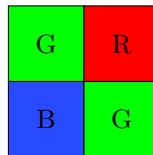


# Линейная интерполяция цветных изображений

Влад Шахуро, Александр Николаев, Нина Тулупцева



Фильтр Байера — это матрица цветных фильтров для фотосенсора фотоаппарата. Каждая ячейка матрицы представляет собой фильтр одного из цветов: R, G или B. Цвет левого верхнего пикселя зависит от модели используемой камеры. Мы будем использовать фильтр со следующим расположением цветов:



Матрица является устройством, воспринимающим спроецированное на него изображение. Вследствие использования фильтров каждый фотоприемник воспринимает лишь 1/3 цветовой информации участка изображения, а 2/3 отсекается фильтром. Недостающие компоненты цвета рассчитываются процессором камеры на основании данных из соседних ячеек в результате интерполяции (demosaicing). В данном задании необходимо научиться восстанавливать полноцветное изображение.

## Критерии оценки

Для удобства выполнения задание разделено на пункты. Максимальная оценка за задание — 5 баллов. Разбиение этих баллов между разными юнит-тестами, можно посмотреть в файле `run.py`, в списке `grade_mapping`. Дополнительные вопросы (выделены цветными прямоугольниками в задании) никак не оцениваются. Для выполнения задания нужно скачать из проверяющей системы:

1. Шаблон решения — архив с файлом `bayer.py`.
2. Архив с тестами. Разархивируйте его в папку `tests` и рядом с этой папкой положите тестовый скрипт `run.py`.
3. Скрипт для тестирования `run.py`. Добавьте ему права на выполнение (`chmod +x run.py` в консоли). Для запуска тестов вам понадобится библиотека `pytest`.
4. Дополнительные файлы для решения. В данном задании это архив с файлом `common.py`, который содержит вспомогательные функции для тестов, и скриптом `calibrate.py` для калибровки ограничений по времени в локальных тестах.

Файлы должны располагаться следующим образом:

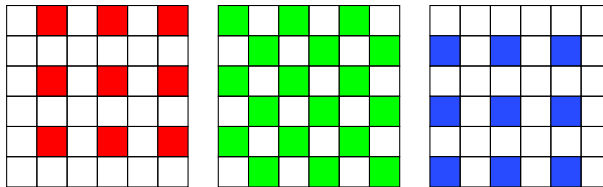
```
|_ bayer.py
|_ common.py
|_ calibrate.py
|_ run.py
|_ tests/
|   |_ 01_unittest_masks_input/
|   |_ 02_unittest_colored_raw_img_input/
|   |_ ...
```

## Калибровка ограничений по времени

Запустите скрипт `calibrate.py`. Он выполнит несколько простейших вычислений на вашем компьютере, замерит время выполнения  $t_{local}$  и запишет в файл `calibration.json` отношение  $t_{system} / t_{local}$ , где  $t_{system}$  — время выполнения тех же операций в тестирующей системе. С помощью этого коэффициента калибруются ограничения по времени в локальных тестах. Соответственно, в тестирующей системе выставлен коэффициент, равный единице.

Обратите внимание, что локальный коэффициент приблизительный. Баллы за задание выставляются по результатам в тестирующей системе.

### 1. Маски байеровского шаблона для трех каналов



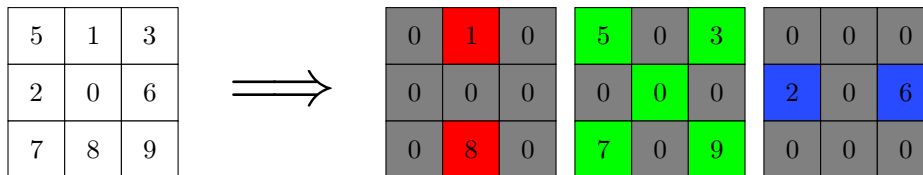
Пример масок по шаблону байера. Цветные клетки соответствуют значению 1, белые — значению 0.

Напишите функцию `get_bayer_masks(n_rows, n_cols)`, которая для заданного размера изображения создает маски для красного, зеленого и синего канала по шаблону байера. Маска — это бинарный массив типа `bool` размером `(n_rows, n_cols, 3)`. Вам могут пригодиться функции `numpy.tile` и `numpy.dstack`. Проверьте реализацию с помощью юнит-тестов:

```
$ ./run.py unittest masks
```

### 2. Цветное изображение с неизвестными значениями

Используя маски для трех каналов, получите из одноканального изображения трехканальное с неизвестными значениями. Например, из одноканального изображения размером `(3, 3)` можно получить трехканальное изображение размером `(3, 3, 3)`.



Серым цветом обозначены неизвестные значения. Обратите внимание, что в зеленом канале в центральной ячейке цвет известен и равен 0. То есть 0 в данном случае означает не отсутствие значения, а само значение.

