

# Haskell program to calculate optimal real-time communication strategies for a three-step system

Aditya Mahjan

May 15, 2008

## Introduction

This program finds the best memoryless and real-time communication schemes for a three-step communication system. In order to be more efficient, parts of the code are written specifically for a three-step system rather than for a generic system. However, it should be simple to extend these ideas to larger systems. This code is written in literate Haskell; Haskell is a lazy functional language, which appeared to be a good match for the task at hand.

```
{-# OPTIONS_GHC -XUnicodeSyntax #-}
module Main (memoryless, realtime, realtime') where
import Array ((!), Array, array, inRange, Ix)
import Data.Function (on)
import Data.List (minimumBy)
import qualified Data.Map as Map (Map.fromList, Map.toList, Map.Map, Map.elems)
import Test.QuickCheck ((==>), quickCheck)
```

## Details of the communication system

Haskell, being a functional language, allows us to write code in a non-sequential manner. So, we give the details of the communication system upfront. This is the communication system.

```
initial    = getInitial    0.4
source     = getSource     0.0 0.1
channel    = getChannel    0.0 0.1
distortion = getDistortion 1.0 1.0
```

We explain what each of these terms mean one-by-one. We assume that the source is a binary time-homogeneous first-order Markov source. The variable `initial` denotes the initial pmf (probability mass function) of the source. Since the source is binary, we need only one quantity to characterize it, viz, the probability of being in state `0` in the beginning.

```
type Matrix = Array (Bit, Bit) Rational
type Vector = Array Bit Rational
getInitial :: Rational → Vector
getInitial p = array (Bit 0, Bit 1) [(Bit 0,p), (Bit 1,1-p)]
```

The variable `source` represents the matrix of transition probability of the source. It is characterized by two quantities: the crossover probabilities at states `0` and `1`.

```

getSource :: Rational -> Rational -> Matrix
getSource po p1 = array bounds list where
  bounds = ((Bit 0, Bit 0),(Bit 1, Bit 1))
  list   = [ ((Bit 0, Bit 0), 1 - po )
            , ((Bit 0, Bit 1), po     )
            , ((Bit 1, Bit 0), p1     )
            , ((Bit 1, Bit 1), 1 - p1 )]

```

The variable `channel` represents the channel matrix. It is also characterized by two quantities: the crossover probabilities for inputs `0` and `1`

```

getChannel :: Rational -> Rational -> Matrix
getChannel = getSource

```

The variable `distortion` represents the distortion function. It is characterized by two parameters: the distortion when `0` is not identified, and the distortion when `1` is not identified.

```

getDistortion :: Rational -> Rational -> Matrix
getDistortion do d1 = array bounds list where
  bounds = ((Bit 0, Bit 0),(Bit 1, Bit 1))
  list   = [ ((Bit 0, Bit 0), 0 )
            , ((Bit 0, Bit 1), do)
            , ((Bit 1, Bit 0), d1)
            , ((Bit 1, Bit 1), 0 )]

```

## Preliminaries

All the alphabets in this code are binary, so we define some commonly used lists.

```

newtype Bit = Bit Int deriving (Eq, Show, Ord, Ix)
fromBit (Bit x) = fromIntegral x
bits   = [Bit 0, Bit 1]
bits2  = [ (x,y) | x <- bits, y <- bits]
bits3  = [(x,y,z) | x <- bits, y <- bits, z <- bits]

```

## The codebook

We want a common syntax to represent codes (both the encoding strategies and the decoding strategies). We decide to enumerate codes based on their base2 representation. For example, there are 16 codes for two inputs:

AB	0	1	2	3	...	16
11	0	0	0	0	...	1
10	0	0	0	0	...	1
01	0	0	1	1	...	1
00	0	1	0	1	...	1

We define codebook that does this. First we define a data type Code and a means to display its "number"

```

type Code = Array (Bit,Bit,Bit) Bit

- instance Show Code where
-   show = showCode3

showCode3 :: Code → String
showCode3 f = show value where
  value = 128*f'(Bit 1, Bit 1, Bit 1) + 64*f'(Bit 1, Bit 1, Bit 0)
        + 32*f'(Bit 1, Bit 0, Bit 1) + 16*f'(Bit 1, Bit 0, Bit 0)
        + 8*f'(Bit 0, Bit 1, Bit 1) + 4*f'(Bit 0, Bit 1, Bit 0)
        + 2*f'(Bit 0, Bit 0, Bit 1) + f'(Bit 0, Bit 0, Bit 0)
  f' = fromBit.(f!)

