Глава 12

Анализ изображений

Вступление

В предыдущей главе мы рассматривали различные преобразования изображений в OpenCV. Все они преобразовывали входное изображение в выходное, так что результатом была по-прежнему некоторая картинка. В этой главе речь пойдет об операциях, которые полностью меняют представление изображения.

Обычно новое представление также является массивом, но его значения совсем не обязательно описывают яркость пикселей. Так, первая рассматриваемая функция – *дискретное преобразование Фурье* – строит массив, являющийся частотным представлением входного изображения. А бывает так, что результатом преобразования является список компонентов, а вовсе не массив, как, например, в случае *преобразования Хафа*.

Наконец, мы изучим методы сегментации изображений, позволяющие представить изображение в виде набора связных областей.

Дискретное преобразование Фурье

Для любого множества значений, индексированных дискретным (целочисленным) параметром, можно определить *дискретное преобразование Фурье* (ДПФ)[1], аналогичное преобразованию Фурье непрерывной функции. Если x_0, x_1, x_2, ..., x_{N-1} – комплексные числа, то одномерное ДПФ определяется формулой (где $i = \sqrt{-1}$):

$$g_k = \sum_{n=0}^{N-1} f_n e^{-\frac{2\pi i}{N} kn}.$$

Аналогичное преобразование можно определить для двумерного массива чисел (и, разумеется, для более высоких размерностей тоже):

$$g_{k_x, k_y} = \sum_{n_x=0}^{N_x-1} \sum_{n_y=0}^{N_y-1} f_{n_x, n_y} e^{-\frac{2\pi i}{N}(k_x n_x + k_y n_y)}.$$

---

[1] Жозеф Фурье [Fourier] первым открыл, что некоторые функции можно разложить в бесконечный ряд других функций и тем положил начало дисциплине, которая сейчас называется анализом Фурье. К числу основных учебником по разложению в ряд Фурье относится книга Морзе [Morse53] по применениям в физике и Папулиса [Papoulis62]. Быстрое преобразование Фурье открыли Кули и Тюки в 1965 году [Cooley65], хотя Карл Гаусс наметил основные шаги еще в 1805 году [Johnson84]. Применение в компьютерном зрении впервые описано в работе Балларда и Брауна [Ballard82].

Казалось бы, что в общем случае, для вычисления N членов g_k необходимо $O(N^2)$ операций. Однако существует несколько алгоритмов *быстрого преобразования Фурье* (БПФ), способных вычислить их за время $O(N \log N)$.

cv::dft(): дискретное преобразование Фурье

Один такой алгоритм реализует функция `cv::dft()`, умеющая вычислять БПФ для одномерных и двумерных массивов. В последнем случае можно вычислить либо двумерное преобразование, либо одномерные преобразования каждой строки (это гораздо быстрее, чем вызывать `cv::dft()` несколько раз):

```
void cv::dft(
  cv::InputArray  src,                // входной массив (вещественный или комплексный)
  cv::OutputArray dst,                // выходной массив
  int             flags = 0,          // обращение и другие флаги
  int             nonzeroRows = 0     // сколько строк обрабатывать
);
```

Входной массив может быть одно- или двухканальным и должен иметь тип с плавающей точкой. Если канал один, то предполагается, что элементы – вещественные числа, а результат будет представлен в компактном формате *симметричной комплексно-сопряженной матрицы* (CCS)[1]. Если канала два, то они интерпретируются как вещественная и мнимая части входных данных. В таком случае результаты никак не упаковываются и часть памяти расходуется впустую под хранение нулей как во входном, так и в выходном массиве[2].

Ниже показано, как упаковываются результаты в симметричной комплексно-сопряженной матрице.

Для одномерного массива:

$$\left[\begin{array}{cccccccccc} \mathrm{Re}\,Y_0 & \mathrm{Re}\,Y_1 & \mathrm{Im}\,Y_1 & \mathrm{Re}\,Y_2 & \mathrm{Im}\,Y_2 & \ldots & \mathrm{Re}\,Y_{\left(\frac{N}{2}-1\right)} & \mathrm{Im}\,Y_{\left(\frac{N}{2}-1\right)} & \mathrm{Re}\,Y_{\left(\frac{N}{2}\right)} \end{array}\right].$$

---

[1] В таком представлении размер выходного массива для одноканального изображения совпадает с размером входного, потому что элементы, заведомо равные нулю, опущены. В случае двухканального (комплексного) массива размеры входного и выходного массивов, конечно, тоже совпадают.

[2] Используя двухканальное представление, не забудьте явно присвоить мнимым частям значение 0. Проще всего это сделать, создав нулевую матрицу мнимых частей функцией `cv::Mat::zeros()`, а затем объединив ее функцией `cv::merge()` с матрицей вещественных частей для получения временного массива, к которому применяется `cv::dft()` (быть может, на месте). В результате получится неупакованная комплексная матрица спектра.

Для двумерного массива:

$$\begin{bmatrix}
\mathrm{Re}\,Y_{00} & \mathrm{Re}\,Y_{01} & \mathrm{Im}\,Y_{01} & \mathrm{Re}\,Y_{02} & \mathrm{Im}\,Y_{02} & \dots & \mathrm{Re}\,Y_{0,\frac{N_x}{2}-1} & \mathrm{Im}\,Y_{0,\frac{N_x}{2}-1} & \mathrm{Re}\,Y_{0,\frac{N_x}{2}} \\
\mathrm{Re}\,Y_{10} & \mathrm{Re}\,Y_{11} & \mathrm{Im}\,Y_{11} & \mathrm{Re}\,Y_{12} & \mathrm{Im}\,Y_{12} & \dots & \mathrm{Re}\,Y_{1,\frac{N_x}{2}-1} & \mathrm{Im}\,Y_{1,\frac{N_x}{2}-1} & \mathrm{Re}\,Y_{1,\frac{N_x}{2}} \\
\mathrm{Re}\,Y_{20} & \mathrm{Re}\,Y_{21} & \mathrm{Im}\,Y_{21} & \mathrm{Re}\,Y_{22} & \mathrm{Im}\,Y_{22} & \dots & \mathrm{Re}\,Y_{2,\frac{N_x}{2}-1} & \mathrm{Im}\,Y_{2,\frac{N_x}{2}-1} & \mathrm{Im}\,Y_{2,\frac{N_x}{2}} \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
\mathrm{Re}\,Y_{\frac{N_y}{2}-1,0} & \mathrm{Re}\,Y_{N_y-3,1} & \mathrm{Im}\,Y_{N_y-3,1} & \mathrm{Re}\,Y_{N_y-3,2} & \mathrm{Im}\,Y_{N_y-3,2} & \dots & \mathrm{Re}\,Y_{N_y-3,\frac{N_x}{2}-1} & \mathrm{Im}\,Y_{N_y-3,\frac{N_x}{2}-1} & \mathrm{Re}\,Y_{N_y-3,\frac{N_x}{2}} \\
\mathrm{Im}\,Y_{\frac{N_y}{2}-1,0} & \mathrm{Re}\,Y_{N_y-2,1} & \mathrm{Im}\,Y_{N_y-2,1} & \mathrm{Re}\,Y_{N_y-2,2} & \mathrm{Im}\,Y_{N_y-2,2} & \dots & \mathrm{Re}\,Y_{N_y-2,\frac{N_x}{2}-1} & \mathrm{Im}\,Y_{N_y-2,\frac{N_x}{2}-1} & \mathrm{Im}\,Y_{N_y-2,\frac{N_x}{2}} \\
\mathrm{Re}\,Y_{\frac{N_y}{2},0} & \mathrm{Re}\,Y_{N_y-1,1} & \mathrm{Im}\,Y_{N_y-1,1} & \mathrm{Re}\,Y_{N_y-1,2} & \mathrm{Im}\,Y_{N_y-1,2} & \dots & \mathrm{Re}\,Y_{N_y-1,\frac{N_x}{2}-1} & \mathrm{Im}\,Y_{N_y-1,\frac{N_x}{2}-1} & \mathrm{Re}\,Y_{N_y-1,\frac{N_x}{2}}
\end{bmatrix}$$

Присмотритесь внимательнее к индексам этих массивов. Некоторые значения в массиве гарантированно равны `0` (точнее, некоторые значения f_k гарантированно вещественные). Заметим также, что последняя строка матрицы присутствует, только если N_y четно, и что последний столбец присутствует, только если N_x четно. Если двумерный массив рассматривается как N_y отдельных одномерных массивов, а не как входные данные для полного двумерного преобразования, то все строки результата будут аналогичны строке, показанной выше для одномерного массива.

Третий аргумент `flags` говорит, какую операцию следует выполнить. Это битовая маска, так что флаги можно объединить с помощью оператора OR. Преобразование, с которого мы начали, называется прямым преобразованием Фурье и выбирается по умолчанию. Обратное преобразование[1] отличается от прямого только знаком экспоненты и масштабным коэффициентом. Чтобы выполнить обратное преобразование без масштабного коэффициента, задайте флаг `cv::DFT_INVERSE`. Чтобы добавить масштабный коэффициент, задайте флаг `cv::DFT_SCALE`, тогда результат будет умножен на N^{-1} (или $(N_xN_y)^{-1}$ в случае двумерного преобразования). Масштабирование необходимо, если мы последовательно применяем сначала прямое, затем обратное преобразование и хотим вернуться в исходную точку. Поскольку комбинация флагов `cv::DFT_INVERSE` и `cv::DFT_SCALE` встречается очень часто, для нее есть сокращенные обозначения: `cv::DFT_INV_SCALE` или `cv::DFT_INVERSE_SCALE`, если вы не фанатик лаконичности. Последний флаг, `cv::DFT_ROWS`, говорит `cv::dft()`, что двумерный массив следует рассматривать как набор N_y одномерных векторов длины N_x и преобразовывать их по отдельности. Такое одновременное выполнение значительно быстрее последовательного. С помощью флага `cv::DFT_ROWS` можно также реализовать ДПФ в трех и более измерениях.

---

[1] Для обратного преобразования входной массив упаковывается, как описано выше. Это разумно, потому что если мы сначала выполнили прямое преобразование, а потом обратное, то ожидаем получить исходные данные – конечно, если не забудем про флаг `cv::DFT_SCALE`!

По умолчанию прямое преобразование порождает результат в формате CCS (т. е. размеры входного и выходного массивов совпадают), но можно попросить OpenCV не делать это, задав флаг `cv::DFT_COMPLEX_OUTPUT`. Тогда получится полный комплексный массив (со всеми нулями). Наоборот, когда к комплексному массиву применяется обратное преобразование, результат обычно также будет комплексным. Но если входной массив обладает комплексно-сопряженной симметрией[1], то можно попросить OpenCV породить вещественный массив (который будет меньше входного), передав флаг `cv::DFT_REAL_OUTPUT`.

