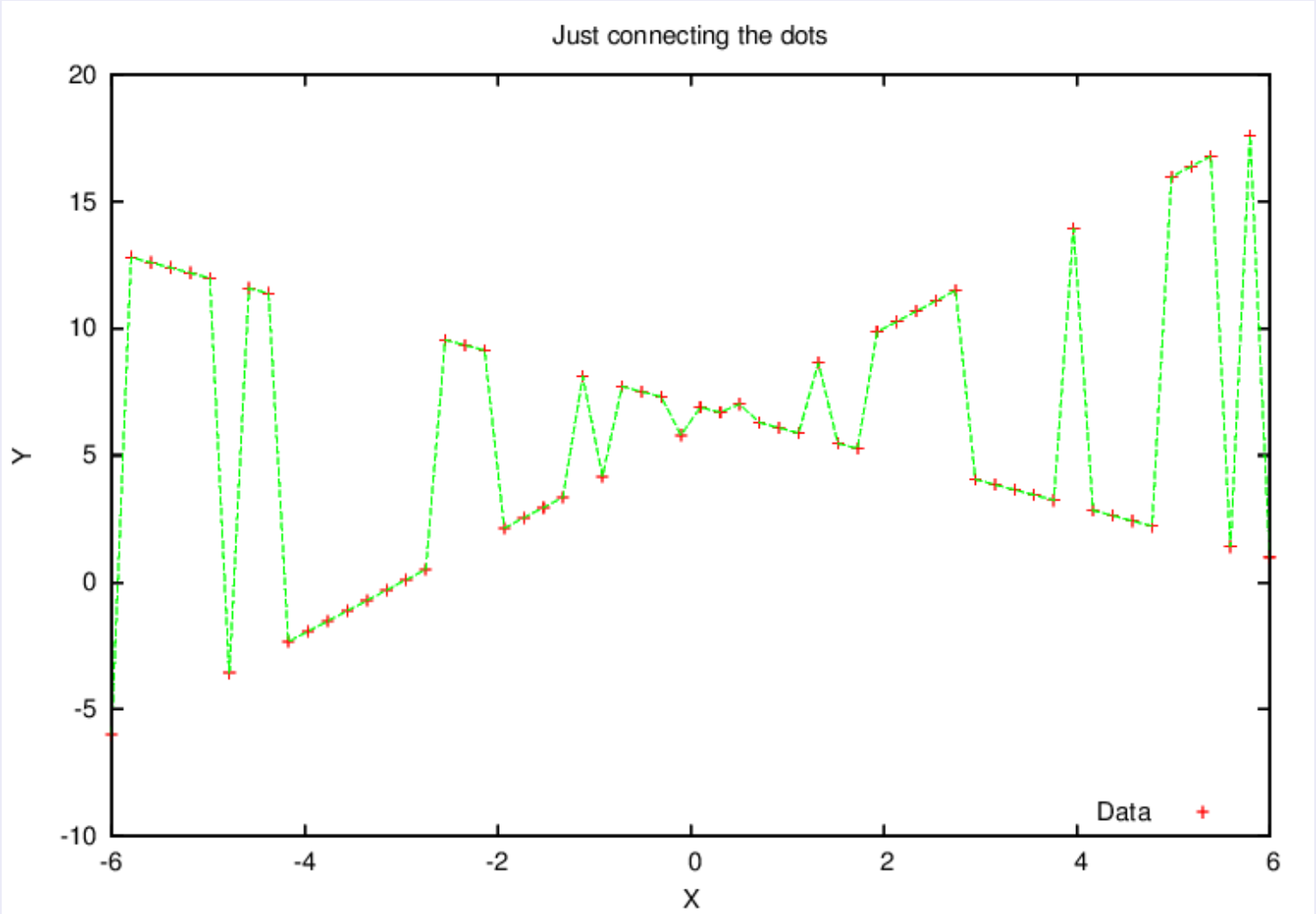
This is the tabular form of a function stack of the sort used to perform partitioning regression.

Available operations and terminals

Operation	Arguments	Definition
Neg	1	Negate argument
Scl	1	Scale argument by ephemeral constant
Sqt	1	Square root of argument
Sqr	1	Square of argument
Sin	1	Sine of argument
Cos	1	Cosine of argument
Atn	1	Arctangent of argument
+	2	Addition
-	2	Subtraction
*	2	Multiplication
Pro/	2	Protected division$< 1E \pm 6$
Max	2	Maximum of arguments
Min	2	Minimum of arguments
Wavg	2	Weighted average using positive decimal of ephemeral constant as weight for first argument

The function stack has 12 nodes, but only those that are relevant to the first node in the function stack are displayed. This is called **object compilation** (even though no compilation takes place). A **node** contains an operation and its arguments possibly including an ephemeral constant, nodes farther down the list, and input variables.

An Example of the Problem



The data are drawn at random from two lines; a connect the dots model is not a simple, or particularly helpful, mathematical model.

