Declarative Equations, Compositional Strategies: Solving Differential Systems with Lazy Splines

Functional Pearl

Gershom Bazerman Jeff Polakow

Deutsche Bank

{gershom.bazerman,jeff.polakow}@db.com

Abstract

A simple Haskell encoding of Euler's method of integration is presented. From this encoding, a general solver for continuous differential equations is developed, by way of lazy splines. Various refinement strategies are introduced to improve accuracy. The result is a declarative, compositional library for solving differential equations.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; G.1.7 [*Numerical Analysis*]: Ordinary Differential Equations

Keywords streams, differential equations

1. Introduction

Differential equations, which relate the values of functions to those of their derivatives, arise in all walks of life and fields of study. Although specific systems of differential equations permit symbolic solution, in the general case, differential systems can only be solved through methods of numeric approximation. There is no single best numeric solver—depending on the class of the problem, various solvers will vary in efficiency, precision, and convergence. Furthermore, there is no general method to produce guaranteed, rather than approximate, error bounds. In the words of A.C. Hindmarsh, a pioneer in the field of ordinary differential equation (ODE) solvers: "For any given ODE solver that you like, there's always a problem for which it gives the wrong answer and the answer can be as wrong as you like; the error in it can be as large as you like." (Hindmarsh 2005)

As solvers become more complex and powerful, increasing numbers of parameters become available to modify, with their effects and interaction varying from problem to problem. For example, the introduction of adaptive methods (see section 9) to increase correctness and efficiency generally means that users are presented with more, rather than fewer choices, as they must now choose upper bounds, lower bounds, and tolerances. To quote Hindmarsh again: "Users have always had trouble choosing the tolerances that they input to these solvers, and would frequently say, 'can't you write a solver that doesn't need to ask for that information, and just chooses its own tolerance?' People have tried to do that. But in the end, to cover the full variety of problems that people solve, you have to ask people for tolerance information." (Hindmarsh 2005)

The structure of standard numeric solvers does not readily allow developers to manage and think about increasing levels of complexity. Though we cannot vanquish the complexities of numerical solvers, nor tame the proliferation of choices involved in their invocation, we can try to organize the complexity. The goal of this paper is to expose the choices and approximations that arise in numeric solving (and their attendant sequencing and relationships) using a functional, compositional style that developers can manipulate and reason about.

We begin with a differential equation solver over time series, based on Euler's method and well known in functional programming folklore. This Euler solver is progressively improved, while retaining its elegant, declarative character, and applied to increasingly complex problems to solve—survival rates, oscillating springs, and the growth of flames. Along the way, we move from the initial time series representation to one based on lazy splines—sequences of polynomials tagged with duration.

Various refinement strategies are introduced to improve accuracy and computational efficiency. These strategies are declarative and compositional. The result is not a single "solver" but rather a library, which serves as a toolkit from which a whole family of solvers may be constructed and experimented with.

Although we make heavy use of Haskell's non-strict evaluation and minor use of it's type class machinery, one could easily adapt our implementation—at the cost of some elegance—to any functional language. The code presented in this paper is self-contained, relying only on standard Haskell libraries. It is available on Hackage as the package `lazysplines`.

2. Ducks

Consider a simple problem in survival analysis. We wish to produce a survival curve for a population of ducks—a function that tells us the probability that a duck will survive to a given age. The information we have at hand is a hazard function, which is of arbitrary shape and derived from direct observation. This hazard function maps the age of ducks to their probability of death at that age (and implicitly assumes that a duck has survived up until the point that it dies, i.e., ducks can only die once).

It is tempting to determine the probability of duck death by a given age (the complement of the survival curve) simply by integrating the hazard function. After all, the chance that a duck dies by a given age is given by the sum total of the chance that it has died at any age preceding that age. However, as mentioned above, the hazard function takes into account survival probabilities—i.e., it is conditional on the survival of a duck up its time of death.

Therefore, such an approach would be incorrect, and would in fact predict that eventually the probability of duck death rises above 100 percent.

Correctly determining the probability of duck death requires constructing a system of differential equations. First, the survival curve, $duckSurvival(t)$, is defined in terms of a duck lifetime distribution function, $duckLifetime(t)$.

$$duckSurvival(t) = 1 - duckLifetime(t)$$

Next, the lifetime distribution function is defined in terms of its derivative, an event density function, $duckLifetime'(t)$. This function represents the instantaneous probability of duck death at a given age, i.e., **not** conditional on survival to said age.

$$duckLifetime(t) = \int duckLifetime'(t)\, dt$$

Finally, the density function is defined in terms of the the hazard rate, $duckDeathAtAge(t)$, and survival curve.

$$duckLifetime'(t) = duckDeathAtAge(t) * duckSurvival(t)$$

This constitutes a mutually recursive system of differential equations that presents a classic initial value problem, whose solution, given an initial value at $t = 0$, may be numerically approximated by any number of methods.

If all inputs to this problem are discretized to a regular interval (say one duck year), the above functions can be directly encoded in Haskell using time series (i.e., `[Double]`). In this encoding, lists are treated as functions whose domain is integers and whose range is doubles. Values are lifted to constant functions by use of `repeat`, pointwise operations such as multiplication and subtraction are accomplished by the standard Haskell library function `zipWith`, and integration is simply a running sum:

```
integrateList =
    snd . mapAccumL (\k x -> (k + x, k + x)) 0
```

Note that the second projection of `mapAccumL`'s result is lazily produced, and thus `integrateList` lazily produces its result; as this laziness lies at the heart of our solver, we have included the definition of `mapAccumL` in appendix A for the interested reader's convenience.

We start our encoding with a time series of observed data:

```
duckDeathAtAgeList =
    -- no deaths before 10 duck years
    replicate 10 0 ++
    -- 20% chance of death next 10 duck years
    replicate 10 0.2 ++
    -- increasing chance of death onwards
    [0.21,0.22 .. 0.35]
```

and an initial value for the equation:

```
-- no chance of death in first duck year
initialLifeList = [0]
```

We can now directly transcribe the previous definitions into Haskell:

```
survivalList =
    zipWith (-) (repeat 1) duckLifeList

duckLifeList =
    initialLifeList ++ integrateList duckLife'List

duckLife'List =
    zipWith (*) duckDeathAtAgeList survivalList
```

Thanks to the laziness of `zipWith` and `integrateList`, the definition of `survivalList` produces a time series which is an approximate numerical solution to the above formulas.

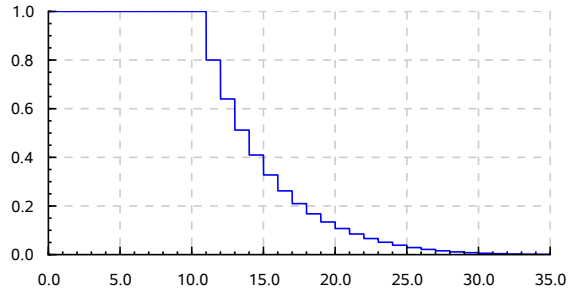

Figure 1. Duck survival, by Euler approximation (`survivalList`).

