Introduction

Here is the final implementation of P^2 algorithm, an algorithm for estimation of quantiles under strict memory limits.

```
{-# LANGUAGE OverloadedStrings      #-}
{-# LANGUAGE DuplicateRecordFields  #-}
{-# LANGUAGE RecordWildCards        #-}
{-# LANGUAGE FlexibleContexts       #-}
```

We use only strict code in the implementation to reduce thunks.

```
{-# LANGUAGE BangPatterns           #-}
{-# LANGUAGE MagicHash              #-}
```

```
import Data.List (sort)
import Data.Maybe (isJust, fromJust)
```

Another implementation trick is to use unboxed data types for certain calculations and vectors.

```
import GHC.Exts
import qualified Data.Vector.Unboxed as V
```

Modules included only for testing.

```
import Debug.Trace (trace, traceIO)
import Control.Monad (replicateM, forM)
import System.Random (randomIO, randomRIO, setStdGen, mkStdGen)
import qualified Statistics.Quantile as S
import Statistics.Distribution (genContinuous)
import Statistics.Distribution.Normal (standard)
import System.Random.MWC (create, Gen (..))
```

For benchmarking purposes include "criterion".

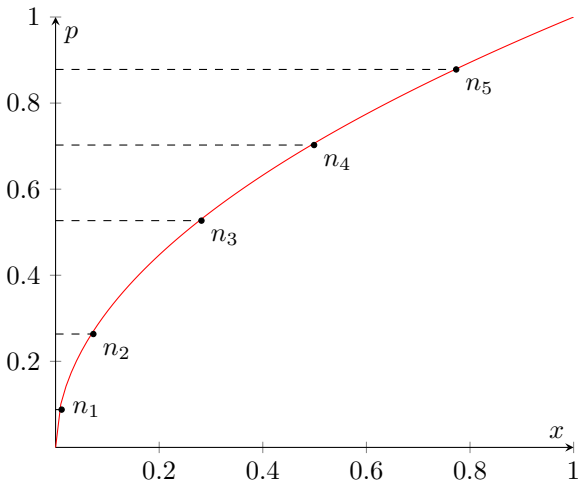
```
import Criterion.Main (defaultMain, bgroup, bench, whnf)
```

Data structure

Common interface for both PSquare and MPSquare implementation.

```
class QuantileEstimator a where
    add :: a -> Double -> a
    quantile :: a -> Double
```

Data structure describing P^2 state. It contains five markers on the cumulative distribution function curve that estimate quantiles.




```
data PSquare = PSquare
  { p :: Double
```

Quantile we need to estimate, $p \in (0, 1)$

```
  , n :: !(V.Vector Double)
```

Current marker positions.

```
  , ns :: !(V.Vector Double)
```

Desired marker positions corresponding to the estimated quantile:

- $n_1^{'} = 1$
- $n_2^{'} = (n-1)\frac{p}{2} + 1$
- $n_3^{'} = (n-1)p + 1$
- $n_4^{'} = (n-1)\frac{1+p}{2} + 1$
- $n_5^{'} = n$

```
  , dns :: !(V.Vector Double)
```

Precomputed increments in the desired marker positions.

- $dn_1^{'} \leftarrow 0$
- $dn_2^{'} \leftarrow \frac{p}{2}$
- $dn_3^{'} \leftarrow p$
- $dn_4^{'} \leftarrow \frac{1+p}{2}$
- $dn_5^{'} \leftarrow 1$

```
  , q :: !(V.Vector Double)
```

Array of marker heights, q_i is the height of ith marker.

