Exercise 1: DC Prediction of Blocks

- Update the block prediction function (first two parameters are new)

```cpp
Block BlockCoder::getPrediction( int cIdx, const Pos& pos, int log2BlkSize );
```

This function shall create a \( (2^{\log_2\text{BlkSize}}) \times (2^{\log_2\text{BlkSize}}) \) block with all samples being equal to the average (rounded to an integer) of the causal neighbourhood. The parameter cIdx specifies the color plane index (0 for Y, 1 for Cb, and 2 for Cr) and pos specifies the position of the top-left sample of the current block.

- The value of the prediction samples should be set equal to the average of
  - the already reconstructed samples directly above the current block and
  - the already reconstructed samples directly left to the current block.
- Use this new function in the encoding and decoding of blocks
- Measure whether the compression efficiency gets better or not

**Note:**
- The class BlockCoder has a pointer to the reconstructed picture.
- Each reconstructed block is written to the reconstructed picture directly after it was coded. Hence, the above and left samples are always available.
- We need some special handling at the top and left image border.
Exercise 2: Quantization (Part 1)

- Implement a class Quantizer

```cpp
class Quantizer {
public:
    Quantizer();
    ~Quantizer();

    void quantize ( Block& blk, int QP ) const; // quantization
    void deQuantize( Block& blk, int QP ) const; // reconstruction

private:
    ...
};
```

- The function `quantize()` shall quantize all samples of the block. As a result all samples should be replaced with the resulting quantization indexes.

- The function `deQuantize()` shall reconstructed all samples of the block. The input block consists of quantization indexes. As a result, the block shall contain the reconstructed samples.

- The quantization and reconstruction are controlled by the quantization parameter QP as will be further explained on the next slide(s).
Exercise 3: Quantization (Part 2)

- The quantization step size $\Delta$ shall be given by the quantization parameter QP according to
  \[
  \Delta \approx 2^{QP/4}
  \]
  The quantization parameter can range from 0 to 31, inclusive. Hence, we can use quantization step sizes from $\Delta = 1$ to $\Delta \approx 215.3$.

- For quantization, the quantization indexes $q$ shall be obtained by simple rounding
  \[
  q = \text{round} \left( \frac{s}{\Delta} \right),
  \]
  where $s$ represents the input sample.

- The reconstructed samples are obtained according to
  \[
  s' = q \cdot \Delta
  \]

- The quantization step size can be a non-integer value. In our implementation, we want to approximate the quantization and reconstruction operations using integer arithmetic as will be discussed on the next slide.
Exercise 3: Quantization (Part 3)

- Approximate the reconstruction \( s' = q \cdot \Delta \) by the integer operation

\[
 s' = ( \text{dScale}[\text{QP}] \cdot q + 2^3 ) \gg 4
\]

- Determine the array \( \text{dScale}[32] \) in the constructor \text{Quantizer::Quantizer()}.
- Use the function \text{Block::scale()} for the actual reconstruction operation.

- Approximate the quantization \( q = \text{round} \left( \frac{s}{\Delta} \right) \) by the integer operation

\[
 q = ( \text{qScale}[\text{QP}] \cdot s + 2^{15} ) \gg 16
\]

- Determine the array \( \text{qScale}[32] \) in the constructor \text{Quantizer::Quantizer()} in such a way that it best matches the other array \( \text{dScale}[32] \).
- Use the function \text{Block::scale()} for the actual quantization operation.

- Add an instance of \text{Quantizer} to the class \text{BlockCoder} and use it for
  - \text{In compressBlock():} For quantizing prediction error samples (i.e., after prediction but before the entropy coding).
  - \text{In reconstructBlock():} For reconstruction prediction error samples (i.e., after entropy decoding but before adding the prediction samples).
Exercise 4: Test Framework + Testing of Quantizer

Now, since we implemented a quantizer, we are able to trade-off bit rate and reconstruction quality. For evaluating the performance of our codec (now and in future), we will vary the quantization parameter from 0 to 31 and measure the reconstruction quality (PSNR) and bit rate for all QP points.

The updated framework includes a sub-folder simus with two scripts:

- runCodec.sh: This script can be used for running our codec with all QPs between 0 and 31, and storing the bit rates and PSNR values in a data file.
- createPlots.sh: This script can be used for creating plots for comparing the coding efficiency of two or more versions of our codec (and JPEG).

Tasks:

- Install gnuplot, pdfjoin, and image magick (if not done already)
- Make yourself familiar with the provided scripts.
- Run the script runCodec.sh for JPEG (and all test images).
- Run the script runCodec.sh for our codec (and all test images).
- Compare the coding efficiency of our coded with that of JPEG using the plotting script createPlots.sh.
Exercise 5: Next Steps for Improving our Codec

- How good is our codec in comparison to JPEG? (look at the plots)

- Think about the following:
  - What are the difference between our codec and JPEG?
  - How can we further improve the coding efficiency of our codec?
  - What should we implement next?