Чтобы понять смысл аргумента `nonzero_rows`, мы должны немного отвлечься и объяснить, что в алгоритмах ДПФ некоторые длины входных векторов оказываются предпочтительнее, как и некоторые размеры массивов. В большинстве алгоритмов оптимально, когда размер равен степени двойки. В алгоритме, используемом в OpenCV, наилучшими являются длины векторов и размерности массивов вида $2^p3^q5^r$ для некоторых целых p, q, r. Поэтому обычно создается массив большего размера, заполненный нулями, а потом в него копируется наш массив. Для удобства существует служебная функция `cv::getOptimalDFTSize()`, которая принимает длину вектора и возвращает наименьшее число вышеупомянутого вида, большее или равное этой длине. Несмотря на такое дополнение, функции `cv::dft()` можно сказать, что обрабатывать добавленные строки не нужно (а в случае обратного преобразования указать, какие строки результата нас не интересуют). Так или иначе, аргумент `nonzero_rows` показывает, сколько строк содержат значимые данные. Это немного сокращает время вычислений.

cv::idft(): обратное дискретное преобразование Фурье

Выше мы видели, что функция `cv::dft()` может выполнять не только дискретное преобразование Фурье, но и обратную операцию (при соответствующем значении аргумента `flags`). Часто предпочтительнее, хотя бы из сообщений понятности кода, иметь отдельную функцию, которая по умолчанию выполняет обратную операцию.

```
void cv::idft(
  cv::InputArray  src,                // входной массив (вещественный или комплексный)
  cv::OutputArray dst,                // выходной массив
  int             flags = 0,          // флаги
  int             nonzeroRows = 0     // сколько строк обрабатывать
);
```

Вызов `cv::idft()` в точности эквивалентен вызову `cv::dft()` с флагом `cv::DFT_INVERSE` (любые другие флаги также допустимы).

cv::mulSpectrums(): умножение спектров

Во многих приложениях, где требуется вычислять ДПФ, приходится также поэлементно перемножать результирующие спектры. Поскольку в результате получаются комплексные числа, обычно упакованные в компактный формат CCS, было бы утомительно распаковывать их и производить умножение с помощью «обычных»

---

[1] Это не значит, что он представлен в формате CCS, он лишь обладает симметрией, как было бы, если бы (например) он получился в результате применения прямого преобразования к вещественному массиву. Отметим также, что вы *сообщаете* OpenCV, что массив обладает симметрией, – и она вам верит на слово, не проверяя, так ли это на самом деле.

матричных операций. По счастью, в OpenCV имеется функция `cv::mulSpectrums()`, которая делает именно то, что нужно:

```
void cv::mulSpectrums(
  cv::InputArray  src1,              // входной массив (ccs или комплексный)
  cv::InputArray  src2,              // входной массив (ccs или комплексный)
  cv::OutputArray dst,               // выходной массив
  int             flags,             // построчное умножение
  bool            conj = false       // true - взять комплексно сопряженный к src2
);
```

Первые два аргумента – массивы, которые должны быть либо одноканальными спектрами в формате CCS, либо двухканальными комплексными спектрами – как результаты `cv::dft()`. Третий аргумент – выходной массив такого же размера и типа, как входные. Аргумент `conj` сообщает, что делать. Если он равен `false`, то производится попарное умножение, а если `true`, то элемент первого массива умножается на число, комплексно сопряженное соответственному элементу второго массива[1].

Свертка с помощью дискретного преобразования Фурье

Скорость вычисления свертки можно заметно увеличить, воспользовавшись ДПФ, что гарантируется теоремой о свертке [Titchmarsh26], которая связывает свертку в пространственной области с умножением в частотной области [Morse53; Bracewell65; Arfken85][2]. Для этого мы сначала вычисляем преобразование Фурье изображения и сверточного фильтра. После этого свертку в преобразованном пространстве можно вычислить за время, линейное зависящее от числа пикселей в изображении. В исходном коде такой свертки, заимствованном из документации по OpenCV (пример 12.1), есть интересные примеры использования `cv::dft()`.

Пример 12.1 ❖ Использование `cv::dft()` и `cv::idft()` для ускорения вычисления свертки

```
#include <opencv2/opencv.hpp>
#include <iostream>

using namespace std;

int main(int argc, char** argv) {
  if(argc != 2) {
    cout << "Преобразование Фурье\nПорядок вызова: " <<argv[0] <<" <imagename>" << endl;
    return -1;
  }

  cv::Mat A = cv::imread(argv[1],0);
  if( A.empty() ) { cout << "Не могу загрузить " << argv[1] << endl; return -1; }

  cv::Size patchSize( 100, 100 );
  cv::Point topleft( A.cols/2, A.rows/2 );
  cv::Rect roi( topleft.x, topleft.y, patchSize.width, patchSize.height );
```

[1] Основное назначение этого аргумента – реализация корреляции в пространстве Фурье. Оказывается, что единственное различие между сверткой (см. следующий раздел) и корреляцией состоит в том, что при умножении спектров второй массив заменяется комплексно сопряженным.

[2] Напомним, что в OpenCV ДПФ реализуется как БПФ, если размер данных таков, что БПФ быстрее.

```
  cv::Mat B = A( roi );

  int dft_M = cv::getOptimalDFTSize( A.rows+B.rows-1 );
  int dft_N = cv::getOptimalDFTSize( A.cols+B.cols-1 );

  cv::Mat dft_A = cv::Mat::zeros( dft_M, dft_N, CV::F32 );
  cv::Mat dft_B = cv::Mat::zeros( dft_M, dft_N, CV::F32 );

  cv::Mat dft_A_part = dft_A( Rect(0, 0, A.cols,A.rows) );
  cv::Mat dft_B_part = dft_B( Rect(0, 0, B.cols,B.rows) );

  A.convertTo( dft_A_part, dft_A_part.type(), 1, -mean(A)[0] );
  B.convertTo( dft_B_part, dft_B_part.type(), 1, -mean(B)[0] );

  cv::dft( dft_A, dft_A, 0, A.rows );
  cv::dft( dft_B, dft_B, 0, B.rows );

  // задайте последний параметр равным false, чтобы вычислять свертку, а не корреляцию
  cv::mulSpectrums( dft_A, dft_B, dft_A, 0, true );
  cv::idft( dft_A, dft_A, DFT_SCALE, A.rows + B.rows - 1 );

  cv::Mat corr = dft_A( Rect(0, 0, A.cols + B.cols - 1, A.rows + B.rows - 1) );
  cv::normalize( corr, corr, 0, 1, NORM_MINMAX, corr.type() );
  cv::pow( corr, 3., corr );

  cv::B ^= cv::Scalar::all( 255 );

  cv::imshow( "Image", A );
  cv::imshow( "Correlation", corr );
  cv::waitKey();
  return 0;
}
```

Здесь мы сначала создаем и инициализируем входные массивы. Затем создаем два массива с оптимальными для алгоритма ДПФ размерами, копируем в них исходные массивы и вычисляем преобразования. После этого перемножаем спектры и применяем к произведению обратное преобразование. Самая медленная[1] часть операции – преобразования; для изображения $N \times N$ они занимают время $O(N^2 \log N)$, так что именно столько времени уходит на процедуру в целом (в предположении, что ядро свертки имеет размер $M \times M$ и $N > M$). Это гораздо меньше, чем время $O(N^2 M^2)$, требуемое для вычисления свертки без ДПФ.

cv::dct(): дискретное косинусное преобразование

Для вещественных данных часто достаточно вычислить только половину дискретного преобразования Фурье. *Дискретное косинусное преобразование* (ДКП) [Ahmed74; Jain77] определяется аналогично полному ДПФ по формуле:

$$c_k = \left(\frac{1}{N}\right)^{\frac{1}{2}} x_0 + \sum_{n=1}^{N-1} \left(\frac{2}{N}\right)^{\frac{1}{2}} x_n \cos\left(\left(k + \frac{1}{2}\right)\frac{n}{N}\pi\right).$$

[1] Говоря «самая медленная», мы имеем в виду «асимптотически самая медленная», т. е. для очень больших N эта часть алгоритма занимает основное время. Это важное уточнение. На практике, как мы видели в разделе, посвященном сверткам, накладные расходы на переход в частотное пространство не всегда оправданы. При свертке с небольшим ядром это обычно не имеет смысла.

Подобное преобразование можно определить и в пространстве большей размерности. Отметим, что, по соглашению, к косинусному преобразованию и обратному к нему применяется нормировочный множитель (для дискретного преобразования Фурье такого соглашения нет).

Основные идеи ДПФ применимы и к ДКП, но все коэффициенты теперь вещественные[1]. Сама же функция выглядит так:

```
void cv::dct(
  cv::InputArray  src,                // входной массив (четного размера)
  cv::OutputArray dst,                // выходной массив
  int             flags = 0           // построчно, обращение
);
```

Аргументы `cv::dct()` почти такие же, как у `cv::dft()`, но, поскольку результаты вещественные, нет нужды в какой-то особой упаковке выходного массива (или входного в случае обратного преобразования). Однако, в отличие от `cv::dft()`, во входном массиве должно быть четное число элементов (при необходимости можно дополнить массив нулем в конце). Аргумент `flags` может быть равен `cv::DCT_INVERSE`, если требуется обратное преобразование, и допускает комбинирование с `cv::DCT_ROWS` для той же цели, что в случае `cv::dft()`. Из-за различных соглашений о нормировке результат прямого и обратного косинусного преобразования всегда нормирован, поэтому аналога флага `cv::DFT_SCALE` у `cv::dct()` нет.

Как и в случае `cv::dft()`, производительность сильно зависит от размера массива. На самом деле на очень низком уровне `cv::dct()` вызывает `cv::dft()` для массива, размер которого ровно вдвое меньше размера исходного. Поэтому оптимальный размер массива, передаваемого `cv::dct()`, равен удвоенному оптимальному размеру массива, подаваемого на вход `cv::dft()`. Следовательно, оптимальный размер для `cv::dct()` вычисляется так:

```
size_t optimal_dct_size = 2 * cv::getOptimalDFTSize( (N+1)/2 );
```

где `N` – фактический размер преобразуемых данных.

cv::idct(): обратное дискретное косинусное преобразование

Как и в случае `cv::idft()` и `cv::dft()`, для вычисления обратного преобразования нужно указать подходящее значение аргумента `flags`. Как и раньше, код зачастую становится понятнее, если воспользоваться отдельной функцией обратного преобразования:

```
void cv::idct(
  cv::InputArray  src,                // входной массив
  cv::OutputArray dst,                // выходной массив
  int             flags = 0           // рассматривать как массив строк
);
```

Вызов `cv::idct()` в точности эквивалентен вызову `cv::dct()` с флагом `cv::DCT_INVERSE` (любые другие флаги также допустимы).

---

[1] Дотошный читатель может возразить, что косинусное преобразование применяется к вектору, определенному не в симметричной области. Но `cv::dct()` просто рассматривает вектор так, будто он зеркально продолжен на отрицательные индексы.

Интегральные изображения

OpenCV позволяет вычислить интегральное изображение с помощью функции `cv::integral()`. *Интегральным изображением* [Viola04] называется структура данных, позволяющая быстро вычислять суммы по подобластям[1]. Такое суммирование полезно во многих приложениях, в том числе при вычислении *вейвлетов Хаара*, применяемых в некоторых алгоритмах распознавания лиц и им подобных.

В OpenCV поддерживаются три варианта интегрального изображения: *сумма*, *сумма квадратов* и *сумма с наклоном*. Во всех случаях размер выходного изображения больше размера входного на 1 в каждом направлении.

Стандартное интегральное изображение `sum` вычисляется по формуле:

$$sum(x, y) = \sum_{y'<y} \sum_{x'<x} image(x', y').$$

Изображение `square-sum` вычисляется как сумма квадратов:

$$sum_{square}(x, y) = \sum_{y'<y} \sum_{x'<x} [image(x', y')]^2.$$

А `tilted-sum` – это сумма, но для изображения, повернутого на 45 градусов:

$$sum_{tilted}(x, y) = \sum_{y'<y} \sum_{abs(x'-x)<y} image(x', y').$$

С помощью этих интегральных изображений можно вычислять суммы, средние и стандартные отклонения по любой вертикальной или «наклонной» прямоугольной области изображения. Например, найдем сумму по области, описываемой двумя противоположными вершинами (x_1, y_1) и (x_2, y_2), где $x_2 > x_1$ и $y_2 > y_1$:

$$\sum_{y_1 \le y < y_2} \sum_{x_1 \le x < x_2} image(x, y) = [sum(x_2, y_2) - sum(x_1, y_2) - sum(x_2, y_1) + sum(x_1, y_1)].$$

Таким способом можно быстро вычислять размытие, приближенные градиенты, средние и стандартные отклонения, а также корреляцию блоков даже для окон переменного размера.