Partitioning Regression with Function Stacks

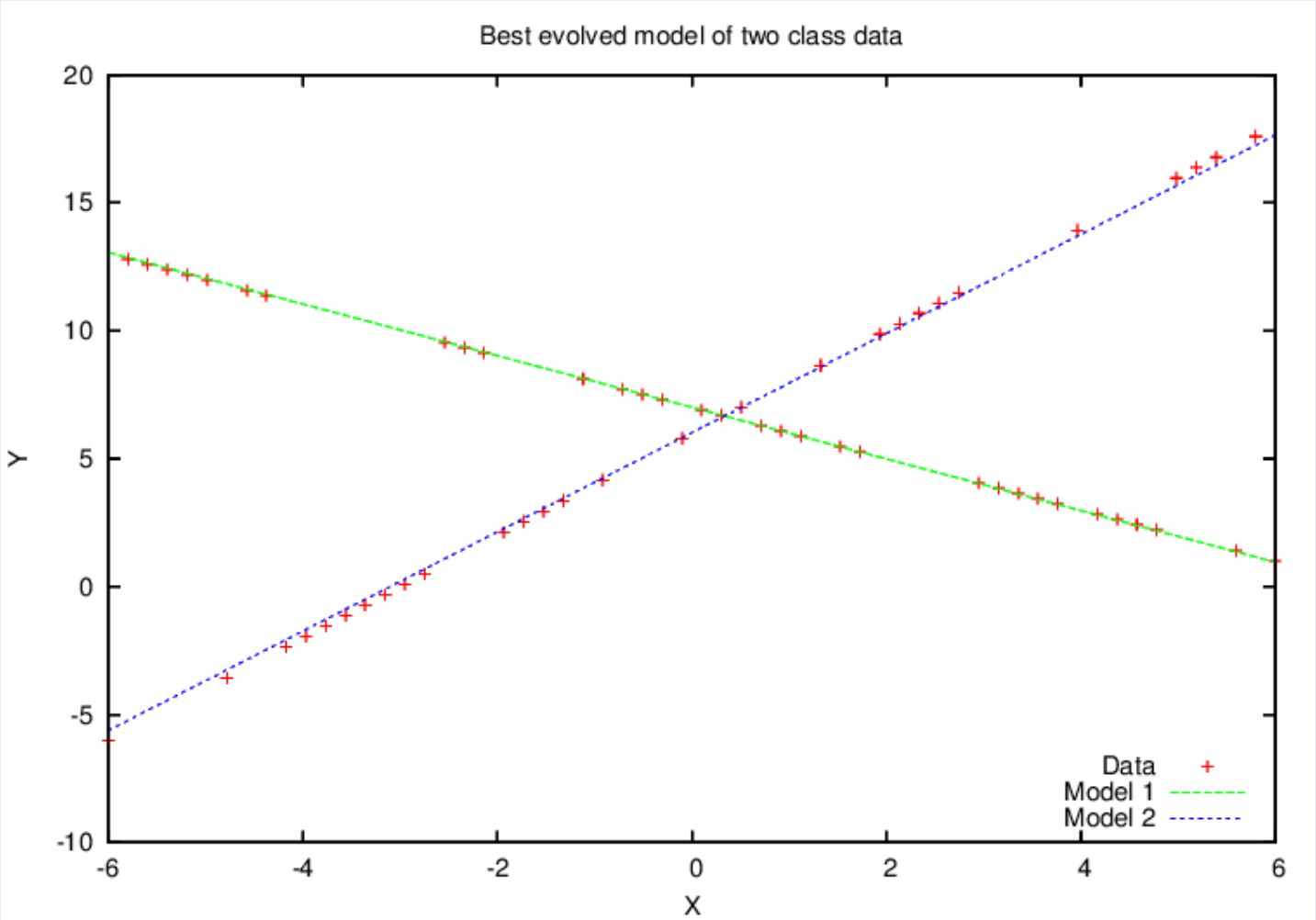
While we normally read the output of a function stack from the lowest index node, **the output of any node may be used as a model of the data**. This means that we have several models with access not only to the same data, but one another.

The fitness function used to perform partitioning regression with is based on the same minimum error assignment used to partition data. Suppose that e_i^k is the error $y - y_{pred}$ between the data and its predicted value for data point i using model k. Then the fitness of a multi-model for n data points, encoded as a function stack, is:

$$fitness = \sum_{i=1}^{n} min\left((e_i^0)^2, (e_i^1)^2, \ldots, (e_i^k)^2\right) \quad (1)$$