- q_1 is the minimal observation
- q_2 is the estimate of $\frac{p}{2}$ quantile
- q_3 is the estimate of p quantile
- q_4 is the estimate of $\frac{1+p}{2}$ quantile

```
  }
  deriving (Show)

defaultPSquare = PSquare
  { p = 0.0,
    n = V.empty,
    ns = V.empty,
    dns = V.empty,
    q = V.empty
  }
```

P^2 algorithm

```
instance QuantileEstimator PSquare where
```

As mentioned before, the actual estimation value is the height of middle marker.

```
  quantile ps@PSquare {..} = q `get` 2
```

When adding a new observation, we need to accumulate necessary amount of elements to start estimating.

```
  add !ps !value
    | V.length (q ps) < 4 = addNewValue ps value
    | V.length (q ps) == 4 = initPSquare $ addNewValue ps value
    | otherwise = updatePSquare ps value
    where
      addNewValue :: PSquare -> Double -> PSquare
      addNewValue !ps@PSquare {..} !value = ps { q = V.snoc q value }

      initPSquare :: PSquare -> PSquare
      initPSquare !ps@PSquare {..} = ps
        { q = V.fromList $ sort $ V.toList q
        , n = V.fromList $ [0..4]
        , ns = V.fromList $ [0, 2 * p, 4 * p, 2 + 2 * p, 4]
        , dns = V.fromList $ [0, p/2, p, (1 + p)/2, 1]
        }
```

```
updatePSquare :: PSquare -> Double -> PSquare
updatePSquare !ps !value =
  let
    !ps' = ps `insertObservation` value
    !qn' = adjustMarkers ps'
  in ps' { q = V.map fst qn'
         , n = V.map snd qn'
         }
```

Insert a new observation and adjust marker positions.

```
insertObservation :: PSquare -> Double -> PSquare
insertObservation !ps@PSquare {..} !value =
  let
    !k = ps `findIntervalFor` value
    !newQ = ps `adjustIntervalsFor` value
    !newN = ps `incrementPositionFor` k
    !newNs = V.zipWith (+) ns dns
  in ps { n = newN, ns = newNs, q = newQ }
  where
```

Note the order following possibility, it's more likely to have a new value within the interval.

```
location :: PSquare -> Double -> Int
location !ps@PSquare {..} !value
  | value < q `get` 0 = -1
  | value <= q `get` 1 = 0
  | value <= q `get` 2 = 1
  | value <= q `get` 3 = 2
  | value <= q `get` 4 = 3
  | value > q `get` 4 = 4
```

To which interval the new observation x belongs to? Returns $k, q_k \leq x < q_{k+1}$

```
findIntervalFor :: PSquare -> Double -> Int
findIntervalFor !ps !value
  | location ps value == -1 = 0
  | location ps value == 4 = 3
  | otherwise = location ps value
```

Adjust intervals taking into account the new observation. If the new value fits into an existing interval, it doesn't change anything, otherwise adjust extreme values q_1 and q_5 to extend the corresponding intervals and include the new value.

```
adjustIntervalsFor :: PSquare -> Double -> V.Vector Double
adjustIntervalsFor !ps@PSquare {..} !value
  | location ps value == -1 = q V.// [(0, value)]
  | location ps value == 4 = q V.// [(4, value)]
  | otherwise = q
```

Increment position of markers $n_i, i \in [k, 5]$. We do this because the new observation doesn't change the intervals (except extreme values), but do shift other values to the right when inserted into the interval k.

```
incrementPositionFor :: PSquare -> Int -> V.Vector Double
incrementPositionFor !ps@PSquare {..} k =
  ifor n $ \i v ->
    if i >= k + 1
      then v + 1
      else v
```

Adjust height of markers q_2, q_3, q_4. We need to find how marker positions have shifted and recalculate marker heigh accordingly.

```
adjustMarkers :: PSquare -> V.Vector (Double, Double)
adjustMarkers !ps@PSquare {..} = for (V.enumFromN 0 5) adjustMarkers'
  where
```

Nothing to do for the extreme markers q_1, q_5.

```
    adjustMarkers' 0 = (q `get` 0, n `get` 0)
    adjustMarkers' 4 = (q `get` 4, n `get` 4)
```

For q_2, q_3, q_4 find if they are shifted and to which direction.

```
    adjustMarkers' i =
      let
        !d = (ns `get` i) - (n `get` i)
```

One trick to notice here is the branchless version of signum function, which slightly reduces branch misprediction rate.

```
        !direction = branchlessSignum d
        !qParabolic = parabolic ps i direction
```

