compass

Claude Heiland-Allen

2013–2015

Contents

1 example.hs

```
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE TypeFamilies #-}
import Math.Compass

import Diagrams.Prelude hiding (circle)
import qualified Diagrams.Prelude as D
import Diagrams.Backend.SVG.CmdLine (defaultMain)

import Data.Complex
import Data.Maybe (fromJust)

point (EPoint (x :+ y)) = D.circle 0.02 # translate (r2 (x, y))
curve (ELine (EPoint z) (EPoint w)) = fromVertices [ p2 (px, py), p2 (qx, qy) ]
  where
    px :+ py = z + 4 * (w - z)
    qx :+ qy = w + 4 * (z - w)
curve (ECircle (EPoint (x :+ y)) r) = D.circle r # translate (r2 (x, y))

example embedding = mconcat (map point [a0, b0, c0, d0, e0] ++ map curve [l0, m0↙
    ↳ , n0])
  where
    a0 = EPoint ((-0.5) :+ 0.25)
    b0 = EPoint (  0.5  :+ 0.5 )
    c0 = EPoint ((-0.5) :+ 0.5 )
    d0 = EPoint (  0.5  :+ 0.25)
    Just a = unembedPoint embedding a0
    Just b = unembedPoint embedding b0
    Just c = unembedPoint embedding c0
    Just d = unembedPoint embedding d0
    l = line a b
    m = line c d
    n = circle e a
    [e] = intersect l m
    l0 = embedConstruct embedding l
    m0 = embedConstruct embedding m
    n0 = embedConstruct embedding n
```

```
    e0 = embedPoint embedding e

main = defaultMain . hcat . map (pad 1.1 . centerXY . lc black . lw 0.002 . ↙
  ↳ withEnvelope' s . clipBy' s . stroke) $
  [ example euclidean :: Path V2 Double
  , example poincareDisc `mappend` D.circle 1
  , example poincareHalfPlane `mappend` hrule 4
--  , example stereographic
  ]
  where
    s = square 2.2
    withEnvelope' x y = withEnvelope (x `asTypeOf` y) y
    clipBy' x y = withEnvelope (x `asTypeOf` y) y
```

2 Math/Compass/Class.hs

```
{-# LANGUAGE TypeFamilies #-}
module Math.Compass.Class where

class Compass s where
  type Scalar s :: *
  data Point s :: *
  data Vector s :: *
  data Construct s :: *
  data Transform s :: *
  origin :: Point s
  one :: Vector s
  vector :: Point s -> Vector s
  line :: Point s -> Point s -> Construct s
  circle :: Point s -> Point s -> Construct s
  intersect :: Construct s -> Construct s -> [Point s]
  distance :: Point s -> Point s -> Scalar s
  angle :: Point s -> Point s -> Point s -> Scalar s
  scale :: Scalar s -> Vector s -> Vector s
  rotate :: Scalar s -> Vector s -> Vector s
  compose :: Transform s -> Transform s -> Transform s
  transform :: Transform s -> Point s -> Point s
  translation :: Vector s -> Transform s
  rotation :: Scalar s -> Transform s

  perpendicularBisector :: Point s -> Point s -> Construct s
  perpendicularBisector p q = l
    where
      c = circle p q
      c' = circle q p
      [p', q'] = intersect c c'
      l = line p' q'

  midpoint :: Point s -> Point s -> Point s
  midpoint p q = m
    where
      l = line p q
      l' = perpendicularBisector p q
      [m] = intersect l l'

data Embed s t = Embed
  { embedPoint :: Point s -> Point t
```

```
    , unembedPoint :: Point t -> Maybe (Point s)
    , embedConstruct :: Construct s -> Construct t
    , unembedConstruct :: Construct t -> Maybe (Construct s)
  }
```

