Разработка алгоритма обнаружения движения в среде программирования *Mathematica*

Вадим Залива
lord@crocodile.org

Аннотация

В данной статье мы рассмотрим пример использования среды *Mathematica* для быстрого прототипирования простого алгоритма обнаружения движения. Кроме общих приёмов работы с *Mathematica*, мы познакомим читателя с некоторыми понятиями из области машинного зрения, цифровой обработки изображении и сигналов. При разработке мы применим некоторые из подходов функционального программирования, поддерживаемых *Mathematica*.

In this article, we will use Mathematica to develop a prototype of a simple motion detection algorithm. We will introduce the basics of using Mathematica environment and use this exercise to illustrate some basic concepts from the fields of machine vision, digital image processing and signal processing. We will be using some of the functional programming capabilities provided by Mathematica.

Обсуждение статьи ведется по адресу:
http://fprog.ru/2011/issue7/vadim-zaliva-motion-detection/discuss/.

1.1. Задача

Недавно передо мной встала задача разработать программу, которая позволяет использовать мобильный телефон (iPhone) в качестве системы сигнализации. Телефон при помощи видеокамеры наблюдает за помещением и, если обнаружено движение, подает звуковой сигнал, фотографирует движущийся объект и передает изображение в Интернет. Ключевая часть этой программы — это алгоритм определения движения. Конечно, можно было бы его сразу написать его на Objective-C и производить отладку на телефоне, но такой процесс был бы довольно-таки трудоемким. Вместо этого мы решили использовать среду программирования *Mathematica*[1] для проверки и отладки алгоритма, который затем будет переписан на Objective-C.

Пакет *Mathematica* — продукт фирмы *Wolfram Research* — является мощным вычислительным инструментом, используемым в академической и инженерной среде. Работа обычно ведется в *записных книжках* — интерактвных документах содержащих формулы, графики, данные и текст. Они обычно сохраняются в файлах с расширением *.nb*. Код выполняется по мере его ввода и результаты выводятся попеременно с кодом. Это чрезвычайно удобный режим для интерактивной работы. Сложные функции можно оформить в виде отдельных *модулей* (файлов с расширением *.m*), на которые можно ссылаться из записных книжек.

Пакет *Mathematica* умеет работать с видеокамерой, но поскольку нашей целью было быстрое прототипирование алгоритма, то мы не воспользовались этой возможностью, а работали с заранее записанным видеофайлом. Также, поскольку этот алгоритм планировалось в дальнейшем реализовывать на другом языке, мы старались применять простейшие базовые функции, а не использовать более мощные встроенные функции пакета. Так, например, мы не воспользовались стандартными функциями работы с изображениями для масштабирования кадров и перевода их в серое цветовое пространство (grayscale).

Первая рабочая версия данного алгоритма, включая запись тестового видеоролика (созданную при помощи того же iPhone), была разработана примерно за 2 часа. Это демонстрирует удобство пакета *Mathematica* для быстрого решения подобных задач. В дальнейшем алгоритм был реализован и использован в программе iSentry для iPhone[2].

Для начала мы записали тестовый видеоролик, на котором производилась отладка алгоритма. Это шестисекундный ролик, на котором на столе под лампой стоит детская игрушка (динозавр). Через некоторое время в кадре появляется iPhone, брошенный на стол. Еще через некоторое время в кадре появляется рука, которая двигает игрушку. Ролик можно скачать отдельно[3].