Напишите функцию `get_colored_img(raw_img)`, выполняющую описанное преобразование, и обратную функцию `get_raw_img(colored_img)`. Проверьте реализацию с помощью юнит-тестов:

```
$ ./run.py unittest colored_raw_img
```

### 3. Билинейная интерполяция

Напишите функцию `bilinear_interpolation(raw_img)`, которая принимает на вход одноканальное изображение. Функция возвращает трехканальное изображение, восстанавливая неизвестные значения следующим образом:

1. Рассматривается окно размером  $3 \times 3$  с центром в данном пикселе.
2. *Неизвестное* значение цветовой компоненты (R, G или B) в данном пикселе вычисляется как среднее всех *известных* значений пикселей данной цветовой компоненты в этом окне.

Поведение функции на границе изображения, где окно  $3 \times 3$  выходит за изображение, выберите на собственное усмотрение, граничные пиксели при тестировании игнорируются. Проверьте функцию с помощью юнит-тестов и тестов на реальных изображениях:

```
$ ./run.py unittest bilinear
$ ./run.py unittest bilinear_img
$ ./run.py unittest bilinear_img_fast
```

Тесты `img` и `img_fast` содержат одни и те же изображения, но `img_fast` имеет более жесткое ограничение по времени. Чтобы получить максимальный балл за задание, нужно пройти оба теста. Для прохождения теста `img_fast` циклы по сторонам изображений должны происходить “внутри” `numpy`. Подробнее про эффективность вычислений в соответствующем [разделе](#).

Шаблон с решением содержит код, который запускает функции для интерполяции на реальном изображении и сохраняет восстановленное изображение. Код отрабатывает при самостоятельном запуске `bayer.py` (т.е. вне скрипта `run.py`).

### 4. Улучшенная линейная интерполяция

Приступим теперь к реализации более сложной версии интерполяции изображения. Прочитайте [статью](#) и реализуйте предложенный в ней алгоритм. В этой статье предлагается рассматривать окно размером  $5 \times 5$  пикселей и для вычисления *неизвестных* значений суммировать с разными коэффициентами *известные* значения из разных цветовых каналов. При реализации обратите внимание на следующие детали:

1. Линейные фильтры в статье ненормированы (т.е. сумма коэффициентов фильтров не равна 1). Чтобы сохранить яркость изображения, нужно поделить все коэффициенты на сумму их значений в данном фильтре.
2. Для избежания переполнений при работе с числами типа `uint8` переведите изображение в более широкий тип.
3. В результате суммирования значений по окрестности могут получаться значения, выходящие за границы диапазона допустимых значений. Поэтому все полученные при помощи интерполяции значения нужно обрезать с помощью функции `numpy.clip`.

Напишите функцию `improved_interpolation(raw_img)`, реализующую улучшенную линейную интерполяцию. Проверьте её с помощью юнит-тестов и тестов на реальных изображениях:

```
$ ./run.py unittest improved
$ ./run.py unittest improved_img
$ ./run.py unittest improved_img_fast
```

## 5. Метрика PSNR

Для численной оценки качества работы алгоритма билинейной интерполяции воспользуемся метрикой PSNR (peak signal-to-noise ratio), которая используется для сравнения изображений:

$$MSE(I_1, I_2) = \frac{1}{C \cdot H \cdot W} \sum_{i,j,c} (I_1(i, j, c) - I_2(i, j, c))^2$$

$$PSNR(I_{pred}, I_{gt}) = 10 \log_{10} \left( \frac{\max(I_{gt})^2}{MSE(I_{pred}, I_{gt})} \right)$$

Здесь  $C, H, W$  — количество каналов, высота и ширина изображений,  $I_{pred}$  — результат работы алгоритма интерполяции, а  $I_{gt}$  — эталонное изображение (ground truth).

Напишите функцию `compute_psnr(img_pred, img_gt)`, вычисляющую данную метрику. Если среднеквадратичное расстояние (MSE) между изображениями равно нулю, то функция должна поднимать исключение `ValueError`. Перед вычислениями сконвертируйте изображения в тип `float64`. Проверьте реализацию с помощью юнит-тестов и тестов на реальных изображениях:

```
$ ./run.py unittest psnr
```

Подсчитайте среднее значение PSNR для билинейной и улучшенной интерполяции и по этим значениям сделайте вывод о том, какой алгоритм работает лучше. Эталонные изображения можно найти в папках с тестами. Коррелируют ли ваши выводы с визуальным качеством изображений?

? Зависит ли значение PSNR от того, в каком диапазоне (например, от 0 до 1 или от 0 до 255) находятся значения пикселей изображений?

? PSNR — стандартная метрика оценки качества изображений. Приведите пример двух очень похожих изображений, на которых метрика будет иметь низкое значение. Предложите собственную метрику качества, которая может адекватно сравнить эти два изображения.

# Эффективность тензорных вычислений на Python

Для успешного выполнения заданий курса нужно уметь писать код, который работает быстро, эффективно используя возможности тензорных библиотек (в первую очередь [NumPy](#), а также [PyTorch](#), [CuPy](#) и других). В разделе описана методология написания эффективного кода на Python с использованием таких библиотек.