Чтобы стало понятнее, рассмотрим изображение 7×5, показанное на рис. 12.1; высота столбика, ассоциированного с каждым пикселем, отражает яркость этого пикселя. Та же информация показана на рис. 12.2 в числовом виде слева и в интегральной форме справа. Для вычисления стандартного интегрального изображения $I'(x, y)$ мы пробегаем по строкам, применяя следующую рекуррентную формулу:

$$I'(x, y) = I(x, y) + I'(x - 1, y) + I'(x, y - 1) - I'(x - 1, y - 1).$$

Последний член вычитается, потому что он учтен дважды: во втором и в третьем членах. Проверьте эту формулу, просчитав по ней некоторые значения на рис. 12.2.

---

[1] Приведенная ссылка содержит наиболее подробное описание метода, но впервые его применение в компьютерном зрении было предложено в 2001 году в статье «Robust Real-Time Object Detection» тех же авторов. Метод использовался в машинной графике еще в 1984 году, и тогда интегральное изображение называлось *таблицей суммарных площадей* (summed area table).

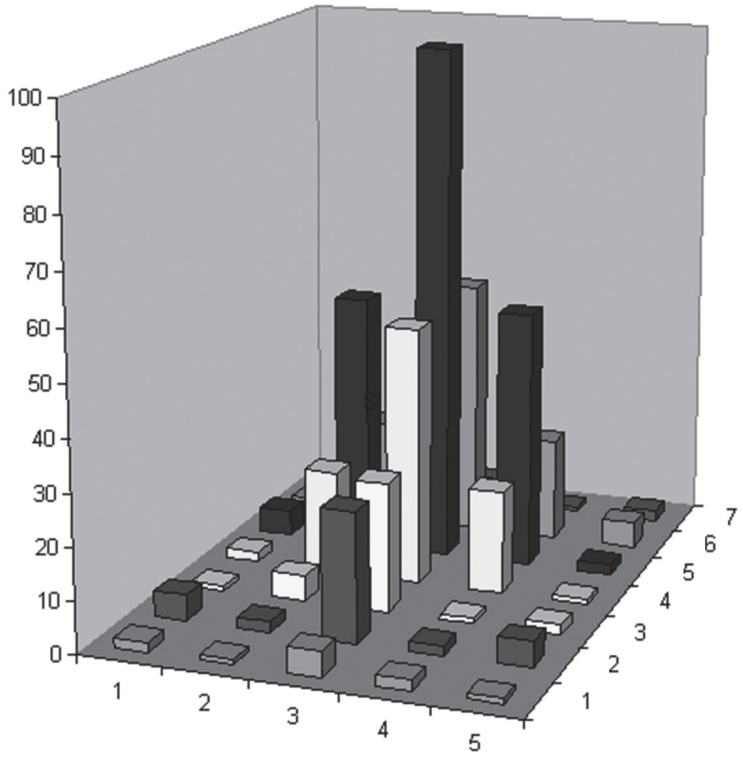

Рис. 12.1 ❖ Изображение 7×5, представленное в виде столбчатой диаграммы, в которой высота столбика равна яркости пикселя

По рис. 12.2 видно, что для суммирования по прямоугольнику со значениями 20 во всех четырех вершинах нужно вычислить выражение 398 – 9 – 10 + 1 = 380. Таким образом, для вычисления суммы по любому прямоугольнику нужно произвести четыре арифметических действия, т. е. сложность составляет $O(1)$.

1	2	5	1	2
2	**20**	**50**	**20**	5
5	**50**	**100**	**50**	2
2	**20**	**50**	**20**	1
1	5	25	1	2
5	2	25	2	5
2	1	5	2	1

0	0	0	0	0	0
0	1	3	8	9	11
0	3	**25**	**80**	**101**	108
0	8	**80**	**235**	**306**	315
0	10	**102**	**307**	**398**	408
0	11	108	338	430	442
0	16	115	370	464	481
0	18	118	378	474	492

Рис. 12.2 ❖ То же изображение 7×5, представленное в числовом виде (начало координат расположено в левом верхнем углу), и его интегральное изображение 8×6

cv::integral(): интегральное изображение в виде суммы

Варианты интегрального изображения в C++ различаются только аргументами (иногда это сбивает с толку). В функции для вычисления суммы их три.

```
void cv::integral(
  cv::InputArray  image,              // входной массив
  cv::OutputArray sum,                // выходной массив сумм
  int             sdepth = -1         // глубина выходного массива (например, cv::F32)
);
```

Первые два аргумента – входное и выходное изображения. Если размер входного изображения $W \times H$, то выходное будет иметь размер $(W + 1) \times (H + 1)$[1]. Аргумент `sdepth` задает глубину выходного изображения (суммы) и может принимать значения `CV::S32`, `CV::F32`, `CV::F64`.[2]

cv::integral(): интегральное изображение в виде суммы квадратов

Этот вариант вычисляет та же функция, что и обычное интегральное изображение, но добавляется еще один выходной массив для суммы квадратов.

```
void cv::integral(
  cv::InputArray  image,              // входной массив
  cv::OutputArray sum,                // выходной массив сумм
  cv::OutputArray sqsum,              // выходной массив сумм квадратов
  int             sdepth = -1         // глубина выходного массива (например, cv::F32)
);
```

Наличие аргумента `sqsum` типа `cv::OutputArray` означает, что функция `cv::integral()` помимо обычной суммы должна вычислить также сумму квадратов. Как и раньше, `sdepth` задает глубину выходных изображений и может принимать значения `CV::S32`, `CV::F32`, `CV::F64`.

cv::integral(): интегральное изображение в виде суммы с наклоном

Как и в случае суммы квадратов, для вычисления суммы с наклоном используется та же функция, но с еще одним дополнительным аргументом.

```
void cv::integral(
  cv::InputArray  image,              // входной массив
  cv::OutputArray sum,                // выходной массив сумм
  cv::OutputArray sqsum,              // выходной массив сумм квадратов
  cv::OutputArray tilted,             // выходной массив сумм с наклоном
  int             sdepth = -1         // глубина выходного массива (например, cv::F32)
);
```

---

[1] Сюда включена строка нулей, поскольку сумма нулевого числа членов равна нулю.

[2] Отметим, что хотя `sum` и `tilted_sum` могут иметь 32-разрядный тип с плавающей точкой, если входное изображение имеет такой же тип, рекомендуется, особенно для больших изображений, задавать глубину 64 разряда, поскольку современные изображения могут насчитывать много миллионов пикселей.

В этом варианте `cv::integral()` вычисляется дополнительный массив `tilted`, остальные аргументы интерпретируются, как и раньше.

Детектор границ Кэнни

Искать границы в изображениях можно с помощью простых фильтров, например фильтра Лапласа, но этот метод допускает значительное усовершенствование. В 1986 году Дж. Кэнни улучшил фильтр Лапласа, и теперь этот алгоритм называется *детектором границ Кэнни* [Canny86]. Одно из отличий от фильтра Лапласа, описанного в предыдущей главе, заключается в том, что вычисляются первые производные по x и y, которые затем объединяются в четыре производные по направлению. Точки, в которых эти производные достигают локального максимума, считаются кандидатами на включение в границы. Самое важное новшество в алгоритме Кэнни – этап, на котором отдельные пиксели-кандидаты объединяются в *контуры*[1].

Для формирования контуров алгоритм применяет к пикселям *гистерезисный порог*. Это означает, что определены два порога: верхний и нижний. Если градиент в пикселе больше верхнего порога, то считается, что пиксель принадлежит границе; если меньше нижнего, то пиксель отклоняется. Если же градиент заключен между порогами, то пиксель принимается, только если он связан с пикселем, в котором градиент больше верхнего порога. Кэнни рекомендовал выбирать отношение порогов между 2:1 и 3:1. На рис. 12.3 и 12.4 показаны результаты применения `cv::Canny()` к контрольному изображению и к фотографии при отношении порогов 5:1 и 3:2 соответственно.

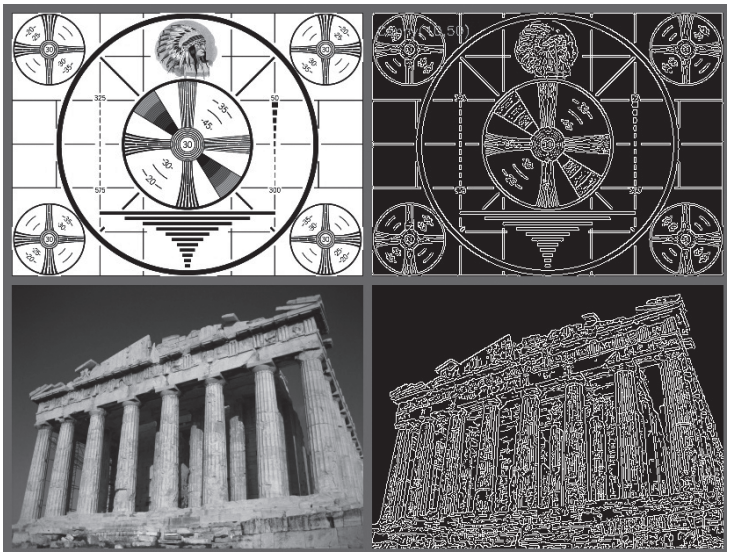

Рис. 12.3 ❖ Результаты детектора Кэнни для двух изображений в случае, когда верхний порог равен 50, а нижний – 10

[1] Мы еще много будем говорить о контурах ниже. А в ожидании этих открытий имейте в виду, что функция `cv::Canny()` на самом деле не возвращает объекты-контуры; если они вам нужны, то придется построить их, применив к результату `cv::Canny()` функцию `cv::findContours()`. Подробно о контурах написано в главе 14.

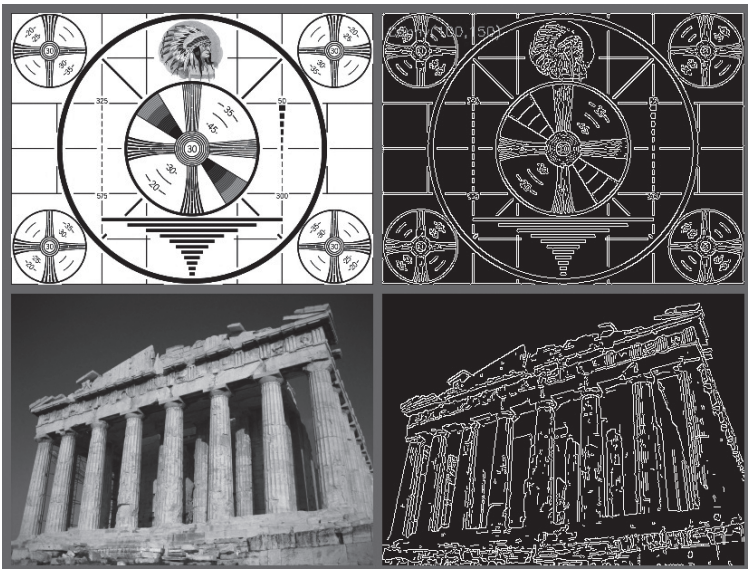

Рис. 12.4 ❖ Результаты детектора Кэнни для двух изображений в случае, когда верхний порог равен 150, а нижний – 100

cv::Canny()

Реализация алгоритма распознавания границ Кэнни в OpenCV преобразует исходное изображение в «изображение границ».

```
void cv::Canny(
  cv::InputArray  image,              // входное одноканальное изображение
  cv::OutputArray edges,              // выходное изображение границ
  double          threshold1,         // нижний порог
  double          threshold2,         // верхний порог
  int             apertureSize = 3,   // размер ядра Собеля
  bool            L2gradient = false  // true=L2-норма (более точная)
);
```

Функция `cv::Canny()` получает входное изображение, обязательно одноканальное, и выходное изображение, тоже полутоновое (но в действительности оно будет булевым). Далее задаются оба порога. Аргумент `apertureSize` задает размер ядра операторов Собеля для вычисления производных, которые вызываются из `cv::Canny()`. Последний аргумент `L2gradient` говорит, нужно ли вычислять градиент по направлению «правильно» – с использованием нормы L_2, или быстрее, но менее точно с использованием нормы L_1. Если `L2gradient` равен `true`, то градиент вычисляется по такой формуле:

$$\left|grad(x,y)\right|_{L_2} = \sqrt{\left(\frac{dI}{dx}\right)^2 + \left(\frac{dI}{dy}\right)^2}.$$

А если `L2gradient` равен `false`, то по более быстрой:

$$\left|grad(x,y)\right|_{L_1} = \left|\frac{dI}{dx}\right| + \left|\frac{dI}{dy}\right|.$$

Преобразования Хафа

Преобразование Хафа[1] – это метод отыскания прямых линий, окружностей и других простых фигур в изображении. Оригинальный вариант применялся для сравнительно быстрого поиска прямых в бинарном изображении, но впоследствии был обобщен на другие элементы.