3 Math/Compass/Euclidean.hs

```
{-# LANGUAGE TypeFamilies, StandaloneDeriving, FlexibleInstances #-}
module Math.Compass.Euclidean where

import Math.Compass.Class

import Data.Complex (Complex((:+)), magnitude, phase, cis)

data E r = E

instance RealFloat r => Compass (E r) where
    type Scalar (E r) = r
    data Point (E r) = EPoint !(Complex r)
    data Vector (E r) = EVector !(Complex r)
    data Construct (E r) = ELine !(Point (E r)) !(Point (E r)) | ECircle !(Point (E r)) !r
    data Transform (E r) = ETransform !r !r !r !r !r !r
    origin = EPoint 0
    one = EVector 1
    vector (EPoint p) = EVector p
    line = ELine
    circle p q = ECircle p (distance p q)

    intersect (ELine (EPoint (x1 :+ y1)) (EPoint (x2 :+ y2))) (ELine (EPoint (x3 :+ y3)) (EPoint (x4 :+ y4)))
--      | m == 0 || n == 0 = [] -- coincident
      | d == 0 = [] -- parallel
      | otherwise = [EPoint (x :+ y)]
      where
        d = (y4 - y3) * (x2 - x1) - (x4 - x3) * (y2 - y1)
        m = (x4 - x3) * (y1 - y3) - (y4 - y3) * (x1 - x3)
        n = (x2 - x1) * (y1 - y3) - (y2 - y1) * (x1 - x3)
        x = x1 + (x2 - x1) * m / d
        y = y1 + (y2 - y1) * m / d

    intersect (ELine (EPoint (x1 :+ y1)) (EPoint (x2 :+ y2))) (ECircle (EPoint (x3 :+ y3)) r3)
      | d == 0 = [EPoint ((x1 + u  * (x2 - x1)) :+ (y1 + u  * (y2 - y1)))]
      | d >  0 = [EPoint ((x1 + u1 * (x2 - x1)) :+ (y1 + u1 * (y2 - y1))), EPoint ((x1 + u2 * (x2 - x1)) :+ (y1 + u2 * (y2 - y1)))]
      | otherwise = []
      where
        a = (x2 - x1) * (x2 - x1) + (y2 - y1) * (y2 - y1)
        b = 2 * ((x2 - x1)*(x1 - x3) + (y2 - y1)*(y1 - y3))
        c = x3*x3 + y3*y3 + x1*x1 + y1*y1 - 2*(x3*x1 + y3*y1) - r3*r3
        d = b*b - 4 * a * c
        u  =  -b / (2 * a)
        u1 = (-b + sqrt d) / (2 * a)
        u2 = (-b - sqrt d) / (2 * a)

    intersect (ECircle (EPoint (x0 :+ y0)) r0) (ECircle (EPoint (x1 :+ y1)) r1)
```

```
      | d == 0 && r0 == r1 = [] -- coincident
      | d == r0 + r1 = [EPoint (x2 :+ y2)]
      | d >  r0 + r1 = [] -- disjoint
      | d <  abs (r0 - r1) = [] -- concentric
      | otherwise = [EPoint ((x2 + x3) :+ (y2 + y3)), EPoint ((x2 - x3) :+ (y2 -↙
          ↳ y3))]
      where
        dx = x1 - x0
        dy = y1 - y0
        d  = sqrt(dx * dx + dy * dy)
        a  = (r0 * r0 - r1 * r1 + d * d) / (2 * d);
        h  = sqrt(r0 * r0 - a * a)
        x2 = x0 + a * (x1 - x0) / d
        y2 = y0 + a * (y1 - y0) / d
        x3 =  h * (y1 - y0) / d
        y3 = -h * (x1 - x0) / d

    intersect c l = intersect l c

    distance (EPoint p) (EPoint q) = magnitude (p - q)

    angle (EPoint p1) (EPoint p2) (EPoint p3) = phase ((p1 - p2) / (p3 - p2))

    scale s (EVector (x :+ y))= EVector (s * x :+ s * y)

    rotate a (EVector v) = EVector (cis a * v)

    compose (ETransform a b c d e f) (ETransform u v w x y z) = ETransform (a * ↙
        ↳ u + b * x) (a * v + b * y) (a * w + b * z + c) (d * u + e * x) (d * v ↙
        ↳ + e * y) (d * w + e * z + f)

    transform (ETransform a b c d e f) (EPoint (x :+ y)) = EPoint ((a * x + b * ↙
        ↳ y + c) :+ (d * x + e * y + f))

    translation (EVector (x :+ y)) = ETransform 1 0 x 0 1 y

    rotation a = ETransform c (-s) 0 s c 0
      where
        c :+ s = cis a

    midpoint (EPoint p) (EPoint q) = EPoint ((p + q) / 2)

deriving instance Read r => Read (Point (E r))
deriving instance Read r => Read (Vector (E r))
deriving instance Read r => Read (Construct (E r))
deriving instance Read r => Read (Transform (E r))
deriving instance Show r => Show (Point (E r))
deriving instance Show r => Show (Vector (E r))
deriving instance Show r => Show (Construct (E r))
deriving instance Show r => Show (Transform (E r))
deriving instance Eq r => Eq (Point (E r))
deriving instance Eq r => Eq (Vector (E r))
deriving instance Eq r => Eq (Construct (E r))
deriving instance Eq r => Eq (Transform (E r))

euclidean :: Embed (E r) (E r)
euclidean = Embed
```

```
    { embedPoint = id
    , unembedPoint = Just
    , embedConstruct = id
    , unembedConstruct = Just
    }
```