В данном контексте оптимизация относится к реализации некоего алгоритма в коде. Ускоряется уже написанный код, соответствующий алгоритму, а сам алгоритм не меняется. Соответственно, в этом разделе предполагается, что:

- Вы определились с выбором алгоритма. То есть у вас есть алгоритм для поставленной задачи и вас устраивает его [асимптотическая сложность](#) — например, потому что алгоритм с лучшей асимптотической сложностью потребляет больше памяти, не имеет параллельной реализации или вообще не существует.
- У вас есть реализация алгоритма в коде. Приступайте к оптимизации **только** после того, как убедитесь, что программа работает корректно.

Процесс оптимизации, как правило, состоит из (а) замеров времени работы программы, (b) выбора фрагмента кода для ускорения и (с) ускорения выбранного фрагмента. Для замеров времени работы всей программы или отдельных фрагментов кода можно использовать модуль [timeit](#). В [jupyter](#) ноутбуках и [ipython](#) шеллах данный функционал удобно оформлен в виде магических команд [%timeit](#) и [%%timeit](#).

Процессоры отличаются наборами поддерживаемых инструкций и количеством ядер, а операционные системы — версиями низкоуровневых библиотек, потому по возможности проводите замеры на целевых машинах и на целевых входных данных.

Остаток раздела содержит рассказ о некоторых причинах эффективности тензорных библиотек и ссылки на полезные ресурсы. Для каждого пункта есть примеры с демонстрацией нюансов и рекомендациями по написанию кода. В примерах, где явно не указаны характеристики ЭВМ для замеров, использовалась машина с AMD Ryzen 9 5900x и ОС Linux.

## Скомпилированный код

Хорошая скорость работы NumPy и аналогичных библиотек достигается в первую очередь тем, что функции для работы с массивами написаны на низкоуровневых компилируемых языках — C, C++, Fortran. Модули оптимизируются компилятором и транслируются в машинный код, а во время выполнения программы на Python импортируются в бинарном виде. Дополнительно работа NumPy ускоряется использованием [SIMD-инструкций](#), специализированных библиотек ([BLAS](#), [LAPACK](#), [MKL](#) и пр.) или ручной оптимизацией.

**Пример.** Поведение низкоуровневых оптимизаций зависит от конкретной ЭВМ. Можно предположить, что из-за [локальности](#) обращения к памяти подсчет суммы в каждой строке `.sum(axis=1)` должен отрабатывать быстрее, чем подсчет в каждом столбце `.sum(axis=0)` (элементы по умолчанию хранятся [построчно](#)). Замеры на разных машинах показывают, что на практике неверна ни эта теория, ни даже обратная:

		<code>.sum(axis=0)</code>	<code>.sum(axis=1)</code>
AMD Ryzen 5 2600	Ubuntu 22.04	44 ms	35 ms
AMD Ryzen 9 5900x	Kubuntu 24.04	27 ms	26 ms
Intel Core i7-11850H	Ubuntu 22.04	53 ms	63 ms

Вы можете провести замеры на своем компьютере с помощью `timeit`:

```
python -m timeit -s "import numpy; a = numpy.ones((10000, 10000));" -- "a.sum(axis=0)"
python -m timeit -s "import numpy; a = numpy.ones((10000, 10000));" -- "a.sum(axis=1)"
```

**Пример.** В результате оптимизаций функции NumPy отработывают быстрее, чем их реализации на “чистом” Python. Например, для массива `a` из 100000 элементов `a.sum()` (17.2  $\mu$ s) выполняется на порядки быстрее, чем суммирование в цикле `for` (9 ms) и даже чем встроенная функция `sum` (5 ms), хоть последняя тоже написана с использованием C. В меньшей степени это относится и к операциям, в которых сложнее использовать SIMD: `a.sort()` примерно в два раза быстрее для `a` типа `np.ndarray`, чем для `a` типа `list`.

**Вывод.** В целом, чем меньше логики программы написано на Python, тем быстрее она выполняется. Простой перенос кода (в первую очередь циклов `for`) “внутри” NumPy уже дает ощутимое ускорение и позволяет пройти ограничения по времени на задания курса.

## Групповое выполнение операций

Нужно выполнять как можно больше вычислений в одном вызове функции тензорной библиотеки и по возможности заменять фрагмент кода из нескольких операций одной функцией. Во-первых, реализация через одну функцию может выполнять операции параллельно или использовать другие оптимизации. Во-вторых, запуск функций интерпретатором Python может быть связан с накладными расходами, которые накапливаются, если вызов происходит в теле цикла.

**Пример.** Матричное умножение в NumPy выполняется оператором `@`, который соответствует `np.matmul`. Для матриц `A` и `B` размера  $10000 \times 10000$  умножение `A @ B` (5.37 ms) выполнится быстрее, чем умножение строк `A[i]` на `B` отдельно в цикле `for` из 10000 итераций (51.5 ms). Ускорение достигается в том числе тем, что в реализациях BLAS (на которых основан `np.matmul`) операции *general matrix multiplication* (`gemm`) вычисляются с использованием *блочного умножения*.

**Пример.** В NumPy есть специальные функции, которые эффективно выполняют сразу несколько операций. Например, для перемножения трех квадратных матриц  $1000 \times 1000$  `np.linalg.multi_dot` выполняется на  $\approx 25\%$  быстрее, чем два `matmul`. Кроме того, вы можете использовать функцию `np.einsum`, которая объединяет в себе суммирование, изменение размерностей, матричное / поэлементное умножение и другие операции, а с флагом `optimize = True` применяет дополнительные оптимизации.

