JFP **21** (3): 309–329, 2011. © Cambridge University Press 2011
doi:10.1017/S0956796811000086

A combinator library for the design of railway track layouts

BARNEY STRATFORD
(*e-mail:* `barney_stratford@fastmail.fm`)

Abstract

In the design of railway track layouts, there are only a small number of geometric configurations that are used in practice, and a number of constraints as to how those configurations can be fitted together to create a whole layout. In order to solve these problems, we construct a Haskell combinator library. The library has been used for the design of real-world track layouts.

1 Introduction

Railway preservation is a peculiarly British phenomenon. A preserved railway is effectively a working museum, offering the public the chance to see how railways were run in past times. Many of them are operated by steam locomotives that only the older generation can remember in service. Such railways are staffed almost entirely by volunteers, who relish the chance to get away from their day jobs and help to look after big, beautiful, smelly machines.

The Mid-Norfolk Railway is one such organisation. There has developed a pressing need to be able to efficiently design track layouts to fit within various constraints, such as the amount and shape of land available, and the facilities that are required to be included in the design. There is software available to the professional rail industry to solve such problems, but this is unavailable to a non-profit organisation, making the present work necessary. The opportunity to study the underlying geometry, and to work the problems through ourselves seemed too good to miss, and has resulted in a further "real world" use for functional programming. The methods used here are similar to those used in the industry 20 years ago, before computers were powerful enough to display fancy graphical interfaces.

The first use to which this library has been put is the design of a passing loop. On a single-track railway, it is clearly impossible for trains heading in opposite directions to pass each other. To overcome this, a short double-track section is provided (along with the associated signalling systems) where the trains can pass, thus greatly increasing the number of trains that the line can carry at once (see Figure 1).

Our new passing loop was built at Thuxton, with construction work starting in early 2009. Around 1,000 man-days of volunteer labour were used in the building of the loop and signalling system, and the total cost came to 50,000 UK pounds.

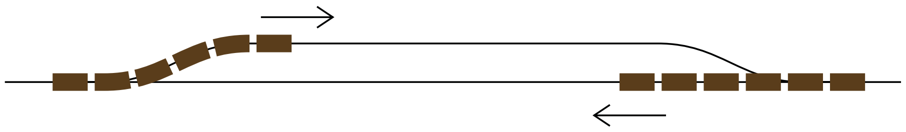

Fig. 1. A simplistic passing loop, enabling two trains to pass on a single-track railway.

Our library is developed using Haskell (Bird 1998; Peyton Jones 2003), whose elegant syntax makes it attractive for our purposes. Haskell has found a wide application for the design of domain-specific languages, and the present paper continues this theme. In particular, the problems that we solve here can be expressed much more elegantly when functions are first-class objects. The paper also provides a general pattern for combinator libraries that solve numerical problems, particularly in Section 3.

Why did we take this approach to solving the problem? We wanted our library to be able to calculate track layouts based on the constraints that they must satisfy. We want to describe what a layout should look like, or what the end result should achieve, rather than have to say precisely how the layout is to be built. This naturally suggests a declarative style of programming, which is the forte of the functional programming languages. Our declarative approach made it possible to use the system for rapid prototyping, as it is easy to take pre-built sections of track layout and simply attach them together to see what the result looks like.

1.1 A little history

The system described here has undergone two or three major design changes and hundreds of smaller tweaks to get it into its current form. We began with a very concrete representation of our various datatypes, and special-purpose code to solve each of the constraints that we encountered. It quickly became clear that this method was not flexible enough for the intended purpose, as we were finding it necessary to solve new constraints all the time, and each new situation required new code. It was quickly becoming an unmaintainable mess.

Instead, we have used a very general datatype to describe all our track layouts without needing to go into any special cases. We also built a powerful constraint solver that enables us to specify arbitrary constraints that our layout is to satisfy. We are not limited to solving only the problems that were considered important by the designers.

The generality and flexibility of our system are its key strengths, for every rule has its exceptions, and every general principle will have specific situations in which it does not apply. Much of the domain-specific knowledge that needs to be applied to a layout design comes from experience, judgment or knowledge of the site in question. Human input into the design process is absolutely essential.

In initial versions of this software, we had hoped to use the type system to provide guarantees of the validity of a track layout. For example, a layout is invalid if two adjacent sections of track do not meet up: a train would derail in the gap. When we implemented these kinds of guarantees, we found that it caused the program's complexity to increase, and it became too difficult to read code that was littered with constructors. The ability to produce invalid track layouts is not a great inconvenience, for a glance at the finished design will show up such glaring errors in an instant. Experience has shown that an attempt to produce an invalid layout will usually result in the machine being unable to find a layout that satisfies the constraints. In any case, it would only have been possible to catch the simplest kinds of errors using such methods, and human inspection would still be required to ensure that the many engineering rules are followed.

2 A railway track primer

We begin by giving the basic definitions of the railway-specific technical terms that are used in this paper. The construction of railway track is a diverse subject, with many different designs, some experimental, having been used by the various railway companies that have existed throughout history. An authoritative reference is Cope (1993).

2.1 Plain line

A section of railway track consists of two steel rails mounted on *sleepers* (known as *ties* in the USA) that hold the rails in place. The track is laid on a bed of small, angular stones (known as *ballast*) that transfer and spread the weight of passing trains to the track bed.

One job of the sleepers is to maintain the *gauge* of the track, which is the distance between the inside surfaces of the rails. In current British practice, the gauge is 1435 mm.

In order to ensure a smooth ride and to reduce wear on the track and trains, it is important that the geometry of the track is maintained. Where the track curves, the outside rail will be higher than the inside (called *cant*) so that passing trains lean into the curve and there is no net horizontal force on the track.

In order to ensure that a train's passage is as smooth as possible, it is highly desirable to keep the cant almost constant. When the cant is unchanging over a section of track, the curvature must also remain constant, and so railway tracks are mostly designed using circular arcs and straight line segments.

