Automatic Differentiation with Scala

By John Mount 6-15-2020, permanent url:

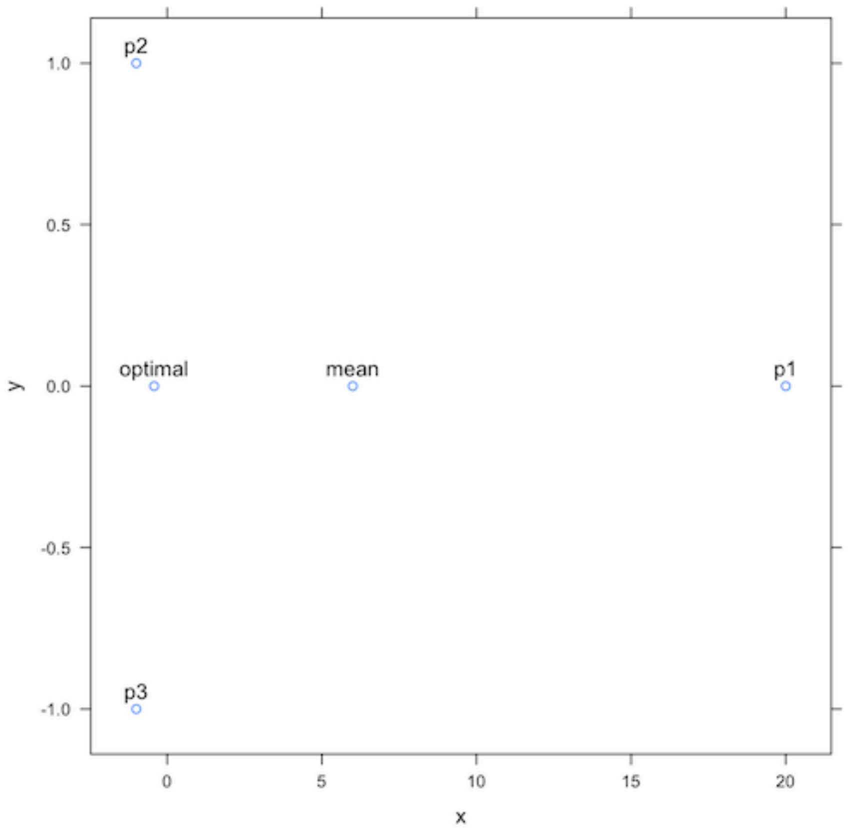
http://www.win-vector.com/blog/2010/06/automatic-differentiation-with-scala/

This article is a worked-out exercise in applying the Scala type system to solve a small scale optimization problem. For this article we supply complete Scala source code (under a GPLv3 license) and some design discussion.

Usually we work using a combination of databases, Java, optimization libraries and analysis suites (like R). The reason is that, for our typical problems, Java hits a sweet spot of trading off runtime performance against ease of development and maintenance. In the tens of gigabytes range (data sets larger than the Wikipedia but smaller than the Web) Java outperforms the scripting languages (Ruby, Python ...) and is much easer to develop in and document than C++. This sweet spot is both subjective and situational- if the tasks were smaller and in a services framework Python is a better choice, if performance is paramount then C or C++ (with the STL) and Hadoop are a better choice, if pre-built statistical libraries are needed then R becomes a better choice. For the type problem we present here Scala is a very good choice.

Our Example Problem

Our small scale problem is this: we have a number of target points on a map and we want to pick a central point to *directly* connect to all of these points with wire. Our goal is to minimize the total amount of wire used. This problem is called the "Geometric Median". So we are trying to find a point that minimizes the sum of distances from our chosen center to every target point. If we were trying to minimize the sum of squared distances from our chosen center to every target point the answer would be obvious: the average or mean (which by Hooke's law is also the point where a set of identical springs would relax to). The mean is in fact a fairly good guess, but you can do better (which could important if the "wire" is expensive, such as cutting irrigation or drainage ditches). For example given the three target points (20,0), (-1,-1) and (-1,1) the optimal point is (-0.42,0) not the mean (6,0) and the choice of optimal point represents an over 19% savings in total wiring distance (see figure).



