Санкт-Петербургский государственный университет
Факультет прикладной математики-процессов управления
Кафедра теории управления

Горнак Дарья Дмитриевна

Магистерская диссертация

Построение матриц Ляпунова для систем со многими запаздываниями

Направление 01.04.02 – “Прикладная математика и информатика”
Магистерская программа
“Методы прикладной математики и информатики в задачах управления”

Научный руководитель, Харитонов В.Л.
доктор физ.-мат. наук,
профессор

Рецензент,
кандидат физ.-мат. наук Сумачева В. А.

Санкт-Петербург
2017 г

Содержание

Введение

Функционал и матрица Ляпунова

Многочисленные процессы, основанные на передаче информации, энергии, массы и т.д, сопровождаются запаздыванием. Это запаздывание может быть обусловлено самыми разными причинами: например, электрический сигнал или ограниченность скорости протекания технологических процессов и т.п. Неучитывание запаздывания в моделях может привести к существенным ошибкам в расчетах. Например, рассмотрим следующую систему дифференциальных уравнений с запаздыванием:

$$\frac{dx(t)}{dt} = \sum_{j=0}^{m} A_j x(t - jh)\,, \qquad t \geq 0\,, \qquad (1)$$

где A_j - данные вещественные матрицы $n \times n$, и h положительное запаздывание. Для исследования устойчивости и поведения систем с запаздыванием применяется функционал Ляпунова-Красовского полного вида:

$$v_0(\varphi) = \varphi^T(0)U(0)\varphi(0) + \sum_{j=1}^{m} 2\varphi^T(0) \int_{-h_j}^{0} U\big(-h_j - \theta\big)A_j\varphi(\theta)d\theta + \sum_{k=1}^{m}\sum_{j=1}^{m} \int_{-h_k}^{0} \varphi^T(\theta_1){A_k}^T \left[\int_{-h_j}^{0} U\big(\theta_1 + h_k - \theta_2 - h_j\big)A_j\varphi(\theta_2)d\theta_2\right] d\theta_1,$$

где начальная функция $\varphi : [-mh, 0] \rightarrow R^n$.

Пусть

$$U(\tau) = \int_0^{\infty} K^T(t)WK(t + \tau)dt\,,$$

Для системы (1) матрица $U(\tau)$ называется матрицей Ляпунова, связанной с матрицей W. Матрица K(t) – является фундаментальной матрицей.

Определение. Матрица K(t) размерности n×n называется фундаментальной матрицей системы (1), если она удовлетворяет матричному уравнению

$$\frac{d}{dt}K(t) = \sum_{j=0}^{m} K(t-h_j)A_j\,, \qquad t \geq 0,$$

С начальными условиями $K(t) = 0_{n\times n}$ для $t < 0$, $K(0) = I$.

В рамках работы рассматривается алгоритм построение матрицы Ляпунова для систем дифференциальных уравнений с несколькими запаздываниями. Рассмотрим ситуации, когда появляются системы с большим количеством запаздываний.

Системы с большим количеством запаздываний

Рассмотрим систему следующего вида:

$$\frac{dx(t)}{dt} = A_0x(t) + A_1x(t-h_1) + A_2x(t-h_2), \qquad t \geq 0\,,$$

где $h_1 = {}^{p_1}\!/_{q_1}, h_2 = {}^{p_2}\!/_{q_2}$. Тогда, чтобы применить алгоритм вычисления матрицы Ляпунова, нужно привести h_k к виду $h_k = kh$. Для этого посчитаем $N = \text{НОК}(q_1, q_2)$, и тогда

$$h_1 = {}^{p_1}\!/_{q_1} = b_1 \cdot h,$$

$$h_2 = {}^{p_2}\!/_{q_2} = b_2 \cdot h,$$

где $h = {}^{1}\!/_{N}$, b_j – целое число. После такой замены количество запаздываний станет не 2 как в первоначальной системе, а больше. Например, если $h_1 = 3/4\,, h_2 = 5/9$, то НОК(4,9) = 36, следовательно, $h = 1/36\,, b_1 = 20,\ b_2 = 27$. Отсюда видно, что количество запаздываний становится равно 27. Все A_j, кроме двух, данных в первоначальной системе, будут нулевыми, но при этом матрицы промежуточных вычислений будут сильно разреженными. В результате возникает проблема хранения и работы с разреженными матрицами. В рамках работы будут изучены методы

хранения и работы с разреженными матрицами, а так же будет реализован алгоритм вычисления матрицы Ляпунова.

Обзор литературы

Системы дифференциальных уравнений с запаздывающим аргументом со второй половины прошлого века были описаны во многих работах [3]. Одним из главных методов исследования таких систем является метод функционалы Ляпунова-Красовского. В данном функционале вводится матрица Ляпунова. Работа [1] посвящена матрице Ляпунова. Алгоритм ее получения так же приведен в этой работе.

Работы [7,8] исследуются для нахождения численных способов решения систем линейных алгебраических уравнений.

В работе [2] приведен алгоритм вычисления экспоненты от матрицы, использующий n степеней матрицы.

В работе [9] рассматриваются разреженные матрицы, а так же алгоритмы хранения и методы работы с ними.

Работы [4,5,6,10] рассматриваются для реализации программных алгоритмов с наименьшей потерей памяти и времени работы программы. Так же в данных ссылках находится документация по языкам программирования, для лучшего использования всех доступных возможностей выбранных языков.

1 Постановка задачи

Рассмотрим систему

$$\frac{dx(t)}{dt} = \sum_{j=0}^{m} A_j x(t - \text{jh}), t \geq 0 \qquad (1.1)$$

где A_j - данные вещественные матрицы $n \times n$, и h положительное запаздывание.

Зададим начальную функцию φ : $[-mh, 0] \rightarrow R^n$.

Обозначим решение соответствующие этой начальной функции как $x(\theta, \varphi)$.

$$x(\theta, \varphi) = \varphi(\theta), \qquad \theta \in [-mh, 0]$$

И пусть $x_t(\varphi)$ обозначает ограниченное решение на отрезке $[t - mh, t]$

$$x_t(\varphi): \theta \rightarrow x(t + \theta, \varphi), \theta \in [-mh, 0].$$

Введем определение матрицы Ляпунова для системы (1.1)

Определение: Матрица $U(\tau)$ называется матрицей Ляпунова для системы (1.1) связанной с симметрической матрицей W если она удовлетворяет свойствам :

1. Динамическое свойство

$$\frac{d}{d\tau} U(\tau) = \sum_{j=0}^{m} U(\tau - \text{jh}) A_j, \tau \geq 0.$$

2. Симметрическое свойство

$$U(-\tau) = U^T(\tau), \tau \geq 0$$

3. Алгебраическое свойство

$$\sum_{j=0}^{m} [U(-\text{jh}) A_j + A_j^T U(\text{jh})] = -W$$

В работе предлагается алгоритм построение этой матрицы с акцентом на большое количество запаздываний.

2 Алгоритм вычисления матрицы Ляпунова

Определим вспомогательные матрицы [1] вида

$$Y_j(\tau) = U(jh + \xi), \qquad \xi \in [0, h]$$

Пусть матрица Ляпунова связана с симметрической матрицей W, тогда вспомогательные матрицы удовлетворяют системе обыкновенных дифференциальных матричных уравнений:

$$\begin{cases} \dfrac{d}{d\xi} Y_j(\xi) = \sum\limits_{k=0}^{m} Y_{j-k}(\xi) A_k, \qquad j = 0,1,..,m-1 \\ \dfrac{d}{d\xi} Y_{-j}(\xi) = -\sum\limits_{k=0}^{m} A_k^T Y_{-j+k}(\xi), \qquad j = 1,..,m \end{cases}$$

И граничным условиям

$$Y_j(0) = Y_{j-1}(h), \qquad j = -m+1, -m+2, \ldots, 0, \ldots, m-1,$$

$$-W = \sum_{k=0}^{m-1} [Y_{-k}(0) A_k + A_k^T Y_k(0)] + A_m^T Y_{-m}(0) + Y_{m-1}(h) A_m.$$

Для начала необходимо определить произведение матриц Кронекера

Определение. $A \otimes B$ называется произведением Кронекера, если

$$A \otimes B = \begin{pmatrix} b_{11}A & b_{21}A & \cdots & b_{n1}A \\ b_{12}A & b_{22}A & \cdots & b_{n2}A \\ \vdots & \vdots & \ddots & \vdots \\ b_{1n}A & b_{21}A & \cdots & b_{nn}A \end{pmatrix}$$

Для векторизации матриц столбцы матрицы записываются друг под другом, и получается вектор обозначаемый vec.

Для упрощения вычислений запишем систему и граничные условия в векторной форме.

$$\frac{d}{d\tau} \begin{pmatrix} y_{-m}(\tau) \\ y_{-m+1}(\tau) \\ \vdots \\ y_{m-2}(\tau) \\ y_{m-1}(\tau) \end{pmatrix} = L \begin{pmatrix} y_{-m}(\tau) \\ y_{-m+1}(\tau) \\ \vdots \\ y_{m-2}(\tau) \\ y_{m-1}(\tau) \end{pmatrix}, \qquad (2.1)$$

$$L = \begin{pmatrix} -A_0^T \otimes I & -A_1^T \otimes I & \cdots & \cdots & -A_m^T \otimes I & 0 & \cdots & 0 \\ 0 & -A_0^T \otimes I & \cdots & \cdots & -A_{m-1}^T \otimes I & -A_m^T \otimes I & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & \cdots & -A_0^T \otimes I & -A_1^T \otimes I & -A_2^T \otimes I & \cdots & -A_m^T \otimes I \\ I \otimes A_m & I \otimes A_{m-1} & \cdots & \cdots & I \otimes A_0 & 0 & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & \cdots & I \otimes A_m & I \otimes A_{m-1} & \cdots & \cdots & I \otimes A_o \end{pmatrix}$$

где $y_j(\tau) = vec\left(Y_j(\tau)\right), j = -m, -m+2, \ldots, 0, \ldots, m-1.$

$$M \begin{pmatrix} y_{-m}(0) \\ \cdots \\ y_{m-1}(0) \end{pmatrix} + N \begin{pmatrix} y_{-m}(h) \\ \cdots \\ y_{m-1}(h) \end{pmatrix} = -\begin{pmatrix} w \\ \cdots \\ 0 \end{pmatrix} \qquad (2.2)$$

