```haskell
-------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------
--portOpt.hs file

--module Main(main) where
import PSO
import Data.List
import System.Random
import System.Environment(getArgs)
import Data.Time
import Data.Map hiding (map,foldr)
import Text.Printf

-- Example of adjustment parameters for PSO
-- (taken from M.E.H Pedersen, Tuning & Simplifying Heuristical Optimization)
wpg1 :: (Double,Double,Double)
wpg1 = (-0.16,1.89,2.12)
-- (taken from )
wpg2 :: (Double,Double,Double)
wpg2 = (0.7,1.45,1.49)

----Example Number of particles
--np1 = 100 -- for testing
----Example Number of iterations
--nit1 = 1000-- for testing

--------------------------------------------
----------- Financial variables ------------
--------------------------------------------
----Risky characteristic
--risk = 0.5 -- for testing
----Risk aversion
--aversion = 3 -- for testing
----Required expected return
--reqExpR = 0.01 -- for testing
--------------------------------------------
--------------------------------------------
--Penalty parameter
penPara = 0.1
--Penalty value
penVal = 1/penPara
--------------------------------------------
--------------------------------------------
--------------------------------------------

---------------------------------------------------------------
------- Auxiliary function calling sequential PSO scheme ------
---------------------------------------------------------------
portSeq :: [String] -> WPGparams -> Int -> Int -> (Position -> Double) -> Boundings -> IO()
portSeq names wpg np nit f bo
  = do sg <- getStdGen
       let bestPos = psoSEQ sg wpg np nit f bo
       putStr "Best value: "
       print (fst bestPos)
       putStr "Best position: "
       print (snd bestPos)
       outPutFile names (snd bestPos)
---------------------------------------------------------------
---------------------------------------------------------------

main = do
    --Getting the name of the file with asset information
    putStrLn "Enter file name including extension, eg 'assets.txt'."
    file <- getLine
    src <- readFile file
    --src <- readFile "readText.txt" -- for testing
    let triples   = map (split.words) (lines src)
    let names = extractName triples :: [String]
    let rateR = extractRate triples :: [Double]
    let expR = extractExp triples :: [Double]
    let nAssets = length rateR
    -------------------For Testing------------------------------------
    -- Getting settings for PSO
    -- Number of particles for PSO
    putStrLn "Enter the number of particles for the swarm."
    np' <- getLine
    let np = read np'
    -- Number of iteration for PSO
    putStrLn "Enter the number of iterations for the PSO to run."
    nit' <- getLine
    let nit = read nit'
    -- Getting settings for portfolio function
    -- Risk
    putStrLn "Enter level of risk (0.4-0.9), where 0.4 is least risky."
    risk' <- getLine
    let risk = read risk'
    -- Risk aversion
    putStrLn "Enter a level for risk aversion, recommended 3."
    aversion' <- getLine
    let aversion = read aversion'
    -- Required portfolio return
    putStrLn "Enter your required portfolio return, eg '0.02'."
    reqExpR' <- getLine
    let reqExpR = read reqExpR'
    ------------------------------------------------------------------
        --Return of portfolio
        portR :: Position -> Double
        portR w = sum [x*y | x <- w, y <- rateR]
        --Expected Portfolio return
        expPortR :: Position -> Double
        expPortR w = sum [x*y | x <- w, y <- expR]
        --Portfolio function
        port :: Position -> Double
        port w = risk*(max 0 ((portR w)-(expPortR w)))+(1-risk)*(((max 0 ((expPortR w)-(portR w))))**
(1/aversion))-(expPortR w)
        --Unconstrained porfolio function
        mainPortFunction :: Position -> Double
        --mainPortFunction w = (port w) + penVal*(abs((expPortR w)-reqExpR)) + penVal*(abs((sum w) -
1))
        mainPortFunction w = (port w) + penVal*((expPortR w)-reqExpR) + penVal*(abs((sum w) - 1))
        --Weight bound is set to this to induce diversification
        weightBounds = replicate nAssets (0.05,0.35)
    let pso = portSeq names wpg1 np nit mainPortFunction weightBounds
    --outPutFile names (snd pso)
    pso
    return ()
  where
    --insert (s, g1, g2) = insertWith (++) s [g1,g2]
    split [name,rateR,expR] = (name, read rateR, read expR) :: (String,Double,Double)
    extractName xs = [d | (d,_,_) <- xs] :: [String]
    extractRate xs = [d | (_,d,_) <- xs]
    extractExp  xs = [d | (_,_,d) <- xs]
    portReturn w e = sum [x*y | x <- w, y <- e]

--Used to print the results from the PSO to a file
--It either creates or changes a file.
