Carnegie Mellon University

MSE

Data Analytics for Materials Science

A.D. (Tony) Rollett, Richard LeSar, Jacob Hochhalter (Univ. of Utah)

Dept. Materials Sci. Eng.

Carnegie Mellon University

Symbolic Regression

Revised: 25th Apr., 2021

Outline

The main objective of these notes is to introduce you to the use of *symbolic regression* in materials science.

Neural nets are a powerful tool for analysis and building regression relationships. However, one could say that they are limited to linear algebra except for the inclusion of rectification and related operations in neural nets. Sometimes, one wants to allow for the development of more complex mathematical relationships.

The main objective symbolic regression is to discover the best functional form (i.e., mathematical equation) that fits your data.

Genetic programming

https://gplearn.readthedocs.io/en/stable/intro.html

- Symbolic regression is actually a subset of a larger scope known as *genetic programming*.
- The objective, just as in linear regression, is to fit a dataset.
- As with any regression, a *cost function* or *error function* must be defined. A commonly employed measure is RMSE (Root Mean Square Error). Other names include ***fitness*** (which acknowledges Darwin in the use of an evolving solution), *score*, *error*, and *loss*.

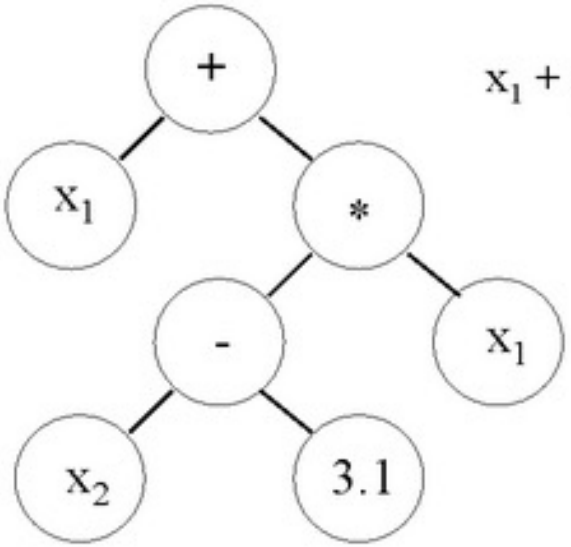
  'mean absolute error' is the magnitude of the error
  'mse' for mean squared error
  'rmse' for root mean squared error
  'pearson', for Pearson's product-moment correlation coefficient
  'spearman' for Spearman's rank-order correlation coefficient
- The latter two methods pertain to indirect optimization

Building blocks

- The building blocks for the functional forms are mathematical operator (multiply, divide, add …), analytic functions (exponential, log, power law …), constants (numerical values) and constitutive parameters.
- A *fitness function* must be defined (equivalent to *objective function*) along with a choice or error metric (previous slide).
- The *functional set* comprises all the operators and functions and the *terminal set* comprises all the constants and constitutive parameters.
- As the fitting proceeds, an evolutionary algorithm is used to explore different combinations of building blocks. This is, of course, computationally expensive.

Formulae vs. Trees

- A common approach to visualizing genetic programming is to depict an expression (as opposed to an equation) in the form of a *tree*.
- In fact, this is more significant than visualization because it illustrates how one can build an expression (formula) in the computer.
- In Roehrig's example, one starts with the binary operator "+" and then moves down a level to add e.g., a constant and another operator such as multiply, etc.



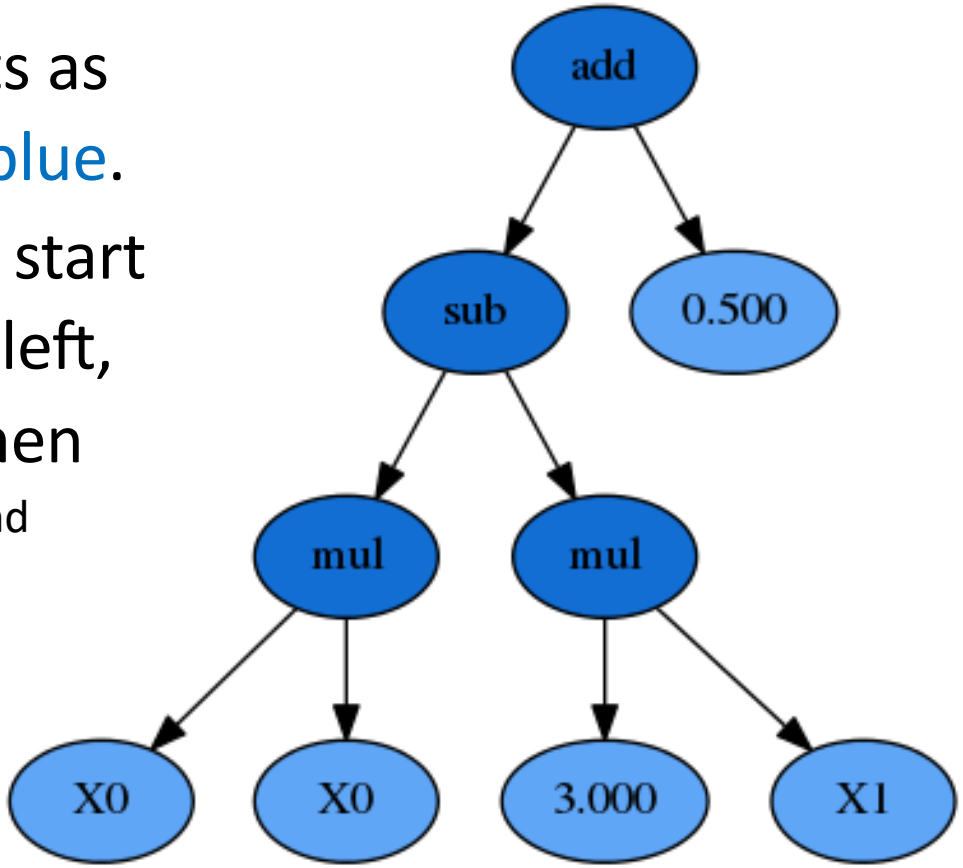
$x_1 + ((x_2 - 3.1) * x_1)$

http://www.contrib.andrew.cmu.edu/~roehrig/CompFin/Week4/symbreg.htm

Syntax Tree

Alternatively, a *syntax tree* has *functions* as interior nodes, dark blue, and variables & constants as leaves (terminal nodes), light blue.

To interpret the tree, one can start with the LH leaves. At bottom left, multiply X0*X0. Then 3*X1. Then apply the subtraction (next, 2nd level up). Finally add 0.5 (top node).