Преобразование Хафа для поиска прямых

Основная идея преобразования Хафа для поиска прямых состоит в том, что любая точка бинарного изображения может принадлежать некоторому множеству возможных прямых линий. Если представить прямую уравнением $y = ax + b$, то точка исходного изображения преобразуется в геометрическое место точек на параметрической плоскости (a, b), соответствующих всем прямым, проходящим через эту точку (рис. 12.5). Если преобразовать каждый ненулевой пиксель входного изображения в такое множество точек выходного изображения и просуммировать по нему, то прямые во входном изображении (т. е. на плоскости (x, y)) будут выглядеть как локальные максимумы в выходном изображении (на плоскости (a, b)). Поскольку суммирование производится по вкладам каждой точки, плоскость (a, b) обычно называется *аккумуляторной плоскостью*.

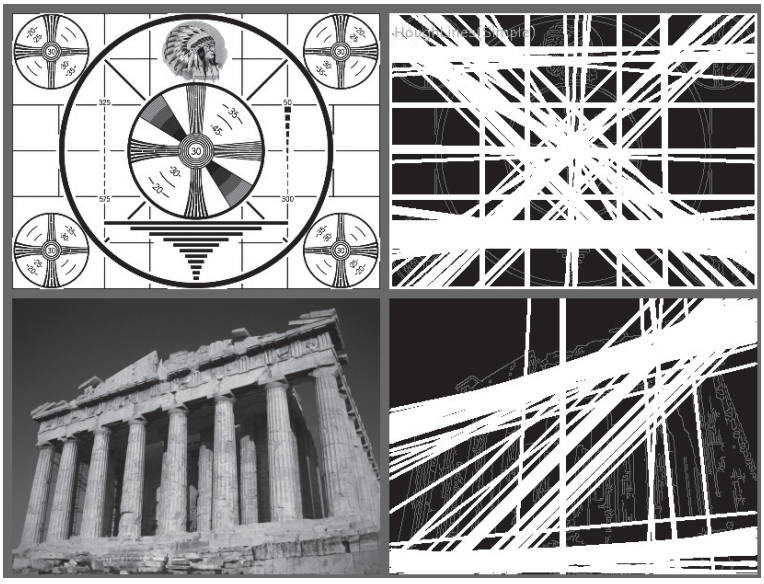

Рис. 12.5 ❖ Преобразование Хафа находит много прямых в изображении, одни ожидаемые, другие – нет

Возможно, вам пришла в голову мысль, что угловой коэффициент и точка пересечения с осью ординат – не лучший способ описания всех прямых, проходящих через данную точку (из-за существенно различной плотности прямых как функции углово-

[1] Хаф разработал это преобразование для экспериментальной физики [Hough59]; к компьютерному зрению его применили Дуда и Харт [Duda72].

го коэффициента и связанным с этим фактом, что угловой коэффициент изменяется от $-\infty$ до $+\infty$). Поэтому на самом деле при численных расчетах используется другая параметризация. Прямая представляется точкой (ρ, θ) в полярной системе координат, так что представленная некоторой точкой прямая проходит через нее перпендикулярно ее радиусу-вектору (рис. 12.6). Такая прямая описывается уравнением

$$\rho = x\cos\theta + y\sin\theta.$$

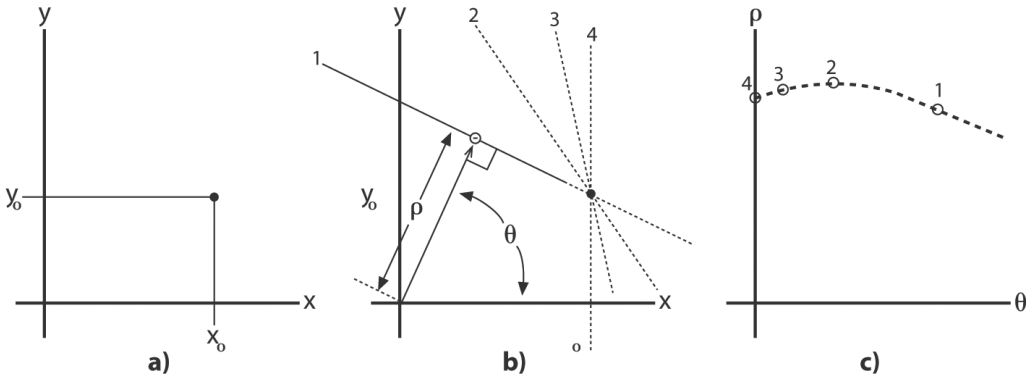

Рис. 12.6 ❖ Точка (x_0, y_0) на плоскости изображения (a) принадлежит множеству прямых, каждая из которых параметризуется различными значениями ρ и θ (b); на плоскости (ρ, θ) совокупность таких прямых образует кривую характерной формы (c)

В OpenCV преобразование Хафа реализовано так, что это вычисление не видно пользователю, а просто возвращаются локальные максимумы на плоскости (ρ, θ). Но вы должны понимать, что за ними стоит, чтобы осмысленно задавать аргументы функции.

OpenCV поддерживает три варианта преобразования Хафа для поиска прямых: *стандартное преобразование Хафа* (SHT) [Duda72], *многомасштабное преобразование Хафа* (MHT) и *прогрессивное вероятностное преобразование Хафа* (PPHT)[1]. Алгоритм SHT был рассмотрен выше. Немного усовершенствованный алгоритм MHT более точно находит прямые. PPHT – вариант этого алгоритма, который, среди прочего, вычисляет не только ориентацию, но и протяженность отрезка прямой (рис. 12.7). Он называется «вероятностным», потому что аккумулирует не все возможные точки на аккумуляторной плоскости, а только часть. Идея в том, что если пик достаточно высокий, то для нахождения будет достаточно и части точек, результатом такого предположения является заметное сокращение времени вычислений.

---

[1] *Прогрессивное преобразование Хафа* (PHT) предложено Кириати, Элдаром и Брукштейном в 1991 году [Kiryati91], а PPHT – Матасом, Галамбосом и Киттлером в 1999 [Matas00].

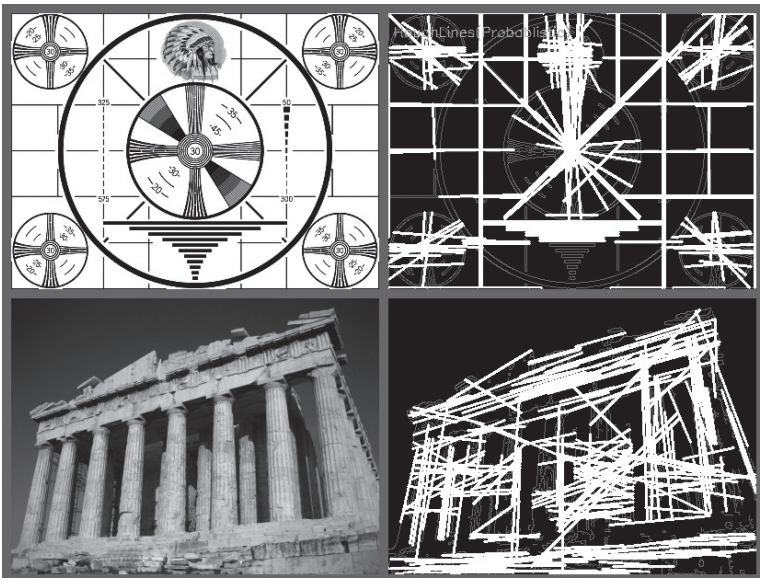

Рис. 12.7 ❖ Сначала выполнен детектор границ Кэнни (param1=50, param2=150), его результаты показаны серым цветом, а затем – прогрессивное вероятностное преобразование Хафа (param1=50, param2=10), результаты показаны белым цветом на том же рисунке. Как видим, преобразование Хафа обычно находит сильные прямые границы

cv::HoughLines(): стандартное и многомасштабное преобразования Хафа

Стандартное и многомасштабное преобразования Хафа реализованы одной функцией `cv::HoughLines()`, выбор алгоритма зависит от двух необязательных аргументов.

```
void cv::HoughLines(
  cv::InputArray  image,              // входное одноканальное изображение
  cv::OutputArray lines,              // двухканальный массив Nx1
  double          rho,                // разрешающая способность по ро (в пикселях)
  double          theta,              // разрешающая способность по тета (в радианах)
  int             threshold,          // ненормированный порог аккумулятора
  double          srn = 0,            // уточнение rho (для МНТ)
  double          stn = 0             // уточнение theta (для МНТ)
);
```

Входное изображение (первый аргумент) должно быть 8-разрядным, но рассматривается оно как бинарное (все ненулевые пиксели эквивалентны). Второй аргумент – массив для размещения найденных прямых – будет создан как двухканальный массив $N \times 1$ с плавающей точкой (число столбцов N будет равно числу возвращенных прямых)[1]. Каналы будут содержать значения ро (ρ) и тета (θ) для каждой найденной прямой.

---

[1] Как обычно, в зависимости от типа объекта, переданного в аргументе `lines`, это может быть либо массив $1 \times N$ с двумя каналами, либо `std::vector<>` с N элементами типа `Vec2f`.

Аргументы `rho` и `theta` задают требуемую разрешающую способность для прямых (т. е. квантизацию аккумуляторной плоскости). Значение `rho` измеряется в пикселях, значение `theta` – в радианах; таким образом, аккумуляторную плоскость можно представлять себе как двумерную сетку `rho` пикселей на `theta` радиан. Аргумент `threshold` определяет порог, при котором алгоритм сообщает о найденной прямой. На практике его применение наталкивается на определенные сложности: он не нормирован, поэтому для алгоритма SHT будьте готовы увеличить его с ростом размера изображения. Напомним, что этот аргумент, по существу, говорит, сколько точек (на границе внутри изображения) должно принадлежать прямой, чтобы алгоритм вернул эту прямую.