outPutFile names pos = do
    t <- getCurrentTime
    appendFile ("output-" ++ (time t)) ((time t) ++ "\nOptimal Portfolio:\n" ++ (printStuff names pos)
++ "\n" )
    --appendFile ("output-" ++ (time t)) ("Expected Portfolio Return is: " ++ "\n\n" )
  where time t = show (toGregorian $ utctDay t)

printStuff [] [] = []
printStuff (n:ns) (p:ps) = n ++ "\t" ++ toPerc p ++ "% \n" ++ (printStuff ns ps)

-- Turns a number into a percentage
toPerc :: Double -> String
toPerc = printf "%.2f" . (*100)

---- Turns a number into a percentage
--toPerc :: Double -> Double
--toPerc x = 100*(myRound x 4)
---- Rounds a number to s decimal points
```

```haskell
--myRound n s = fromIntegral (round (n * factor)) / factor
--    where factor = fromIntegral (10^s)
-------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------

--PSO.hs file
-------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------
--------------------------------------------------------------
-- Generic scheme to deal with Particle Swarm Optimization --
-- The code contains a generic sequential Haskell function --
-- (psoSEQ) as well as three parallel implementations in   --
-- Eden (pso, psoVar, psoVar2).                            --
-- Some example functions are also included (but they are  --
-- not exported).                                          --
--                                                         --
-- Created by: P. Rabanal, I. Rodriguez, F. Rubio          --
-- Last modified: June 2012                                --
--------------------------------------------------------------

module PSO(psoSEQ,                                     -- Sequential PSO scheme
           --pso,psoVar,psoVar2,                         -- Parallel versions of PSO
           Position,Speed,Boundings,WPGparams,Particle   -- Auxiliary types
           ) where
--import Control.Parallel.Eden
import Control.DeepSeq
import Data.List
import System.Random


----------------------------
-- Basic types to be used --
----------------------------

-- Types and functions dealing with R^n positions
type Position = [Double]     -- Assuming R^n
type Speed = Position
type Boundings = [(Double,Double)]  -- (Lower,upper) for each dimension

infixl 7 *&
(*&) :: Double -> Speed -> Speed
x *& xs = map (x*) xs

infixl 6 -&
(-&) :: Speed -> Speed -> Speed
xs -& ys = zipWith (-) xs ys

infixl 6 +&
(+&) :: Speed -> Speed -> Speed
xs +& ys = zipWith (+) xs ys

-- w,p,g parameters
type WPGparams = (Double, Double, Double)

-- Particle: Best local value, best global value, current position and speed, best local position,
best global position
type Particle = (Double,Double,Position,Speed,Position,Position)


-----------------------------------
-- Generic PSO sequential scheme --
-----------------------------------

-- General sequential pso scheme
psoSEQ :: RandomGen a => a             -- Random generator
          -> WPGparams                 -- Standard adjustment parameters
          -> Int                       -- Number of particles to be used
          -> Int                       -- Maximum number of iterations
          -> (Position -> Double)      -- Fitness function
          -> Boundings                 -- Search space boundaries
          -> (Double,Position)         -- Value and position of best fitness
----With constriction factor.
```

```haskell
--psoSEQ sg (_,x2,x3) np it f bo = obtainBest (pso' rss wpg' it f bo initParticles)
--  where initParticles = initialize sg np bo f
--        rss = makeRss np (randomRs (0,1) sg)
--        wpg' = ((wpgFunc x2 x3),x2,x3)
--wpgFunc :: Double -> Double -> Double
--wpgFunc x2 x3 = 2 / (abs (2 - (x2+x3)- sqr ((x2+x3)**2 -4*(x2+x3))))
----Without constriction factor
psoSEQ sg wpg np it f bo = obtainBest (pso' rss wpg it f bo initParticles)
  where initParticles = initialize sg np bo f
        rss = makeRss np (randomRs (0,1) sg)

-- Sequential function taking care of the execution of a given number of iterations it
-- of the basic PSO algorithm. It is used both from the sequential and parallel schemes.