```

Now, the definition of codebook. It is based on a recursive evaluation of base2 coefficients. The function codebook is called many times; in the original implementation it was taking 40% of the time. So, we decided to use a memoized version. The original version is now called codebook'

```

codebook :: Int → Code
codebook = (!) codebookArray

codebookArray :: Array Int Code
codebookArray = array bounds list where
  bounds = (0,255)
  list = do
    n ← [0..255]
    let codeFunction = codebook' n
        let code = array ( (Bit 0,Bit 0,Bit 0)
                          , (Bit 1,Bit 1,Bit 1) ) codelist where
            codelist = do
              (x,y,z) ← bits3
              let output = codeFunction [x,y,z]
                  return ((x,y,z), output)
        return (n, code)

codebook' :: Int → ([Bit] → Bit)
codebook' n [] | n < 2 = Bit n
              | otherwise = error $ "index out of bounds:" ++ show n

codebook' n list@(Bit x:xs) = codebook' (n `func` s) xs where
  s = 2^(2^(length list - 1))
  func | x == 0 = rem
       | x == 1 = div
       | otherwise = error "coefficient must be binary"

```

Lets do a quick check to see if the codebook is working properly.

```

prop_Codebook n = inRange (0,15) n ==> show n == showCode3 (codebook n)
  where types = n :: Int

```

**Code filters** We need a function to check if a code (from the 3-step codebook) measurable at time 1 and 2. The following function does that:

```
isMeasurable :: Int → Code → Bool
isMeasurable time code = and conditions where
  conditions | time == 1 = map check1 bits
             | time == 2 = map check2 bits2
             | otherwise = [True]
  check1 x = all (code!(x, Bit 0, Bit 0)==) [code!(x,y,z) | (y,z) ← bits2 ]
  check2 (x,y) = code!(x,y,Bit 0) == code!(x,y,Bit 1)
```

We also need a function to check if a code (from a 3-step codebook) is memoryless at time 1, 2 and 3.

```
isMemoryless :: Int → Code → Bool
isMemoryless time code = and conditions where
  conditions | time == 1 = map check1 bits
             | time == 2 = map check2 bits
             | time == 3 = map check3 bits
  check1 x = all (code!(x, Bit 0, Bit 0)==) [code!(x,y,z) | (y,z) ← bits2]
  check2 y = all (code!(Bit 0, y, Bit 0)==) [code!(x,y,z) | (x,z) ← bits2]
  check3 z = all (code!(Bit 0, Bit 0, z)==) [code!(x,y,z) | (x,y) ← bits2]
```

**Code families** We define two code families, `memorylessCodes` and `realtimeCodes`. Later, we will find optimal encoders and decoders from within each family.

```
allCodes :: [Code]
allCodes = map codebook [0..255]

realtimeCodes, memorylessCodes :: Array Int [Code]
realtimeCodes = array (1,3) list where
  list = do
    t ← [1,2,3]
    let codes = filter (isMeasurable t) allCodes
        return(t,codes)

memorylessCodes = array (1,3) list where
  list = do
    t ← [1,2,3]
    let codes = filter (isMemoryless t) allCodes
        return(t,codes)
```

## Joint Distributions (Information states)

```
type Pair = (Bit, Bit)
```

This function computes the joint distribution of  $(X_1, Y_1)$  for any stage 1 code.

```
distribution1 :: Code → Array Pair Rational
distribution1 code = array bounds list where
  bounds = ((Bit 0, Bit 0), (Bit 1, Bit 1))
  list = do
```

```

(x1,y1) ← bits2
let z1 = code!(x1,Bit 0,Bit 0)
- TODO: raise an error if code is not measurable at time 1
let p = initial!x1 * channel!(z1, y1)
return ((x1,y1), p)

```

The next function computes the joint distribution of  $(X_1, Y_1, X_2, Y_2)$  (represented as  $((X_1, Y_1), (X_2, Y_2))$ ) since this makes it easier to write the joint distribution for stage 3. Haskell does not define an instance of `Ix` for  $(t_1, t_2, t_3, t_4, t_5, t_6)$ . The instance of  $(t_1, t_2, t_3, t_4, t_5)$  defined by GHC uses unsafe operators which are not exported. So, we cannot define an instance for  $(t_1, \dots, t_6)$  using unsafe operators. The normal definitoin can be unoptimized. Hence, this grouping trick).

```

distribution2 :: Array Pair Rational → Code
              → Array (Pair, Pair) Rational
distribution2 dist1 code = array bounds list where
  bounds = ( ((Bit 0,Bit 0),(Bit 0,Bit 0))
            , ((Bit 1,Bit 1),(Bit 1,Bit 1)) )
  list = do
    (x1,y1) ← bits2
    (x2,y2) ← bits2
    let z2 = code!(x1,x2,Bit 0)
    - TODO: raise an error if code is not measurable at time 2
    let p = channel!(z2, y2)
          * source !(x1, x2)
          * dist1 !(x1, y1)
    return ( ((x1,y1), (x2,y2)), p)