где w=vec(W) и пусть $O = 0_{n \times n}$ тогда

$$M = \begin{pmatrix} A_m^T \otimes I & I \otimes A_{m-1} & \cdots & A_0^T \otimes I + I \otimes A_0 & A_1^T \otimes I & \cdots & A_{m-1}^T \otimes I \\ O \otimes O & I \otimes I & O \otimes O & \cdots & \cdots & \cdots & O \otimes O \\ O \otimes O & O \otimes O & I \otimes I & \cdots & \cdots & \cdots & O \otimes O \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ O \otimes O & \cdots & \cdots & \cdots & \cdots & \cdots & I \otimes I \end{pmatrix}$$

$$N = \begin{pmatrix} O \otimes O & O \otimes O & O \otimes O & \cdots & I \otimes A_m \\ -I \otimes I & O \otimes O & \cdots & \cdots & O \otimes O \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ O \otimes O & \cdots & O \otimes O & -I \otimes I & O \otimes O \end{pmatrix}$$

Из системы (2.1) следует, что

$$\begin{pmatrix} y_{-m}(h) \\ \cdots \\ y_{m-1}(h) \end{pmatrix} = e^{Lh} \begin{pmatrix} y_{-m}(0) \\ \cdots \\ y_{m-1}(0) \end{pmatrix}$$

Заменим в равенстве (2.2) вектор при N, получаем

$$[M + Ne^{Lh}] \begin{pmatrix} y_{-m}(0) \\ \cdots \\ y_{m-1}(0) \end{pmatrix} = -\begin{pmatrix} w \\ \cdots \\ 0 \end{pmatrix}$$

Предполагается, что система имеет решение, тогда это решение соответствует системе (2.2).

$$\begin{pmatrix} y_{-m}(\tau) \\ \cdots \\ y_{m-1}(\tau) \end{pmatrix} = e^{Lh} \begin{pmatrix} y_{-m}(0) \\ \cdots \\ y_{m-1}(0) \end{pmatrix}$$

Так как

$$Y_j(\tau) = U(jh + \xi), \qquad \xi \in [0, h]$$

Решение вспомогательной системы можно использовать для вычисления матрицы Ляпунова для системы (1.1).

Теорема

Система (1.1) удовлетворяет условиям Ляпунова тогда и только тогда, когда

$$\det(M + Ne^{Lh}) \neq 0$$

Доказательство теоремы приведено в [1].

Как можно увидеть из представленного выше алгоритма, вычисление матрицы Ляпунова сводится к следующим матричным операциям:

1. Сложение.
2. Умножение.
3. Произведение Кронекера.
4. Вычисление матричной экспоненты.

3 Работа с разреженными матрицами

При вычислении матрицы Ляпунова (особенно при большом количестве запаздываний) возникает работа с разреженными матрицами. **Определение.** Разреженной матрицей называют матрицу с преимущественно нулевыми элементами.

Для хранения и работы с этими матрицами используются специальные алгоритмы.

3.1 Хранение разреженной матрицы

Существуют различные форматы хранения разреженных матриц. Одни предназначены для хранения матриц специального вида (например, ленточных), другие обеспечивают работу с матрицами общего вида. В работе будет использоваться разреженный строчной формат [9].

Разреженный строчный формат - это одна из наиболее широко используемых схем хранения разреженных матриц. Эта схема предъявляет минимальные требования к памяти и в то же время оказывается очень удобной для нескольких важных операций над разреженными матрицами: сложения, умножения, перестановок строк и столбцов, транспонирования, решения линейных систем с разреженными матрицами коэффициентов как прямыми, так и итерационными методами и т. д.

В соответствии с рассматриваемой схемой для хранения матрицы A требуется три одномерных массива:

1. массив ненулевых элементов матрицы A, в котором они перечислены по строкам от первой до последней (обозначим его опять как values);
2. массив номеров столбцов для соответствующих элементов массива values (обозначим его как cols);
3. массив указателей позиций, с которых начинается описание очередной строки (обозначим его pointer). Описание k-й строки хранится в позициях с pointer[k]-й по (pointer[k+1]–1)-ю

массивов values и cols. Если pointer[k]=pointer[k+1], то k-я строка пустая. Если матрица A состоит из n строк, то длина массива pointer будет n+1.

Данный способ представления также является полным, и упорядоченным, поскольку элементы каждой строки хранятся в соответствии с возрастанием столбцовых индексов.

Для примера рассмотрим представление матрицы в разреженном строчном формате:

$$A = \begin{bmatrix} 1 & -1 & 0 & -3 & 0 \\ -2 & 5 & 0 & 0 & 0 \\ 0 & 0 & 4 & 6 & 4 \\ -4 & 0 & 2 & 7 & 0 \\ 0 & 8 & 0 & 0 & -5 \end{bmatrix}$$

values	(1, -1, -3, -2, 5, 4, 6, 4, -4, 2, 7, 8, -5);
cols	(1, 2, 4, 1, 2, 3, 4, 5, 1, 3, 4, 2, 5);
pointer	(1, 4, 6, 9, 12, 14).

Разреженный строчный формат обеспечивает эффективный доступ к строчкам матрицы; доступ к столбцам по-прежнему затруднен. Поэтому предпочтительно использовать этот способ хранения в тех алгоритмах, в которых преобладают строчные операции.

3.2 Вычисление матричной экспоненты

Матричную экспоненту можно вычислить по формуле

$$e^{At} = E + \sum_{i=1}^{\infty} \frac{(At)^i}{i!} = \sum_{i=0}^{\infty} \frac{(At)^i}{i!}, \qquad (3.1)$$

где **E** — единичная матрица, **t** — время, связано с необходимостью расчета высоких степеней матрица **A**. Получим формулу, позволяющую определить матричную экспоненту с помощью **n** степеней матрицы **A** (**n** — ее порядок). Пусть характеристический полином матрицы **A** имеет вид [2]:

$$f(\lambda) = \lambda^n - p_1\lambda^{n-1} - p_2\lambda^{n-2} - \cdots - p_n \quad (3.2)$$

По теореме Гамильтона-Кэли матрица **A** удовлетворяет равенству:

$$A^n - p_1 A^{n-1} - p_2 A^{n-2} - \cdots - p_n E = 0,$$

откуда

$$A^n = p_1 A^{n-1} + p_2 A^{n-2} + \cdots + p_n E. \qquad (3.3)$$

Следуя методу Д.К. Фаддеева, коэффициент $p_1 = Sp\ A$, остальные же коэффициенты характеристического полинома определяются по рекуррентному соотношению

$$p_k = \frac{s_k - p_1 s_{k-1} - \cdots - p_{k-1} s_1}{k}, \quad k = \overline{2, n}$$

где $s_k = Sp\ A^k$— след матрицы A^k (сумма элементов, стоящих на главной диагонали).

Далее введем обозначение: $q_{0,k} = p_k$; иначе для m>0 (при натуральном m)

$$q_{m,k} = p_k q_{m-1,1} + q_{m-1,k+1}, \quad (3.4)$$

где полагаем что $q_{m-1,n+1} = 0$.

Умножим обе части соотношения (3.3) на матрицу **A** с учетом введенных обозначений. Получим

$$\begin{aligned} A^{n+1} &= q_{0,1} A^n + q_{0,2} A^{n-1} + \cdots + q_{0,n} A \\ &= (p_1 q_{0,1} + q_{0,2}) A^{n-1} + (p_2 q_{0,1} + q_{0,3}) A^{n-2} + (p_3 q_{0,1} + q_{0,4}) A^{n-3} \\ &+ \cdots + (p_{n-1} q_{0,1} + q_{0,n}) A + p_n q_{0,1} E. \qquad (3.5) \end{aligned}$$

Выражение (3.5) можно переписать как

$$A^n = q_{1,1} A^{n-1} + q_{1,2} A^{n-2} + \cdots + q_{1,n} E. \qquad (3.6)$$

Теперь умножим обе части равенства (3.6) на матрицу **A**, подставив при этом в полученное соотношение формулу (3.3):

$$\begin{aligned} A^{n+2} &= q_{1,1} A^n + q_{1,2} A^{n-1} + \cdots + q_{1,n} A \\ &= (p_1 q_{1,1} + q_{1,2}) A^{n-1} + (p_2 q_{1,1} + q_{1,3}) A^{n-2} + (p_3 q_{1,1} + q_{1,4}) A^{n-3} \\ &+ \cdots + (p_{n-1} q_{1,1} + q_{1,n}) A + p_n q_{1,1} E. \qquad (3.7) \end{aligned}$$

Тогда из выражения (3.7) с помощью последовательного умножения на матрицу **A** обеих его частей следует, что

$$A^{n+m} = q_{m,1}A^{n-1} + q_{m,2}A^{n-2} + \cdots + q_{m,n}E = \sum_{k=0}^{n-1} A^k q_{m,n-k}$$

Теперь представим матричную экспоненту как

$$e^{At} = \sum_{k=0}^{n-1} A^k \frac{t^k}{k!} + \sum_{m=0}^{\infty} \frac{t^{n+m}}{(n+m)!} \sum_{k=0}^{n-1} A^k q_{m,n-k}$$

$$= \sum_{k=0}^{n-1} A^k \frac{t^k}{k!} + \sum_{k=0}^{n-1} A^k \sum_{m=0}^{\infty} q_{m,n-k} \frac{t^{n+m}}{(n+m)!} .$$

Откуда имеем

$$e^{At} = \sum_{k=0}^{n-1} A^k \left[\frac{t^k}{k!} + \sum_{m=0}^{\infty} \frac{q_{m,n-k}}{(n+m)!} t^{n+m} \right]$$

$$= \sum_{k=0}^{n-1} A^k \left[\frac{t^k}{k!} + \sum_{m=0}^{\infty} r_{m,k} t^{n+m} \right]. \quad (3.8)$$

В формуле суммирование идет до бесконечности. При вычислении же экспоненты нужно определять момент остановки суммирования. Он зависит от точности, которую нужно получить. Рассмотрим критерий остановки алгоритма. Алгоритм должен остановиться, когда достигнет такого N, при котором ошибка вычисления экспоненты будет ниже заданной точности ε.

$$\sum_{m=0}^{\infty} \frac{q_{m,n-k}}{(n+m)!} t^{n+m} = \sum_{m=0}^{N} \frac{q_{m,n-k}}{(n+m)!} t^{n+m} + \sum_{m=N+1}^{\infty} \frac{q_{m,n-k}}{(n+m)!} t^{n+m}$$

Положим $t = h$. И запишем оценку ошибки экспоненты

$$\left\| \sum_{k=0}^{n-1} A^k \sum_{m=N+1}^{\infty} \frac{q_{m,n-k}}{(n+m)!} h^{n+m} \right\| =$$

$$\sum_{k=0}^{n-1}\|A\|^k \sum_{m=N+1}^{\infty}\frac{q_{m,n-k}}{(n+m)!}h^{n+m} < \varepsilon \qquad (3.9)$$

Ошибка меньше заданного ε. Следовательно, из формулы (3.9) можно выразить насколько маленькой должна быть сумма. Тогда введем утверждение.

