Contents

1
FFX: Fast, Scalable, Deterministic Symbolic Regression Technology 1
Trent McConaghy

Chapter 1

FFX: FAST, SCALABLE, DETERMINISTIC SYMBOLIC REGRESSION TECHNOLOGY

Trent McConaghy[1]
[1]*Solido Design Automation Inc., Canada*

Abstract

Symbolic regression is a common application for genetic programming (GP). This paper presents a new non-evolutionary technique for symbolic regression that, compared to competent GP approaches on real-world problems, is orders of magnitude faster (taking just seconds), returns simpler models, has comparable or better prediction on unseen data, and converges reliably and deterministically. We dub the approach FFX, for Fast Function Extraction. FFX uses a recently-developed machine learning technique, pathwise regularized learning, to rapidly prune a huge set of candidate basis functions down to compact models. FFX is verified on a broad set of real-world problems having 13 to 1468 input variables, outperforming GP as well as several state-of-the-art regression techniques.

Keywords: technology, symbolic regression, genetic programming, pathwise, regularization, real-world problems, machine learning, lasso, ridge regression, elastic net, integrated circuits

1. Introduction

Consider when we type "A/B" into a math package. This is a least-squares (LS) linear regression problem. The software simply returns an answer. We do not need to consider the intricacies of the theory, algorithms, and implementations of LS regression because others have already done it. LS regression is fast, scalable, and deterministic. *It just works*.

This gets to the concept of "technology" as used by Boyd: "We can say that solving least-squares problems is a (mature) technology, that can be reliably used by many people who do not know, and do not need to know, the details" (Boyd and Vandenberghe, 2004). Boyd cites LS and linear programming as representative examples, and convex optimization getting close. Other examples might include linear algebra, classical statistics, Monte Carlo methods, software compilers, SAT solvers[1], and CLP solvers[2].

(McConaghy et al., 2010) asked: "What does it take to make genetic programming (GP) a technology? ... to be adopted into broader use beyond that of expert practitioners? ... so that it becomes another standard, off-the-shelf method in the 'toolboxes' of scientists and engineers around the world?"

This paper asks what it takes to make symbolic regression (SR) a technology. SR is the automated extraction of whitebox models that map input variables to output variables. GP (Koza, 1992) is a popular approach to do SR, with successful applications to real-world problems such as industrial processing (Smits et al., 2010; Castillo et al., 2010), finance (Korns, 2010; Kim et al., 2008), robotics (Schmidt and Lipson, 2006), and integrated circuit design (McConaghy and Gielen, 2009).

Outside the GP literature, SR is rare; there are only scattered references such as (Langley et al., 1987). In contrast, the GP literature has dozens of papers on SR every year; even the previous GPTP had seven papers involving SR (Riolo et al., 2010). In a sense, the home field of SR is GP. This means, of course, that when authors aim at SR, they start with GP, and look to modify GP to improve speed, scalability, reliability, interpretability, etc. The improvements are typically 2x to 10x, but fall short of performance that would makes SR a "technology" the way LS or linear programming is.

We are aiming for SR as a technology. What if we did not constrain ourselves to using GP? To GP researchers, this may seem heretical at first glance. But if the aim is truly to improve SR, then this should pose no issue. And in fact, we argue that the GP literature is still an appropriate home for such work, because (a) GP authors doing SR deeply care about SR problems, and (b) as already

[1] for boolean satisfiability problems
[2] for constraint logic programming

mentioned, GP is where all the SR publications are. Of course, we can draw inspiration from GP literature, but also many other potentially-useful fields.

This paper presents a new technique for SR, called FFX – Fast Function Extraction. Because of its speed, scalability, and deterministic behavior, FFX has behavior approaching that of a technology. FFX's steps are:

- Enumerate to generate a massive set of linear and nonlinear basis functions.
- Use pathwise regularized learning to find coefficient values for the basis functions in mapping to y. Pathwise learning actually returns a *set* of coefficient vectors; with each successive vector explaining the training data better but with greater risk of overfitting. This has the computational cost of a single LS regression, thanks to recent developments in machine learning (Friedman et al., 2010; Zou and Hastie, 2005).
- Nondominated-filter to the number of bases versus the testing or training error.

While FFX does not use GP directly, it will become evident that its aims and design are GP-influenced.

We will compare FFX to a competent GP-SR approach on a set of real-world problems. We will see that FFX returns simpler models, has comparable or better prediction on unseen data, and is completely deterministic. Figure 1-4 summarizes the key result of this paper. Furthermore, we will show how to successfully scale FFX to real-world problems having $>$1000 input variables, which to our knowledge is the most input variables that any SR technique has attacked[1].

The rest of this paper is organized as follows. Section 2 describes the SR problem. Section 3 describes pathwise regularized learning. Section 4 describes the FFX algorithm, and section 5 presents results using the algorithm. Section 6 scales up FFX, guided by theory. Section 7 gives results using the scalable FFX algorithm. Section 8 gives related work in GP and elsewhere. Section 9 concludes.

2. SR Problem Definition

Given: $\boldsymbol{X}$ and $\boldsymbol{y}$, a set of $\{\boldsymbol{x_j}, y_j\}, j = 1..N$ data samples where $\boldsymbol{x_j}$ is an n-dimensional point j and y_j is a corresponding output value. Determine: a set of symbolic models $M = m_1, m_2, \ldots$ that provide the Pareto-optimal tradeoff between minimizing model complexity $f_1(m)$ and minimizing expected future model prediction error $f_2 = E_{x,y}L(m)$ where $L(m)$ is the squared-error loss function $y - m(\boldsymbol{x}))^2$. Each model m maps an n-dimensional input $\boldsymbol{x}$ to a scalar

[1]To be precise: attacked directly, without pre-filtering input variables or transforming to a smaller dimensionality

output value $\hat{y}$, i.e. $\hat{y} = m(\boldsymbol{x})$. Complexity is *some* measure that differentiates the degrees of freedom between different models; we use the number of basis functions.

We restrict ourselves to the class of generalized linear models (GLMs) (Nelder and Wedderburn, 1972). A GLM is a linear combination of N_B basis functions $B_i; i = 1, 2, ..., N_B$:

$$\hat{y} = m(\boldsymbol{x}) = a_0 + \sum_{i=1}^{N_B} a_i * B_i(\boldsymbol{x}) \tag{1.1}$$

3. Background: Pathwise Regularized Learning

Least-squares (LS) learning aims to find the values for each coefficient a_i in equation (1.1) that minimize $||\boldsymbol{y} - \boldsymbol{X} * a||^2$, where the $\boldsymbol{X}$ and $\boldsymbol{y}$ are training data. Therefore LS learning aims to minimize training error; it does not acknowledge testing error (future model prediction error). Because it is singularly focused on training error, LS learning may return model coefficients a where a few coefficients are extremely large, making the model overly sensitive to those coefficients. This is overfitting.

Regularized learning aims to minimize the model's sensitivity to overfit coefficient values, by adding minimization terms that are dependent solely on the coefficients: $||a||^2$ or $||a||_1$. This has the implicit effect of minimizing expected future model prediction error. The overall problem formulation is:

$$a^* = minimize\ ||\boldsymbol{y} - \boldsymbol{X} * a||^2 + \lambda_2||a||^2 + \lambda_1||a||_1 \tag{1.2}$$

Including both regularization terms is an *elastic net* formulation of regularized learning (Zou and Hastie, 2005)[1]. To make the balance between λ_1 and λ_2 explicit, we can set $\lambda_1 = \lambda$ and $\lambda_2 = (1 - \rho) * \lambda$, where λ is now the regularization weight, and ρ is a "mixing parameter."

A *path* of solutions sweeps across a set of possible λ values; returning an a for each λ. Interestingly, we can start at a *huge* value of λ, where all a_i are zero; then work towards smaller λ, uniformly on a log scale. Figure 1-1 illustrates: the path starts on the far left, and the with λ decreasing (going right), coefficients a_i take nonzero values one at a time.

An extremely fast variant of pathwise elastic nets was recently developed / rediscovered: coordinate descent (Friedman et al., 2010). At each point on the path, coordinate descent solves for coefficient vector a by: looping through

[1]The middle term (quadratic term, like ridge regression), encourages correlated variables to group together rather than letting a single variable dominate, and makes convergence more stable. The last term (l_1 term, like lasso), drives towards a sparse model with few coefficients, but discourages any coefficient from being too large. $||a||_1 = \sum_i |a_i|$.