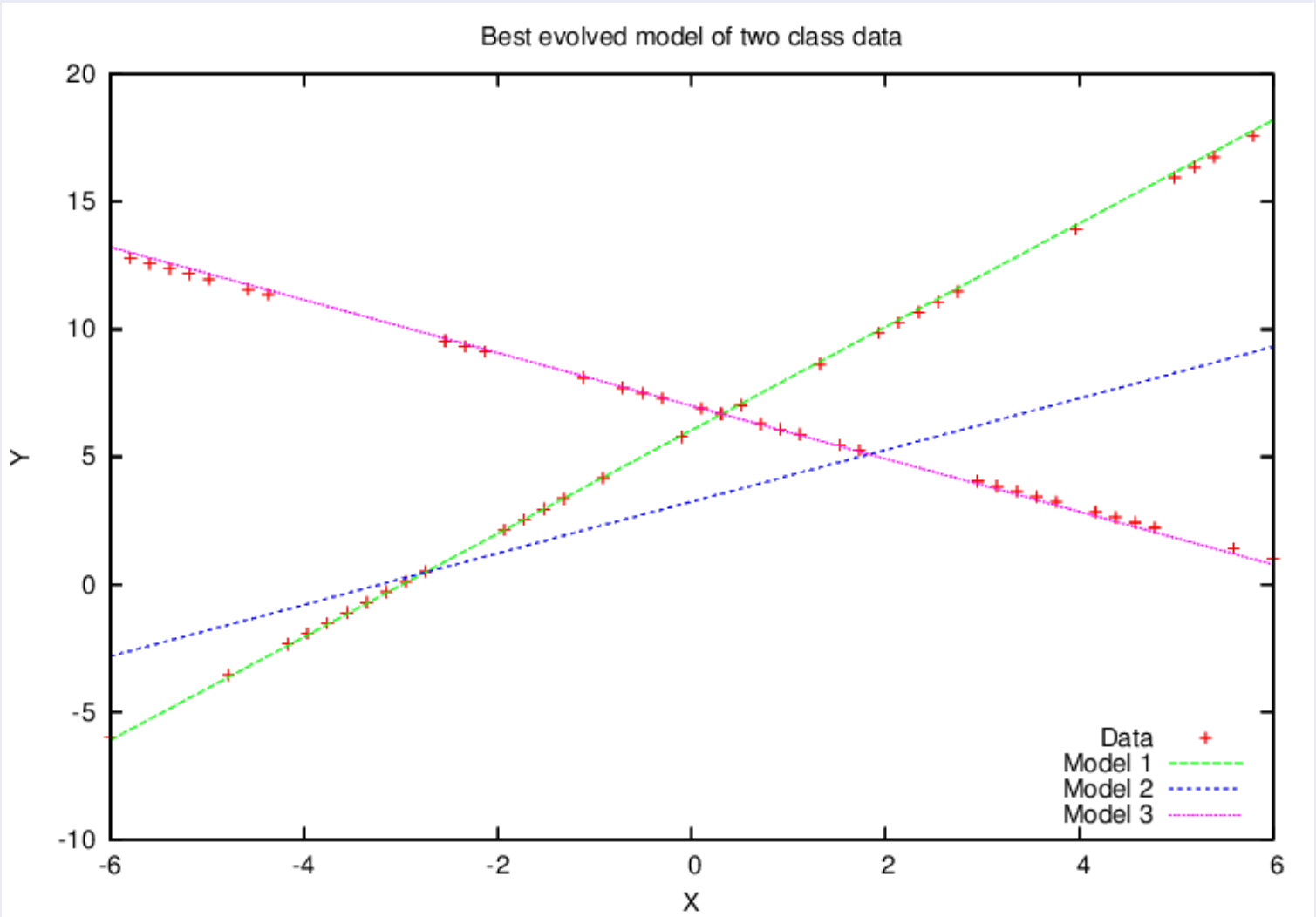
The sum of minimal squared error, minimizing across the available models. This is called the **best squared error** (BSE) function for multiple models.

A Solution to Having Two Data Models



These are the data drawn at random from two lines again; using best squared error to evolve a function that the output of node zero (Model 1) is the green line, the output of node 1 (Model 2) is the dotted blue line.

