Одеський національний університет імені І. І. Мечникова

(повне найменування вищого навчального закладу)

Інститут математики, економіки і механіки

(повне найменування інституту/факультету)

Кафедра обчислювальної математики

(повна назва кафедри)

Д и п л о м н а р о б о т а

бакалавра

(освітньо-кваліфікаційний рівень)

на тему: «Симетрична проблема власних значень. Паралельні методи обчислювання»

Виконав: студент денної форми навчання
спеціальності 8.04030101 Прикладна математика

Нікіфоров Олексій Анатолійович

(прізвище, ім'я, по-батькові)

Керівник кандидат фіз.-мат. наук, доцент, Вербіцький В.В.

(науковий ступінь, вчене звання, прізвище та ініціали, підпис)

Рецензент кандидат технічних наук., професор, Мороз В.В. _____

(науковий ступінь, вчене звання, прізвище та ініціали)

Рекомендовано до захисту:

Протокол засідання кафедри

№_____ від _____ р.

Завідувач кафедри

_____ _____

(підпис) (прізвище, ініціали)

Захищено на засіданні ЕК №_____

протокол №____ від _____ р.

Оцінка_____/___ ___/_____

(за національною шкалою, шкалою ECTS, бали)

Голова ЕК

_____ _____

(підпис) (прізвище, ініціали)

Одеса – 2017

Зміст

Вступ

Задача знаходження власних значень і власних векторів є базовою задачею математики. З такою обчислювальною проблемою доводиться стикатися, наприклад, при дослідженні коливань різних механічних систем, коливальних і електронних спектрів молекул і кристалів, при обробці зображень у методі головних компонент тощо.

Більшість сучасних алгоритмів вирішення повної проблеми власних значень працює у два етапи: перший етап полягає у редукції симетричної матриці до тридіагональної форми. Після цього вирішується задача для тридіагональної матриці, яка має привабливу структуру з точки зору розпаралелювання. Одним із самих швидких та високопаралельних алгоритмів для тридіагональної матриці є метод “Divide and Conquer”. Основну ідею алгоритму “Divide and Conquer” і його попередника - QR-алгоритму, який достатньо довго використовувався на практиці для цього класу задач, ми розглянемо у основній частині. Паралельну реалізацію цих алгоритмів можна знайти у бібліотеці програмування числової лінійної алгебри LAPACK (Linear Algebra Package). LAPACK використовує іншу бібліотеку BLAS і разом з нею є найбільш популярними в своєму класі.

Альтернативою цим методам є один з найстарших практичних методів пошуку власних значень - метод Якобі. Цей алгоритм програє у швидкості QR-алгоритму, але він не втратив актуальності за рахунок того, що він дає змогу знаходити малі власні значення з достатньо великою точністю.

Багатоядерні системи є стандартом в даний час не тільки в області суперкомп’ютерів, а й у настільних комп’ютерах і ноутбуках. Навіть мобільні телефони мають багатоядерні процесори. Тому паралельні алгоритми

можуть бути використані всюди.

Об'єктом дослідження є симетричні матриці.

Предметом дослідження є побудова високоточних паралельних алгоритмів пошуку власних значень на основі алгоритму Якобі.

Ціль роботи: отримання нових знань і практичного досвіду для створення ефективних паралельних алгоритмів пошуку власних значень для симетричних матриць.

Список літератури

[1] В. В. Вербіцький. Введення в числові методи алгебри / В. В. Вербіцький, В. В. Реут. — Одеський національний університет імені І. І. Мечникова, 2015. — P. 135–142.

[2] Дж. Голуб. Матричные вычисления. Пер. с англ. / Дж. Голуб, Ч. Ван Лоун. — 2 edition. — Москва "Мир", 1999. — P. 368–420.

[3] Дж. Деммель. Вычислительная линейная алгебра. Теория и приложения / Дж. Деммель. — Мир, 2001. — P. 206–274.

[4] Н. Н. Калиткин. Численные методы / Н. Н. Калиткин. — 1 edition. — Главная редакция физико-математической литературы "Наука", 1978. — P. 88–156.

[5] B. Parlett. The Symmetric Eigenvalue Problem / B. Parlett. — Society for Industrial and Applied Mathematics, 1987.

[6] B. Chapman. Using OpenMP: Portable Shared Memory Parallel Programming / B. Chapman, G. Jost, R. van der Pas. — The MIT Press; Scientific and Engin edition, 2007. — 8.

