Московский авиационный институт
(национальный исследовательский университет)
Институт №8 «Информационные технологии и прикладная математика»
Кафедра вычислительной математики и программирования

Курсовая работа по численным методам

Тема

“Решение систем линейных алгебраических уравнений с симметричными разреженными матрицами большой размерности. Метод сопряженных градиентов”

Студент: Арсланов Тимур М.
Группа: М80-402Б-20
Руководитель: Пивоваров Д.Е.

Оценка:
Дата:

Москва
2023

ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

Разрежённая матрица, это матрица с преимущественно нулевыми элементами. В противном случае, если большая часть элементов матрицы ненулевые, матрица считается плотной.

Среди специалистов нет единого мнения в определении того, какое именно количество ненулевых элементов делает матрицу разрежённой. Будем считать, что это число равно $O(n)$

Концептуально разреженность соответствует системам с небольшим количеством парных взаимодействий. Например, рассмотрим линию шаров, соединенных пружинами от одного к другому: это разреженная система, поскольку связаны только соседние шары. Напротив, если бы одна и та же линия шаров имела пружины, соединяющие каждый шар со всеми другими шарами, система соответствовала бы плотной матрице. Концепция разреженности полезна в комбинаторике и таких прикладных областях, как теория сетей и численный анализ, которые обычно имеют низкую плотность важных данных или соединений. Большие разреженные матрицы часто появляются в научных или инженерных приложениях при решении уравнений в частных производных.

При хранении разреженных матриц и манипулировании ими на компьютере это полезно и часто необходимо использовать специализированные алгоритмы и структуры данных, которые используют разреженную структуру матрицы. Специализированные компьютеры были созданы для разреженных матриц, поскольку они распространены в области машинного обучения. Операции с использованием стандартных структур и алгоритмов с плотной матрицей медленны и неэффективны при применении к большим разреженным матрицам, поскольку обработка и память тратятся на нули. Разреженные данные по своей природе легче сжимать и, следовательно, требуют значительно меньше памяти. Некоторыми очень большими разреженными матрицами невозможно манипулировать с помощью стандартных алгоритмов плотных матриц

Метод сопряженных градиентов

Рассмотрим линейных уравнений:

$$Ax = b.$$

$$A = A^T > 0$$

Матрица A – положительно-определенная, симметричная.

Тогда процесс решения СЛАУ можно представить следующей задачей:

$$F(x) = \frac{1}{2}(Ax, x) - (b, x) \to inf, x \in \mathbb{R}^n$$

Здесь A - симметричная положительно определённая матрица размера $n \times n$. Такая задача оптимизации называется квадратичной. Заметим, что $F'(x) = Ax - b$. Условие экстремума функции $F'(x) = 0$ эквивалентно системе $Ax - b = 0$. Функция F достигает своей нижней грани в единственной точке x_*, определяемой уравнением $Ax_* = b$. Таким образом, данная задача оптимизации сводится к решению системы линейных уравнений $Ax = b$

Идея метода сопряжённых градиентов состоит в следующем:

Пусть $\{p_k\}_{k=1}^n$ – базис в $\mathbb{R}^n$. Тогда для любой точки $x_0 \in \mathbb{R}^n$ вектор $x_* - x_0$ раскладывается по базису $x_* - x_0 = \alpha_1 p_1 + \cdots + \alpha_n p_n$. Таким образом, x_* представимо в виде

$$x_* = x_0 + \alpha_1 p_1 + \cdots + \alpha_n p_n$$

Каждое следующее приближение вычисляется по формуле:

$$x_k = x_0 + \alpha_1 p_1 + \cdots + \alpha_k p_k$$

<u>Определение</u> Два вектора p и q называются сопряжёнными относительно симметричной матрицы B, если $(Bp, q) = 0$