pso' _ _ 0 _ _ pas = pas
pso' (rs:rss) wpg it f bo pas = rnf newPas `seq` pso' rss wpg (it-1) f bo newPas
  where newPas = oneStep rs wpg f bo pas

-- Basic sequential function implementing one step of the basic PSO algorithm
oneStep :: [(Double,Double)] -> WPGparams -> (Position -> Double) -> Boundings
             -> [Particle] -> [Particle]
oneStep rs wpg f bo pas
  | null newBests = newPas
  | otherwise     = map (updateGlobalBest newBvBp) newPas
  where  newBsPas  = zipWith (updateParticle wpg f bo) rs pas
         newPas    = map snd newBsPas
         newBests  = (map snd (filter fst newBsPas))
         newBvBp   = obtainBest [minimum newBests]

updateGlobalBest (newBv,newBp) (blv,bgv,po,s,blp,bgp) = (blv,newBv,po,s,blp,newBp)

updateParticle :: WPGparams -> (Position -> Double) -> Boundings
                    -> (Double,Double) -> Particle -> (Bool,Particle)
updateParticle (w,p,g) f bo (rp,rg) (blv,bgv,po,s,blp,bgp)
  | newFit < bgv = (True,(newFit,newFit,newPosition,newSpeed,newPosition,newPosition))
  | newFit < blv = (False,(newFit,bgv,newPosition,newSpeed,newPosition,bgp))
  | otherwise    = (False,(blv,bgv,newPosition,newSpeed,blp,bgp))
  where newSpeed = limitRange (replicate (length bo) (-20000,20000)) (w*&s +& p*rp*&(blp-&po) +& g*rg*&
(bgp-&po))
        newPosition = limitRange bo (po +& newSpeed)
        newFit = f newPosition

limitRange bo xs = zipWith limit1 bo xs
limit1 (l,u) n = min (max n l) u


-- Initialization of the particles
initialize sg np bo f = map (addBest bpos) nearlyPos
  where ndim = length bo
        ps,pos :: [Position]
        ps = randomPs (ndim*np) bo' sg
        pos = take np ps
        ss :: [Speed]
        ss = map (-& (map fst bo))  (drop np ps)
        fs = map f pos
        nearlyPos = zip3 fs pos ss
        bpos = (fsnd3 . minimum) nearlyPos

        bo'= map includeAsymmetry bo
        includeAsymmetry = id  -- In case no asymmetric initialization is used
--        mitad (lowR,upR) = (upR/3,upR) -- Example of asymmetric initialization

        addBest (bv,bpo) (fv,po,s) = (fv,bv,po,s,po,bpo)

{--
----------------------------------
-- Generic PSO parallel schemes --
----------------------------------


-- General parallel pso scheme (version 1)
pso :: RandomGen a => a             -- Random generator
       -> WPGparams                 -- Standard PSO adjustment parameters
       -> Int                        -- Particles to be used
       -> Int                        -- Iterations in each parallel step
       -> Int                        -- Number of parallel iterations
       -> Int                        -- Number of parallel processes
       -> (Position -> Double)       -- Fitness function
       -> Boundings                  -- Search space boundaries
       -> (Double,Position)          -- Value and position of best fitness
pso sg wpg np pit it nPE f bo = last bests
  where initParticles = initialize sg np bo f

        pass = shuffle nPE initParticles
        sgs = tail (generateSGs (nPE+1) sg)

        pouts :: [ [(Double,Position)] ]
        pouts = [process (psoP (sgs!!i) wpg pit f bo) # (pass !!i,bests1) | i<-[0..nPE-1]]  `using`
spine

        bests :: [(Double,Position)]
        bests = map (minimum) (transp pouts)

        bests1 = take it (obtainBest initParticles : bests)


-- General parallel pso scheme (version 1)
-- Now the different speeds of different processors is taken into account
-- (speeds parameter) so that more tasks are assigned to faster processors.