4 Math/Compass.hs

```
{-# LANGUAGE MultiParamTypeClasses, FlexibleInstances, FunctionalDependencies #-}
module Math.Compass
  ( module Math.Compass.Class
  , module Math.Compass.Euclidean
  , module Math.Compass.Hyperbolic
  ) where

import Math.Compass.Class
import Math.Compass.Euclidean
import Math.Compass.Hyperbolic
```

5 Math/Compass/Hyperbolic.hs

```
{-# LANGUAGE TypeFamilies, StandaloneDeriving, FlexibleInstances #-}
module Math.Compass.Hyperbolic where

import Data.Complex (Complex((:+)), magnitude, phase, polar, mkPolar, cis)
import Data.Maybe (mapMaybe)

import Math.Compass.Class
import Math.Compass.Euclidean

data H r = H

poincareDisc :: RealFloat r => Embed (H r) (E r)
poincareDisc = Embed
  { embedPoint = \(HPoint z w) -> EPoint (z / w)
  , unembedPoint = \(EPoint z) -> if magnitude z < 1 then Just (HPoint z 1) else Nothing
  , embedConstruct = \c -> case c of
      HLine p q
        | a2 < 1e-12 || abs d < 1e-12 -> line (embedPoint poincareDisc p) (embedPoint poincareDisc q)
        | otherwise -> circle (EPoint (ux :+ uy)) (embedPoint poincareDisc p)
        where
          ax :+ ay = fromPoint p
          bx :+ by = fromPoint q
          a2 = ax * ax + ay * ay
          b2 = bx * bx + by * by
          cx = ax / a2
          cy = ay / a2
          c2 = cx * cx + cy * cy;
          d = 2 * (ax * (by - cy) + bx * (cy - ay) + cx * (ay - by))
          ux = (a2*(by - cy) + b2*(cy - ay) + c2*(ay - by)) / d
          uy = (a2*(cx - bx) + b2*(ax - cx) + c2*(bx - ax)) / d
          dx = ux - bx
          dy = uy - by
```

```
            r = sqrt(dx * dx + dy * dy)
        HCircle centre radius -> circle centre' p'
          where
            v = (normalize . vector) centre
            p = transform (translation (scale (distance origin centre - radius) v)↙
                ↳ ) origin
            q = transform (translation (scale (distance origin centre + radius) v)↙
                ↳ ) origin
            p' = embedPoint poincareDisc p
            q' = embedPoint poincareDisc q
            centre' = midpoint p' q'
    , unembedConstruct = error "unembedConstruct poincareDisc"
    }

poincareHalfPlane :: RealFloat r => Embed (H r) (E r)
poincareHalfPlane = Embed
    { embedPoint = \(HPoint z w) -> EPoint (-i * (z + w) / (z - w))
    , unembedPoint = \(EPoint z@(_ :+ y)) -> if y > 0 then Just (HPoint (z - i) (z↙
        ↳ + i)) else Nothing
    , embedConstruct = \c -> case c of
        HLine p q -> case intersect (perpendicularBisector p' q') (line origin (↙
            ↳ EPoint 1)) of
          [c'@(EPoint (cx :+ _))] | abs cx < 1e6 -> circle c' p'
          _ -> line p' q'
          where
            p' = embedPoint poincareHalfPlane p
            q' = embedPoint poincareHalfPlane q
        HCircle p r ->
