Overall

The core algorithm updates a linked grid of economic unit squares all in one. Given a valid world state, |updateWorld| will update it one step by a time difference ΔT.

```
-- updateWorld :: Float
            -- -> World
            -- -> World
updateWorld deltaT world =
```

We initialise a WorldState object from the environment parameters given, and the ΔT.

```
    let wState = WorldState (worldParams world) deltaT
```

We update the square's neighbours by a call to |makeNeighbour| - this updates their price and trade information to current.

```
        updateNeighbours square = square { sqNeighbours = map (\(neigh, c) -> (makeNeighbour . findSquare world . neighRef $ neigh, c)) . sqNeighbours $ square }
```

We run |updateSquare| on each square to get the new square list.

```
        processUpdateSquare sq = runReader (updateSquare sq) wState
        newSquares = map updateNeighbours . map (processUpdateSquare) . worldSquares $ world
```

And generate the new world state.

```
    in  world { worldSquares = newSquares }
```

Then everything else:

```
updateSquare :: Square -> Reader WorldState Square
updateSquare square = do
    let production = produceGoods (sqPop square)
    let supplies = calcSupplies production (sqTradeIn square) (sqTradeOut square)
    let tradeVolumes = calcTradeVolumes (sqTradeIn square) (sqTradeOut square)
    let localUtilities = calcLocalUtilities (sqPop square) supplies

    let inTrades = newTradeIn (sqRef square) (sqNeighbours square)