The real-world situation is a little more complicated than this, however, as the curvature must necessarily change sometimes, for example at the end of a straight section of track. One can't abruptly change the curvature, for then the cant would also have to change abruptly to match, which isn't allowed. Instead, the curvature changes linearly from one value to another over a short section of track; this is a *transition curve*. In order to keep things simple, we have not included transition curves in this version of the combinator library although there is no reason why they couldn't be added at a later date. For the current application, in which the trains will be moving fairly slowly, this is not a serious shortcoming.

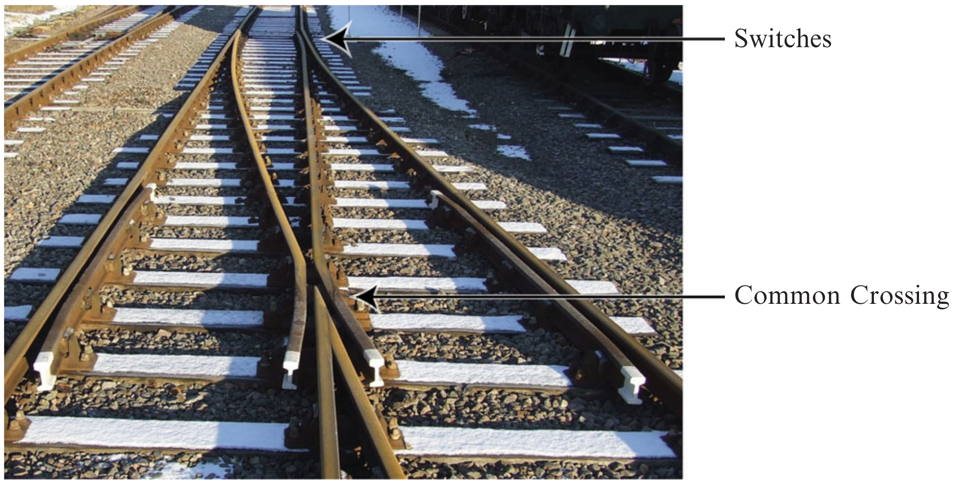

Fig. 2. A turnout.

2.2 Switches and crossings

Two tracks will often be required to converge to a single line, and this is achieved by means of *turnouts*. A turnout consists of two important parts: the *switches* that move to divert a train from one line to another, and a *common crossing* that enables the rails of the two routes to cross each other (see Figure 2).

Switches and crossings come in a variety of standard sizes, and which to be used depends on such factors as the speed of passing trains and the space available. It is possible to manufacture switches and crossings to any required size, but the use of a non-standard size greatly increases the cost and time required. The author only knows of one occasion where this has been necessary (on the London Underground), and layouts are almost exclusively designed with the standard sizes in mind.

3 Manipulating circles and straight lines

The basic elements of a railway track layout are circular arcs and straight line segments. A track layout is formed by connecting several such curves together. This section develops the machinery that we will subsequently use to create and manipulate circles and lines.

Traditionally, circles are specified by giving their centre and radius. When designing track layouts, we do not always know this information. Instead, we may have to specify that an unknown circle is tangent to some given circle, or that it passes through a certain point. We will define functions that allow us to describe circles in terms of the conditions that they must satisfy.

In order to manipulate circles and lines in an effective manner, it is necessary to make a careful choice of representation. Early versions of our combinator library considered circles and straight lines as separate cases – a very concrete representation.

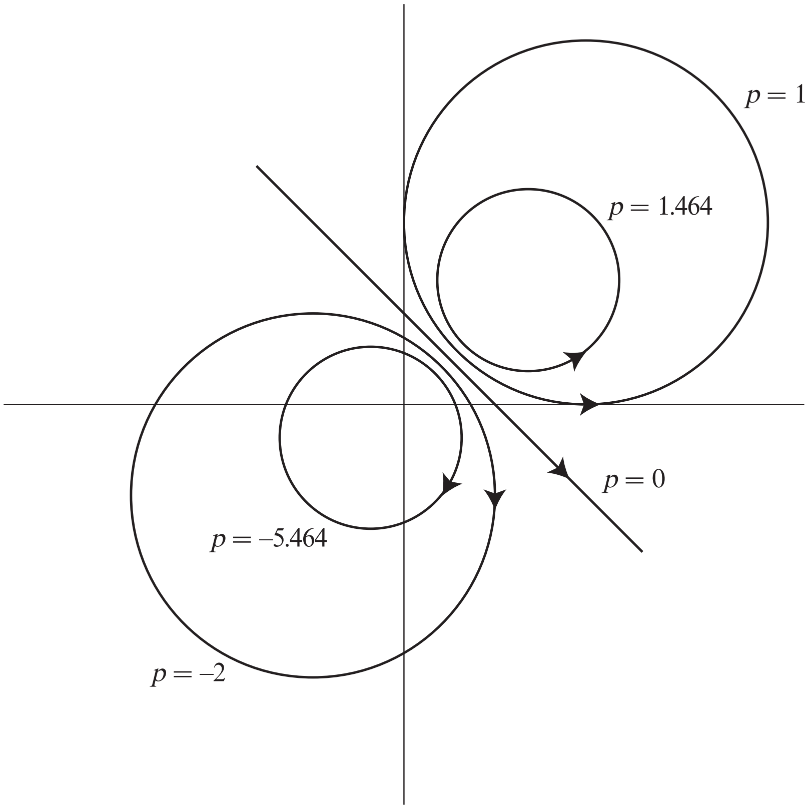

Fig. 3. Oriented circles in $\mathbb{R}^2$. Values of p vary and $a = b = k = 1$.

This led to a number of corner cases and increased the amount of calculation that was necessary. Instead, we use a slightly more general description of circles due to Pfeiffer and van Hook (1993) such that straight lines are merely a special case that does not need separate consideration. Simple continuity arguments can then be applied to prove – for free – that the library behaves correctly when faced with a straight line, without further calculation being necessary. Another useful consequence of the representation is that we only rarely need to use any trigonometric functions.

Hiding these details inside a combinator library means that the end user will not have to think about them, or even be aware that they exist.