Опишем способ построения базиса $\{p_k\}_{k=1}^{n}$ в методе сопряжённых градиентов. В качестве начального приближения x_0 выбираем произвольный вектор. На каждой итерации α_k выбираются по правилу:

$$\alpha_k = \arg\min_{\alpha_k} F(x_{k-1} + \alpha_k p_k)$$

Базисные вектора $\{p_k\}$ вычисляются по формулам:

$$p_1 = -F'(x_0)$$

$$p_{k+1} = -F'(x_k) + \beta_k p_k$$

Коэффициенты β_k выбираются так, чтобы векторы p_k и p_{k+1} были сопряжёнными относительно A.

$$\beta_k = \frac{(F'(x_k), Ap_k)}{(Ap_k, p_k)}$$

Если обозначить за $r_k = b - Ax_k = -F'(x_k)$, то после нескольких упрощений получим окончательные формулы, используемые при применении метода сопряжённых градиентов на практике:

$$\begin{cases} r_1 = b - Ax_0 \\ p_1 = r_1 \\ \alpha_k = \dfrac{(r_k, r_k)}{(Ap_k, p_k)} \\ x_{k+1} = x_k + \alpha_k p_k \\ r_{k+1} = r_k - \alpha_k Ap_k \\ \beta_k = \dfrac{(r_{k+1}, r_{k+1})}{(r_k, r_k)} \\ p_{k+1} = r_{k+1} + b_k p_k \end{cases}$$

Для оценки скорости сходимости верна следующая грубая оценка:

$$\|x_k - x_*\| \le \frac{\sqrt{\kappa(A)} - 1}{\sqrt{\kappa(A)} + 1} \|x_0 - x\|, \text{где}$$

$\kappa(A) = \|A\| \cdot \|A^{-1}\| = \frac{\lambda_1}{\lambda_n}$. Она позволяет оценить скорость сходимости, если известны оценки для максимального λ_1 и минимального λ_n собственных значений матрицы A. На практике чаще всего используют следующий критерий останова:

$$\|r_k\| < \varepsilon \text{ или } \frac{\|r_k\|}{\|b\|} < \varepsilon.$$

ПРАКТИЧЕСКАЯ ЧАСТЬ

При хранении и преобразовании разреженных матриц в ЭВМ бывает полезно, а часто и необходимо, использовать специальные алгоритмы и структуры данных, которые учитывают разрежённую структуру матрицы. Операции и алгоритмы, применяемые для работы с обычными, плотными матрицами, применительно к большим разрежённым матрицам работают относительно медленно и требуют значительных объёмов памяти.

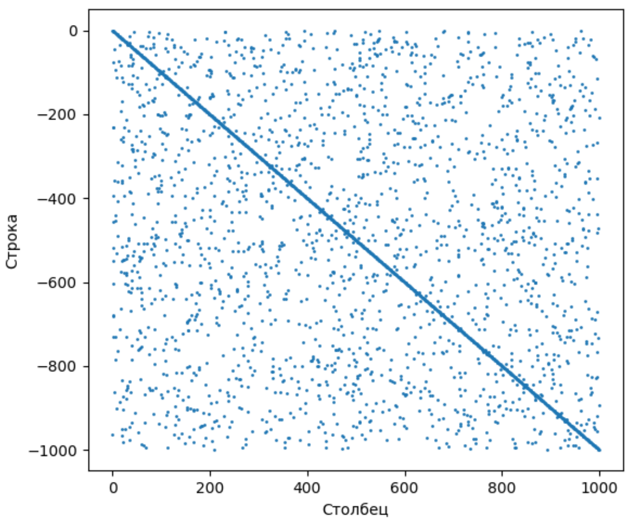
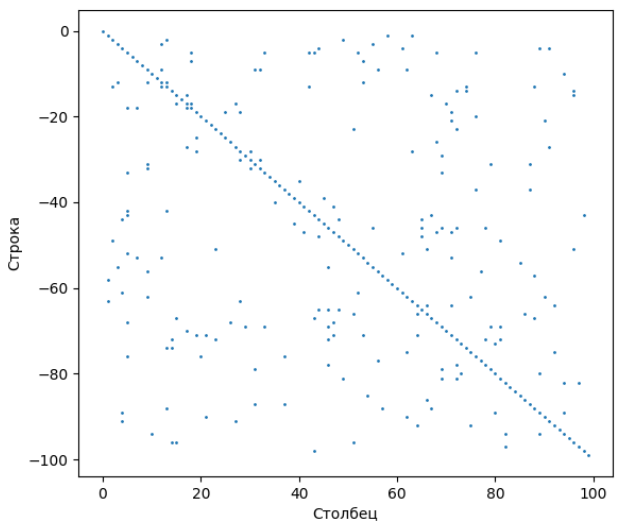
Для хранения разряженных матриц был использован принцип хеш-таблицы:

Ключ указывает на номер строки, а значение – индексы ненулевых элементов. Замечу, что добавление новых ключей в таблицу, происходит за $O(1)$

Для метода сопряжённых градиентов требуется симметричная положительно определённая матрица. Идея построения в следующем:

Сначала генерируем матрицу со случайными значениями из равномерного распределения. Затем получаем симметричную матрицу путем деления пополам суммы матрицы и транспонированной матрицы. Затем к диагональным элементам прибавляется случайное число подчиненное равномерному закону распределения. В итоге получаем искомую матрицу с заданными свойствами

РЕЗУЛЬТАТЫ

n = 1000 Матрица A: Число ненулевых элементов = 3000 eps = 1e-08 Число итераций = 131 Проверим решение: норма вектора Ax - b = 0.004441894252522616 Полученное значение Точное значение [[-185.59826771 -185.5982666] [-468.0983268 -468.09832764] [225.04986606 225.04986572] ... [364.93331807 364.93331909] [-172.98814372 -172.98814392] [-204.24918876 -204.24919128]]	
n = 100 Матрица A: Число ненулевых элементов = 296 eps = 1e-08 Число итераций = 40 Проверим решение: норма вектора Ax - b = 3.664408177428413e-05 Полученное значение Точное значение [[23.24657795 23.24657822] [-2.41845057 -2.41845059] [-45.83818801 -45.83818817] … [29.09274864 29.09274864] [45.86014576 45.86014557] [-32.28776965 -32.28776932]]	

ВЫВОД

Был реализован метод сопряженных градиентов для сопряженных матриц. Хранение разряженных матриц было реализовано на хеш-таблицах, доступ к которым получился за O(1). Решение было сравнено с точным, в результате чего убедились в правильности решения.