A Solution with Three Data Models

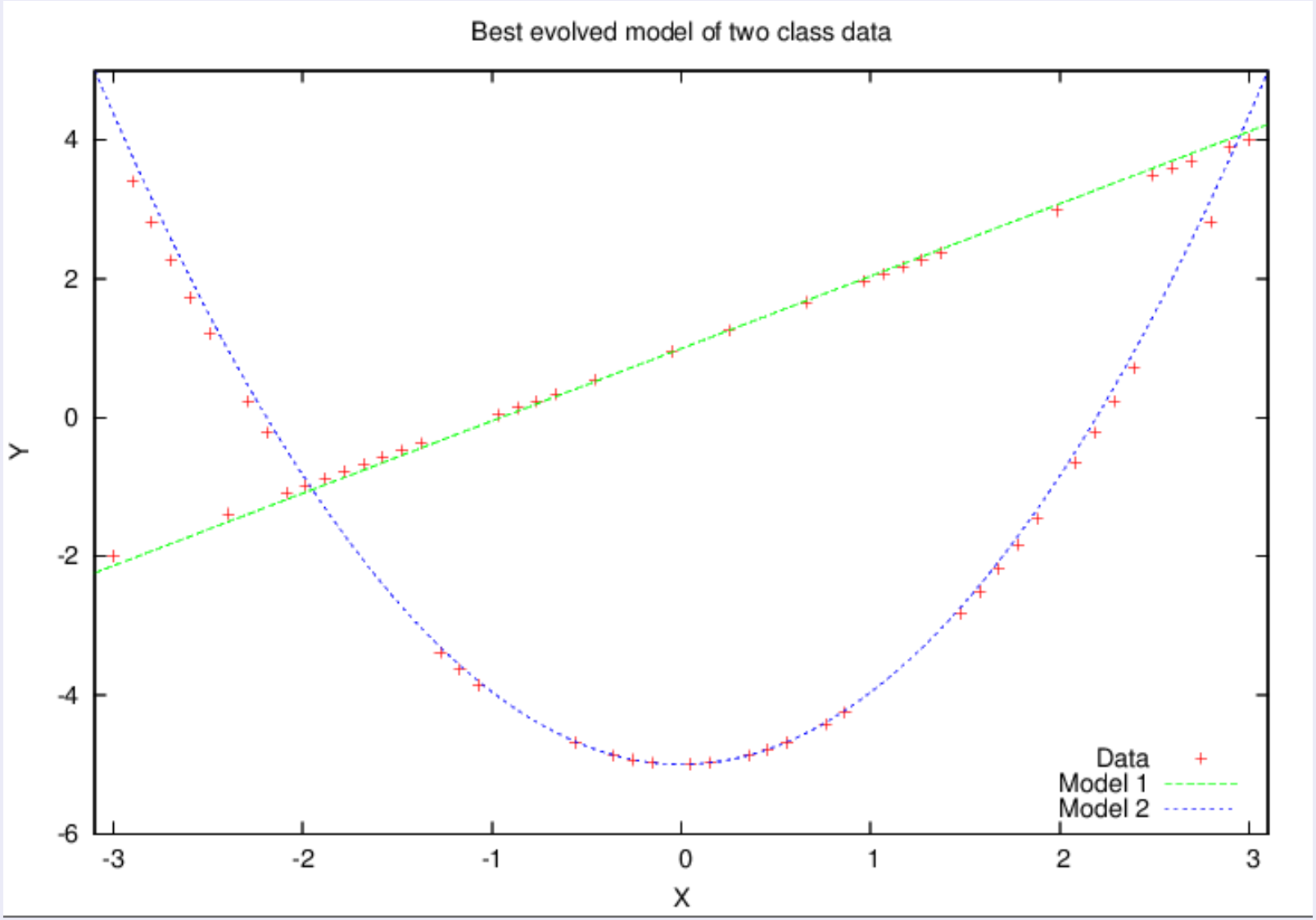


Too many models? For this data at least, not a problem. The third line "captured" one point...

Some Perspective...

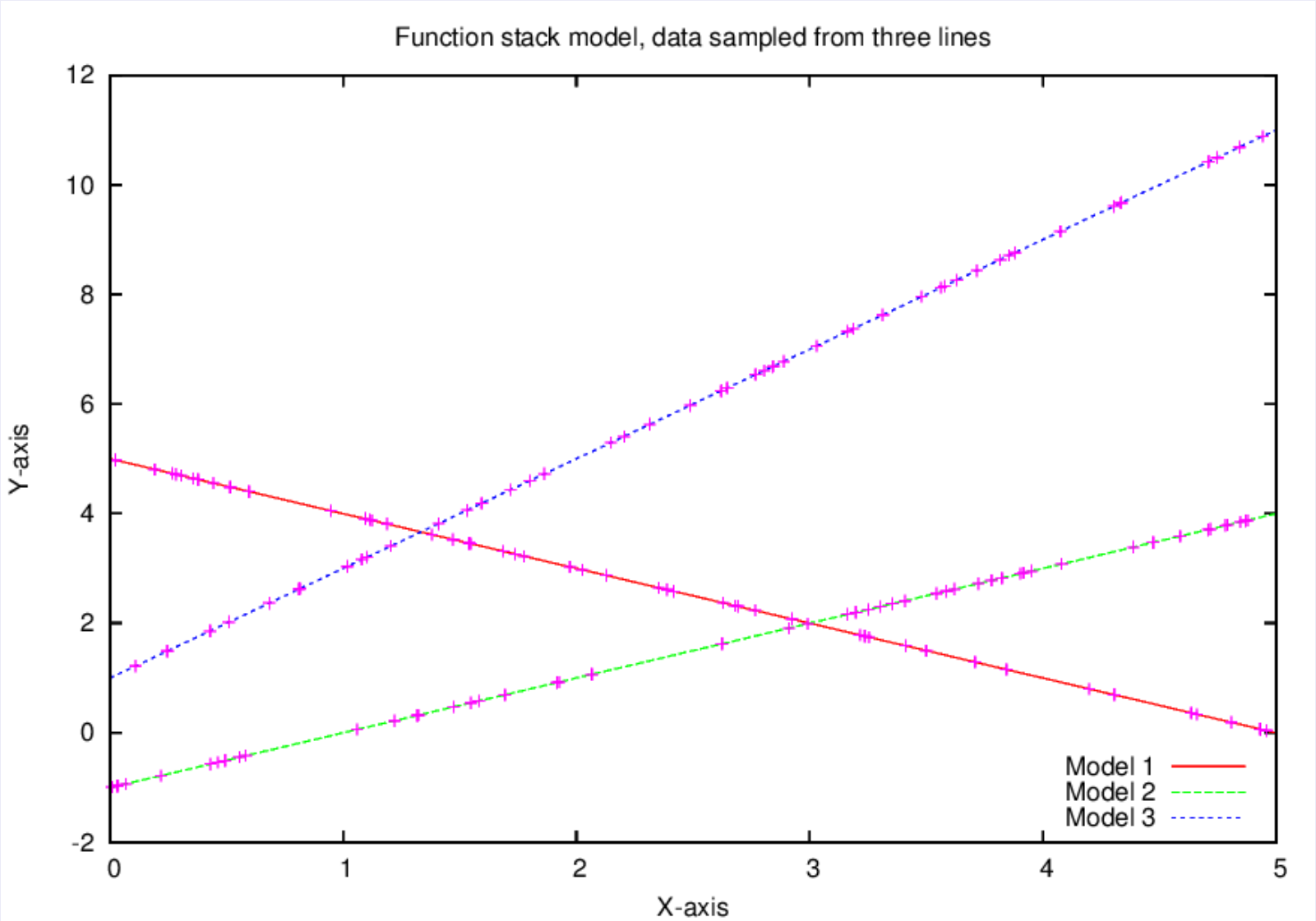
- If we categorize points by the line they belong to, we get a clustering of the data.
- If we find k models, then mapping the points onto their error for each models transforms the data so that the clusters are very obvious.
- At present models can share nodes in the function stack quite easily. This can be changed.
- At present the ephemeral constants are not optimized except by being occasionally changed by mutation. A constant optimization step is easy to add.
- It is potentially easy to add extinction operators to remove models that are capturing too few points - which would permit approximation of the correct number of models, so long as we started with too many.