Аргументы `srn` и `stn` в алгоритме стандартного преобразования Хафа не используются, они управляют поведением его обобщения – многомасштабного преобразования Хафа (MHT). В MHT они задают более высокое разрешение, с которым следует вычислять параметры прямых. Сначала MHT вычисляет положения прямых с точностью, заданной аргументами `rho` и `theta`, а затем уточняет результаты с коэффициентами `srn` и `stn` соответственно (т. е. окончательная разрешающая способность по ро равна `rho`, поделенному на `srn`, а по тета – `theta`, поделенному на `stn`). Если оба параметра оставить равными 0, то будет работать алгоритм SHT.

cv::HoughLinesP(): прогрессивное вероятностное преобразование Хафа

```
void cv::HoughLinesP(
  cv::InputArray  image,              // входное одноканальное изображение
  cv::OutputArray lines,              // 4-канальный массив Nx1
  double          rho,                // разрешающая способность по ро (в пикселях)
  double          theta,              // разрешающая способность по тета (в радианах)
  int             threshold,          // ненормированный порог аккумулятора
  double          minLineLength = 0,  // требуемая минимальная длина отрезка
  double          maxLineGap = 0      // требуемый промежуток между отрезками
);
```

Функция `cv::HoughLinesP()` похожа на `cv::HoughLines()`, но имеет два важных отличия. Во-первых, в аргументе `lines` передается четырехканальный массив (или вектор объектов типа `Vec4i`). Каналами будут координаты (x_0,y_0) и (x_1,y_1) (в таком порядке) концов найденных отрезков. Во-вторых, у двух последних аргументов другая семантика: `minLineLength` и `maxLineGap` задают минимальную длину возвращаемых отрезков и промежуток между коллинеарными отрезками, при котором алгоритм не будет объединять их в один более длинный отрезок.

Преобразование Хафа для поиска окружностей

Преобразование Хафа для поиска окружностей [Kimme75] (рис. 12.8) работает примерно так же, как описанное выше преобразование для поиска прямых. «Примерно» – потому что если бы мы попытались сделать нечто в точности аналогичное, то аккумуляторную *плоскость* пришлось бы заменить аккумуляторным *пространством* с тремя измерениями: x, y (координаты центра окружности) и r (радиус окружности). Потребовалось бы намного больше памяти, а скорость резко упала бы. В OpenCV эта проблема обходится благодаря использованию более хитроумного *градиентного метода Хафа*.

Градиентный метод Хафа работает следующим образом. Сначала выполняется этап выделения границ (функция `cv::Canny()`). Затем для каждой ненулевой точки в изображении границ рассматривается локальный градиент (для его вычисления сначала вычисляются первые производные Собеля по x и по y функцией `cv::Sobel()`). Зная градиент,

мы инкрементируем каждую точку на отрезке с таким угловым коэффициентом (от заданного минимального до заданного максимального расстояния) в аккумуляторе. Вместе с тем запоминаются координаты каждого ненулевого пикселя в изображении границ. Кандидаты на роль центров выбираются среди тех точек (двумерного) аккумулятора, которые одновременно превышают заданный порог и больше всех своих непосредственных соседей. Эти кандидаты сортируются в порядке значений аккумулятора, так что центры с наибольшим числом поддерживающих пикселей располагаются в начале. Затем для каждого центра рассматриваются все ненулевые пиксели (напомним, что их список был построен ранее). Эти пиксели сортируются по расстоянию до центра. Перебирая радиусы в порядке от наименьшего к наибольшему, мы находим тот единственный радиус, который поддержан наибольшим числом ненулевых пикселей. Центр оставляется, если он имеет достаточную поддержку со стороны ненулевых пикселей в изображении границ и располагается на достаточном удалении от ранее выбранных центров.

Эта реализация работает гораздо быстрее и, что, пожалуй, важнее, обходит проблему разреженности трехмерного аккумулятора, приводящую к шуму и нестабильности результатов. С другой стороны, у этого алгоритма есть и недостатки, о которых следует знать.

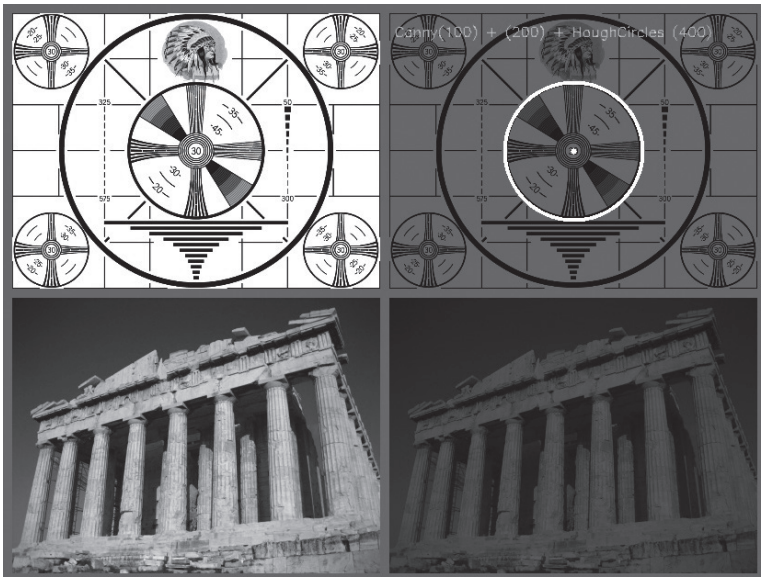

Рис. 12.8 ❖ Преобразование Хафа для поиска окружностей находит окружности в контрольном изображении и не находит на фотографии (где их и нет)

Во-первых, использование производных Собеля для вычисления локального градиента – и сопутствующее предположение о том, что градиент можно рассматривать как локальный тангенс угла наклона, – вычислительно неустойчиво. Оно может давать правильные результаты «в большинстве случаев», но следует ожидать некоторой зашумленности выхода.

Во-вторых, для каждого кандидата на роль центра рассматривается весь набор ненулевых пикселей в изображении границ, поэтому если выбрать порог аккумулятора слишком низким, то алгоритм будет работать долго. В-третьих, поскольку для каждо-

го центра выбирается только одна окружность, при наличии концентрических окружностей алгоритм найдет только одну.

Наконец, поскольку центры рассматриваются в порядке возрастания ассоциированных с ними значений аккумулятора, а новые центры не сохраняются, если находятся слишком близко к уже найденным, то при наличии концентрических или почти концентрических окружностей алгоритм может отдавать предпочтение окружностям большего радиуса («может» – из-за шума, связанного с производными Собеля; для гладких изображений с бесконечной разрешающей способностью никакой неопределенности бы не было).

cv::HoughCircles(): преобразования Хафа для поиска окружностей

```
void cv::HoughCircles(
  cv::InputArray  image,              // входное одноканальное изображение
  cv::OutputArray circles,            // трехканальный массив Nx1 или вектор Vec3f
  int             method,             // всегда cv::HOUGH_GRADIENT
  double          dp,                 // разрешающая способность аккумулятора (отношение)
  double          minDist,            // требуемый промежуток между окружностями
  double          param1 = 100,       // верхний порог детектора Кэнни
  double          param2 = 100,       // ненормированный порог аккумулятора
  int             minRadius = 0,      // минимально допустимый радиус
  int             maxRadius = 0       // максимально допустимый радиус
);
```

Входное изображение `image`, как и раньше, должно быть 8-разрядным. В отличие от функции `cv::HoughLines()`, которая требует бинарного изображения, `cv::HoughCircles()` автоматически вызывает `cv::Sobel()`[1], поэтому может работать с более общим полутоновым изображением.

Выходной массив `circles` будет матрицей или вектором в зависимости от того, что передано функции `cv::HoughCircles()`. Если используется матрица, то `circles` будет одномерным массивом типа `CV::F32C3`, а его три канала служат для кодирования координат центра и радиуса окружности. Если же используется вектор, то он должен иметь тип `std::vector<Vec3f>`. Аргумент `method` всегда должен быть равен `cv::HOUGH_GRADIENT`.

Аргумент `dp` – разрешающая способность аккумуляторного изображения. Он позволяет создать аккумулятор с меньшим разрешением, чем у входного изображения. Это имеет смысл, т. к. нет причин ожидать, что присутствующие в изображении окружности естественно квантуются так же, как ширина или высота самого изображения. Если `dp` равен `1`, то разрешения совпадают; если `dp` больше `1` (например, `2`), то разрешение аккумулятора будет во столько раз меньше (в данном случае вдвое). Значения меньше `1` не допускаются.

Аргумент `minDist` задает минимальное расстояние между окружностями, при котором алгоритм будет считать их различными.

Аргументы `param1` и `param2` – порог детектора Кэнни и порог аккумулятора. Напомним, что детектору Кэнни на самом деле нужны два порога. При вызове `cv::Canny()` из `cv::HoughLines()` первому (верхнему) порогу присваивается значение `param1`, а второму (нижнему) – в два раза меньшее. Аргумент `param2` в точности аналогичен аргументу `threshold` функции `cv::HoughLines()`.

---

[1] Вызывается именно `cv::Sobel()`, а не `cv::Canny()`, поскольку `cv::HoughCircles()` должна оценивать ориентацию градиента в каждом пикселе, а это трудно сделать, имея только бинарное изображение границ.

Последние два аргумента – минимальный и максимальный радиусы распознаваемых окружностей. Только такие окружности будут представлены в аккумуляторе. В программе из примера 12.2 демонстрируется использование функции `cv::HoughCircles()`.

Пример 12.2 ❖ Применение `cv::HoughCircles()` для возврата последовательности окружностей, найденных в полутоновом изображении

```
#include <opencv2/opencv.hpp>
#include <iostream>
#include <math.h>

using namespace cv;
using namespace std;

int main(int argc, char** argv) {
  if(argc != 2) {
    cout << "Поиск окружностей методом Хафа\nПорядок вызова: " <<argv[0]
         <<" <imagename>\n" << endl;
    return -1;
  }

  cv::Mat src, image;
  src = cv::imread( argv[1], 1 );
  if( src.empty() ) { cout << "Не могу загрузить " << argv[1] << endl; return -1; }

  cv::cvtColor(src, image, cv::BGR2GRAY);
  cv::GaussianBlur(image, image, Size(5,5), 0, 0);

  vector<cv::Vec3f> circles;
  cv::HoughCircles(image, circles, cv::HOUGH_GRADIENT, 2, image.cols/10);

  for( size_t i = 0; i < circles.size(); ++i ) {
    cv::circle(src,
      cv::Point(cvRound(circles[i][0]), cvRound(circles[i][1])),
      cvRound(circles[i][2]),
      cv::Scalar(0,0,255), 2, cv::AA
    );
  }

  cv::imshow( "Hough Circles", src);
  cv::waitKey(0);
  return 0;
}
```

Стоит принять во внимание, что какие бы трюки мы ни использовали, никуда не деться от того факта, что окружность описывается тремя параметрами (x, y, r), а не двумя (ρ и θ), как прямая линия. Поэтому алгоритму поиска окружностей по необходимости требуется больше памяти и времени, чем алгоритму поиска прямых. А раз так, то стоит наложить на длину радиуса настолько строгие ограничения, насколько позволяют обстоятельства, чтобы держать накладные расходы под контролем[1]. Пре-

[1] Функция `cv::HoughCircles()` достаточно хорошо находит центры, но с определением радиуса иногда промахивается. Поэтому если приложению необходим только центр (или если радиус можно найти каким-то другим способом), лучше игнорировать радиус, полученный от `cv::HoughCircles()`.

образование Хафа было обобщено на произвольные геометрические фигуры в работе Балларда [Ballard81], где объекты рассматриваются как наборы градиентных границ.

Дистанционное преобразование

Дистанционным преобразованием изображения называется новое изображение, каждый пиксель которого равен расстоянию от соответственного пикселя входного изображения до ближайшего к нему нулевого пикселя, измеренному в соответствии с заданной метрикой. Сразу понятно, что типичным входом для дистанционного преобразования должно быть какое-то изображение границ. В большинстве приложений это результат некоторого детектора границ, например детектора Кэнни, предварительно инвертированный (так что границы обозначаются нулями, а остальные точки – значениями, отличными от нуля).