1.2. Алгоритм

Работа с видео и картинками

Запустим пакет *Mathematica* и создадим пустую *записную книжку*. Допустим, что ролик сохранён в файле с именем *movement.mov* и находится в той же директории, что и наша *записная книжка*. Теперь мы можем набрать нашу первую строку кода на *Mathematica*:

```
> raw = Import[ToFilename[NotebookDirectory[], "movement.mov"],
    "Data"];
```

[1] Wolfram Research, Inc., Mathematica, Version 8.0, Champaign, IL (2010).
[2] http://itunes.apple.com/us/app/isentry/id396777365
[3] http://www.crocodile.org/lord/MotionDetection/movement.mov

Синтаксис пакета *Mathematica* использует для указания аргументов функций квадратные, а не круглые скобки. Функция *NotebookDirectory* возвращает полный путь каталога, в котором расположена текущая *записная книжка*. Параметр *ToFileName* формирует абсолютный путь из каталога и имени файла. Обычно функция *Import* сама пытается «угадать» формат файла, но можно его указать и явно. Для каждого поддерживаемого типа файлов эта функция умеет загружать разные элементы. В данном случае нас интересуют данные в виде массивов чисел, что мы и указываем при помощи второго параметра *«Data»*. Результат присваивается переменной с именем *raw*. Точка с запятой после выражения подавляет вывод результата выполнения непосредственно под строкой с выражением. Результатом выполнения функции будет список кадров.

Чтобы узнать количество кадров в прочитанном файле, можно воспользоваться функцией *Length*. Результатом работы функции будет количество кадров (элементов списка), сохранённое в переменной *nframes*.

```
> nframes = Length[raw]
196
```

В случае вложенных структур данных *Length* считает количество элементов на самом верхнем уровне. В данном случае получилось 196 кадров. Если же мы захотим получить все измерения (предполагая что, имеет место регулярная многомерная структура), то можно воспользоваться функцией *Dimensions*:

```
> Dimensions[raw]
{196, 240, 320, 3}
```

Теперь рассмотрим подробнее, как представлены кадры. Базовая структура данных — это список. В данном случае у нас есть список из 196 элементов (кадров). Каждый кадр — это матрица размером 240 строк по 320 элементов каждая. В *Mathematica*, в отличие от языков типа R, нет специальной структуры данных для матриц — они представлены как вложенные списки. Каждый элемент представляет собой список из трёх значений, соответствующих RGB-компонентам цветовой модели. Каждая компонента может принимать значения от 0 до 255. Чтобы узнать значение цвета первого пикселя в первой строке первого кадра, достаточно написать следующее выражение (поскольку квадратные скобки используются в нотации вызова функций, для доступа к элементам массивов используются двойные квадратные скобки):

```
> raw[[1,1,1]]
{42, 33, 20}
```

В *Mathematica* есть встроенный тип данных *Image*, который мы будем использовать для визуализации данных. Все, что нам нужно знать об этом типе — это то, что можно его создать из матрицы значений пикселей (первый кадр) с помощью следующей команды:

```
> Image[raw[[1]], "Byte"]
```

Перевод в серую шкалу

Для определения движения информация о цвете нам не нужна, и поэтому нам будет гораздо проще работать не с цветной картинкой, представленной 3-мя компонентами, а с изображением в серой шкале, где цвет представлен одним значением. Для перевода RGB-цвета в серую шкалу мы воспользуемся известной формулой:

$$I = 0.3 \times R + 0.59 \times G + 0.11 \times B$$

Эта формула основана на чувствительности человеческого глаза к различным частям спектра [1]. Поскольку в нашем случае картинки будет обрабатывать программа, а не человеческий глаз, то точные значения не так важны, и поэтому можно было бы использовать 33% для каждого компонента.

Преобразование всех кадров в серую шкалу выполняется следующей командой:

```
> orig = Map[(# . {0.3,0.59,0.11})&, raw, {3}];
```