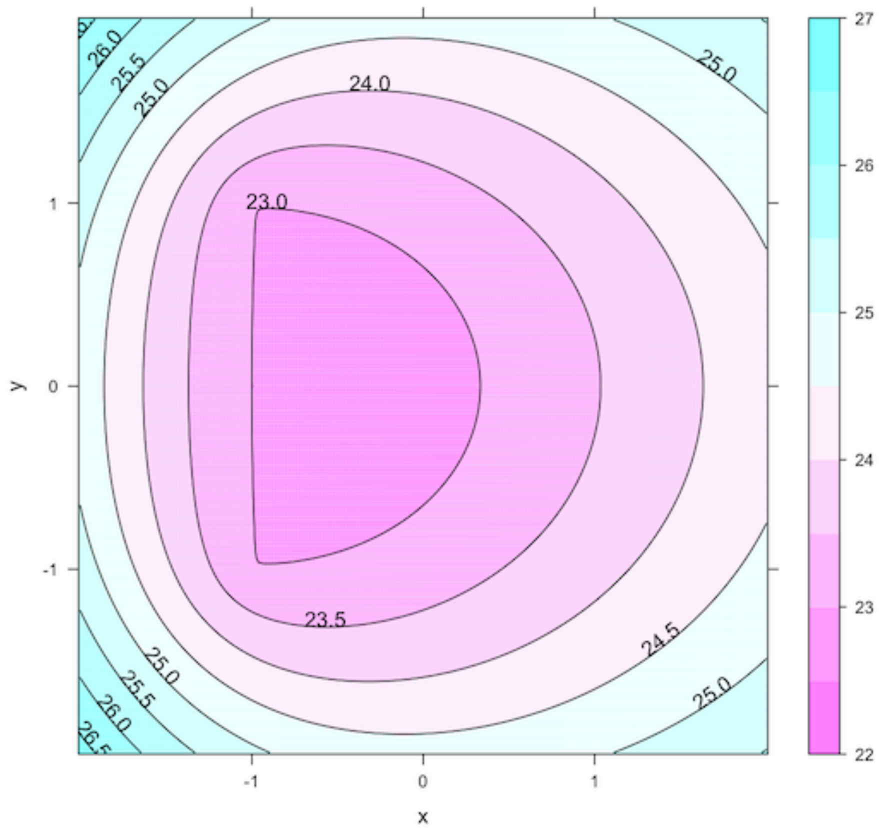
This is a substantial saving in cost.

The problem changes as we consider variations. If indirect connections (such as routing one point through another, which may or may not be possible for reasons of capacity or safety) and multiple new centers are allowed we then have an instance of the Steiner Tree Problem which is harder to solve (since it is known to be NP complete). If no new centers are allowed (all routing must be between pre-existing target points) then we have a Spanning Tree Problem- which admits very quick solutions.

We bring up the geometric median as a mere example. We don't intend for our code to solve only the geometric median problem and we don't intend to touch on the literature of specialized methods for solving the geometric median problem. Instead we are trying to demonstrate the speed you can develop prototype solutions if you have a few good tools (like various optimizers) available in your toolkit. Numeric optimizers may sound exotic, but they often are the kind of thing you want to experiment with and link directly into your code.

Optimization as General Tool

Now that we have the example problem we can describe a solution strategy. In this case the solution uses code "we wished we had lying around" before we started on the problem. We will pretend we have the tools we want ready to solve our problem and then we will pay our debt and build the required tools. The issue is that there is not an obvious closed form for the solution of the geometric median problem. So we are forced to work a bit harder. In this case harder means we need to solve an optimization problem. Consider the contour plot of the total wiring cost as function of where we choose to place our center. Our optimal point (-0.42,0) had wiring cost of 22.73 and the contour plot given here shows concentric regions of solution positions with higher cost.



In general it is unwise to throw an optimizer at an arbitrary problem and hope to find the globally best solution. But in this case (and in many similar situations) we can prove that a simple local optimizer will in fact find the unique best solution. This is a property of the problem not of the optimizer. The concentric regions shown in the contour plot have a very nice shape: they are convex. That is: they have no intrusions- for any two points drawn from one of these shapes the straight line segment between these points stays inside the given shape. We don't have to depend on observation- we can actually prove this is always the case for this problem. The wiring cost from a proposed center to any single target point is a convex function of where we choose to place our center (a convex function is a function whose graph never reaches above the secant line drawn between any two points on its graph). The total wiring cost is just the sum of the wiring costs to each target point. And to finish: the sum of a collection of convex functions is itself a convex function. Since the contour plot of a convex function has only convex shapes and we have proven the statement.

