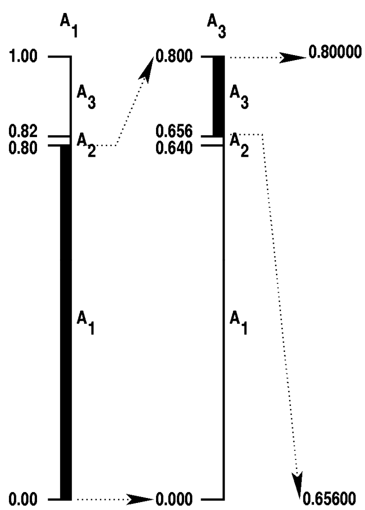
Rescaling Scheme (Encoding)

- So far, we addressed the cases when the interval is entirely confined to
  - the lower half of the unit interval, i.e., [0, 1/2), or
  - the upper half of the unit interval, i.e., [1/2, 1)
- For the last case, i.e., when the interval is containing the midpoint of the unit interval and the interval is contained in the interval [1/4, 3/4)
  - Double the interval by using the [1/4, 3/4) → [0, 1) mapping
  - After the encoder doing this map, no immediate information is stored/sent to the decoder; instead, the fact that we have used the [1/4, 3/4) → [0, 1) mapping is recorded ***at the encoder side***
  - Later on,
    - if the interval gets confined to the lower half of the unit interval, the encoder store/send to the decoder 0, followed by 1 (or more) to represent the number of application of the [1/4, 3/4) → [0, 1) mapping
    - if the interval gets confined to the upper half of the unit interval, the encoder store/send to the decoder 1, followed by 0 (or more) to represent the number of application of the [1/4, 3/4) → [0, 1) mapping

Topic 04: Codeword Encoding--Arithmetic Encoding

Rescaling Scheme (Encoding)



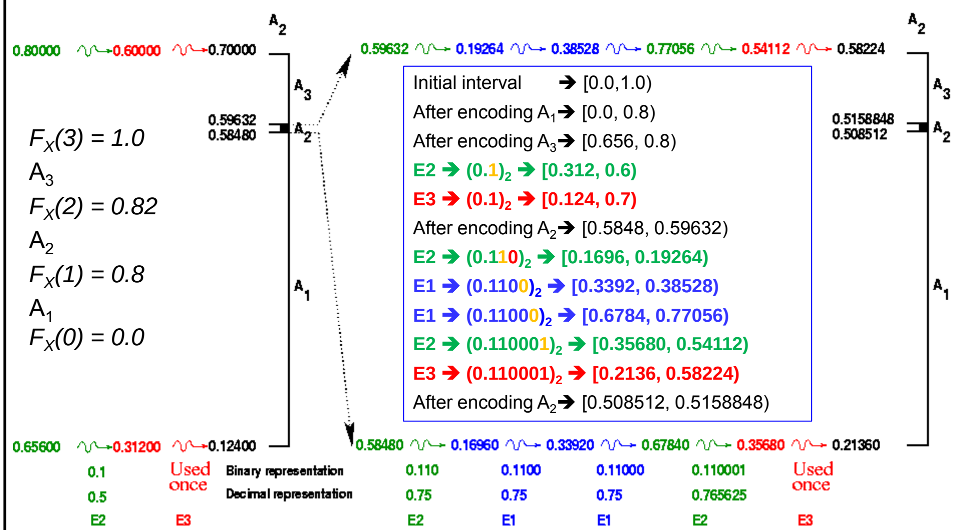
$F_X(3) = 1.0$
A_3
$F_X(2) = 0.82$
A_2
$F_X(1) = 0.8$
A_1
$F_X(0) = 0.0$

To be continued in the next pages

Initial interval ➔ [0.0,1.0)
After encoding A_1 ➔ [0.0, 0.8)
After encoding A_3 ➔ [0.656, 0.8)