psoVar :: RandomGen a => a           -- Random generator
          -> WPGparams               -- Standard PSO adjustment parameters
          -> Int                     -- Particles to be used
          -> Int                     -- Iterations in each parallel step
          -> Int                     -- Number of parallel iterations
          -> [Double]                -- Speed of processors
          -> (Position -> Double)    -- Fitness function
          -> Boundings               -- Search space boundaries
          -> (Double,Position)       -- Value and position of best fitness
psoVar sg wpg np pit it speeds f bo = last bests
  where initParticles = initialize sg np bo f
        nPE = length speeds

        pass = shuffleRelative speeds initParticles
        sgs = tail (generateSGs (nPE+1) sg)

        pouts :: [ [(Double,Position)] ]
        pouts = [process (psoP (sgs!!i) wpg pit f bo) # (pass !!i,bests1) | i<-[0..nPE-1]]  -- `using`
spine

        bests,bests1 :: [(Double,Position)]
        bests = map (minimum) (transp pouts)

        bests1 = take it (obtainBest initParticles : bests)



-- General parallel pso scheme (version 3)
-- In addition to taking care of different processors speed, now the number of
-- iterations in each parallel step can be different (an input list is provided)
psoVar2 :: RandomGen a => a           -- Random generator
           -> WPGparams               -- Standard PSO adjustment parameters
           -> Int                     -- Particles to be used
           -> [Int]                   -- Iterations in each parallel step
           -> [Double]                -- Speed of processors
           -> (Position -> Double)    -- Fitness function
           -> Boundings               -- Search space boundaries
           -> (Double,Position)       -- Value and position of best fitness
psoVar2 sg wpg np pits speeds f bo = last bests
  where initParticles = initialize sg np bo f
        nPE = length speeds
        it = length pits

        pass = shuffleRelative speeds initParticles
        sgs = tail (generateSGs (nPE+1) sg)
        pouts :: [ [(Double,Position)] ]
        pouts = [process (psoPV (sgs!!i) wpg pits f bo) # (pass !!i,bests1) | i<-[0..nPE-1]]  --
`using` spine

        bests,bests1 :: [(Double,Position)]
        bests = map (minimum) (transp pouts)

        bests1 = take it (obtainBest initParticles : bests)


-- Basic process function used by the first and second parallel schemes
psoP :: RandomGen a => a
        -> WPGparams -> Int -> (Position -> Double) -> Boundings
        -> ([Particle],[(Double,Position)]) -> [(Double,Position)]
psoP sg wpg pit f bo (pas,[]) = []
psoP sg wpg pit f bo (pas,newBest:newBests)
  = newOut : psoP sg2 wpg pit f bo (newPas,newBests)
  where rss = makeRss (length pas) (randomRs (0,1) sg1)
        (sg1,sg2)=split sg
        pas' = if newBest < oldBest
                  then map (updateGlobalBest newBest) pas
                  else pas
        newPas = pso' rss wpg pit f bo pas'
        newOut = obtainBest newPas
        oldBest = obtainBest pas

-- Basic process function used by the third parallel scheme
psoPV :: RandomGen a => a
        -> WPGparams -> [Int] -> (Position -> Double) -> Boundings
        -> ([Particle],[(Double,Position)]) -> [(Double,Position)]
psoPV sg wpg pits f bo (pas,[]) = []
psoPV sg wpg (pit:pits) f bo (pas,newBest:newBests)
  = newOut : psoPV sg2 wpg pits f bo (newPas,newBests)
  where rss = makeRss (length pas) (randomRs (0,1) sg1)
        (sg1,sg2)=split sg
        pas' = if newBest < oldBest
                  then map (updateGlobalBest newBest) pas
                  else pas
        newPas = pso' rss wpg pit f bo pas'
        newOut = obtainBest newPas
        oldBest = obtainBest pas
--}

-------------------------
-- Auxiliary functions --
-------------------------

shuffle n xs
  | null dr = take n (map (:[]) iz ++ repeat [])
  | otherwise = zipWith (:) iz (shuffle n dr)
  where (iz,dr) = splitAt n xs