3.1 Basics

We begin by considering how to represent the straight line segments and circular arcs that our layouts will be constructed from. In all that follows, we will use the word 'circle' to include straight lines, which can be considered to have infinite radius. Where we wish to exclude straight lines, we will refer to 'proper circles'. All our circles will be oriented.

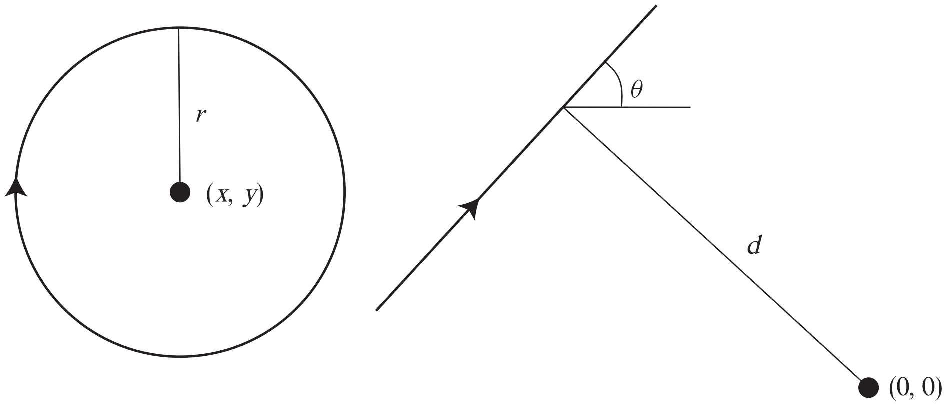

Fig. 4. Specifying oriented circles and straight lines.

Let $(p, a, b, k) \in \mathbb{R}^4$ be a non-zero vector. Then, the locus $C\ (p, a, b, k)$ of points $(x, y) \in \mathbb{R}^2$ satisfying

$$p(x^2 + y^2) - 2ax - 2by + k = 0$$

is a circle. In the case where $p = 0$, then it is readily seen that this equation represents a straight line. When $p \neq 0$, then the equation can be rearranged to give

$$\left(x - \frac{a}{p}\right)^2 + \left(y - \frac{b}{p}\right)^2 = \left(\frac{a}{p}\right)^2 + \left(\frac{b}{p}\right)^2 - \frac{k}{p}.$$

This is the equation of a proper circle. When $C\ (p, a, b, k)$ is a proper circle with its centre at (α, β) and of radius r, then

$$(\alpha, \beta) = \left(\frac{a}{p}, \frac{b}{p}\right)$$

and

$$r^2 = \left(\frac{a}{p}\right)^2 + \left(\frac{b}{p}\right)^2 - \frac{k}{p}.$$

See Figure 3 for some examples.

Note that $C\ (p, a, b, k) = C\ (-p, -a, -b, -k)$. We can use this fact to allow us to specify the orientation of our circles. When $p > 0$, then we consider the arrow to be pointing anticlockwise, and when $p < 0$, then it is clockwise. When $p = 0$, then we have a straight line, and we take the orientation to point in a direction parallel to the vector $(b, -a)$.

We introduce a datatype to hold this information.

```
> data Circle a = Circle {p, a, b, k :: a}
```

We also introduce some simple combinators to build circles. Proper circles are specified by giving the centre and radius, while lines are given by the angle from horizontal and the minimum distance to the origin as in Figure 4.

```
> anticlockwise_circle :: Double -> Double -> Double -> Circle Double
> anticlockwise_circle x y r =
>   Circle 1 x y (x ^ 2 + y ^ 2 - r ^ 2)

> clockwise_circle :: Double -> Double -> Double -> Circle Double
> clockwise_circle x y r =
>   Circle (-1) (-x) (-y) (r ^ 2 - x ^ 2 - y ^ 2)

> line :: Double -> Double -> Circle Double
> line d theta = Circle 0 (-sin theta) (cos theta) (2 * d)
```

Finally, we make our `Circle` type into an instance of `Read` and `Show`.

3.2 The Newton–Raphson iteration

One of the main aims of this combinator library is to be able to express a circle in terms of the properties that it satisfies. For example, we might have surveyed a site (possibly with a theodolite or GPS receiver) and found that an unknown circle passes through a known point (x, y) and is tangent to two known circles c_1 and c_2. Because there may be more than one solution satisfying these constraints, we have to estimate roughly where the solution lies. In Haskell code, we would then wish to say something like:

```
> c = find estimate $ satisfying
>   [passing_through x y,
>   tangent_to c1,
>   tangent_to c2]
```

and expect c to be a circle satisfying these constraints.

Each of the constraints can be translated into a function on circles whose value goes to 0 when the constraint is satisfied. This enables us to turn a difficult geometric problem into a less-difficult numerical one (zeroing several functions simultaneously), which can be solved by using a multi-dimensional Newton–Raphson iteration. Although seen only infrequently in undergraduate mathematics courses, this is a natural generalisation of the one-dimensional case that finds the zeros of a single function, and the unfamiliar reader is referred to (Press *et al.* 2007).

Why are we going to all the effort of setting up this numerical algorithm, when it is perfectly possible to produce an exact formula for each of the problems we might wish to solve? In early versions of the library, this was precisely the approach that we took, producing reams of code, for example to solve the problem just given. We found that such exact formulæ became big, ugly and unwieldy rather quickly. Each time a new problem arose, it became necessary to go back to the drawing board and find yet another new formula to solve it. We were spending a considerable amount of time on this and found the whole process somewhat dispiriting.

Using the combinator-based approach, complex problems can be built up from simpler constituent parts, using only a few primitives. This is what functional programming is all about.

When we are using a standard mathematical library to calculate trigonometric functions, square roots and even floating-point division, it is easy to forget that these algorithms invariably depend on some kind of iterative process under the bonnet, and that their ‘exact’ calculations are restricted by the machine’s accuracy limits. The results given by the ‘approximate’ methods used throughout this paper are therefore no less valid than the results we would obtain by performing a direct calculation.