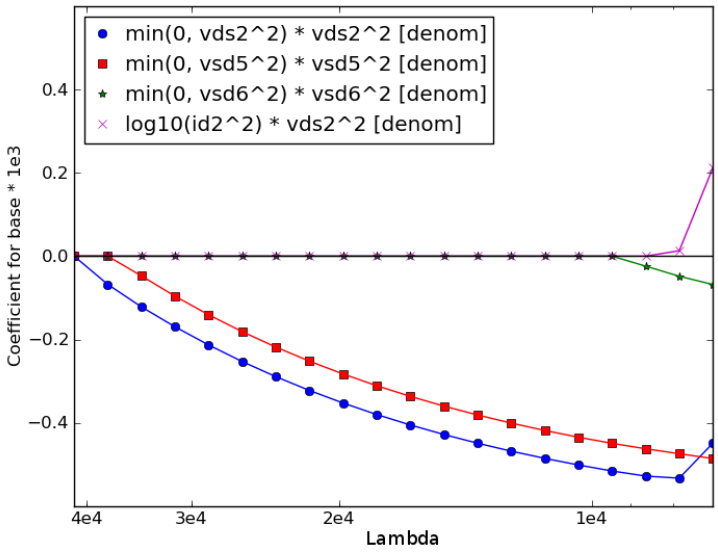

Figure 1-1. A path of regularized regression solutions: each vertical slice of the plot gives a vector of coefficient values a for each of the respective basis functions. Going left to right (decreasing λ), each coefficient a_i follows its own path, starting at zero then increasing in magnitude (and sometimes decreasing).

each a_i one at a time, updating the a_i through a trivial formula while holding the rest of the parameters fixed, and repeating until a stabilizes. For speed, it uses "hot starts": at each new point on the path, coordinate descent starts with the previous point's a.

Some highly useful properties of pathwise regularized learning are:

- Learning speed is comparable or better than LS.
- Unlike LS, can learn when there are fewer samples than coefficients $N < n$.
- Can learn thousands or more coefficients.
- It returns a whole *family* of coefficient vectors, with different tradeoffs between number of nonzero coefficients and training accuracy.

For further details, we refer the reader to (Zou and Hastie, 2005; Friedman et al., 2010).

4. FFX Algorithm

The FFX algorithm has three steps, which we now elaborate.

FFX Step One. Here, FFX generates a massive set of basis functions, where each basis function combines one or more interacting nonlinear subfunctions.

Table 1-1 gives the pseudocode. Steps 1-10 generate univariate bases, and steps 11-20 generate bivariate bases (and higher orders of univariate bases). The algorithm simply has nested loops to generate all the bases. The $eval$ function (line 5, 9, and 18) evaluates a base b given input data $\boldsymbol{X}$; that is, it runs the function defined by b with input vectors in $\boldsymbol{X}$. The $ok()$ function returns $False$ if any evaluated value is $\inf$, - $\inf$, or NaN, e.g. as caused by divide-by-zero, log on negative values, or negative exponents on negative values. Therefore, ok filters away all poorly-behaving expressions. Line 16 means that expressions of the form $op() * op()$ are not allowed; these are deemed too complex.

Table 1-1. Step One: GenerateBases()

Inputs: $\boldsymbol{X}$ #input training data
Outputs: $\boldsymbol{B}$ #list of bases

```
# Generate univariate bases
1.  B1 = {}
2.  for each input variable v = {x1, x2, ...}
3.      for each exponent exp = {0.5, 1.0, 2.0 }
4.          let expression b_exp = v^exp
5.          if ok(eval(b_exp, X))
6.              add b_exp to B1
7.              for each operator op = {abs(), log10, ...}
8.                  let expression b_op = op(b_exp)
9.                  if ok(eval(b_op, X))
10.                     add b_op to B1

# Generate interacting-variable bases
11. B2 = {}
12. for i = 1 to length(B1)
13.     let expression b_i = B1[i]
14.     for j = 1 to i − 1
15.         let expression b_j = B1[j]
16.         if b_j is not an operator # disallow op() * op()
17.             let expression b_inter = b_i * b_j
18.             if ok(eval(b_inter, X))
19.                 add b_inter to B2
20. return B = B1 ∪ B2
```

FFX Step Two. Here, FFX uses pathwise regularized learning (Zou and Hastie, 2005) to identify the best coefficients and bases when there are 0 bases, 1 base, 2 bases, and so on.

Table 1-2 gives the pseudocode. Steps 1-2 create a large matrix $\boldsymbol{X_B}$ which has evaluated input matrix $\boldsymbol{X}$ on each of the basis functions in $\boldsymbol{B}$. Steps 3-4 determine a log-spaced set of N_λ values; see (Zou and Hastie, 2005) for motivations here. Steps 5-16 are the main work, doing pathwise learning.

At each iteration of the loop it performs an elastic-net linear fit (line 11) from $\boldsymbol{X_B} \mapsto \boldsymbol{y}$ to find the linear coefficients $\boldsymbol{a}$. As the loop iterates, N_{bases} tends to increase, because with smaller λ there is more pressure to explain the training data better, therefore requiring the usage of more nonzero coefficients. Once a coefficient value a_i is nonzero, its magnitude tends to increase, though sometimes it will decrease as another coefficient proves to be more useful in explaining the data.

FFX step two is like standard pathwise regularized learning, *except* that whereas the standard approach covers a whole range of λ such that all coefficients eventually get included, FFX stops as soon as there are more than $N_{max-bases}$ (e.g. 5) nonzero coefficients (line 9). Naturally, this is because in the SR application, expressions with more than $N_{max-bases}$ are no longer interpretable. In practice, this makes an enormous difference to runtime; for example, if there are 7000 possible bases but the maximum number of bases is 5, and assuming that coefficients get added approximately uniformly with decreasing λ, then only 5/7000 = 1/1400 = 0.07% of the path must be covered.

Table 1-2. Step Two: PathwiseLearn()

Inputs: $\boldsymbol{X}$, $\boldsymbol{y}$, $\boldsymbol{B}$ #input data, output data, bases
Outputs: $\boldsymbol{A}$ #list of coefficent-vectors

Compute $\boldsymbol{X_B}$
1. for i = 1 to $length(\boldsymbol{B})$
2. $\quad \boldsymbol{X_B}[\boldsymbol{i}]$ = eval($\boldsymbol{B}[\boldsymbol{i}]$, $\boldsymbol{X}$)

Generate $\boldsymbol{\lambda_{vec}}$ = range of λ values
3. $\lambda_{max} = max(|\boldsymbol{X}^T\boldsymbol{y}|)/(N * \rho)$
4. $\boldsymbol{\lambda_{vec}} = logspace(log_{10}(\lambda_{max} * eps), log_{10}(\lambda_{max}), N_\lambda)$

Main pathwise learning
5. $\boldsymbol{A}$ = {}
6. $N_{bases} = 0$
7. i = 0
8. $\boldsymbol{a}$ = $\{0, 0, \ldots\}$ #length n
9. while $N_{bases} < N_{max-bases}$ and $i < length(\boldsymbol{\lambda_{vec}})$
10. $\quad \lambda = \boldsymbol{\lambda_{vec}}[\boldsymbol{i}]$
11. $\quad \boldsymbol{a} = elasticNetLinearFit(\boldsymbol{X_B}, \boldsymbol{y}, \lambda, \rho, \boldsymbol{a})$
12. $\quad N_{bases}$ = number of nonzero values in $\boldsymbol{a}$
13. $\quad$ if $N_{bases} < N_{max-bases}$
14. $\quad\quad$ add $\boldsymbol{a}$ to $\boldsymbol{A}$
15. $\quad i = i + 1$
16. return $\boldsymbol{A}$

In short, the special property of pathwise regularized learning, to start with zero coefficients and incrementally insert them (and therefore insert bases), reconciles extremely well with the SR objectives trading off complexity versus

accuracy with an upper bound on complexity. To a GP practitioner, it feels like doing a whole multi-objective optimization for the cost of a single LS solve.

FFX Step Three. Here, FFX filters the candidate functions to a nondominated set that trades off number of bases and error.

Table 1-2 gives the pseudocode. Steps 1-8 take the coefficients and bases determined in previous FFX steps, and simply combine them to create a set of candidate models M_{cand}. Steps 9-13 apply standard nondominated filtering to the models, with objectives to minimize complexity(number of bases) and error.

Table 1-3. Step Three: NondominatedFilter()

Inputs: $\boldsymbol{A}$, $\boldsymbol{B}$ # coefficient vectors, bases
Outputs: M # Pareto-optimal models

Construct candidate models
1. M_{cand} = {}
2. for i = 1 to length(||$\boldsymbol{A}$||)
3. $\quad \boldsymbol{a} = \boldsymbol{A}[\boldsymbol{i}]$
4. $\quad a_0 = \boldsymbol{a}[\boldsymbol{0}]$ # offset
5. $\quad \boldsymbol{a_{nz}}$ = nonzero values in $\boldsymbol{a}$ (ignoring offset)
6. $\quad \boldsymbol{B_{nz}}$ = expressions in $\boldsymbol{B}$ corr. to nonzero values in $\boldsymbol{a}$
7. $\quad m$ = model(a_0, $\boldsymbol{a_{nz}}$, $\boldsymbol{B_{nz}}$), following eqn. (1.1)
8. $\quad$ add m to M_{cand}

Nondominated filtering
9. $\boldsymbol{f_1}$ = $numBases(m)$, for each m in M_{cand}
10. $\boldsymbol{f_2}$ = $testError(m)$ or $trainError(m)$, for each m in M_{cand}
11. J = nondominatedIndices($\boldsymbol{f_1}$, $\boldsymbol{f_2}$)
12. $M = M_{cand}[j]$ for each j in J
13. return M

Rational Functions Trick. For maximum coverage of possible functions, FFX leverages a special technique inspired by (Leung and Haykin, 1993) to include rational functions, with negligible extra computational cost. The general idea is: learning the coefficients of a rational function can be cast into a linear regression problem, solved with linear regression, then back-transformed into rational function form. Let us elaborate:

A rational function has the form:

$$\hat{y} = m(\boldsymbol{x}) = \frac{a_0 + \sum_{i=1}^{N_{BN}} a_i * B_i(\boldsymbol{x})}{1.0 + \sum_{i=N_{BN}+1}^{N'_B} a_i * B_i(\boldsymbol{x})} \tag{1.3}$$

where N'_B is the number of numerator bases (N_{BN}) plus the number of denominator bases (N_{BD}).