ЛИСТИНГ ПРОГРАММЫ

```
import numpy as np
from matplotlib import pyplot as plt


# Разреженная матрица
class SparseMatrix:
    def __init__(self, object={}, shape=None, dtype=np.float32):
        self.__data = object
        if len(object) > 0 and shape is None:
            self.__shape = (max(object) + 1, max(max(object.values(), key=lambda x: max(x))) + 1)
        elif shape is not None:
            self.__shape = shape
        else:
            self.__shape = (0, 0)

    # Доступ к элементу
    def __getitem__(self, key):
        if self.shape[0] == 1:  # вектор
            key = (0, key)
        if self.shape[1] == 1:  # вектор
            key = (key, 0)
        if isinstance(key, int):  # (int)
            if key in self.__data:
                return SparseMatrix({0: self.__data[key]}, shape=(1, self.shape[1]))
            else:
                return SparseMatrix(dict(), shape=(1, self.shape[1]))
        elif isinstance(key[0], int) and isinstance(key[1], int):  # (int, int)
            if key[0] in self.__data and key[1] in self.__data[key[0]]:
                return self.__data[key[0]][key[1]]
            elif key[0] < self.__shape[0] and key[1] < self.__shape[1]:
                return 0
            else:
                raise IndexError
        elif isinstance(key[0], int) and key[1].start is None and key[1].stop is None and key[
            1].step is None:  # (int, :)
            if key[0] in self.__data:
                return SparseMatrix({0: self.__data[key[0]]}, shape=(1, self.shape[1]))
            else:
                return SparseMatrix(dict(), shape=(1, self.shape[1]))
        elif key[0].start is None and key[0].stop is None and key[0].step is None and
isinstance(key[1],

int):  # (:, int)
            d = dict()
            for i in self.__data:  # цикл по строкам
                if key[1] in self.__data[i]:
                    d[i] = dict()
                    d[i][0] = self.__data[i][key[1]]
            return SparseMatrix(d, shape=(self.shape[0], 1))

    # Назначение по ключу
    def __setitem__(self, key, value):
        if value == 0:
            if key[0] in self.__data and key[1] in self.__data[key[0]]:
                del self.__data[key[0]][key[1]]
                if len(self.__data[key[0]]) == 0:
                    del self.__data[key[0]]
        else:
            if not key[0] in self.__data:
                self.__data[key[0]] = dict()
            self.__data[key[0]][key[1]] = value

    def __iter__(self):  # итератор на начало
        self.__ij = [0, 0]
        return self

    def __str__(self):
        return str(self.to_dense()) + "\nIn memory: " + str(self.__data)

    def __next__(self):
        m, n = self.shape
        i, j = self.__ij  # текущее значение
        if i < m and j < n:
            self.__ij[1] += 1
            return i, j
        elif i == m - 1 and j == n:
            raise StopIteration
```

```
        else:
            self.__ij = [i + 1, 0]
            return i + 1, 0

    @property
    def shape(self):
        return self.__shape

    # Вернуть массив значений
    @property
    def values(self):
        res = []
        for i in sorted(self.__data):
            for j in sorted(self.__data[i]):
                res += [self.__data[i][j]]
        return res

    # Вернуть индексы ненулевых значений
    @property
    def indices(self):
        x = []
        y = []
        for i in sorted(self.__data):
            for j in sorted(self.__data[i]):
                x += [j]
                y += [i]
        return x, y

    def transpose(A):
        d = {}
        for i in sorted(A.__data):
            for j in sorted(A.__data[i]):
                if not j in d:
                    d[j] = {}
                d[j][i] = A.__data[i][j]
        d_sort = {}
        return SparseMatrix(d_sort, shape=(A.shape[1], A.shape[0]))

    # Преобразовать в плотную матрицу
    def to_dense(self):
        res = np.zeros(self.shape)
        for i in sorted(self.__data):  # цикл по строкам
            for j in sorted(self.__data[i]):  # цикл по столбцам
                res[i, j] = self.__data[i][j]
        return res

    # Преобразовать в разреженную матрицу
    def to_sparse(A):
        d = {}
        for i in range(A.shape[0]):
            for j in range(A.shape[1]):
                if A[i, j] != 0:
                    if not i in d:
                        d[i] = {}
                    d[i][j] = A[i, j]
        return SparseMatrix(d, shape=A.shape)

    # Умножение матриц
