Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н. Э. Баумана (национальный исследовательский университет)»
(МГТУ им. Н. Э. Баумана)

ФАКУЛЬТЕТ Фундаментальные науки

КАФЕДРА Прикладная математика

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К ДОМАШНЕЙ РАБОТЕ

ПО КУРСУ:

"Методы численного решения задача линейной алгебры"

Студент

ФН2–31М

(Группа)

_____ Д. И. Богданов

(Подпись, дата) (И. О. Фамилия)

_____ А. С. Родин

(Подпись, дата) (И. О. Фамилия)

2024 г.

Содержание

1. Постановка задачи

Нужно сформировать матрицу размером 10×10 по следующему принципу. В качестве базовой матрицы , берется известная матрица, которая получается после дискретизации одномерного оператора Лапласа методом конечных разностей или методом конечных элементов. На равномерной сетке:

$$A_0 = \{a_{ij}\}_{i,j=\overline{1,n}}$$

где

$$a_{ij} = \begin{cases} 2, & i = j, \\ -1, & |i-j| = 1, \\ 0, & \text{else.} \end{cases}$$

Для данной матрицы известны аналитические формулы для собственных значений $(n = 10)$

$$\lambda_j^0 = 2(1 - \cos\frac{\pi j}{n+1}), \quad j = \overline{1,n}.$$

и компонент собственных векторов (вектора имеют 2-норму равную 1):

$$z_j^0(k) = \sqrt{\frac{2}{n+1}} \sin\frac{\pi jk}{n+1}, \quad k = \overline{1,n}.$$

Итоговая матрица получается по формулам:

$$A = A_0 + \delta A,$$

$$\delta A_{ij} = \begin{cases} \dfrac{c}{i+1}, & i \neq j, \\ 0, & i = j, \end{cases}$$

$$c = \frac{N_{var}}{N_{var}+1}\varepsilon.$$

где N_{var} — номер варианта (совпадает с номером студента в списке в журнале группы), ε - параметр, значение которого задается далее.

1.1. Задание 1

Взять матрицу для значения $\varepsilon = 0.1$, убрать последний столбец и сформировать из первых 9 столбцов матрицу $\hat{A}$ размера 10×9. Решить линейную задачу наименьших квадратов для вектора невязки

$$r = \hat{A}x - b,$$

где вектор b размерности 10×1 нужно получить по следующему алгоритму: выбрать вектор x_0, размерности 9×1 и для него вычислить $b = \hat{}_0$.

Для решения поставленной задачи использовать QR разложение: для вариантов с четным номером использовать соответствующий алгоритм, основанный на методе вращений Гивенса, для вариантов с нечетным номером - алгоритм, основанный на методе отражений Хаусхолдера.

После получения решения сделать оценку величины $||x - x_0||_2/||x_0||_2$.

1.2. Задание 2

Для матрицы найти все ее собственные значения ($\lambda_j, j = \overline{1,10}$) и собственные вектора ($z_j, j = \overline{1,10}$, с 2-нормой равной 1) с помощью неявного QR-алгоритма со сдвигом (с предварительным приведением матрицы к форме Хессенберга) для трех вариантов: $\varepsilon = 10^{-1}, 10^{-3}, 10^{-6}$.

По итогам расчетов нужно сделать сводную таблицу, в которой указать следующие величины: $\lambda_j - \lambda_j^0$ и $||z_j - z_j^0||_2$ для $j = \overline{1,10}$.

2. Решение линейной задачи наименьших квадратов в с помощью QR-разложения методом отражений Хаусхолдера

В рамках данной задачи реализованы следующие алгоритмы:

- Отражение Хаусхолдера;
- Обычное QR-разложение;
- QR-разложение для метода LLS;
- Обратный ход метода Гаусса;
- Linear Least Squares с помощью QR-разложения.

Расчетная реализация всех алгоритмов выполнена на языке `C++` с использованием библиотеки `Eigen` для базовых матричных операций. Для отладки программы и проверки корректности результатов реализован вспомогательный скрипт на `Wolfram Mathematica`, использующий встроенные методы для получения необходимых разложений. При указании асимптотической сложности алгоритмов далее будет подразумеваться $m \sim n$.