Let us perform simple algebraic manipulations to transform this problem. First, we multiply both sides by the denominator:

$$y * \left(1.0 + \sum_{i=N_{BN}+1}^{N'_B} a_i * B_i(\boldsymbol{x})\right) = a_0 + \sum_{i=1}^{N_{BN}} a_i * B_i(\boldsymbol{x}) \tag{1.4}$$

Then we expand the left-hand side:

$$y + \sum_{i=N_{BN}+1}^{N'_B} a_i * B_i(\boldsymbol{x}) * y = a_0 + \sum_{i=1}^{N_{BN}} a_i * B_i(\boldsymbol{x}) \tag{1.5}$$

where $B_i(\boldsymbol{x}) * y$ is element-wise multiplication, i.e. $B_i(\boldsymbol{X_j}) * \boldsymbol{y_j}$ for each data point j. Now, subtract to isolate y on the left-hand side:

$$y = a_0 + \sum_{i=1}^{N_{BN}} a_i * B_i(\boldsymbol{x}) - \sum_{i=N_{BN}+1}^{N'_B} a_i * B_i(\boldsymbol{x}) * y \tag{1.6}$$

Finally, let us define a new set of basis functions.

$$B'_i = \begin{Bmatrix} B_i & i \leq N_{BN} \\ B_i * y & otherwise \end{Bmatrix} \tag{1.7}$$

At the end of FFX step 1, we had N_B basis functions. Before we start step 2, we insert all N_B functions into both the numerator and denominator; therefore $N_{BN} = N_{BD} = N_B$, and $N'_B = 2 * N_B$. We redefine the basis functions according to eqn. (1.7). Then, all the subsequent FFX steps are performed with these new basis functions. Once the coefficients are found, the final model is extracted by applying the algebraic manipulations in reverse: eqn. (1.6), then eqn. (1.5), then eqn. (1.4).

This concludes the description of the FFX algorithm. Note that for improved scalability, FFX must be adapted according to section 6.

5. Medium-Dimensional Experiments

This section presents experiments on medium-dimensional problems. (Section 7 will give higher-dimensional results.)

Experimental Setup

Problem Setup. We use a test problem used originally in (Daems et al., 2003) for posynomial fitting, but also studied extensively using GP-based SR (McConaghy and Gielen, 2009). The aim is to model performances of a well-known analog circuit, a CMOS operational transconductance amplifier (OTA).

The goal is to find expressions for the OTA's performance measures: low-frequency gain (A_{LF}), phase margin (PM), positive and negative slew rate (SR_p, SR_n), input-referred offset voltage (V_{offset}), and unity-gain frequency (f_u). [1]

Each problem has 13 input variables. Input variable space was sampled with orthogonal-hypercube Design-Of-Experiments (DOE) (Montgomery, 2009), with scaled dx=0.1 (where dx is % change in variable value from center value), to get 243 samples. Each point was simulated with SPICE. These points were used as training data inputs. Testing data points were also sampled with DOE and 243 samples, but with dx=0.03. Thus, this experiment leads to somewhat localized models; we could just as readily model a broader design space, but this allows us to compare the results to (Daems et al., 2003). We calculate normalized mean-squared error on the training data and on the separate testing data: $nmse = \sqrt{\sum_i ((\widehat{y}_i - y_i)/(max(\boldsymbol{y}) - min(\boldsymbol{y}))^2)}$

FFX Setup. Up to $N_{max-bases}$=5 bases are allowed. Operators allowed are: $abs(x)$, $log_{10}(x)$, $min(0, x)$, $max(0, x)$; and exponents on variables are $x^{1/2}$ (=$\sqrt{(x)}$), x^1 (=x), and x^2. By default, denominators are allowed; but if turned off, then negative exponents are also allowed: $x^{-1/2}$ (=$1/\sqrt{(x)}$), x^{-1} (=$1/x$), and x^{-2} (=$1/x^2$). The elastic net settings were ρ = 0.5, $\lambda_{max} = max|\boldsymbol{X}^T\boldsymbol{y}|/(N * \rho)$, $eps = 10^{-70}$, and N_λ=1000.

Because the algorithm is not GP, there are no settings for population size, number of generations, mutation/crossover rate, selection, etc. We emphasize that the settings in the previous paragraph are very simple, with no tuning needed by users.

Each FFX run took ≈5 s on a 1-GHz single-core CPU.

Reference GP-SR Algorithm Setup. CAFFEINE is a state-of-the-art GP-based SR approach that uses a thoughtfully-designed grammar to constrain SR functional forms such that they are interpretable by construction. Key settings are: up to 15 bases, population size 200, and 5000 generations. Details are in (McConaghy and Gielen, 2009). Each CAFFEINE run took ≈10 minutes on a 1-GHz CPU.

Experimental Results

This section experimentally investigates FFX behavior, and validates its prediction abilities on the set of six benchmark functions.

FFX Data Flow. To start with, we examine FFX behavior in detail on a test problem. Recall that FFX has three steps: generating the bases, pathwise

[1] We log-scale f_u so that learning is not wrongly biased towards high-magnitude samples of f_u.

learning on the bases, and pruning the results via nondominated filtering. We examine the data flow of these steps on the A_{LF} problem.

The first step in FFX generated 176 candidate one-variable bases, as shown in Table 1-4. These bases combined to make 3374 two-variable bases, some of which are shown in Table 1-5. This made a total of 3550 bases for the numerator; and another 3550 for the denominator[1].

Table 1-4. For FFX step 1: The 176 candidate one-variable bases.