## Условные операторы в Python

Использование операторов `if` часто мешает написанию эффективных программ, так как может препятствовать переносу большего фрагмента кода в NumPy (см. первые два пункта).

**Пример.** Ниже приведены две программы с оператором `if` в цикле. Они эквивалентны с точки зрения итогового значения массива `b`, однако реализация справа выполняется в два раза быстрее, так как все `a[i].sum` перенесены в `a.sum` (пусть и ценой выполнения лишних вычислений).

<pre>a = np.ones((100, 50, 50)) b = np.zeros((100,)) for i in range(100):     if i % 17 &lt; 8: b[i] = a[i].sum()</pre>	<pre>a = np.ones((100, 50, 50)) b = a.sum(axis=(1, 2)) for i in range(100):     if i % 17 &gt;= 8: b[i] = 0</pre>
---	---

В большинстве случаев условные конструкции в Python можно целиком занести внутрь операций NumPy или, по крайней мере, вынести их за пределы “горячих” циклов (т.е. циклов, которые содержат много итераций и составляют значительную часть выполнения программы, см. (а) из процесса оптимизации).

## Условные операторы в NumPy

Ветвления в низкоуровневом коде тоже влияют на быстродействие. Условные переходы, которые плохо предсказываются, ведут к опустошению конвейера команд и промахам в кэше, замедляя программу.

**Пример.** Создадим массивы `a`, `b` и `c` как `np.random.rand(100000)` со случайными элементами из `[0, 1]`. Далее приведены замеры времени работы `np.where(a > frac, b, c)` в зависимости от `frac`. Количество ошибок предсказаний (и, как следствие, время выполнения) тем больше, чем ближе соотношение `True` и `False` в маске к 1.

frac	0.0	0.25	0.5	0.75	1.0
Время	88.5 $\mu$ s	228 $\mu$ s	347 $\mu$ s	230 $\mu$ s	89.7 $\mu$ s

## Выбор элементов по маске

Кроме `np.where` в NumPy есть и другие способы работать с элементами по булевой маске или индексам. А если маска не случайна, как в последнем примере, а имеет структуру, то в функциях типа `np.where` начинается прогнозирование ветвлений и в целом в нумпу появляется больше возможностей для эффективной реализации.

**Пример.** Реализации через `np.where`, `a[idxs]` и `a[mask]` работают с разной скоростью в зависимости от структуры маски, баланса `True` и `False` и т.д. Но если требуется взять небольшое количество элементов, `a[idxs]` зачастую выполняется быстрее (особенно если `idxs` уже посчитан). Ниже приведены замеры для массива `a` из последнего примера.

Реализация	Время	Комментарий
<code>np.where(mask, 42.0, a)</code>	78 $\mu$ s	<code>mask = a &lt; 0.01</code>
<code>a[mask] = 42.0</code>	23 $\mu$ s	<code>mask = a &lt; 0.01</code>
<code>a[idxs] = 42.0</code>	2 $\mu$ s	<code>idxs, = (a &lt; 0.01).nonzero()</code>

**Пример.** Маска может быть случайной лишь частично — например, состоять из непрерывных фрагментов только из `True` или только из `False`. В таком случае успешных предсказаний ветвлений будет больше.

**Пример.** Некоторые маски со структурой можно разложить на операции простой индексации. Например, присваивание для каждой второй пятерки элементов массива реализуется как `a.reshape(-1, 10)[:5, :5] = 42.0` (такой код отработает почти в два раза быстрее, чем индексация по булевой маске).

## Вычисления в разных типах данных

Разные типы данных (целочисленные, с плавающей точкой, логические и другие) отличаются размером в памяти и скоростью выполнения операций. При этом быстродействие вычислений для конкретного типа данных зависит от поддержки этого типа в микроархитектуре CPU или GPU. Производительность разных `np.intXX` и `np.floatXX` нужно тестировать отдельно на целевых ЭВМ.

**Пример.** По умолчанию в NumPy массивы создаются с типом `float64`. Обычно операции с `float32` быстрее, чем с `float64`, так что если двойная точность для вычислений избыточна, можно ускорить программу и работать с `float32`.

**Пример.** На многих CPU нет схем для оптимизированных вычислений в `float16`. В таком случае работа с `float16` может оказаться на порядок медленнее, чем с `float32` (в отличие от GPU, где для быстрой работы нейронных сетей обычно добавлена аппаратная поддержка более “узких” типов `float`).

## Конвертация типов

Приведение одного типа к другому занимает время и может происходить неявно при выполнении операций с операндами разных типов. Операции над единственным массивом (`np.sum`) тоже могут вернуть массив другого типа.