Додатки

Додаток А

Код програми алгоритму Якобі з очистками Ґівенса

```
function [e, iter] = jacobi3(A, epsilon)
n = size(A, 1);
iter = 1;
done = 0;
working = 1;
stat = working;
off = offDiagonalSum(A);
fnorm = sqrt(sum(diag(A).^2) +  off);
eps = epsilon * fnorm; % tol = fnorm * eps
offset = 1;
step = 2;
while (stat == working)
    sm = offDiagonalSum(A);
    for p = 1 + offset : step : n - 1
        q = p + 1;
        [A, value] = jacobi_rotation(A, p, q, eps);
        sm = sm  - value;
        stat = working;
```

```
        end
        if (sm < eps)
            e = diag(A);
            stat = done;
        end
        %rotated = hess(A);
        rotated = givens_trid(A);
        A = rotated;
        iter = iter + 1;
        offset = mod(offset + 1, 2);
    end
    end
    %calculates offdiagonal sum*
    function [sum] = offDiagonalSum(A)
    [m, n] = size(A);
    sum = 0;
    for i = 1 : m
        for j = i + 1 : n
            if i ~= j
                sum = sum + A(i, j) ^ 2;
            end
        end
    end
    sum = sum * 2;
    end
    %calculates jacobi rotation
    %for a given matrix A and indexes p, q
    function [D, changedValue]
        = jacobi_rotation(A, p, q, epsilon)
    D = A;
    n = size(A, 1);
```

```
    tau = ((D(q, q) - D(p, p))) / (2 * D(p, q));
    t = sign(tau) / (abs(tau) + sqrt(1 + tau ^ 2));
    c = 1.0 / sqrt(t * t + 1);
    s = c * t;
    if abs(tau) == 0
        c = sqrt(2)/2;
        s = c;
    end
% end
changedValue = 2* D(p, q)^2;
D(q, p) = (c ^ 2 - s ^ 2) * A(p, q) +
     s * c * (A(p, p) - A(q, q));
D(p, q) = D(q, p);
D(q, q) = c ^ 2 * A(q, q) +
     2 * c * s * A(q, p) + s ^ 2 * A(p, p);
D(p, p) = s ^ 2 * A(q, q) - 2 * c * s * A(q, p) +
     c ^ 2 * A(p, p);
for k = 1 : n
    if (k == p) || (k == q)
        continue
    end
    D(p, k) = c * A(p, k) - s * A(q, k);
    D(k, p) = D(p, k);
end
for k = 1 : n
    if (k == p) || (k == q)
        continue
    end
    D(q, k) = s * A(p, k) + c * A(q, k);
    D(k, q) = D(q, k);
end
```

```
end
%calculates full Givens reduction for a given matrix A
function D = givens_trid(A)
    D = A;
    N = size(A, 1);
    for r = 1: N - 2
        for i = r + 2 : N
            x = sqrt(A(r, r+1)^2 + A(r,i)^2);
            c = A(r, r+1)/x;
            s = A(r, i) / x;
            if x == 0
                c = 1;
                s = 0;
            end
            for j = 1 : N
                D(j, r+1) = c * A(j,r+1) + s * A(j,i);
                D(j, i) = -s * A(j,r+1) + c * A(j,i);
            end
            A = D;
            for j = 1 : N
                D(r+1, j) = c * A(r+1, j) + s * A(i, j);
                D(i, j) = -s * A(r+1, j) + c * A(i, j);
            end
            A = D;
        end
    end
end
```

