

# Описание программной части робота-художника

Латыпов Владимир Витальевич

15 июля 2021 г.

## 1. Формулировка задачи

Для того, чтобы робот-художник нарисовал что-либо, ему нужно предоставить данные в определённом формате, а именно — не набор пикселей, как требуется для показа на мониторе, а набор «мазков»: это связано с конструкцией самого робота. Мазки решено было представлять в виде кривых безье второго порядка (то есть квадратичных), к которым добавлены параметры «толщина» и «цвет».

Но на вход подаются рисунки не в векторном, а в растровом формате. Найти такую комбинацию мазков, которая бы лучше всего соответствовала картине/изображению — задача нетривиальная, имеющая множество решений.

Конечно же, я выбрал решать задачу самостоятельно, а не использовать готовые библиотеки.

## 2. Описание программы в общих чертах

Таким образом, решено было использовать эвристические алгоритмы оптимизации: Генетический алгоритм и Симуляция отжига.

Функцию ошибки необходимо задать таким образом, чтобы она отражала качество полученной комбинации мазков, причём в любой точке направление её уменьшения соответствовало направлению улучшения результата. Помимо напрашивавшегося MSE

$$MSE = \sum_{y=0}^{y < height} \sum_{x=0}^{x < width} \sum_{c \in \{r,g,b\}} \left( \overrightarrow{\text{rendered}_{x,y,c}} - \overrightarrow{\text{original}_{x,y,c}} \right)^2 \quad (1)$$

, повсеместно используемого при работе с изображениями, функция ошибки «наказывает» наложение мазков, а также

## 3. Технические аспекты

Программа написана на языке программирования C++ (так как требовалась максимальная скорость), сборка осуществляется с помощью CMake. Проект можно скомпилировать под Windows (компилятор MSVC) и под Linux (тестировалось на g++-10).

Код хранится в

Для работы с изображениями используется OpenCV, но не модуль машинного обучения, а лишь примитивные операции с изображениями: чтение из популярных форматов, сохранение в них, хранение и копирование матрицы пикселей и т.д.

Для работы проекта также необходима библиотека «*pythonic*»: она написана мной, подключается также через CMake. Она отвечает за базовые функции и структуры данных. Я использую её во всех более или менее крупных проектах на C++. В ней на данный момент есть:

- Простые вспомогательные функции для работы со строками, контейнерами, форматированного вывода
- Вызов питоновской библиотеки *matplotlib* для построения графиков
- Базовые алгоритмы наподобие бинарного поиска и дерева отрезков
- Функционал для работы со временем, в том числе — анализатор последовательных запусков процесса
- Платформонезависимая работа с кодировками и файловыми системами
- Примитивы для вычислительной геометрии
- Функции для работы со статистикой
- Многомерный шаблонный массив с количеством измерений, изменяемом в run-time
- Сглаживание функций и построение примерной функции распределения в пространстве с заданной размерностью по набору *sampler*-ов с помощью гауссовых ворот
- Функционал для работы с многопоточностью, в том числе — *thread pool*, умеющий снимать нагрузку с ожидающих потоков с помощью *std::condition\_variable*.

Реализация

## 4. Алгоритмы оптимизации

### 4.1. Общий принцип ГА

Идея работы Генетического алгоритма заимствована у природы.

Точно также, как в ходе эволюции происходит появление оптимального организма для заданных условий, в ходе работы алгоритма ищется набор параметров функции, при котором фитнес-функция максимальна.

В природе тот, кто лучше приспособлен к окружающей среде, в большей степени получает доступ к размножению (с помощью различных механизмов), а при размножении новая особь наследует признаки каждого из родителей.

Это позволяет именно лучшим чертам, по каким-либо причинам появившимся у особей, переходить в следующее поколение. Эти черты появляются через мутации — небольшие случайные изменения в геноме.

Из принципа работы можно понять, что алгоритм эвристический: сложно доказать его сходимость или что-либо гарантировать с вероятностью 100%. Зато исследования показывают, что именно этот алгоритм даёт лучшие результаты для самых сложных функций. Далее мы увидим, какие меры предпринимаются, чтобы не дать алгоритму попасть в локальный минимум, не добравшись до глобального.