**Пример.** За неявной конвертацией нужно следить. Например, для (нецелочисленного) деления / с операндами `float32` и `int32` вычисления проходят в `float64`, что приводит к дополнительным затратам на приведение типов. Ниже показаны несколько других примеров.

a.dtype	b.dtype	(a / b).dtype	Время	Комментарий
float64	float64	float64	261 $\mu$ s	Операнды изначально в <code>float64</code>
float64	int32	float64	353 $\mu$ s	$\approx 100 \mu$ s на конвертацию <code>int32</code> $\rightarrow$ <code>float64</code>
float32	float32	float32	101 $\mu$ s	Результат в <code>float32</code> , приведения нет
float32	int16	float32	195 $\mu$ s	Результат в <code>float32</code> , приведение <code>int16</code> $\rightarrow$ <code>float32</code>
float32	int32	float64	464 $\mu$ s	Результат в <code>float64</code> , приводятся оба операнда
int32	float32	float64	470 $\mu$ s	
int32	int32	float64	471 $\mu$ s	Оба операнда целочисленные, результат в <code>float64</code>
int64	int64	float64	1220 $\mu$ s	

## Полезные ресурсы

Ниже приведены ссылки на полезные функции NumPy и другие ресурсы.

- `np.argwhere`, `np.argmax`, `np.argsort`, `np.where`. Для `np.argmax` обратите внимание на формат возвращаемого значения.
- `np.linalg.multi_dot`, `np.einsum` и библиотека `einops`, которая поддерживает NumPy, PyTorch и другие библиотеки и для тензорных операций использует нотацию, вдохновленную соглашением Эйнштейна.
- `np.take_along_axis`, `np.put_along_axis`, `np.roll`.
- `np.lib.stride_tricks.as_strided`, `np.lib.stride_tricks.sliding_window_view` позволяют переопределить атрибуты `shape` и `strides` массива. Результирующий массив может уже не быть непрерывным в памяти (`contiguous`), будьте осторожны в применении этих функций.
- `np.unravel_index`, `np.ravel_multi_index` переводят индексы элементов между форматами для многомерного массива и его `flattened`-версии.



# HIGH-QUALITY LINEAR INTERPOLATION FOR DEMOSAICING OF BAYER-PATTERNED COLOR IMAGES

*Henrique S. Malvar, Li-wei He, and Ross Cutler*

Microsoft Research  
One Microsoft Way, Redmond, WA 98052, USA

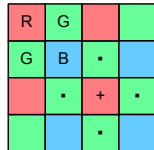
## ABSTRACT

This paper introduces a new interpolation technique for demosaicing of color images produced by single-CCD digital cameras. We show that the proposed simple linear filter can lead to an improvement in PSNR of over 5.5 dB when compared to bilinear demosaicing, and about 0.7 dB improvement in R and B interpolation when compared to a recently introduced linear interpolator. The proposed filter also outperforms most nonlinear demosaicing algorithms, without the artifacts due to nonlinear processing, and a much reduced computational complexity.

## 1. INTRODUCTION

Digital cameras are quite popular today; many users are switching away from regular film photography. For cost reduction, most digital cameras use a single charge-coupled device (CCD) sensor, with the CCD pixels preceded in the optical path by a color filter array (CFA) in a Bayer mosaic pattern, as shown in Fig. 1. For each  $2 \times 2$  set of pixels, two diagonally opposed pixels have green filters, and the other two have red and blue filters. Since G carries most of the luminance information for humans, its sampling rate is twice that of R and B.

We call demosaicing the problem of interpolating back the image captured with a CFA, so that for every CCD pixel we can associate a full RGB value. The simplest approach to demosaicing is bilinear interpolation [1]–[4], in which the three color planes are independently interpolated using symmetric bilinear interpolation from the nearest neighbors of the same color. As expected, bilinear interpolation generates significant artifacts, especially across edges and other high-frequency content, since it doesn't take into account the correlation among the RGB values. Practical demosaicing algorithms take such correlation into ac-



**Figure 1.** Typical Bayer mosaic for color image capture in single-CCD digital cameras; the G subimage has twice as many pixels as the R and B subimages.

count, either with better linear filters [4], or with nonlinear filters that adapt interpolation smoothness to a measure of image activity or edginess [1]–[4].

In this paper we present a simple linear demosaicing filter, with better performance and lower complexity than that in [4]. Our filter also outperforms many nonlinear algorithms. In Section 2 we quickly review some of the techniques proposed for improved demosaicing performance, and in Section 3 we present our new linear filter. Performance comparisons and conclusions are presented in Sections 4 and 5, respectively.

## 2. BEYOND BILINEAR DEMOSAICING

Referring to Fig. 1, in bilinear demosaicing the green value  $g(i,j)$  at a pixel position  $(i,j)$  that falls in a red or blue pixel, is computed by the average of the neighboring green values in a cross pattern:

$$\hat{g}(i,j) = \frac{1}{4} \sum_{(m,n) \in \{(0,-1), (0,1), (-1,0), (1,0)\}} g(i+m, j+n) \quad (1)$$

which corresponds to estimating the green value at the pixel marked '+' in Fig. 1 as the average of the observed green values marked '•'. At image boundaries, only pixels that fall within the image are included, and the scaling adjusted. For the R and B colors the same equation applies (with a diagonal cross pattern), except that for pixel positions  $(i,j)$  that fall in a green pixel only two red neighboring values are averaged to produce an interpolated red value; the same holds for blue. Besides the computational simplicity of (1), its output value is guaranteed to have the same dynamic range of the input value, so no overflow logic is needed on the output value.