Note that the above encoding is no different in nature than the familiar formula for the Fibonacci sequence:

```
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

Specifically, these are both encodings, using a co-recursive data structure, of a form of recurrence relation—a function whose value at a point is defined by its previous values.

The solution presented here is in fact a naive Euler approximation, with a fixed timestep of one duck year (figure 1). Indeed, when working with recursive lists, we are limited to the domain of discretely defined functions, and can hope for no better. We could improve accuracy by reducing the step size, but of course the computational complexity and round-off error would increase correspondingly, In fact, the true answer would be the limit of such solutions as the timestep approached zero—obviously impossible to compute.

Having a single fixed timestep affects everything we are working with. Furthermore, wild instabilities will arise when this technique is applied to an equation that exhibits variations at an interval which interferes infelicitously with the timestep. Therefore, rather than working with lists, we would like to work directly with continuous valued functions, for both computational efficiency and improved correctness.

3. In Search of Continuity

The first obstacle to treating lists as continuous functions is simply that, to "evaluate" a function at any given value, we are indexing into a list with `!!` which, in our case, is of type `[Double] -> Int -> Double`. Clearly, if we can only index at discrete integral values, then we are working with a discrete function. We introduce the operator "at", as a typeclass, so that we may experiment with various more suitable implementations for functions than `[Double]`:

```
class Sampleable a where
    at :: a -> Double -> Double
```

The time series encoding of functions can be made an instance of `Sampleable` as follows:

```
instance Sampleable [Double] where
    at x v =  x !! truncate v
```

This implementation simply "throws out" the non-integral portion of its input, as it can make no use of it. Making time series sampleable allows us to change their semantics; rather than conceiving of lists as an imprecise representation of smooth functions, we can take the attitude that time series are in fact a precise representation of step functions, as encoded by a series of constant func-

tions.[1] The problem is no longer that we cannot deal with functions over a continuous domain—it is simply that we can only deal with a limited class of them.

To increase the range of function representation, the component pieces, or segments, can be generalized from constant functions to polynomial functions (themselves represented by `[Double]`, a list of coefficients of increasing order). The polynomial encoding yields easily differentiable and integrable piecewise continuous functions.

```
type Poly = [Double]

-- Horner's scheme for polynomial evaluation
instance Sampleable Poly where
    at x v = foldr (\c val -> c + v * val) 0 x

instance Sampleable [Poly] where
    at x v = poly `at` frac
      where
        poly = x !! int
        (int,frac) = properFraction v
```

Code for working with polynomials (as well as power series, their infinite generalization) has been developed by M. Douglas McIlroy (1999). We use a slight variation of his work, presented in appendix B.

The above encoding is very close to what we are looking for. However, it is lacking in one particular regard—we must choose a single domain size, or duration, for all segments (1 in the instance above); i.e., representing a function whose domain is 0 through 10000 requires 10000 polynomials. Not only is such a representation clunky, and far from compact, but it suffers from the same problems mentioned at the end of section 2 resulting from a fixed segment size (albeit with more information encoded in each segment). Ideally, a segment should be as large as possible to encode the information we possess within a desired accuracy, but no larger; therefore, we should allow segments of variable duration.

4. Working with Splines

Variable duration segments are accomplished simply by packing each `Poly` with the duration for which it is valid.

```
type PolySegment = (Double, Poly)
```

This representation, a piecewise continuous list of arbitrary-length polynomials, is a spline—a standard tool of the trade in numeric analysis.

```
type Spline = [PolySegment]

duration :: Spline -> Double
duration = sum . map fst

maxDuration = 10000  -- an arbitrary limit

liftS :: Double -> Spline
liftS x = [(maxDuration,[x])]
```

`Sampleable` instances are easy to produce for both segments and splines.

```
instance Sampleable PolySegment where
  at (_,poly) pt = poly `at` pt

instance Sampleable Spline where
  -- assume pt >= 0
  at spline pt = go spline pt where
    go ((dur,poly):xs) v
        | v <= dur  = poly `at` v
        | otherwise = go xs (v - dur)
    go [] _ = error $
        "Sampling spline of out bounds " ++
        show spline ++ " at: " ++ show pt
```

Differentiating these lazy splines is simply done by differentiating each segment, where `diff` is polynomial differentiation as defined in appendix B[2]:

```
deriveSpline :: Spline -> Spline
deriveSpline = map (second diff)
```

Integration over lazy splines is a modification of the time series integration code, accumulating the integral of each segment, where `integ` is polynomial integration as defined in appendix B:

```
integrateSpline :: Spline -> Spline
integrateSpline =
    snd . mapAccumL go 0 . map (second integ)
  where
    go :: Double -> PolySegment -> (Double, PolySegment)
    go acc (dur,poly) = (v, seg)
      where v = acc + poly `at` dur
            seg = (dur, realToFrac acc + poly)
```

`integrateSpline f` produces a new `Spline` whose value at x represents the definite integral of `f` from 0 to x. Thus the following identity always holds[3]:

$$\texttt{integrateSpline}\,(x \texttt{ ++ } y) = \texttt{integrateSpline}\,x \texttt{ ++ } (v + \texttt{integrateSpline}\,y)$$
$$\textit{where}\quad v = \texttt{integrateSpline}\,x\ \texttt{`at`}\ \texttt{duration}\,x$$

Note that both `deriveSpline` and `integrateSpline` lazily consume and produce.

A pattern for pointwise operations on splines, inspired by Elliott (2008), may be abstracted:

```
inSpline2 :: (Poly -> Poly -> Poly) ->
             Spline -> Spline -> Spline
inSpline2 op ((xd,x):xs) ((yd,y):ys)
    | xd == yd  = (xd, v) : inSpline2 op xs ys
    | xd <  yd  = (xd, v) : inSpline2 op xs (y':ys)
    | otherwise = (yd, v) : inSpline2 op (x':xs) ys
  where
    v = x `op` y
    x' = splitPoly yd (xd,x)
    y' = splitPoly xd (yd,y)
    splitPoly d (dur,poly) = (dur - d, shiftBy d poly)
inSpline2 _ _ _ = []
```

A binary operation over two splines is applied to the first segment of each. If there is a mismatch between their sizes, a new segment is created holding the "remainder", and attached back to the rest of the spline before iteration continues. Note that the remainder segment's polynomial is suitably shifted to reflect the proper starting

[1] We do not, of course, have a precise representation of differential equations over step functions, as discretization affects both derivation and integration.

[2] While a more general version of `second` is found in `Control.Arrow`, for the purposes of this paper, readers may think of it as `second :: (b -> c) -> (a,b) -> (a,c)`

[3] This is related to the second fundamental theorem of calculus: if $f(x) = g'(x)$ then $\int_a^b f(x)\,dx = g(b) - g(a)$ where `integrateSpline y` loosely corresponds to $\int_a^b f(x)\,dx$

point. This shifting is accomplished by polynomial composition, `#` (whose definition is in appendix B):

```
shiftBy :: Double -> Poly -> Poly
shiftBy d poly = poly # [d,1]
```

In other words, shifting a polynomial function $f(x)$ by a constant d is accomplished by composing $f(x)$ with the polynomial $d + x$.[4]

We may now elegantly define a `Num` instance for `Spline`:

```
instance Num Spline where
    fromInteger = liftS . fromInteger
    negate = map (second negate)
    (+)    = inSpline2 (+)
    (*)    = inSpline2 (*)
```

A `Fractional` instance can be defined in a similar fashion, with the caveat that polynomial division diverges in a number of common cases. Using common techniques for finding real-valued zeros of polynomials, one could also implement reasonably efficient `abs` and `signum` functions. Cyclic trigonometric functions can be encoded with a high degree of precision as infinite cycles of their Taylor expansion over an appropriate range. In fact, as Karcmarcuk (2001) observed, any function which may be placed into the framework of automatic differentiation can be transformed from a lazy tower of derivatives at a point into a power series expansion of its Taylor approximation about that point. Thus, one can, almost automatically (fixing the number of terms to truncate the power series at, and the interval at which to sample the provided function), derive spline approximations for nearly any numeric function so desired.

5. Ducks, Redux

The hazard function for duck death can now be recast as a `Spline` (where we have specified linear interpolation between the sample points):

```
duckDeathAtAge = [ (10, [0]),
                   (10, [0.2]),
                   (15, [0.2, 0.01]) ]
```

After defining an initial function for survival probability over an appropriately short span of time, the original duck death problem can be encoded directly and continuously:

```
initialLife = [(1, [0])]

survival = 1 - duckLife

duckLife = initialLife ++ integrateSpline duckLife'

duckLife' = duckDeathAtAge * survival
```

In the definition of `initialLife`, an initial value has been extruded into an initial segment of non-zero length. This step, analogous to providing a set of initial values to bootstrap from in traditional multistep methods, moves us from a precise encoding of the problem to an approximate encoding of the problem. Such an approximation is necessary; if the initial segment were to have a length of zero, any solutions based on it would be equivalent to `let x = x in x`—i.e., $\perp$. Unfortunately, the choice of a non-zero length initial segment will create a host of complications for us down the road. The various ways to address these complications

[4] Note that while most operations on polynomials (i.e., finite sequences of coefficients) would work equally well on power series (i.e., infinite streams of coefficients), the `shiftBy` function diverges in the latter case, and therefore, unfortunately, the code presented here cannot be generalized to power series.

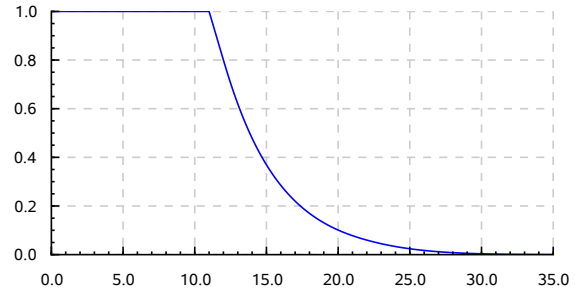

Figure 2. Duck survival, by spline approximation (`survival`).

can be viewed as the core of all numerical methods for solving systems of differential equations. We will return to this issue in subsequent sections.

We see that, as with the initial list solution of section 2, by providing a sufficiently lazy piecewise structure, and by preserving laziness in our functions, recurrence relations such as the approximate solution to a differential equation arise naturally. The graph of `survival` (figure 2) is a smoother version of that obtained earlier by the Euler method (figure 1). Direct examination of the resulting spline reveals many vanishingly small coefficients at the higher powers. It would be nice to provide users with the ability to truncate these polynomials at some given point, and hence make a trade off between precision and speed.

6. Differential Refinement Combinators

Polynomial truncation is the first of several differential refinement combinators we will introduce. These combinators modify the precision and computational efficiency of spline solutions to differential equations; they are composable building blocks of "strategies" for solving ODEs. As such, we require each differential refinement combinator, f, to satisfy the following property:

Preservation of Definition.
$\forall\,(s{::}\ \texttt{Spline})\ (x{::}\ \texttt{Double}).$
$\exists v.\ s\ \texttt{`at`}\ x = v\ \ implies\ \ \exists v'.\ f\ s\ \texttt{`at`}\ x = v'$
where $v, v' \neq \perp$

In other words, differential refinement combinators preserve the domain and laziness of splines.

While `Splines` are piecewise continuous by construction, they are not necessarily fully continuous, i.e., a `Spline`'s segments might not match up. We define continuity for a `Spline`, s, as follows:

$continuous\ s$ iff

$\forall(d_i, p_i)\ (d_{i+1}, p_{i+1}) \in s\ .\ \ p_i\ \texttt{`at`}\ d_i = p_{i+1}\ \texttt{`at`}\ 0$

$where\ \ s = [(d_0, p_0),\ (d_1, p_1)\ldots]$

We further require each differential refinement combinator, f, to satisfy the following property:

Preservation of Continuity.
$\forall\,(s{::}\ \texttt{Spline})\ .\ \ continuous\ s\ \ implies\ \ continuous\ (f\ s)$

For a large class of differential refinement combinators—those which do not alter the durations of individual segments—a higher-order function (`mapSpline`) can be produced which guarantees the above properties are satisfied. Preservation of Definition is enforced through mapping lazily over each segment of a spline in turn, preserving individual durations. In the solver we have

developed, the initial segment is directly provided by the user; therefore `mapSpline` leaves this segment unchanged. The function provided as an argument to `mapSpline` can depend both on the duration of the individual segment it operates on, as well as the sum total duration traversed thus far.

Preservation of Continuity is enforced by matching the initial value of each resultant segment to the terminal value of the segment prior. There is one complication involved. For those argument functions which preserve the initial values of the polynomials on which they operate, this matching can be done prior to their application. However, for those argument functions which do not preserve the initial values of the polynomials on which they operate, matching must be performed subsequent to their application. Furthermore, care must be taken in this matching not only to properly set their initial value, but to preserve their terminal value as well—i.e., to maintain continuity in a minimally destructive fashion. The choice of when to perform matching is determined by a boolean argument provided to `mapSpline`.

```
mapSpline :: Bool ->
             (Double -> Double -> Poly -> Poly) ->
             Spline ->
             Spline
mapSpline _           _ []         = []
mapSpline matchFirst f (seg:segs) =
    -- leave first segment unchanged
    seg :
    (snd $ mapAccumL go (dur0, seg `at` dur0) segs)
  where
    dur0 = fst seg

    go (totalDur, lastVal) (dur, poly) =
        ((totalDur + dur, fnc' `at` dur), fnc')
      where
        fnc' = (dur, poly')

        poly'
            -- modify translated segment
          | matchFirst = f totalDur dur $
                         match lastVal poly

            -- translate modified segment
          | otherwise  = matchScale lastVal dur $
                         f totalDur dur poly

-- replace 0-degree coefficient of a Poly
match lastVal (_:xs) = lastVal:xs
match lastVal x      = [lastVal]

-- Alter the first point, preserve the point at dur
matchScale v dur poly@(x:xs) = v : map (* scale) xs
           where height = poly `at` (dur - x)
                 diff   = x - v
                 scale = height / (height - diff)
matchScale v _ x = [v]
```

With `mapSpline` in hand, production of a combinator for polynomial truncation (`trimmingTo`) is simple, as is rewriting `duckLife'` to use only polynomials of fewer than, e.g., 15 degrees.