Arity

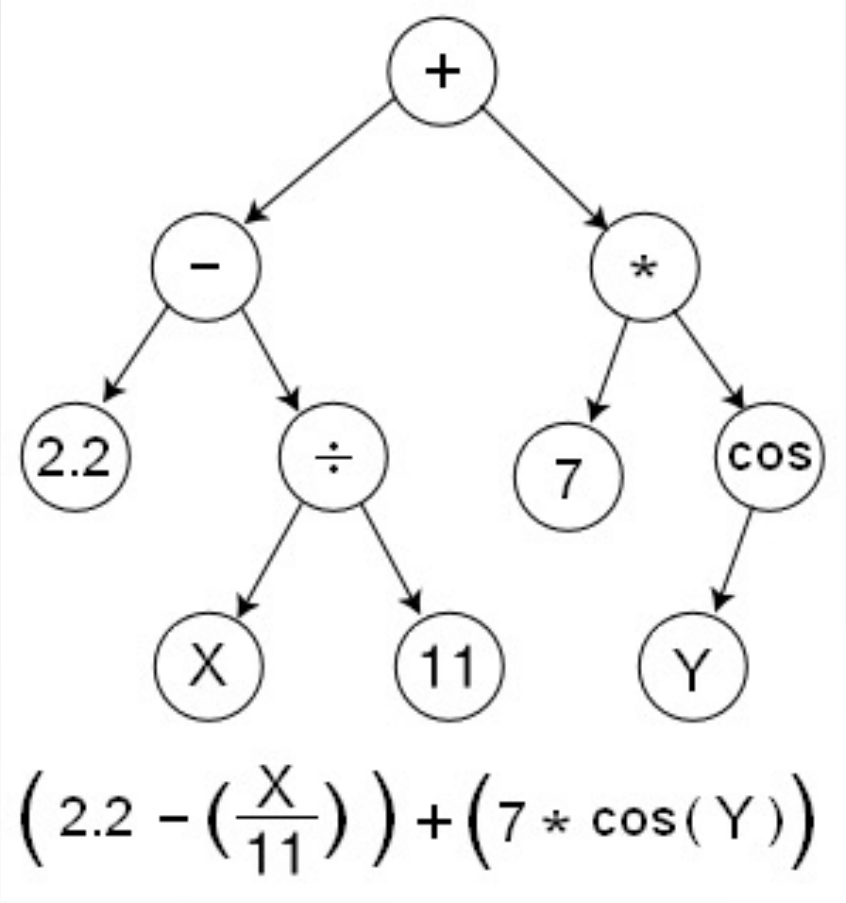
Each function has an "arity", which just means the number of arguments that it takes. The arithmetic operators have *arity=2*. As another example, taking the absolute value has *arity=1*.

Standard (default) set shown on the RHS.

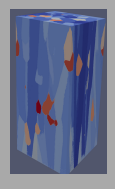
You can select which functions you want to use.

```
‘add’ : addition, arity=2.
‘sub’ : subtraction, arity=2.
‘mul’ : multiplication, arity=2.
‘div’ : division, arity=2.
‘sqrt’ : square root, arity=1.
‘log’ : log, arity=1.
‘abs’ : absolute value, arity=1.
‘neg’ : negative, arity=1.
‘inv’ : inverse, arity=1.
‘max’ : maximum, arity=2.
‘min’ : minimum, arity=2.
‘sin’ : sine (radians), arity=1.
‘cos’ : cosine (radians), arity=1.
‘tan’ : tangent (radians), arity=1
```

- In a more complicated example:

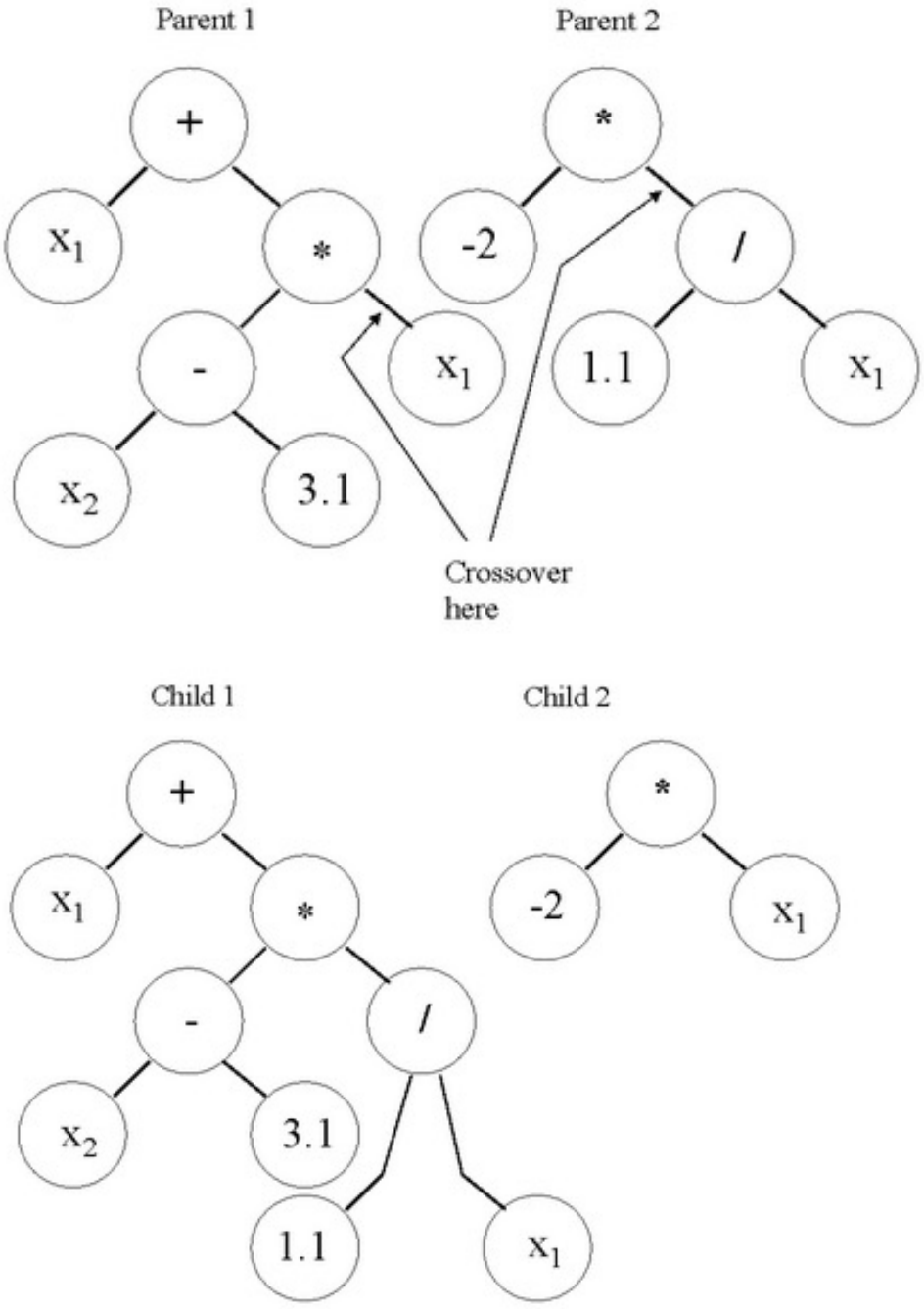


https://en.wikipedia.org/wiki/Symbolic_regression



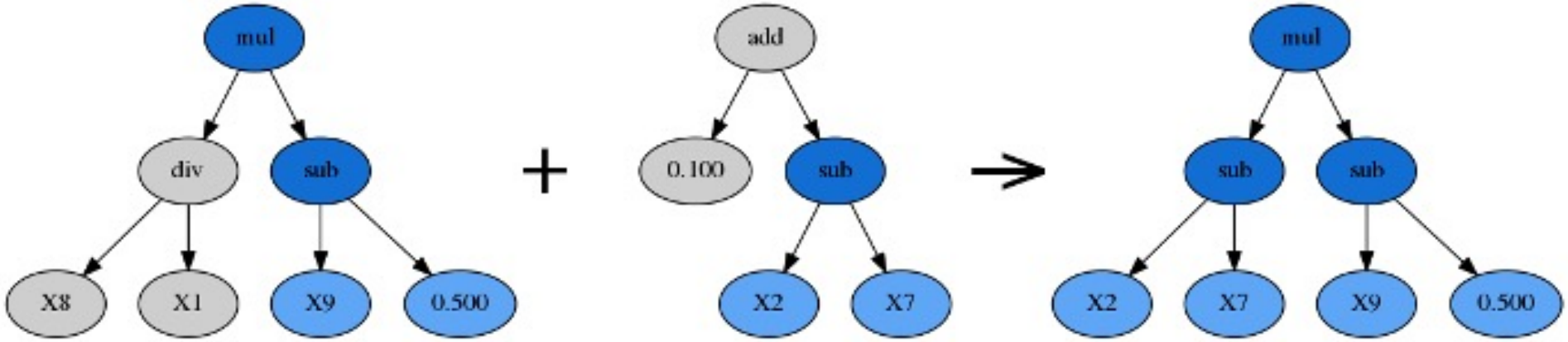
Multiple Trees, Mixing & Matching

- The easy part to understand is the use of multiple trees, each of which is used to compute a set of predicted values, from each of which one obtains an error estimate. This, of course, determines the *fitness* of each tree. There is a *population* of trees.
- The harder part to understand is the swapping of elements and sub-structures between trees. This is, however, exactly, how genetic programming works, i.e., by mixing together elements (genetic material, if you like) between different members of the *population*.



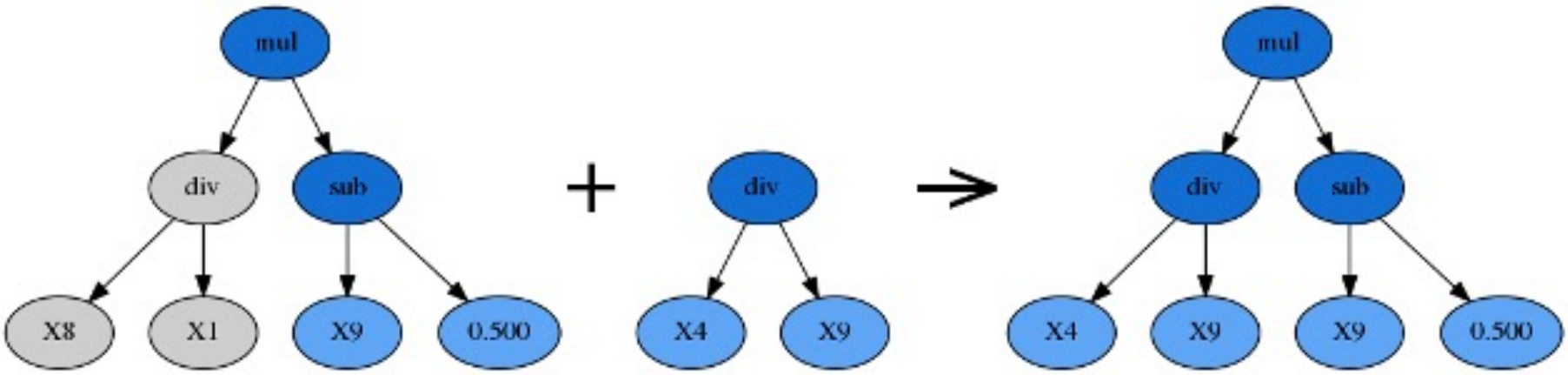
Crossover

Quoting from gplearn: "Crossover is the principle method of mixing genetic material between individuals and is controlled by the p_crossover parameter. Unlike other genetic operations, it requires two tournaments to be run in order to find a parent and a donor.
Crossover takes the winner of a tournament and selects a random subtree from it to be replaced. A second tournament is performed to find a donor. The donor also has a subtree selected at random and this is inserted into the original parent to form an offspring in the next generation."



Subtree Mutation

Quoting from gplearn: "Subtree mutation is one of the more aggressive mutation operations and is controlled by the p_subtree_mutation parameter. The reason it is more aggressive is that more genetic material can be replaced by totally naive random components. This can reintroduce extinct functions and operators into the population to maintain diversity.
Subtree mutation takes the winner of a tournament and selects a random subtree from it to be replaced. A donor subtree is generated at random and this is inserted into the parent to form an offspring in the next generation."

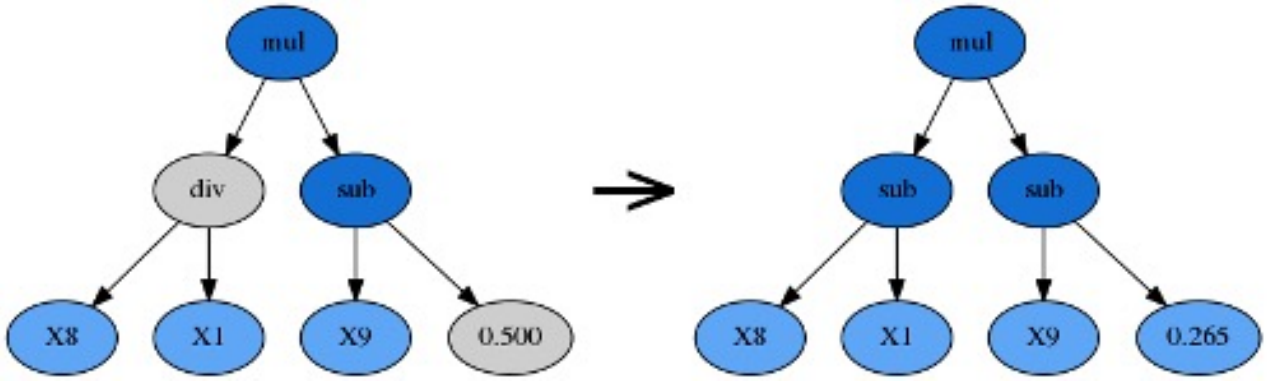


Point Mutation

Quoting gplearn: "Point mutation is probably the most common form of mutation in genetic programming. Like subtree mutation, it can also reintroduce extinct functions and operators into the population to maintain diversity.