2.1. QR-разложение и его эффективная реализация

Алгоритм QR-разложения имеет следующий вид:

Algorithm 1: QR Factorization (naive)

Data: $n \leq m$

Input: $A_{m \times n}$

Output: $Q_{m \times n}$, $R_{n \times n}$

$\tilde{Q} \leftarrow I_{m \times m}$;

$\tilde{R} \leftarrow A_{m \times n}$;

for $i \leftarrow 1$, $i \leq \min\{m-1, n\}$, $i \leftarrow i+1$ **do**

$k \leftarrow m - i$;

$u_i \leftarrow \text{HouseholderReflection}(\tilde{R}[i{:}m, i])_{k \times 1}$;

$\tilde{P}_i \leftarrow I_{k \times k} - 2u_i u_i^T$;

$P_i \leftarrow I_{m \times m}$;

$P_i[i{:}m, i{:}m] \leftarrow \tilde{P}_i$;

$\tilde{Q} \leftarrow \tilde{Q} \cdot P_i$;

$\tilde{R} \leftarrow P_i \cdot \tilde{R}$;

end

$Q \leftarrow \tilde{Q}[1{:}m, 1{:}n]$;

$R \leftarrow \tilde{R}[1{:}n, 1{:}n]$;

Записанный в явном виде алгоритм QR-разложения имеет сложность $O(n^4)$ в силу наличия матричного умножения, это решается если подставить матрицу P_i явно и расписать матричные умножения как 2 умножения матрицы на вектор. Приходим к алгоритму, эффективному для практической реализации:

Algorithm 2: QR Factorization

Data: $n \leq m$

Input: $A_{m \times n}$

Output: $Q_{m \times n}$, $R_{n \times n}$

$\tilde{Q} \leftarrow I_{m \times m}$;

$\tilde{R} \leftarrow A_{m \times n}$;

for $i \leftarrow 1$, $i \leq \min\{m-1, n\}$, $i \leftarrow i+1$ **do**

$k \leftarrow m - i$;

$u_i \leftarrow \text{HouseholderReflection}(\tilde{R}[i{:}m, i])_{k \times 1}$;

$\tilde{Q}[1{:}m, i{:}m] \leftarrow \tilde{Q}[1{:}m, i{:}m] - 2\tilde{Q}[1{:}m, i{:}m] \cdot u_i \cdot u_i^T$;

$\tilde{R}[i{:}m, i{:}n] \leftarrow \tilde{R}[i{:}m, i{:}n] - 2u_i \cdot (u_i^T \cdot \tilde{R}[i{:}m, i{:}n])$;

end

$Q \leftarrow \tilde{Q}[1{:}m, 1{:}n]$;

$R \leftarrow \tilde{R}[1{:}n, 1{:}n]$;

Полученный алгоритм имеет сложность $O(n^3)$. Далее, под QR-разложением будет подразумеваться именно эта реализация. HouseholderReflect() в данном случае является вспомогательной процедурой, имеющей сложность $O(n)$:

Algorithm 3: Householder Reflection

Input: $x_{k \times 1}$

Output: $u_{k \times 1}$

$u \leftarrow x_{k \times 1}$;

$u[1] \leftarrow \text{sign}(u[1])||u||_2$;

$u \leftarrow u/||u||_2$;

2.2. Алгоритм LLS и вспомогательные процедуры

Алгоритм линейной задачи наименьших квадратов имеет следующий вид:

Algorithm 4: LLS

Data: $n \leq m$

Input: $A_{m \times n}$, $b_{m \times 1}$

Output: $x_{m \times 1}$

$\{Q, R\} \leftarrow \text{QRFactorization}(A)$;

$x \leftarrow \text{GaussianBackwardsElimination}(R, Q^T \cdot b)$;

тут GaussianBackwardsElimination() — процедура обратного хода метода Гаусса, имеет следующую реализацию:

Algorithm 5: Gaussian Backwards Elimination

Data: A is upper-triangular

Input: $R_{n \times n}$, $rhs_{n \times 1}$

Output: $x_{n \times 1}$

$x \leftarrow rhs_{n \times 1}$;

for $i \leftarrow n$, $i \geq 1$, $i \leftarrow i - 1$ **do**

  for $j \leftarrow i + 1$, $j \leq n$, $j \leftarrow j + 1$ **do**

    $x[i] \leftarrow x[i] - R[i, j] \cdot x[j]$;

  end

  $x[i] \leftarrow x[i]/R[i, i]$;

end

2.3. Улучшение алгоритма LLS

Алгоритм LLS можно улучшить если встроить вычисление правой части $Q^T b$ сразу в QRFactorization. Получим следующую модификацию QR-разложения:

Algorithm 6: QR Factorization for LLS

Data: $n \leq m$

Input: $A_{m\times n},\ b_{m\times 1}$

Output: $\beta_{n\times 1},\ R_{n\times n}$

$\tilde{\beta} \leftarrow b_{m\times 1}$;

$\tilde{R} \leftarrow A_{m\times n}$;

for $i \leftarrow 1,\ i \leq \min\{m-1, n\},\ i \leftarrow i+1$ **do**

$k \leftarrow m - i$;

$u_i \leftarrow \text{HouseholderReflection}(\tilde{R}[i{:}m, i])_{k\times 1}$;

$\gamma \leftarrow -2u_i^T \cdot \tilde{\beta}[i{:}m]$;

$\tilde{\beta}[i{:}m] \leftarrow \tilde{\beta}[i{:}m] + \gamma \cdot u_i$

$\tilde{R}[i{:}m, i{:}n] \leftarrow \tilde{R}[i{:}m, i{:}n] - 2u_i \cdot (u_i^T \cdot \tilde{R}[i{:}m, i{:}n])$;

end

$\beta \leftarrow \tilde{\beta}[1{:}n]$;

$R \leftarrow \tilde{R}[1{:}n, 1{:}n]$;

Соответсвенно, разложение в LLS станет происходить не в $\{Q, R\}$, а сразу в матрицу и вектор правой части СЛАУ $\{Q^T b, R\}$.

2.4. Результаты

Произведено тестовое QR-разложение матрицы $\hat{A}$, проверена ортогональность матрицы Q и приблизительное совпадение $QR \approx A$. Результаты сходятся с разложением с помощью встроенного метода `QRDecomposition[]` в пакете `Wolfram Mathematica`:

В качестве тестового вектора выбран $x_0{:}x_i = i^2$. Для него решена задача наименьших квадратов относительно невязки, получен вектор x_{LLS} имеющий следующую норму ошибки:

$$err_{LLS} \approx 3.99 \cdot 10^{-16}.$$

Результаты LLS также сверены с помощью скрипта.

3. Получение собственных значений матрицы с помощью QR-алгоритма со сдвигами

В рамках данной задачи реализованы следующие алгоритмы:

- Приведение матрицы к Хессенберговой форма;
- QR-алгоритм без сдвигов (для отладочных целей);
- $O(N^2)$ QR-разложение для верхне-хессенберговых матриц с вычислением RQ;
- QR-алгоритм со сдвигами и приведением изначальной матрицы к Хессенберговой форме;
- Метод обратной итерации для определения СВ.

3.1. QR-алгоритм

Algorithm 7: QR-algorithm (naive)

Input: $A_{n\times n}$

Output: $T_{n\times n}$

$T \leftarrow A_{n\times n}$;

for $it \leftarrow 0$, $it \leq \text{limit}$, $it \leftarrow it + 1$ **do**

  $\sigma \leftarrow \sigma_i$;

  $\{Q, R\} \leftarrow \text{QRFactorization}(A - \sigma I)$;

  $T \leftarrow R \cdot Q + \sigma I$;

end

Результатом работы алгоритма является матрица T из разложения Шура $A = Q^*TQ$, элементы главной диагонали данной матрицы являются собственными значениями соответствующего оператора.

Заметим, что на данном этапе сложно сформулировать хороший критерий остановки, поэтому итерируемся фиксированное число раз или до тех пор пока разница между итерациями не станет достаточно малой. Вопрос определения σ_i также пока что не конкретезируется, в случае $\sigma_i = 0$ получаем метод обычной QR-итерации без сдвига. И то и другое будет конкретизировано в следующем разделе при улучшении метода.

Данный алгоритм требует $O(n^4)$ операций т.к. QR-разложение обходится в $O(n^3)$, матричное умножение также требует $O(n^3)$ и ожидаемое число итераций до сходимости считаем пропорциональным размеру.