```
infixl 1 `trimmingTo`
trimmingTo :: Spline -> Int -> Spline
trimmingTo spline power =
    mapSpline True go spline
  where
    go _ _ s = take power s
```

```
survival = 1 - duckLife

duckLife = initialLife ++ integrateSpline duckLife'

duckLife' = duckDeathAtAge * survival
    `trimmingTo` 15
```

The placement of `trimmingTo` is somewhat arbitrary and, while not noticeable in this example, different placements can result in noticeably different results. In general, we believe that it makes the most sense to trim at the lowest derivative, as any later placement would generate additional higher-order coefficients only to discard them.

The choice of a maximum polynomial length roughly corresponds to the choice of order for a traditional ODE solver. Hence, while in this case it appears to present a simple time/accuracy tradeoff, the story will not always be so clear. However, for the examples in this paper, coefficients beyond order fifteen will never have a noticeably positive impact on the result, and so we will apply (`` `trimmingTo` 15``) as a matter of course to speed up calculations without sacrificing noticeable precision.

7. Springs

We appear to be on the right track. But how do we fare with a more complicated equation? The following system of differential equations describes the oscillation of a mass at the end of a spring, such that its acceleration at any given point is a function of its position.

$$spring''(t) = -36 * spring(t)$$

$$spring(t) = \iint spring(t)\, dt^2$$

with the following initial values:

$$spring(0) = -0.5 \quad spring'(0) = 1$$

As above, the mathematical equations can be transcribed in a fairly straightforward manner:

```
initialSpring = [(0.01, [-0.5, 1, 18])]

spring'' = -36 * spring
    `trimmingTo` 15

spring = initialSpring ++
           liftS (initialSpring `at` 0.01) +
           integrateSpline (integrateSpline spring'')
```

A few issues are worth particular note. First, as with the equation in section 2, an appropriate initial segment duration must be chosen. Too small, and needless work will be performed, but too large and accuracy will diminish greatly. The length of the initial segment is chosen here for illustrative purposes—large enough to illustrate flaws in our approach, but small enough so as not to produce answers that are thoroughly outlandish. Second: Unlike the duck equation, the initial segment of the spring equation is not 0. Thus, its terminal value must be explicitly added to the second integral of `spring''`, as indicated by the identity on `integrateSpline` noted in section 4. Finally: Our initial step, as determined by initialSpring, includes a second degree coefficient. This coefficient was derived by substituting the initial value of $spring(0)$ into the equation for $spring''$.[5]

[5] In general, for the methods we describe, a more accurate initial segment will yield more accurate results. Still better accuracy may at times be achieved by performing additional calculations to yield the initial seg-

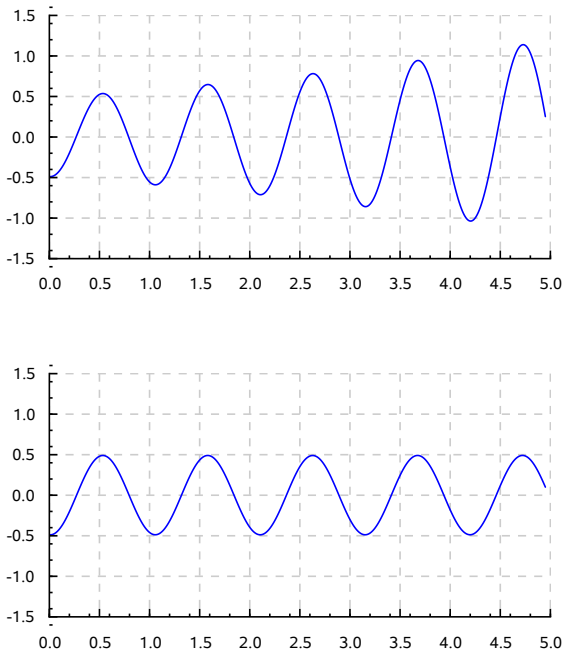

Figure 3. Spring equation, without (above) and with (below) delay correction.

Analytically, one would expect the above equations to provide a cosine function. However, the plot of the Haskell solution yields something quite different (figure 3, top). Clearly, something is missing from the exposition thus far. What went wrong?

Examining the above code more closely, and keeping in mind the `Sampleable` instance for `Spline` from section 4, we notice that `spring ‘at‘ t` is not determined by `spring’’ ‘at‘ t` but rather `spring’’ ‘at‘ (t - 0.01)`, where 0.01 is the duration of `initialSpring`. Due to that pesky non-zero initial segment length, the above functions do not solve the intended equation at all, but rather a closely related delay differential equation. Some mechanism is necessary to close up the interval introduced by the initial segment. One method of doing so is to replace every element of `spring’’ ‘at‘ t` by a modified element, approximating `spring’’ ‘at‘ (t + 0.01)` through forward polynomial extrapolation, i.e., producing a differential refinement combinator that performs an appropriate amount of shifting on each polynomial segment.

```
infixl 1 ‘extrapForward‘
extrapForward :: Spline -> Double -> Spline
extrapForward spline delta =
    mapSpline False go spline
  where
    go _ _ s = shiftBy delta s
```

We pass the parameter `False` to `mapSpline` since `shiftBy` potentially changes a segment’s starting value; it must be applied to a segment before translation to preserve continuity.

ment, such as determining the coefficients via a single-step method such as implicit Euler or Runge-Kutta. These methods, as well as adaptive methods for determining initial step size, are known fields of study which are beyond the scope of this paper.

The spring equation may now be written in a manner which corrects for delay:

```
spring2’’ = -36 * spring2
    ‘trimmingTo‘ 15
    ‘extrapForward‘ 0.01

spring2   = initialSpring ++
              liftS (initialSpring ‘at‘ 0.01) +
              integrateSpline
                 (integrateSpline spring2’’)
```

Indeed, the result is now a periodic cycle between -0.5 and 0.5 (figure 3, bottom). As with `trimmingTo`, the placement of `extrapForward` is somewhat arbitrary. However it makes sense to conduct forward extrapolation immediately after trimming, as extrapolation is more stable and efficient when carried out over lower-order polynomials.

8. Flames

Up to this point, our approach has been an **explicit** method—i.e., one in which the state of a system is calculated directly from its state at prior points in time. In fact, our approach relies on a tacit “claim” that a polynomial approximation of a segment of the derivative of a function is a good predictor for the future values of that segment. This is not always a good assumption. In particular, a certain class of “stiff” equations will cause any explicit method to be be quite unstable. Stiffness is an elusive quality, not immediately apparent in the plot of a function. One way to think of stiff equations is that they posses components with a mix of time scales, where there is a rapid damping associated with the shorter timescale (Hindmarsh 2005).

The following system of equations, modeling the growth of a flame, is a good example of ”stiffness”:

$$flame'(t) = flame(t)^2 * (1 - flame(t))$$

$$flame(t) = \int flame'(t)\, dt$$

$$flame(0) = 0.01$$

This function starts out with a gradual slope, which begins to increase very rapidly at around 95, and then levels off sharply to a constant value of 1 at around 110 (Moler 2003).

The above equations can be transcribed into Haskell in a familiar fashion. As in section 7, an additional initial coefficient has been computed by substituting the initial value into the system of equations. We choose here an initial segment of length one, as it is large enough that differences in the quality of various solutions will be evident, but small enough that the solutions will not be absurdly poor.

