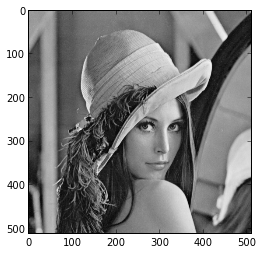
Даже небольшие изображения требуют много памяти для хранения. Так, если мы кодируем яркость каждого пикселя одним байтом, то изображение одного кадра формата FullHD (1920×1080) займёт почти два мегабайта. Благодаря активной работе учёных и программистов в настоящее время степень сжатия данных вплотную подошла к теоретическому пределу, который не так уж и велик. Основная причина – в изображениях из реального мира значения яркости редко бывают одинаковыми даже у соседних пикселей. Однако алгоритмы сжатия очень любят, когда в данных есть закономерность (лучше всего – длинные последовательности нулей). Таким образом, стоит задача преобразовать изображение так, чтобы оно хорошо сжималось классическими алгоритмами (ценой потери незначительных деталей).

Рассмотрим фрагмент первой строки яркостей из известного изображения «Lenna»



154, 155, 156, 157, 157, 157, 158, 156

Видно, что соседние числа очень близки. Чтобы получить желаемые нули или хотя бы что-то близкое к ним, можно закодировать отдельно первое число, а потом рассматривать лишь отличия каждого числа от предыдущего.

154, 1, 1, 1, 0, 0, 1, -2

Такой метод в самом деле используется и называется дельта-кодированием. Но у него есть серьёзные недостаток — он нелокальный. То есть нельзя взять кусочек последовательности и узнать, какие именно яркости в нём закодированы без декодирования всех значений перед этим кусочком.

Попробуем поступить иначе. Не будем пытаться сразу получить хорошую последовательность, попробуем улучшить её хотя бы немного.  
  
Для этого разобьём все числа на пары и найдём полусуммы и полуразности значений в каждой из них.

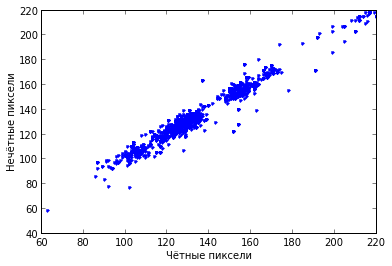
(154, 155), (156, 157), (157, 157), (158, 156)

(154.5, 0.5), (156.5, 0.5), (157, 0.0), (157, -1.0)

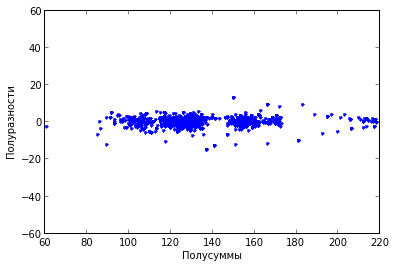
Очевидно, зная полусумму a и полуразность d можно найти и сами значения:  
первое значение в паре = a — d,  
второе значение в паре = a + d.

Это преобразование было предложено в 1909 году Альфредом Хааром и носит его имя.

Для полусумм получаем распределение



Для полуразностей:



Видно, что полуразности находятся в гораздо более узком диапазоне значений. А это значит, что на них можно потратить меньше одного байта, что упрощает сжатие. А ведь можно и обнулить часть значений без особой потери качества

Математически преобразование Хаара описывается матрицей (c учетом поправки на искажения при повороте и растяжении)

https://habrastorage.org/storage2/273/e45/790/273e45790c5fa83a4d75835da559c08f.png

На которую необходимо умножить вектор пикселей (2x1 в данном случае)

https://habrastorage.org/storage2/d89/5df/577/d895df5778fa86e8b8a73a1735fbdb0d.png

Так как матрица ортогональна, обратную можно получить простым транспонированием.

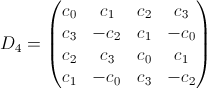
https://habrastorage.org/storage2/eb9/308/2bd/eb93082bd00b72e12be6510373598424.png

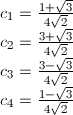
Таким образом, преобразование Хаара — это пара фильтров, разделяющих сигнал на низкочастотную и высокочастотную составляющие. Чтобы получить исходный сигнал, нужно просто снова объединить эти составляющие.