Мы используем функцию *Map*, которая применяет указанное выражение к каждому элементу структуры данных. В функциональных языках обычно эта функция определена для списков и применяет выражение к элементам списка первого уровня. В *Mathematica* она также позволяет работать с вложенными списками произвольной глубины. Функция *Map* имеет три параметра. Первый параметр — это *функция*. Второй параметр — это список произвольной вложенности. Третий, опциональный параметр указывает, к каким уровням вложенности списка применять указанную функцию. В нашем случае его значение *{3}* обозначает, что применять функцию нужно к элементам 3-го уровня вложенности. Напомним, что в нашей структуре данных первый уровень — это кадры, второй — строки кадра, а третий — пиксели в строке. То есть, функция будет применена к каждому пикселю, который будет ей передан как список из трёх элементов. Поскольку у нас нет отдельной функции для перевода RGB в серую шкалу, то первым параметром *Map* мы передаем то, что в *Mathematica* называется *pure function*, а в других языках известно как *анонимная функция*, или *лямбда-выражение*. В нашем примере она определена при помощи оператора *'&'*. Выражение перед *'&'* является телом функции. На формальные параметры можно ссылаться, используя имена *#1, #2, #3* и так далее. Для удобства *#1* можно сокращать до *#*. Вычисления, которые мы делаем, довольно просты: список компонентов цвета для каждого пикселя мы поэлементно умножаем на коэффициенты {0.3,0.59,0.11} и суммируем то, что получилось (используя скалярное произведение векторов). Результатом работы всего

выражения будет список кадров, в котором каждый элемент представлен уже не тройкой RGB, а одним значением оттенка серого цвета.

Image умеет работать и с такими изображениями, и мы можем вывести первый кадр, чтобы взглянуть на результат:

```
> Image[orig[[1]], "Byte"]
```

Масштабирование

Исходные кадры имели размер 320x240 пикселей. Для нашей задачи даже такое довольно-таки низкое разрешение излишне. Кроме того, поскольку наше приложение будет работать на мобильном телефоне, где процессорные ресурсы ограничены и использование CPU связано с расходом батарейки, то имеет смысл уменьшить кадры, по крайней мере, в два раза по каждому измерению, что уменьшит количество пикселей в четыре раза. В принципе, нам не обязательно масштабировать пропорционально, но для простоты мы будем использовать пропорциональное масштабирование, так что фактор масштабирования у нас будет одинаков для обоих измерений:

```
> scaleFactor = 2;
```

Процесс масштабирования картинки (с уменьшением размера) на множитель *scaleFactor* называется *downsamping*. Обычно этот процесс делается в 2 этапа:

1) Применение фильтра нижних частот (low-pass filter) для удовлетворения теоремы Котельникова [2] (в англоязычной литературе известной как *Nyquist–Shannon sampling theorem*);
2) Уменьшение количества пикселей выбором элементов с шагом *scaleFactor*.

Рассмотрим эти шаги подробнее. Не вдаваясь в теорию информации, достаточно сказать, что на шаге 1 мы применим так называемый усредняющий фильтр. Интуитивное определение этого фильтра довольно простое: каждая точка будет заменена на среднее значение ее окрестности с заданным радиусом. В более общем случае, определение значения точки через ее окрестность — довольно частая операция в обработке изображений и обычно выражается через *свертку*. В функциональном анализе свертка двух функций f и g определяется как:

$$[f * g](x) = \int_{-\infty}^{\infty} f(u)g(x-u)du$$

Для двухмерных растровых изображений мы можем записать дискретную форму этой формулы:

$$F(x,y) = \sum_i \sum_j f(i,j)g(x-i, y-j)$$

В этой формуле *f(x,y)* — это значения пикселей изображения до применения свертки. *F(x,y)* это функция, которая дает значения после применения свертки. Собственно свертка задается функцией *g(x,y)*. Для большинства практических применений функция g опеределена на небольшом диапазоне значений (обычно в несколько пикселей). Ее удобно представлять с помощью матрицы, которую иногда называют *ядром свертки*. Применение такой операции свертки к изображенииям очень хорошо распараллеливается и часто выполняется при помощи GPU.