Point mutation takes the winner of a tournament and selects random nodes from it to be replaced. Terminals are replaced by other terminals and functions are replaced by other functions that require the same number of arguments as the original node. The resulting tree forms an offspring in the next generation.

Functions and terminals are randomly chosen for replacement as controlled by the p_point_replace parameter which guides the average amount of replacement to perform."



Initialization

The gplearn documentation has useful advice on intialization, https://gplearn.readthedocs.io/en/stable/intro.html.

Consider hyperparameters such as: init_depth, init_method, population_size.

R package

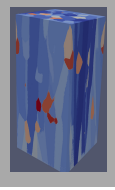
- **symbolicRegression** is available but it is very basic compared to what one can find in python, e.g., NASA/bingo
- **gramEvol** is another package that does include the crucial aspect of *evolution*, i.e., it includes genetic programming. See https://www.r-bloggers.com/symbolic-regression-genetic-programming-or-if-kepler-had-r/
- **rgp** is a genetic programming framework which supports symbolic regression

Symbolic regression has the disadvantage of having a much larger space to search than does standard regression, because not only is the search space in symbolic regression infinite, but there are an infinite number of models which will perfectly fit a finite data set (provided that the model complexity isn't artificially limited).

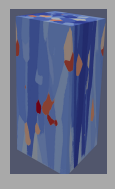
It will generally take a symbolic regression algorithm longer to find an appropriate model and parametrization than traditional regression techniques.

The calculations can be sped up by limiting the set of building blocks provided to the algorithm, based on existing knowledge of the system that produced the data. *In the end, using symbolic regression is a decision that has to be balanced with how much is known about the underlying system.*

paraphrased from: https://en.wikipedia.org/wiki/Symbolic_regression
https://analyticsindiamag.com/how-to-avoid-overfitting-in-neural-networks/



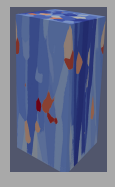
Our first example: Kepler's third law



Between 1609 and 1619, Kepler published his three laws of planetary motion

- The orbit of a planet is an ellipse with the Sun at one of the two foci.
- A line segment joining a planet and the Sun sweeps out equal areas during equal intervals of time.
- The square of a planet's orbital period is proportional to the cube of the length of the semi-major axis of its orbit, i.e., the period is proportional to the maximum distance from the sun to the planet (semi-major axis) to the 3/2 power.

https://analyticsindiamag.com/how-to-avoid-overfitting-in-neural-networks/

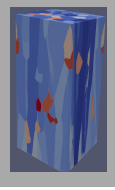


Data:

	planets	distance	period
1	Venus	0.72	0.61
2	Earth	1.00	1.00
3	Mars	1.52	1.84
4	Jupiter	5.20	11.90
5	Saturn	9.53	29.40
6	Uranus	19.10	83.50

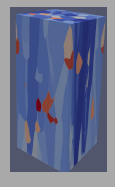
We will use symbolic regression to find the functional relation between period and distance.

https://analyticsindiamag.com/how-to-avoid-overfitting-in-neural-networks/

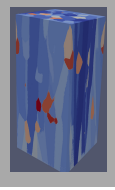


```
install.packages('gramEvol')
library("gramEvol")

# Kepler's third law (distance in AU)

planets <- c("Venus", "Earth", "Mars", "Jupiter", "Saturn", "Uranus")
distance <- c(0.72, 1.00, 1.52, 5.20, 9.53, 19.10)
period <- c(0.61, 1.00, 1.84, 11.90, 29.40, 83.50)
data.frame(planets, distance, period)
```

```
  planets distance period
1   Venus     0.72   0.61
2   Earth     1.00   1.00
3    Mars     1.52   1.84
4 Jupiter     5.20  11.90
5  Saturn     9.53  29.40
6  Uranus    19.10  83.50
```

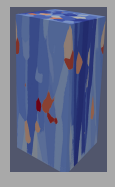
```
# grule is a function from the gramEvol package
▸ruleDef <- list(expr = grule(op(expr, expr), func(expr), var),
▸+                  func = grule(sin, cos, tan, log, sqrt),
▸+                  op = grule('+', '-', '*', '/', '^'),
▸+                  var = grule(distance,distance^n, n),
▸+                  n = grule(1, 2, 3, 4, 5, 6, 7, 8, 9))

# set the grammar from ruleDef
grammarDef <- CreateGrammar(ruleDef)
grammarDef
<expr> ::= <op>(<expr>, <expr>) | <func>(<expr>) | <var>
<func> ::= `sin` | `cos` | `tan` | `log` | `sqrt`
<op>   ::= "+" | "-" | "*" | "/" | "^"
<var>  ::= distance | distance^<n> | <n>
<n>    ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

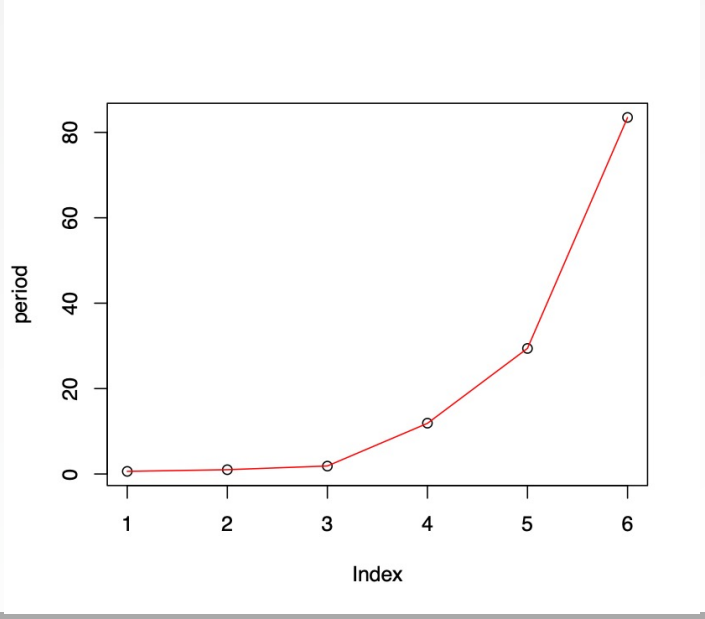
```
# GrammarRandomExpression generates expressions. Maximum depth of recursion in # rules in
grammar
set.seed(134)
GrammarRandomExpression(grammarDef, 6)
[[1]]
expression(log(distance^6))
[[2]]
expression(tan(sin(distance^1)) * tan(distance^4))
[[3]]
expression(3)
[[4]]
expression(cos(((log(sin(distance * (distance/distance)))^distance^6)^distance^3)^2))
[[5]]
expression(9)
[[6]]
expression((distance - distance^1)^distance
```

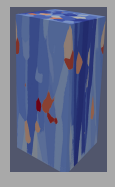
```
SymRegFitFunc <- function(expr) {
 result <- eval(expr)
  if (any(is.nan(result)))
    return(Inf)
  return (mean(log(1 + abs(period - result))))
}
```

$$\sum_{i=1}^{N} \left[\log\left(1 + |\hat{y}_i - y_i|\right)\right] / N$$

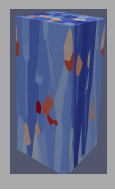