Exploiting the correlation among the RGB channels is the main idea for improving demosaicing performance. Specifically, we can assume that in a luminance/chrominance decomposition, the chrominance components don't vary much across pixels. In the constant-hue approach of Freeman [2],[5], the green channel is bilinearly interpolated, and then the R and B channels are interpolated such as to maintain a constant hue, defined as the R/G and B/G ratios. Even at the expense of computing those divisions, it still produces visible artifacts [2]. Better results can be obtained by starting with bilinearly interpolated G pixels, and

applying median filters to the interpolated values of the color differences  $R - G$  and  $B - G$  [2].

Improved performance can be obtained with gradient-based methods [2],[6], which typically estimate edge directions and adjust the interpolation formulas so that filtering is performed preferentially along edge directions, and not across them. For example, for interpolating the green channel in Laroche and Prescott's method, the R and B channels are used to determine edge directions, which determine unequal weights to the terms in (1) for the green channel; the color differences  $R - G$  and  $B - G$  are then interpolated [2],[6]. Hamilton and Adams' method improves on Laroche and Prescott's by considering both first- and second-order pixel differences [6],[7]. Chang, Cheung, and Pang improved on that by considering a variable number of gradients [8]. A simpler but efficient algorithm that uses soft decision rules to combine interpolation results from horizontal and vertical directions is presented in [9].

Iterative methods can lead to further improvement by using results from B and R interpolation to correct the G interpolation, and vice-versa. That is the basis of Kimmel's approach 0, where the interpolation steps are based on a combination of the constant-hue and gradient-based methods. A technique based on iterative projections is presented in [6]. It has the best performance to date on a popular set of standard test images, but it has a very high complexity (as many as 480 operations per input pixel). So, the method in [6] can be seen as setting performance bounds to what can be achieved with more practical algorithms.

### 3. IMPROVED LINEAR INTERPOLATION

It is interesting to note that most of the algorithms discussed in the previous section use nonlinear filters whose region of support go beyond immediately adjacent pixels. That raises a simple question, which was not addressed in [1]–[9]: what is the best performance we can get from a linear filter whose region of supports extends to, say, a  $5 \times 5$  pixel region?

Pei and Tam proposed a good solution along those lines [4], by first estimating R and G via bilinear interpolation, then estimating the color differences  $B - G$  and  $R - G$  from those, smoothing the color differences via 4-point averages (under the assumption that the color differences are slowly-varying), and finally using those to interpolate the G values and then to do a final estimate of the R and B values. Their unrolled interpolation equations are equivalent to linear filters operating on a  $5 \times 5$  pixel window around the current pixel [4].

#### 3.1 The Proposed Interpolation Method

Instead of using a constant or near-constant hue approach like the methods described above, we propose the use of the following criterion: edges have much stronger luminance than chrominance components. Thus, when we look at interpolation of a green value at a red pixel location, for example, we don't discard the red value at that location – it is valuable information! Rather, we compare that red value to its estimate for a bilinear interpolation for the nearest red samples. If it is different from that estimate, it probably means there's a sharp luminance change at that pixel,

and thus we correct a bilinearly interpolated green value by adding a portion of this estimated luminance change.

Specifically, to interpolate G values at an R location, for example (the '+' pixel in Fig. 1), we use the formula

$$\hat{g}(i, j) = \hat{g}_B(i, j) + \alpha \Delta_R(i, j) \quad (2)$$

where  $\Delta_R(i, j)$  is the gradient of R at that location, computed by

$$\Delta_R(i, j) \triangleq r(i, j) - \frac{1}{4} \sum_{(m,n) \in \{(0,-2), (0,2), (-2,0), (2,0)\}} r(i+m, j+n) \quad (3)$$

The subscript  $B$  in (2) means bilinearly interpolated, as in (1). So, we correct the bilinear interpolation estimate by a measure of the gradient  $\Delta_R$  for the known color at the pixel location, which is estimated by the simple formula in (3). The gain factor  $\alpha$  controls the intensity of such correction.

Thus, our method is in fact a gradient-corrected bilinear interpolated approach, with a gain parameter to control how much correction is applied. For interpolating G at blue pixels, the same formula is used, but corrected by  $\Delta_B(i, j)$ . For interpolating R at green pixels, we use the formula

$$\hat{r}(i, j) = \hat{r}_B(i, j) + \beta \Delta_G(i, j) \quad (4)$$

with  $\Delta_G(i, j)$  determined by a 9-point region, as shown in Fig. 2. For interpolating R at blue pixels, we use the formula