Теперь представьте, что мы возьмем матрицу размером $(2 * \text{scaleFactor} + 1) \times (2 * \text{scaleFactor} + 1)$. В нашем случае в ней будет $(2 * \text{scaleFactor} + 1)^2$ элементов. Установим каждый элемент этой матрицы равным $\frac{1}{(2*\text{scaleFactor}+1)^2}$.

```
> meanKernel = BoxMatrix[scaleFactor]/((2*scaleFactor + 1)^2);
> MatrixForm[meanKernel]
```

$$\begin{pmatrix} \frac{1}{25} & \frac{1}{25} & \frac{1}{25} & \frac{1}{25} & \frac{1}{25} \\ \frac{1}{25} & \frac{1}{25} & \frac{1}{25} & \frac{1}{25} & \frac{1}{25} \\ \frac{1}{25} & \frac{1}{25} & \frac{1}{25} & \frac{1}{25} & \frac{1}{25} \\ \frac{1}{25} & \frac{1}{25} & \frac{1}{25} & \frac{1}{25} & \frac{1}{25} \\ \frac{1}{25} & \frac{1}{25} & \frac{1}{25} & \frac{1}{25} & \frac{1}{25} \end{pmatrix}$$

Такую матрицу можно получить, выполнив следующую команду:

```
> meanKernel = BoxMatrix[scaleFactor]/((2*scaleFactor+1)^2);
> MatrixForm[meanKernel]
```

Это и есть наше ядро свертки для усредняющего фильтра с радиусом *scaleFactor*. Если его применить к каждому пикселю изображения, то мы получим среднее значение из окрестности 25-ти соседних пикселей, включая текущий.

Функция *ListConvolve* позволяет применить ядро к списку значений. Количество измерений списка и ядра могут быть произвольными, но должны совпадать. Дополнительный параметр уточняет, как обрабатывать значения около краев, когда применяемый фильтр выходит за границы данных. Детальное описание опций *ListConvolve* можно посмотреть в документации по *Mathematica*. Используя уже знакомую нам функцию *Map*, можно применить *mean* фильтр ко всем кадрам:

```
> origc = Map[ListConvolve[meanKernel, #, {-1, -1}, 0]&, orig];
```

Как и ожидалось, результирующая сглаженная картинка выглядит «размытой». Не зря *mean* фильтр еще называют сглаживающим фильтром.

```
> Image[origc[[1]], "Byte"]
```

Теперь можно выполнить второй шаг и выбрать пиксели с шагом *scaleFactor*. Выбор элементов из списка можно сделать при помощи функции *Take*. В нашем случае мы указываем ей два критерия выбора — по одному на каждое измерение. Критерий задан в виде списка вида *{начальное_значение, конечное_значение, шаг}*.

```
> origdim = Dimensions[orig]
{196, 240, 320}
> bwm = Map[Take[#, {1, origdim[[2]], scaleFactor}, {1, origdim[[3]],
    scaleFactor}]&, origc];
```

В результате получаем массив кадров уменьшенного размера:

```
> Dimensions[bwm]
{196, 120, 160}
```

Давайте посмотрим, что у нас получилось. На этот раз мы не просто визуализируем изображение одного из кадров, а применим функцию под названием *Manipulate*. Это один из очень удобных и мощных инструментов в *Mathematica*, позволяющий визуализировать данные, динамически изменяя значения параметров. Опять же, мы не будем вдаваться во все возможности и способы использования этого механизма, а покажем его простейший вариант, который очень уместен в нашем случае:

```
> Manipulate[Image[bwm[[f]], "Byte"], {f, 1, nframes, 1}]
```

Первым параметром передается выражение. В нашем случае оно преобразует кадр с номером *f* из списка *bwm* в формат *Image*. Заметьте, что в отличие от *Map*, это не функция, а просто фрагмент кода, и, соответственно, мы не используем оператор '*&*'. Второй параметр — это список, состоящий из имени переменной, начального значения, конечного значения и шага. Функция *Manipulate* позволяет пользователю изменять значения переменной, и при каждом изменении пересчитывает выражение с новыми значениями, показывая результат. Можно изменять более одного значения, но в нашем случае нам достаточно номера кадра. Двигая ручку *f*, пользователь может покадрово прокручивать получившееся видео. Можно также перейти к произвольному кадру, введя его номер, или включить механизм автоматической прокрутки (анимацию).