But how does this help us? There is a standard technique to find "local minima" of a function by inspecting a function for places where the gradient is zero (points where there is no obvious down hill direction on the contour plot). This technique usually can only be guaranteed to find local minima (places where no small change improves your situation). But there is no guarantee that the local minimum you find is in fact the global minimum (the best possible solution). Except when you are dealing with a convex function. When a function is convex then all of the local minima are always grouped together into a single convex connected shape (if not a line drawn between two remote minima would violate the convexity definition). And if the function is never flat then this set is a single unique point: the unique best solution. Our inspection technique will be a gradient driven optimizer- that is an optimizer that when the gradient is non-zero improves its objective by running down hill and halts when the gradient is zero.

The stated function to minimize is to sum the distance from our proposed center to each target point. We can write this as the sum of the distances:

$$f(x) = \sum_p ||x - p||_2$$

($||z||_2$ denotes $\sqrt{\sum_i z_i^2}$ which is the traditional Euclidean or L2 distance). This function actually has one one subtle flaw that we will deal with in the appendix (see: Fixing Smoothness).

Using Scala to Apply the Optimization Solution

To find our optimal center placement using Scala we first write our cost or objective as a Scala function:

```
val dat:Array[Array[Double]] = Array(
  Array( 20, 0.0),
  Array( -1.0, 1.0),
  Array( -1.0, -1.0)
)



def fx(p:Array[Double]):Double = {
  val dim = p.length
  val npoint = dat.length
  var total = 0.0
  for(k <- 0 to (npoint-1)) {
    var term = 0.0
    for(i <- 0 to (dim-1)) {
      val diff = p(i) - dat(k)(i)
      term = term + diff*diff
    }
    total = total + scala.math.sqrt(term)
  }
  total
}
```

Scala is succinct and it is a great connivence to have a function definition capture data from its environment. What we would like to do is generate an initial guess as the solution (we use the mean as our initial guess) and then call an optimizer (in this case a conjugate gradient optimizer) to do all the work:

```
val p0:Array[Double] = mean(dat)
val (pF,fpF) = CG.minimize(fx,p0)
```

At this point we would be done, except the conjugate gradient method (which is superior to gradient descent and many the non-gradient methods) requires a gradient. We could provide a numeric estimate of the gradient by the following divided difference method:

```
def gradientD(f:Array[Double]=>Double,p:Array[Double]):Array[Double] = {
  val xdim = p.length
  val p2 = copy(p)
  val base = f(p2)
  val ret = new Array[Double](xdim)
  val delta = 1.0e-6
  for(i <- 0 to (xdim-1)) {
    p2(i) = p(i) + delta
    val fplus = f(p2)
    p2(i) = p(i)
    val diff = (fplus-base)/delta
    ret(i) = diff
  }
  ret
  }
```

This numeric divided difference method often outperforms non-derivative optimization methods (like Powell's Method and the Nelder-Mead Amoeba method). But the technique can run into numeric difficulties. We can remedy this if we are willing to write our function in a slightly more general way. If we re-encode our function in a generic manner we can use automatic differentiation (not to be confused with numeric differentiation or with symbolic differentiation) to produce a reliable gradient for optimization. What we need to do is re-write our function to work over an abstract field of numbers instead of only the machine supplied doubles. In fact what we need to do is specify a generic function that will work over any field, with the field to be determined later. The code to do this in Scala is very similar to the non-generic code:

```
   val genericFx = new VectorFN {
      def apply[Y <: NumberBase[Y]](p:Array[Y]):Y = {
        val field = p(0).field
        val dim = p.length
        val npoint = dat.length
        var total = field.zero
        for(k <- 0 to (npoint-1)) {
          var term = field.zero
          for(i <- 0 to (dim-1)) {
            val diff = p(i) - field.inject(dat(k)(i))
            term = term + diff*diff
          }
          total = total + smoothSQRT(term)
        }
        total
      }
    }
```