$$\hat{r}(i, j) = \hat{r}_B(i, j) + \gamma \Delta_B(i, j) \quad (5)$$

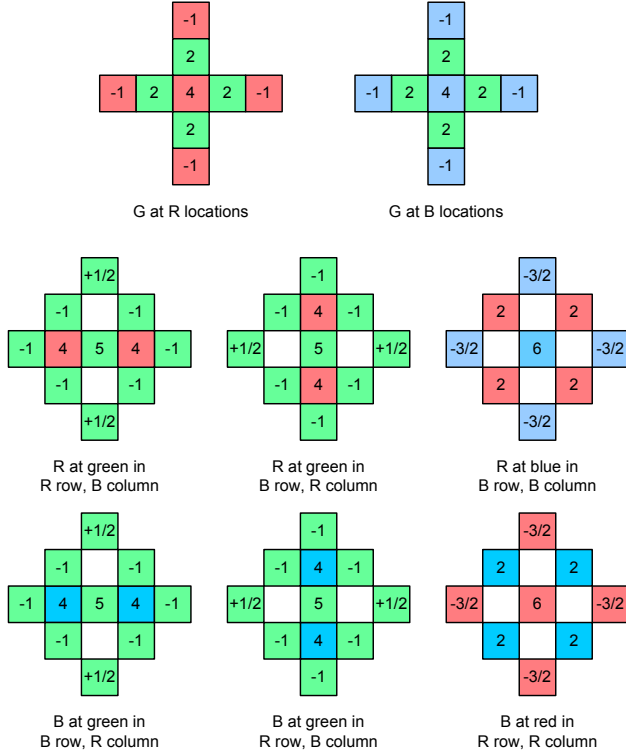
with  $\Delta_B(i, j)$  computed on a 5-point region, also shown in Fig. 2. The formulas for interpolating B are similar, by symmetry.

#### 3.2 Gradient-Correction Gains

To determine appropriate values for the gain parameters  $\{\alpha, \beta, \gamma\}$ , we used a Wiener approach; that is, we computed the values that led to minimum mean-square error interpolation, given second-order statistics computed from a good data set (the Kodak image set used in [6]). We then approximated the optimal Wiener coefficients by integer multiples of small powers of  $1/2$ , with the final result  $\alpha = 1/2$ ,  $\beta = 5/8$ , and  $\gamma = 3/4$ . From the values of  $\{\alpha, \beta, \gamma\}$  we can compute the equivalent linear FIR filter coefficients for each interpolation case. The resulting coefficient values, shown in Fig. 2, make the filters quite close (within 5% in terms of mean-square error) to the optimal Wiener filters for a  $5 \times 5$  region of support.

Thus, we believe that the only way to design a practical linear filter with a lower mean-square interpolation error would be to use larger regions of support. That would not only increase computational complexity and memory footprint, but it would also lead to ringing artifacts around edges.

One way to evaluate computational complexity is to count the number of nonzero filter coefficients within the  $5 \times 5$  regions of support. For Pei-Tam's method [4], there are 9 nonzero coefficients for the G channel, and an average of 13 each for the R and B channels. Our method has a slightly lower complexity: 9 coefficients for the G channel and 11 each for R and B.



**Figure 2.** Filter coefficients for our proposed linear

#### 4. PERFORMANCE

We compared our proposed method to several others, using the Kodak image set [6], by simulating a Bayer sampling array (simply by keeping only one of each of the RGB values for each pixel, as in Fig. 1) and then applying the various interpolation algorithms. This subsampling approach is not really representative of digital cameras, which usually employ careful lens design to effectively perform a small amount of lowpass filtering to reduce the aliasing due to the Bayer pattern subsampling. However, since all papers in the references perform just subsampling, with no lowpass filtering, we did the same so we could compare results. We have also tested all interpolation methods with small amounts of Gaussian lowpass filtering before Bayer subsampling, and found that the relative performances of the methods are roughly the same, with or without filtering. The improvement in peak-signal-to-noise ratio (PSNR) over bilinear interpolation is shown in Table 1. Except for Freeman’s method, most approaches lead to more than 5 dB improvement, and our proposed method on average outperforms all others.

Besides PSNR measurements, we need to verify visual quality. Fig. 3 shows the interpolated results for one of the images in the Kodak set. From those we see that our proposed method leads to a quality similar to a good nonlinear method such as Kimmel’s 0, but with a much lower complexity. Compared to Pei-Tam linear filters, our method produces slightly less visible color fringing distortion, as well as a lower PSNR, at a slightly reduced computational complexity.