Мы закончили предварительную обработку данных, и теперь готовы перейти собственно к алгоритму определения движения.

Оценка количества движения

Как мы будем определять движение? Простейший подход — смотреть, насколько изменилось изображение между кадрами. Просто сравнивать кадр с предыдущим будет недостаточно, так как программа будет чувствительна к мелким вариациям освещения и цифровому «шуму» камеры. Более надежный подход — сравнивать текущий кадр с усредненным значением нескольких предыдущих. Это будет довольно легко реализовать и при работе в режиме реального времени, накапливая несколько последних кадров в специальном буфере и используя их для сравнения. Такое сглаживание кадра по нескольким предыдущим называется частным случаем *каузального фильтра (causal filter)*. Название подчеркивает, что его значение зависит только от текущего и предыдущих кадров, но не от будущих.

Для начала определим функцию, которая берет несколько кадров и строит на основании их «средний» кадр. Определение довольно-таки очевидное:

```
> avgFrame[d_] := Total[d]/Length[d];
```

Список кадров указан формальным параметром *d_* . В *Mathematica* существует довольно мощный механизм выбора подходящей функции на основании типов и количества параметров. Формальные параметры задаются в виде *шаблонов (patterns)*. Все, что нужно знать на данном этапе — это то, что в определении функции имена формальных параметров должны содержать символ подчеркивания («_»), а в теле функции на них нужно ссылаться без

этого символа. Нашей функции *avgFrame* передается на вход список кадров в виде матриц. Функция *Total* их суммирует поэлементно. Получившуюся матрицу мы делим на количество кадров, тем самым получив на выходе матрицу средних значений пикселей заданных кадров.

Теперь можно определить функцию, реализующую сглаживающий фильтр. На вход передаем список кадров и размер окна сглаживания (количество кадров). На этот раз используем операцию *Map* не по самим кадрам, а по их индексам. Из исходного массива мы выбираем подмножество элеменов, используя двойные квадратные скобки (оператор *Part*). Диапазон выбираемых значений задается в формате *от ;; до* (оператор *Span*).

```
>avgFilter[l_, w_] := Map[avgFrame[
   l[[Max[1,#-w] ;; #]]]&, Range[Length[l]]];
```

Теперь мы можем применить наш фильтр. Значение окна подбирается экспериментально в зависимости от желаемой «чувствительности» алгоритма и частоты получения кадров. Допустим, мы будем использовать значение 10. При частоте 30 кадров в секунду это соответствует одной третьей доле секунды.

```
> bwms = avgFilter[bwm, 10];
```

Таким образом мы получили кадры, которые «сглажены» по времени. Если их визуализировать, то в местах, где нет движения, изображение мало отличается от оригинального, а в местах, где наблюдается движение, движущиеся части немного размыты. Сам по себе этот промежуточный результат малоинтересен. Более интересно взглянуть на разницу между несглаженным по времени кадром (*bwm*) и сглаженным (*bwms*). На статических кадрах разницы не будет, и результатом будет кадр с нулевыми значениями пикселей (черный). На кадрах, где присутствует движение, мы увидим изменившуюся часть. Посмотрим на пару показательных кадров:

В кадре номер 50 телефон движется, «въезжая» в кадр:

```
> Image[bwm[[50]]-bwms[[50]], "Byte"] // ImageAdjust
```