spine [] = ()
spine (x:xs) = spine xs

shuffleRelative speeds tasks = splitWith normalized tasks
  where normalized = map (round.(m*).(/total)) speeds
        total = sum speeds
        m = fromIntegral (length tasks)

splitWith [n] xs = [xs]
splitWith (n:ns) xs = firsts:splitWith ns rest
  where (firsts,rest) = splitAt n xs

transp :: [[a]] -> [[a]]
transp [] = []
transp ([]: _ ) = []
transp xss = map head xss : transp (map tail xss)

obtainBest :: [Particle] -> (Double,Position)
obtainBest pas = (bv,bp)
  where (_,bv,_,_,_,bp) = head pas
```

```haskell
generateSGs 0 sg = []
generateSGs 1 sg = [sg]
generateSGs n sg = sg1:generateSGs (n-1) sg2
  where (sg1,sg2) = split sg

randomPs n bo sg = transpose (map (take n) xss')
  where xss = map (flip randomRs sg) bo
        xss' = zipWith drop [0,n..] xss

makeRss :: Int -> [a] -> [[(a,a)]]
makeRss np rs = tuple2 iz : makeRss np dr
  where (iz,dr) = splitAt (2*np) rs
        tuple2 [] = []
        tuple2 (x:y:zs) = (x,y): tuple2 zs

fsnd3 (x,y,z) = (x,y)

--------------
-- EXAMPLES --
--------------

-- Example of adjustment parameters
-- (taken from M.E.H Pedersen, Tuning & Simplifying Heuristical Optimization)
wpg1 :: (Double,Double,Double)
wpg1 = (-0.16,1.89,2.12)

-- Examples of fitness functions with corresponding boundings. taken from Yao et al
-- (Evolutionary Programming made faster, IEEE Trans. on Evolutionary Computation)

bo1 = replicate 30 (-100,100)
fit1 xs = sum (map sqr xs)

bo2 = replicate 30 (-10,10)
fit2 xs = sum xs' + (foldr (*) 1 xs')
  where xs' = map abs xs

bo3 = replicate 30 (-100,100)
fit3 xs = sum [sqr (sum ys) | ys <- tail(inits xs)]

bo4 = replicate 30 (-100,100)
fit4 xs = maximum (map abs xs)

bo5 = replicate 30 (-30,30)
fit5 xs = sum (zipWith f xs1 xs)
  where xs1 = tail xs
        f x1 x = 100*sqr (x1-sqr x)+sqr(x-1)

bo6 = replicate 30 (-100,100)
fit6 xs = fromIntegral (sum (map f xs))
  where f x = sqr(floor (x+0.5))
sqr x = x*x

bo8 = replicate 30 (-500,500)
fit8 xs = sum (map fit8' xs)
  where fit8' xi = -xi * sin (sqrt (abs xi))

bo9 = replicate 30 (-5.12,5.12)
fit9 xs = sum (map fit9' xs)
  where fit9' xi = sqr xi - 10*cos(2*pi*xi) + 10

bo10 = replicate 30 (-32,32)
fit10 xs = -20*exp(-0.2*sqrt(sum (map sqr xs)/n')) - exp(sum (map f' xs) / n') + 20 + exp 1
  where n' = fromIntegral (length xs)
        f' xi = cos (2*pi*xi)

bo11 = replicate 30 (-600,600)
fit11 xs = sum (map sqr xs) / 4000 - prod (zipWith f xs [1..]) + 1
  where f x i = cos (x/sqrt i)

prod xs = foldr (*) 1 xs
```

```
bo12 = replicate 30 (-50,50)
fit12 xs = (10*sqr(sin (pi*y1))+sum (map f yn1) + sqr(yn-1) )*pi/30 + sum (map fu xs)
   where f y = sqr (y-1)
         y1 = head ys
         yn = last ys
         yn1 = init ys
         ys = map obtainY xs
         obtainY x = 1 + (x+1)/4
         fu x = uf (x,10,100,4)
uf (x,a,k,m)
           | x > a = k * (x-a)**m
           | x < -a = k * ((-x)-a)**m
           | otherwise = 0

bo13 = replicate 30 (-50,50)
fit13 xs = 0.1*(sqr (sin (3*pi*x1)) + sum (zipWith f xs xs1) + sqr(xn-1)*(1+sqr(sin (2*pi*xn)))   ) +
sum (map fu xs)
  where xs1 = tail xs
        x1 = head xs
        xn = last xs
        fu x = uf (x,5,100,4)
        f x xx = sqr (x-1) * (1+sqr(sin(3*pi*xx)))

fit14 [x1,x2] = 1/(1/500 + sum (map f [0..24]))
  where f j = 1 / (fromIntegral j+1 + (x1 - fa1!!j)**6 + (x2 - fa2!!j)**6)
        fa1 = concat (replicate 5 [-32,-16,0,16,32])
        fa2 = concat (map (replicate 5) [-32,-16,0,16,32])
bo14 = replicate 2 (-65536,65536)

fit15 [x1,x2,x3,x4] = sum (zipWith f as bs)
  where f a b = sqr (a-(x1*(sqr b + b*x2)/(sqr b + b*x3 + x4)))
        as = [0.1957,0.1947,0.1735,0.16,0.0844,0.0627,0.0456,0.0342,0.0323,0.0235,0.0246]
        bs = map (1/) [0.25,0.5,1,2,4,6,8,10,12,14,16]
bo15 = replicate 4 (-5,5)

fit16 [x,y] = 4*x^2 - 2.1*x^4 + (x^6)/3 + x*y - 4*y^2 + 4*y^4
bo16 = [(-5,5),(-5,5)]

fit17 [x,y] = (y - (5.1*x^2/(4*pi^2)) + (5*x)/pi - 6)^2 + 10*(1-1/(8*pi))*cos x +10
bo17 = [(-5,10),(0,15)]
```