```

```haskell
    pops <- newPop (sqPrices square) (sqPop square)
    prices <- newPrices supplies tradeVolumes localUtilities (sqNeighbours square) (sqPrices square)
    outTrades <- newTradeOut supplies (sqPrices square) (sqNeighbours square) (sqTradeOut square)

    let flatOutTrades = flattenTrade (sqTradeIn square) outTrades

    return $ square { sqPop = pops, sqPrices = prices, sqTradeIn = inTrades, sqTradeOut = flatOutTrades }




-- POPULATION & PRODUCTION

newPop :: GoodPrices -> SquarePop -> Reader WorldState SquarePop
newPop prices pops =
  reader $ \wState ->
    let popMoveFactor = paramLK wState "popMoveFactor"

        fs = map (*popMoveFactor) . map (\(r, p) -> findG0 (resProduct r) prices * resDerivative r p / resRetention r) $ pops
        f_av = sum fs / genericLength fs
        forces = zip pops . map (\f -> f - f_av) $ fs
        correctedForces = map (\((r, p), f) -> if f + p < 0 then ((r, p), 0) else ((r, p), f)) forces

        supplyPops   = filter ((<=0).snd) $ correctedForces
        capacityPops = filter ((> 0).snd) $ correctedForces
        availSupply    = negate . sum . map snd $ supplyPops
        availCapacity  = sum . map snd $ capacityPops

        movement = min availSupply availCapacity
        shiftedSupply   = map (\((r, p), f) -> (r, p + movement * f / availSupply  )) supplyPops
        shiftedCapacity = map (\((r, p), f) -> (r, p + movement * f / availCapacity)) capacityPops

    in  if movement > 0 then shiftedSupply ++ shiftedCapacity
                        else pops


produceGoods :: SquarePop -> GoodQuantities
produceGoods pops =
```

```haskell
    let produceGood (r, p) = (resProduct r, resProduce r p)
    in  map produceGood pops



-- PRICES

calcSupplies :: GoodQuantities -> GoodTrades -> GoodTrades -> GoodQuantities
calcSupplies production tradeIn tradeOut =
    map (\(g, q) -> (g, q + sumTradeGood g tradeIn - sumTradeGood g tradeOut)) production

calcTradeVolumes :: GoodTrades -> GoodTrades -> GoodQuantities
calcTradeVolumes tradeIn tradeOut =
    let goods = nub $ goodList tradeIn ++ goodList tradeOut
    in  map (\g -> (g, sumTradeGood g tradeIn + sumTradeGood g tradeOut)) goods

calcLocalUtilities :: SquarePop -> GoodQuantities -> GoodUtilities
calcLocalUtilities pops goodQs =
    let totalPop = sum . map snd $ pops
        calcUtility base params = \supply -> totalPop * base * (sum . map (\(i, a) -> a * (supply / totalPop + 1) ** (-i)) . zip [0..] $ params)
    in  map (\(g, q) -> (g, calcUtility (goodUtilityBase g) (goodUtilityParams g) q)) goodQs


newPrices :: GoodQuantities -> GoodQuantities -> GoodUtilities -> SquareNeighbours ->  GoodPrices -> Reader WorldState GoodPrices
newPrices supplies tradeVs utilities neighbours prices =
  reader $ \wState ->
    let localResponse = paramLK wState "priceLocalResponseFactor"
        tradeResponse = paramLK wState "priceTradeResponseFactor"
        deltaT = stateDeltaT wState

        findNeighbourPrice good neigh = findG0 good . neighPrices $ neigh

        facA = \good -> localResponse * findG0 good supplies * findG0 good utilities
                      + tradeResponse * findG0 good tradeVs / genericLength neighbours * foldl
```

```
                          (\psum (neigh, c) -> psum + c * findNeighbourPrice good neigh) 0 neighbours

        facB = \good -> localResponse * findG0 good supplies
                      + tradeResponse * findG0 good tradeVs / genericLength neighbours * foldl (\psum (neigh, c) -> psum + c) 0 neighbours

        newPrice a b oldPrice dT = if b /= 0 then 1/b * (a + (b * oldPrice - a) * exp ((-b) * dT)) else oldPrice

    in  map (\(g, p) -> (g, newPrice (facA g) (facB g) p deltaT)) prices





-- TRADES

newTradeIn :: SquareRef -> SquareNeighbours -> GoodTrades
newTradeIn ref neighbours =
    let findSelf _ [] = []
        findSelf ref ((r,gq):xs) = if r == ref then gq else findSelf ref xs
    in  map (\(neigh, nc) -> (neighRef neigh, findSelf ref (neighTradeOut neigh))) neighbours


newTradeOut :: GoodQuantities -> GoodPrices -> SquareNeighbours -> GoodTrades -> Reader WorldState GoodTrades
newTradeOut supplies prices neighbours tradeOut =
  reader $ \wState ->
    let tradeMoveFactor = paramLK wState "tradeMoveFactor"
        deltaT = stateDeltaT wState

        idealDelta neigh c good = tradeMoveFactor * deltaT * findG0 good supplies * c * ((findG0 good . neighPrices $ neigh) - findG0 good prices)
        totalIDelta good = sum . map (\(neigh, c) -> idealDelta neigh c good) $ neighbours
        tradeScaleFactor good = let supply = findG0 good supplies; totalD = totalIDelta good in if totalD > 0 then supply / max supply totalD else 1.0
```

```

        actualDelta ref good = let c = findNConn neighbours ref
                                   neigh = findNeigh neighbours ref
                               in  idealDelta neigh c good * tradeScaleFactor good

    in  map
         (\(ref, gQs) -> (ref, map
                                 (\(g, q) -> (g, max 0 $ q + actualDelta ref g))
                                 gQs))
         tradeOut


flattenTrade :: GoodTrades -> GoodTrades -> GoodTrades
flattenTrade tradeIn tradeOut =
    let findTrade []           _        = []
        findTrade ((r,gQs):ts) neighRef = if r == neighRef then gQs else findTrade ts neighRef
        flatten inT (g, outQ)  = let inQ = findG0 g inT in (g, max (outQ - inQ) 0)

        inOutTrades = map (\(r, t) -> (r, findTrade tradeIn r, t)) tradeOut
        newTrades = map (\(r, inT, outT) -> (r, map (flatten inT) outT)) inOutTrades
    in newTrades










-- HELPERS

findGdef :: a -> Good -> [(Good, a)] -> a
findGdef def g gl = case find ((==g).fst) $ gl of Just (g, v) -> v
                                                  Nothing     -> def
findG0 = findGdef 0

```

```
sumTradeGood :: Good -> GoodTrades -> Float
sumTradeGood g trades = foldl (\sum (n, gqs) -> sum + findGO g gqs) 0 trades

goodList :: GoodTrades -> [Good]
goodList tr = nub . foldl (\acc (n, gqs) -> acc ++ map fst gqs) [] $ tr

findSquare :: World -> SquareRef -> Square
findSquare world ref = head . filter ((==ref).sqRef) . worldSquares $ world


findNTup neighbourList neighbourRef = head . filter ((==neighbourRef).neighRef.fst) $ neighbourList

findNeigh :: SquareNeighbours -> SquareRef -> Neighbour
findNeigh neighbourList = fst . findNTup neighbourList

findNConn :: SquareNeighbours -> SquareRef -> Float
findNConn neighbourList = snd . findNTup neighbourList


makeNeighbour :: Square -> Neighbour
makeNeighbour square =
    Neighbour (sqRef square) (sqPrices square) (sqTradeOut square)


paramLK :: WorldState -> String -> Float
paramLK ws n = head . map snd . filter ((==n).fst) . stateParams $ ws
```