    def dot(A, B, sparse=True):
        if isinstance(A, SparseMatrix) and isinstance(B, SparseMatrix):  # разреженная на
разреженную
            res = dict()
            b = dict()  # оставляем только столбцы
            for i in range(B.shape[1]):
                tmp = B[:, i]
                if len(tmp.__data) != 0:  # отбрасываем нулевые столбцы
                    b[i] = tmp
            for i in sorted(A.__data):  # по ненулевым строкам 1-й матрицы
                for j in b:  # по ненулевым столбцам 2-й матрицы
                    inter = set(A.__data[i]) & set(b[j].__data)  # пересечение
                    if len(inter) > 0:  # имеют общие элементы
                        s = 0
                        for k in inter:  # вычисление суммы
                            s += A.__data[i][k] * b[j][k]
                        if s != 0:
                            if not i in res:
                                res[i] = dict()
                            res[i][j] = s
            return SparseMatrix(res, shape=(A.shape[0], B.shape[1])) if sparse else
```

```
SparseMatrix(res, shape=(
            A.shape[0], B.shape[1])).to_dense()
        elif isinstance(A, SparseMatrix) and isinstance(B, np.ndarray):  # разреженная на плотную
            res = dict()
            for i in sorted(A.__data):  # по ненулевым строкам 1-й матрицы
                for j in range(B.shape[1]):  # по столбцам 2-й матрицы
                    s = 0
                    for k in sorted(A.__data[i]):  # по ненулевым столбцам 1-й матрицы
                        s += A.__data[i][k] * B[k][j]
                    if s != 0:
                        if not i in res:
                            res[i] = dict()
                        res[i][j] = s
            return SparseMatrix(res, shape=(A.shape[0], B.shape[1])) if sparse else
SparseMatrix(res, shape=(
            A.shape[0], B.shape[1])).to_dense()


# Равномерно-распределённая матрица
def uniform(shape=(10, 10), alpha=None, a=0, b=1):
    if alpha is None:
        alpha = 1 / shape[0]

    data = {}
    l = np.random.randint(0, shape[0], np.ceil(shape[0] * shape[1] * alpha).astype(int))
    m = np.random.randint(0, shape[1], np.ceil(shape[0] * shape[1] * alpha).astype(int))
    for i, j in zip(l, m):
        if not i in data:
            data[int(i)] = {}
        data[int(i)][int(j)] = np.random.uniform(a, b)
    return SparseMatrix(data, shape=shape)


# Генерирование положительно-определеннной и симметричной матрицы
def SPD(n=10, alpha=None, a=1, b=None):
    if alpha is None:
        alpha = 1 / n
    if b is None:
        b = n

    l = np.random.randint(0, n, np.ceil(n * n * alpha).astype(int))
    m = np.random.randint(0, n, np.ceil(n * n * alpha).astype(int))

    A = np.zeros(shape=(n, n))

    for i, j in zip(l, m):
        A[i, j] = np.random.uniform(0, 1)

    A = 0.5 * (A + np.transpose(A))
    A = A + np.diag(np.random.uniform(a, b, size=n))

    return SparseMatrix.to_sparse(A)


# Метод сопряжённых градиентов
def conjugate_gradient_method(A, b, x0 = None, norm = lambda x: np.linalg.norm(x, ord = np.inf),
eps = 1e-5):
    b = np.reshape(b, (-1, 1))
    x = b if x0 is None else np.reshape(x0, (-1, 1))
    r = b - SparseMatrix.dot(A, x, sparse = False)
    z = r
    b_norm = norm(b)
    r_dot_new = np.dot(np.reshape(r, -1), np.reshape(r, -1))
    i = 0
    while norm(r) / b_norm >= eps:
        r_dot_old = r_dot_new
        alpha = r_dot_old / np.dot(np.reshape(SparseMatrix.dot(A, z, sparse = False), -1),
np.reshape(z, -1))
        x = x + alpha * z
        r = r - alpha * SparseMatrix.dot(A, z, sparse = False)
        r_dot_new = np.dot(np.reshape(r, -1), np.reshape(r, -1))
        beta = r_dot_new / r_dot_old
        z = r + beta * z
        i += 1
    print("Число итераций =", i)
    return np.reshape(x, -1)


# Размер матрицы
```

```
n = 5
A = SPD(n)
print("n =", n)
x_true = np.random.uniform(- n / 2, n / 2, size = (A.shape[1], 1)).astype(np.float32)
b = np.reshape(SparseMatrix.dot(A, x_true, sparse = False), -1)
x, y = A.indices
d = A.values
print("Матрица A:")
print("Число ненулевых элементов =", len(A.values))

plt.scatter(x, -np.array(y))
plt.xlabel("Строка")
plt.ylabel('Столбец')
plt.show()

eps = 1e-8
print("eps =", eps)
x = conjugate_gradient_method(A, b, eps = eps)
print("Проверим решение:")
A_dense = A.to_dense()
print("||Ax - b|| =", np.linalg.norm(np.dot(A_dense, x) - b, ord = np.inf))
print("res\t true\n", np.concatenate((np.reshape(x, (-1, 1)), np.reshape(x_true, (-1, 1))), axis
= 1))
```