3.2. Улучшение алгоритма с помощью приведения к Хессенберговой форме и редукции размерности

Приведенный выше алгоритм QR-итерации можно улучшить, внесем следующий набор изменений в логику метода:

1. Приведем матрицу A к верхне-хессенберговой форме с помощью отражений Хаусхолдера (Hessenberg Reduction), данная операция корректна т.к. не меняет собственные числа матрицы, приведение к верхне-хессенберговой форме требует $O(n^3)$ операций;
2. Реализуем модификацию QRFactorization() для верхне-хессенберговых матриц, сложность разложения упадет с $O(n^3)$ до $O(n^2)$;
3. Встроим получение матрицы RQ в процедуру QR-разложения, это позволит избежать явного умножения матриц и снизит сложность получения RQ до $O(n^2)$, таким образом вся QR-итерация снизится с $O(n^4)$ до $O(n^3)$;
4. Выберем сдвиг $\sigma_i = T_{nn}$, данный выбор сдвига обеспечивает стремление последней строки к соответствующей строке матрицы Шура, итерации производим до тех пор пока значение $|T_{n,n-1}|$ не станет меньше некоторого ε, после чего n-е значение на диагонали считаем найденым и

редуцируем задачу к работе с блоком $(n-1) \times (n-1)$. Получили конкретизацию сдвига и условия остановки.

Соответсвенно, алгоритм QR-итерации придет к виду:

Algorithm 8: QR-algorithm

Input: $A_{n \times n}$

Output: $T_{n \times n}$

$it \leftarrow 0$;

$k \leftarrow n$;

$T \leftarrow \text{HessenbergReduction}(A)_{n \times n}$;

while $k \geq 2$ & $it \leq \text{limit}$ **do**

  $\sigma \leftarrow T[k, k]$;

  $RQ \leftarrow \text{QRFactorizationForHessenberg}(A - \sigma I)_{k \times k}$;

  $T[1{:}k, 1{:}k] \leftarrow RQ$;

  if $|T[k, k-1]| < \varepsilon$ **then**

    $k \leftarrow k - 1$;

  end

end

Вспомогательная процедура HessenbergReduction() имеет алгоритм:

Algorithm 9: Hessenberg reduction

Input: $A_{n \times n}$

Output: $H_{n \times n}$

$H \leftarrow A_{n \times n}$;

for $i \leftarrow 1$, $i \leq n-2$, $i \leftarrow i+1$ **do**

  $k \leftarrow m - i - 1$;

  $u_i \leftarrow \text{HouseholderReflection}(R[i+1{:}n, i])_{k \times 1}$;

  $H[i+1{:}n, i{:}n] \leftarrow H[i+1{:}n, i{:}n] - 2u_i \cdot (u_i^T \cdot H[i+1{:}n, i{:}n])$;

  $H[1{:}n, i+1{:}n] \leftarrow H[1{:}n, i+1{:}n] - 2H[1{:}n, i+1{:}n] \cdot u_i \cdot u_i^T$;

end

Использование отражений Хаусхолдера для редукции к верхне-хессенберговой форме во многом аналогично их применению в QR-разложении, с той разницей, что вместо обнуления всех s_i поддиагональных элементов происходит обнуление $s_i - 1$.

QR-разложение в силу Хессенберговой формы матрицы можем записать работающее для блоков ширины 2, что снизит сложность до $O(n^2)$:

Algorithm 10: QR Factorization for Hessenberg matrices

Data: A is an upper-hessenberg matrix

Input: $A_{n\times n}$

Output: $Q_{n\times n}$, $R_{n\times n}$, $RQ_{n\times n}$

$Q \leftarrow I_{n\times n}$;

$R \leftarrow A_{n\times n}$;

$V \leftarrow \text{Zero}_{n\times n}$;

for $i \leftarrow 1,\ i \leq n-1,\ i \leftarrow i+1$ **do**

$k \leftarrow m - i$;

$u_i \leftarrow \text{HouseholderReflection}(R[i{:}i+1, i])_{k\times 1}$;

$Q[1{:}n, i{:}i+1] \leftarrow Q[1{:}n, i{:}i+1] - 2Q[1{:}n, i{:}i+1] \cdot u_i \cdot u_i^T$;

$R[i{:}i+1, i{:}n] \leftarrow R[i{:}i+1, i{:}n] - 2u_i \cdot (u_i^T \cdot R[i{:}i+1, i{:}n])$;

$V[i{:}i+1, i] \leftarrow u_i$;

end

$RQ \leftarrow R$;

for $i \leftarrow 1,\ i \leq n-1,\ i \leftarrow i+1$ **do**

$v_i \leftarrow V[i{:}i+1, i]$;

$RQ[1{:}m, i{:}i+1] \leftarrow RQ[1{:}m, i{:}i+1] - 2RQ[1{:}m, i{:}i+1] \cdot v_i \cdot v_i^T$;

end

Заметим, что непосредственно для QR-алгоритма матрицы Q, R не нужны, их расчет реализован для общности и для отладочных целей.