$v_{sg1}^{0.5}$, $abs(v_{sg1}^{0.5})$, $max(0, v_{sg1}^{0.5})$, $min(0, v_{sg1}^{0.5})$, $log_{10}(v_{sg1}^{0.5})$, v_{sg1}, $abs(v_{sg1})$, $max(0, v_{sg1})$, $min(0, v_{sg1})$, $log_{10}(v_{sg1})$, v_{sg1}^2, $max(0, v_{sg1}^2)$, $min(0, v_{sg1}^2)$, $log_{10}(v_{sg1}^2)$, $v_{gs2}^{0.5}$, $abs(v_{gs2}^{0.5})$, $max(0, v_{gs2}^{0.5})$, $min(0, v_{gs2}^{0.5})$, $log_{10}(v_{gs2}^{0.5})$, v_{gs2}, $abs(v_{gs2})$, $max(0, v_{gs2})$, $min(0, v_{gs2})$, $log_{10}(v_{gs2})$, v_{gs2}^2, $max(0, v_{gs2}^2)$, $min(0, v_{gs2}^2)$, $log_{10}(v_{gs2}^2)$, $v_{ds2}^{0.5}$, $abs(v_{ds2}^{0.5})$, $max(0, v_{ds2}^{0.5})$, $min(0, v_{ds2}^{0.5})$, $log_{10}(v_{ds2}^{0.5})$, v_{ds2}, $abs(v_{ds2})$, $max(0, v_{ds2})$, $min(0, v_{ds2})$, $log_{10}(v_{ds2})$, v_{ds2}^2, $max(0, v_{ds2}^2)$, $min(0, v_{ds2}^2)$, $log_{10}(v_{ds2}^2)$, $v_{sg3}^{0.5}$, $abs(v_{sg3}^{0.5})$, $max(0, v_{sg3}^{0.5})$, $min(0, v_{sg3}^{0.5})$, $log_{10}(v_{sg3}^{0.5})$, v_{sg3}, $abs(v_{sg3})$, $max(0, v_{sg3})$, $min(0, v_{sg3})$, $log_{10}(v_{sg3})$, v_{sg3}^2, $max(0, v_{sg3}^2)$, $min(0, v_{sg3}^2)$, $log_{10}(v_{sg3}^2)$, $v_{sg4}^{0.5}$, $abs(v_{sg4}^{0.5})$, $max(0, v_{sg4}^{0.5})$, $min(0, v_{sg4}^{0.5})$, $log_{10}(v_{sg4}^{0.5})$, v_{sg4}, $abs(v_{sg4})$, $max(0, v_{sg4})$, $min(0, v_{sg4})$, $log_{10}(v_{sg4})$, v_{sg4}^2, $max(0, v_{sg4}^2)$, $min(0, v_{sg4}^2)$, $log_{10}(v_{sg4}^2)$, $v_{sg5}^{0.5}$, $abs(v_{sg5}^{0.5})$, $max(0, v_{sg5}^{0.5})$, $min(0, v_{sg5}^{0.5})$, $log_{10}(v_{sg5}^{0.5})$, v_{sg5}, $abs(v_{sg5})$, $max(0, v_{sg5})$, $min(0, v_{sg5})$, $log_{10}(v_{sg5})$, v_{sg5}^2, $max(0, v_{sg5}^2)$, $min(0, v_{sg5}^2)$, $log_{10}(v_{sg5}^2)$, $v_{sd5}^{0.5}$, $abs(v_{sd5}^{0.5})$, $max(0, v_{sd5}^{0.5})$, $min(0, v_{sd5}^{0.5})$, $log_{10}(v_{sd5}^{0.5})$, v_{sd5}, $abs(v_{sd5})$, $max(0, v_{sd5})$, $min(0, v_{sd5})$, $log_{10}(v_{sd5})$, v_{sd5}^2, $max(0, v_{sd5}^2)$, $min(0, v_{sd5}^2)$, $log_{10}(v_{sd5}^2)$, $v_{sd6}^{0.5}$, $abs(v_{sd6}^{0.5})$, $max(0, v_{sd6}^{0.5})$, $min(0, v_{sd6}^{0.5})$, $log_{10}(v_{sd6}^{0.5})$, v_{sd6}, $abs(v_{sd6})$, $max(0, v_{sd6})$, $min(0, v_{sd6})$, $log_{10}(v_{sd6})$, v_{sd6}^2, $max(0, v_{sd6}^2)$, $min(0, v_{sd6}^2)$, $log_{10}(v_{sd6}^2)$, i_{d1}, $abs(i_{d1})$, $max(0, i_{d1})$, $min(0, i_{d1})$, i_{d1}^2, $max(0, i_{d1}^2)$, $min(0, i_{d1}^2)$, $log_{10}(i_{d1}^2)$, $i_{d2}^{0.5}$, $abs(i_{d2}^{0.5})$, $max(0, i_{d2}^{0.5})$, $min(0, i_{d2}^{0.5})$, $log_{10}(i_{d2}^{0.5})$, i_{d2}, $abs(i_{d2})$, $max(0, i_{d2})$, $min(0, i_{d2})$, $log_{10}(i_{d2})$, i_{d2}^2, $max(0, i_{d2}^2)$, $min(0, i_{d2}^2)$, $log_{10}(i_{d2}^2)$, $i_{b1}^{0.5}$, $abs(i_{b1}^{0.5})$, $max(0, i_{b1}^{0.5})$, $min(0, i_{b1}^{0.5})$, $log_{10}(i_{b1}^{0.5})$, i_{b1}, $abs(i_{b1})$, $max(0, i_{b1})$, $min(0, i_{b1})$, $log_{10}(i_{b1})$, i_{b1}^2, $max(0, i_{b1}^2)$, $min(0, i_{b1}^2)$, $log_{10}(i_{b1}^2)$, $i_{b2}^{0.5}$, $abs(i_{b2}^{0.5})$, $max(0, i_{b2}^{0.5})$, $min(0, i_{b2}^{0.5})$, $log_{10}(i_{b2}^{0.5})$, i_{b2}, $abs(i_{b2})$, $max(0, i_{b2})$, $min(0, i_{b2})$, $log_{10}(i_{b2})$, i_{b2}^2, $max(0, i_{b2}^2)$, $min(0, i_{b2}^2)$, $log_{10}(i_{b2}^2)$, $i_{b3}^{0.5}$, $abs(i_{b3}^{0.5})$, $max(0, i_{b3}^{0.5})$, $min(0, i_{b3}^{0.5})$, $log_{10}(i_{b3}^{0.5})$, i_{b3}, $abs(i_{b3})$, $max(0, i_{b3})$, $min(0, i_{b3})$, $log_{10}(i_{b3})$, i_{b3}^2, $max(0, i_{b3}^2)$, $min(0, i_{b3}^2)$, $log_{10}(i_{b3}^2)$

Table 1-5. For FFX step 1: Some candidate two-variable bases (there are 3374 total).

$log_{10}(i_{b3}^2) * i_{d2}^2$, $log_{10}(i_{b3}^2) * i_{b1}^{0.5}$, $log_{10}(i_{b3}^2) * i_{b1}$, $log_{10}(i_{b3}^2) * i_{b1}^2$, $log_{10}(i_{b3}^2) * i_{b2}^{0.5}$, $log_{10}(i_{b3}^2) * i_{b2}$, $log_{10}(i_{b3}^2) * i_{b2}^2$, $log_{10}(i_{b3}^2) * i_{b3}^{0.5}$, $log_{10}(i_{b3}^2) * i_{b3}$, $log_{10}(i_{b3}^2) * i_{b3}^2$

(and 3364 more)

The second FFX step applied pathwise regularized learning on the 7100 bases (3550 numerator + 3550 denominator), as illustrated in Figure 1-1 (previously shown to introduce pathwise learning). It started with maximum lambda (λ), where all coefficient values were 0.0, and therefore there are 0 (far left of figure). Then, it iteratively decreased λ and updated the coefficient estimates. The first base to get a nonzero coefficient was $min(0, v_{ds2}^2) * v_{ds2}^2$ (in the denominator). At a slightly smaller λ, the second base to get a nonzero coefficient was $min(0, v_{sd5}^2) * v_{sd5}^2$ (also in the denominator). These remain the only two bases for several iterations, until finally when λ shrinks below 1e4, a third base is

[1] See "Rational Functions Trick" in section 4.

added. A fourth base is added shortly after. Pathwise learning continued until the maximum number of bases (nonzero coefficients) was hit.

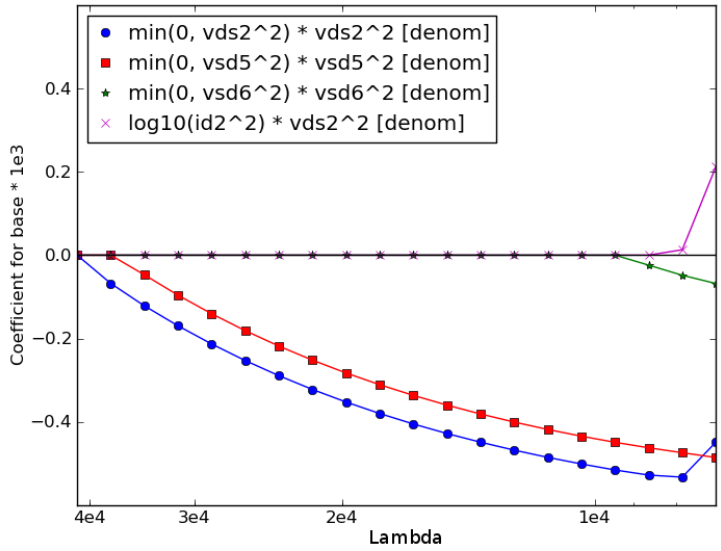

Figure 1-2. For FFX step 2: Pathwise regularized learning following on A_{LF}.

The third and final FFX step applies nondominated filtering to the candidate models, to generate the Pareto Optimal sets that trade off error versus number of bases (complexity). Figure 1-3 shows the outcome of nondominated filtering, for the case when error is training error, and for the case when error is testing error. Training error for this data is higher than testing error because the training data covers a broader input variable range (dx = 0.1) than the testing data (dx = 0.03), as section 5 discussed.

Extracted Whitebox Models. Table 1-6 shows the lowest test-error functions extracted by FFX, for each test problem. First, we see that the test errors are all very low, <5% in all cases. Second, we see that the functions themselves are fairly simple and interpretable, at most having two basis functions. For A_{LF}, PM, and SR_n, FFX determined that using a denominator was better. We continue to find it remarkable that functions like this can be extracted in such a computationally lightweight fashion. For SR_p, FFX determined that the most predictive function was simply a constant (2.35e7). Interestingly, it combined univariate bases of the same variable to get higher-order bases, for example $min(0, v_{ds2}^2) * v_{ds2}^2$ in A_{LF}.

Recall that FFX does is designed to not just return the function with the lowest error, but a whole set of functions that trade off error and complexity. It

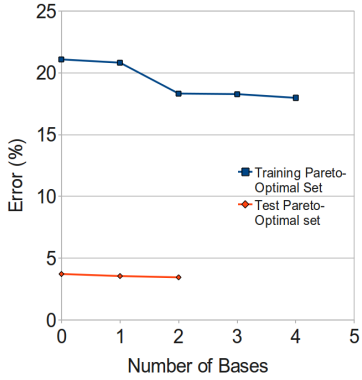

Figure 1-3. For FFX step 3: results of nondominated filtering to get the Pareto optimal tradeoff of error versus number of bases, in modeling A_{LF}. Two cases are shown: when error is on the training data, and when error is on testing data.