Notice that code is very similar to the "def fx()" code. The key differences are that we had to define genericFx as extending a trait (a type of Scala interface) called VectorFN and inside this trait extension we defined a parameterized function name apply(). apply() is a generic function that is willing to work over any type Y where Y is at least of type NumberBase[Y] (we will get more into what that means in a moment). The difference in notation is that while the Scala function *syntax* can not specify a generic function with free type parameters (the incompletely specified Y) the Scala *semantics* are strong enough to implement this. In fact standard function definitions (such as "def fx()") are just syntactic sugar for extending the Scala built-in Function1 trait. With a generic objective function in hand all we need is conjugate gradient code that is expecting a VectorFN (and willing to call apply() instead of just using naked function parenthesis) and some type NumberBase[Y] that can compute gradients for us. The Scala compiler can specialize our genericFx() into one version for quick calculation and another for gradients. How this is done is what we will discuss next. From our point of view our problem is solved with the following one line of code:

```
val (pF,fpF) = CG.minimize(genericFx,p0)
```

This should always be your goal- build sufficient preparation so your last step is a "obvious one liner."

What Tools we Wish we Had Lying Around

We supply in our example some workable conjugate gradient code, but that is standard so we will not discuss it. What is of interest (and facilitated by Scala's parametrized type system) is the implementation of dual numbers as a framework to supply automatic differentiation. An implementation of dual numbers as a NumerBase[DualNumber] type is the core of our demonstration.

Dual numbers are an algebraic structure written as pairs of real numbers "(a,b)". The arithmetic table for dual numbers is given below:

(a,b) + (c,d)	=	((a+c) , (b+d))
(a,b) - (c,d)	=	((a-c) , (b-d))
(a,b) * (c,d)	=	((a*c) , (a*d+b*c))
(a,b) / (c,d)	=	((a/c) , ((b*c-a*d)/(a*a)))

In a dual number (a,b) "a" is the "large" or "standard" part of the number. You can check from the arithmetic table that the pair of dual numbers (a,0) and (c,0) behave just as we would expect the real numbers a and c to behave. In the dual number (a,b) "b" is the "small" or "ideal" portion of the number.

From the multiplication rule above we can observe two rules: (0,b) * (c,0) = (0,b*c) (something small times anything else is small) and (0,b)*(0,d) = (0,0) (two small things become zero when multiplied). Essentially the dual numbers are carrying around the first two terms of a Taylor series, so if we evaluate a function over the dual numbers (instead of over the reals) we get as a result both the function value and the function derivative. We can check that these numbers obey the usual laws of arithmetic (associative, commutative, distributive, identities and inverses). The punchline is that over the dual numbers the divided difference estimate of f'(x) (the derivative of f() evaluated at x) is in fact exact in the sense that f((x,1)) = (f(x),f'(x)) (or f((x,0)+(0,1)) - f((x,0)) = (0, f'(x))). Implementing the DualNumber class is little more than transcribing the above arithmetic table into Scala.

We have already seen how to write code that uses NumberBase[Y] types (genericFx() itself is an example). A more complicated example is the CG.minimize() code which not only accepts a generic function (in the form of VectorFN) but then specializes it to NumberBase[DualNumber] to compute gradients and also specializes to NumberBase[MDouble] for quick calculation during line searches (MDouble is just an adapter for machine Doubles, used for speed). The ability to re-specialize a function is one of the advantages of a parameterized type system. The DualNumbers are an example of forward automatic differentiation. We could also use the same object framework to capture a representation of the computation path and apply more sophisticated methods such as reverse automatic differentiation.

We give a link to a jar containing complete Scala source code including this example, the DualNumber implementation, a conjugate gradient implementation and some JUnit tests (all under a GPLv3 license) and will go on to describe some of the design decisions. The code is the bulky part of this work, so we will move on to discuss something more compact: types.

Types

If code is ever beautiful it is only when it is succinct. Among the most succinct forms of code are individual type signatures and interfaces (though the indiscriminate repetition of type signatures is rightly considered ugly bloat, which Scala works to avoid). Since we are distributing complete source we will describe only types and method signatures. The entry points to the code are the JUnit tests (organized in the ScalaDiff/test source directory and depending on JUnit which was not included) and the demo program in ScalaDiff/src/demo/Demo.scala).