A marker considered to be off position if the distance is more than 1.

```
        !markerOff = (
             (d >= 1 && n `get` (i + 1) - n `get` i > 1) ||
             (d <= -1 && n `get` (i - 1) - n `get` i < -1))
```

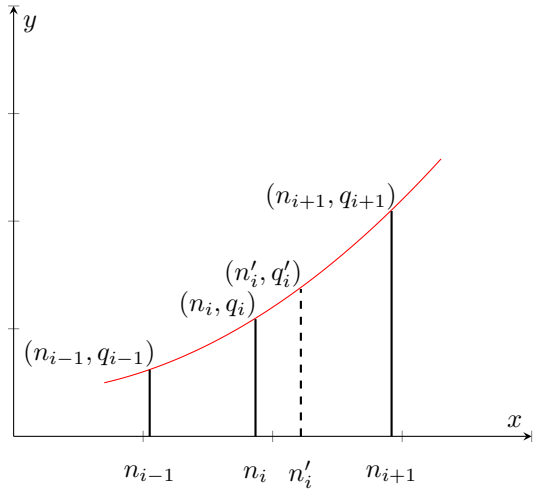
if it's indeed off the ideal place too much, recalculate the position and height.

```
        !n' = if markerOff
          then (n `get` i) + direction
          else (n `get` i)
        !q' = if markerOff
          then
```

For the algorithm to work the marker heights must always be in a nondecreasing order. If P^2 formula predict a value outside of (q_{i-1}, q_{i+1}) interval, ignore it and use linear prediction.

```
            if (q `get` (i - 1)) < qParabolic && qParabolic < (q `get` (i + 1))
              then qParabolic
              else linear ps i direction
          else q `get` i
    in (q', n')
```

P^2 or piecewise-parabolic prediction formula, which assumes that the curve passing through any three adjasent markers is a parabola $q_i = an_i^2 + bn_i + c$.



Here i is number of the marker, $d = \pm 1$ - direction taken from the previous computations.

```
parabolic :: PSquare -> Int -> Double -> Double
parabolic !ps@PSquare {..} i (D# d) =
    let
      !term1 = d /## (n_i1 -## n_i_1)
      !term2 = (n_i -## n_i_1 +## d) *## (q_i1 -## q_i) /## (n_i1 -## n_i)
      !term3 = (n_i1 -## n_i -## d) *## (q_i -## q_i_1) /## (n_i -## n_i_1)
    in
      D# (q_i +## term1 *## (term2 +## term3))
  where
    (D# q_i) = q `get` i
    (D# q_i1) = q `get` (i + 1)
    (D# q_i_1) = q `get` (i - 1)
    (D# n_i) = n `get` i
    (D# n_i1) = n `get` (i + 1)
    (D# n_i_1) = n `get` (i - 1)
```

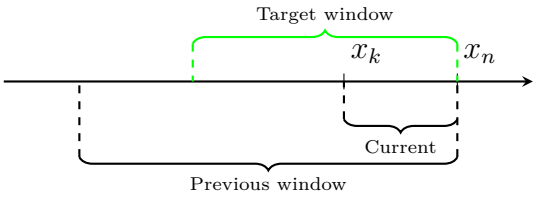
Linear value adjustion, i - the number of marker, $d = \pm 1$ - direction taken from previous computations.

```
linear :: PSquare -> Int -> Double -> Double
```

```
linear !ps@PSquare {..} i (D# dIn) =
  D# (q_i +## dIn *## (q_id -## q_i) /## (n_id -## n_i))
  where
    d = double2Int# dIn
    (D# q_i) = q `get` i
    (D# q_id) = q `get` (i + (I# d))
    (D# n_id) = n `get` (i + (I# d))
    (D# n_i) = n `get` i
```

Moving P^2 algorithm

The idea here is to maintain two estimators over the data set the previous and the current one. In this way instead of the target window quantile estimation we get two estimations, one for the bigger window and one for the smaller window, as on the following diagram.