3.3. Обратная итерация

Зная собственные значения из разложения Шура, собственные векторы можем получить с помощью метода обратной итерации:

Algorithm 11: Inverse iteration

Input: $A_{n\times n}$, λ_0

Output: λ, $z_{n\times 1}$

$i \leftarrow 0$;

$x \leftarrow \text{Ones}_{n\times 1}/n$; /* $||x||_2 = 1$ */

repeat

  $i \leftarrow i + 1$;

  $x \leftarrow \text{GaussianElimination}(A - \lambda_0 I, x)$;

  $x \leftarrow x/||x||_2$; /* $||x||_2 = 1$ */

  $\lambda_i \leftarrow x^T \cdot A \cdot x$;

until $|\lambda_i - \lambda_{i-1}| < \varepsilon$;

$\lambda \leftarrow \lambda_i$;

$z \leftarrow x_{n\times 1}$;

В данном случае GaussianElimination() является вспомогательной процедурой для решения СЛАУ, выбор конкретно данного метода существенным не является. В программе реализован вариант с частичным выбором ведущего элемента и предобуславливателем Якоби.

3.4. Результаты

Получены собственные значения и векторы матрицы A, результаты сравнения с аналитическими результатами приведены ниже для $\varepsilon = 10^{-1}, 10^{-3}, 10^{-6}$:

j	$\lvert\lambda_j^0 - \lambda_j\rvert$	Итераций до редукции	$\lvert\lvert z_j^0 - z_j\rvert\rvert_2$	Обратных итераций
1	0.0383792	5	0.0615102	1
2	0.00164998	4	0.0551268	1
3	0.00198987	9	0.0322314	1
4	0.00445514	13	0.00774489	1
5	0.00474202	12	0.00858965	1
6	0.00674079	17	0.00774881	1
7	0.00682261	15	0.00972806	1
8	0.00718661	19	0.0116313	1
9	0.00664725	17	0.0136519	1
10	0.00542461	129	0.0105261	1

Таблица 1. Ошибки при $\varepsilon = 10^{-1}$

j	$\lvert\lambda_j^0 - \lambda_j\rvert$	Итераций до редукции	$\lvert\lvert z_j^0 - z_j\rvert\rvert_2$	Обратных итераций
1	0.000395515	9	0.000543584	1
2	$1.1599 \cdot 10^{-5}$	2	0.000475849	1
3	$1.4605 \cdot 10^{-5}$	13	0.000301104	1
4	$4.5041 \cdot 10^{-5}$	8	$7.3896 \cdot 10^{-5}$	1
5	$4.8062 \cdot 10^{-5}$	17	$8.4203 \cdot 10^{-5}$	1
6	$6.7395 \cdot 10^{-5}$	12	$7.7519 \cdot 10^{-5}$	1
7	$6.8274 \cdot 10^{-5}$	20	$9.7227 \cdot 10^{-5}$	1
8	$7.1855 \cdot 10^{-5}$	14	0.000116509	1
9	$6.6562 \cdot 10^{-5}$	22	0.000137413	1
10	$5.4530 \cdot 10^{-5}$	129	0.000106285	1

Таблица 2. Ошибки при $\varepsilon = 10^{-3}$

j	$\lvert\lambda_j^0 - \lambda_j\rvert$	Итераций до редукции	$\lVert z_j^0 - z_j\rVert_2$	Обратных итераций
1	$3.9562 \cdot 10^{-7}$	16	$5.4294 \cdot 10^{-7}$	1
2	$1.1556 \cdot 10^{-8}$	1	$4.8413 \cdot 10^{-7}$	1
3	$1.4555 \cdot 10^{-8}$	19	$3.0090 \cdot 10^{-7}$	1
4	$4.5045 \cdot 10^{-8}$	4	$8.7816 \cdot 10^{-8}$	1
5	$4.8068 \cdot 10^{-8}$	22	$8.4187 \cdot 10^{-8}$	1
6	$6.7395 \cdot 10^{-8}$	8	$7.4401 \cdot 10^{-8}$	1
7	$6.8274 \cdot 10^{-8}$	28	$9.7226 \cdot 10^{-8}$	1
8	$7.1855 \cdot 10^{-8}$	11	$6.7154 \cdot 10^{-7}$	1
9	$6.6563 \cdot 10^{-8}$	30	$1.3742 \cdot 10^{-7}$	1
10	$5.4533 \cdot 10^{-8}$	81	$1.0629 \cdot 10^{-7}$	3

Таблица 3. Ошибки при $\varepsilon = 10^{-6}$

Пример сводки результатов расчета для малого n (в силу вербозности результата при $n = 10$) приведен в приложении.