```
initialFlame = [(1, [0.01, 9.9e-5])]

flame’ = flame^2 * (1 - flame)
    ‘trimmingTo‘ 15
    ‘extrapForward‘ 1

flame  = initialFlame ++
           (liftS (initialFlame ‘at‘ 1) +
           integrateSpline flame’)
```

As we feared, this code does not produce a good answer and begins to diverge into incoherence at some point around 109—i.e., at the “stiff” portion, where even a small change in `flame` yields a significant change in `flame’`; for example, `flame ‘at‘ 110` evaluates to `4.71e8`.

The divergence is an artifact caused by higher-order polynomials. Higher order polynomials, under forward extrapolation, will exaggerate effects that occur in too rapid a timescale. In the stiff region, i.e., that area characterized by a relatively rapid damping, this exaggeration rapidly multiplies and the solution diverges. Things can be brought back under control by limiting the size of polynomials to a low order— i.e., using `(‘trimmingTo‘ 2)` instead of `(‘trimmingTo‘ 15)`. Although this solution does not diverge, it is nonetheless very poor. One can also obtain a solution that does not diverge by greatly reducing the size of the initial step, but only at a significant computational cost. Clearly, we will need to expand the stock of our techniques in order to get a decent result in a reasonable span of time.

In fact, when dealing with stiff problems, it is necessary to use **implicit** methods—i.e., methods which involve (numerically) solving an equation dependent on the state of the system at prior points in time *and the current point in time*.[6] Explicit forward polynomial approximation now becomes the first half of a predictor-corrector pair. At each step, we add an implicit corrector component, calculated by the minimization of some constraint. To minimize constraints, we will need a root solver. The root solver used in this paper (for which code is provided in appendix C) has the following type signature:

```
findValue :: Double ->
             (a -> Double) ->
             (Double -> a) ->
             a
```

Given a tolerance, a fitness function on some value and a method of generating such a value from a `Double`, `findValue` finds a value suitable to within the specified tolerance.

The choice of a corrector component can yield a variety of properties. For this paper, we have chosen to use a function that assumes that the initial value and “shape” of each polynomial component are correct, but that it may be scaled wrongly. This additional component may be viewed as a translated and scaled version of the original.

```
scaleRest :: Poly -> Double -> Poly
scaleRest (x:xs) c = x : map (* c) xs
```

`scaleRest` has the convenient property that, as c approaches zero, the result becomes increasingly flat. Thus, it is particularly suited to taming the wild swings brought on by forward extrapolation in regions of stiffness.

Now we introduce a `SplinePredicate`—a fitness function, to be applied to each polynomial component of a spline. These predicates are allowed to depend upon the start time of the component (i.e., the sum of the durations of all preceding segments) as well as its duration. Although time varying predicates are not used in this paper, it is not hard to imagine examples requiring such generality.

```
type SplinePredicate =
    Double -> Double -> Poly -> Double
```

With these tools in hand in hand, we can produce a new differential refinement combinator that given a spline, a tolerance and a predicate, iteratively corrects each segment of the spline to minimize the predicate.

```
infixl 1 ‘satisfying‘
satisfying :: Spline ->
              (Double, SplinePredicate) ->
              Spline
satisfying spline (tol, p) =
    mapSpline True go spline
  where
    go t d fnc = findValue tol (p t d) $
                 scaleRest fnc
```

Adding this new constraint to the previous function is easy. We begin with an appropriate `SplinePredicate`[7]:

```
flamePred :: SplinePredicate
flamePred t d f  = v’ - v^2 * (1 - v)
  where v  = f ‘at‘ d
        v’ = diff f ‘at‘ d
```

This predicate, minimizing defect at the end of each segment, is trivially algebraically derived from the initial equation. The mechanism provided is very general, and one might choose to minimize any derived equation—e.g., solving for defect not only at the endpoint of a segment, but across its whole span.

As with initial step size, the tolerance used here has been chosen for illustrative purposes.

```
flame2’ = flame2^2 * (1 - flame2)
    ‘trimmingTo‘ 15
    ‘extrapForward‘ 1

flame2 = initialFlame ++
           liftS (initialFlame ‘at‘ 1) +
           integrateSpline flame2’
    ‘satisfying‘ (0.00001, flamePred)
```

Indeed this now produces a reasonable answer (figure 4, top).

Note that the introduction of a `SplinePredicate` means that the formula must be written twice—once at the value level, as a direct recurrence, and once as a predicate, in the form of an expression to minimize. Furthermore, the placement of `satisfying` is fixed by the predicate, which should be passed a piece of `flame2` rather than a piece of `flame2’`. [8]

The same approach—calculating a derived equation that, if the solution were exact, would be zero, can be used to generate a graph of local defect across the whole of a spline. The following function, given any solution to the flame equation, yields a spline which is its pointwise local defect[9]:

```
flameDefect f = deriveSpline f - (f ^ 2) * (1 - f)
```

This gives a convenient measure of the accuracy of solutions (figure 4, bottom).

9. Flames, Reframed

While the solver as developed thus far is accurate, it is nonetheless not very adaptive. A step size, once fixed by the initial segment, remains so for the whole of a solution. Similarly, a maximum order, once fixed by `trimmingTo`, remains so for the whole of a solution.

[6] This presents a suggestive symmetry: Explicit solvers generate the fixpoint of a recurrence relation. Implicit solvers also involve, at each step, a numerical solution to the fixpoint of a derived relation.

[7] Note that we have not used the equivalent and more natural definition `diff f - f^2 * (1-f)) ‘at‘ d` for `flamePred` because polynomial multiplication is very expensive.

[8] Due to the way we have written the various combinators, a predicate must be written in terms of a function and its derivatives. This limits us to adding implicit constraints only to, depending how it is viewed, the lowest derivative or highest integral of any given function. Alternate, slightly more complicated, versions of these functions are possible, which pass to predicates not only a single spline segment, but an infinite tower of its integrals.

[9] We use the term “defect” rather than “error” since it does not measure the distance from the actual solution to a system of differential equations, but rather the extent, at each point in time, to which the approximate solution fails to satisfy the system.

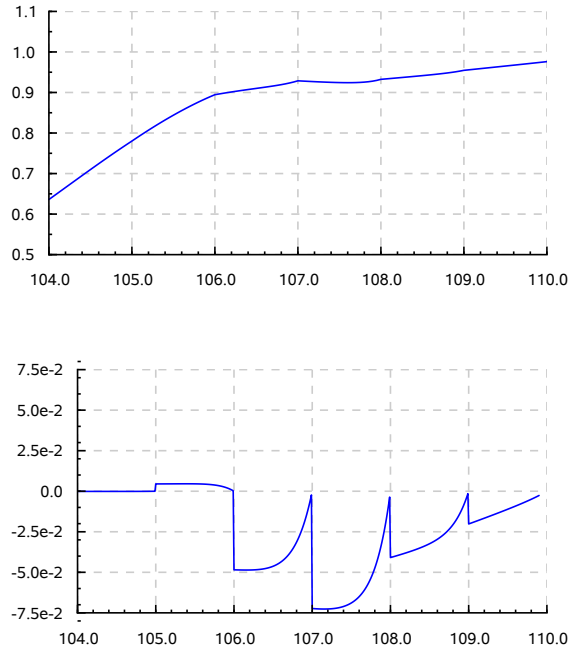

Figure 4. `flame2` (above) and `flameDefect flame2` (below), in the region of stiffness.

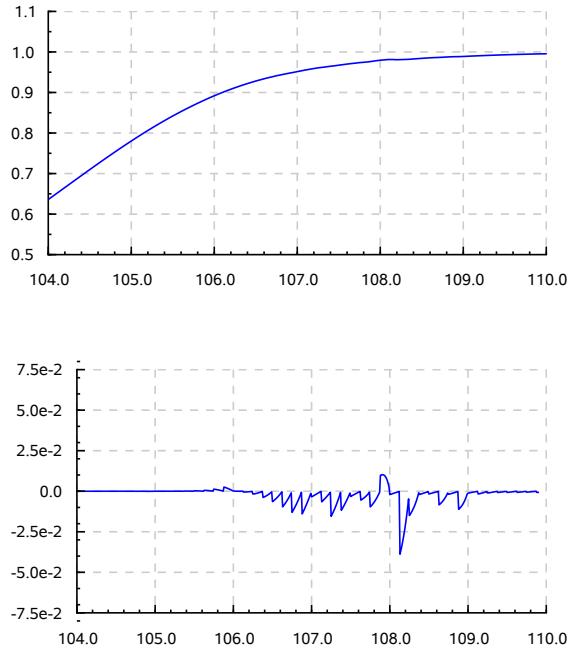

Figure 5. `flame3` (above) and `flameDefect flame3` (below), in the region of stiffness.

Some parts of a function may be very smooth and stable, possessing only components which change relatively slowly, while other parts of a function may operate on a very different timescale. Clearly, there is something to be gained by tightening the step only when needed, and lengthening the stride when permissible. Similarly, it would be far better to use polynomials of higher degree when they produce better approximations, but to discard higher order polynomial coefficients when they degrade the accuracy of calculations. To accomplish this, a new set of differential refinement combinators must be introduced.

The first combinator, `splitWhen`, recursively splits each segment of a spline in half when it fails to satisfy a given predicate within some tolerance, and as limited by some minimum duration.

```
infixl 1 `splitWhen`
splitWhen :: Spline ->
             (Double, Double, SplinePredicate) ->
             Spline
splitWhen spline (tol, minsize, p) =
    go 0 spline
  where
    go _ [] = []
    go t (f@(dur,poly):fs)
        | doSplit   = go t (f' : f'' : fs)
        | otherwise = f : go t' fs
      where

        -- split if predicate is not satisfied
        -- and duration is not below minsize
        doSplit = dur > minsize &&
                  abs (p t dur poly) > tol

        -- two segments, each with half
        -- the duration of the original
        dur' = dur / 2
        f'   = (dur', poly)
        f''  = (dur', shiftBy dur' poly)

        t' = t + dur
```

This splitting function can then be applied within the recursive "knot" of a differential solver. Here it is placed before the `satisfying` clause, so that the solution may immediately be improved. `splitWhen` on its own has no effect on the numeric results of a solution. Thus, if the order were reversed, there would be no improvement in the accuracy of a result until the following step, after the smaller segments had passed through the recursive loop.

```
flame3' = flame3^2 * (1 - flame3)
    `trimmingTo` 15
    `extrapForward` 1

flame3 = initialFlame ++
           liftS (initialFlame `at` 1) +
           integrateSpline flame3'
    `splitWhen`  (0.00001, 0.125, flamePred)
    `satisfying` (0.00001, flamePred)
```

Indeed, this solution with splitting noticeably improves accuracy (figure 5).[10]

`splitWhen` determines when time should be sacrificed for accuracy. Now we provide a combinator which serves as its inverse. `extendWhen` determines when accuracy should be sacrificed for time. Segments are extended until a predicate exceeds some given tolerance, as limited by some maximum duration. This extension

[10] The tolerances for various differential refinement combinators may obviously be set individually. However, for the sake of simplicity, in `flame3` and all following equations, we take all tolerances to be uniform.

is accomplished recursively doubling segment length. Once a segment is extended, that additional duration must be chopped from subsequent segments, until the process is ready to begin again.

```
infixl 1 `extendWhen`
extendWhen :: Spline ->
              (Double, Double, SplinePredicate) ->
              Spline
extendWhen spline (tol,maxlen,p) =
    go (spline `at` 0, 0) 0 spline
  where
    go _ _ [] = []
    go (lastVal, time) chop ((oldDur,oldPoly):fs)

          -- This segment has been subsumed
          -- by previous extensions,
          -- disregard it.
        | dur <= 0  =
            go (lastVal, time) (negate dur) fs

          -- Attempt to extend this segment
        | otherwise =
            (dur', poly') :
            go (lastVal', time') chop' fs
      where

        -- The segment as chopped
        -- to reflect prior extensions
        poly = shiftBy chop oldPoly
        dur  = oldDur - chop

        chkPred d =
            (abs $ p time d poly) < tol

        -- greatest permissible duration
        -- that satisfies the predicate
        dur' = lastDef dur .
               takeWhile chkPred .
               -- list of possible durations
               takeWhile (<= maxlen) $
               iterate (* 2) dur

        chop' = dur' - dur
        time' = time + dur'
        lastVal' = poly `at` dur'
        poly' = matchScale lastVal dur' poly

lastDef def []       = def
lastDef _ [x]        = x
lastDef def (x : xs) = lastDef def xs
```

Again, the introduction of this differential refinement combinator is compositional. It is placed here following `satisfying`, as it makes sense to try to extend results only after they have been adjusted.