Table 1-6. Functions with lowest test error as extracted by FFX, for each test problem. Extraction time per problem was ≈5 s on a 1-GHz machine.

Problem	**Test error (%)**	**Extracted Function**
A_{LF}	3.45	$\frac{37.020}{1.0-1.22e\text{-}4*min(0,v_{ds2}^2)*v_{ds2}^2-4.72e\text{-}5*min(0,v_{sd5}^2)*v_{sd5}^2}$
PM	1.51	$\frac{90.148}{1.0-8.79e\text{-}6*min(0,v_{sg1}^2)*v_{sg1}^2+2.28e\text{-}6*min(0,v_{ds2}^2)*v_{ds2}^2}$
SR_n	2.10	$\frac{-5.21e7}{1.0-8.22e\text{-}5*min(0,v_{gs2}^2)*v_{gs2}^2}$
SR_p	4.74	$2.35e7$
V_{offset}	2.16	$-0.0020 - 1.22e\text{-}23 * min(0, v_{gs2}^2) * v_{gs2}^2$
$log_{10}(f_u)$	2.17	$0.74 - 1.10e\text{-}5 * min(0, v_{sg1}^2) * v_{sg1}^2$ $+1.88e\text{-}5 * min(0, v_{ds2}^2) * v_{ds2}^2$

does this efficiently by exploiting pathwise learning. Table 1-7 illustrates the Pareto optimal set extracted by FFX for the A_{LF} problem.

Prediction Abilities. Figure 1-4 compares FFX to GP-SR, linear models, and quadratic models in terms of average test error and build time. The linear and quadratic models took <1 s to build, using LS learning. GP-SR and FFX predict very well, and linear and quadratic models predict poorly. GP-SR has much longer model-building time than the rest. In sum, FFX has the speed of linear/quadratic models with the prediction abilities of GP-SR.

Table 1-7. Pareto optimal set (complexity vs. test error) for A_{LF} extracted by FFX.

Test error (%)	**Extracted Function**
3.72	37.619
3.55	$\frac{37.379}{1.0-6.78e\text{-}5*min(0,v_{ds2}^2)*v_{ds2}^2}$
3.45	$\frac{37.020}{1.0-1.22e\text{-}4*min(0,v_{ds2}^2)*v_{ds2}^2-4.72e\text{-}5*min(0,v_{sd5}^2)*v_{sd5}^2}$

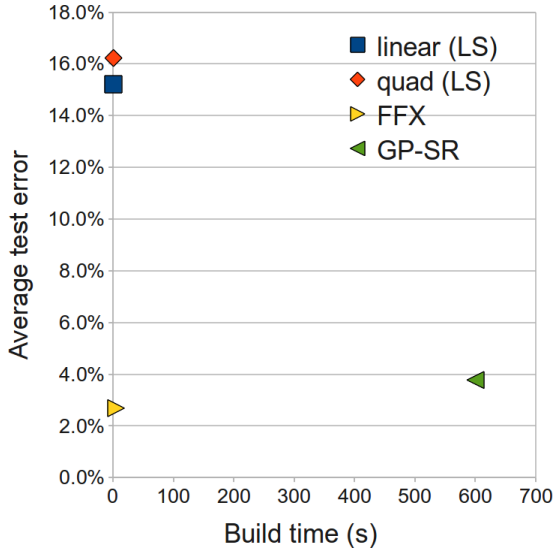

Figure 1-4. Average test error (across six test problems) versus build time, comparing linear, quadratic, FFX, and GP-SR

Table 1-8 compares the test error for linear, quadratic, FFX, and GP-SR models; plus the approaches originally compared in (McConaghy and Gielen, 2005): posynomial (Daems et al., 2003), a modern feedforward neural network (FFNN) (Ampazis and Perantonis, 2002), boosting the FFNNs, multivariate adaptive regression splines (MARS) (Friedman, 1991), least-squares support vector machines (SVM) (Suykens et al., 2002), and kriging (gaussian process models) (Sacks et al., 1989). Lowest-error values are in bold.

From Table 1-8, we see that of all the modeling approaches, FFX has the best average test error; and best test error in four of the six problems, coming close in the remaining two.

Table 1-8. Test error (%) on the six medium-dimensional test problems.

Approach	A_{LF}	PM	SR_n	SR_p	V_{offset}	f_u	**Avg.**
Linear (LS)	17.2	11.9	15.6	20.5	7.1	19.0	15.21
Quadratic (LS)	18.5	12.2	15.7	22.7	7.4	20.9	16.23
FFX (this work)	3.5	**1.5**	**2.1**	**4.7**	2.2	**2.2**	**2.69**
GP-SR	**2.8**	2.6	3.9	7.4	1.0	5.0	3.78
Posynomial	6.5	9.7	78.0	31.0	**0.8**	5.9	21.98
FFNN	5.0	6.8	9.5	8.2	2.9	9.3	6.93
Boosted FFNN	5.3	2.8	9.7	14.0	1.4	10.0	7.19
MARS	4.4	1.8	5.4	7.2	1.2	9.4	4.88
SVM	11.5	5.8	4.1	10.0	1.8	12.7	7.64
Kriging	7.3	3.8	5.1	8.9	2.2	7.3	5.75

6. FFX Scaling

Experimental Setup

So far, we have tested FFX on several problems with 13 input variables. What about larger real-world problems? We consider the real-world integrated circuits listed in Table 1-9. The aim is to map process variables to circuit performance outputs. Therefore, these problems have hundreds or thousands of input variables.

The data was generated by performing Monte Carlo sampling: drawing process points from the process variables' pdf, and simulating each process point using Hspice^{TM}, to get output values. The opamp and voltage reference had 800 Monte Carlo sample points, the comparator and GMC filter 2000, and bitcell and sense amp 5000. The data is chosen as follows: sort the data according to the y-values; every 4th point is used for testing; and the rest are used for training[1].

Initial Scaling Experiments

We ran FFX on the larger circuit problems. In the larger circuits, it failed miserably, getting out-of-memory errors.

[1]This is faster than cross-validation, yet gives consistent, reliable answers.

Table 1-9. Twelve higher-dimensional test problems across six circuits.

Circuit	# Devices	# Input Variables	Outputs Modeled
opamp	30	215	AV (gain), BW (bandwidth) PM (phase margin), SR (slew rate)
bitcell	6	30	$cell_i$ (read current)
sense amp	12	125	$delay$, pwr (power)
voltage reference	11	105	$DVREF$ (difference in voltage), PWR (power)
GMC filter	140	1468	$ATTEN$ (attenuation), IL
comparator	62	639	BW (bandwidth)

To understand why, we can analyze FFX's computational complexity.

FFX Computational Complexity

Let us determine the computational complexity of FFX, for each step. This can be viewed as the core theory for FFX.

Step One. Let e be the number of exponents and o be the number of nonlinear operators. Therefore the number of univariate bases per variable is $(o+1)*e$. (The $+1$ is when no nonlinear operator is applied; or, equivalently, unity). With n as the number of input variables, then the total number of univariate bases is $(o+1)*e*n$. With N samples, the univariate part of step one has a complexity of $O((o+1)*e*n*N)$. Since e and o are constants, this reduces to $O(n*N)$. The number of bivariate bases is $p = O(n^2)$, so the bivarate part of step one has complexity $O(n^2 * N)$.

Step Two. The cost of an older elastic-net pathwise technique, LARS, was approximately that of one LS fitting according to p.93 of (Hastie et al., 2008). Since then, the coordinate descent algorithm (Friedman et al., 2010) has been shown to be 10x faster. Nonetheless, we will use LS as a baseline. With p input variables, LS fitting with QR decomposition has complexity $O(N * p^2)$. Because $p = O(n^2)$, FFX has approximate complexity $O(N * n^4)$.

Step Three. Reference (Deb et al., 2002) shows that nondominated filtering has complexity $O(N_o * N_{nondom})$ where N_o is the number of objectives, and N_{nondom} is the number of nondominated individuals. In the SR cases, N_o is a constant (at 2) and $N_{nondom} \leq N_{max-bases}$ where $N_{max-bases}$ is a constant ($\approx$5). Therefore, FFX step three complexity is $O(1)$.

The complexity of FFX is the maximum of steps one, two, and three; which is $O(N * n^4)$. $\square$

Given this, the fact that FFX hits limits of computational resources when n is large is not surprising. In the largest circuit, n = 1468, therefore n^4 = 4.64e12.