4. Листинг расчетной программы на C++

source/utils.hpp

```cpp
#pragma once

#include "firstparty/proto_utils.hpp"
#include "thirdparty/Eigen/Dense"
#include "thirdparty/Eigen/src/Core/Matrix.h"
#include "thirdparty/Eigen/src/Core/util/Meta.h"
#include <limits>



using Matrix    = Eigen::MatrixXd;
using Vector    = Eigen::VectorXd;
using RowVector = Eigen::RowVectorXd;
using Idx       = Eigen::Index; // Eigen uses signed (!) indeces

constexpr bool collapse_small_values = false;

// Eigen has formatting options built-in, but I prefer the style of my own package.
// Eigen stores matrices as col-major so we do a matrix view into the CR memory layout.
inline std::string stringify_matrix(const Matrix& eigen_matrix) {
    using namespace utl;

    if constexpr (collapse_small_values) {
        utl::mvl::Matrix<double> mvl_matrix(
            eigen_matrix.rows(), eigen_matrix.cols(),
            [&](std::size_t i, std::size_t j) { return (std::abs(eigen_matrix(i, j)) < 1e-12) ? 0. : eigen_matrix(i, j); });
        return utl::mvl::format::as_matrix(mvl_matrix);
    } else {
        mvl::ConstMatrixView<double, mvl::Checking::BOUNDS, mvl::Layout::CR> view(
            eigen_matrix.rows(), eigen_matrix.cols(), eigen_matrix.data());
        return mvl::format::as_matrix(view);
    }
}
```

source/slae.hpp

```cpp
#pragma once

#include "utils.hpp"
#include <cassert>



// Backwards gaussian elimination. O(N^2) complexity.
//
// Assumes 'R' to be upper-triangular matrix.
//
inline Vector backwards_gaussian_elimination(const Matrix& R, Vector rhs) {
    assert(R.rows() == R.cols());
    assert(R.rows() == rhs.rows());

    for (Idx i = R.rows() - 1; i >= 0; --i) {
        for (Idx j = i + 1; j < R.cols(); ++j) rhs(i) -= R(i, j) * rhs(j);
        rhs(i) /= R(i, i);
    }

    return rhs;
}


// Forward gaussian elimination. O(N^3) complexity.
//
// Partial pivoting.
//
inline void partial_piv_forward_gaussian_elimination(Matrix& A, Vector& rhs) {
    assert(A.rows() == A.cols());
    assert(A.rows() == rhs.rows());

    for (Idx i = 0; i < A.rows(); ++i) {
        // Partial pivot
        Idx i_max = i;
        for (Idx ii = i; ii < A.rows(); ++ii)
            if (std::abs(A(ii, i)) > std::abs(A(i_max, i))) i_max = ii;

        if (i != i_max) {
            // Swap rows (matrix)
            const Vector tmp_A   = A.row(i);
            A.row(i)             = A.row(i_max);
            A.row(i_max)         = tmp_A;
            // Swap rows (rhs)
            const double tmp_rhs = rhs(i);
            rhs(i)               = rhs(i_max);
            rhs(i_max)           = tmp_rhs;
        }

        // Elimination (normalize current row)
        const double factor = 1. / A(i, i);
        for (Idx j = i; j < A.cols(); ++j) A(i, j) *= factor;
        rhs(i) *= factor;

        // Elimination (substract current row from all the rows below)
        for (Idx k = i + 1; k < A.rows(); ++k) {
```

```
57              const double first = A(k, i);
58              for (Idx j = i; j < A.cols(); ++j) A(k, j) -= first * A(i, j);
59              rhs(k) -= first * rhs(i);
60          }
61      }
62  }
63
64  // Forward + backwards gaussian elimination. O(N^3) complexity.
65  //
66  // Partial pivoting. Jacobi preconditioner.
67  //
68  inline Vector partial_piv_gaussian_elimination(Matrix A, Vector rhs) {
69      // Jacobi preconditioner
70      //
71      // When testing N = 4 one of the (A - lambda I) matrices with epsilon = 0.1
    was almost degenerate (det = 1e-7),
72      // without preconditioning it went into 'nan's, Jacobi was just good enough to
    prevent this.
73      //
74      // N = 10 case is luckier and doesn't require this, but why not do it
    regardless.
75      //
76      Matrix preconditioner     = Matrix::Zero(A.rows(), A.cols());
77      preconditioner.diagonal() = A.diagonal();
78      A                         = preconditioner * A;
79      rhs                       = preconditioner * rhs;
80
81      // Gaussian elimination
82      partial_piv_forward_gaussian_elimination(A, rhs);
83      return backwards_gaussian_elimination(A, rhs);
84  }
```