```
flame4' = flame4^2 * (1 - flame4)
    `trimmingTo` 15
    `extrapForward` 1

flame4 = initialFlame ++
           liftS (initialFlame `at` 1) +
           integrateSpline flame4'
    `splitWhen`  (0.00001, 0.125, flamePred)
    `satisfying` (0.00001, flamePred)
    `extendWhen` (0.00001, 8, flamePred)
```

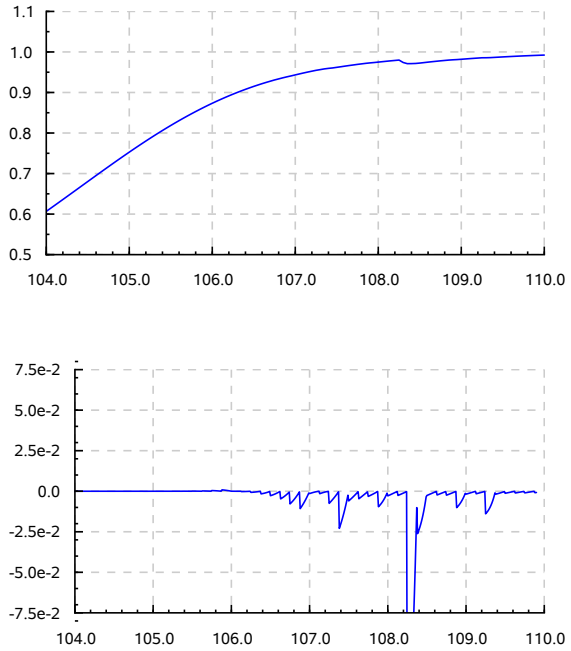

Figure 6. `flame4` (above) and `flameDefect flame4` (below), in the region of stiffness.

This solution, as one would expect, is noticeably worse (figure 6), but also somewhat faster—taking 5.66 seconds in `ghci` to evaluate to 200, as compared to 8.03 seconds for `flame3`.[11]

10. Adaptive Order

Having introduced an adaptive step size, we now turn our attention to the order of polynomials. It is bothersome that the order of polynomial chosen is at once so important to the result, and so arbitrary. Too large and the equation is unstable. Too small, and accuracy falls drastically. Rather than the naive `trimmingTo` function, we introduce a combinator that takes a a `SplinePredicate`, and attempts to choose an order intelligently, only keeping terms that improve the fitness of the result. At least two terms are always preserved, as `satisfying` requires at least one term beyond the initial value in order to modulate the result.

```
infixl 1 `trimSmart`
trimSmart :: Spline ->
             SplinePredicate ->
             Spline
trimSmart spline p =
    mapSpline True go spline
  where
    go t dur poly =
        headDef poly .
        map fst .
        dropWhile chkPred $
        -- pair adjacent candidate
        -- polynomials
        zip polys (drop 1 polys)
      where
```

[11] All measurements done on an AMD Opteron 8300.

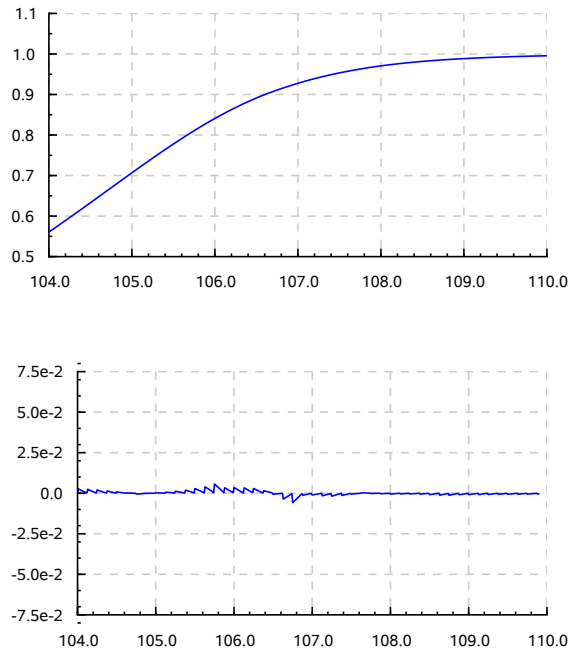

Figure 7. `flame5` (above) and `flameDefect flame5` (below), in the region of stiffness.

```
        -- possible trimmed polynomials
        polys = drop 2 . inits $  poly

        chkPred (x,x') = p t dur x >
                         p t dur x'

headDef def []  = def
headDef _ (x:_) = x
```

Adding this new modification is again compositional. We place `trimSmart` so as to discard undesirable coefficients as early as possible.

```
flame5' = flame5^2 * (1 - flame5)
    `extrapForward` 1

flame5 = initialFlame ++
           liftS (initialFlame `at` 1) +
           integrateSpline flame5'
    `trimSmart`  flamePred
    `splitWhen`  (0.00001, 0.125, flamePred)
    `satisfying` (0.00001, flamePred)
    `extendWhen` (0.00001, 8, flamePred)
```

This provides a significant boost in speed and precision. In fact, the preceding code, which yields an excellent estimation (figure 7), executes (as interpreted in `ghci`) to 200 in 0.71 seconds.

11. Related Work

The method presented here is distinct from any the authors have encountered, but has many similarities to the Adams method (Brown et al. 1989). Like the Adams method, it is a multistep method that uses forward polynomial extrapolation. At each step, an Adams method of order n constructs an interpolating polynomial based on the values of the previous n steps. The method here, on the other hand, creates a polynomial via a direct translation of the system of differential equations into Haskell functions over splines. In both cases, segments are determined by estimations at previous points. However, in the method described here, the combination of these estimations is not necessarily linear.

Lazy differential techniques over time series have been discussed previously by Karcmarcuk (1999), and over power series by M. Douglas McIlroy (1999, 2001). Work on functional reactivity and streams is of course very related as well (Elliott 2008; Rutten 2003). Much of this material, including its relation to finite calculus, has been expanded on in a recent Functional Pearl (Hinze 2008). Our innovation is simply in recognizing the power we gain by combining these preexisting techniques and applying them to the field of approximate differential solving. Also of particular interest is Pavlovic and Escardo (1998), which explains that "when applying standard methods for solving differential equations, we are actually using coinduction without even realizing it." Unsurprisingly, we have seen that approximating differentials is coinductive as well.

12. Discussion

Proceeding by simple steps, we have produced a reasonably performant, implicit/explicit, variable step, variable order differential solver, built with a combination of numeric and algebraic techniques. In this presentation, we have sometimes sacrificed efficiency in the service of clarity. Certain aspects, particularly in dealing with stiff equations, have significant room for improvement. In any case, we do not claim to have invented a better general solver for differential equations.

Like any in the zoo of differential solvers, there are some cases where the sort of solver we have presented is appropriate, and many where a better choice is possible. Traditional methods are optimized for efficient matrix calculation, while ours is forced to perform increasingly expensive polynomial calculations. Furthermore, when we work with traditional methods, we have a great deal of knowledge about their characteristics—their stability, their local error, and so forth.

However, the method we describe does possess a few particularly useful properties. Its result is not a single number at a point, but a declarative description of the entirety of a function. Partial results are memoized in the data structure as we proceed. Delay differential equations, normally a more complicated addition to differential solvers, fall naturally out of our technique.[12]

Furthermore our representation, like many functional programming techniques, is good to think with, and excellent to explore with. Traditional multistep methods include "magic numbers" of various sorts (generally coefficients), derived from a separate analytical step.[13] The method presented here does not have such numbers. Nor is it hemmed in to any particular algorithm. Rather, we have built a combinator library, with which it is possible to manipulate and modify differential expressions and approaches as first-class values within Haskell. When working with this library users are, in a sense, not applying any sort of numerical approximation algorithm as such, but simply providing a declarative statement regarding the function that they seek to model, and the appropriate strategy for their purposes. It would be worthwhile to explore what this expressivity yields in terms of formalization—for exam-

[12] At least, when they are of a constant delay. We believe that application of this technique to variable delay equations would be both possible and interesting.

[13] These numbers are distinct from tolerances, which are parameters to algorithms, not intrinsic parts of algorithms themselves.

ple, coinductive proof techniques could be used to express bounds on error.

13. Future Work

The code presented here has not been optimized for speed. We believe that both stream fusion and speculative parallelism could be applied to great effect. While we have provided some basic strategies for refining numeric integration, we have not fully studied their interactions with one another. Furthermore, other strategies can doubtless be introduced.

Minor variations in what has been described can produce a set of related techniques. Rather than doubles, polynomials may be constructed out of higher precision numeric types. More interestingly, the move from lists to splines may be viewed as a shift from an algebra over a flat basis function to an algebra over the universe of polynomials. This basis set could be augmented by exponential functions (yielding tension splines), or replaced by trigonometric functions. We believe that an efficient wavelet representation is possible as well. If an arbitrary monoid ring is taken as the source of the basis, multivariate expressions can be introduced—and, accordingly, solutions to partial differential equations. If the data representation is moved from a list into a tree-like structure, it is possible to lazily traverse functions in multiple dimensions, although subject to certain unfortunate constraints regarding order.

The type of duration may likewise be altered. Through the introduction of an explicit representation for infinitesimals, this work transfers from a traditional numeric space into that of smooth infinitesimal analysis— convenient for modeling which includes both continuous and discrete components.

Finally the underlying representation can be packed with a dictionary of `(a -> Double, Double -> b)`. In the course of this paper, we have moved from something only slightly resembling a function (a list), to something which, under the morphism 'at', resembles a function much more closely. With the packed dictionary, splines can be used to represent a more general class of functions; indeed, they can now be made an instance of the standard typeclasses `Applicative` and `Arrow`. As such, this may provide a foundation for one form of efficient functional reactivity.

A. Appendix: `mapAccumL`

From the `Data.List` documentation in the basic libraries...
The 'mapAccumL' function behaves like a combination of 'map' and 'foldl'; it applies a function to each element of a list, passing an accumulating parameter from left to right, and returning a final value of this accumulator together with the new list.