3.2.1 Circles: three dimensions or four?

Note that a circle in the plane is specified by giving three coordinates: its centre and radius. Our representation of circles has *four* parameters, and one of these must therefore be redundant. We could perform the Newton–Raphson iteration in four dimensions, finding values of p, a, b and k for which all the constraints go to zero, but this would ignore the redundancy in our representation of circles. Failing to use all the available information would adversely affect the convergence. Instead, we will show how we can reduce the problem to three dimensions before performing the iteration.

There are many ways to perform this reduction of dimension. The most important criterion when selecting one is that it must be surjective (up to multiplication by a positive constant). We would also like it to be numerically well-conditioned and to be easy to compute.

Suppose that we initially make a guess that $C\ (p,a,b,k)$ is a circle that is close to the solution to whatever problem we are trying to solve. We will specify an arbitrary circle $C\ (p',a',b',k')$ by making k' into a function of the other three variables, thereby removing a dimension.

When $k < 0$, then we define

$$k' - k = (p'^2 - p^2) + (a'^2 - a^2) + (b'^2 - b^2).$$

This defines a paraboloid. Because $k < 0$, the origin lies inside the paraboloid and so any straight line that begins at the origin will intersect this paraboloid exactly once.

For positive values of k, we need to be more careful, since sufficiently large values of k will cause this paraboloid to move so that the origin is no longer inside it. We can work around this by negating the left-hand side whenever $k \geqslant 0$:

$$-(k' - k) = (p'^2 - p^2) + (a'^2 - a^2) + (b'^2 - b^2).$$

This gives us our projection function, for suitable types `a` and `b`. As a minor optimisation, we can use the identity $x^2 - y^2 = (x - y)(x + y)$ to save operations.

```
> circle :: Circle a -> [b] -> Circle b
> circle (Circle p a b k) [p', a', b'] =
>   Circle p' a' b' k'
>   where
>   x = (p' - p) * (p' + p) +
>      (a' - a) * (a' + a) + (b' - b) * (b' + b)
>   k'
>      | k <  0 = k + x
>      | k >= 0 = k - x
```

By construction, our initial guess $C\ (p, a, b, k)$ is already a point on our paraboloid.

3.3 Handling derivatives

A constraint in our system is expressed as a function taking a circle to a real number. The real number goes to zero precisely when the constraint is satisfied. A set of constraints are all satisfied exactly when all of the functions go to zero simultaneously. In situations where we wish to find the simultaneous zeros of several functions of several variables, there is really only one method available: Newton–Raphson.

The problem with using this method is that we need to know not only the values of the functions, but also all of their partial derivatives. A function on circles takes three values, which we are denoting by p, a and b. We calculate k from these values so we can ensure we have only a three-dimensional problem. We give a Haskell type that can contain not only the value of a function at a point, but also its partial derivatives (Karczmarczuk 1998).

```
> data Result = R {value, dp, da, db :: Double} deriving (Read, Show)
```

A set of constraints then has a Haskell type isomorphic to `Circle a -> [Result]` for a suitable type `a`.

Since `Result` is really just `Double` with some extra information added on, we can make it an instance of `Eq` and `Ord`.

```
> instance Eq Result
>   where
>   x == y = value x == value y

> instance Ord Result
>   where
>   compare x y = compare (value x) (value y)
```

Given a pair of `Result`s, we can perform all the basic arithmetic operations using the standard rules of differentiation. For example,

```
> x + y = R (value x + value y) (dp x + dp y)
>   (da x + da y) (db x + db y)
```

This quickly becomes tedious, however, for we have to give almost identical expressions for `dp`, `da` and `db`. To ease the pain, we give a helper function that enables us to write the expression for each derivative only once.

```
> result :: Double -> ((Result -> Double) -> Double) -> Result
> result value derivative =
>   R value (derivative dp) (derivative da) (derivative db)
```

Using this helper function, we can write

```
> x * y = result (value x * value y)
>   (\d -> value x * d y + value y * d x)
```

or even

```
> sin x = result (sin (value x)) (\d -> d x * cos (value x))
```

In fact, `Result` is an instance of `Floating`, giving access to the full range of mathematical operations while keeping track of all the partial derivatives with respect to *p*, *a* and *b*.

3.3.1 Coercions

Any `Double` value can be coerced to a `Result` type by assuming that the value is a constant. Likewise, a constant `Circle Double` can be coerced to a `Circle Result`.

```
> coerce_constant :: Double -> Result
> coerce_constant x = R x 0 0 0

> coerce_circle :: Circle Double -> Circle Result
> coerce_circle (Circle p a b k) =
>   Circle (R p 0 0 0) (R a 0 0 0) (R b 0 0 0) (R k 0 0 0)
```

To perform these coercions at appropriate times, we create a typeclass. For convenience, and to avoid clutter later on, our class is a subclass of `Floating` and `Ord`.

```
> class (Floating a, Ord a) => CircleType a
>   where
>   coerce_constant :: Double -> a
>   coerce_circle :: Circle Double -> Circle a
```

Both `Double` and `Result` are instances of `CircleType`, with the obvious definitions. This typeclass will be used extensively when we come to dealing with circle transformers in Section 4.1.

3.4 Satisfying constraints

We have developed all of the machinery that we will use when finding a circle that satisfies some given constraints, so we show how to assemble these parts into a functioning whole.

We have already seen that a constraint has a type isomorphic to `Circle a -> [Result]` for suitable a. Let `c :: Circle a -> [Result]` be a constraint and let `f :: Circle a -> Circle a` be some function on circles. Then `c . f` is a new constraint, and there are plenty of situations in which we may want to construct such a constraint. For example, at a common crossing, the right-hand rail of one route may cross over the left-hand rail of another at a fixed angle. When a section of track is represented by giving its centre line, then we will have to apply a transformation to both circles and state that the *transformed* circles cross over at the given angle.

Suppose that a is `Double`. A problem arises because all of the derivatives in `c . f` may be incorrect, and so the Newton–Raphson iteration may fail to converge on the correct value. We have transformed a circle using `f` without applying the Chain Rule for differentiation.