source/qr_factorization.hpp

```cpp
#pragma once

#include "utils.hpp"



// Householder reflection operation. O(N) complexity.
//
// Template so we can take vectors/blocks/views as an argument and not force a copy.
//
template <class VectorType>
Vector householder_reflect(const VectorType& x) {

    // u = { x[0] + sign(x[0]) * ||x||_2, x[1], x[2], ... , x[K] }
    Vector u = x;
    u(0) += utl::math::sign(u(0)) * u.norm();

    return u.normalized();
}

// QR factorization. O(N^3) complexity.
//
// Original alg would be:
// ------------------------------------------------------------------------
// -   Q_wave = I;                                                        -
// -   R_wave = A;                                                        -
// -   for i = 1,min(M-1,N) {                                            -
// -      ui                = House(R_wave[i:M, i])      // O(N)        -
// -      pi_wave           = I - 2 ui ui^T              // O(N^2)      -
// -      pi                = I                          //              -
// -      pi[i:M, i:M]      = pi_wave                    //              -
// -      R_wave[i:M, i:N]  = pi_wave * R_wave[i:M, i:N] // O(N^3)      -
// -      Q_wave[0:M, i:M]  = Q_wave[0:M, i:M] * pi_wave // O(N^3)      -
// -   }                                                                 -
// -   return { Q[0:M, 0:N], R[0:N, 0:N] }                              -
// ------------------------------------------------------------------------
//
// After the algorithm we end up with a following decomposition:
// A = p1 * ... * pN * rcat[ R 0 ]
//     ^^^^^^^^^^^^^   ^^^^^^^^^^^
//     Q_wave          R_wave
// where Q_wave and R_wave are "extended" matrices Q and R, to get proper QR we need to trim a few rows/cols at the end
//
// This alg also results in O(N^4). We can rewrite it by substituting 'pi_wave' directly and doing
// 2 matrix*vector products instead of 1 matrix*matix, which brings complexity down to O(N^3).
//
// ------------------------------------------------------------------------
// -   Q_wave = I;                                                        -
// -   R_wave = A;                                                        -
// -   for i = 1,min(M-1,N) {                                            -
// -      ui                = House(R_wave[i:M, i])          // O(N)    -
// -      R_wave[i:M, i:N] -= 2 ui (ui^T * R_wave[i:M, i:N]) // O(N^2)  -
// -      Q_wave[0:M, i:M]  -= Q_wave[0:M, i:M] * 2 ui ui^T  // O(N^2)  -
```

```cpp
// -   }                                                                    -
// -   return { Q[0:M, 0:N], R[0:N, 0:N] }                                  -
// ---------------------------------------------------------------------------
//
inline std::pair<Matrix, Matrix> qr_factorize(const Matrix& A) {
    const auto M = A.rows();
    const auto N = A.cols();

    Matrix Q_wave = Matrix::Identity(M, M);
    Matrix R_wave = A;

    for (Idx i = 0; i < std::min(M - 1, N); ++i) {
        const Vector ui = householder_reflect(R_wave.block(i, i, M - i, 1));
// O(N)
        R_wave.block(i, i, M - i, N - i) -= 2. * ui * (ui.transpose() *
R_wave.block(i, i, M - i, N - i)); // O(N^2)
        Q_wave.block(0, i, M, M - i) -= Q_wave.block(0, i, M, M - i) * 2. * ui *
ui.transpose();           // O(N^2)
    }

    return {Q_wave.block(0, 0, M, N), R_wave.block(0, 0, N, N)};
}

// A variant of QR-decomposition used for linear least squares. O(N^3)
complexity.
//
// Is is more efficient since in LSQ we don't need 'Q' explicitly,