Для вычисления дистанционного преобразования есть два метода. В первом используется маска, обычно размера 3×3 или 5×5. Каждая точка маски определяет «расстояние» от точки в этой позиции до центра маски. Расстояния до более далеких точек строятся (приближенно) в виде последовательностей «ходов», определяемых элементами маски. Это означает, что чем больше маска, тем точнее расстояние. Если используется этот метод, то маска автоматически выбирается в соответствии с заданной метрикой по правилу, известному OpenCV. Этот метод разработан Боргефорсом (1986) [Borgefors86]. Второй метод вычисляет точные расстояния и принадлежит Фельценсцвальбу [Felzenszwalb04]. Время работы обоих методов линейно зависит от общего числа пикселей, но точный алгоритм чуть медленнее.

Допустимо несколько метрик, в т. ч. классическая метрика L2 (евклидова). На рис. 12.9 показаны примеры применения дистанционного преобразования к контрольному изображению и к фотографии.

Пользовательская маска 3×3
(a = 1, b = 1.5)

4.5	4	3.5	3	3.5	4	4.5
4	3	2.5	2	2.5	3	4
3.5	2.5	1.5	1	1.5	2.5	3.5
3	2	1	0	1	2	3
3.5	2.5	1.5	1	1.5	2.5	3.5
4	3	2.5	2	2.5	3	4
4.5	4	3.5	3	3.5	4	4.5

Пользовательская маска 5×5
(a = 1, b = 1.5, c = 2)

4.5	3.5	3	3	3	3.5	4.5
3.5	3	2	2	2	3	3.5
3	2	1.5	1	1.5	2	3
3	2	1	0	1	2	3
3	2	1.5	1	1.5	2	3
3.5	3	2	2	2	3	3.5
4	3.5	3	3	3	3.5	4

Рис. 12.9 ❖ Сначала отработал детектор границ Кэнни с параметрами param1=100 и param2=200, затем выполнено дистанционное преобразование и результат масштабирован с коэффициентом 5 для большей наглядности

cv::distanceTransform(): непомеченное дистанционное преобразование

Результатом функции cv::distanceTransform() является выходное 32-разрядное изображение с плавающей точкой (типа CV::F32).

```
void cv::distanceTransform(
  cv::InputArray  src,                // входное изображение
  cv::OutputArray dst,                // выходное изображение
  int             distanceType,       // метрика
  int             maskSize            // размер маски
);
```

У функции cv::distanceTransform() два параметра. Первый, distanceType, определяет используемую метрику и может принимать значения cv::DIST_C, cv::DIST_L1, cv::DIST_L2. В первых двух случаях для каждого пикселя входного изображения вычисляется расстояние до ближайшего нулевого пикселя в виде целого числа шагов по линиям сетки. Разница состоит в том, что в случае cv::DIST_C рассматривается 4-связная сетка (без диагоналей), а в случае cv::DIST_L1 – 8-связная (с диагоналями). Если же distanceType равен cv::DIST_L2, то cv::distanceTransform() вычисляет точное евклидово расстояние.

Аргумент maskSize может принимать значения 3, 5 или cv::DIST_MASK_PRECISE. В первых двух случаях используется метод Боргефорса с маской 3×3 или 5×5 соответственно. Для метрик cv::DIST_L1 и cv::DIST_C всегда можно использовать маску 3×3, и результат получится точным. В случае cv::DIST_L2 метод Боргефорса всегда оказывается приближенным, а использование большей маски 5×5 дает лучшую аппроксимацию ценой небольшого замедления вычислений. Значение cv::DIST_MASK_PRECISE означает, что нужно использовать алгоритм Фельценсцвальба (если задана метрика cv::DIST_L2).

cv::distanceTransform(): помеченное дистанционное преобразование

Алгоритм дистанционного преобразования можно попросить не только вычислять минимальное расстояние, но и сообщать, какой объект находится на этом расстоянии. Такие «объекты» называются *связными компонентами*. О связных компонентах мы будем говорить гораздо подробнее в главе 14, пока считайте, что это области исходного изображения, состоящие из непрерывно связанных нулевых пикселей.

```
void cv::distanceTransform(
  cv::InputArray  src,                          // входное изображение
  cv::OutputArray dst,                          // выходное изображение
  cv::OutputArray labels,                       // идентификаторы связных компонент
  int             distanceType,                 // метрика
  int             maskSize,                     // размер маски
  int             labelType = cv::DIST_LABEL_CCOMP  // как помечать
);
```

Если задан массив labels, то после выполнения cv::distanceTransform() он будет иметь такой же размер, как dst. В этом случае связные компоненты вычисляются ав-

томатически и в каждый пиксель `labels` записывается метка ближайшей компоненты. По существу, массив меток – не что иное, как дискретная диаграмма Вороного.

Допустим, нас интересует метка пикселя. Для любого нулевого пикселя в `src` расстояние до ближайшего нуля равно `0`. Кроме того, метка этого пикселя совпадает с меткой связной компоненты, которой он принадлежит. Поэтому, чтобы узнать, какая метка присвоена данному ненулевому пикселю, нужно просто найти этот пиксель в массиве `labels`.

Аргумент `labelType` может быть равен `cv::DIST_LABEL_CCOMP` или `cv::DIST_LABEL_PIXEL`. В первом случае функция автоматически находит связные компоненты из нулевых пикселей и присваивает каждой уникальную метку, во втором все нулевые пиксели получают разные метки.

Сегментация

Сегментация изображения – обширная тема, которой мы уже несколько раз касались и к которой вернемся снова в более сложных контекстах. Здесь же мы рассмотрим несколько библиотечных функций, которые реализуют либо сами методы сегментации, либо примитивы, которые впоследствии будут использованы для построения собственных алгоритмов. Отметим, что в настоящее время не существует общего «магического» решения проблемы сегментации, в этой области ведутся активные исследования. Но тем не менее разработано много хороших методов, которые надежно ведут себя хотя бы в некоторых конкретных задачах и дают прекрасные практические результаты.

Заливка

Заливка [Heckbert90; Shaw04; Vandevenne04] – чрезвычайно полезная функция, которая часто применяется для пометки или выделения частей изображения для последующей обработки или анализа. Заливка позволяет построить на основе входного изображения маски, которые будут использованы на последующих этапах, чтобы ускорить обработку или ограничить ее только пикселями, указанными в маске. Сама функция `cv::floodFill()` принимает необязательную маску, которая позволяет управлять выполнением заливки (например, произвести многократную заливку одного изображения).

В OpenCV заливка – это обобщение функциональности, которую вы, наверное, привыкли ожидать от типичного графического редактора. В обоих случаях в изображении выбирается *начальная точка*, и все похожие на нее соседние точки закрашиваются одним цветом. Разница в том, что соседние пиксели необязательно должны быть одного цвета[1]. Результатом операции заливки всегда является одна непрерывная область. Функция `cv::floodFill()` закрашивает соседний пиксель, если он попадает в указанный диапазон (от `lowDiff` до `highDiff`) относительного текущего пикселя или начального значения `seed` (в зависимости от аргумента `flags`). На заливку можно наложить ограничение с помощью аргумента `mask`. У функции `cv::floodFill()` два перегруженных варианта: с аргументом `mask` и без него.

```
int cv::floodFill(
  cv::InputOutputArray image,                  // входное изображение, от 1 до 3 каналов
  cv::Point            seed,                   // начальная точка
```

[1] Пользователям современных графических программ стоит заметить, что в большинстве из них применяется алгоритм заливки, очень напоминающий `cv::floodFill()`.

```
  cv::Scalar          newVal,                 // в какой цвет красить пиксели
  cv::Rect*           rect,                   // внешняя граница закрашиваемой области
  cv::Scalar          lowDiff = cv::Scalar(), // нижняя граница цветового различия
  cv::Scalar          highDiff = cv::Scalar(),// верхняя граница цветового различия
  int                 flags                   // локально/глобально, только маска
);

int cv::floodFill(
  cv::InputOutputArray image,                 // входное изображение, от 1 до 3 каналов
  cv::InputOutputArray mask,                  // 8-разрядное, w+2 x h+2 (Nc=1)
  cv::Point           seed,                   // начальная точка
  cv::Scalar          newVal,                 // в какой цвет красить пиксели
  cv::Rect*           rect,                   // внешняя граница закрашиваемой области
  cv::Scalar          lowDiff = cv::Scalar(), // нижняя граница цветового различия
  cv::Scalar          highDiff = cv::Scalar(),// верхняя граница цветового различия
  int                 flags                   // локально/глобально, только маска
);
```

Входное изображение `image` должно быть 8-разрядным или с плавающей точкой и иметь один или три канала. Функция `cv::floodFill()` может его модифицировать. Заливка начинается с точки `seed`. Пикселю в этой точке присваивается значение `newVal`, как и всем остальным пикселям, которые алгоритм закрашивает. Пиксель закрашивается, если его яркость не меньше яркости закрашенного соседа минус `lowDiff` и не больше яркости закрашенного соседа плюс `highDiff`. Если в аргументе `flags` поднят флаг `cv::FLOODFILL_FIXED_RANGE`, то пиксель сравнивается не со своими соседями, а с начальной точкой. Первый пример заливки показан на рис. 12.10.

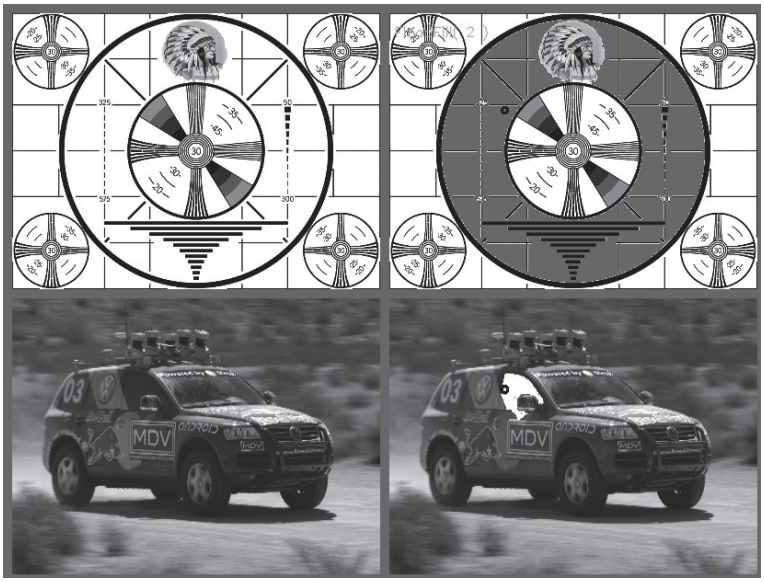

Рис. 12.10 ❖ Результаты заливки
(верхнее изображение заливалось серым цветом, нижнее – белым),
начиная с темного кружка, который в обоих случаях расположен рядом с центром.
Оба параметра `lowDiff` и `highDiff` равны 7.0

Аргумент mask определяет маску, которая может быть как входом, так и выходом cv::floodFill(). В первом случае маска ограничивает заливаемую область, а во втором показывает, какие участки были залиты. Массив mask должен быть 8-разрядным и одноканальным, а его ширина и высота должны быть на два пикселя больше, чем у исходного изображения[1].

Если mask подается на вход cv::floodFill(), то алгоритм никогда не будет пересекать при заливке ненулевые пиксели маски. Поэтому если вы не хотите, чтобы маска блокировала заливку, предварительно обнулите ее.

Если массив mask задан, то он используется и как выход. Для каждого закрашенного пикселя в маске будет установлено ненулевое значение. Если в аргументе flags поднят флаг cv::FLOODFILL_MASK_ONLY, входное изображение не модифицируется, а заполняется только маска.