In fact, the right thing to do is to take `a` to be `Result`, for then we can apply whatever transformations we choose and the derivatives will be calculated correctly.

Observe that circles of type `Circle Double` are known, constant circles, whereas circles of type `Circle Result` are always unknowns that contain the current best estimate. We can therefore make some highly suggestive type synonym declarations.

```
> type Known = Double
> type Unknown = Result
```

Although a constraint function returns a list of `Results`, these lists are really rather atomic. It doesn't make much sense to count their elements, or to split them up. Their ordering is unimportant and it makes no difference if an element appears more than once. All we want to be able to do is to combine lists of constraint functions together.

```
> newtype Results = Rs {unRs :: [Result]}

> satisfying :: [Circle Unknown -> Results] ->
>   Circle Unknown -> Results
> satisfying constraints = concatRs . getRs
>   where
>   concatRs = Rs . concat . map unRs
>   getRs = flip map constraints . flip ($)
```

Each step in the Newton–Raphson iteration involves calculating the value and derivatives of all our constraint functions and then applying the inverse of the Jacobian to the column vector of values. To keep the technicalities to a minimum, we won't go into the details of how this works. A function to perform a typical Newton–Raphson iteration would then be defined as

```
> find :: Circle Known -> (Circle Unknown -> Results) -> Circle Known
> find c@(Circle p a b k) constraints =
>   circle c $ newton_raphson [p, a, b]
>   (unRs . constraints . circle c)
```

where `newton_raphson` has type

```
> newton_raphson :: [Double] -> ([Result] -> [Result]) -> [Double]
```

and where `newton_raphson first_guess constraints` is a list of values that satisfy the constraints. This function has been designed so that it can cope with many of the mishaps that occur in practice. It is not possible to guarantee that `newton_raphson` can always discover a solution that satisfies the constraints, however, as they might be contradictory or the initial guess might be too far from a solution. It will always terminate, and it will signal an error if the resulting circle is not close enough to a position where the constraints are satisfied.

In general, it really is necessary to provide an initial estimate for the result, as there are often multiple solutions to the problems we wish to solve. Experience has shown that there are typically two solutions, and we have to distinguish between them.

4 Circle combinators

Everything is now in place to enable us to find circles that satisfy given constraints. We only need the combinators that transform circles and evaluate the constraints.

4.1 Circle transformers

4.1.1 Reversing orientation

Railway track layouts are not, in general, orientable. For example, a triangular junction would enable a train to turn round by doing a three-point turn. As such, a track layout is only *locally* orientable: we can paint arrows on the tracks, but for some layouts it may be inescapable that two tracks converge with their arrows pointing in opposite directions. To construct layouts where this happens, it will be necessary to be able to reverse the orientation of our circles. We have defined circles so this can be achieved by simply negating p, a, b and k.

```
> reverse_circle :: CircleType a => Circle a -> Circle a
> reverse_circle (Circle p a b k) = Circle (-p) (-a) (-b) (-k)
```

4.1.2 Offsetting

It is often the case that two railway tracks follow each other side-by-side. We provide a function to offset a given circle by a given amount to the right when facing in the direction of the arrow. The new circle is concentric with the old one, or, in the case of two straight lines, the two are parallel.

Suppose we wish to offset our circle $C\ (p, a, b, k)$ by an amount δ, giving the circle $C\ (p, a, b, k')$. When the circle is oriented anticlockwise, then we will add δ to its radius r, whereas δ will be subtracted from the radius of a clockwise circle. Hence,

$$\frac{k}{p} = \left(\frac{a}{p}\right)^2 + \left(\frac{b}{p}\right)^2 - r^2$$

and

$$\frac{k'}{p} = \left(\frac{a}{p}\right)^2 + \left(\frac{b}{p}\right)^2 - (r \pm \delta)^2.$$

Expanding the second expression and substituting the first twice yields

$$\frac{k'}{p} = \frac{k}{p} \mp 2\delta\sqrt{\left(\frac{a}{p}\right)^2 + \left(\frac{b}{p}\right)^2 - \frac{k}{p}} - \delta^2.$$

We now multiply through by p. Note that we add δ to the radius when p is positive and subtract it when p is negative. Hence, $\mp p = -|p|$, and the multiplication goes right inside the square root.

$$k' = k - 2\delta\sqrt{a^2 + b^2 - kp} - p\delta^2$$

Although we haven't considered straight lines as a special case, we can prove that this function behaves as expected when faced with one by using a continuity argument at $p = 0$. It is a recurring theme that everything that works correctly on proper circles will also work on straight lines for free.

```
> offset_circle :: CircleType a => Double -> Circle a -> Circle a
> offset_circle amount (Circle p a b k) = Circle p a b k'
>   where
>   amount' = coerce_constant amount
>   k' = k - 2 * amount' * sqrt (a ^ 2 + b ^ 2 - k * p) -
>     p * amount' ^ 2
```

4.1.3 Others

Other operations on circles could include translation, rotation around the origin and scaling. These are little used in practice, so are not discussed further. Implementation of these operations is left as an exercise for the reader.

4.2 Constraints

4.2.1 Passing through a point

Suppose we have surveyed a site and determined that a section of track passes through the point (x, y). We will wish to express this as a constraint in our system.

By definition, the circle $C\ (p, a, b, k)$ passes through (x, y) when

$$p(x^2 + y^2) - 2ax - 2by + k = 0.$$

The left-hand side can be used as our constraint function, giving:

```
> passing_through :: Double -> Double -> Circle Unknown -> Results
> passing_through x y (Circle p a b k) = Rs [value]
>   where
>   x' = coerce_constant x
>   y' = coerce_constant y
>   value = p * (x' ^ 2 + y' ^ 2) - 2 * a * x' - 2 * b * y' + k
```

4.2.2 Common crossings and tangent curves

At a common crossing, the two rails cross over at a fixed, pre-defined angle. We will therefore require a constraint that says that two circles cross at a fixed angle. It is also often necessary to specify that two circles are tangent to each other, which can be achieved as a special case with the angle set to zero. It is not enough that the curves simply touch each other; the orientations also have to match.