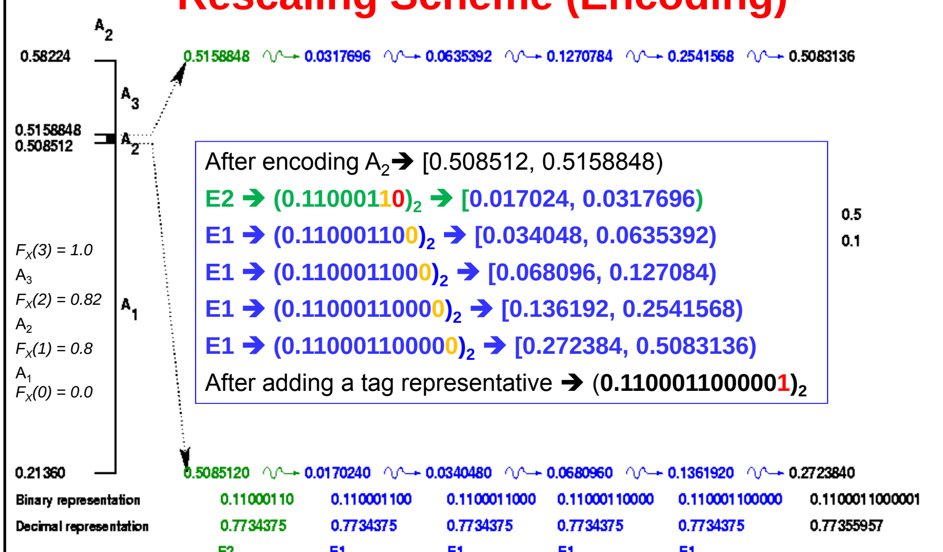
An arithmetic encoding example: the [1/4, 3/4) → [0, 1) mapping was **applied**

Rescaling Scheme (Encoding)



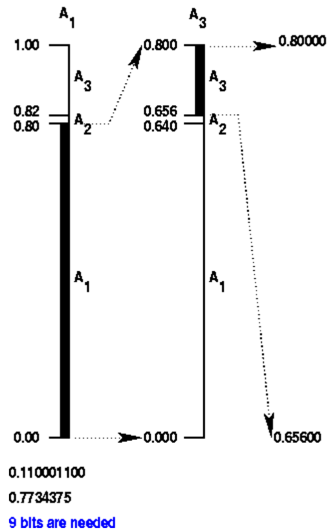
An arithmetic encoding example: the [1/4, 3/4) → [0, 1) mapping was **applied**

Rescaling Scheme (Encoding)



An arithmetic encoding example: the [1/4, 3/4) → [0, 1) mapping was **applied**

Rescaling Scheme (Decoding)



Initial interval ➔ [0.0,1.0),
Tag ➔ $(0.110001100)_2$, i.e., 0.7734375

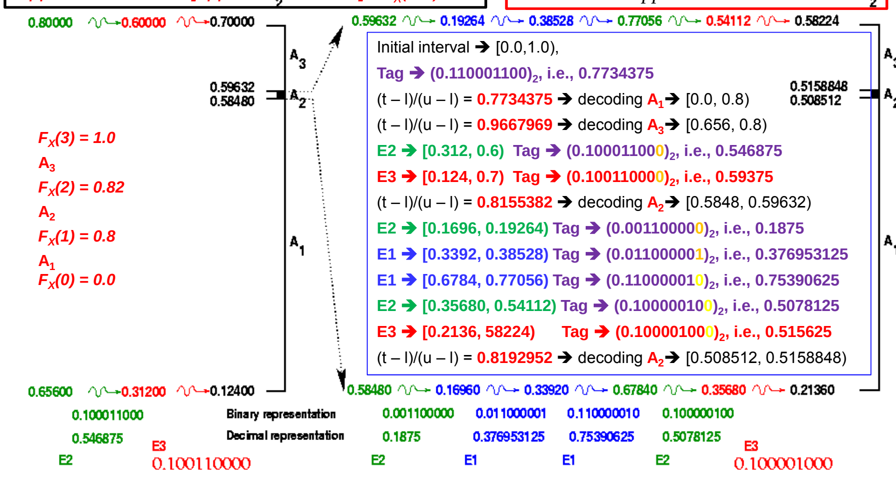
To be continued
in the next pages

An arithmetic decoding example: the encoded bit-stream is $(0.1100011000001)_2$
the [1/4, 3/4) → [0, 1) mapping was **applied**

Rescaling Scheme (Decoding)

$lower^{(k)} = lower^{(k-1)} + [upper^{(k-1)} - lower^{(k-1)}] \times F_X(A^{(k)}-1)$
$upper^{(k)} = lower^{(k-1)} + [upper^{(k-1)} - lower^{(k-1)}] \times F_X(A^{(k)})$

$$F_x(A^{(k)}-1) \le \frac{tag - lower^{(k-1)}}{upper^{(k-1)} - lower^{(k-1)}} < F_x(A^{(k)})$$