Если используется маска заливки, то пиксели маски, соответствующие перекрашенным пикселям, по умолчанию устанавливаются в 1. Не пугайтесь, если, выведя маску на экран, не увидите ничего, кроме черного пятна: пиксели установлены, как надо, но чтобы они стали видны, их нужно масштабировать. Ведь, согласитесь, разница между 0 и 1 на шкале яркости от 0 до 255 почти незаметна.

Два флага, которые можно поднять в аргументе flags, мы уже упомянули: cv::FLOODFILL_FIXED_RANGE и cv::FLOODFILL_MASK_ONLY. Помимо них, можно еще прибавить число 4 или 8[2], которые говорят, должен ли алгоритм заливки рассматривать массив пикселей как 4-связный или 8-связный. В первом случае считается, что соседей у пикселя только четыре (слева, справа, сверху и снизу), а во втором к ним добавляются соседи по диагонали – всего восемь.

Аргумент flags состоит из трех разных частей и интуитивно не очевиден. В *младшие* 8 бит (0–7) можно записать одно из двух значений: 4 или 8, они управляют связностью сетки. *Старшие* 8 бит (16–23) содержат флаги cv::FLOODFILL_FIXED_RANGE и cv::FLOODFILL_MASK_ONLY. А вот *средние* биты (8–15) содержат числовое значение, а именно то, которым заполняется маска. Если все эти биты равны 0, то маска заполняется значением 1 (по умолчанию), а любая другая комбинация интерпретируется как 8-разрядное целое без знака. Все три части можно объединить оператором OR. Например, если вы хотите иметь 8-связный массив, сравнивать значение пикселя с начальной точкой и заполнять только маску (не изображение) значением 47, то передайте в качестве аргумента такое выражение:

```
flags = 8
      | cv::FLOODFILL_MASK_ONLY
      | cv::FLOODFILL_FIXED_RANGE
      | (47<<8);
```

---

[1] Это необходимо для упрощения и ускорения внутреннего алгоритма. Отметим, что поскольку маска больше исходного изображения, пикселю (x, y) изображения соответствует пиксель (x + 1, y + 1) маски. Это отличная возможность воспользоваться функцией cv::Mat::getSubRect().

[2] Мы написали «прибавить», но не надо забывать, что flags – это битовая маска. По счастью, в двоичном представлении чисел 4 и 8 ровно один ненулевой бит. Поэтому можно использовать как сложение, так и логическую дизъюнкцию (например, flags = 8 | cv::FLOODFILL_MASK_ONLY).

На рис. 12.11 показан результат заливки для демонстрационных изображений. Задание флага `cv::FLOODFILL_FIXED_RANGE` с широким диапазоном привело к заливке большей части изображения (начиная с черного кружка). Следует отметить, что аргументы `newVal`, `lowDiff` и `highDiff` имеют тип `cv::Scalar`, поэтому в них можно задать сразу три канала. Например, если положить `lowDiff = cv::Scalar(20,30,40)`, то будет установлен нижний порог 20 для красного, 30 для зеленого и 40 для синего.

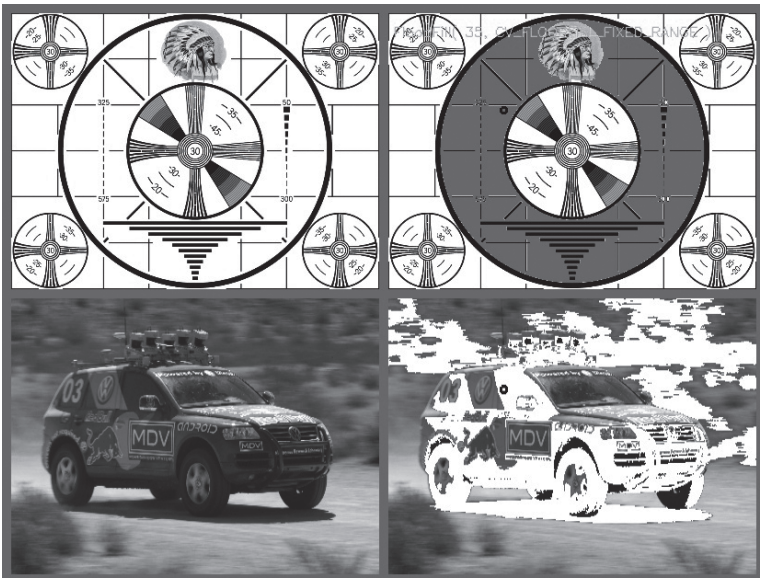

Рис. 12.11 ❖ Результаты заливки
(верхнее изображение заливалось серым цветом, нижнее – белым),
начиная с темного кружка, который в обоих случаях расположен рядом с центром.
В этом случае был выбран фиксированный диапазон,
а разность между верхним и нижним порогами составила 25.0

Алгоритм водораздела

Во многих практических задачах требуется сегментировать изображение, но нет возможности задать какую-то маску фона. В таких случаях оказывается эффективен *алгоритм водораздела* [Meyer92], который преобразует линии в «горы», а однородные области – в «долины», чем можно воспользоваться для сегментирования. Алгоритм сначала вычисляет градиент яркостного изображения; при этом образуются долины, или *котловины* (низменные области), в которых сосредоточена текстура, и горы, или *гребни* (возвышенные области, соответствующие границам), в которых доминируют линии. Затем он последовательно заливает котловины, начиная с указанных пользователем точек, пока области не встретятся. Объединившиеся области сегментируются как один объект. Таким образом, котловины, связанные с маркерной точкой, становятся «собственностью» этого маркера. Области, принадлежащие различным маркерам, являются сегментами изображения.

Точнее, алгоритм водораздела позволяет пользователю (или другому алгоритму) пометить части объекта или фона как таковые. Или вызывающая программа может нарисовать простую линию или набор линий, сказав тем самым, что алгоритм должен

«сгруппировать точки, похожие на эти». Затем алгоритм водораздела сегментирует изображение, отдав этим маркерам во владение связанные с ними долины в градиентном изображении, которые определены найденными в нем границами. Этот процесс иллюстрируется на рис. 12.12.

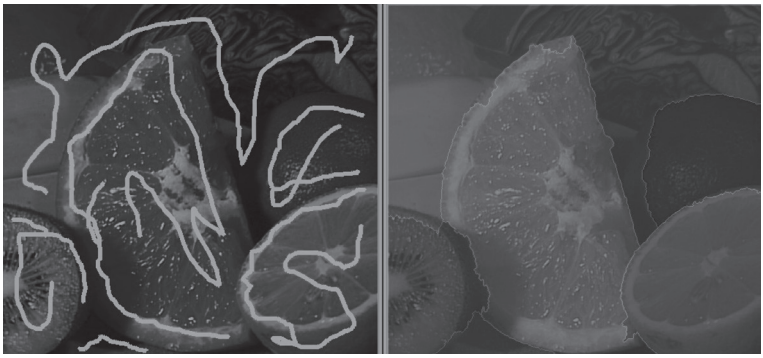

Рис. 12.12 ❖ Алгоритм водораздела:
после того как пользователь пометил объекты, которые не должны быть разделены (слева), алгоритм строит из помеченных областей сегменты (справа)

Алгоритм водораздела реализуется следующей функцией:

```
void cv::watershed(
  cv::InputArray       image,          // 8-разрядное трехканальное входное изображение
  cv::InputOutputArray markers         // одноканальное 32-разрядное с плавающей точкой
);
```

Здесь входное изображение `image` должно быть 8-разрядным трехканальным (цветным), а `markers` – одноканальным целым (типа `CV::S32`) того же размера. На входе все элементы `markers` равны `0`, *за исключением* тех, в которые вызывающая программа записала положительные числа, показывающие, что некоторые области составляют единое целое. Например, на рис. 12.12 слева апельсин мог быть помечен числом 1, лимон – числом 2, лайм – числом 3, фон в верхней части – числом 4 и т. д.

После завершения работы алгоритма все пиксели `markers`, которые были равны 0, принимают значение одного из маркеров (т. е. мы надеемся, что пиксели апельсина станут равны 1, пиксели лимона – 2 и т. д.), за исключением граничных пикселей, которым будет присвоено значение –1. На рис. 12.2 справа показан результат сегментации.

Есть искушение предположить, что все области будут разделены границами, в которых пиксели равны –1. Но это не так. В частности, если два соседних пикселя имели ненулевые и разные значения, то они останутся соседними и не будут разделены пикселем –1.

Алгоритм GrabCut

Алгоритм GrabCut, предложенный в работе Ротера, Колмогорова и Блейка [Rother04], обобщает алгоритм разрезания графа (GraphCut) [Boykov01] на задачу сегментации изображения, управляемую пользователем. Этот алгоритм может порождать отличную сегментацию, зачастую не имея ничего, кроме прямоугольника, ограничивающего сегментируемый объект на переднем плане.

В исходном алгоритме разрезания графа использовались помеченные пользователем области переднего и заднего плана, с помощью которых строились гистограммы распределения для этих двух классов областей. Далее предполагалось, что непомеченные области переднего и заднего плана должны иметь похожее распределение, а также что эти области, скорее всего, гладкие и связные (т. е. представляют собой набор пятен). Затем строился *функционал энергии*, который принимал минимальные значения (стоимость) на решениях, удовлетворяющих этим предположениям, и максимальные – на решениях, которые им не удовлетворяют. Окончательный результат получался путем минимизации функционала энергии[1].

Алгоритм GrabCut развивает GraphCut в нескольких направлениях. Во-первых, гистограмма заменяется другой моделью (гауссовой смесью), что позволяет обрабатывать цветные изображения. Во-вторых, задача минимизации функционала энергии решается итеративно, что дает в целом лучшие результаты и существенно расширяет возможности пометки пользователем. Допустима даже односторонняя пометка, когда помечаются пиксели только переднего или только заднего плана (в GraphCut необходимо помечать те и другие).

Реализация в OpenCV позволяет вызывающей программе просто задать прямоугольник вокруг сегментируемого объекта, тогда пиксели вне этого прямоугольника считаются принадлежащими заднему плану, а передний план никак не специфицируется. Альтернативно пользователь может задать полную маску, в которой пиксели классифицируются как заведомо переднего плана, заведомо заднего плана, потенциально переднего плана и потенциально заднего плана[2]. В этом случае «заведомые» области используются для классификации остальных. Соответствующая функция называется `cv::grabCut()`:

```
void cv::grabCut(
  cv::InputArray       img,
  cv::InputOutputArray mask,
  cv::Rect             rect,
  cv::InputOutputArray bgdModel,
  cv::InputOutputArray fgdModel,
  int                  iterCount,
  int                  mode = cv::GC_EVAL
);
```

Функция вычисляет разметку входного изображения `img` и сохраняет ее в выходном массиве `mask`. Массив `mask` можно использовать также как входной, этот режим задается аргументом `mode`. Если в `mode` поднят флаг `cv::GC_INIT_WITH_MASK`[3], то заданные в маске значения используются для инициализации разметки изображения.

---

[1] Задача его минимизации не тривиальна. На практике она решается методом *Mincut* (алгоритм нахождения минимального разреза), от которого получили названия алгоритмы GraphCut и GrabCut.

[2] Как ни странно, в реализации алгоритма не предусмотрена априорная метка «не знаю».

[3] На самом деле реализация `cv::grabCut()` устроена так, что явно задавать флаг `cv::GC_INIT_WITH_MASK` не нужно, поскольку инициализация по маске – поведение по умолчанию. То есть если не задан флаг `cv::GC_INIT_WITH_RECT`, то инициализация по маске производится. Однако это задается не в значении аргумента по умолчанию, а в самой логике функции, и, значит, не гарантируется, что такое поведение сохранится в будущем. Поэтому лучше задавать один из флагов `cv::GC_INIT_WITH_MASK` или `cv::GC_INIT_WITH_RECT` явно; это не только защищает от будущих изменений, но и делает код понятнее.