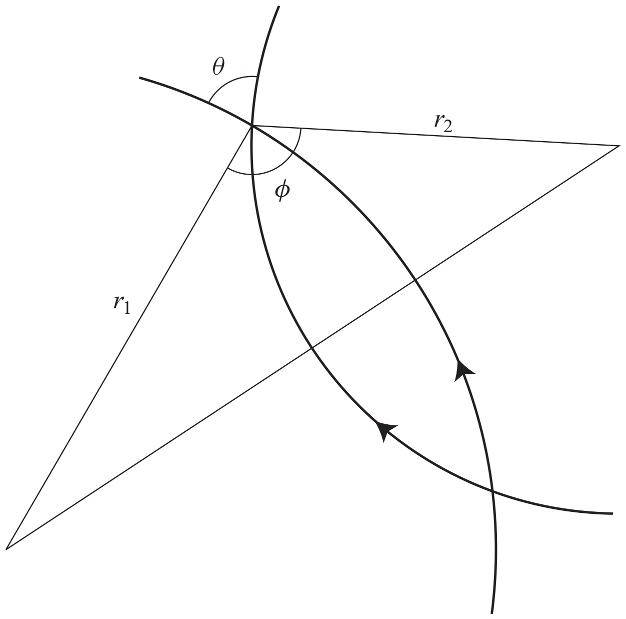

Fig. 5. Oriented circles crossing at a given angle θ.

Let $C\ (p_1, a_1, b_1, k_1)$ and $C\ (p_2, a_2, b_2, k_2)$ be two overlapping circles, and let θ be the angle at their intersection. If one was to stand at the crossing point, facing in the direction of the arrow of $C\ (p_1, a_1, b_1, k_1)$, then one would have to turn anticlockwise by an amount θ to be facing in the direction of the arrow of $C\ (p_2, a_2, b_2, k_2)$.

In Figure 5, note that $\phi = \pi - \theta$ in the case where the circles have opposite orientation, and $\phi = \theta$ where the orientations are the same. Letting r_1 and r_2 be the radii of the circles, the cosine rule therefore gives us that

$$\left(\frac{a_1}{p_1} - \frac{a_2}{p_2}\right)^2 + \left(\frac{b_1}{p_1} - \frac{b_2}{p_2}\right)^2 = r_1^2 + r_2^2 \pm 2r_1 r_2 \cos\theta$$

where the $\pm$ is positive when the orientations of the circles are opposed and negative when aligned. Multiplying out and cancelling terms gives

$$\begin{aligned} -2\frac{a_1 a_2}{p_1 p_2} - 2\frac{b_1 b_2}{p_1 p_2} &= -\frac{k_1}{p_1} - \frac{k_2}{p_2} \\ &\quad \pm 2\sqrt{\left(\frac{a_1}{p_1}\right)^2 + \left(\frac{b_1}{p_1}\right)^2 - \frac{k_1}{p_1}}\sqrt{\left(\frac{a_2}{p_2}\right)^2 + \left(\frac{b_2}{p_2}\right)^2 - \frac{k_2}{p_2}}\cos\theta. \end{aligned}$$

We now rearrange and multiply through by $p_1 p_2$. Recall that the $\pm$ sign is positive when $p_1 p_2$ is negative and negative when $p_1 p_2$ is positive. Hence, $\pm p_1 p_2 = -|p_1 p_2|$ and so we can multiply right into the square roots, giving

$$k_1 p_2 + k_2 p_1 + 2\sqrt{a_1^2 + b_1^2 - k_1 p_1}\sqrt{a_2^2 + b_2^2 - k_2 p_2}\cos\theta - 2a_1 a_2 - 2b_1 b_2 = 0.$$

We can use the left-hand side of this equation as the constraint function for the Newton–Raphson iteration, giving the following code:

```
> crossing_angle :: Double -> Circle Known -> Circle Unknown ->
>   Results
> crossing_angle theta c1 c2 = Rs [value]
>   where
>   Circle p1 a1 b1 k1 = coerce_circle c1
>   Circle p2 a2 b2 k2 = c2
>   sqrt1 = sqrt (a1 ^ 2 + b1 ^ 2 - k1 * p1)
>   sqrt2 = sqrt (a2 ^ 2 + b2 ^ 2 - k2 * p2)
>   value = k1 * p2 + k2 * p1 - 2 * a1 * a2 - 2 * b1 * b2 +
>     2 * sqrt1 * sqrt2 * coerce_constant (cos theta)
```

The constraint that two circles are tangent to each other is a special case of this:

```
> tangent_to :: Circle Known -> Circle Unknown -> Results
> tangent_to = crossing_angle 0
```

4.2.3 Fixed radius

A constraint that is often required is that the radius of the curve is fixed at some pre-defined value. The radius r of $C\ (p,a,b,k)$ is given by

$$r^2 = \left(\frac{a}{p}\right)^2 + \left(\frac{b}{p}\right)^2 - \frac{k}{p}.$$

Rearranging this gives us

$$rp = \pm\sqrt{a^2 + b^2 - kp}.$$

When r is positive, then we get an anticlockwise circle when p is also positive, and hence, when we take the positive square root. Similarly, we take the negative square root when we wish to obtain a clockwise circle. This gives us our constraint functions.

```
> anticlockwise_radius :: Double -> Circle Unknown -> Results
> anticlockwise_radius r (Circle p a b k) = Rs [value]
>   where
>   r' = coerce_from_double r
>   value = r' * p - sqrt (a ^ 2 + b ^ 2 - k * p)

> clockwise_radius :: Double -> Circle Unknown -> Results
> clockwise_radius r (Circle p a b k) = Rs [value]
>   where
>   r' = coerce_from_double r
>   value = r' * p + sqrt (a ^ 2 + b ^ 2 - k * p)
```

The constraint that ensures we have a straight line is simply $p = 0$.

```
> is_line :: Circle Unknown -> Results
> is_line (Circle p a b k) = Rs [p]
```

4.2.4 Concentric circles

We will sometimes need to be able to express the constraint that two sections of track follow each other side-by-side: they are concentric. Two circles $C\ (p_1, a_1, b_1, k_1)$ and $C\ (p_2, a_2, b_2, k_2)$ are concentric precisely when the vectors $(p_1, a_1, b_1)^T$ and $(p_2, a_2, b_2)^T$ are collinear.