Что нам это даёт? Пусть у нас есть фотография-портрет. Низкочастотная составляющая несёт в себе информацию об общей форме лица, о плавных перепадах яркости. Высокочастотная — это шум и мелкие детали.

Обычно, когда мы смотрим на портрет, нас больше интересует низкочастотная составляющая, а значит при сжатии часть высокочастотных данных можно отбросить. Тем более, что, как мы выяснили, она обычно имеет меньшие значения, а значит более компактно кодируется.

Для уменьшения искажений используется преобразование Добеши (пример – вейвлет D4)





А также вейлеты больших порядков (коэффициенты можно узнать из специальных таблиц)

Допустим, мы выбрали для программирования вейвлет-преобразование D4. Чтобы каждый раз не проводить вычисления, поместим коэффициенты в список.

private double[] CL = {

(1 + Math.Sqrt(3)) / (4 \* Math.Sqrt(2)),

(3 + Math.Sqrt(3)) / (4 \* Math.Sqrt(2)),

(3 - Math.Sqrt(3)) / (4 \* Math.Sqrt(2)),

(1 - Math.Sqrt(3)) / (4 \* Math.Sqrt(2))

};

Буква L означает, что это коэффициенты первой строки, относящейся к фильтру низких (L, low) частот.

Коэффициенты высокочастотного фильтра (HPF, high-pass filter) будем находить при помощи отдельной функции, записывающей список в обратном порядке и чередующей знаки.

private void HpfCoeffs()

{

int N = CL.Length;

CH = new double[N];

for (int i = 0; i < N; i++)

{

CH[i] = Math.Pow(-1, i) \* CL[N - i - 1];

}

}

Теперь, когда у нас есть коэффициенты, мы можем выполнить само преобразование. Его результатом будет список, содержащий взвешенные суммы с коэффициентами поочерёдно из списков CL и CH.

Взвешенные суммы (суммы попарных произведений) — это произведения строк матрицы на вектор-столбец. Чётные строки матрицы — это низкочастотый фильтр, а нечётные — высокочастотный. Вот откуда чередование.  
  
Списки из взвешенных суммы, вычисляемых «вдоль» другого списка в математике называются свёртками. Есть эффективные алгоритмы вычисления этих свёрток на основе преобразования Фурье. (Внезапно, правда? Преобразование Фурье — это синусы и косинусы, и вдруг для суммирования используется!) Но мы сторонники простоты и не будем заниматься преждевременными оптимизациями. Посчитаем в лоб, так понятнее. А оптимизации и красивые хаки оставим читателю как упражнение.  
  
Функцию назовём pconv. P — от слова pair (пара), а conv — convolution (свёртка).

private double[] Pconv(double[] data, int delta = 0)

{

double[] result = new double[data.Length];

int N = CL.Length;

int M = data.Length;

int iResult = 0;

for (int k = 0; k < M; k += 2)

{

double sL = 0;

double sH = 0;

for (int i = 0; i < N; i++)

{

sL += data[(k + i - delta) % M] \* CL[i];

sH += data[(k + i - delta) % M] \* CH[i];

}

result[iResult++] = sL;

result[iResult++] = sH;

}

return result;

}

Для изображения



Получаем результат



Для перемешанных значений полусумм и полуразностей (в случае с D2) эффективного сжатия добиться не получится. Впрочем, что мешает переупорядочить изображение?

private void Beautify()

{

int dim = inImage.Width;

double[,] data = new double[dim, dim];

int N = dim \* dim / 4;

int temp\_i = 0;

double[] temp\_pixels = new double[N];

temp\_i = 0;

temp\_pixels = new double[N];

for (int i = 0; i < dim; i += 2)

{

for (int j = 0; j < dim; j += 2)

{

temp\_pixels[temp\_i++] = outMatrix[i, j];

}

}

temp\_i = 0;

for (int i = 0; i < dim / 2; i += 1)

{

for (int j = 0; j < dim / 2; j += 1)

{

data[i, j] = temp\_pixels[temp\_i++];

}

}

temp\_i = 0;

temp\_pixels = new double[N];

for (int i = 1; i < dim; i += 2)

{

for (int j = 0; j < dim; j += 2)