sectionДодаток Б

Код програми алгоритму Якобі з паралельним упорядкуванням

```
#pragma once
#define omptest true
#include <vector>
#include <limits>
#include <cmath>
#ifdef omptest
#include <omp.h>
#endif

#include "matrix.h"
#include "timer.h"

namespace parallel_jacobi
{
template<typename T>
inline void minmax(T a, T b, T& min, T& max) {
if (a > b) {
        max = a;
        min = b;
}
else {
        min = a;
        max = b;
}
}

//=============================================================
```

```
// Functions to determine convergence
//=====================================================
class converge_off_threshold {
const double t;
double tlast;
public:
// eps = tol*||A||_F
converge_off_threshold(const double tolerance,
                       const matrix& mat)
        : t(tolerance * frobenius_norm(mat)) { }

bool not_converged(matrix& mat) {
        tlast = off_diagonal_magnitude(mat);
        return tlast > t;
}
};

class converge_max_iterations {
int i;
const int imax;
public:
converge_max_iterations(const int iterations)
        : imax(iterations), i(0) { }

bool not_converged(matrix& mat) {
        return ++i < imax;
}
};

#undef max;
class converge_off_difference {
```

```
double lastnorm;
double lastnorm2;
const double t;
public:
converge_off_difference(const double tolerance)
        : t(tolerance)
        , lastnorm(std::numeric_limits<double>::max()) { }

bool not_converged(matrix& mat) {
        double o = off_diagonal_magnitude(mat);

        if ((lastnorm - o)<t)
                return false;

        lastnorm2 = lastnorm;
        lastnorm = o;
        return true;
}
};

class music_permutation {
std::vector<int> top, bot;
const int n;
public:
music_permutation(int n) : n(n) {
        const int m = n / 2;
        top.resize(m);
        bot.resize(m);
        for (int i = 0; i<m; ++i) {
                top[i] = 2 * i;
                bot[i] = 2 * i + 1;
```

```
        }
}

inline void get(int k, int& p, int& q) {
        minmax(top[k], bot[k], p, q);
}

void permute() {
        // no permutation possible for 2x2 matrix
        if (n == 2) return;
        int m = n / 2;
        // Store end element to move down
        // after bottom row is shifted.
        int e = top[m - 1];
        top[0] = bot[0];
        for (int k = 0, l = m - 1; k<m - 1; ++k, --l) {
                bot[k] = bot[k + 1];
                top[l] = top[l - 1];
        }
        // Move stored end element down to bot.
        bot[m - 1] = e;
        // Fix first top value.
        top[0] = 0;
}
};

template<typename T>
void symmetric_schur_new(const matrix& A,
        const unsigned int p,
        const unsigned int q, T& c, T& s)
{
```

```
const T epsilon = 1e-3;
if (fabs(A(p, q)) > epsilon)
{
T tau = (A(q, q) - A(p, p)) / (2.0 * A(p, q));
T t;
if (tau >= 0) t = 1.0 / (tau + sqrt(1.0 + tau*tau));
else t = -1.0 / (-tau + sqrt(1.0 + tau*tau));
c = 1.0 / sqrt(1.0 + t*t);
s = t*c;
}
else
{
        c = 1.0;
        s = 0.0;
}
}

//=============================================================
// Pre-multiply Jacobi rotation matrix
//=============================================================
template<typename T>
inline void premultiply(matrix& mat, const int p,
         const int q, const T c,
const T s)
{
typedef matrix::value_type value_type;
const int n = mat.size();
value_type *rowp = mat.get_row(p),
        *rowq = mat.get_row(q);
for (int i = 0; i<n; ++i, ++rowp, ++rowq)
{
```

```
        const value_type mpi = *rowp,
                         mqi = *rowq;
        *rowp = c*mpi + -s*mqi;
        *rowq = s*mpi + c*mqi;
}
}

//=============================================================
// Post-multiply Jacobi rotation matrix
//=============================================================
template<typename T>
inline void postmultiply(matrix& mat, const int p,
        const int q, const T c,
const T s)
{
typedef matrix::value_type value_type;
for (int i = 0; i<mat.size(); ++i)
{
        const value_type mip = mat(i, p),
                         miq = mat(i, q);
        mat(i, p) = c*mip + -s*miq;
        mat(i, q) = s*mip + c*miq;
}
}

//=============================================================
// Run the parallel jacobi algorithm
// mat - the matrix to operate on
// sc - a class with a function not_converged(mat)
// to determine when to stop
// the algorithm.
```

```
//==========================================================
template<class StoppingCriterion, class Permutation>
void run(matrix& mat, StoppingCriterion& sc,
        Permutation& pe, int &iter)
{
typedef matrix::value_type value_type;
//omp_set_num_threads(8);
const int n = mat.size();
const int m = n / 2;
iter = 0;
value_type* si = new value_type[m];
value_type* co = new value_type[m];
const bool isodd = (n > mat.actual_size());
bool not_converged = true;

//root.start();
while (not_converged) {

        for (int set = 0; set<n; ++set) {
#ifdef omptest
#pragma omp parallel for default(none)
                shared(mat, n, si, co, pe, isodd)
#endif
        for (int k = 0; k<m; ++k) {
                int p, q;
                pe.get(k, p, q);
        if (isodd && (p == n || q == n)) continue;
        symmetric_schur_new(mat, p, q, co[k], si[k]);
        premultiply(mat, p, q, co[k], si[k]);
        }
#ifdef omptest
```

```
#pragma omp parallel for default(none)
                shared(mat, n, si, co, pe, isodd)
#endif
        for (int k = 0; k<m; ++k) {
                int p, q;
                pe.get(k, p, q);
                if (isodd && (p == n || q == n)) continue;

                postmultiply(mat, p, q, co[k], si[k]);
        }
        pe.permute();
        }
        not_converged = sc.not_converged(mat);
        iter++;
}
delete[] si;
delete[] co;
}
}
inline double frobenius_norm(const matrix& mat)
{
typedef matrix::value_type value_type;
const int n = mat.size();
double mag = 0.0;
for (int i = 0; i<n; ++i)
{
        const value_type *a = mat.get_row(i);
        for (int j = 0; j<n; ++j, ++a)
                mag += (*a)*(*a);
}
return std::sqrt(mag);
```

```
}
inline double off_diagonal_magnitude(const matrix& mat)
{
typedef matrix::value_type value_type;
const int n = mat.size();
double mag = 0.0;
for (int i = 0; i<n; ++i)
{
        const value_type *a = mat.get_row(i);
        for (int j = 0; j<n; ++j, ++a)
        {
                if (i != j) mag += (*a)*(*a);
        }
}
return std::sqrt(mag);
}
```