Let P be the plane passing through the point $(p_1, a_1, b_1)^T$ and perpendicular to the line joining that point to the origin. Let x be the real number such that $x(p_2, a_2, b_2)^T$ lies on the plane P. We can readily see that the two vectors are collinear when

$$x \begin{pmatrix} p_2 \\ a_2 \\ b_2 \end{pmatrix} = \begin{pmatrix} p_1 \\ a_1 \\ b_1 \end{pmatrix}.$$

It is easily shown that $x = |(p_1, a_1, b_1)^T|^2 / \big((p_1, a_1, b_1)^T . (p_2, a_2, b_2)\big)$, which gives us that

$$(p_1^2 + a_1^2 + b_1^2) \begin{pmatrix} p_2 \\ a_2 \\ b_2 \end{pmatrix} = (p_1 p_2 + a_1 a_2 + b_1 b_2) \begin{pmatrix} p_1 \\ a_1 \\ b_1 \end{pmatrix}.$$

This rearranges to

$$\begin{pmatrix} p_2(a_1^2 + b_1^2) - p_1(a_1 a_2 + b_1 b_2) \\ a_2(p_1^2 + b_1^2) - a_1(p_1 p_2 + b_1 b_2) \\ b_2(p_1^2 + a_1^2) - b_1(p_1 p_2 + a_1 a_2) \end{pmatrix} = 0,$$

giving us our constraint functions. Note that we have *three* functions, when the problem of positioning the centre of the circle is a *two*-dimensional problem. This is because these three constraints are not independent of each other, but any pair of constraints is independent. We have designed our Newton–Raphson iteration to be able to cope with this: it simply deals with it gracefully by throwing away the redundant constraint.

```
> concentric_with :: Circle Known -> Circle Unknown -> Results
> concentric_with c1 c2 = Rs [value1, value2, value3]
>   where
>   Circle p1 a1 b1 k1 = coerce_circle c1
>   Circle p2 a2 b2 k2 = c2
>   value1 = p2 * (a1 ^ 2 + b1 ^ 2) - p1 * (a1 * a2 + b1 * b2)
>   value2 = a2 * (p1 ^ 2 + b1 ^ 2) - a1 * (p1 * p2 + b1 * b2)
>   value3 = b2 * (p1 ^ 2 + a1 ^ 2) - b1 * (p1 * p2 + a1 * a2)
```

We can also give a special case for directly specifying the centre point:

```
> centred_at :: Double -> Double -> Circle Unknown -> Results
> centred_at x y = concentric_with (Circle 1 x y 0)
```

4.3 Examples

Having seen these various combinators for transforming circles and for specifying constraints that they must satisfy, we give a few small examples to illustrate how they can be combined.

4.3.1 Creating a small layout

Suppose that we have surveyed a site and found that a particular section of track has radius of curvature r and passes through points (x_1, y_1) and (x_2, y_2). Suppose we wish to orient the circle so that the rotation is in a clockwise direction as we move from the first to the second point. Then, we can calculate where this section of track lies as follows:

```
> circle1 :: Circle Known
> circle1 = find guess1 $ satisfying
>   [passing_through x1 y1,
>   passing_through x2 y2,
>   clockwise_radius r]
```

Suppose the this circle adjoins a straight line that passes through point (x, y). Then, the straight line is specified as:

```
> circle2 :: Circle Known
> circle2 = find guess2 $ satisfying
>   [tangent_to circle1,
>   passing_through x y,
>   is_line]
```

Now suppose that these two circles are actually on a double-track section of line. Standard practice is to separate adjacent lines by 3.405 m.

```
> [circle1', circle2'] = map (offset_circle 3.405) [circle1, circle2]
```

4.3.2 Over- and under-specifying circles

The constraint solver is designed so that it can handle under- or over-specified circles, providing the specification is self-consistent.

```
> circle3 :: Circle Known
> circle3 = find (clockwise_circle 2 0 2) $ satisfying
>   [tangent_to (clockwise_circle 1 1 1),
>   tangent_to (clockwise_circle 3 3 1),
>   tangent_to (clockwise_circle 5 1 1),
>   clockwise_radius 3,
>   clockwise_radius 3]
```

When a circle is under-specified, the result is a circle that satisfies the constraints and is in some sense close to the initial estimate.

```
> circle4 :: Circle Known
> circle4 = find (clockwise_circle 2 0 2) $ satisfying
>   [anticlockwise_radius 1]
```

This facility comes into its own when trying to find an initial estimate of a solution but one of the constraints is being awkward: we can temporarily omit the bad constraint from our list and find a circle that satisfies the remaining constraints. This then becomes the initial estimate for the problem that includes the difficult constraint.

4.3.3 More complicated relationships

Imagine that a site has a manhole at known position (x, y) and that we want to specify that the track passes exactly d metres away from the manhole. How do we write this constraint?

```
> near_manhole :: Double -> Double -> Double ->
>   Circle Unknown -> Results
> near_manhole x y d = passing_through x y . offset_circle d
```

We are taking the *unknown* circle, offsetting it, and specifying that the transformed unknown circle passes through the known point.

4.3.4 The user's perspective

The end user of our system does not necessarily know anything about how we're representing circles. There's no hint of the underlying mechanism by which the calculations are performed. As we can see from these examples, we only have to tell the system about the relationship between the various elements that the layout is built from. We can even manipulate unknown circles if we wish, and the system will happily deal with them in an appropriate manner. Our language is strongly compositional, with no side effects.