Note: Size is necessary here, as we don't have any other meaning of tracking progress through the window. The activeEstimator values contain only 5 markers, not the actual values.

```
data MPSquare = MPSquare
  { windowSize :: Int,
    size :: Int,
    lastWindowEstimation :: Maybe Double,
    lastEstimator :: Maybe PSquare,
    activeEstimator :: PSquare
  }
  deriving (Show)

defaultMPSquare = MPSquare
  { windowSize = 0,
    size = 0,
    lastWindowEstimation = Nothing,
    lastEstimator = Nothing,
    activeEstimator = defaultPSquare
  }
```

```
instance QuantileEstimator MPSquare where
  add !mps@MPSquare {..} !value
```

On this stage fill active estimator with values.

```
    | not lastReady && size < windowSize =
      mps { size = size + 1
          , activeEstimator = activeEstimator `add` value }
```

Wrap up the first window and remember an estimator for it for the next window.

```
    | not lastReady && size == windowSize = mps
      { lastWindowEstimation = Just $ quantile activeEstimator
      , lastEstimator = Just activeEstimator
      , activeEstimator = defaultPSquare { p = p activeEstimator } `add` value
      , size = 0
      }
```

Last estimator is already present, so fill both active and the last one with values from the new window.

```
    | lastReady && size < windowSize =
      mps { size = size + 1
          , activeEstimator = activeEstimator `add` value
          , lastEstimator = fmap (`add` value) lastEstimator}
```

Wrap up another window, reset the last remembered estimator with the new one from the current window.

```
    | lastReady && size == windowSize = mps
      { lastWindowEstimation = Just $ quantile activeEstimator
      , lastEstimator = Just activeEstimator
      , activeEstimator = add defaultPSquare { p = p activeEstimator } value
      , size = 0
      }
    where
      lastReady = isJust lastEstimator

  quantile !mps@MPSquare {..}
```

If there isno estimator from the previous window yet and the active one is not ready yet:

```
    | not lastReady && not activeReady = 0
```

If there is no an estimator from the previous window yet, but there is already enough values in the active one:

```
    | not lastReady && activeReady = quantile activeEstimator
```

Last estimator is already remembered, but the active one is not ready yet:

```
    | lastReady && not activeReady = fromJust lastWindowEstimation
```

Last estimator is there and the active one is also ready, combine them both:

```
    | lastReady && activeReady =
      let
        !w2 = size ./. windowSize
        !w1 = 1.0 - w2
      in
        w1 * (quantile $ fromJust lastEstimator) +
        w2 * (quantile activeEstimator)
   where
     lastReady = isJust lastEstimator
     activeReady = size > 4
```

Testing

Function to generate a vector of random values of predefined size.

```
randomValues :: Int -> IO (V.Vector Double)
randomValues size = V.replicateM size (randomIO :: IO Double)
```

Function to generate a vector of values following normal distribution (0, 1) of predefined size.

```
normalDist :: Int -> Gen RealWorld -> IO (V.Vector Double)
normalDist size gen = V.replicateM size (genContinuous standard gen)
```

Function to generate a vector of values with sin pattern, noise and outliers.

```
sinPattern' :: Int -> IO (V.Vector Double)
sinPattern' size = do
  batches <- replicateM nrWaves $ sinPattern batch
  pure $ V.concat batches
  where
    nrWaves = 4
    batch = size `div` nrWaves

sinPattern :: Int -> IO (V.Vector Double)
sinPattern fullSize = V.generateM fullSize $ \i -> do
  let x = (fromIntegral (i `div` multiplexing))/(fromIntegral size) * 2 * pi
  noise <- randomRIO (-3, 3) :: IO Double
  outlier <- randomRIO (0, 200) :: IO Int
  outlierValue <- if outlier == 200
    then randomRIO (20, 50) :: IO Double
    else pure 0
  pure $ 10 + (sin x) * 5 + noise + outlierValue
  where
```

```
    multiplexing = 2
    size = fullSize `div` multiplexing

main = longRun
```