#	ch	A	B	C	D	E	F
1	R	3.44	5.58	4.17	8.17	4.74	5.56
	G	0.00	5.26	6.62	5.55	6.09	6.09
	B	3.74	5.82	3.94	8.05	5.28	5.87
2	R	2.46	3.99	3.68	-1.52	4.05	4.34
	G	0.00	4.60	5.10	2.20	4.76	4.76
	B	3.34	5.04	4.66	4.75	4.67	4.48
3	R	3.80	6.05	6.56	4.67	5.68	6.84
	G	0.00	6.66	6.27	0.12	7.52	7.52
	B	3.76	5.98	6.14	7.17	5.34	6.50
4	R	3.48	5.80	4.47	8.22	5.16	5.81
	G	0.00	5.20	6.19	5.02	5.92	5.92
	B	3.38	5.50	4.42	6.63	4.18	4.97
5	R	3.89	5.67	5.99	4.61	4.82	6.20
	G	0.00	5.79	5.67	-3.01	5.71	5.71
	B	3.65	5.44	6.05	5.04	4.67	5.81
6	R	3.49	6.29	3.42	9.50	5.22	5.68
	G	0.00	6.16	6.63	6.41	5.72	5.72
	B	3.50	6.27	3.33	9.12	4.99	5.52
7	R	3.70	6.00	5.09	7.97	4.75	5.75
	G	0.00	5.95	6.14	3.75	5.90	5.90
	B	3.48	5.86	4.71	7.75	4.29	5.38
8	R	3.68	5.92	5.71	7.41	5.19	6.29
	G	0.00	6.19	6.83	6.23	6.86	6.86
	B	3.46	5.65	5.56	8.08	4.87	5.92
9	R	3.40	5.30	5.01	6.22	5.04	5.68
	G	0.00	5.00	5.83	2.30	5.84	5.84
	B	3.64	5.45	5.15	6.54	4.83	5.47
10	R	3.33	5.75	4.48	7.57	4.07	4.91
	G	0.00	6.10	6.44	5.91	5.64	5.64
	B	3.46	5.98	4.53	7.82	4.61	5.31
11	R	3.58	4.68	5.97	7.56	5.43	5.88
	G	0.00	4.25	5.65	1.62	6.15	6.15
	B	3.43	4.49	5.65	5.49	4.93	5.47
12	R	2.45	4.03	3.89	0.94	3.25	3.98
	G	0.00	4.80	5.99	4.58	5.60	5.60
	B	3.62	5.35	4.45	8.40	5.64	5.76
13	R	3.40	5.62	4.16	8.01	4.66	5.33
	G	0.00	5.05	5.85	4.26	5.47	5.47
	B	3.40	5.62	4.29	6.60	4.52	5.20
14	R	3.74	5.23	6.08	6.59	4.98	5.77
	G	0.00	4.98	5.55	1.20	6.02	6.02
	B	3.70	4.97	5.88	5.78	5.08	5.76
15	R	3.41	4.84	6.01	6.00	5.42	6.00
	G	0.00	4.58	5.66	0.73	5.92	5.92
	B	3.21	4.39	5.32	5.25	4.60	5.17
mean		2.31	5.40	5.32	5.36	5.20	5.68

**Table 1.** PSNR improvement in dB over bilinear interpolation for various interpolation methods, for the first 15 images in the Kodak set [6] A) Freeman [2],[5]; B) Hamilton-Adams [6],[7]; C) Chang et. al. [8]; D) Kimmel 0; E) Pei-Tam [4]; F) proposed method. Data in columns B–D are from Table III in [6].



**Figure 3.** Demosaicing results for various interpolation algorithms. From left to right; top row: original, bilinear, Freeman [2],[5]; middle row: Laroche-Prescott [2],[6], Hamilton-Adams [6],[7], Chang et. al. [8]; bottom row: Pei-Tam [4], Kimmel [3], and our proposed method.

## 5. CONCLUSION

We presented a new demosaicing method for color interpolation of images captured from a single CCD using a Bayer color filter array. The proposed linear filters are nearly optimal in a Wiener sense, and in fact outperform many more complex nonlinear filters. Compared to a recently introduced linear demosaicing approach [4], our filters produce a 0.5 dB improvement in interpolated image quality (a 12% reduction in mean-square error), which comes from the same quality for the green channel and about 0.7 dB improvement for the R and B channels. Compared to [4], our method also leads to a small reduction (roughly 12%) in computational complexity.

## REFERENCES

- [1] P. Longère, X. Zhang, P. B. Delahunt, and D. H. Brainard, "Perceptual assessment of demosaicing algorithm performance," *Proc. IEEE*, vol. 90, pp. 123–132, Jan. 2002.
- [2] R. Ramanath, W. E. Snyder, and G. L. Bilbro, "Demosaicking methods for Bayer color arrays," *J. Electronic Imaging*, vol. 11, pp. 306–315, July 2002.
- [3] R. Kimmel, "Demosaicing: image reconstruction from color CCD samples," *IEEE Trans. on Image Processing*, vol. 8, pp. 1221–1228, Sept. 1999.
- [4] S.-C. Pei and I.-K. Tam, "Effective color interpolation in CCD color filter array using signal correlation," *Proc. ICIP*, pp. 488–491, Sept. 2000.
- [5] D. R. Cok, "Reconstruction of CCD images using template matching," *Proc IS&T Annual Conf. / ICPS*, pp. 380–385, 1994.
- [6] B. K. Gunturk, Y. Altunbasak, and R. M. Mersereau, "Color plane interpolation using alternating projections," *IEEE Trans. on Image Processing*, vol. 11, pp. 997–1013, Sept. 2002.
- [7] J. E. Adams, Jr., "Design of practical color filter array interpolation algorithms for digital cameras," *Proc. SPIE*, vol. 3028, pp. 117–125, Feb. 1997.
- [8] E. Chang, S. Cheung, and D. Y. Pan, "Color filter array recovery using a threshold-based variable number of gradients," *Proc. SPIE*, vol. 3650, pp. 36–43, Jan. 1999.
- [9] X. Wu and N. Zhang, "Primary-consistent soft-decision color demosaic for digital cameras," *Proc. ICIP*, vol. I, pp. 477–480, Sept. 2003.