          let EPoint (a :+ b) = embedPoint poincareHalfPlane p
              y = b * cr
              z = b * sqrt (cr * cr - 1)
              cr = cosh r
          in  circle (EPoint (a :+ y)) (EPoint (a :+ (y + z)))
    , unembedConstruct = error "unembedConstruct poincareHalfPlane"
    }
    where
      i = 0 :+ 1

instance RealFloat r => Compass (H r) where
    type Scalar (H r) = r
    data Point (H r) = HPoint !(Complex r) !(Complex r)
    data Vector (H r) = HVector !(Complex r)
    data Construct (H r) = HLine !(Point (H r)) !(Point (H r)) | HCircle !(Point↙
        ↳ (H r)) !r
    data Transform (H r) = HTransform !(Complex r) !(Complex r) !(Complex r) !(↙
        ↳ Complex r)
    origin = HPoint 0 1
    one = HVector 1
    vector p@(HPoint z w) = HVector (mkPolar (distance p origin) (phase (z / w))↙
        ↳ )
    line = HLine
    circle p q = HCircle p (distance p q)

    rotate a (HVector v) = HVector (cis a * v)
    scale s (HVector (x :+ y)) = HVector (s * x :+ s * y)

    intersect l m = mapMaybe (unembedPoint poincareDisc) $ intersect (↙
```

```
        ↳ embedConstruct poincareDisc l) (embedConstruct poincareDisc m)

    distance p1 p2 = acosh (1 + 2 * magnitudeSquared (u - v) / ((1 - ↙
        ↳ magnitudeSquared u) * (1 - magnitudeSquared v)))
      where
        u = fromPoint p1
        v = fromPoint p2

    angle p1 p2 p3 = angle (t p1) origin (t p3)
      where
        t = embedPoint poincareDisc . transform (translation . scale (-1) . ↙
            ↳ vector $ p2)

    compose (HTransform a b c d) (HTransform u v x y) = HTransform (a * u + b * ↙
        ↳ x) (a * v + b * y) (c * u + d * x) (c * v + d * y)

    transform (HTransform a b c d) (HPoint z w) = HPoint (a * z + b * w) (c * z ↙
        ↳ + d * w)

    translation (HVector v)
      | l > 0 = rotation s `compose` m `compose` rotation (negate s)
      | otherwise = HTransform  1 0 0 1
      where
        (l, s) = polar v
        e = exp l
        f = (e + 1) :+ 0
        g = (e - 1) :+ 0
        m = HTransform f g g f

    rotation a = HTransform (cis a) 0 0 1

fromPoint (HPoint z w) = z / w

magnitudeSquared (x :+ y) = x * x + y * y

normalize (HVector v)
  | v == 0 = HVector 1
  | otherwise = HVector (signum v)

deriving instance Read r => Read (Point (H r))
deriving instance Read r => Read (Vector (H r))
deriving instance Read r => Read (Construct (H r))
deriving instance Read r => Read (Transform (H r))
deriving instance Show r => Show (Point (H r))
deriving instance Show r => Show (Vector (H r))
deriving instance Show r => Show (Construct (H r))
deriving instance Show r => Show (Transform (H r))
deriving instance Eq r => Eq (Point (H r))
deriving instance Eq r => Eq (Vector (H r))
deriving instance Eq r => Eq (Construct (H r))
deriving instance Eq r => Eq (Transform (H r))
```