Compare P^2 (moving window version) estimation with the exact one over the normally distributed data (standard, (0, 1)).

```
distribution :: IO MPSquare
distribution = do
    setStdGen $ mkStdGen 1
    gen <- create
    let size = 1000
    values <- normalDist size gen
    let startPs = (defaultPSquare {p = 0.5})
    let startMps = defaultMPSquare
          { activeEstimator = startPs
          , windowSize = 100
          }

    foldForM (V.drop 100 values) startMps $ \mps' i v -> do
      let result = mps' `add` v
      putStrLn $
        show v ++ " " ++
        show (quantile mps') ++ " " ++
```

Here index `i` is not over `values`, rather than over values without first `window` elements.

```
        show (S.quantile S.spss 50 100 $ V.slice i 100 values)
      pure $ result
```

Test convergence of P^2 algorithm towards the exact value with growing dataset.

```
convergence :: IO [()]
convergence = do
  forM [1..100] $ \i -> do
    setStdGen $ mkStdGen 1
    gen <- create
    let size = i * 100
    values <- randomValues size
    let startPs = (defaultPSquare {p = 0.95})
    let ps = V.foldl' add startPs values
    putStrLn $
      show (quantile ps) ++ " " ++
      show (S.quantile S.spss 95 100 values)
```

Comparison of P^2 algorithm (moving window version) against the exact values for sin-distributed data with noise and outliers.

```
convergenceMoving :: IO MPSquare
convergenceMoving = do
  let i = 1000
  let size = i * 100
  let window = 100
  values <- sinPattern' size
  let startPs = (defaultPSquare {p = 0.95})
  let startMps = defaultMPSquare
        { activeEstimator = startPs
        , windowSize = window
        }
  let firstValues = V.take window values
  let mps = V.foldl' add startMps firstValues

  foldForM (V.drop window values) mps $ \mps' i value -> do
    let result = mps' `add` value
    putStrLn $
      show value ++ " " ++
      show (quantile result) ++ " " ++
```

Here index `i` is not over `values`, rather than over values without first `window` elements.

```
      show (S.quantile S.spss 95 100 $ V.slice i window values)
    pure $ result
```

Performance testing functions.

```
testExact :: V.Vector Double -> Double
testExact values = S.quantile S.spss 95 100 values

testEstimation :: V.Vector Double -> Double
testEstimation values = quantile $ V.foldl' add startPs values
  where
    startPs = (defaultPSquare {p = 0.95})
```

Simlpy a long run with waiting between data generation and the algorithm run to run perf.

```
longRun :: IO ()
longRun = do
  !values <- randomValues 500000000
  print $ show $ map (\i -> testEstimation values) [1..1000]
```

Benchmarking with criterion.

```
benchmark :: IO ()
benchmark = do
  !values100000 <- randomValues 100000
  !values10000 <- randomValues 10000
```

```
  !values1000 <- randomValues 1000

  defaultMain [
    bgroup "Estimation"
      [ bench "1000" $ whnf testEstimation values1000
      , bench "10000" $ whnf testEstimation values10000
      , bench "100000" $ whnf testEstimation values100000
      ]
    ]
```

Utility functions

To shave off a bit more, use unsafe version of index access.

```
get :: V.Vector Double -> Int -> Double
get = V.unsafeIndex
```

Silly replacements.

```
ifor = flip V.imap
for = flip V.map
foldForM val acc f = V.ifoldM' f acc val
```

Branchless version of signum, taken from https://gitlab.haskell.org/ghc/ghc/-/issues/9342

```
branchlessSignum :: Double -> Double
branchlessSignum !(D# x) = D# ( int2Double# ((x >## 0.0##) -# (x <## 0.0##)) )
```

To simplify numeric type conversions from `Int` to `Double` in some formulas a bit, introduce special arithmetic operators that will do this conversion under the hood. Mostly it was necessary in the original "boxed" implementation of parabolic and linear functions.

```
(./.) :: (Integral a) => a -> a -> Double
(./.) x y = (fromIntegral x) / (fromIntegral y)
```