5 PDF file generation

Having calculated the layout of a section of railway track, we want a graphical representation of it. Adobe's PDF (Adobe Systems Inc. 2000) seems like a suitable format, being freely available for use and elegantly designed. This paper is not the place to discuss the technical details of the PDF format: the reader is referred to Adobe Systems (2000) instead.

Most of the work of writing to the PDF file will be handled by a plotting monad, enabling us to say, for example

```
> main = plotPDF plotter_settings $ do
>   colour Red                        -- Change the colour of the plot.
>   plot [circle1, circle2, etc]      -- Plot list of tangent circles.
>   mark x y                          -- Put a cross on the map at
>                                     -- these coordinates.
>   plot_circle circle3               -- Plot a single circle.
```

The details of these functions will be elided, as they are not conceptually difficult. For completeness, we will simply give a type signature and explanation of what they do.

Because there is no Haskell PDF package available at present that is capable of modifying a PDF file, `plotPDF` is rather primitive and requires the PDF file to be specially prepared. This is discussed further in the comments in the downloadable code.

In order for the plotter to produce its output, it must be informed of various pieces of information, such as the name of the file to plot to, the position of the origin, which way is north, and the scale of the plot.

```
> data PlotterSettings = PlotterSettings {file :: FilePath,
>   origin :: (Double, Double), north :: Double, scale :: Double}
```

We then convert our plotting monad into an IO computation by using `plotPDF`:

```
> plotPDF :: PlotterSettings -> Plot a -> IO a
```

5.1 Plotting commands

When we wish to plot a section of track onto the map, we will pass a list of circles to the `plot` function. All adjacent circles in the list are required to be tangent to one another. The result will be a PDF plot of those circles, joined at their tangent points.

```
> plot :: [Circle] -> Plot ()
```

As well as plotting the railway lines, it was found useful to be able to change the colour of the plot. This enhances the clarity of the resulting plot and can be used to convey further information about what is to be built. In common usage, black represents track that will not be affected by construction works, while green is for track that will be removed and red is for track that will be added. (Think: red stops and green goes.)

```
> data Colour = Red | Green | Blue | Purple | Black | Grey | Brown
>   deriving (Read, Show, Eq)

> colour :: Colour -> Plot ()
```

It proved to be very useful to be able to mark points and circles onto the plot for debugging purposes. This was particularly true when trying to find initial estimates for the Newton–Raphson iteration. It also helped a great deal when the iteration refused to converge – often, plotting a few circles showed that a solution was impossible. Even when the iteration did converge, it was reassuring to be able to check that it had converged on the intended solution.

```
> mark :: Double -> Double -> Plot ()
> plot_circle :: Circle -> Plot ()
```

6 Conclusion

The practical need for a library for the design of railway track layouts has led to the use of Haskell for the purpose. A functional programming language is ideally suited for the task, as it enables us to express the relationships between the various track layout components in a clean manner. The system described here facilitates the complete process of designing a track layout, including a wide range of combinators to cover most situations that arise in practice. It has been used 'for real' in the design of the passing loop at Thuxton and in other places as well.

It came as quite a surprise when we realised how to represent circles and straight lines in the uniform manner described in Section 3. Previously, we had been representing these as two separate cases using an algebraic datatype. Our case-free representation made it possible to specify and solve arbitrary constraints using our Newton–Raphson iterator, and this proved to be the key step in the development of our package. A consequence of this is that we don't need to use Haskell's pattern-matching facility at all: each datatype has a single constructor. The problem is that straight lines are a limiting case of proper circles, whereas algebraic datatypes would tend to treat the two cases entirely separately and ignore the continuity that exists in this situation.

A key benefit of the use of a Newton–Raphson iteration has been that our constraints can be used in arbitrary combinations. We can create new constraints by applying transformations to existing ones, and we can even introduce completely novel constraints that were not conceived of when the package was designed – all without modifying the underlying constraint solver. It has been a recurring theme during the development of the software that attempting to produce exact solutions to problems usually resulted in overwhelming complexity, whereas our approximate solutions are highly accurate and offer many practical benefits.

We have given an illustration of how the Newton–Raphson method can be used, in general, for finding the zeros of a function without having to explicitly calculate all of the derivatives by hand. This saves a great deal of time spent debugging and is much more reliably correct. Although the method is applied to just a single problem here, a large and important class of numerical constraint problems can be solved using these methods.

Importantly, the design of this software enables its use for rapid prototyping. Typically, the user would create a rough layout that may not satisfy all of the requirements, plot the PDF, and then, successively refine. Even for such a simple layout as Thuxton, this cycle was repeated many hundreds of times to experiment with the various different options that were available to us. 'What would happen if I were to put that there instead of here?' 'Could I use a bigger one of those?' 'How sharp would that curve have to be?'

The code described in this paper can be downloaded from the JFP web site, along with the actual plans used at Thuxton. Photos of the construction of the passing loop (including some by the author) can be found at `http://www.mnr.org.uk/photos/thuxton/`.

Acknowledgments

Thanks to Ralf Hinze and Colin Runciman for their helpful comments on earlier drafts of this paper.

References

Adobe Systems (2000) *PDF Reference*. Adobe Press.

Bird, R. (1998) *Introduction to Functional Programming Using Haskell*. 2nd ed., Prentice Hall.

Cope, G. H. (1993) *British Railway Track*. 6th ed., Permanent Way Institution.

Karczmarczuk, J. (2001) Functional differentiation of computer programs. *Higher-Order and Symb. Comput*, **14**(1), 35–57.

Peyton Jones, S. (ed) (2003) *Haskell 98 Language and Libraries—The Revised Report*. Cambridge, England, UK: Cambridge University Press.

Pfeiffer, R. E. & van Hook, C. (1993) Circles, vectors, and linear algebra. *Math. Mag*., **66**(2), 75–86.

Press, W. H., Teukolsky, S. A., Vetterling, W. T. & Flannery, B. P. (2007) *Numerical Recipes*. 3rd ed., Cambridge, England, UK: Cambridge University Press.