```
> set.seed(262)
> suppressWarnings(ge <- GrammaticalEvolution(grammarDef, SymRegFitFunc, terminationCost
= 0.05))
> ge
Grammatical Evolution Search Results:
  No. Generations:  5
  Best Expression:  sqrt(distance^3)
  Best Cost:        0.0201895728693592

> best.expression <- ge$best$expression
> data.frame(distance, period, Kepler = sqrt(distance^3), GE = eval(best.expression))
  distance period     Kepler         GE
1     0.72   0.61  0.6109403  0.6109403
2     1.00   1.00  1.0000000  1.0000000
3     1.52   1.84  1.8739819  1.8739819
4     5.20  11.90 11.8578244 11.8578244
5     9.53  29.40 29.4197753 29.4197753
6    19.10  83.50 83.4737743 83.4737743
```

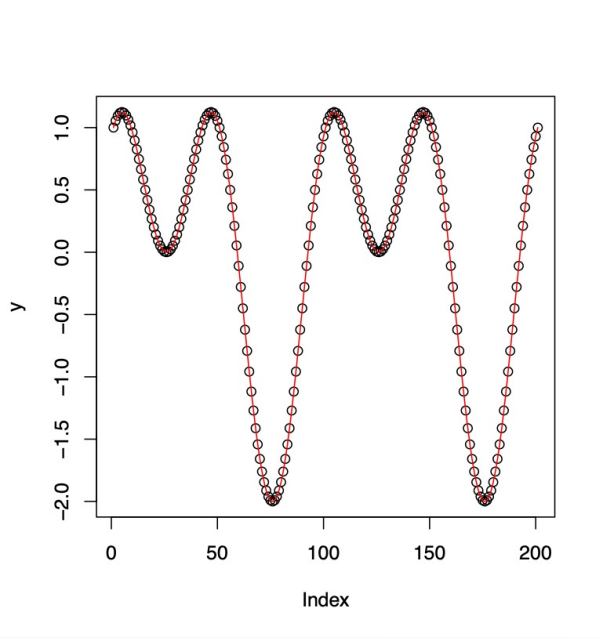


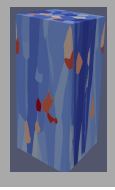


Two more examples



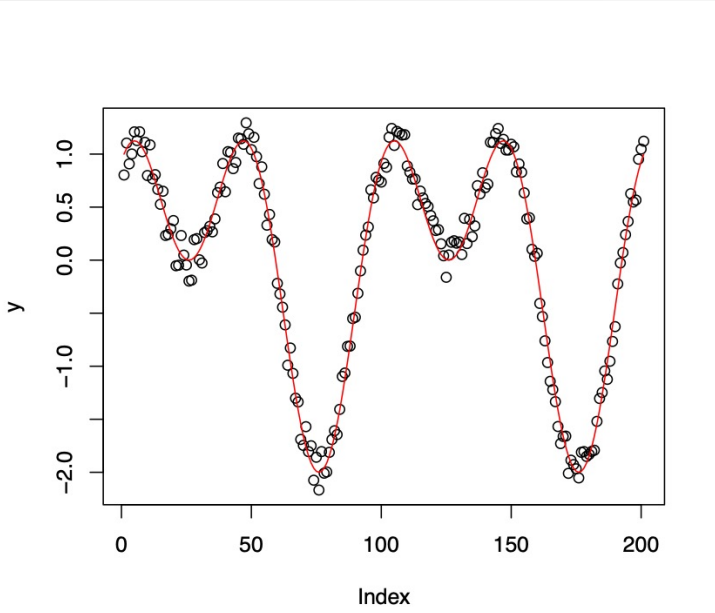
```
x <- seq(0, 4*pi, length.out = 201)
y <- sin(x) + cos(x + x)
plot(y)

ruleDef <- list(expr = grule(op(expr, expr), func(expr), var),
+                 func = grule(sin, cos),
+                 op = grule('+', '-', '*'),
+                 var = grule(x))
grammarDef <- CreateGrammar(ruleDef)

SymRegFitFunc <- function(expr) {
+   result <- eval(expr)
+   if (any(is.nan(result)))
+     return(Inf)
+   return (mean(log(1 + abs(y - result))))
+ }

set.seed(314)
ge <- GrammaticalEvolution(grammarDef, SymRegFitFunc, terminationCost = 0.05, iterations = 5000,
max.depth = 5)
ge
Grammatical Evolution Search Results:
  No. Generations:  2149
  Best Expression:  sin(x) + cos(x + x)
  Best Cost:        0
> plot(y)
> points(eval(ge$best$expressions), col = "red", type = "l")
```

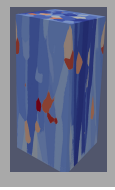




Example 1: $y = \sin(x) + \cos(x + x)$

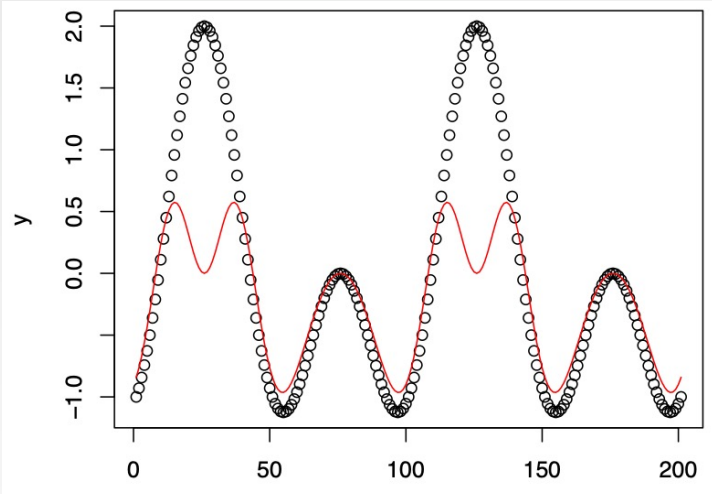
```
x <- seq(0, 4*pi, length.out = 201)
y <- jitter(sin(x) + cos(x + x),amount=0.2)
plot(y)