To be a usable arithmetic type (like DualNumber or MDouble) you must extend the following parameterized abstract class:

```scala
abstract class NumberBase[NUMBERTYPE <: NumberBase[NUMBERTYPE]] {
  // basic arithmetic
  def + (that: NUMBERTYPE):NUMBERTYPE
  def - (that: NUMBERTYPE):NUMBERTYPE
  def unary_-():NUMBERTYPE
  def * (that: NUMBERTYPE):NUMBERTYPE
  def / (that: NUMBERTYPE):NUMBERTYPE  // that not equal to zero
  // more complicated
  def pow(that:Double):NUMBERTYPE
  def exp:NUMBERTYPE
  def log:NUMBERTYPE // this is positive
  // comparison functions
  def > (that: NUMBERTYPE):Boolean
  def >= (that: NUMBERTYPE):Boolean
  def == (that: NUMBERTYPE):Boolean
  def != (that: NUMBERTYPE):Boolean
  def < (that: NUMBERTYPE):Boolean
  def <= (that: NUMBERTYPE):Boolean
  // utility
  def field:Field[NUMBERTYPE]
}
```

In particular DualNumber extends NumberBase[DualNumber]. This deliberate circular reference has a big purpose: it allows publicly visible contravariant return types (returning nearly the exact type we really are instead of a base type). This allows us to have strict type arguments so that trying to add a MDouble to DualNumber is a type error (even though they both extend the same base class). The automatic differentiation technique encapsulated in the DualNumber class only works if all of the calculation is in the DualNumber types and this strict type enforcement allows the compiler to help prevent results sneaking in and out through other types. All of the methods on NumberBase are obviously related to arithmetic except the field() method. This method gives us access to a Field object which is responsible for carrying around the runtime type information (this is a common problem in Java and Scala, that some type information known at compile type such choice of template types is not easily accessed at runtime). The Field class is as follows:

```
abstract class Field [NUMBERTYPE <: NumberBase[NUMBERTYPE]] {
  def zero:NUMBERTYPE             // return canonical zero in field
  def one:NUMBERTYPE              // return canonical one in field
  def inject(v:Double):NUMBERTYPE  // return canonical representation of number in field
  def project(v:NUMBERTYPE):Double // return standard-number represented in field
  def array(n:Int):Array[NUMBERTYPE] // return an array of this type
```

The Field class is where we have factories for numbers (zero, one, arrays, injection from standard Doubles), casting (projection back to standard Doubles).

With these types defined we can actually read intent off some of the method signatures.

For example our conjugate gradient optimizer is accessed through the following method signature:

```
 def minimize(fn:VectorFN,x0:Array[Double]):(Array[Double],Double) // return x,f(x)
```

The above can be read as: CG.minimize() requires a VectorFN (our trait representing single argument functions with a free type parameter) and an initial point (in standard Doubles). The code will the return a pair of the optimum point and the function evaluated at the optimum point. From the type signature we can see that CG.minimize() expects to re-specialize the function "fn" to types of its own choosing (else it could have accepted a parameterized argument instead of our custom trait) and will handle all up-conversion and down-conversion between machine Doubles and NumberBase[Y]'s itself. This sort of type information is hard to express (let alone enforce) in a dynamically typed language.

A slightly more complicated example is the lineMinD() method:

```
def lineMinD[Y<:NumberBase[Y]](field:Field[Y],
 f:Array[Y]=>Y,
 xm:Array[Double],
 di:Array[Double]):(Array[Double],Double)
```

Notice it is willing to work with any type parameterized function (which means it is willing to let the caller pick the actual type of NumberBase[Y] and work with that). Most callers will call with Y=MDouble (the wrapper for machine Doubles) and lineMin() will then work with that (without ever really knowing the actual underlying type).