Утверждение.

$$\sum_{m=N+1}^{\infty}\frac{q_{m,n-k}}{(n+m)!}h^{n+m} < \frac{\varepsilon}{\sum_{k=0}^{n-1}\|A\|^k}$$

для любого заданного ε.

Указанное выше утверждение позволяет нам определить N, для заданной точности вычисления экспоненты.

Алгоритм вычисления экспоненты, описанный выше, применяется в связи с тем, что в ходе решения появляются матрицы больших размерностей. Он обеспечивает вычисление матричной экспоненты, используя только n степеней матрицы, в то время как классический алгоритм не детерминирован, и вычисления степеней матрицы продолжаются, пока не выполнится условие выхода из алгоритма.

4 Реализация программы

В данном разделе описана реализованная в рамках работы программа для вычисления матрицы Ляпунова (uml-схема рис 1). Программа реализована на языке программирования C++, с использованием метода ООП. Данный язык программирования был выбран по нескольким причинам. Во-первых, при реализации алгоритма вычисления матрицы в программе Matlab матрица Ляпунова была вычислена только до размерности матриц 5×5 и 5 запаздываний, в C++ написана длинная арифметика для повышения точности вычисления и размерности матриц. Во-вторых, в дальнейшем появиться возможность распараллеливания программы.

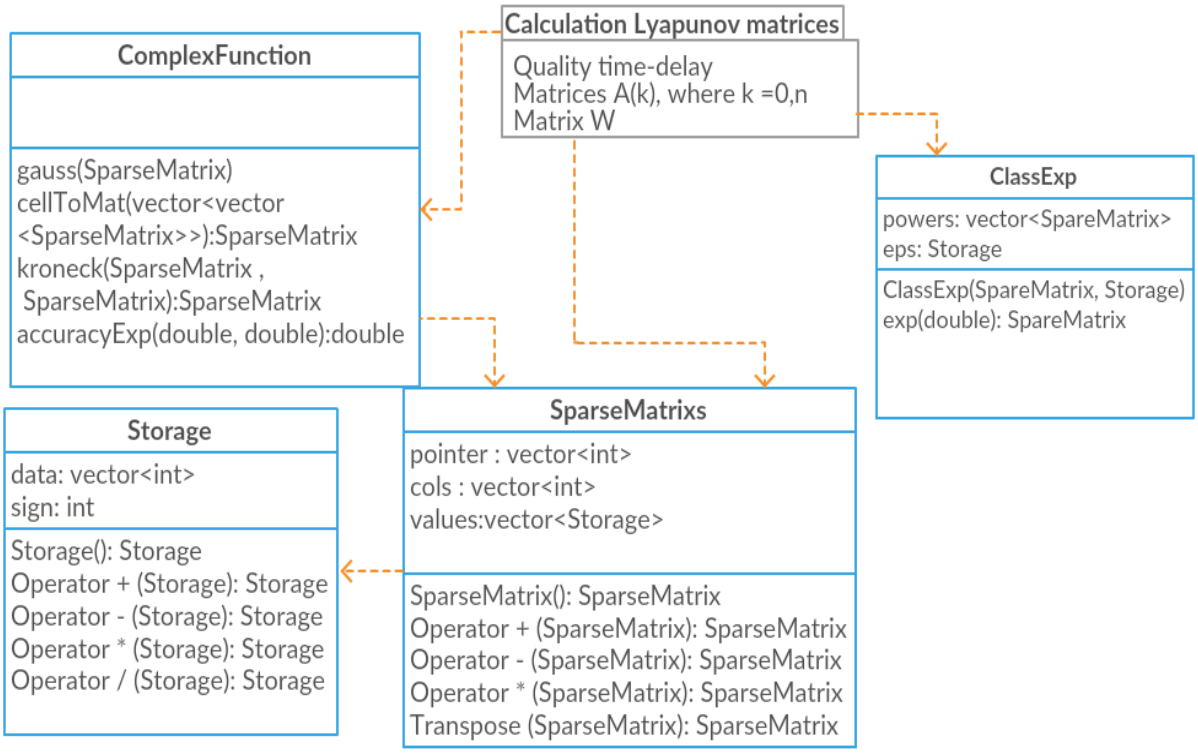


Рис.1. Схема реализации программы

4.1 Пакет Caluculation Lyapunova

В данном пакете реализован алгоритм вычисления матрицы Ляпунова. На пакету поступает текстовый файл. В нем в первой строке указываются три числа: n – размер матриц A, m – количество запаздываний и h – начальное запаздывание. Дальше пропускается строка, и вводятся m+2

матрицы. Возвращает пакет матрицу Ляпунова, вычисленную на всем промежутке $[-mh, 0]$.

Пример входного файла для матриц 3×3 и количества запаздываний 3:

```
3 3 1

0.7 4 0.1
0.1 1 0.5
0.9 5 1


0.2 0.5 0.2
6 1 4
4 5 0.9


0.2 0.5 0.1
6 1 0.5
1 2 0.6


0.4 0.7 0.1
6 5 0.1
9 2 0.3

1 1 1
1 1 1
1 1 1
```

4.2 Класс SpareMatrix

Данный класс хранит разреженные матрицы и выполняет операции с ними. Матрица хранится в трех массивах в виде, описанном в пункте 3.1.

В классе присутствуют следующие методы:

1) *SparseMatrix()* – Конструктор класса, на вход ничего не принимает.

2) *SparseMatrix(int size, bool l)* – конструктор класса, на вход принимает размер матрицы и флаг, какая эта матрица единичная или нулевая.

3) *SparseMatrix operator+(SparseMatrix left, SparseMatrix right)* – метод класса, переопределяющий операцию сложения для двух элементов класса.

4) *SparseMatrix operator-(SparseMatrix left, SparseMatrix right)* – метод класса, переопределяющий операцию вычитания для двух элементов класса.

5) *SparseMatrix operator*(SparseMatrix left, SparseMatrix right)* – метод класса, переопределяющий операцию усножения для двух элементов класса.

6) *SparseMatrix operator*(double left, SparseMatrix right)* – метод класса, переопределяющий операцию умножения для элемента класса и числа с плавающей точкой.

7) *SparseMatrix operator-(SparseMatrix i)* – метод класса, переопределяющий операцию унарного минуса для элемента класса.

8) *SparseMatrix trans()* – метод класса, реализующий транспонирование матрицы.

9) *double matrixTrace()* – метод класса, реализующий вычисление следа матрицы.

10) *int size()* – метод класса, возвращающий размер матрицы.

11) *vector<vector<double>> sparseToMat()* – метод класса, восстанавливающий разреженную матрицу.

12) *double norm()* – метод класса, реализующий вычисление нормы матрицы.

Ниже более подробно описана реализация методов класса.

4.2.1 Сложение двух разреженных матриц

Данный метод класса перегружает операцию сложения между двумя разреженными матрицами. Алгоритм сложения двух разреженных матриц следующий:

1. Создать матрицу результата.
2. Первый цикл проходит по строкам матриц.
3. Второй цикл идет до тех пор, пока в массиве значений обеих матриц не закончится рассматриваемая строка. В этом цикле присутствуют следующие проверки:
    a. Если элементы матриц находятся в одном столбце, то сложить их и добавить в массив значений матрицы результата, а также добавить номер столбца в массив столбцов. И увеличить счетчики массива элементов обеих матриц.
    b. Если столбец, в котором находится элемент левой матрицы, меньше столбца, в котором находиться элемент правой матрицы, то добавить в матрицу результата элемент левой матрицы, а также добавить в массив столбцов номер данного столбца. И увеличить счетчик массива элемента левой матрицы.
    c. Если столбец, в котором находится элемент правой матрицы, меньше столбца, в котором находиться элемент левой матрицы, то добавить в матрицу результата элемент правой матрицы, а также добавить в массив столбцов номер данного столбца. И увеличить счетчик массива элемента правой матрицы.
4. После завершения второго цикла добавить в массив строк матрицы результата размер массива столбцов этой матрицы.
5. После завершения первого цикла в матрице результата находится результат сложения переданных двух матриц. Вернуть из метода матрицы ответа.

4.2.2 Вычитание из одной разреженной матрицы другую

Данный метод класса перегружает операцию вычитания из одной разреженной матрицы другую. Алгоритм вычитания двух разреженных матриц следующий:

1. Создать матрицу результата.
2. Первый цикл проходит по строкам матриц.
3. Второй цикл идет до тех пор, пока в массиве значений обеих матриц не закончится рассматриваемая строка. В этом цикле присутствуют следующие проверки:
    a. Если элементы матриц находятся в одном столбце, то вычесть из элемента левой матрицы элемент правой матрицы и результат добавить в массив значений матрицы результата, а также добавить номер столбца в массив столбцов. И увеличить счетчики массива элементов обеих матриц.
    b. Если столбец, в котором находится элемент левой матрицы, меньше столбца, в котором находиться элемент правой матрицы, то добавить в матрицу результата элемент левой матрицы, а также добавить в массив столбцов номер данного столбца. И увеличить счетчик массива элемента левой матрицы.
    c. Если столбец, в котором находится элемент правой матрицы, меньше столбца, в котором находиться элемент левой матрицы, то добавить в матрицу результата элемент правой матрицы, умноженный на (-1), а также добавить в массив столбцов номер данного столбца. И увеличить счетчик массива элемента правой матрицы.
4. После завершения второго цикла добавить в массив строк матрицы результата размер массива столбцов этой матрицы.
5. После завершения первого цикла в матрице результата находится результат сложения переданных двух матриц. Вернуть из метода матрицу ответа.

4.2.3 Умножение двух разреженных матриц

Данный метод класса перегружает операцию умножения двух разреженных матриц. Алгоритм умножения двух разреженных матриц следующий:

1. Создать матрицу результата.
2. Первый цикл проходит по строкам матриц.
3. Второй цикл идет до тех пор, пока в массиве значений левой матрицы не закончится рассматриваемая строка. В этом цикле выполняются следующие действия:
    a. Пройти по всем элементам правой матрицы и умножить их на элементы, рассматриваемой на данном шаге строки. Прибавить результат умножения к элементу массива промежуточных значений, номер элемента определяем по значению столбца правой матрицы, в котором находится, рассматриваемый элемент.
    b. После умножения всех значений правой матрицы сохранить значение массива в вектор значений матрицы результата. А так же обновить вектор столбцов, для каждого элемента.
    c. Добавить размер вектора столбцов в вектор строк матрицы результата.
4. После завершения первого цикла в матрице результата находится результат умножения переданных двух матриц. Вернуть из метода матрицу ответа.