Modifying FFX for Scalability

We can improve FFX to have a computational complexity is $O(N * n^2)$, as follows. We adapt the procedure in Table 1-1 to be stepwise: first learn univariate coefficients; then only combine the $k \leq O(\sqrt{n})$ most important basis functions with each other for candidate bivariate coefficients; then learn the coefficients on the combinations of most-important univariate bases. This means that each linear learning has $\leq O(n)$ basis functions; therefore overall complexity is $O(N * n^2)$.

This adaptation can be seen as a "batch" approach to stepwise-forward regression like that in MARS (Friedman, 1991).

We took another cue from MARS to improve model flexibility, by adding *hinge* basis functions $max(0, x - thr)$, and $max(0, thr - x)$. These operators add "turn off" some regions of input space and focus on remaining regions. For each hinge operator at each variable x_j, we allowed 5 different threshold values thr, uniformly distributed from $minx_j + 0.2 * (maxx_j - minx_j)$ to $minx_j + 0.8 * (maxx_j - minx_j)$; where $minx_j$ and $maxx_j$ are the minimum and maximum values seen for x_j in all training samples.

In preliminary experiments, we found that FFX would give a more thorough sets of results if we re-ran it on different high-level settings as shown in table 1-10, and merged the results.[1]

Table 1-10. FFX runs on each of these settings, and merges the results.

Inter-actions	**Denom-inator**	**Expon-entials**	**Log/Abs Operators**	**Hinge Functions**	Notes
					linear
Y					quadratic
		Y	Y	Y	
	Y	Y	Y	Y	
Y				Y	
Y	Y				
Y	Y			Y	
Y		Y	Y		
Y			Y		
Y			Y	Y	

FFX settings were like in section 5, except up to 250 bases were allowed.

The overall runtime per problem was ≈30 s on a single-core 1-GHz CPU.

7. High-Dimensional Experiments

This section presents results using the scaled-up FFX, on the high-dimensional modeling problems described in section 6.

[1]This, of course, can be trivially parallelized.

Table 1-11 shows the lowest test error found by FFX, compared to other approaches. FFX always gets the lowest test error, and many other approaches failed badly. FFX did find it easier to capture some mappings than others.

Table 1-11. Test error (%) on the twelve high-dimensional test problems. The quadratic model failed because it had too samples for the number of coefficients. GP-SR and FFNN failed, either because test error was $\gg$100% or model build time took unreasonably long (several hours).

Approach	opamp $\boldsymbol{AV}$	opamp $\boldsymbol{BW}$	opamp $\boldsymbol{PM}$	opamp $\boldsymbol{SR}$	bitcell $\boldsymbol{cell_i}$	sense amp $\boldsymbol{delay}$
Lin (LS)	1.7	1.3	1.3	3.2	12.7	3.4
Quad (LS)	FAIL	FAIL	FAIL	FAIL	12.5	3.5
FFX	**1.0**	**0.9**	**1.0**	**2.0**	**12.4**	**3.0**
GP-SR	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL
FFNN	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL

Approach	sense amp $\boldsymbol{pwr}$	voltage reference $\boldsymbol{DVREF}$	voltage reference $\boldsymbol{PWR}$	GMC filter $\boldsymbol{ATTEN}$	GMC filter $\boldsymbol{IL}$	comparator $\boldsymbol{BW}$
Lin (LS)	3.5	2.4	22.8	16.4	17.3	27.2
Quad (LS)	2.9	2.8	40.4	FAIL	FAIL	FAIL
FFX	**2.7**	**1.0**	**2.0**	**7.0**	**8.5**	**17.0**
GP-SR	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL
FFNN	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL

Figure 1-5 shows the tradeoff of equations, for each modeling approach. Each dot represents a different model, having its own complexity and test error. For a given subplot, the simplest model is a constant, at the far left. It also has the highest error. As new bases are added (higher complexity), error drops. The curves have different signatures. For example, we see that when the opamp BW model (top center) gets 2 bases, its error drops from 6.8% to 1.9%. After that, additional bases steadily improve error, until the most complex model having 31 bases has 1% error. Or, for opamp PM (top right), there is little reduction in error after 15 bases.

In many modeling problems, FFX determined that just linear and quadratic terms were appropriate for the best equations. These include the the simpler opamp PM functions, GMC filter IL, GMC filter $ATTEN$, opamp SR (for errors $>$ 2.5%), and bitcell $cell_i$. But in some problems, FFX used more strongly nonlinear functions. These include voltage reference $DVREF$, sense amp $delay$, and sense amp pwr. Let us explore some models in more detail.

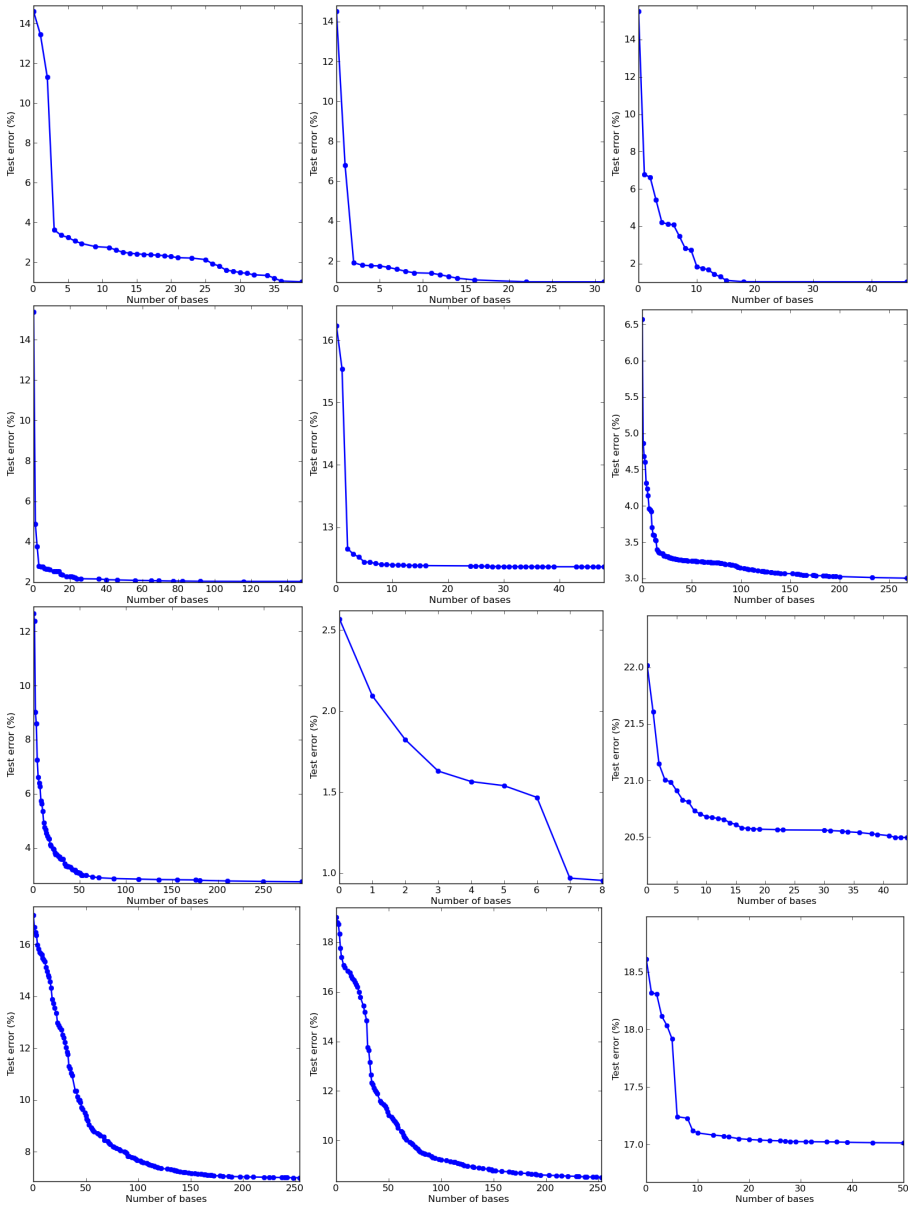

Figure 1-5. Test error vs. Complexity. Top row left-to-right: opamp AV, opamp BW, opamp PM. Second-from-top row: opamp SR, bitcell $cell_i$, sense amp $delay$. Third-from-top row: sense amp pwr, voltage reference $DVREF$, voltage reference PWR. Bottom row: GMC filter $ATTEN$, GMC filter IL, comparator BW.

Table 1-12 shows some functions that FFX extracted for opamp PM. At 0 bases is a constant, of course. From 1 to 4 bases, FFX adds one more linear base at a time, gradually adding resolution to the model. At 5 bases, it adds a base that has both an $abs()$ operator, and an interaction term: $abs(dvthn) * dvthn$. It keeps adding bases up to a maximum of 46 bases. By the time it gets to 46 bases, it has actually started using a rational model, as indicated by the $/(1 + \ldots)$ term.

Table 1-13 shows some functions that FFX extracted for voltage reference $DVREF$. It always determines that a rational with a constant numerator is the best option. It uses the hinge basis functions, including interactions when 3 or more bases are used. It only needs 8 bases (in the denominator) to capture error of 0.9%. Of the 105 possible variables, FFX determined that variable $dvthn$ was highly useful, by reusing it in many ways. $dvthp$ and dxw also had prominence.