set.seed(314)
ge <- GrammaticalEvolution(grammarDef, SymRegFitFunc,
terminationCost = 0.05, iterations = 5000, max.depth = 5)

ge
Grammatical Evolution Search Results:
  No. Generations:  5000
  Best Expression:  sin(x) + cos(x + x)
  Best Cost:        0.0923240003917875

plot(y)
points(eval(ge$best$expressions), col = "red", type = “l")
```



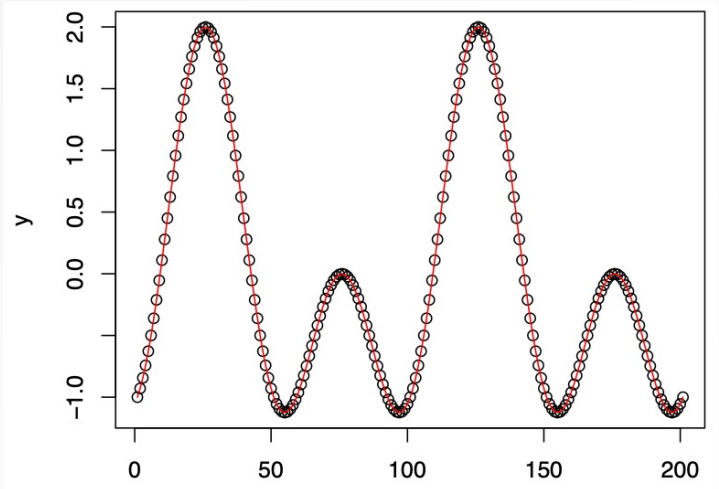


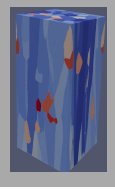
Example 1: $y = \sin(x) + \cos(x + x)$ (noisy data)

```
x <- seq(0, 4*pi, length.out = 201)
y <- sin(x) - cos(x + x)
plot(y)

set.seed(314)
ge <- GrammaticalEvolution(grammarDef, SymRegFitFunc,
terminationCost = 0.05, iterations = 5000, max.depth = 5)
ge
Grammatical Evolution Search Results:
  No. Generations:  5000
  Best Expression:  sin(x - cos(x) + x) * cos(x)
  Best Cost:        0.279197898412931
```



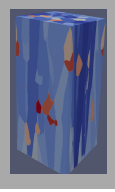

```
set.seed(213)
ge <- GrammaticalEvolution(grammarDef, SymRegFitFunc,
terminationCost = 0.05, iterations = 5000, max.depth = 5)
ge
Grammatical Evolution Search Results:
  No. Generations:  1397
  Best Expression:  sin(x) - cos(x + x)
  Best Cost:        0
```





Example 2: $y = \sin(x) - \cos(x + x)$

Genetic programming, an Evolutionary Algorithm

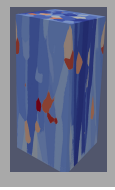


Genetic Algorithm (GA) is a search-based optimization technique based on the principles of **Genetics and Natural Selection**. It is frequently used to find optimal or near-optimal solutions to difficult problems which otherwise would take a lifetime to solve.

In GAs, we have a **pool or a population of possible solutions** to the given problem. These solutions then undergo recombination and mutation (like in natural genetics), producing new children, and the process is repeated over various generations. Each individual (or candidate solution) is assigned a fitness value (based on its objective function value) and the fitter individuals are given a higher chance to mate and yield more "fitter" individuals.

Genetic Algorithms are sufficiently randomized in nature, but they perform much better than random local search (in which we just try various random solutions, keeping track of the best so far), as they exploit historical information as well.

https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_introduction.htm



What are genetic algorithms?

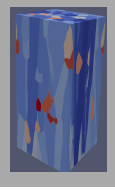
The following is an example of a generic single-objective genetic algorithm.