4.2.4 Умножение матрицы на число

Данный метод класса перегружает операцию умножения разреженной матрицы на число слева. Алгоритм умножения следующий:

1. Создать матрицу ответа

2. Пробежать по всему массиву элементов матрицы и умножить каждый элемент на число, новый элемент добавить в массив элементов матрицы ответа.

3. Скопировать из данной матрицы в матрицу ответа массивы колонок и строк.

4. Вернуть из метода матрицу ответа.

4.2.5 Операция унарного минуса

Данный метод класса перегружает операцию унарного минуса разреженной матрицы. Алгоритм следующий:

1. Создать матрицу ответа

2. Пробежать по всему массиву элементов матрицы и умножить каждый элемент на (-1), новый элемент добавить в массив элементов матрицы ответа.

3. Скопировать из данной матрицы в матрицу ответа массивы колонок и строк.

4. Вернуть из метода матрицу ответа.

4.2.6 Транспонирование матрицы

Данный метод класса транспонирует разреженную матрицу. Алгоритм следующий:

1. Создать матрицу ответа

2. Пробежать по всему массиву элементов матрицы и сохранить элементы каждого столбца отдельно. При этом сохранив, с какой строки начинается каждый столбец.

3. Собрать массив значений матрицы ответа как будто сохраненные столбцы это строки, причем положение элементов в вектор строк. А номера строк сохранить в массив столбцов.

4. Вернуть из метода матрицу ответа.

4.2.7 Получение следа матрицы

Данный метод класса возвращает след разреженной матрицы. Алгоритм следующий:

1. Сложить все диагональные элементы матрицы.
2. Вернуть из метода полученную сумму.

4.2.8 Получение размера матрицы

Данный метод класса возвращает размер разреженной матрицы. Размером матрицы является размер вектора столбцов минус один

4.2.9 Восстановление разреженной матрицы в нормальный вид

Данный метод класса возвращает вместо разреженной матрицы двумерный массив, в котором присутствуют все элементы матрицы, включая нулевые.

4.2.10 Получение нормы матрицы

Данный метод класса возвращает евклидову норму матрицы.

1. Возводится в квадрат каждый элемент матрицы.
2. Суммируются квадраты элементов.
3. Извлекается квадратный корень из суммы квадратов всех элементов.

4.3 Класс Storge

Данный класс используется для хранения и работы с большими или слишком маленькими числами, так как в результате выполнения алгоритма вычисления матричной экспоненты числа могут выйти за диапазон хранения стандартных типов. Число хранится в виде массива. Отдельно хранится знак числа, а так же положение плавающей точки[4,10].

4.4 Класс ComplexFunction

В данном классе хранятся сложные функции над разреженными матрицами. В классе реализованы следующие методы:

1) *ComplexFunction()* – метод создает элемент класса.
2) *SparseMatrix cellToMat(vector<vector<SparseMatrix>> matrix)* – метод переводит блочную матрицу в разреженную.

3) *SparseMatrix kroneck(SparseMatrix left, SparseMatrix right)* – метод реализующий произведение Кронекера для разреженных матриц.

4) *int gauss(vector < vector<double> > a, vector<double> & ans)* – метод реализующий Метод Гаусса для СЛАУ.

5) double accuracyExp(double normM, double eps) – метод вычисляющий точность вычисления экспоненты.

4.4.1 Перевод блочной матрицы в разреженную матрицу

Данный метод класса переводить разреженные матрицы из блочного представления в разреженную матрицу. Алгоритм действий следующий:

1. Создать матрицу ответа.
2. Цикл проходит по строкам блочной матрицы.
    a. Последовательно в массив матрица ответа добавить массивы значений матриц находящихся в строке блочной матрицы. При добавлении каждого элемента значений добавить в элемент столбцов его номер его столбца + номер столбца блочной матрицы.
    b. По завершению строки добавить в массив строк матрицы ответа размер массива столбцов матрицы ответа.
3. По завершению цикла вернуть матрицу ответа.

4.4.2 Произведение Кронекера

Данный метод класса реализует произведение Кронекера для разреженных матриц. Алгоритм действий следующий:

1. Создать матрицу ответа.
2. Цикл проходит по массиву значений правой матрицы.
    a. Каждое значение правой матрицы умножается на левую матрицу, и результат сохраняется в блочную матрицу, по координатам номер столбца в правом массиве и номер элемента в этом столбце.

3. По завершению цикла перевести получившуюся блочную матрицу в разреженную. Вернуть полученную разреженную матрицу

4.4.3 Метод Гаусса

Данный метод класса реализует алгоритм Гаусса-Жордана для системы алгебраических линейных уравнений [10]. Алгоритм заключается в последовательном исключении переменных из уравнения до тех пор, пока в уравнении не останется только по одной переменной. Если $n = m$ (как в нашем случае) то алгоритм Гаусса-Жордана стремится привести матрицу системы к единичной матрице — ведь после того как матрица стала единичной, решение системы очевидно — решение единственно и задаётся получившимися коэффициентами b_i.

4.4.4 Точность экспоненты

В данном методе вычисляется точность, с которой должен производиться алгоритм вычисления экспоненты. На вход метода передается норма матрицы и точность, с которой должна быть вычислена экспонента. На выходе получаем такой ε, при подстановке которого в алгоритм вычисления экспоненты, точность самой экспоненты будет не меньше переданного значения.

4.5 Класс ClassExp.

В данном классе вычисляется матричная экспонента. Для этого при создании элемента класса рассчитываются все n степеней матрицы, от которой, в последствие, нужно посчитать экспоненту. Так же при создании элемента передается и точность, с которой должен вычислять алгоритм. На вход функции расчета самой экспоненты передается t. Математическая формула для вычисления экспоненты описана в пункте 3.2. Алгоритм вычисления следующий:

1. Сначала нужно инициировать результат значением нулевой матрицы.

2. Далее выполнить для $k = \overline{0, n-1}$ следующую последовательность операций

a. Вычислить сумму $\sum_{m=0}^{\infty} r_{m,k} t^{m+n}$. В качестве критерия прекращения суммирования использовать условие $\left| r_{m,k} t^{m+n} \right| < \varepsilon$, где ε – задано для каждой матрицы, и вычисляется при помощи функции *accuracyExp()* класса ComplexFunction.

b. Используя значения, полученные ранее, определить произведение $A^k \left[\frac{t^k}{k!} + \sum_{m=0}^{\infty} r_{m,k} t^{m+n} \right]$ и прибавить его к текущему значению результата.

При вычислении матричной экспоненты с помощью данного алгоритма используется рекуррентное соотношение (3.4). При больших значениях m и k большинство значений q будут рассчитываться повторно много раз. Поскольку $q(m, k)$ является чистой функцией (зависит только от входных аргументов), то будет разумно применить стратегию сохранения результата.

5 Интерфейс программы

В данной программе предусмотрен пользовательский интерфейс (Рис. 2). Он предоставляет возможность рассчитать матрицу Ляпунова для системы, описанной в файле. Так же выдает графики компонент матрицы. И позволяет вывести на график или экран матрицу Ляпунова для заданного τ.

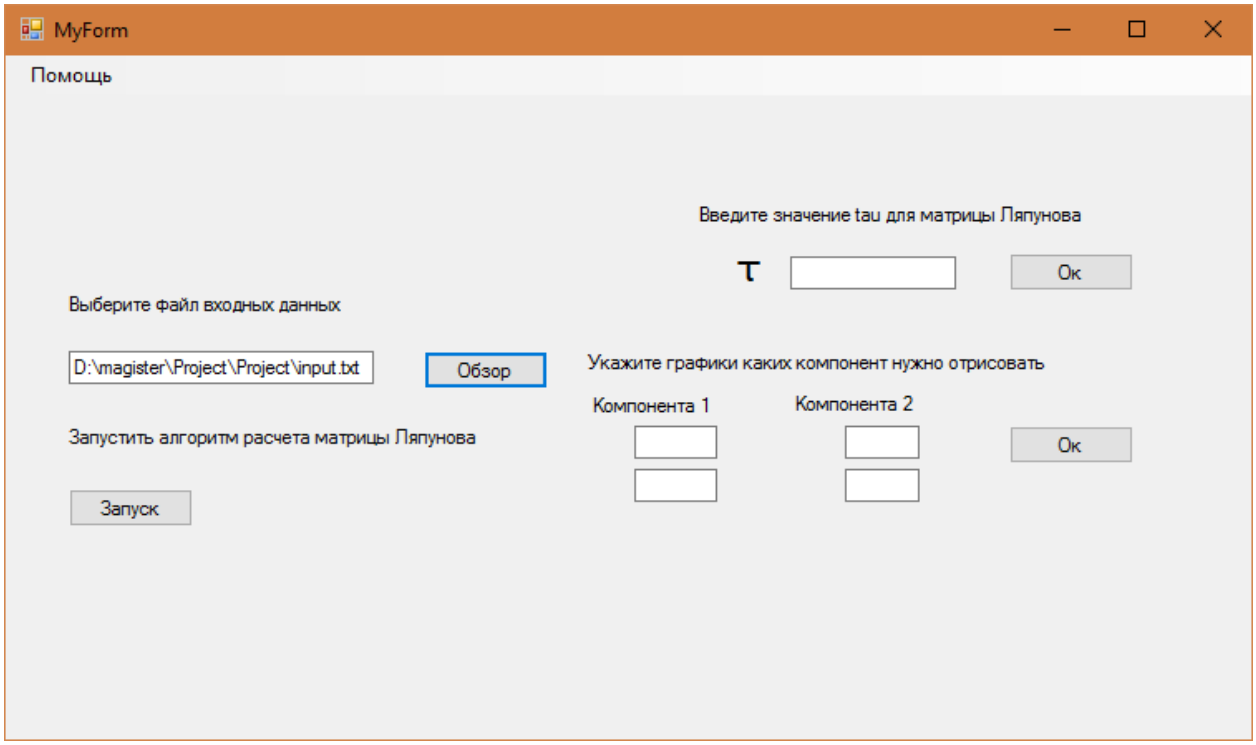


Рис. 2 Пример интерфейса программы

6 Пример работы программы

Пример 1

Рассмотрим систему вида

$$\frac{dx(t)}{dt} = A_0 x(t) + A_1 x(t - h_1) + A_2 x(t - h_2),$$

где $h_1 = {}^1\!/_3, h_2 = {}^1\!/_2$, $\text{НОК}(3,2) = 6$

$$h = \frac{1}{\text{НОК}(3,2)} = \frac{1}{6} = 0.167$$

$$h_1 = {}^1\!/_3 = 2{\times}0.167,$$
$$h_2 = {}^1\!/_2 = 3{\times}0.167$$

Размерность матриц $n = 3$, запаздываний $m = 3$.

$$A_0 = \begin{pmatrix} 0{,}7 & 4 & 0{,}1 \\ 0{,}1 & 1 & 0{,}5 \\ 0{,}9 & 5 & 1 \end{pmatrix}, \quad A_1 = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix},$$

$$A_2 = \begin{pmatrix} 0{,}2 & 0{,}5 & 0{,}1 \\ 6 & 1 & 0{,}5 \\ 9 & 2 & 0{,}3 \end{pmatrix}, A_2 = \begin{pmatrix} 0{,}4 & 0{,}7 & 0{,}1 \\ 6 & 5 & 0{,}1 \\ 9 & 2 & 0{,}3 \end{pmatrix},$$

$$W = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}.$$

На рисунках рис. 3, рис.4 представлены графики компонент вычисленной матрицы Ляпунова.