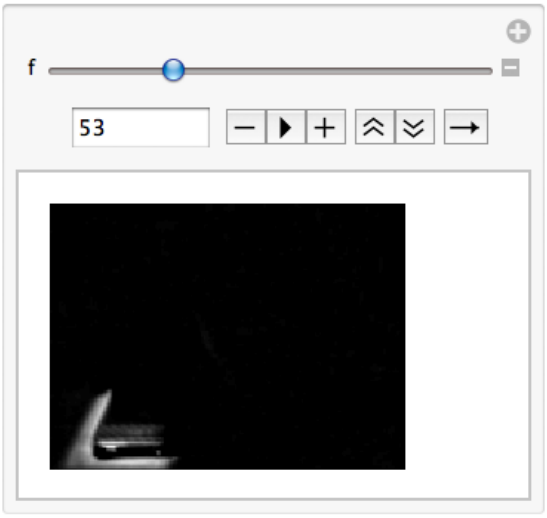
В кадре 139 мы видим движущуюся руку:

```
> Image[bwm[[139]]-bwms[[139]], "Byte"] // ImageAdjust
```

Кроме знакомой функции *Image*, тут использована еще функция *ImageAdjust*. Эта функция масштабирует значения пикселей, чтобы они были в диапазоне от 0 до 1, что позволяет лучше рассмотреть темные кадры. Кроме того, тут использован оператор «//». Это постфикс-нотация применения функций. Запись *F[x] // G* эквивалентна *G[F[x]]*.

И, как обычно, можно воспользоваться функцией *Manipulate*, чтобы посмотреть на произвольные кадры, интерактивно прокручивая список.

```
> Manipulate[ImageAdjust[Image[bwm[[f]]-bwms[[f]], "Byte"],
     {f, 1, nframes, 1}]]
```



Попробуем представить объем изменений в виде одного числа. Количество движения в указаном кадре корреллирует с тем, насколько текущий кадр (из списка *bwm*) отличается от усредненного значения нескольких предыдущих (из списка *bwms*). Оценить разницу этих двух кадров удобнее всего с помощью среднеквадратичного отклонения разницы соответствующих пикселей, поделенного на среднее значение пикселя в текущем кадре:

$$\mathrm{CV}(\mathrm{RMSD}(x,y)) = \frac{\mathrm{RMSD}(x,y)}{\bar{x}} = \frac{\sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - y_i)^2}}{\bar{x}}$$

На *Mathematica* это можно выразить вот так:

```
> errs = Map[RootMeanSquare[Flatten[bwm[[#]]-bwms[[#]]]] /
                Mean[Flatten[bwm[[#]]]],
              Range[nframes]];
```

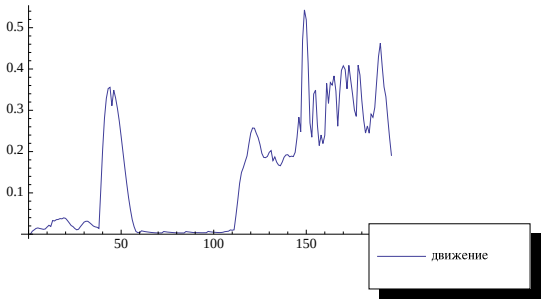
Заметим, что мы уменьшаем размерность данных при вызове функции *Mean*, используя функцию *Flatten*. В противном случае мы получим не одно среднее значение по всем пикселям кадра, а вектор средних значений столбцов.

В переменной *errs* у нас должен был получится список значений, отражающих количество движения в каждом кадре. Попробуем нарисовать график этих значений. Команда *Needs* позволяет подгружать внешние модули. В данном случае мы используем модуль *PlotLegends*, позволяющий показывать на графике легенду.

```
> Needs["PlotLegends`"]
```

Собственно рисование графика выполняется функцией *ListLinePlot*.

```
> ListLinePlot[errs, PlotLegend -> {"движение"}, LegendPosition ->
    {0.8, -0.8}]
```



Можно заметить два всплеска активности: первый в диапазоне кадров 40—60, а второй — начиная с кадра 110.

Сглаживание сигнала при помощи *линейной регрессии*

Используя получившийся график, уже можно попробовать обнаружить движение, но для начала имеет смысл его немного сгладить, чтобы уменьшить влияние случайного шума. Сглаживать будем опять же при помощи *каузального фильтра*. На этот раз применим для сглаживания линейную регрессию. Идея довольно проста — в качестве сглаженного значения точки будем использовать ее регрессию по нескольким предыдущим точкам. Для этого постараемся найти прямую, которая ближе всего описывает предыдущие точки. Искомое значение — точка на этой прямой с абсциссой, соответствующей текущему (последнему) значению.