### 4.2. Термины

Набор параметров представляется в виде «генома» — некой структуры данных, содержащей информацию об этом наборе. Особь — в контексте алгоритма будет использоваться в качестве синонима к геному. Геном состоит из генов — каждый из них содержит информацию о каком-либо признаком (в случае природы) или параметре (в случае ГА).

В каждый момент времени алгоритм работает с популяцией — набором геномов. Полный аналог популяции в природе. Мутация — как и в реальной жизни — небольшое случайное изменение генома без строго определённого направления.

### 4.3. Примерная последовательность действий ГА

В общих чертах работа ГА выглядит так:

Инициализация: Сгенерировать случайную популяцию, каждый ген каждого генома — в заданных пределах. Затем — повторять, пока не за-

кончится заданное количество итераций или не будет достигнуто требуемое значение фитнесс-функции:

1. Посчитать фитнесс-функции для каждой из особей. Для большинства задач этот шаг занимает большую часть времени выполнения, поэтому нужно оптимизировать именно его, в частности — параллелизовать, запуская независимые вычисления на нескольких потоках.
2. Каким-то образом отобрать особи на скрещивание
3. Произвести скрещивание, получив «отпрысков» — часть нового поколения
4. Сформировать новое поколение, используя, возможно, в разных пропорциях, различные источники геномов, а именно:
  - Отпрысков, полученных на предыдущем шаге в результате скрещивания
  - Лучшие особи из прошлой популяции
  - Случайные особи — чтобы не дать алгоритму сойтись раньше времени, попав в локальный минимум
  - Возможно, результаты скрещивания особей в числе больше 2
5. Произвести мутации в некоторых особях этого поколения (лучшие из мутаций внедряются в популяцию на следующих итерациях)
6. Если алгоритм подходит к концу, добавить лучший геном из предыдущего поколения (чтобы он не подвергся мутации)

#### 4.4. Авторские Модификации в ГА

Учитывая тот факт, что в большинстве задач, решаемых мною, большая часть вычислительного времени ( $>> 95\%$ ) используется для подсчёта функции ошибки, а не для манипуляций с геномами (это подтверждается результатами профайлинга). То есть задача состоит в том, чтобы минимизировать количество подсчётов функции ошибки, даже ценой более долгой работы с геномами.

Первое изменение — я решил отказаться от дискретного кодирования геномов. Для алгоритма по каждой переменной задан её диапазон.

Традиционный подход — разделить диапазон на  $2^N$  частей и кодировать номер части в геноме как битовую последовательность из  $N$  бит. Для подсчёта функции ошибки этот номер перекодируется назад в соответствующую точку непрерывной величины. Мутаций в данном случае является изменение случайного количества каких-то битов этого номера. А скрещивание обычно происходит путём

Предварительное тестирование показало большую эффективность этого метода по сравнению с традиционным подходом, однако планируется провести тщательное тестирование (см. ??)

Алгоритм реализован мной на языке C++, он хранится в GitHub репозитории <https://github.com/donRumata03/PowerfulGA>.

## 5. Наблюдения

### 5.1. Неравенство зон

Когда я заметил, что зоны, на которые Adobe Illustrator делит изображение, могут очень сильно отличаться в размере (отношение площадей может достигать 1000 раз), мне захотелось измерить это неравенство численно, чтобы при разработке нового алгоритма измерять его качество в том числе по этому параметру.

Существует большое количество метрик, я решил выбрать основные из них:

- Индекс Джини (вместе с кумулятивным графиком распределения дохода (также известен как кривая Лоренца))
- Процент «дохода» 1% самых богатых от общего «дохода» (В случае зон вместо дохода используется занимаемая площадь).
- Процент самых богатых, имеющих в сумме 50% от общего дохода.