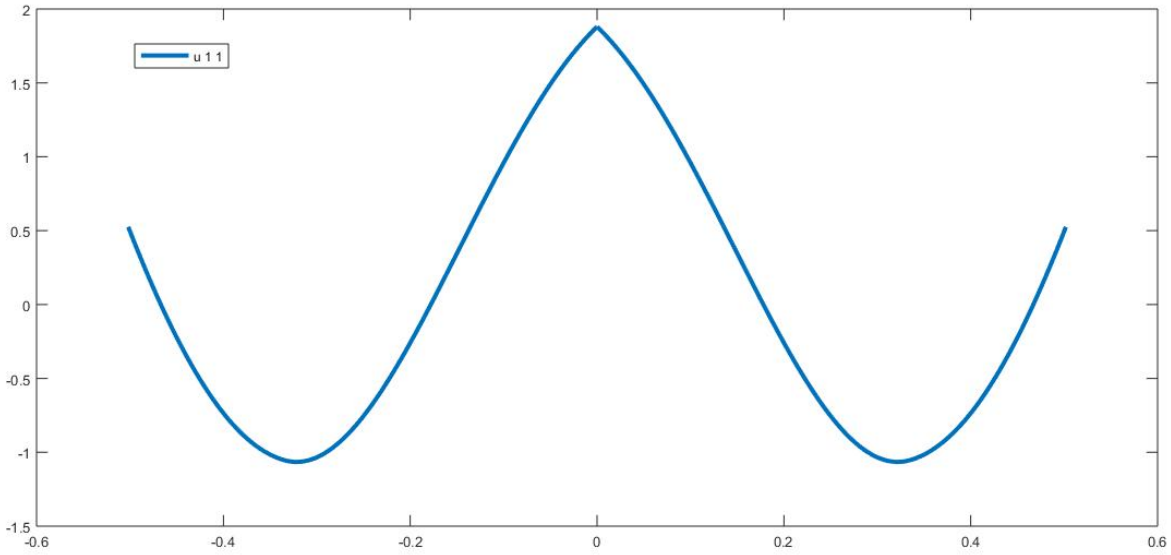


Рис 3. График компоненты $u_{1\,1}$. Показывает выполнение свойства симметрии матрицы U

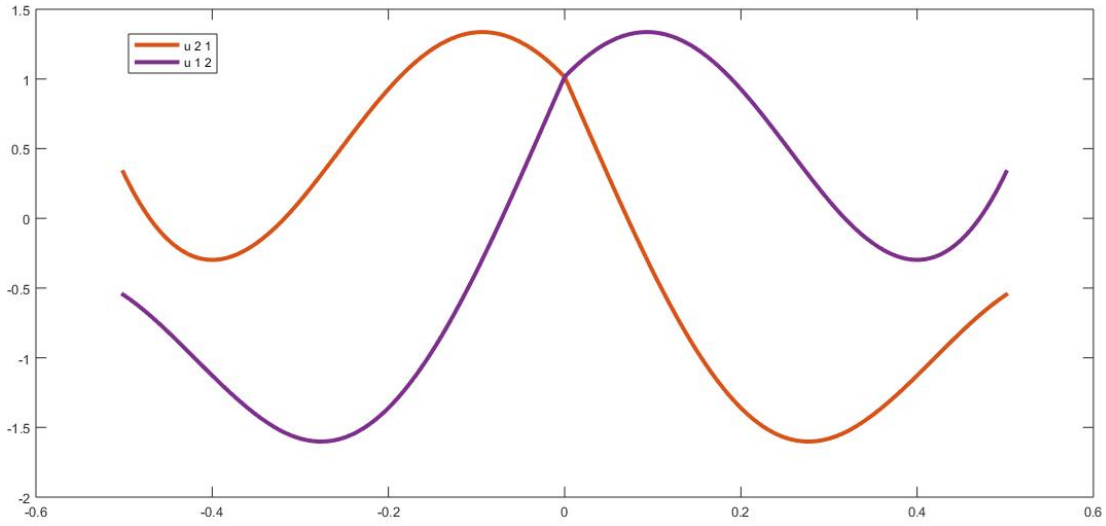


Рис 4. График компоненты $u_{2\,1}$ и $u_{1\,2}$.. Показывает выполнение свойства симметрии матрицы U

Пример 2

Рассмотрим систему вида

$$\frac{dx(t)}{dt} = A_0x(t) + A_1x(t-h_1) + A_2x(t-h_2),$$

где $h_1 = {}^1/_4, h_2 = {}^1/_7$, $\text{НОК}(4,7) = 28$

$$h = \frac{1}{\text{НОК}(4,7)} = \frac{1}{28} = 0.036$$

$$h_1 = {}^{1}/_{4} = 7{\times}0.036,$$

$$h_2 = {}^{1}/_{7} = 4{\times}0.036$$

Размерность матриц $n = 3$, запаздываний $m = 7$.

$$A_0 = \begin{pmatrix} 0{,}7 & 4 & 0{,}1 \\ 0{,}1 & 1 & 0{,}5 \\ 0{,}9 & 5 & 1 \end{pmatrix}, \quad A_1 = A_2 = A_3 = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix},$$

$$A_4 = \begin{pmatrix} 0{,}2 & 0{,}5 & 0{,}1 \\ 6 & 1 & 0{,}5 \\ 9 & 2 & 0{,}3 \end{pmatrix}, \quad A_5 = A_6 = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix},$$

$$A_7 = \begin{pmatrix} 0{,}4 & 0{,}7 & 0{,}1 \\ 6 & 5 & 0{,}1 \\ 9 & 2 & 0{,}3 \end{pmatrix}, W = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}.$$

На рисунках рис. 5, рис.6 представлены графики компонент вычисленной матрицы Ляпунова.

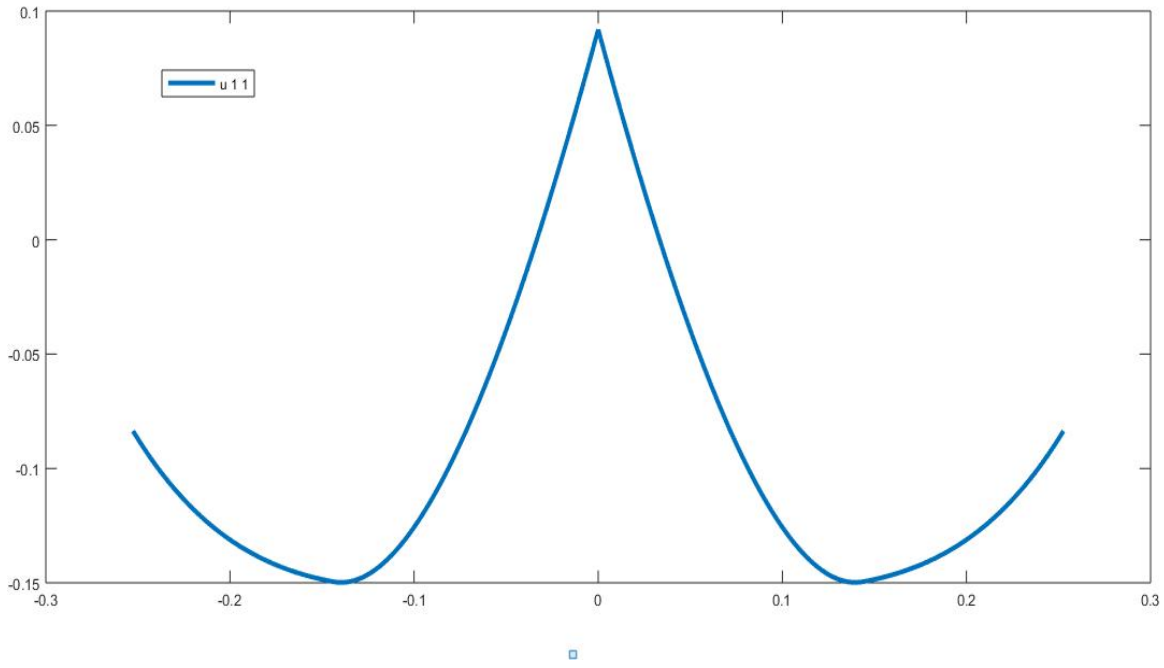


Рис 5. График компоненты $u_{1\,1}$. Показывает выполнение свойства симметрии матрицы U

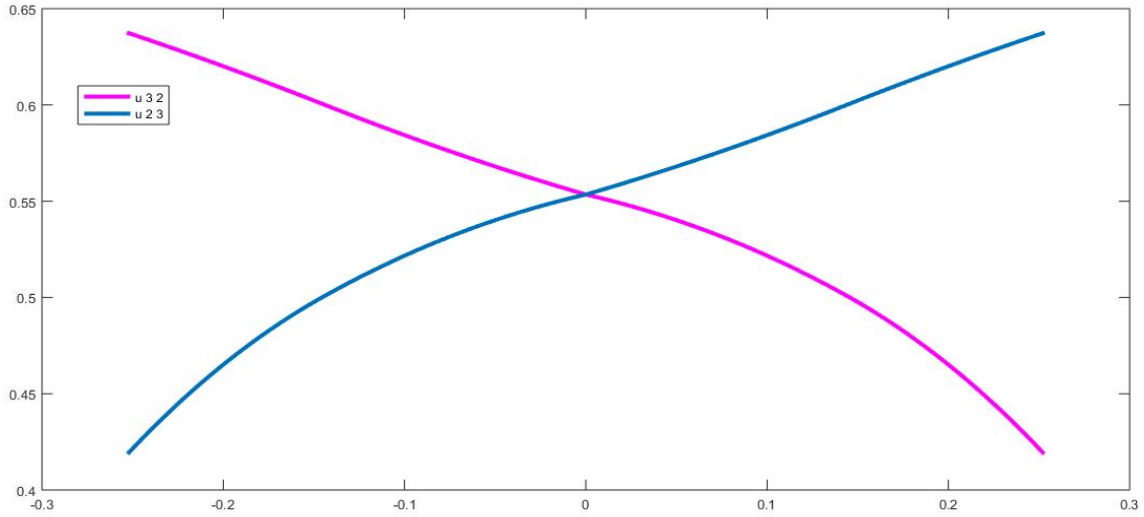


Рис 6. График компоненты u_{32} и u_{23}. Показывает выполнение свойства симметрии матрицы U