Harder Than Lines



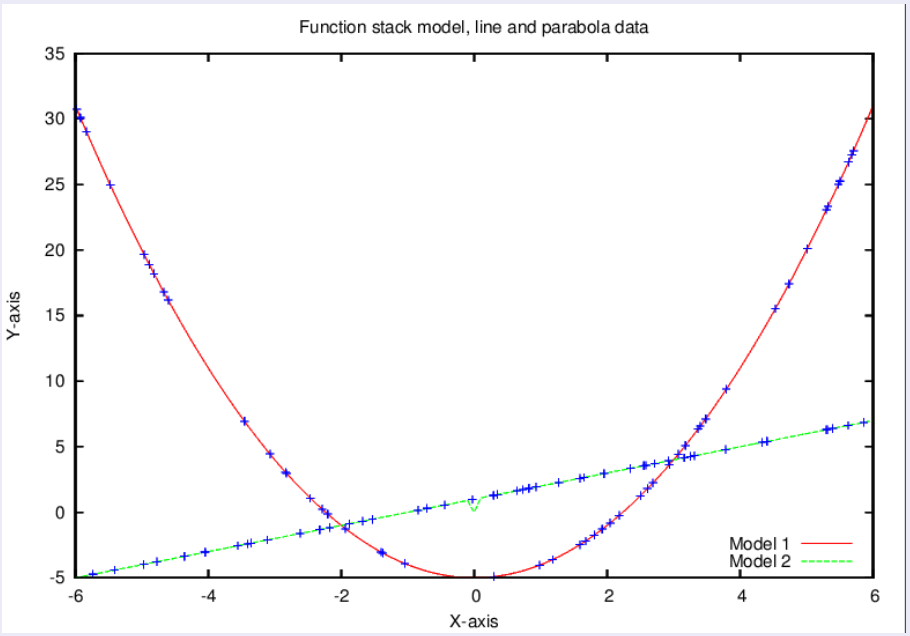
Another demonstration of two models working well to partition points.

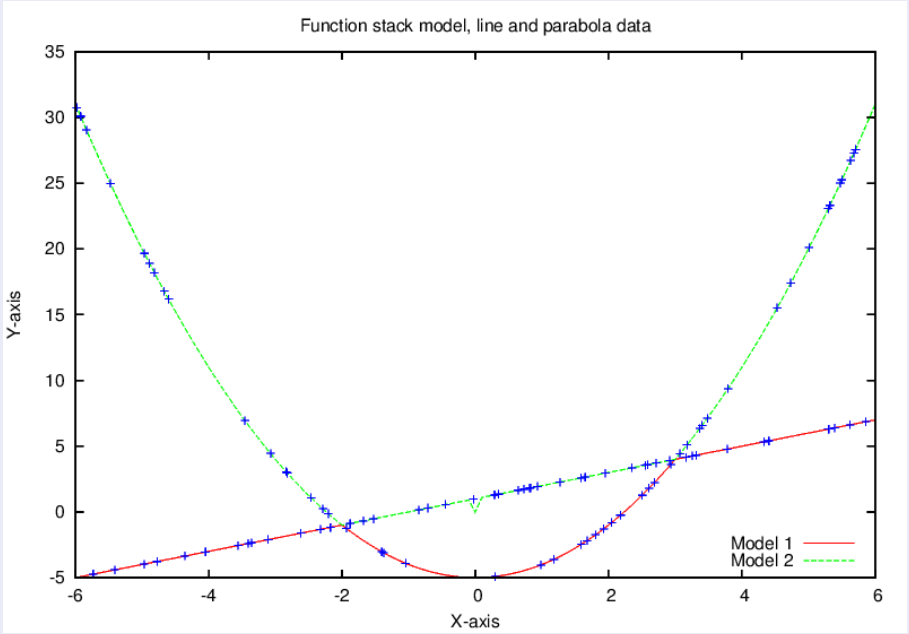
More Lines



This shows that we can do samples from three different lines.

Evil Twin Studies I





The function stack can **generate an original point of view**. The left hand model is a perfectly good partition of points drawn from the parabola and points drawn from the line. Following line color, the second picture did something fairly different. Its hard to say this is "wrong" but it is very different.