An arithmetic decoding example: the encoded bit-stream is $(0.1100011000001)_2$
the [1/4, 3/4) → [0, 1) mapping was **applied**

Rescaling Scheme

- Applying the rescaling scheme
  - Allows us to better utilize the limited precision that we possess
  - Allows us to produce the same result as if we had an unlimited precision
  - Guarantees that the interval used to encode any symbol will never be less than 0.25

Integer Implementation

- Similar to the floating point algorithm, but:
  - The initial interval length is set to 2^m, where m is defined as:

$$m = \lceil \log_2 (Total\_count) \rceil + 2$$

  - The internal *[0,1)* will be mapped to *[0, 2^m)*, i.e,
    - 0 gets mapped to $\overbrace{00000\cdots0}^{m\ times}$
    - 0.5 gets mapped to $1\overbrace{00000\cdots0}^{m-1\ times}$
    - 1 gets mapped to $\overbrace{11111\cdots1}^{m\ times}$

Integer Implementation

- Instead of updating the intervals as:
- $\text{lower}^{(n)} = \text{lower}^{(n-1)} + [\text{upper}^{(n-1)} - \text{lower}^{(n-1)}] \times F_X(A^{(n)} - 1)$
- $\text{upper}^{(n)} = \text{lower}^{(n-1)} + [\text{upper}^{(n-1)} - \text{lower}^{(n-1)}] \times F_X(A^{(n)})$

- Intervals are updated as follows:

$$new\_L = current\_L + \left\lfloor (current\_U - current\_L + 1) \times \frac{cumulative\_count(symbol - 1)}{Total\_count} \right\rfloor$$

$$new\_U = current\_L + \left\lfloor (current\_U - current\_L + 1) \times \frac{cumulative\_count(symbol)}{Total\_count} \right\rfloor - 1$$

Integer Implementation

- E1 during encoding ($L = \mathbf{0}\, x_{m-1}x_{m-2}x_{m-3}\ldots x_1$ and $U = \mathbf{0}\, y_{m-1}y_{m-2}y_{m-3}\ldots y_1$):
  - Transmit/store **zero**
  - Transmit/store number of **one**'s equal to *number_of_useage_of_E3*
  - Set *number_of_useage_of_E3* to **zero**
  - Shift out the **zero** in the MSB of the L and shift in a **zero** to its LSB
  - Shift out the **zero** in the MSB of the U and shift in a **one** to its LSB
- E2 during encoding ($L = \mathbf{1}\, x_{m-1}x_{m-2}x_{m-3}\ldots x_1$ and $U = \mathbf{1}\, y_{m-1}y_{m-2}y_{m-3}\ldots y_1$):
  - Transmit/store **one**
  - Transmit/store number of **zero**'s equal to *number_of_useage_of_E3*
  - Set *number_of_useage_of_E3* to **zero**
  - Shift out the **one** in the MSB of the L and shift in a **zero** to its LSB
  - Shift out the **one** in the MSB of the U and shift in a **one** to its LSB
- E3 during encoding ($L = \mathbf{0\ 1}\, x_{m-2}x_{m-3}\ldots x_1$ and $U = \mathbf{1\ 0}\, y_{m-2}y_{m-3}\ldots y_1$):
  - Flip the 2nd MSB of the L from **one** to **zero,**
    Shift out the **zero** in the MSB of the L and shift in a **zero** to its LSB
  - Flip the 2nd MSB of the U from **zero** to **one,**
    Shift out the **one** in the MSB of the U and shift in a **one** to its LSB
  - Increment the *number_of_useage_of_E3* by 1

Integer Implementation

- The decoded symbol is *k*, where *k* is the smallest value that satisfy the following condition:

$$while\left(\left\lfloor \frac{(tag - current\_L + 1)\times Total\_count - 1}{current\_U - current\_L + 1} \right\rfloor \geq cumulative\_count(k)\right)$$