Пример 3

n= 4, m=4, h = 1

$$A_0 = \begin{pmatrix} 0.7 & 4 & 0.1 & 0.9 \\ 0.1 & 1 & 0.5 & 8 \\ 0.9 & 5 & 1 & 2 \\ 0.4 & 0.7 & 9 & 4 \end{pmatrix}, \quad A_1 = \begin{pmatrix} 0.2 & 0.5 & 0.2 & 0.6 \\ 6 & 1 & 4 & 0.4 \\ 4 & 5 & 0.9 & 0.8 \\ 3 & 5 & 0.6 & 0.5 \end{pmatrix},$$

$$A_2 = \begin{pmatrix} 0.2 & 0.5 & 0.1 & 0.2 \\ 6 & 1 & 0.5 & 0.7 \\ 1 & 2 & 0.6 & 0.4 \\ 2 & 7 & 0.9 & 8 \end{pmatrix}, \quad A_3 = \begin{pmatrix} 0.4 & 0.7 & 0.1 & 3 \\ 6 & 5 & 0.1 & 0.3 \\ 9 & 2 & 0.3 & 0.7 \\ 0 & 3 & 0.4 & 1 \end{pmatrix},$$

$$A_4 = \begin{pmatrix} 0.3 & 0.7 & 0.2 & 7 \\ 6 & 8 & 0.7 & 0.5 \\ 9 & 6 & 0.4 & 0.9 \\ 1 & 3 & 0.7 & 4 \end{pmatrix}, \quad W = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}.$$

На рисунках рис. 7, рис.8 представлены графики компонент вычисленной матрицы Ляпунова.

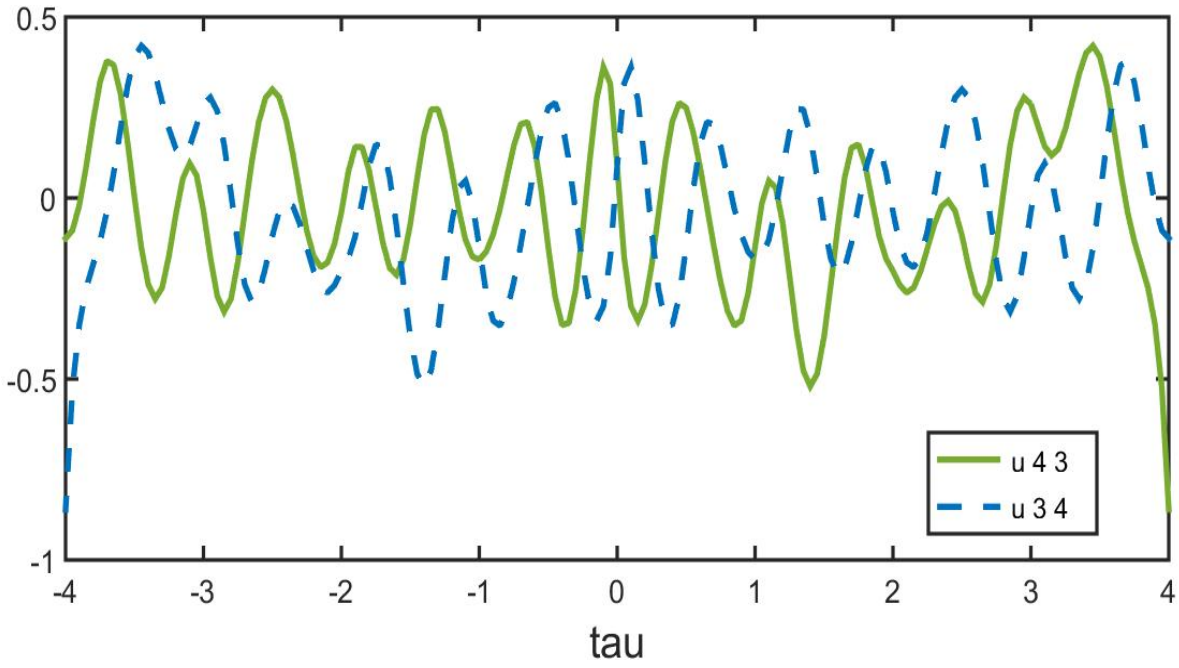


Рис 7. График компонент u_{34} и u_{43}. Показывает выполнение свойства симметрии матрицы U

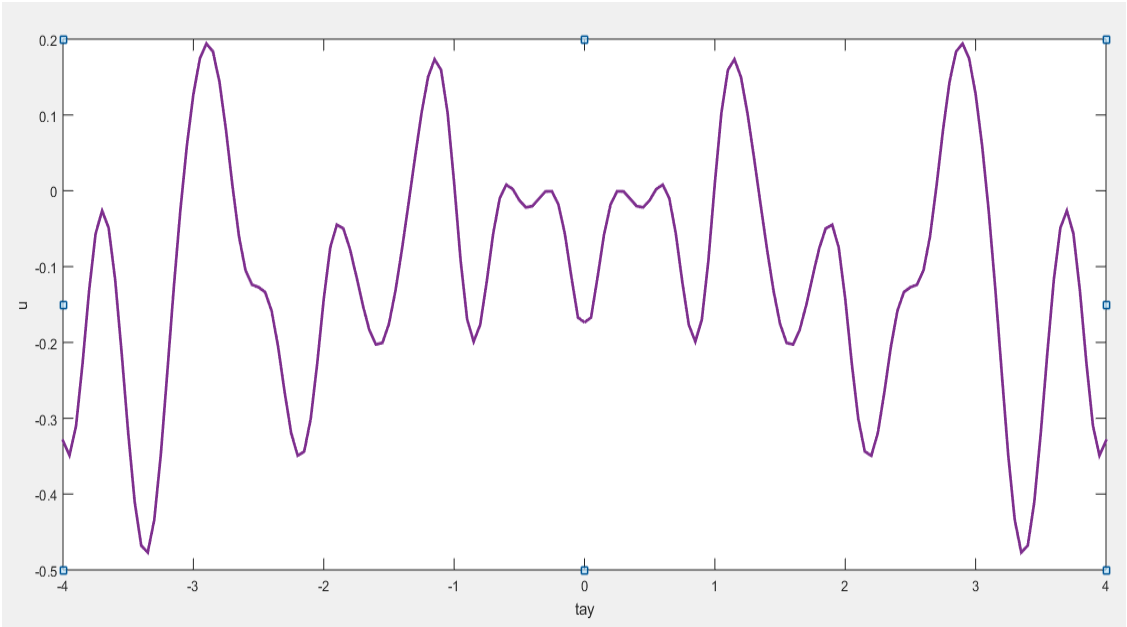


Рис 8. График компоненты u_{33}. Показывает выполнение свойства симметрии матрицы U

7 Характеристики работы программы

Тестирование программы проводилось на компьютере со следующей конфигурацией:

- Процессор Intel(R) Core(TM) i5-3230M CPU 2.60GHz
- Оперативная память - 4 Gb

Характеристики программы по времени и объему памяти указаны в таблицах.

Таблица 1 Характеристика времени выполнения программы

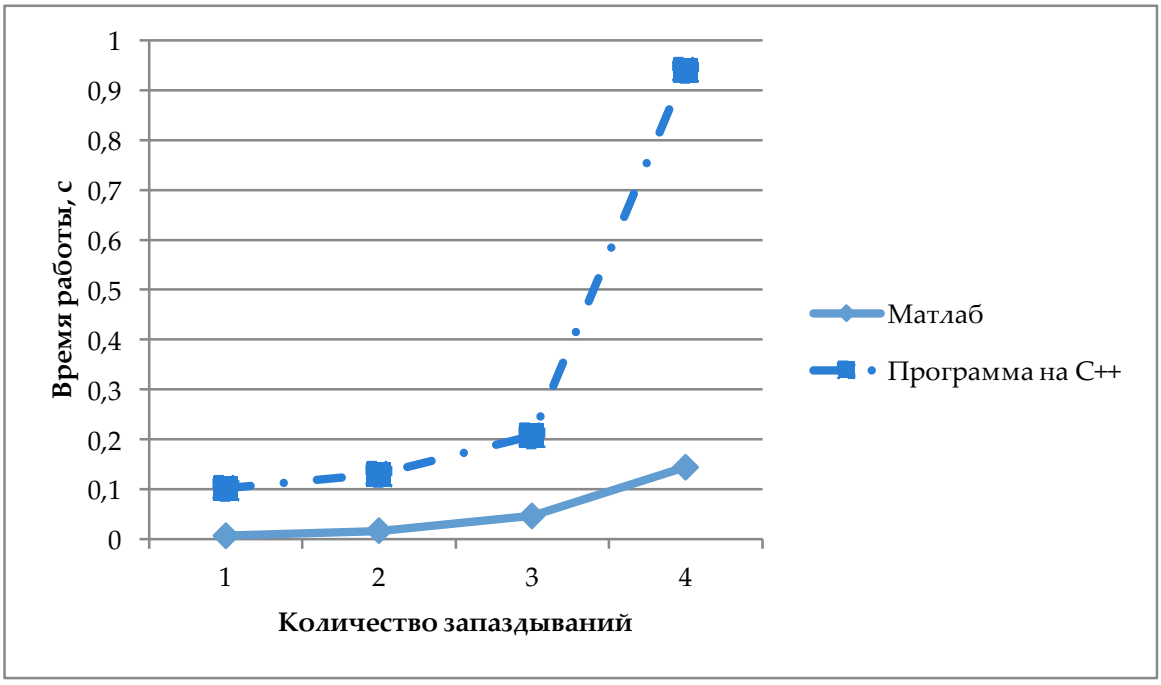
Время работы программы, с	Размерность матриц	Количество запаздываний
0.102	2	1
0.13	2	2
0.207	3	3
0.939	4	4
283	5	5

Таблица 2 Характеристика затрат памяти программы

Затраты памяти, МБ	Размерность матриц	Количество запаздываний
1.1	2	1
1.5	2	2
2.1	3	3
2.8	4	4

8 Сравнительная характеристика с программой, написанной на Matlab

В данном разделе представлена сравнительная характеристика времени работы алгоритма вычисления в программе Matlab и при использовании описанной выше программы на C++. Как можно увидеть из графика программа на C++ работает медленнее при вычислении, но сравнение возможно провести только для размерности 4×4 и 4 запаздываний. Так как дальше алгоритм, написанный в Matlab, не справляется с промежуточными вычислениями. Именно по этой причине было принято решение писать программу в языке программирование C++.