A lot of fans of dynamic languages consider type systems to be mere hairshirt penance. But that is not so. Broken type systems (like Java's collections before erasure parameters were introduced in Java 1.5) are indeed more trouble than they are worth. Working type systems (like C++ Templates/STL, Java 1.5+ and Scala) allow you to solve problems (and enforce decisions) during the design phase (which is much much cheaper than during the deployment phase). You can't set your types in stone (you are likely going to have them subtly wrong for the first few iteration). You must be willing to think like a "language lawyer" to find out what parts of your work can be specified and enforced in the language type system. To use an analogy: static types are your blueprint or your underpainting.

Tests

One argument against static types is that you can get much of their benefit from unit tests. My opinion is you never have enough unit tests, so putting more pressure on your test suite is not wise. Static types plus tests are strictly more powerful than static types alone or tests alone.

Even for this example toy-scale project we have include a JUnit test set to pursue a number of goals:

- Confirm our number implementations (DualNumber and MDouble) correctly model machine Doubles (perform parallel calculations and compare).
- Confirm DualNumber obeys expected laws of algebra composition and cancellation *including the portions that can not be modeled in machine Doubles*.
- Confirm DualNumbers compute gradients.
- Confirm operations of optimizers and optimizer components.

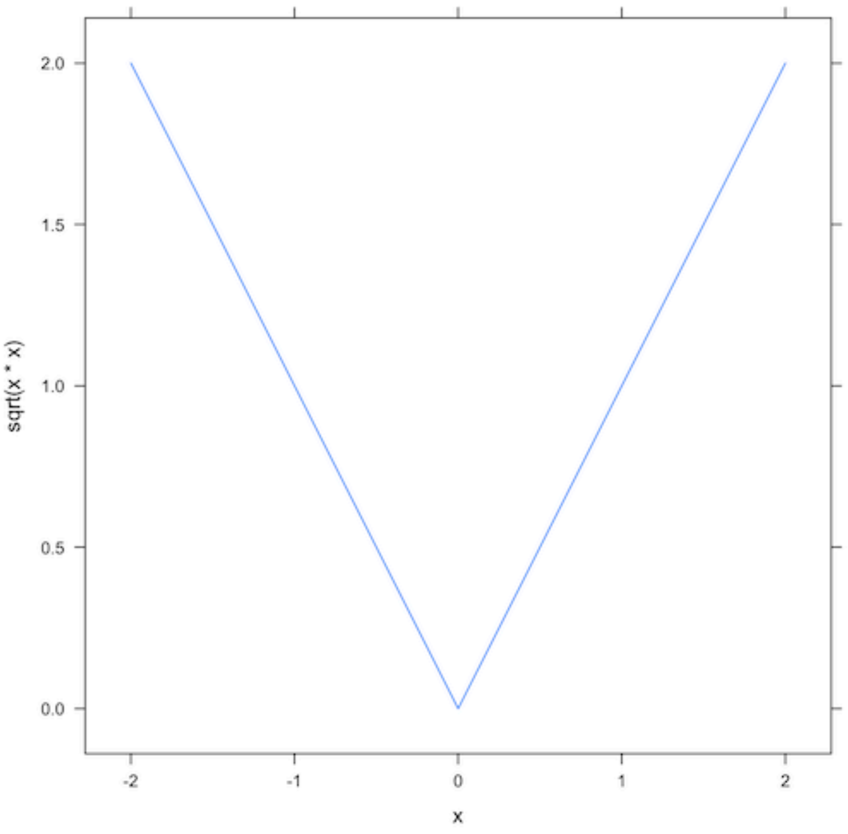
Many of these tests are related, but they don't all imply each other and give different perspective on the errors they catch. For example no amount of parallel computation between DualNumbers and machine Doubles is going to confirm the infinitesimal portion of the DualNumber is propagating correctly (since this is not a property of machine Doubles). So we add extra tests that expect DualNumber to obey algebraic relations like: a*(b+c) = a*b + a*c hold. It is then another step to confirm that whatever the DualNumbers calculate is not only self-consistent, but also models a truncated Taylor Series or differentiation.