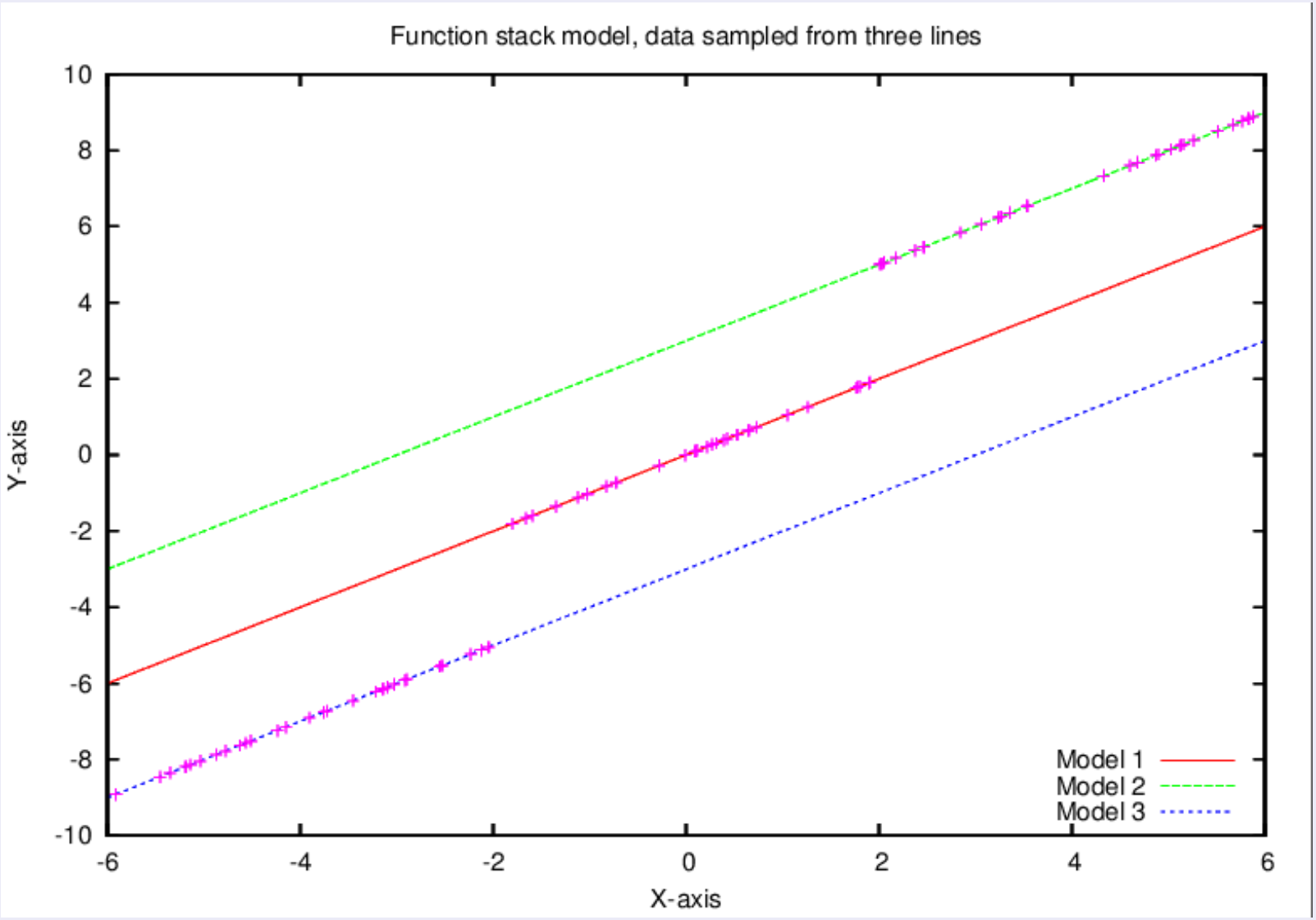
An Assumption About Model Based Partitioning

Something that came up during Joey's (Dr. Joseph Brown's) doctoral research was a paper where the authors were trying to fit a function to single variable data under the assumption that the source of the data changed at various points. Their solution was to evolve change points and models that held before the first, between, and after the last change point. Problems with this include:

- This assumes a very neat serial character to the change of models.
- What is the generalization to multivariate data? Boundaries in one dimension are points. Boundaries in more than one dimension are ?Jordan curves? on a good day. More dimensions? Holy collection of Etruscan snoods, Batmat!