9 Заключения

В рамках работы требовалось построить матрицу Ляпунова для системы (1). В результате был предложен алгоритм нахождения матриц Ляпунова для систем со многими запаздываниями, основанный на теории разреженных матриц. Проведен запуск алгоритма для максимально числа запаздываний равного 30, при размерности матрицы 3×3. Характеристики выполненных запусков и примеры работы приведены выше. Результаты работы были доложены на научной конференции “Процессы управления и устойчивость”. В дальнейшем планируется ввести распараллеливание потоков при вычислении степени матрицы и экспоненты от матрицы.

Список литературы

1. Kharitonov V.L. Time-delay systems. Lyapunov functionals and matrices. Birkhauser, Basel, 2013. 311 p.
2. Безгин С.В., Пчелинцев А.Н. Организация матричных и символьных вычислений для исследования поведения решений обыкновенных дифференциальных уравнений // Системы управления и информационные технологии, 2012. Т. 47, №1. — С. 4-7.
3. Беллман Р., Кук К. Дифференциально-разностные уравнения. М., 1967, 548 с.
4. Длинная арифметика. http://acm.mipt.ru/twiki/bin/view/Algorithms/LongArithmeticsCPP .
5. Документация matlab. http://www.mathworks.com/help/matlab/.
6. Документация по C++. http://ru.cppreference.com/w/cpp/language .
7. Калинтин Н.Н. Численные методы. СПб., 2011, 586 с
8. Самарский А.А., Гулин А.В. Численные методы. М., 1989, 430 с.
9. Писсанецки С. Технология разреженных матриц. М.: Мир, 1988, 416с.
10. Сайт некоторых математических алгоритмов. http://e-maxx.ru/algo.

Приложение

Пакет CalucationLypunov

```
#include "CalculationLypunova.h"
#include "SparseMatrix.h"
#include "ComplexFunction.h"
#include "ClassExp.h"



using namespace std;


int n, m;  // Размер масива У,Х
vector<vector<double>> X; // масив для хранение х-ов
vector<vector<double>> Y; // масив для хранения у-ов
ClassExp* expP;
SparseMatrix ans_mat;


vector<double> getX(int i1, int j1) {
	return Y[(j1-1)*n+(i1-1)];
}

vector<double> getY(int i1, int j1) {
	return X[(j1 - 1)*n + (i1-1)];
}

vector<double> getU(double tau){
	int i = tau / m;
	double tay = tau - i*m;
	i++;
	SparseMatrix mat = expP->exp(tay)*ans_mat;
	int size = mat.values.size();
	vector<double> u;
	for (int q = n*n*(i - 1); q < n*n*i; q++) {
		u.push_back(mat.values[q]);
	}
	return u;
}

void print(SparseMatrix& l) {
	for (int i = 0; i < l.size(); i++) {
		int k = l.pointer[i];
		int h = l.size();
		for (int j = 0; j < l.size(); j++) {
			if (k < l.pointer[i + 1] && !l.cols.empty() && l.cols[k] == j) {
				cout << l.values[k] << " ";
				k++;
			}
			else {
				cout << 0 << " ";
			}
		}
		cout << endl;
	}
	cout << endl;
	cout << endl;
}

vector<SparseMatrix> matrix;
```

```
int matrixLypunova(string input) {
	const char* ch = input.c_str();
	freopen(input.c_str(), "r", stdin);
	freopen("output.txt", "w", stdout);


	double h;

	cin >> n >> m >> h;
	SparseMatrix Zeros(n, 0);
	matrix.push_back(Zeros);
	for (int k = 0; k < m + 2; k++) {
		SparseMatrix a;
		a.pointer.push_back(0);
		for (int i = 0; i < n; i++) {
			for (int j = 0; j < n; j++) {
				double ak;
				cin >> ak;
				if (ak != 0) {
					a.values.push_back(ak);
					a.cols.push_back(j);
				}
			}
			a.pointer.push_back(a.cols.size());
		}
		matrix.push_back(a);
	}
	for (int i = 0; i < m + 2; i++) {
		//	print(matrix[i]);
	}

	ComplexFunction func;
	vector<vector<SparseMatrix>> k(3);
	unsigned int start_time = clock();
	vector<vector<SparseMatrix>> L_cell(2 * m + 1);
	vector<vector<SparseMatrix>> M_cell(2 * m + 1);
	vector<vector<SparseMatrix>> N_cell(2 * m + 1);
	SparseMatrix Zeros2(n*n, 0);
	for (int i = 1; i <= 2 * m; i++) {
		L_cell[i].assign(2 * m + 1, Zeros2);
		M_cell[i].assign(2 * m + 1, Zeros2);
		N_cell[i].assign(2 * m + 1, Zeros2);
	}
	SparseMatrix E(n, 1);

	for (int i = 1; i <= m - 1; i++) {
		for (int j = 0; j <= m; j++) {
			int l = i - j;
			if (i - j <= 0) {
				l = (i - j) + 2 * m;
			}
			L_cell[i][l] = func.kroneck(E, matrix[j + 1]);
		}
	}
	for (int j = 0; j <= m; j++) {
		int k = -j;
		if (-j <= 0) {
			k = -j + 2 * m;
		}
		L_cell[2 * m][k] = func.kroneck(E, matrix[j + 1]);
	}
	for (int i = 1; i <= m; i++) {
		for (int j = 0; j <= m; j++) {
```

```
                int c = -i + j;
                if (c <= 0) {
                        c = (-i + j) + 2 * m;
                }
                L_cell[-i + 2 * m][c] = func.kroneck(-matrix[j + 1].trans(), E);
                //      SparseMatrix re = -matrix[j + 1].trans();
                //      print(re);
        }
    }

    for (int f = -m + 1; f <= m - 1; f++) {
        int l = f;
        if (f <= 0) {
                l = f + 2 * m;
        }
        M_cell[l][l] = func.kroneck(E, E);
        int ind = f - 1;
        if (ind <= 0) {
                ind = ind + 2 * m;
        }
        N_cell[l][ind] = func.kroneck(-E, E);
    }
    int    ind = m - 1;
    if (ind <= 0) {
        ind = ind + 2 * m;
    }
    N_cell[-m + 2 * m][ind] = func.kroneck(E, matrix[m + 1]);
    M_cell[-m + 2 * m][-m + 2 * m] = func.kroneck(matrix[m + 1].trans(), E);
    for (int k = 0; k <= m - 1; k++) {
        int l = k;
        if (k == 0) {
                l = k + 2 * m;
        }
        M_cell[-m + 2 * m][l] = func.kroneck(matrix[k + 1].trans(), E);
        l = -k;
        if (l <= 0) {
                l = -k + 2 * m;
        }
        M_cell[-m + 2 * m][l] = func.kroneck(E, matrix[k + 1]);
    }
    M_cell[-m + 2 * m][2 * m] = func.kroneck(matrix[1].trans(), E) + func.kroneck(E,
matrix[1]);

    SparseMatrix L = func.cellToMat(L_cell);
    SparseMatrix M = func.cellToMat(M_cell);
    SparseMatrix N = func.cellToMat(N_cell);
    //print(L);
    double eps = func.accuracyExp(L.norm(), 1e-7);
    ClassExp pe(L, eps);
    vector <double> w = matrix[m + 2].values;
    vector <double> vec((m - 1)*n*n, 0);
    vector <double> vec2(m*n*n, 0);
    int a = w.size();
    vec.reserve(vec.size() + w.size());
    vec.insert(vec.end(), w.begin(), w.end());
    vec.reserve(vec.size() + vec2.size());
    vec.insert(vec.end(), vec2.begin(), vec2.end());
    SparseMatrix det = (M + (N*pe.exp(h)));
    //print(det);
    vector<vector<double>> mat = det.sparseToMat();
    int b = mat.size();
    int c = vec.size();
    for (int i = 0; i < mat.size(); i++) {
        mat[i].push_back(-vec[i]);
```

```
}
vector <double> ans;
int res = func.gauss(mat, ans);
if (res != 0) {
	for (int i = 0; i < ans.size(); i++) {
		cout << ans[i] << " ";
	}
	cout << endl;
	cout << endl;
}

unsigned int end_time = clock();
unsigned int search_time = end_time - start_time;
cout << "search_time = " << search_time / 1000.0 << endl;

ans_mat.pointer.push_back(0);
for (int i = 0; i < ans.size(); i++) {
	ans_mat.values.push_back(ans[i]);
	ans_mat.cols.push_back(0);
	ans_mat.pointer.push_back(ans_mat.cols.size());
}
//print(ans_mat);
X.assign(n*n, vector<double>());
Y.assign(n*n, vector<double>());
int z = 0;
for (int i = 1; i <= 2 * m; i++) {
	int k = i;
	if (i > m - 1) {
		k = i - 2 * m;
	}
	SparseMatrix mat;
	double step = 0.01;
	double tay = 0;

	for (double tay = 0; tay <= h; tay += step) {
		mat = pe.exp(tay)*ans_mat;
	//	print(mat);
		int size = mat.values.size();
		vector<double> u;
		for (int q = n*n*(i - 1); q < n*n*i; q++) {
			u.push_back(mat.values[q]);
		}
		z = z + 1;
		for (int j = 0; j < n*n; j++) {
			X[j].push_back(u[j]);
		//	cout << u[j] << " ";
			Y[j].push_back(tay + k*h);
		}
	//	cout << endl;
	}

}

for (int l = 0; l < n*n; l++) {
	for (int i = 0; i < Y[l].size(); i++) {
		for (int j = i; j < Y[l].size(); j++) {
			if (Y[l][i] < Y[l][j]) {
				swap(Y[l][i], Y[l][j]);
				swap(X[l][i], X[l][j]);
			}
		}
	}
}
expP = &pe;
```

```
	return 1;
}
```

