УДК 004.421.2:519.683

МЕТОДЫ ОПТИМИЗАЦИИ ВЫЧИСЛИТЕЛЬНОЙ ЭФФЕКТИВНОСТИ АЛГОРИТМОВ ГЕНЕРАЦИИ ФРАКТАЛЬНЫХ СТРУКТУР В СРЕДЕ MATLAB

Н. М. ШАРИПА, Т. А. РОЩУПКИН
(Представлено: К. И. ИВАНОВА)

Оптимизация вычислительной производительности алгоритмов генерации фрактальных структур в среде MATLAB представляет критически важную задачу. В статье рассматриваются методы повышения эффективности рекурсивных алгоритмов на примере построения снежинки Коха, включая предварительное выделение памяти, векторизацию операций и минимизацию избыточных вычислений. Приведены сравнительные характеристики времени выполнения и потребления памяти для различных подходов.

Цель данной статьи – систематизировать практические приемы оптимизации кода в MATLAB для данной задачи, количественно оценить их эффективность и показать применение на реальном примере.

Генерация фракталов относится к классу вычислительно сложных задач из-за своей рекурсивной природы и экспоненциального роста количества обрабатываемых элементов с каждой итерацией. Наивная реализация алгоритмов, характерная для начального этапа разработки, приводит к следующим проблемам:

– Динамическое расширение массивов: частое использование операций конкатенации ([a; b]) в цикле заставляет MATLAB постоянно запрашивать новые блоки памяти и копировать в них данные, что потребляет огромное количество времени.

– Избыточные вычисления: повторное вычисление одних и тех же констант (например, sin(pi/3), sqrt(3)/2) внутри циклов.

– Сложность отладки и тестирования: низкая производительность делает невозможным быстрое тестирование алгоритма с большим количеством итераций, что критично для исследований.

Методы решения обозначенных проблем

1. Предварительное выделение памяти:

Перед началом цикла создается массив максимально необходимого размера, заполненный нулями (zeros()).

Преимущество: MATLAB не тратит время на повторное выделение памяти и копирование данных. Доступ к элементам по индексу (array(i) = value) является одной из самых быстрых операций.

Реализуется при помощи расчета общего числа точек для n итераций по формуле для кривой Коха:

$$N\_points = 3 * (4^n) + 1.$$

2. Векторизация операций:

Основной смысл заключается в замене циклов (for, while) на операции с целыми массивами и матрицами. MATLAB оптимизирован для работы с матрицами.

Преимущество: избегание накладных расходов интерпретатора на каждой итерации цикла.

Реализуется выполнением арифметических операций над векторами и использование поэлементного умножения.

3. Минимизация избыточных вычислений:

Этот метод заключается в вынесение всех вычислений, результат которых не меняется в цикле, за его пределы.

Преимущество: сокращение количества операций.

Реализовать можно, используя предварительный расчет констант (например, cos60 = cos(pi/3)).

4. Оптимизация математических операций:

Замена тяжелых функций на их приближенные значения, если это допустимо точностью задачи.

Преимущество: ускорение вычислений.

В данном случае для реализации cos(pi/3) и sin(pi/3) заменяются на константы 0,5 и sqrt(3)/2 соответственно.

Для наглядности ниже представлены таблицы с результатами профилирования кода до и после оптимизации. Замеры проводились на компьютере с процессором Intel Core i7-1165G7 для 6 итераций построения снежинки Коха.

Сравнение времени выполнения и использования памяти находится в таблице 1, а вклад различных методов оптимизации в таблице 2.

Таблица 1. − Сравнение времени выполнения и использования памяти

Метод реализации	Время выполнения (с)	Потребление памяти (МБ)	Относительное ускорение
Наивный подход (с конкатенацией)	2,45	~2,1	1x (Базовый уровень)
Предварительное выделение	0,31	~0,8	~8x
Векторизация и предвычисление	0,12	~0,8	~20x

Таблица 2. − Вклад различных методов оптимизации

Оптимизация	Вклад в общее ускорение	Основной эффект
Предварительное выделение памяти	~80%	Устранение накладных расходов на управление памятью
Предвычисление констант	~15%	Сокращение количества операций в цикле
Векторизация арифметики	~5%	Более эффективное использование процессора

Вывод по таблицам – наибольший вклад в повышение производительности вносит корректная работа с памятью.