Step One: Generate the initial population of individuals randomly. (First generation)

Step Two: Repeat the following regenerational steps until termination:

1. Evaluate the fitness of each individual in the population (time limit, sufficient fitness achieved, etc.)
2. Select the fittest individuals for reproduction. (Parents)
3. Breed new individuals through crossover and mutation operations to give birth to offspring.
4. Replace the least-fit individuals of the population with new individuals.

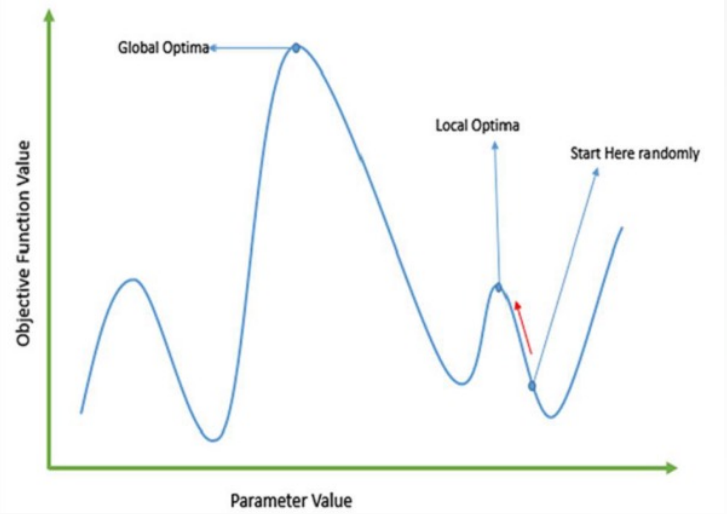
https://en.wikipedia.org/wiki/Evolutionary_algorithm



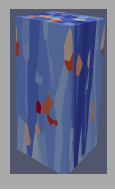
In computer science, there is a large set of problems, which are **NP-Hard**. What this essentially means is that, even the most powerful computing systems take a very long time (even years!) to solve that problem.

In such a scenario, GAs prove to be an efficient tool to provide **usable near-optimal solutions** in a short amount of time.

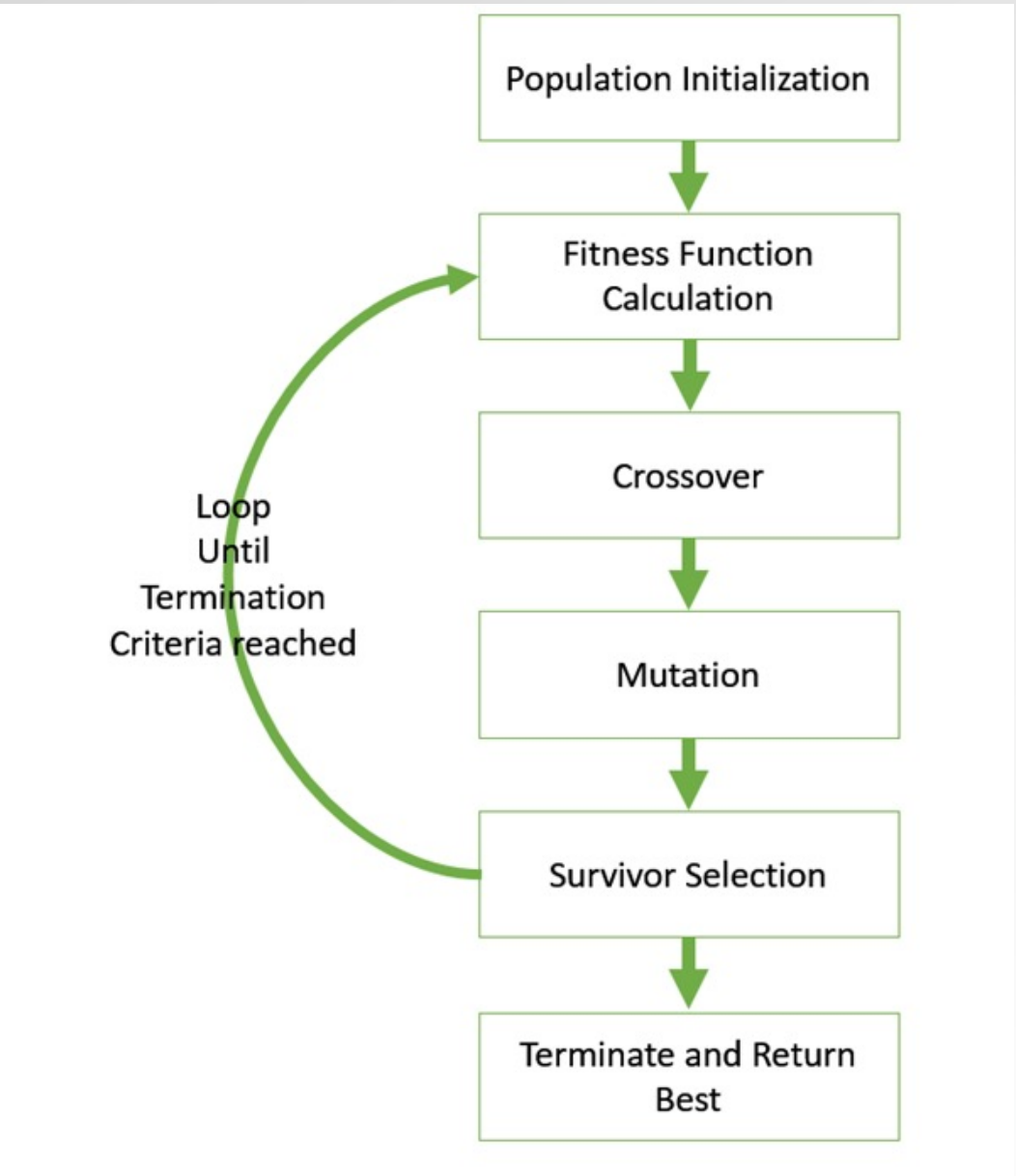
A GA can be a big improvement over gradient-based methods for complex problems with many parameters and you cannot calculate the gradients. Mutations and crossovers enable the algorithm to "get out of" local minima.



https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_introduction.htm

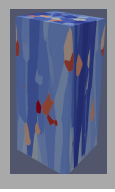


When should genetic algorithms be used?



This gives the basics for the algorithm.

https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_introduction.htm



NASA bingo

- Thanks to a collaboration with Prof. Jacob Hochhalter at the Univ. of Utah, we can use the NASA-bingo package on github. Instructions for installation and use have been posted on Canvas.
- Two example scripts are provided. One generates points along a circle, which is of course a very simple example. The second uses data from fatigue tests conducted by Virkler et al.
- The point of this second example is to show that exploring a range of possible functions leads to the Paris Law in a natural fashion.
- D. A. Virkler, B. M. Hillberry, and P. K. Goel, The Statistical Nature of Fatigue Crack Propagation, *Journal of Engineering Materials and Technology*, **101**, 148-153 (1979).

Fatigue data fitting

- The script virkler_bingo.py contains a number of interesting features.
- One elementary setting is the maximum number of iterations which is encoded as the max in generations allowed for evolution of the population:

```
MAX_GENERATIONS = 3000
```

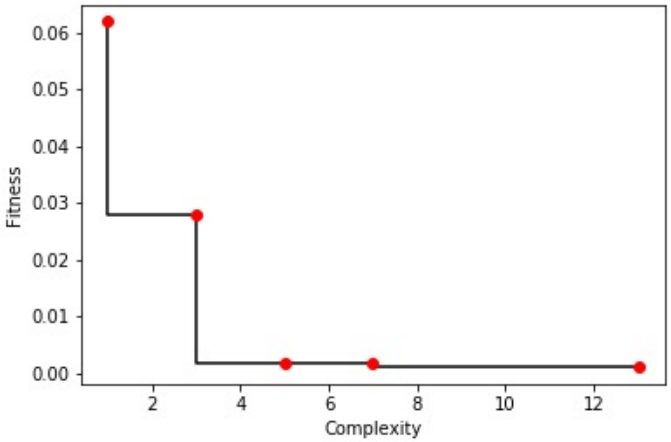
- You can also control the size of the population (smaller is faster!):

```
POP_SIZE = 100
```

- Another key section specifies the mathematical operations to be used:

```
component_generator =
ComponentGenerator(x.shape[1])
component_generator.add_operator("*")
component_generator.add_operator("+")
component_generator.add_operator("-")
component_generator.add_operator("/")
component_generator.add_operator("^")
```

Fatigue data fitting, results



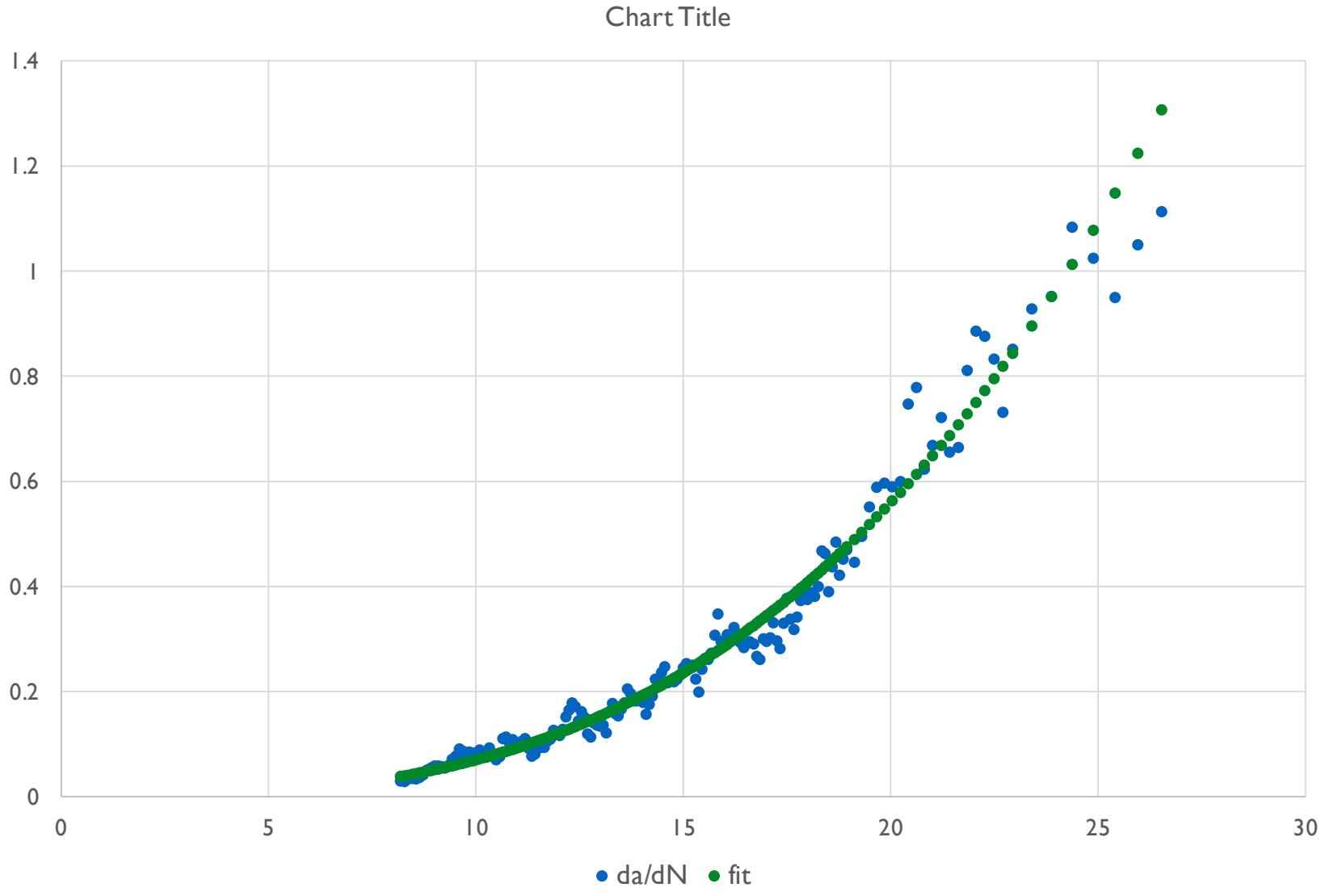
The results require some effort to read:

```
 ×□□W□□ !"#$%&'()*+,-./0123456789:;□ÿ□□□<=□>□□?@ABCDEFGHIJ□KLMN□O□Q□□□□□#&):□□'0<68□9,24.5□-$%(□/1
occurred', ngen=3000, fitness=0.06729947304389196, time=2809.409521)
 Generation:  3000
   FITNESS    COMPLEXITY    EQUATION
 1.851e-03     11    f(X_0) = ((X_0 + 0.1660233929645945 +
0.0022660830552822782)/((0.0022660830552822782)(10884.8107199302)) )(((X_0 + 0.1660233929645945 +
0.0022660830552822782)/((0.0022660830552822782)(10884.8107199302)) )((X_0 + 0.1660233929645945 +
0.0022660830552822782)/((0.0022660830552822782)(10884.8107199302)) ))
 1.921e-03     8    f(X_0) = ((X_0 + (X_0)^(X_0 - (X_0)))((X_0 + (X_0)^(X_0 -
(X_0)))(X_0)))(6.066872390301553e-05)
 1.933e-03     5    f(X_0) = ((X_0)(X_0))((6.992145775141804e-05)(X_0))
 7.145e-03     4    f(X_0) = ((-0.03661352555895907)(X_0))((-0.03661352555895907)(X_0))
 2.799e-02     3    f(X_0) = (0.021920988902594266)(X_0)
 6.199e-02     1    f(X_0) = 0.2912938396548981
```

The simplest that is likely to be useful is the one with complexity=5, which is just a constant * x^3, where x is the ΔK value. The (Pareto) plot shows the trade-off between fitness and complexity. This is the same sort of trade-off as we saw in the context of MLR and best-subset vs. ridge vs. lasso.

Graph of Fitted Curve

Clearly, an excellent fit was obtained! Blue points are data and green are the best fit curve



Testing the procedure

- In the first example with fitting a circle, note the section that generates the data:

```
def generate_circle_pts(r, n):
    # create x,y data points that fall on a circle
    # these are the data that bingo will try and fit and eqn to

    x = np.array([m.cos(m.pi/n*i)*r for i in range(0,n+1)])
    y = np.array([m.sin(m.pi/n*j)*r for j in range(0,n+1)])
    return x.reshape([len(x),1]), y.reshape([len(y),1])
```

- In this case, the set of functions to be used is quite restricted:

```
# tell bingo which mathematical building blocks may be used
   component_generator = ComponentGenerator(x.shape[1])
   component_generator.add_operator("+")
   component_generator.add_operator("-")
   component_generator.add_operator("*")
   component_generator.add_operator("sqrt")
```

- To try a different test of the procedure, simply insert a new "def" of a function that generates datapoints from some other function and allow bingo to fit a function to that data.

Other controls

```
 # tell bingo how error is defined
 fitness = ExplicitRegression(training_data=training_data,
metric='mean squared error')

 # tell bingo how to calibrate coefficients
 local_opt_fitness = ContinuousLocalOptimization(fitness,
algorithm='Nelder-Mead')
```

Open Source vs. Commercial Implementation

"AI Feynman: A physics-inspired method for symbolic regression", Silviu-Marian Udrescu & Max Tegmark, *Science Advances* (2020) Vol. 6, no. 16, eaay2631

Quoting, "… the best competitor by far is the commercial Eureqa software sold by Nutonian Inc. at https://www.nutonian.com/products/eureqa, implementing an improved version of the generic search algorithm outlined in (27)." Ref. 27 is M. Schmidt, H. Lipson, Distilling free-form natural laws from experimental data. *Science* **324** 81–85 (2009).

Summary

- Symbolic Regression combines a small number of key features:
  – ability to identify a set of non-linear functions that fit the x-y data supplied
  – user choice of function primitives (which suggests that one should confine the search to functions that are physically reasonable for the particular data)
  – user choice of error measure, e.g., RMSE
  – user choice of population size
  – reasonable computational cost
  – use of genetic programming to use an evolutionary technique to try different functions

Acknowledgements, References

- **Univ. of Utah:** Prof. Jacob Hochhalter
- **NASA Langley:** Geoffrey F. Bomarito

- Jan Krepl, web notes, https://jankrepl.github.io/symbolic-regression/
- Koza J.R. Genetic Programming, MIT Press, ISBN 0-262-11189-6, 1998
- www.genetic-programming.org

Questions?