Table 1-12. Equations for opamp PM, extracted by FFX.

# Bases	**Test error (%)**	**Extracted Function**
0	15.5	59.6
1	6.8	$59.6 - 0.303 * dxl$
2	6.6	$59.6 - 0.308 * dxl - 0.00460 * cgop$
3	5.4	$59.6 - 0.332 * dxl - 0.0268 * cgop + 0.0215 * dvthn$
4	4.2	$59.6 - 0.353 * dxl - 0.0457 * cgop + 0.0403 * dvthn - 0.0211 * dvthp$
5	4.1	$59.6 - 0.354 * dxl - 0.0460 * cgop - 0.0217 * dvthp + 0.0198 * dvthn + 0.0134 * abs(dvthn) * dvthn$
$\vdots$	$\vdots$	$\vdots$
46	1.0	$(58.9 - 0.136 * dxl + 0.0299 * dvthn - 0.0194 * max(0, 0.784 - dvthn) + \ldots)/(1.0 + \ldots)$

8. Related Work

Related Work in GP

Some GP papers use regularized learning. (McConaghy and Gielen, 2009) runs gradient directed regularization on a large set of enumerated basis functions, and uses those to bias the choice of function building blocks during GP search. FFX is similar, except it does not perform GP after regularized learning, and does not exploit pathwise learning to get a tradeoff. (Nikolaev and Iba, 2001) and (McConaghy et al., 2005) use ridge regression and the PRESS statistic, respectively, as part of the individual's fitness function.

Table 1-13. Equations for voltage reference $DVREF$, extracted by FFX.

# Bases	Test error (%)	Extracted Function
0	2.6	512.7
1	2.1	$504/(1.0 + 0.121 * max(0, dvthn + 0.875))$
2	1.8	$503 - 199 * max(0, dvthn + 1.61) - 52.1 * max(0, dvthn + 0.875)$
⋮	⋮	⋮
8	0.9	$476/(1.0 + 0.105 * max(0, dvthn + 1.61) - 0.0397 * max(0, -1.64 - dvthp) * max(0, dvthn + 0.875) - \ldots)$

Some GP research recasts SR from tree-valued problems towards vector-valued optimization problems. (O'Neill and Brabazon, 2006) casts SR into a string-based space, then solves it with a differential-evolution (DE) variant of grammatical evolution (O'Neill and Ryan, 2003). (McConaghy and Gielen, 2006) casts SR into a vector-valued Euclidian space, but solve it with a combination of vector-valued and traditional tree-valued operators in an EA framework. (Fonlupt and Robilliard, 2011) and others cast SR into a vector-valued Euclidian space, then solve it with vector-valued DE. (Korns, 2010) casts SR into a vector-valued space, and solves with Particle Swarm Optimization. (Topchy and Punch, 2001) and others cast the sub-problem of learning SR coefficients into traditional real-valued optimization problems as the inner loop of memetic learning; the outer loop remains GP-style search.

There are several approaches that recast general tree-valued search into simpler spaces; (Rothlauf, 2006) is a good starting point.

Shifting towards deterministic behavior, Estimation of Distribution Algorithms (EDAs) are sometimes framed as "derandomized" algorithms (Hansen and Ostermeier, 2001)[1]. EDAs have been applied to tree-based search; a recent example is (Looks, 2006). Variance-reduction techniques have also been used to derandomize EAs, such as (Teytaud and Gelly, 2007).

(O'Reilly, 1995) is a thorough example of doing tree-based search with non-evolutionary algorithms (hill climbing, simulated annealing).

Of course, none of these approaches are really *that* closely related to FFX. FFX dispenses with selection, mutation, and crossover. It has no individuals, and no population. At its core, it simply casts SR as one (or two) convex optimization problems, and solves them with off-the-shelf algorithms.

[1]The authors claim CMA-ES is a "completely" derandomized algorithm, but that is not quite accurate, because CMA-ES still relies on drawing samples from a pdf. To be *completely* derandomized, an algorithm has to be deterministic.

GLMs and Universal Approximation

Researchers familiar with generalized linear models (GLMs) may see FFX "merely" as a particular choice of "basis expansions"; for example (Hastie et al., 2008) suggests possible expansions including $log(x_j)$ and $\sqrt{x_j}$. The benefit of this, of course, is that GLM theory applies directly to FFX. However, this sells FFX short; consider the usefulness of other "merely GLM" techniques like CART (with indicator-style bases), MARS (with hinge-function bases), and SVMs (with kernel bases). Their usefulness is precisely due to a particular choice of basis functions, with appropriate algorithmic support framework, and a thoughtfully-chosen application. In our case, the choice of basis functions is driven from an SR perspective; the algorithmic support framework makes special use of the pathwise regularized learning, and includes nondominated filtering; the application is the SR-derived aim to generate whitebox models trading off prediction error versus complexity; and finally the scalable variant of FFX has "batch" stepwise-forward regression and hinge functions.

FFX shares a related philosophy with SVMs: transform a lower-dimensional set of n linear bases to a much larger set of bases $n_{new} \gg n$; then apply linear learning on this larger set, but prune them down (in the case of SVMs, to a set of "support vectors"). Of course, the distance calculation in SVM basis function of $||\boldsymbol{x} - \boldsymbol{x_{svi}}||$ for support vector i is not naturally interpretable, so does not apply to SR problems.

With a sufficiently broad choice of basis functions, FFX would be a universal approximator. But as already discussed, the aim of *specific* GLM techniques is to thoughtfully choose basis functions that reflect their aims. FFX's basis functions FFX are not sufficiently general to give FFX universal approximation. Hinge functions help, but to make FFX fully universal we would need to add more threshold values, and allow iteration to higher orders (similar to MARS). Of course, doing this hurts interpretability.

FFX is not a panacea: because its functional form is not naturally a universal approximator, there will be classes of SR problems that it handles poorly. For example, it cannot tune the coefficients $\{w_0, w_1\}$ inside a nonlinear basis function like $sin(w_0 + w_1 * x_1)$. This is not unlike other "technologies": linear regression can only competently handle linear and weakly nonlinear models; convex optimization can only handle unimodal problems; and so on. But what they trade off for flexibility, they gain in speed and reliability. To our knowledge, of the regression "technologies" that output interpretable models, FFX covers the broadest class of functions. And as we have seen, even with these restrictions, FFX is extremely competitive with GP-SR in finding accurate models on real-world data.

9. Conclusion

This paper presented FFX, a new SR technique that approaches "technology" level speed, scalability, and reliability. Rather than evolutionary learning, it uses a recently-developed technique from the machine learning literature: pathwise regularized learning (Friedman et al., 2010). FFX applies pathwise learning to an enormous set of nonlinear basis functions, and exploits the path structure to generate a set of models that trade off error versus complexity. FFX was verified on six real-world medium-sized SR problems: average training time is ≈5 s (compared to 10 min with GP-SR), prediction error is comparable or better than GP-SR, and the models are at least as compact. FFX was scaled up to perform well on real-world problems with >1000 input variables. Due to its simplicity and deterministic nature, FFX's computational complexity could readily be determined: $O(N * n^2)$; where N is number of samples and n is number of input dimensions.

A python implementation of FFX, along with the real-world benchmark datasets used in this paper, are available at trent.st/ffx.

FFX's success on a problem traditionally approached by GP raises several points. First, stochasticity is not necessarily a virtue: FFX's deterministic nature means no wondering whether a new run on the same problem would work. Second, this paper showed how doing SR does not have to mean doing GP. What about other problems traditionally associated with GP? GP's greatest virtue is perhaps its convenience. But GP is not necessarily the only way; there is the possibility of dramatically different approaches. The problem may be reframed to be deterministic or even convex. As in the case of FFX for SR, there could be benefits like speed, scalability, simplicity, and adoptability; plus a deeper understanding of the problem itself. Such research can help crystallize insight into what problems GP has most benefit, and where research on GP might be the most fruitful; for example, answering specific questions about the nature of evolution, of emergence and complexity, and of computer science.

10. Acknowledgment

Funding for the reported research results is acknowledged from Solido Design Automation Inc.

References

Ampazis, N. and Perantonis, S. J. (2002). Two highly efficient second-order algorithms for training feedforward networks. *IEEE-EC*, 13:1064–1074.

Boyd, Stephen and Vandenberghe, Lieven (2004). *Convex Optimization*. Cambridge University Press, New York, NY, USA.

Castillo, Flor, Kordon, Arthur, and Villa, Carlos (2010). Genetic programming transforms in linear regression situations. In Riolo, Rick, McConaghy, Trent, and Vladislavleva, Ekaterina, editors, *Genetic Programming Theory and Practice VIII*, volume 8 of *Genetic and Evolutionary Computation*, chapter 11, pages 175–194. Springer, Ann Arbor, USA.