Integer Implementation

- E1 during decoding ($L = \mathbf{0}\ x_{m-1}x_{m-2}x_{m-3}\ldots x_1$ and $U = \mathbf{0}\ y_{m-1}y_{m-2}y_{m-3}\ldots y_1$):
  - Shift out the **zero** in the MSB of the L and shift in a **zero** to its LSB
  - Shift out the **zero** in the MSB of the U and shift in a **one** to its LSB
  - *Shift out the **zero** in the MSB of the tag and read the next bit from received bit stream into the tag's LSB*
- E2 during decoding ($L = \mathbf{1}\ x_{m-1}x_{m-2}x_{m-3}\ldots x_1$ and $U = \mathbf{1}\ y_{m-1}y_{m-2}y_{m-3}\ldots y_1$):
  - Shift out the **one** in the MSB of the L and shift in a **zero** to its LSB
  - Shift out the **one** in the MSB of the U and shift in a **one** to its LSB
  - *Shift out the **one** in the MSB of the tag and read the next bit from received bit stream into the tag's LSB*
- E3 during decoding ($L = \mathbf{0\ 1}\ x_{m-2}x_{m-3}\ldots x_1$ and $U = \mathbf{1\ 0}\ y_{m-2}y_{m-3}\ldots y_1$):
  - Flip the 2^{nd} MSB of the L from **one** to **zero,**
    Shift out the **zero** in the MSB of the L and shift in a **zero** to its LSB
  - Flip the 2^{nd} MSB of the U from **zero** to **one,**
    Shift out the **one** in the MSB of the U and shift in a **one** to its LSB
  - *Flip the 2^{nd} MSB of the tag,*
    Shift out the MSB of the tag and read the next bit from received bit stream into the tag's LSB

Integer Implementation (Encoding)

- Initialize *current_L* and *current_U*
- Set *number_of_useage_of_E3* to zero
- While there is a symbol need to be encoded
  - Get a symbol
  - Calculate *new_L* and *new_U*
  - While *E1*, *E2*, or *E3* condition holds
    - If *E1* or *E2* condition holds (i.e., the MSB of *new_L* and *new_U* are the same and equal to *b)*
      - Shift *new_L* to the left by 1 bit and shift in a **zero** to its LSB
      - Shift *new_U* to the left by 1 bit and shift in a **one** to its LSB
      - Transmit/store *b*
      - While(*number_of_useage_of_E3* > 0)
        - Transmit/store complement of *b*
        - Decrement *number_of_useage_of_E3*
    - If *E3* condition holds
      - Shift *new_L* to the left by 1 bit and shift in a **zero** to its LSB
      - Shift *new_U* to the left by 1 bit and shift in a **one** to its LSB
      - Complement the new MSB of *new_L* and *new_U*
      - Increment *number_of_useage_of_E3*
  - Set *current_L to new_L*
  - Set *current_U to new_U*
- Transmit/store the MSB of *current_L* (assume that it is equal to *c*)
- Shift *current_L* to the left by 1 bit and shift in a **zero** to its LSB
- While(*number_of_useage_of_E3* > 0)
  - Transmit/store complement of *c*
  - Decrement *number_of_useage_of_E3*
- Transmit/store the *current_L*

Integer Implementation (Decoding)

- Initialize *current_L* and *current_U*
- Read the first *m* bits of the received bitstream into tag *t*
- While there is a symbol need to be decoded
  - Decode a symbol
  - Calculate *new_L* and *new_U*
  - While *E1*, *E2*, or *E3* condition holds
    - If *E1* or *E2* condition holds (i.e., the MSB of *new_L* and *new_U* are the same and equal to *b)*
      - Shift *new_L* to the left by 1 bit and shift in a **zero** to its LSB
      - Shift *new_U* to the left by 1 bit and shift in a **one** to its LSB
      - Shift tag *t* to the left by 1 bit and read next bit from received bit stream into the LSB of tag *t*
    - If *E3* condition holds
      - Shift *new_L* to the left by 1 bit and shift in a **zero** to its LSB
      - Shift *new_U* to the left by 1 bit and shift in a **one** to its LSB
      - Shift tag *t* to the left by 1 bit and read next bit from received bit stream into the LSB of tag *t*
      - Complement the new MSB of *new_L*, *new_U*, and tag *t*
  - Set *current_L to new_L*
  - Set *current_U to new_U*

Arithmetic Encoding vs Huffman Encoding

- Arithmetic encoding is especially useful when dealing with:
  - small alphabets, such as binary, and
  - alphabets with highly skewed probabilities
- In arithmetic encoding, there is no need to build the entire codebook to encode a message
- It is much easier to make the arithmetic encoder becomes adaptive to changed input statistics than that in Huffman encoder
  - There is no need to generate/preserve a tree

  Consequently, it is easy to separate the modeling procedure from the encoding procedure