Conclusion

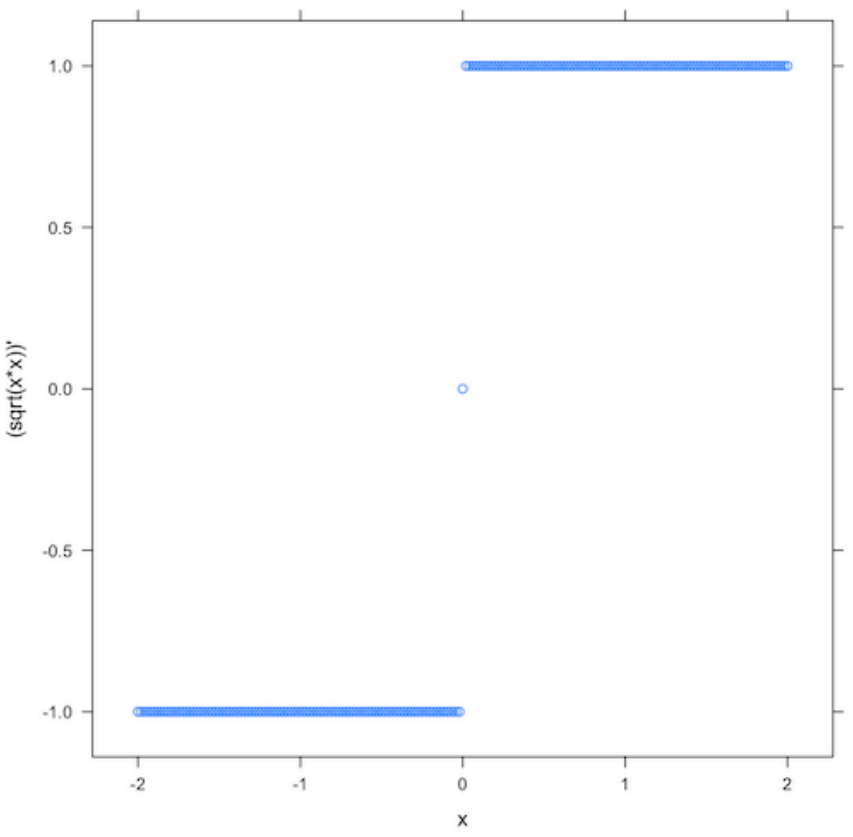
We hope we have demonstrated how the complexity of a mathematical programming problem can be managed by breaking the problem into an objective function that is separate from the optimizer (allowing the optimizer to be both good and hidden) and a static type system (such as Scala) to help enforce required properties of a calculation (such as all numbers being routed though a required representation). With these sort of tools available many formerly hard problems (that are often, unfortunately solved by over-specifying direct inefficient iterative improvement techniques) become "if I can write a reasonable objective function this may already by solved by an optimizer in my library." The more of these tools you have (either in your code or in your reference library) the more of these problems become easy (this is the topic of my earlier paper: The Local to Global Principle).

Appendix: Fixing Smoothness

Our chosen example objective function is very nice (i.e. convex) but it has a small (but correctable) problem. The derivative or gradient or gradient has some jump discontinuities that could cause an optimizer to exit prematurely (not at the global optimum). Consider the simple form of this for wiring a center to a single point at the origin (even in 1 dimension). The wiring cost function is sqrt(x*x) has a cost graph as shown here.



This is convex- but derivative is not smooth as we see in the included graph of the derivative of sqrt(x*x).



So: in this case if the optimizer stops at one of the target points we can't be sure that it stopped at the global optimum (it may have stopped due to the discontinuity in the gradient). For some simple problems the optimum is necessarily at a target point. For example on the number line take the target points 0,1 and x. As long as x≥0 and x≤1 the optimum placement will be x itself.

One way to defend against this is to use some sort of smoothed version of sqrt() that essentially decreases a little faster near the origin. Our cost function becomes:

$$f(x) = \sum_p s(||x - p||_2^2)$$

where s() is our suitable approximation of the sqrt() function. Two candidates are s(x) = (x+tau)^(1/2) and s(x) = x^(1/2 + tau); where tau is a small constant. As long as tau is greater than zero we have no derivative discontinuity in s(x^2) and convexity is preserved (even made a bit stricter). Other ways to deal with this include adding additional coordinates to the problem and small perturbations on these coordinates. Finally, a point found by optimizing with respect to s(x) can be "polished" by re-starting the optimization at the first found solution and using sqrt(x) as the new objective (if the original point is not near any of the target points).