Практическое применение

Рассмотрим поэтапную оптимизацию функции построения снежинки Коха.

Шаг 1: исходный (наивный) код

```
function points = koch_snowflake_naive(n)
   points = [0, 0; 1, 0; 0.5, sqrt(3)/2; 0, 0];
   for i = 1:n
      new_points = [];
      for j = 1:size(points, 1)-1
         p1 = points(j, :);
         p2 = points(j+1, :);

         new_points = [new_points; p1];
         new_points = [new_points; p1 + (p2 - p1)/3];

         % Медленное вычисление внутри цикла
         mid = p1 + (p2 - p1)/2;
         vec = (p2 - p1)/3;
         rot_vec = [vec(1)*cos(pi/3) - vec(2)*sin(pi/3), ...
                    vec(1)*sin(pi/3) + vec(2)*cos(pi/3)];
         new_points = [new_points; p1 + (p2 - p1)/3 + rot_vec];

         new_points = [new_points; p1 + 2*(p2 - p1)/3];
      end
      new_points = [new_points; points(end, :)];
      points = new_points;
   end
end
```

Основными недостатками данного кода являются: динамическая конкатенация, повторные вычисления sin/cos, отсутствие предвыделения.

Шаг 2: полностью оптимизированный код

```
function points = koch_snowflake_optimized(n)
   % 1. Предвычисление констант
   cos60 = 0.5;        % cos(pi/3)
   sin60 = sqrt(3)/2;  % sin(pi/3)
   one_third = 1/3;
   two_thirds = 2/3;

   % 2. Предварительное выделение памяти
   total_points = 3 * (4^n) + 1;
   points = zeros(total_points, 2);
```

```
    % Инициализация начальным треугольником
    points(1:4, :) = [0, 0; 1, 0; 0.5, sqrt(3)/2; 0, 0];
    current_size = 4;
    for iter = 1:n
        % 3. Расчет размера для нового массива и его предвыделение
        new_size = 4 * (current_size - 1) + 1;
        new_points = zeros(new_size, 2);
        new_index = 1;

        for j = 1:current_size - 1
            p1 = points(j, :);
            p2 = points(j+1, :);
            vec = p2 - p1; % Вектор между точками

            % 4. Запись данных по индексу (быстрее конкатенации)
            new_points(new_index, :) = p1;
            new_points(new_index + 1, :) = p1 + one_third * vec;

            % 5. Использование предвычисленных констант
            rot_vec = [vec(1)*cos60 - vec(2)*sin60, ...
                       vec(1)*sin60 + vec(2)*cos60] * one_third;
            new_points(new_index + 2, :) = p1 + one_third * vec + rot_vec;
            new_points(new_index + 3, :) = p1 + two_thirds * vec;

            new_index = new_index + 4;
        end
        new_points(new_index, :) = points(current_size, :); % Last point
        points = new_points;
        current_size = new_size;
    end
end
```

Заключение. Применение системного подхода к оптимизации кода в MATLAB позволяет добиться значительного (в десятки раз) повышения производительности даже для сложных рекурсивных алгоритмов. Ключевым фактором является правильное управление памятью за счет ее предварительного выделения. Представленные методы носят универсальный характер и могут быть успешно применены для ускорения широкого круга вычислительных задач в среде MATLAB, особенно связанных с обработкой больших массивов данных и итерационными процессами.

ЛИТЕРАТУРА

1. MATLAB help center [Электронный ресурс]. - Режим доступа: Programming and Scripts - MATLAB & Simulink – Дата доступа: 12.09.2025.
2. MATLAB. Основы [Электронный ресурс]. - Режим доступа: https://www.mathworks.com/products/matlab.html – Дата доступа: 12.09.2025.
3. ГОСТ 7.1-2003 БИБЛИОГРАФИЧЕСКАЯ ЗАПИСЬ. БИБЛИОГРАФИЧЕСКОЕ ОПИСАНИЕ Общие требования и правила составления Хабр оптимизация кода MATLAB. - Режим доступа: https://habr.com/ru/articles/129550/ – Дата доступа: 12.09.2025