Индекс Джини рассчитывается как отношение площади между кривой Лоренца и «линией равенства» к площади под линией равенства. Иными словами,  $G = \frac{A}{A+B}$  на этой схеме:

Альтернативный способ посчитать коэффициент, использующийся в реализации:

$$G = \frac{\sum_{i=1}^n \sum_{j=i+1}^n |y_i - y_j|}{n \cdot \sum_{i=1}^n y_i} \quad (2)$$

Чем индекс выше, тем большее неравенство наблюдается в стране. Более того, использование именно этой метрики позволяет комплексно

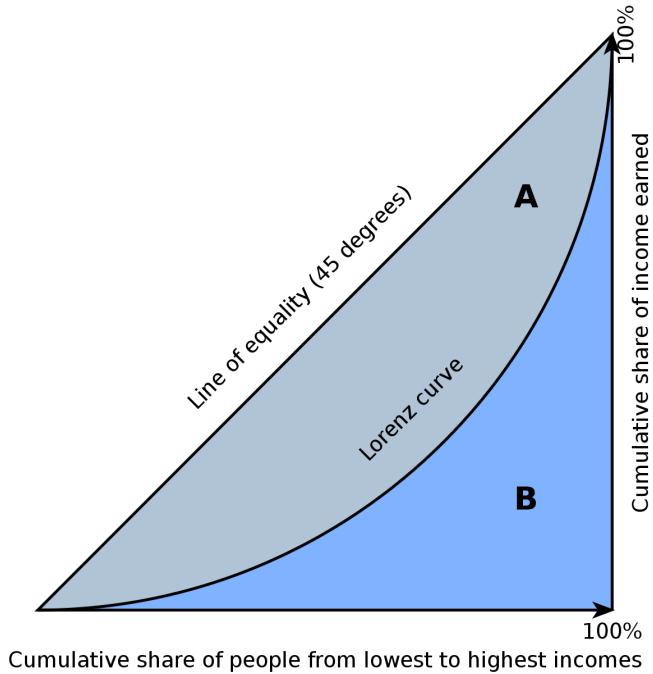


Рис. 1: Типичная кривая Лоренца

оценить неравенство между анализируемыми объектами — в отличие от рассмотрения процентов дохода заданного квентиля.

Результаты оказались впечатляющими:

- $Gini\_index \approx 76\%$
- 1% крупнейших зон покрывают  $\approx 18\%$  изображения
- 6.25% зон покрывают половину изображения

Так выглядит кумулятивный график распределения площади:

Нетрудно заметить, что ни в одной стране мира нет такого неравенства, как среди зон:

Даже в ЮАР индекс Джини составляет 57.8%.

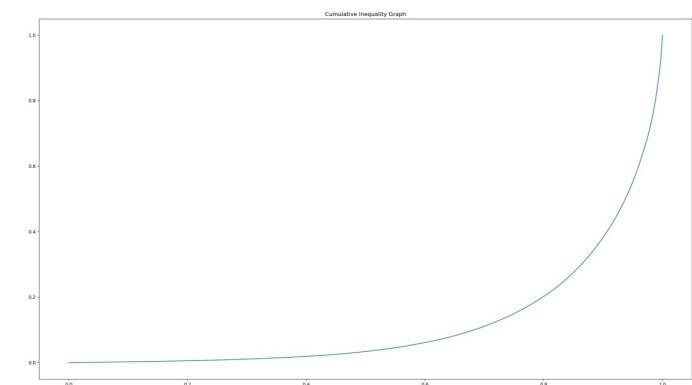


Рис. 2: Кривая лоренца для зон

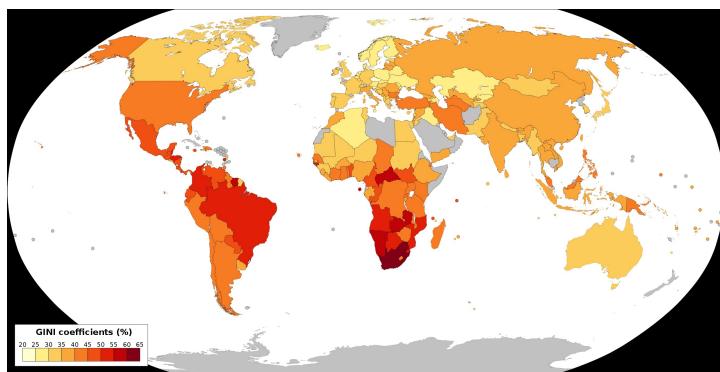


Рис. 3: Индекс Джини по странам мира

## 6. Дальнейшее развитие

Несмотря на то, что программа уже работоспособна, есть ещё много идей и планов по её усовершенствованию:

- Внедрить быстрый пересчёт функции ошибки — это улучшение давно напрашивается, но оно несколько теряет в эффективности из-за того, что в одной мутации в среднем изменяется не так мало мазков (однако это количество убывает со временем). В настоящий момент ведётся работа над внедрением.
- Разделение мазков по слоям. Нетрудно заметить, что при рисовании картин художники сначала проходятся по холсту черновыми

мазками большого размера, а затем — прорабатывают детали. Таких уровней детализации зачастую бывает немало.