Рассмотрим пример, взяв произвольный диапазон значений из списка *errs*. Допустим, что диапазон задан начальным и конечным значениями f и t.

```
> f = 3; t = 13;
```

Сформируем список точек, заданных парой координат. Первая координата — порядковый номер. Вторая — собственно значение. Список первых координат легко получить при помощи функции *Range*, список вторых — используя оператор *Span (;;)* из списка *errs*. Их можно поместить в список из двух элементов, используя фигурные скобки. Если рассматривать получившуюся структуру как матрицу, то ее размерность будет 2 строки и 11 столбцов. Транспонировав ее при помощи функции *Transpose*, мы получим нужный нам список пар:

```
> fdata = Transpose[Range[f, t], errs[[f ;; t]]]
{{3,0.0104027},{4,0.0139062},{5,0.0151486},
 {6,0.0138507},{7,0.012893},{8,0.0117535},
 {9,0.0125816},{10,0.0172805},{11,0.0215209},
 {12,0.0187361},{13,0.0330796}}
```

Применим линейную регрессию. Для начала мы это сделаем при помощи функции *Fit*. Первым параметром передадим пары значений. Второй параметр задает тип многочлена, коэффициенты, которого мы собираемся вычислить. Третий перечисляет переменные многочлена. Результатом является выражение, использующее эти переменные.

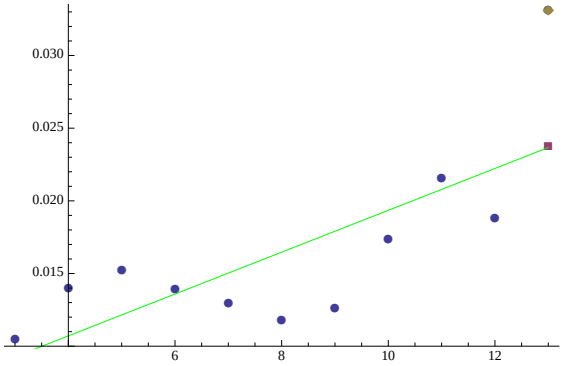
```
> fit = Fit[data, {1, x}, x]
0.00495072 + 0.00143972 x
```

Теперь можно применить это выражение, используя операцию подстановки (*/.*). Эта операция применяет указанные правила подстановки к заданному выражению. Например, можно посчитать последнюю точку:

```
> lastt = {t, fit /. x -> t}
{13, 0.0236671}
```

Итак, мы можем отобразить наш пример:

```
> Show[{ListPlot[{fdata, {lastt}}, {Last[fdata]}, PlotMarkers ->
    Automatic],
        Plot[fit, {x, f, t}, PlotStyle -> Green]}]
```



Синие точки — исходные данные. Зеленая линия — прямая, которой мы их аппроксимировали. Коричневая точка — несглаженное значение. Красная — сглаженное. Мы не будем рассматривать подробно функции *Plot*, *ListPlot* и *Show*, лишь обратим внимание, что *Show* используется для отображения нескольких графиков в общей системе координат.

Оформим этот механизм в виде функции. Сначала определим функцию *lmsmoothPoint*, которая берет список значений равномерно распределенных точек и на основании их вычисляет сглаженное значение последней. Отметим, что здесь используется несглаженное значение, как часть набора данных для регрессии. Мы допускаем, что в переданном нам наборе данных могут быть неопределенные значения. Мы их просто опускаем, используя функцию *Select* совместно с предикатом *NumberQ*, который возвращает *FALSE* для нечисловых значений, таких как, например, иногда используемый в *Mathematica* символ *Indeterminate*. Если после отбрасывания неопределенных значений у нас останется недостаточно данных для линейной регрессии (меньше двух точек), то функция вернет неопределенное значение:

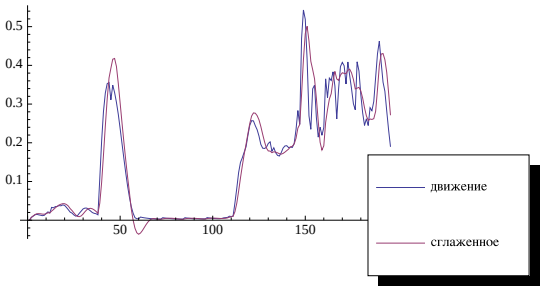
```
> lmsmoothPoint[d_] := Module[{data,rdata},
    data = Transpose[{Range[Length[d]],d}];
    rdata = Select[data,NumberQ[N[#[[2]]]]&];
    If[Length[rdata]<2,
        Last[d],
        Fit[rdata,{1,x},x] /. x-> Length[d]
    ]
]
```

Следующим шагом можно определить функцию, которая принимает набор данных и размер окна сглаживания и применяет сглаживание при помощи линейной регрессии:

```
> lmsmooth[l_,w_] :=
    Map[lmsmoothPoint[l[[Max[1,#-w];;#]]] &, Range[Length[l]]]
```

А теперь применим ее к нашим данным и нарисуем сглаженное и несглаженное значения:

```
> serrs = lmsmooth[errs, 10];
> ListLinePlot[{errs, serrs}, PlotLegend -> {"движение", "сглаженное"},
    LegendPosition -> {0.8, -0.8}]
```



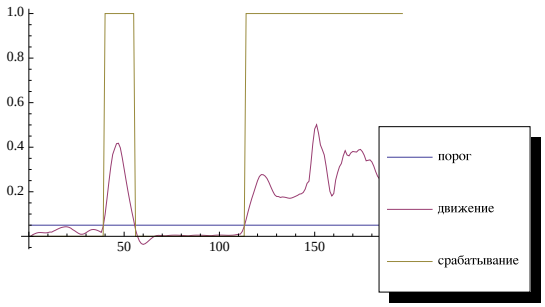
Определение движения

Последний шаг — собственно определение наличия движения в виде логического значения *да/нет*. Мы будем использовать пороговое значение, задаваемое пользователем. Диапазон этого значения будет *[0..1]*. Один из способов получить такие бинарные значения в зависимости от того, что больше — элемент списка или порог — это отнять пороговое значение от элементов списка и применить к ним функцию *UnitStep*. Эта функция возвращает 0 для значений, меньших 0, и 1 для остальных.

```
> motionThreshold = 0.05;
> motionFlag = UnitStep[serrs - motionThreshold];
```

Теперь можно визуализировать окончательный результат работы нашего алгоритма:

```
> ListLinePlot[{Table[motionThreshold, {nframes}], serrs, motionFlag},
     PlotLegend -> {"порог", "движение", "срабатывание"},
     LegendPosition -> {0.8, -0.8}]
```



Заключение

Мы не ставили себе задачи обучить вас языку *Mathematica*. Мы умышленно упростили многие понятия и не вдавались в детали внутреннего представления функций и данных в *Mathematica*. Нашей целью было показать стиль работы с этим пакетом в исследовательской работе на примере разработки этого несложного алгоритма. Надеемся, что заинтересованный читатель самостоятельно сможет изучить множество других возможностей этого пакета, используя материалы, перечисленные в приведенном ниже списке литературы.

Алгоритм, который мы разработали, также был упрощен в учебных целях. В тоже время, даже в таком виде он вполне работоспособен.

Литература

[1] Wikipedia. Grayscale — wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Grayscale&oldid=411295624, 2011. [Online; accessed 27-February-2011].

[2] Wikipedia. Nyquist–shannon sampling theorem — wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Nyquist%E2%80%93Shannon_sampling_theorem&oldid=423361072, 2011. [Online; accessed 14-April-2011].