```

Now we define a function that gives the joint probability of  $((X_1, Y_1), (X_2, Y_2), (X_3, Y_3))$ .

```

distribution3 :: Array (Pair, Pair) Rational → Code
              → Array (Pair, Pair, Pair) Rational
distribution3 dist2 code = array bounds list where
  bounds = ( ((Bit 0,Bit 0), (Bit 0,Bit 0), (Bit 0,Bit 0))
            , ((Bit 1,Bit 1), (Bit 1,Bit 1), (Bit 1,Bit 1)) )
  list = do
    (x1,y1) ← bits2
    (x2,y2) ← bits2
    (x3,y3) ← bits2
    let z3 = code!(x1,x2,x3)
    let p = channel!(z3, y3)
          * source !(x2, x3)
          * dist2 !((x1,y1), (x2,y2))
    return ( ((x1,y1),(x2,y2),(x3,y3)), p)

```

## Optimal Decoding

We define a function to find the best decoder from a given family of codes (can be `memorylessCodes` or `realtimeCodes`). The function returns the optimal performance, and the first decoder that achieves it.

TODO: At some stage, we can consider returning all optimal decoders. But first, we need to figure out how much does that hurt the complexity of the algorithm. For this purpose, we abstract out the code that finds the best from a list.

```
best :: (Ord a) [(a, b)] -> (a, b)
best = minimumBy (compare `on` fst)
```

```
decode1 :: Array Int [Code] -> Array Pair Rational
         -> (Rational, Code)
```

```
decode1 family dist = best values where
  values = do
    g1 ← family ! 1
    let errors = do
          (x1,y1) ← bits2
          let x'1 = g1!(y1, Bit 0, Bit 0)
              let err = distortion!(x1, x'1) * dist!(x1, y1)
              return err
        let cost = sum errors
            return (cost, g1)
```

Now, to find the best decoder at time 2.

```
decode2 :: Array Int [Code] -> Array (Pair, Pair) Rational
         -> (Rational, Code)
```

```
decode2 family dist = best values where
  values = do
    g2 ← family ! 2
    let errors = do
          (x1, y1) ← bits2
          (x2, y2) ← bits2
          let x'2 = g2!(y1, y2, Bit 0)
              let err = distortion!(x2, x'2) * dist!((x1,y1), (x2,y2))
              return err
        let cost = sum errors
            return (cost, g2)
```

and, the best decoder at time 3.

```
decode3 :: Array Int [Code] -> Array (Pair, Pair, Pair) Rational
         -> (Rational, Code)
```

```
decode3 family dist = best values where
  values = do
    g3 ← family ! 3
    let errors = do
          (x1, y1) ← bits2
          (x2, y2) ← bits2
          (x3, y3) ← bits2
          let x'3 = g3!(y1,y2,y3)
              let err = distortion!(x3, x'3) * dist!((x1,y1), (x2,y2), (x3,y3))
              return err
```

```

let cost = sum errors
return (cost, g3)

```

## Information states

Now we construct the reachable set of the information states for each family of codes. We store the states in a Map (Haskell way of doing hashes).

TODO: Again, we only keep one of the optimal communication strategies. We can instead use `Map.fromListWith` and combine all the strategies that lead to the same information state. Until we take care of storing all optimal decoders, doing this does not make sense.

```

state1 :: Array Int [Code]
  → Map.Map(Array Pair Rational) (Rational,(Code, Code))
state1 family = Map.fromList list where
  decode1' = decode1 family
  list = do
    c1 ← family ! 1
    let dist = distribution1 c1
        let (cost,g1) = decode1' dist
            let strategy = (cost,(c1,g1))
            return $! (dist,strategy)

```

Now, we construct all reachable states at time 2 for a given family.

```

states2 :: Array Int [Code]
  → Map.Map(Array (Pair, Pair) Rational)
  (Rational,([Code],[Code]))
states2 family = Map.fromList list where
  decode2' = decode2 family
  list = do
    c2 ← family ! 2
    (dist1, (cost1,(c1,g1))) ← Map.toList $ state1 family
    let dist2 = distribution2 dist1 c2
        let (cost2, g2) = decode2' dist2
            let strategy = (cost1 + cost2, ( [c1,c2], [g1,g2] ))
            return $! (dist2, strategy)

```

and, for states at time 3:

```

states3 :: Array Int [Code]
  → Map.Map(Array (Pair, Pair, Pair) Rational)
  (Rational,([Code],[Code]))
states3 family = Map.fromList list where
  decode3' = decode3 family
  list = do
    c3 ← family ! 3
    (dist2, (cost2,(c12,g12))) ← Map.toList $ states2 family
    let dist3 = distribution3 dist2 c3
        let (cost3, g3) = decode3' dist3