Пример того, как художник (<https://www.youtube.com/watch?v=VaXHtai2alU>) рисует картину по слоям:



Рис. 4: Фон | Рельеф фона | Детализация заднего плана | Основные объекты

Поэтому стоит попробовать сначала заполнять картинку толстыми, грубыми мазками (то есть просто с большей шириной, а в реальной жизни это будет отражаться в большем размере кисти и в более сильном нажатии).

- Добавить использование локальных методов оптимизации. Такие методы, как градиентный спуск и метод Ньютона позволяют достичь гораздо большей скорости сходимости (в случае метода Ньютона — сходимость квадратичная), но требуют умения посчитать градиент функции ошибки в любой точке, а также вектор вторых производных по каждому из аргументов.

Сами алгоритмы реализованы и находятся в этой папке. Предусмотрена опция подсчёта первой и вторых производных через подстановку близких значений параметров:

$$f'(x_0) \approx \frac{f(x_0 + \Delta x) - f(x_0)}{\Delta x} \quad (3)$$

Однако в случае с мазками при маленьких изменениях параметров функция ошибки остаётся неизменной, так как это приводит к такому же набору закрашенных пикселей. Соответственно, нужно либо радикально увеличивать разрешение изображения, либо использовать аналитические методы. То есть нужно математически посчитать изменение функции ошибки при бесконечно малом изменении из параметров функции.

- Организовать систему тестирования различных алгоритмов на различных функциях. Звучит как нечто весьма простое, но реальность сложнее, чем кажется. Напрашивающийся вариант — дать каждому алгоритму заданное количество вычислений функции ошибки и сравнить, какой результат они получат.

Однако функция ошибки нелинейная, поэтому сложно будет понять, насколько сильному различию в качестве алгоритма соответствует полученная численная разница в результатах.

Целесообразно сравнивать количество итераций, требующееся алгоритмам для получения заданного результата. Но и тут не всё так просто: нельзя просто запустить алгоритмы на неограниченное количество итераций и ждать достижения нужного значения функции, так как во многих из них (как минимум — в моей модификации ГА) то, как будет проведена каждая отдельная итерация, сильно зависит от процента выполнения на момент её прохождения: происходит планирование, использующее информацию о максимальном количестве итераций.

Поэтому нет никакого другого выхода, кроме того, чтобы запускать этот алгоритм с рабочим количеством итераций и смотреть, когда он в среднем будет доходить до заданного порога. Это необходимо автоматизировать. В идеальном случае для поиска порога можно было бы использовать бинарный поиск, но в реальности (с поправкой на шум) имеет смысл использовать эвристическую модификацию н-арного поиска (объяснить!). Для полной оценки планируется построить график достаточного количества итераций от требуемого значения функции в интересующей нас зоне.

- Улучшить алгоритм поиска цветов и разделения на зоны.

Сейчас для разделения изображения на зоны используется Adobe Illustrator. По заданному количеству цветов (и, следовательно, уровню детализации) он разделяет изображение на зоны, присваивая каждой какой-то из цветов палитры так, чтобы он хорошо . Сама палитра тоже формируется в ходе работы алгоритма.

Скорее всего, для этого используется один из популярных алгоритмов, описанных здесь, или некая проприетарная их вариация. Зоны, на которые происходит деление, описываются частями плоскости, ограниченными кривыми Безье — «path» формате *svg*. Несмотря на то что формально алгоритм выполняет свою работу, большое количество зон имеет очень продолговатую форму, а

также наблюдается неимоверный разброс в размерах между различными зонами (см. ): всё это уменьшает эффективность процесса.