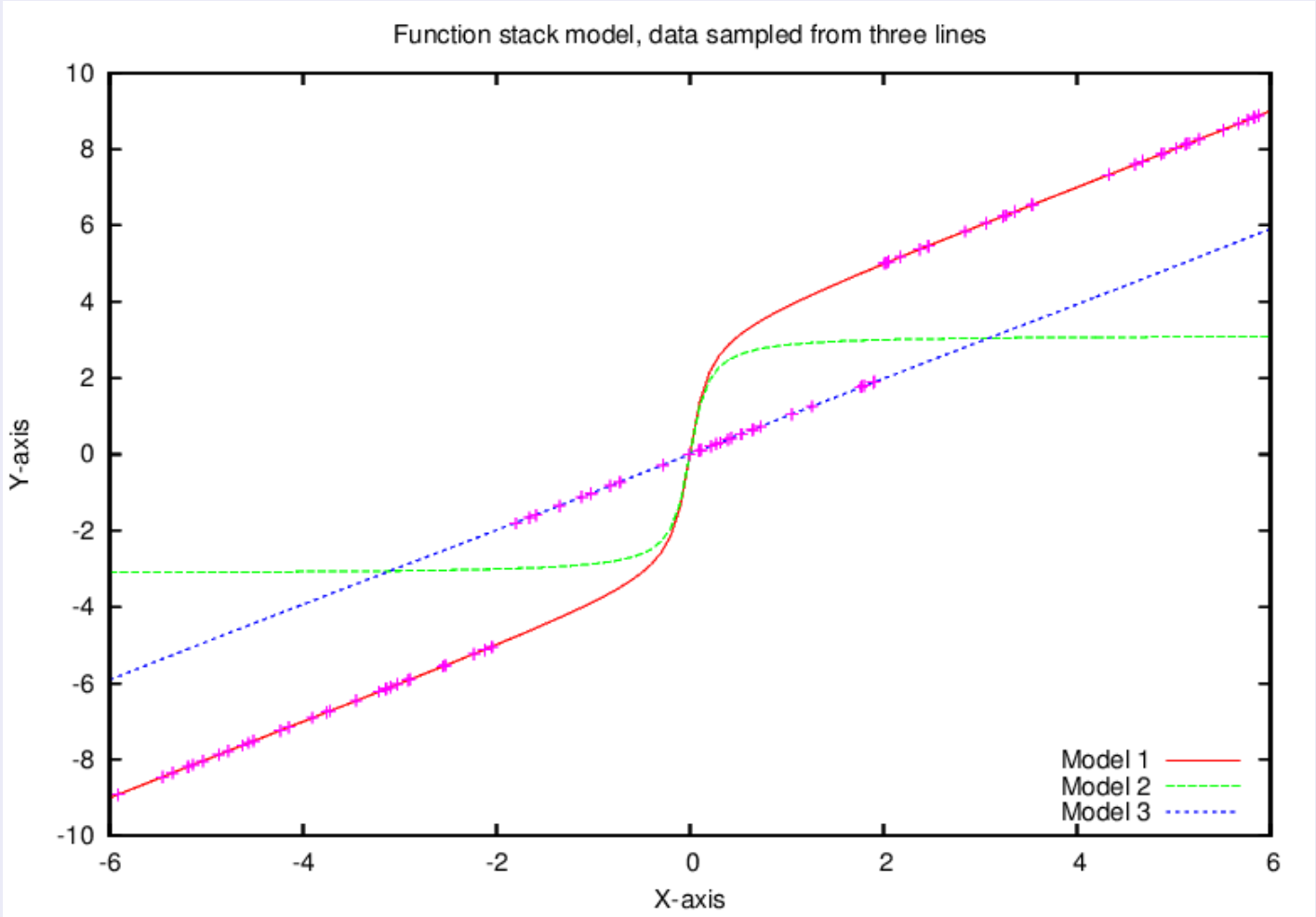
Anyhow we also tried data with the sort of neat x-coordinate partitioning.

Three chunks of Data



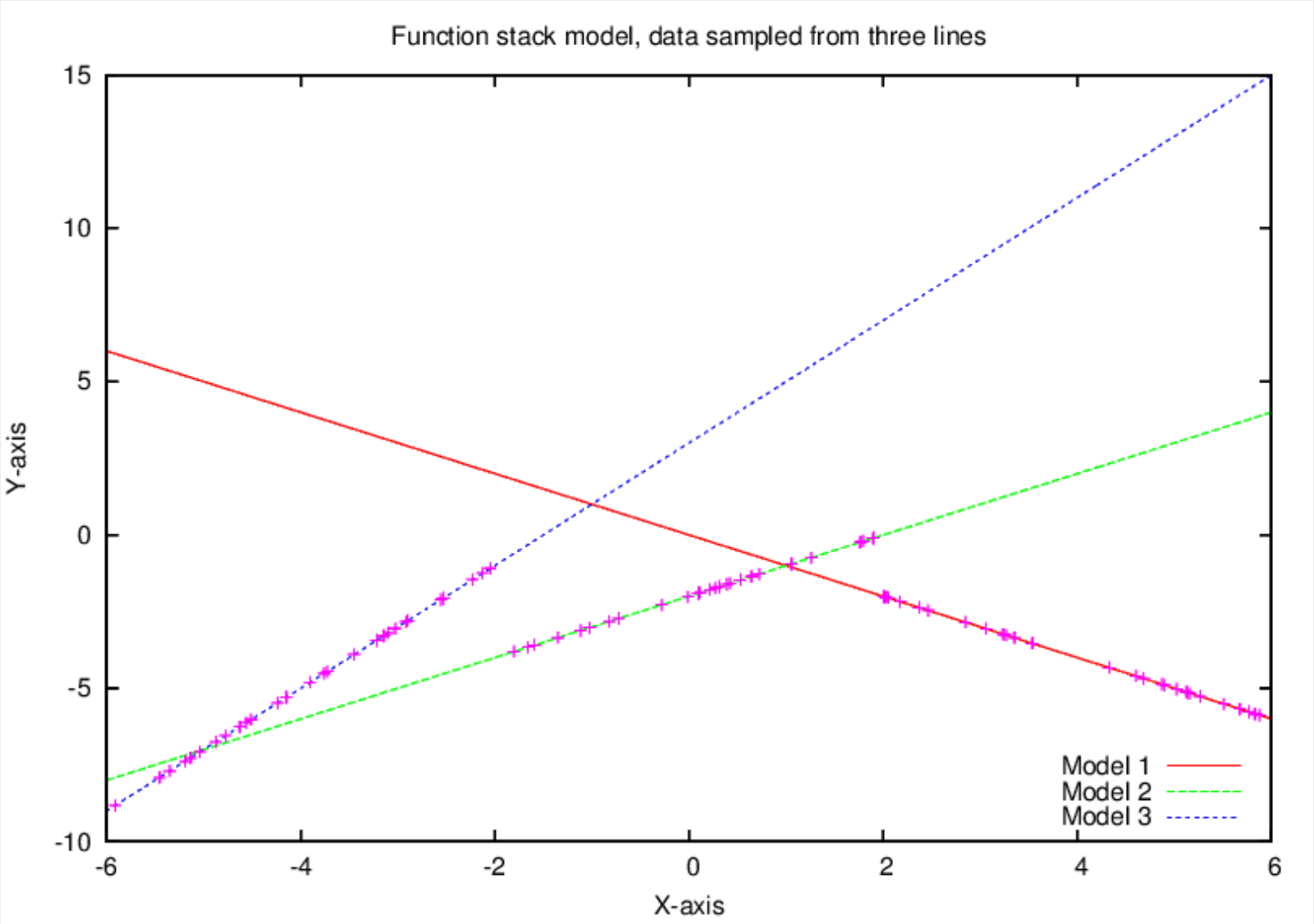
This shows that we can do samples from three different lines when the data a serially partitioned.