Daems, Walter, Gielen, Georges G. E., and Sansen, Willy M. C. (2003). Simulation-based generation of posynomial performance models for the sizing of analog integrated circuits. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 22(5):517–534.

Deb, Kalyanmoy, Pratap, Amrit, Agarwal, Sameer, and Meyarivan, T. (2002). A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE Transactions on Evolutionary Computation*, 6:182–197.

Fonlupt, Cyril and Robilliard, Denis (2011). A continuous approach to genetic programming. In Silva, Sara, Foster, James A., Nicolau, Miguel, Giacobini, Mario, and Machado, Penousal, editors, *Proceedings of the 14th European Conference on Genetic Programming, EuroGP 2011*, volume 6621 of *LNCS*, pages 335–346, Turin, Italy. Springer Verlag.

Friedman, J. H. (1991). Multivariate adaptive regression splines. *Annals of Statistics*, 19(1):1–141.

Friedman, Jerome H., Hastie, Trevor, and Tibshirani, Rob (2010). Regularization paths for generalized linear models via coordinate descent. *Journal of Statistical Software*, 33(1):1–22.

Hansen, N. and Ostermeier, A. (2001). Completely derandomized self-adaptation in evolution strategies. *Evolutionary Computation*, 9(2):159–195.

Hastie, Trevor, Tibshirani, Robert, and Friedman, Jerome (2008). *The elements of statistical learning: data mining, inference and prediction*. Springer, 2 edition.

Kim, Minkyu, Becker, Ying L., Fei, Peng, and O'Reilly, Una-May (2008). Constrained genetic programming to minimize overfitting in stock selection. In Riolo, Rick L., Soule, Terence, and Worzel, Bill, editors, *Genetic Programming Theory and Practice VI*, Genetic and Evolutionary Computation, chapter 12, pages 179–195. Springer, Ann Arbor.

Korns, Michael F. (2010). Abstract expression grammar symbolic regression. In Riolo, Rick, McConaghy, Trent, and Vladislavleva, Ekaterina, editors, *Genetic Programming Theory and Practice VIII*, volume 8 of *Genetic and Evolutionary Computation*, chapter 7, pages 109–128. Springer, Ann Arbor, USA.

Koza, John R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA.

Langley, Pat, Simon, Herbert A., Bradshaw, Gary L., and Zytkow, Jan M. (1987). *Scientific discovery: computational explorations of the creative process*. MIT Press, Cambridge, MA, USA.

Leung, Henry and Haykin, Simon (1993). Rational function neural network. *Neural Comput.*, 5:928–938.

Looks, Moshe (2006). *Competent Program Evolution*. Doctor of science, Washington University, St. Louis, USA.

McConaghy, Trent, Eeckelaert, Tom, and Gielen, Georges (2005). CAFFEINE: Template-free symbolic model generation of analog circuits via canonical form functions and genetic programming. In *Proceedings of the Design Automation and Test Europe (DATE) Conference*, volume 2, pages 1082–1087, Munich.

McConaghy, Trent and Gielen, Georges (2005). Analysis of simulation-driven numerical performance modeling techniques for application to analog circuit optimization. In *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE Press.

McConaghy, Trent and Gielen, Georges (2006). Double-strength caffeine: fast template-free symbolic modeling of analog circuits via implicit canonical form functions and explicit introns. In *Proceedings of the conference on Design, automation and test in Europe: Proceedings*, DATE '06, pages 269–274, 3001 Leuven, Belgium, Belgium. European Design and Automation Association.

McConaghy, Trent and Gielen, Georges G. E. (2009). Template-free symbolic performance modeling of analog circuits via canonical-form functions and genetic programming. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(8):1162–1175.

McConaghy, Trent, Vladislavleva, Ekaterina, and Riolo, Rick (2010). Genetic programming theory and practice 2010: An introduction. In Riolo, Rick, McConaghy, Trent, and Vladislavleva, Ekaterina, editors, *Genetic Programming Theory and Practice VIII*, volume 8 of *Genetic and Evolutionary Computation*, pages xvii–xxviii. Springer, Ann Arbor, USA.

Montgomery, Douglas C. (2009). *Design and analysis of experiments*. Wiley, Hoboken, NJ, 7. ed., international student version edition.

Nelder, J. A. and Wedderburn, R. W. M. (1972). Generalized linear models. *Journal of the Royal Statistical Society, Series A, General*, 135:370–384.

Nikolaev, Nikolay Y. and Iba, Hitoshi (2001). Regularization approach to inductive genetic programming. *IEEE Transactions on Evolutionary Computing*, 54(4):359–375.

O'Neill, Michael and Brabazon, Anthony (2006). Grammatical differential evolution. In Arabnia, Hamid R., editor, *Proceedings of the 2006 International Conference on Artificial Intelligence, ICAI 2006*, volume 1, pages 231–236, Las Vegas, Nevada, USA. CSREA Press.

O'Neill, Michael and Ryan, Conor (2003). *Grammatical Evolution: Evolutionary Automatic Programming in a Arbitrary Language*, volume 4 of *Genetic programming*. Kluwer Academic Publishers.

O'Reilly, Una-May (1995). *An Analysis of Genetic Programming*. PhD thesis, Carleton University, Ottawa-Carleton Institute for Computer Science, Ottawa, Ontario, Canada.

Riolo, Rick, McConaghy, Trent, and Vladislavleva, Ekaterina, editors (2010). *Genetic Programming Theory and Practice VIII*, Genetic and Evolutionary Computation, Ann Arbor, USA. Springer.

Rothlauf, Franz (2006). *Representations for genetic and evolutionary algorithms*. Springer-Verlag, pub-SV:adr, second edition. First published 2002, 2nd edition available electronically.

Sacks, Jerome, Welch, William J., Mitchell, Toby J., and Wynn, Henry P. (1989). Design and analysis of computer experiments. *Statistical Science*, 4(4.409–435):409–427.

Schmidt, Michael D. and Lipson, Hod (2006). Co-evolving fitness predictors for accelerating and reducing evaluations. In Riolo, Rick L., Soule, Terence, and Worzel, Bill, editors, *Genetic Programming Theory and Practice IV*, volume 5 of *Genetic and Evolutionary Computation*, chapter 17, pages –. Springer, Ann Arbor.

Smits, Guido F., Vladislavleva, Ekaterina, and Kotanchek, Mark E. (2010). Scalable symbolic regression by continuous evolution with very small populations. In Riolo, Rick, McConaghy, Trent, and Vladislavleva, Ekaterina, editors, *Genetic Programming Theory and Practice VIII*, volume 8 of *Genetic and Evolutionary Computation*, chapter 9, pages 147–160. Springer, Ann Arbor, USA.

Suykens, J. A. K., Gestel, T. Van, Brabanter, J. De, Moor, B. De, and Vandewalle, J. (2002). *Least Squares Support Vector Machines*. World Scientific, Singapore.

Teytaud, Olivier and Gelly, Sylvain (2007). Dcma: yet another derandomization in covariance-matrix-adaptation. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, GECCO '07, pages 955–963, New York, NY, USA. ACM.

Topchy, Alexander and Punch, William F. (2001). Faster genetic programming based on local gradient search of numeric leaf values. In Spector, Lee, Goodman, Erik D., Wu, Annie, Langdon, W. B., Voigt, Hans-Michael, Gen, Mitsuo, Sen, Sandip, Dorigo, Marco, Pezeshk, Shahram, Garzon, Max H., and Burke, Edmund, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 155–162, San Francisco, California, USA. Morgan Kaufmann.

Zou, Hui and Hastie, Trevor (2005). Regularization and variable selection via the elastic net. *Journal Of The Royal Statistical Society Series B*, 67(2):301–320.

Index

Benchmark problem, 23
Computational complexity, 16–17, 23
Convex, 2, 21–23
Convex optimization, 2, 21–23
Derandomized algorithm, 21
Deterministic, 2–3, 21, 23
Elastic net, 4, 10
Estimation of Distribution Algorithm, 21
Fast Function Extraction, 3, 5–18, 20–23
FFX, 3, 5–18, 20–23
Generalized linear model, 4, 22
GLM, 4, 22
GP adoption, 2
Integrated circuit, 9, 15
Lasso, 4
Least-squares regression, 2, 4
Linear programming, 2
Linear regression, 2, 4, 8
McConaghy Trent, 1
Multi-objective, 3, 5, 8, 11–13, 16, 22
Pathwise regularized learning, 3–8, 11–13, 16, 20, 22–23
Rational functions, 8–9, 20
Real-world problems, 3, 15, 23
Regularized learning, 3–7, 11–12, 20, 22–23
Ridge regression, 4, 20
Scalability, 2–3, 9, 22–23
SR, 2–3, 7, 9–10, 13–14, 16, 18, 21–23
Stochastic, 21, 23
Symbolic regression, 2–3, 7, 9–10, 13–14, 16, 18, 21–23
Technology, 2–3, 23
Theory, 16