Маска должна быть одноканальным изображением типа `cv::U8`, каждый элемент которого принимает одно из следующих значений.

Константа	Числовое значение	Описание
`cv::GC_BGD`	0	Заведомо заднего плана
`cv::GC_FGD`	1	Заведомо переднего плана
`cv::PR_GC_BGD`	2	Потенциально заднего плана
`cv::PR_GC_FGD`	3	Потенциально переднего плана

Аргумент `rect` используется, только если не производится инициализация по маске. Если в `mode` поднят флаг `cv::GC_INIT_WITH_RECT`, то вся область вне этого прямоугольника считается «заведомо задним планом», а область внутри него – «потенциально передним планом».

Следующие два массива используются как временные буферы. При первом вызове `cv::grabCut()` они могут быть пустыми. Но если по какой-то причине нужно будет выполнить несколько итераций алгоритма GrabCut (возможно, разрешив пользователю в промежутке указать дополнительные «заведомые» пиксели), то для продолжения нужно будет указать те массивы (и маску), которые были заполнены в ходе предыдущего вызова (немодифицированные).

На внутреннем уровне алгоритм GrabCut несколько раз вызывает алгоритм Graph-Cut (с небольшими модификациями, упомянутыми выше). Аргумент `itercount` определяет число таких итераций. Обычно задается 10 или 12 итераций, хотя точное число может зависеть от размера и характера обрабатываемого изображения.

Сегментация методом сдвига среднего

Этот метод сегментации ищет пики в распределении цветов [Comaniciu99]. Он тесно связан с *алгоритмом сдвига среднего*, который мы обсудим в главе 17, когда будем говорить о сопровождении объектов. Основное различие между ними в том, что первый анализирует пространственное распределение цветов (и, значит, относится к теме сегментации), а второй прослеживает это распределение на протяжении некоторого времени в последовательных кадрах. Соответствующая функция сегментации называется `cv::pyrMeanShiftFiltering()`.

Имея множество точек в многомерном пространстве (с измерениями *x, y*, синий, зеленый, красный), метод сдвига среднего ищет «скопления» данных с наибольшей плотностью, перемещая по пространству *окно*. Заметим, однако, что области значений пространственных и цветовых переменных могут сильно различаться. Поэтому должна быть возможность задавать разные радиусы окна по разным измерениям. В данном случае нам нужен как минимум один радиус для пространственных переменных (`spatialRadius`) и еще один для цветовых (`colorRadius`). По мере перемещения окна сдвига среднего все пройденные им точки, сходящиеся к некоторому пику данных, оказываются связаны с этим пиком, или «принадлежат» пику. Это отношение владения, расходящееся от самых плотных пиков, и образует сегментацию изображения. На самом деле сегментации подвергается пирамида изображений (см. описание функций `cv::pyrUp()`, `cv::pyrDown()` в главе 11), так что границы цветовых кластеров, найденных на верхнем уровне пирамиды (в уменьшенном изображении), уточняются на более низких.

На выходе алгоритма сегментации сдвигом среднего получается новое «плакатное» изображение, в котором детальная текстура удалена, а цветовые градиенты по

большей части спрямлены. Такое изображение можно подвергнуть дальнейшей сегментации с помощью наиболее подходящего алгоритма (например, `cv::Canny()` в сочетании с `cv::findContours()`, если конечной целью является выделение контуров).

Алгоритм реализуется функцией `cv::pyrMeanShiftFiltering()`:

```
void cv::pyrMeanShiftFiltering(
  cv::InputArray   src,               // 8-разрядное, 3-канальное изображение
  cv::OutputArray  dst,               // 8-разрядное, 3-канальное изображение того же размера, что src
  cv::double       sp,                // пространственный радиус окна
  cv::double       sr,                // цветовой радиус окна
  int              maxLevel = 1,      // максимальный уровень пирамиды
  cv::TermCriteria termcrit = TermCriteria(
    cv::TermCriteria::MAX_ITER | cv::TermCriteria::EPS,
    5,
    1
  )
);
```

Вход `src` и выход `dst` должны быть 8-разрядными трехканальными (цветными) изображениями одинакового размера. Аргументы `spatialRadius` и `colorRadius` определяют способ совместного усреднения по пространственным и цветовым координатам. Для цветных изображений размера 640×480 рекомендуется задавать `spatialRadius = 2`, `colorRadius = 40`. Аргумент `maxLevel` задает число уровней пирамиды изображений. Для цветных изображений размера 640×480 хорошие результаты дает значение 2 или 3.

С параметром `cv::TermCriteria` мы уже встречались при описании алгоритмов. Критерий остановки задается для всех итеративных алгоритмов в OpenCV. Функция сегментации методом сдвига среднего дает хорошие результаты, если не задавать этот параметр вообще, оставив значение по умолчанию.

На рис. 12.13 показан пример сегментации с такими параметрами:

```
cv::pyrMeanShiftFiltering( src, dst, 20, 40, 2);
```

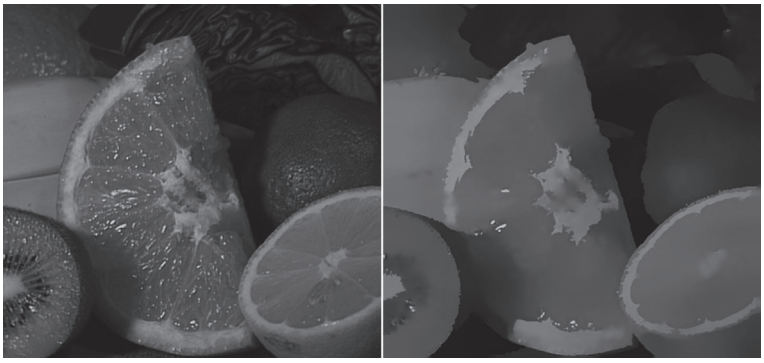

Рис. 12.13 ❖ Сегментация методом сдвига среднего с помощью функции `cv::pyrMeanShiftFiltering()` с параметрами maxLevel=2, spatialRadius=20, colorRadius=40; похожим областям присвоены одинаковые значения, и теперь их можно рассматривать как суперпиксели (большие статистически однородные области), что может заметно ускорить последующую обработку

Резюме

В этой главе мы расширили арсенал методов анализа изображений. Опираясь на преобразования изображений общего вида из предыдущей главы, мы изучили методы, которые позволяют лучше понять изображение и, как мы увидим, образуют фундамент более сложных алгоритмов. Такие инструменты, как дистанционное преобразование, интегральные изображения и различные методы сегментации, оказываются важными строительными блоками для других алгоритмов OpenCV и ваших собственных.

Упражнения

1. В этом упражнении мы поэкспериментируем с параметрами `lowThresh` и `highThresh` функции `cv::Canny()`. Загрузите изображение с интересными линейными структурами. Мы будем использовать три отношения порогов high:low: 1.5:1, 2.75:1, 4:1.
   a. Что вы наблюдаете, когда верхний порог меньше 50?
   b. Что вы наблюдаете, когда верхний порог от 50 до 100?
   c. Что вы наблюдаете, когда верхний порог от 100 до 150?
   d. Что вы наблюдаете, когда верхний порог от 150 до 200?
   e. Что вы наблюдаете, когда верхний порог от 200 до 250?
   f. Подведите итог своим наблюдениям и постарайтесь объяснить результаты.
2. Загрузите изображение, на котором есть четкие прямые и окружности, например вид велосипеда сбоку. Примените преобразования Хафа для поиска прямых и окружностей и посмотрите, какие результаты получаются для вашего изображения.
3. Можете ли вы придумать, как использовать преобразование Хафа для идентификации фигур любого вида с отчетливым периметром?
4. Вычислите преобразования Фурье небольшого нормального распределения и изображения. Перемножьте их и вычислите обратное преобразование Фурье результата. Чего вы добились? Вы обнаружите, что по мере увеличения размера фильтра вычисления в частотной области занимают гораздо меньше времени, чем в пространственной.
5. Загрузите какое-нибудь интересное изображение, преобразуйте его в полутоновое и вычислите интегральное изображение. Теперь воспользуйтесь свойствами интегрального изображения для нахождения горизонтальных и вертикальных прямых в исходном изображении.

   
   Используйте длинные вытянутые прямоугольники, складывайте и вычитайте их на месте.

6. Напишите функцию, которая вычисляет интегральное изображение, повернутое на 45 градусов. Ее можно будет использовать для нахождения суммы по повернутому на 45 градусов прямоугольнику, заданному четырьмя точками.
7. Объясните, как использовать дистанционное преобразование, чтобы автоматически совместить известную фигуру с тестовой, если масштаб известен и остается неизменным. А как это сделать для нескольких масштабов?

8. Напишите функцию, которая размывает изображение, вызывая `cv::GaussianBlur()` с ядром размера 50×50. Измерьте время ее работы. Примените ДПФ гауссова ядра 50×50, чтобы выполнить такое же размытие гораздо быстрее.
9. Напишите функцию, которая будет интерактивно удалять людей из изображения. Сначала вызовите `cv::grabCut()` для сегментации человека, а затем `cv::inpaint()` для заполнения образовавшейся дырки (функция `cv::inpaint()` рассматривалась в предыдущей главе).
10. Возьмите какое-нибудь интересное изображение. Сегментируйте его с помощью функции `cv::pyrMeanShiftFiltering()`. Примените `cv::floodFill()` для маскирования двух из получившихся сегментов, а затем воспользуйтесь этой маской, чтобы размыть все изображение, кроме этих сегментов.
11. Создайте изображение, содержащее квадрат 20×20. Поверните его на произвольный угол. Примените к нему дистанционное преобразование. Создайте изображение квадрата 20×20. Пользуясь получившимся в результате дистанционного преобразования изображением, наложите этот квадрат на повернутый квадрат в созданном ранее изображении.
12. В соревновании автомобилей-роботов 2005 DARPA Grand Challenge авторы из Стэнфордского университета использовали разновидность алгоритма кластеризации по цветам, чтобы отделить дорогу от бездорожья. Цвета выбирались из лазерного снимка трапецеидального участка дороги перед машиной. Все присутствующие в изображении цвета, близкие к цвету этого участка, – и при условии, что они попадали в ту же связную компоненту, что и исходная трапеция, – помечались как дорога. На рис. 12.14 показан результат применения алгоритма водораздела для сегментации дороги после того, как часть изображения внутри дороги была помечена знаком трапеции, а часть вне нее – инвертированным символом «U». Какими неприятностями могло грозить применение этого метода сегментации дороги?

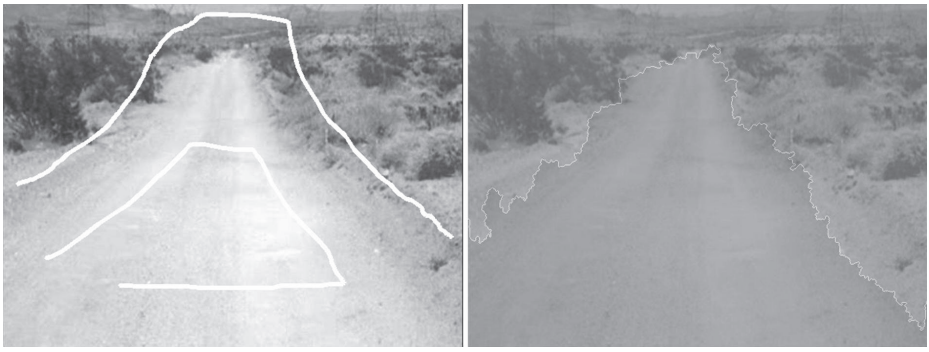

Рис. 12.14 ❖ Использование алгоритма водораздела для выделения дороги: в исходном изображении поставлены маркеры (слева), а алгоритм порождает сегментированную дорогу (справа)