Evil Twin Studies II



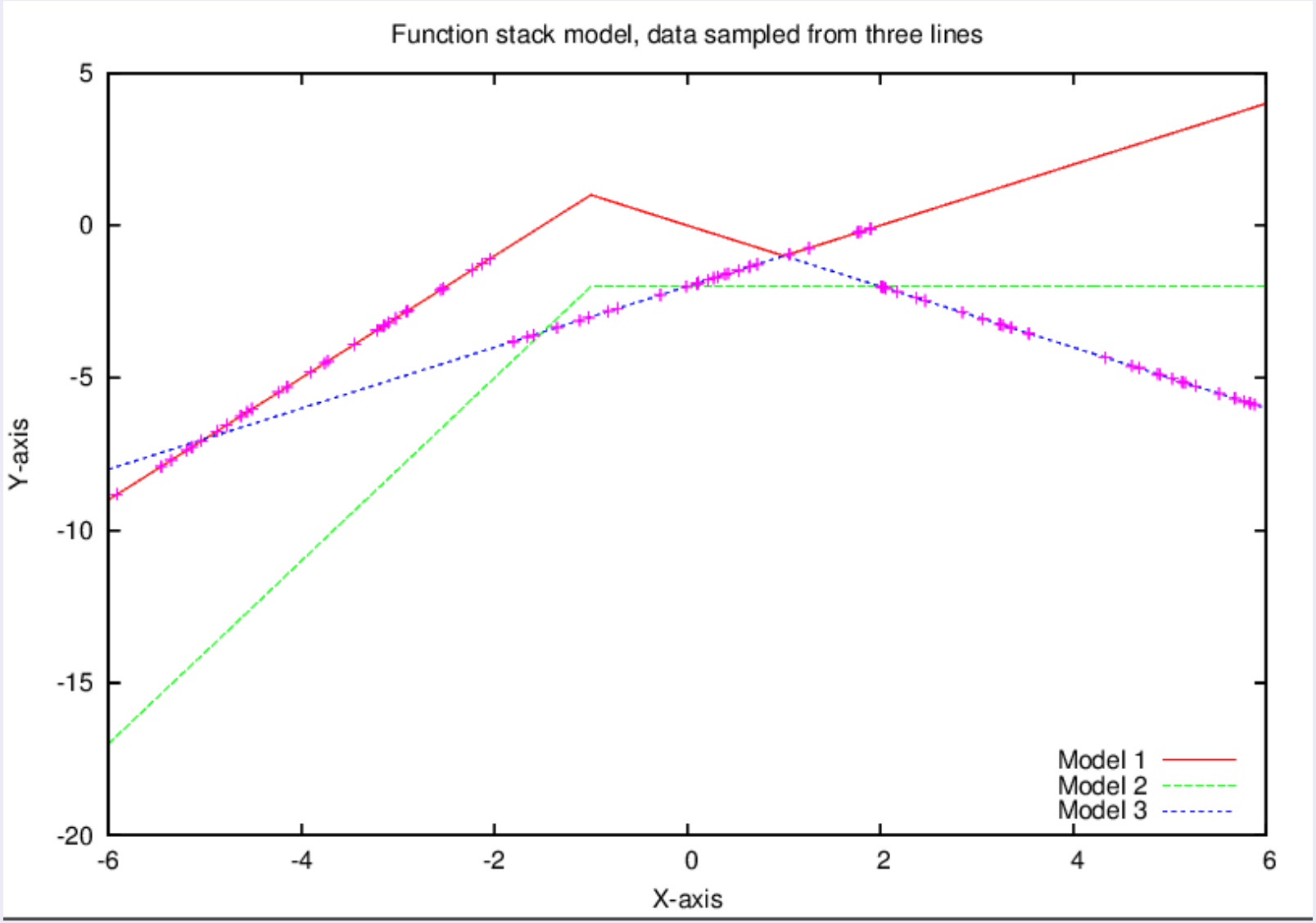
Another very original point of view. This function stack got one of the lines the data was drawn from bang-on, an arctangent model that hit at most one point, and an arctangent-plus-linear trend that got two of the sets of data. **Perhaps the data should not have been drawn from parallel lines?**

Three chunks of Data



This shows that we can do samples from three different lines when the data a serially partitioned and lacking parallel slope.

Evil Twin Studies III



sigh

Lots to Do

- Implement extinction.
- Cluster real data. In more dimensions.
- Uses partitioning regression as associators (Amanda, Jen).
- Play with the operations available to the function stacks.
- Play with where the outputs are taken from - currently all adjacent.
- Play with degree of isolation of the formulas. Lots of possibilities.
- Add ephemeral constant optimizer. PSO? EA?

Many Thanks

NEXT WEEK: Jenifer Garner on Gasses, this time I promise.

Thanks to the *Natrual Science and Engineering Research Council of Canada* and the *University of Guelph* for support of this work.