{

temp\_pixels[temp\_i++] = outMatrix[i, j];

}

}

temp\_i = 0;

for (int i = dim / 2; i < dim; i += 1)

{

for (int j = 0; j < dim / 2; j += 1)

{

data[i, j] = temp\_pixels[temp\_i++];

}

}

temp\_i = 0;

temp\_pixels = new double[N];

for (int i = 0; i < dim; i += 2)

{

for (int j = 1; j < dim; j += 2)

{

temp\_pixels[temp\_i++] = outMatrix[i, j];

}

}

temp\_i = 0;

for (int i = 0; i < dim / 2; i += 1)

{

for (int j = dim / 2; j < dim; j += 1)

{

data[i, j] = temp\_pixels[temp\_i++];

}

}

temp\_i = 0;

temp\_pixels = new double[N];

for (int i = 1; i < dim; i += 2)

{

for (int j = 1; j < dim; j += 2)

{

temp\_pixels[temp\_i++] = outMatrix[i, j];

}

}

temp\_i = 0;

for (int i = dim / 2; i < dim; i += 1)

{

for (int j = dim / 2; j < dim; j += 1)

{

data[i, j] = temp\_pixels[temp\_i++];

}

}

Array.Copy(data, beautifiedMatrix, inImage.Width \* inImage.Height);

}



Уменьшенное изображение в углу и небольшие по модулю (о чём говорит чёрный цвет) коэффициенты в остальной части. Видно, что части, соответствующей уменьшенной копии, диапазон изменения высокий, в то время как высокочастотные коэффициенты близки к нулю.

Выполним обратное преобразование

private void ICoeffs()

{

int N = CL.Length;

iCH = new double[N];

iCL = new double[N];

int i = 0;

for (int k = 0; k < CL.Length; k += 2)

{

int index\_1 = k - 2;

if (index\_1 < 0)

{

index\_1 = CL.Length - k - 2;

}

int index\_2 = k - 1;

if (index\_2 < 0)

{

index\_2 = CL.Length - k - 1;

}

iCL[i] = CL[index\_1];

iCH[i] = CL[index\_2];

i++;

iCL[i] = CH[index\_1];

iCH[i] = CH[index\_2];

i++;

}

}

private void CalculateIDWT()

{

double[,] matrixT = new double[dim, dim];

Array.Copy(matrix, matrixT, dim \* dim);

// Columns

for (int i = 0; i < dim; i++)

{

double[] column = new double[dim];

for (int k = 0; k < dim; k++)

{

column[k] = matrixT[k, i];

}

double[] pconvColumn = new double[dim];

pconvColumn = Pconv(column, iCL.Length - 2);

for (int k = 0; k < dim; k++)

{

matrixT[k, i] = pconvColumn[k];

}

}

// Rows

for (int i = 0; i < dim; i++)

{

double[] row = new double[dim];

for (int k = 0; k < dim; k++)

{

row[k] = matrixT[i, k];

}

double[] pconvRow = new double[dim];

pconvRow = Pconv(row, iCL.Length - 2);

for (int k = 0; k < dim; k++)

{

matrixT[i, k] = pconvRow[k];

}

}

Array.Copy(matrixT, outMatrix, dim \* dim);

}

А теперь давайте проделаем то, ради чего, всё это задумывалось — получим много нулей!  
  
Это можно делать разными способами. Например, можно округлять значения до необходимого количества значащих цифр (частный случай квантования). Но мы поступим проще — заменим коэффициенты, меньшие по модулю некоторого значения, на ноль.

Количество нулей для картинки 512x512 пикселей получилось 224000. Выходит, что мы просто отбросили **85,4%** имеющихся коэффициентов!!!

|  |  |
| --- | --- |
| H:\11 sem\Wavelet\Wavelet\bin\Debug\an_Output.jpg | H:\11 sem\Wavelet\Wavelet\bin\Debug\an_Output.jpg |

Изображения при этом практически неотличимы.

Пример для большей степени сжатия (пороговое значение = 0.5)

|  |  |
| --- | --- |
| H:\11 sem\Wavelet\Wavelet\bin\Debug\an_Output.jpg | H:\11 sem\Wavelet\Wavelet\bin\Debug\an_Output.jpg |