```

```

let strategy = (cost2+cost3, (c12++[c3], g12++[g3]))
return $! (dist3, strategy)

```

## Optimal Codes

```

memoryless, realtime :: (Rational, ([Code], [Code]))
memoryless = best . Map.elems . states3 $ memorylessCodes
realtime   = best . Map.elems . states3 $ realtimeCodes
realtime'  = best . Map.elems $ states3'

```

## Real-time codes revisited

The above code is clean and fast (it takes about 200 seconds to execute), but not fast enough to do a lot of case studies. So, we will try to optimize it. A look at profiler output shows that most of the time is spent on `decode3` function. This function can be optimized by making use of the structural results. The new code takes about 2 seconds to execute.

```

decode3' :: Array (Pair, Pair, Pair) Rational
          → (Rational, Code)
decode3' dist = (sum errors, array bounds decoder) where
  bounds = ((Bit 0, Bit 0, Bit 0), (Bit 1, Bit 1, Bit 1))
  (errors, decoder) = unzip result
  result = do
    (y1, y2, y3) ← bits3
    let (err, output) = optimalDecode (y1, y2, y3)
        return (err, ((y1, y2, y3), output))
  totalD (y1, y2, y3) x' = sum $ do
    (x1, x2, x3) ← bits3
    let d = distortion!(x3, x') * dist!((x1, y1), (x2, y2), (x3, y3))
        return d
  optimalDecode inputs = min (totalD inputs $ Bit 0, Bit 0)
                           (totalD inputs $ Bit 1, Bit 1)

```

Now we can redefine `states3` as follows

```

states3' :: Map.Map(Array (Pair, Pair, Pair) Rational)
          (Rational, ([Code], [Code]))
states3' = Map.fromList list where
  family = realtimeCodes
  list = do
    c3 ← family ! 3
    (dist2, (cost2, (c12, g12))) ← Map.toList $ states2 family
    let dist3 = distribution3 dist2 c3
        let (cost3, g3) = decode3' dist3
            let strategy = (cost2+cost3, (c12++[c3], g12++[g3]))
                return $! (dist3, strategy)

```

## Output

```
main :: IO()
main = do
  putStrLn "====="
  putStrLn "Communication System"
  putStrLn "====="
  putStr   showSystem
  putStrLn "====="
  putStrLn "Memoryless codes"
  putStrLn "====="
  putStr   $ showResult memoryless
  putStrLn "====="
  putStrLn "Real time codes"
  putStrLn "====="
  putStr   $ showResult realtime'
  where
    showResult (cost,(c,g)) = unlines list where
      list = [ unwords ["Performance:"
                       , show cost
                       , showR cost]
             , unwords ["Encoder:", showCodes c]
             , unwords ["Decoder:", showCodes g]]
    showSystem = unlines list where
      list = [ unwords ["Initial:", showR $ initial!Bit 0 ]
             , unwords ["Source:"
                       , showR $ source!(Bit 0,Bit 1)
                       , showR $ source!(Bit 1,Bit 0)]
             , unwords ["Channel:"
                       , showR $ channel!(Bit 0,Bit 1)
                       , showR $ channel!(Bit 1,Bit 0)]
             , unwords ["Distortion"
                       , "do =" , showR $ distortion!(Bit 0,Bit 1)
                       , "d1 =" , showR $ distortion!(Bit 1,Bit 0) ]]
    showR = show . fromRational
    showCodes codes = "[" ++ (unwords $ map showCode3 codes) ++ "]"
```

## Result

```
=====
Communication System
=====
Initial: 0.4
Source: 0.0 0.1
Channel: 0.0 0.1
Distortion do = 1.0 d1 = 1.0
=====
Memoryless codes
=====
Performance: 673%5000 0.1346
Encoder: [15 51 170]
Decoder: [15 51 170]
=====
Real time codes
=====
Performance: 141%2500 5.64e-2
Encoder: [15 63 127]
Decoder: [15 3 1]
```

## Run-time

```
single-code +RTS -ssingle-code.time
2,938,894,952 bytes allocated in the heap
 77,029,560 bytes copied during GC (scavenged)
 39,376,464 bytes copied during GC (not scavenged)
 2,437,120 bytes maximum residency (32 sample(s))

      5607 collections in generation 0 ( 0.18s)
      32 collections in generation 1 ( 0.04s)

      8 Mb total memory in use

INIT time    0.00s ( 0.00s elapsed)
MUT  time    1.89s ( 1.90s elapsed)
GC   time    0.23s ( 0.24s elapsed)
EXIT time    0.00s ( 0.00s elapsed)
Total time   2.12s ( 2.14s elapsed)

%GC time     10.8% (11.2% elapsed)

Alloc rate   1,556,520,806 bytes per MUT second

Productivity 89.2% of total user, 88.1% of total elapsed
```