```
mapAccumL :: (acc -> x -> (acc, y)) ->
             acc ->
             [x] ->
             (acc, [y])

mapAccumL _ s []     = (s, [])
mapAccumL f s (x:xs) = (s'',y:ys)
  where (s', y ) = f s x
        (s'',ys) = mapAccumL f s' xs
```

B. Appendix: Code For Polynomials

Adopted from M. Douglas McIlroy (2007)

```
infixr 9 #
instance Num Poly where
   fromInteger c = [fromInteger c]

   negate fs = map negate fs

   (f:ft) + (g:gt) = f+g : ft+gt
   fs + [] = fs
   [] + gs = gs

   (f:ft) * gs@(g:gt) =
       dropZeros $ f*g : ft*gs + [f]*gt
   _ * _ = []

instance Fractional Poly where
   fromRational c = [fromRational c]

   (0:ft) / gs@(0:gt) = ft/gt
   (0:ft) / gs@(g:gt) = 0 : ft/gs
   (f:ft) / gs@(g:gt) = f/g : (ft-[f/g]*gt)/gs
   [] / (0:gt) = []/gt
   [] / (g:gt) = []
   _ / _ = error "improper polynomial division"

-- Polynomial composition
(f:ft) # gs@(0:gt) = f : gt*(ft#gs)
(f:ft) # gs@(g:gt) = [f] + gs*(ft#gs)
[] # _ = []
(f:_) # [] = [f]

-- Polynomial integration
integ fs = dropZeros $
           0 : zipWith (/) fs (countFrom 1)

-- Polynomial differentiation
diff (_:ft) = zipWith (*) ft (countFrom 1)
diff _ = [0]

countFrom n = n : countFrom (n+1)

dropZeros = foldr f [] where
  f elem acc | elem == 0 && null acc = acc
             | otherwise             = elem:acc
```

C. Appendix: Code for Root Finding

Root solving functions, using Newton's method with numeric differentiation.

We divide our root solver implementation into two pieces. First we generate a stream of candidate answers which (hopefully) converge to an actual root [14]:

```
newton :: Double -> (Double -> Double) -> [Double]
newton initial f = iterate go initial where
    go x = x - (approx / df)
      where approx = f x
            df = (f (x + delta) - approx) / delta
            delta = 0.001
```

Next we have a function to pick the first acceptable candidate:

```
pickValue :: Double -> (a -> Double) -> [a] -> a
pickValue tol f =
    -- give up after 1000 tries
    foldr go err . take 1000
  where
    go x x' = if abs (f x) <= tol then x else x'
    err = error "can't converge"
```

[14] For the sake of simplicity, we fix the differentiation step size at 0.001. A serious approach would likely incorporate some form of automatic differentiation.

Finally, a function, using our root solver, that given a fitness function on some value and a method of generating such a value from a `Double`, finds a value suitable to within some tolerance.

```
findValue :: Double ->
             (a -> Double) ->
             (Double -> a) ->
             a
findValue tol p fnc =
    -- we use an initial guess of 1
    pickValue tol p $ map fnc $ newton 1 (p . fnc)
```

Acknowledgments

This paper was typeset with lhs2TeX, developed by Ralf Hinze and Andres Löh. Graphs were produced with Tim Docker's Haskell Charts library.

References

Peter N. Brown, G.D. Byrne, A.C. Hindmarsh. VODE: A Variable-Coefficient ODE Solver. *SIAM Journal on Scientific and Statistical Computing*, 10(5):1038-1051, September 1989.

Conal Elliott. Sequences, segments, and signals, December 2008. `http://conal.net/blog/posts/sequences-segments-and-signals/`

Alan C. Hindmarsh. Oral history interview by Thomas Haigh, 5 and 6 January, 2005, Livermore, California. Society for Industrial and Applied Mathematics, Philadelphia, PA `http://history.siam.org/oralhistories/hindmarsh.htm`

Ralf Hinze. Functional Pearl: Streams and Unique Fixed Points. *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*. Victoria, BC, Canada, September 2008.

Jerzy Karczmarczuk. Lazy Processing and Optimization of Discrete Squences. Technical report, Dept. of Computer Science, University of Caen, France, 2000.

Jerzy Karczmarczuk. Functional Differentiation of Computer Programs. *Journal of HOSC*, 14(1):35-57, 2001

Dusko Pavlovic and Martin H. Escardo. Calculus in coinductive form. *Proceedings of the 13th Annual IEEE Symposium on Logic in Computer Science*. Indiana, USA, June 1998.

M. Douglas McIlroy. The music of streams. *Information Processing Letters*, (77):189-195, 2001.

M. Douglas McIlroy. Basic operations on power series and polynomials represented by lists, practical version, July 2007. `http://www.cs.dartmouth.edu/~doug/powser.html`

Cleve Moler. Stiff Differential Equations. *MATLAB News & Notes*, May 2003.

J.J.M.M. Rutten. Fundamental study—Behavioural differential equations: a coinductive calculus of streams, automata, and power series. *Theoretical Computer Science*, (308):1-53, 2003.