Класс SparseMatrix

```
#pragma once
#include <vector>
using namespace std;
class SparseMatrix
{
public:
	vector<int> cols;
	vector<double> values;
	vector<int> pointer;
	SparseMatrix() {};
	SparseMatrix(int size, bool l);

	friend const SparseMatrix operator+(const SparseMatrix& left, const SparseMatrix&
right);
	friend const SparseMatrix operator-(const SparseMatrix& left, const SparseMatrix&
right);
	friend const SparseMatrix operator*(const SparseMatrix& left, const SparseMatrix&
right);
	friend const SparseMatrix operator*(const double& left, const SparseMatrix&
right);
	friend const SparseMatrix operator-(const SparseMatrix& i);
	SparseMatrix trans();
	double matrixTrace();
	int size();
	vector<vector<double>> sparseToMat();
	double norm();
};
```

```
#include "SparseMatrix.h"
#include <algorithm>


SparseMatrix::SparseMatrix(int size, bool l)
{
	int n = size;
	for (int i = 0; i < size; i++) {
		if (l) {
			values.push_back(1);
			cols.push_back(i);
			pointer.push_back(i);
		}
		else {
			pointer.push_back(0);
			n = 0;
		}
	}
	pointer.push_back(n);
}


SparseMatrix SparseMatrix::trans()
{
	SparseMatrix ans;
	vector<vector<double>> val(pointer.size() - 1);
	vector<vector<int>> point(pointer.size() - 1);
	int k = 1;
	for (int i = 0; i < values.size(); i++) {
		if (i == pointer[k]) {
```

```
                        k++;
                }
                int h = k - 1;
                point[cols[i]].push_back(h);
                val[cols[i]].push_back(values[i]);
        }
        ans.pointer.push_back(0);
        k = 0;
        for (int i = 0; i < val.size(); i++) {
                for (int j = 0; j < val[i].size(); j++) {
                        ans.values.push_back(val[i][j]);
                        ans.cols.push_back(point[i][j]);
                }
                ans.pointer.push_back(ans.cols.size());
        }
        return ans;
}

double SparseMatrix::matrixTrace()
{
        double ans = 0;
        for (int i = 0; i < pointer.size() - 1; i++) {
                int k = pointer[i];
                while (k < pointer[i + 1]) {
                        if (cols[k] == i) {
                                ans += values[k];
                                break;
                        }
                        k++;
                }
        }
        return ans;
}

int SparseMatrix::size()
{
        return pointer.size() - 1;
}

vector<vector<double>> SparseMatrix::sparseToMat()
{
        int h = size();
        vector<vector<double>> ans(h);
        for (int i = 0; i < h; i++) {
                int k = pointer[i];
                for (int j = 0; j < h; j++) {
                        if (k < pointer[i + 1] && !cols.empty() && cols[k] == j) {
                                ans[i].push_back(values[k]);
                                k++;
                        }
                        else {
                                ans[i].push_back(0);
                        }
                }
        }
        return ans;
}

double SparseMatrix::norm()
{
        double ans = 0;
        for (int i = 0; i < values.size(); i++) {
                ans += values[i] * values[i];
        }
```

```
	ans = sqrt(ans);
	return ans;
}

SparseMatrix const operator+(const SparseMatrix& left, const SparseMatrix& right) {
	SparseMatrix ans;
	ans.pointer.push_back(0);
	for (int i = 0; i <left.pointer.size() - 1; i++) {
		int k = left.pointer[i];
		int j = right.pointer[i];
		while (k < left.pointer[i + 1] || j < right.pointer[i + 1]) {
			if (j < right.pointer[i + 1] && k < left.pointer[i + 1]) {
				if (left.cols[k] == right.cols[j]) {
					ans.values.push_back(left.values[k] + right.values[j]);
					ans.cols.push_back(right.cols[j]);
					j++;
					k++;
				}
				else if (left.cols[k] > right.cols[j]) {
					ans.values.push_back(right.values[j]);
					ans.cols.push_back(right.cols[j]);
					j++;
				}
				else if (left.cols[k] < right.cols[j]) {
					ans.values.push_back(left.values[k]);
					ans.cols.push_back(left.cols[k]);
					k++;
				}
			}
			else if (j < right.pointer[i + 1]) {
				ans.values.push_back(right.values[j]);
				ans.cols.push_back(right.cols[j]);
				j++;
			}
			else if (k < left.pointer[i + 1]) {
				ans.values.push_back(left.values[k]);
				ans.cols.push_back(left.cols[k]);
				k++;
			}
		}
		ans.pointer.push_back(ans.cols.size());
	}
	return ans;
}

SparseMatrix const operator-(const SparseMatrix& left, const SparseMatrix& right) {
	SparseMatrix ans;
	ans.pointer.push_back(0);
	for (int i = 0; i <left.pointer.size() - 1; i++) {
		int k = left.pointer[i];
		int j = right.pointer[i];
		while (k < left.pointer[i + 1] || j < right.pointer[i + 1]) {
			if (j < right.pointer[i + 1] && k < left.pointer[i + 1]) {
				if (left.cols[k] == right.cols[j]) {
					if (left.values[k] - right.values[j] != 0) {
						ans.values.push_back(left.values[k] -
right.values[j]);
						ans.cols.push_back(right.cols[j]);
					}
					j++;
					k++;
				}
				else if (left.cols[k] > right.cols[j]) {
					ans.values.push_back(-right.values[j]);
```

```
                                        ans.cols.push_back(right.cols[j]);
                                        j++;
                                }
                                else if (left.cols[k] < right.cols[j]) {
                                        ans.values.push_back(left.values[k]);
                                        ans.cols.push_back(left.cols[k]);
                                        k++;
                                }
                        }
                        else if (j < right.pointer[i + 1]) {
                                ans.values.push_back(-right.values[j]);
                                ans.cols.push_back(right.cols[j]);
                                j++;
                        }
                        else if (k < left.pointer[i + 1]) {
                                ans.values.push_back(left.values[k]);
                                ans.cols.push_back(left.cols[k]);
                                k++;
                        }
                }
                ans.pointer.push_back(ans.cols.size());
        }
        return ans;
}

SparseMatrix const operator*(const SparseMatrix& left, const SparseMatrix& right) {
        SparseMatrix ans;
        ans.pointer.push_back(0);
        for (int i = 0; i <left.pointer.size() - 1; i++) {
                int k = left.pointer[i];
                int n = left.pointer[i];
                vector<double> values;
                values.assign(right.pointer.size() - 1, 0);
                while (k < left.pointer[i + 1]) {
                        for (int l = 0; l < right.pointer.size() - 1; l++) {
                                if (l == left.cols[k] && k < left.pointer[i + 1]) {
                                        for (int m = right.pointer[l]; m < right.pointer[l +
1]; m++) {
                                                int c = right.cols[m];
                                                values[right.cols[m]] += left.values[k] *
right.values[m];

                                        }
                                        k++;
                                }
                                if (k >= left.pointer[i + 1]) {
                                        break;
                                }
                        }
                }
                for (int j = 0; j < values.size(); j++) {
                        if (values[j] != 0) {
                                int h = values[j];
                                ans.values.push_back(values[j]);
                                ans.cols.push_back(j);
                        }
                }
                ans.pointer.push_back(ans.cols.size());
        }
        return ans;
}

SparseMatrix const operator*(const double& left, const  SparseMatrix& right) {
        SparseMatrix ans;
```

```
	ans.cols = right.cols;
	ans.pointer = right.pointer;
	for (vector<double>::const_iterator it = right.values.begin(); it !=
right.values.end(); ++it) {
		ans.values.push_back(*it * left);
	}
	return ans;
}

SparseMatrix const operator-(const SparseMatrix& i) {
	SparseMatrix ans;
	ans.cols = i.cols;
	ans.pointer = i.pointer;
	for (vector<double>::const_iterator it = i.values.begin(); it != i.values.end();
++it) {
		ans.values.push_back(*it *(-1));
	}
	return ans;
}
```

Класс ClassExp

```
#pragma once
#include "SparseMatrix.h"
#include <vector>
#include <map>
#include <cmath>
using namespace std;

class ClassExp
{

	SparseMatrix a;
	vector<SparseMatrix> powers;
	double eps;

	typedef map<pair<unsigned int, unsigned int>, double> pair_map;

	typedef map<unsigned int, double> atom_map;

	pair_map q_cache;
	atom_map p_cache;

	// получить следующую степень матри
	SparseMatrix nextPower(SparseMatrix prev,
		SparseMatrix a)
	{
		return a * prev;
	}
public:

	ClassExp(SparseMatrix m, double e)
		: a(m), powers(a.size() + 1), eps(e)
	{
		powers[0] = SparseMatrix(a.size(), 1);
		powers[1] = a;


		for (size_t i = 2; i < powers.size(); ++i)
		{
			powers[i] = nextPower(powers[i - 1], a);

		}
	}
```

```
	SparseMatrix exp(const double t);
private:
	double s(const unsigned int k);
	double p(const unsigned int k);
	double q(const unsigned int m, const unsigned int k);
	double factorial(const unsigned int n) const;
};

#include "ClassExp.h"
#include <iostream>
#include <fstream>
SparseMatrix ClassExp::exp(const double t)
{
	SparseMatrix result = SparseMatrix(a.size(), 0);

	int n = a.size();

	for (int k = 0; k < n; ++k)
	{
		//		cout << "k=" << k << endl;
		double temp = 0;
		double r = 0;
		int m = 0;

		double t_k = (pow(t, k) / factorial(k));

		do
		{
			//			cout << "m=" << m << endl;
			double r1 = q(m, n - k);
			double r2 = factorial(m + n);

			r = r1 / r2;

			temp += r*pow(t, n + m);
			m++;

		} while (fabs(r*pow(t, n + m)) > eps);
		if (result.cols.empty()) {
			result = (t_k + temp)*powers[k];
		}
		else {
			result = result + (t_k + temp)*powers[k];
		}
	}

	return result;
}


double ClassExp::s(const unsigned int k)
{
	double result = powers[k].matrixTrace();
	return result;
}

double ClassExp::p(const unsigned int k)
{

	atom_map::const_iterator i = p_cache.find(k);

	if (i != p_cache.end())
	{
		return i->second;
```

```
	}

	if (k == 1)
	{
		double result = a.matrixTrace();
		p_cache[k] = result;
		return result;
	}

	double result = s(k);

	for (unsigned int i = 1; i <= (k - 1); ++i)
	{
		double tmp = p(i) * s(k - i);
		result -= tmp;
	}

	result /= k;

	p_cache[k] = result;

	return result;
}

double ClassExp::q(const unsigned int m, const unsigned int k)
{

	pair_map::const_iterator i = q_cache.find(make_pair(m, k));

	if (i != q_cache.end())
	{
		return i->second;
	}

	if (k == a.size() + 1)
	{
		q_cache[make_pair(m, k)] = 0;
		return 0;
	}

	if (m == 0)
	{
		double result = p(k);

		q_cache[make_pair(m, k)] = result;
		return result;
	}

	double result = (p(k) * q(m - 1, 1)) + q(m - 1, k + 1);

	q_cache[make_pair(m, k)] = result;
	return result;
}

double ClassExp::factorial(const unsigned int n) const
{
	double result = 1;

	for (unsigned int i = 2; i <= n; ++i)
		result *= i;

	return result;
}
```