// we can directly compute 'Q^T b'.
//
inline std::pair<Matrix, Matrix> qr_factorize_lls(const Matrix& A, const Vector&
b) {
    const auto M = A.rows();
    const auto N = A.cols();

    Matrix R_wave = A;
    Vector QTb    = b;

    for (Idx i = 0; i < std::min(M - 1, N); ++i) {
        // ui                = House(R_wave[i:M, i])
        // R_wave[i:M, i:N] -= 2 ui ui^T * R_wave[i:M, i:N]
        const Vector ui = householder_reflect(R_wave.block(i, i, M - i, 1));
// O(N)
        R_wave.block(i, i, M - i, N - i) -= 2. * ui * (ui.transpose() *
R_wave.block(i, i, M - i, N - i)); // O(N^2)

        // gamma     = - 2 ui^T QTb[i:M]
        // QTb[i:M] += gamma * ui
        const Matrix gamma = -2. * ui * QTb.segment(i, M - i).transpose(); //
O(N)
        QTb.segment(i, M - i) += gamma * ui;                               //
O(N^2)
    }

    return {QTb.segment(0, N), R_wave.block(0, 0, N, N)};
}

// A variant of QR-decomposition used decomposing upper-hessenberg matrices in
QR-iteration. O(N^2) complexity.
//
// Returns { Q, R, RQ }. Technically we only need RQ for for the QR-algorithm,
but for testing purposes { Q, R}
```

```
104 // are left the same.
105 //
106 // Same algorithm as regular QR factorization, except instead of blocks of 'M -
    i' rows/cols we
107 // have blocks of '2' rows/cols, which reduces O(N^2) operations to O(N).
108 //
109 inline std::tuple<Matrix, Matrix, Matrix> qr_factorize_hessenberg(const Matrix&
    A) {
110     assert(A.rows() == A.cols());
111 
112     const auto M = A.rows();
113 
114     Matrix Q = Matrix::Identity(M, M);
115     Matrix R = A;
116     Matrix V = Matrix::Zero(M, M);
117 
118     // Compute { Q, R } in O(N^2)
119     for (Idx i = 0; i < M - 1; ++i) {
120         const Vector ui = householder_reflect(R.block(i, i, 2, 1));
    // O(N)
121         R.block(i, i, 2, M - i) -= 2. * ui * (ui.transpose() * R.block(i, i, 2, M
    - i)); // O(N)
122         Q.block(0, i, M, 2) -= Q.block(0, i, M, 2) * 2. * ui * ui.transpose();
    // O(N)
123         V.block(i, i, 2, 1) = ui;
    // O(N)
124     }
125 
126     // Compute { RQ } in O(N^2)
127     Matrix RQ = R;
128     for (Idx i = 0; i < M - 1; ++i) {
129         Vector vi = V.block(i, i, 2, 1);
130         RQ.block(0, i, M, 2) -= RQ.block(0, i, M, 2) * 2. * vi * vi.transpose();
    // O(N)
131     }
132 
133     return {Q, R, RQ};
134 }
135 
136 // Hessenberg QHQ^T-factorization using householder reflections. O(N^3)
    complexity.
137 //
138 // Algorithm:
139 // ---------------------------------------------------------------
140 // -   H = A;                                                     -
141 // -   for i = 1, M - 2 {                                        -
142 // -      ui = House(H[i+1:M, i])                    // O(N)     -
143 // -      H[i+1:M, i:M] -= 2 ui (ui^T * H[i+1:M, i:M]) // O(N^2)  -
144 // -      H[1:M, i+1:M] -= 2 (H[1:M, i+1:M] * ui) ui^T // O(N^2)  -
145 // -   }                                                         -
146 // ---------------------------------------------------------------
147 //
148 inline Matrix hessenberg_reduce(const Matrix& A) {
149     assert(A.rows() == A.cols());
150 
151     const Idx M = A.rows();
152 
153     Matrix H = A;
154 
155     for (Idx i = 0; i < M - 2; ++i) {
156         const Vector ui = householder_reflect(H.block(i + 1, i, M - i - 1, 1));
```

```
157         H.block(i + 1, i, M - i - 1, M - i) -= 2. * ui * (ui.transpose() *
    H.block(i + 1, i, M - i - 1, M - i));
158         H.block(0, i + 1, M, M - i - 1) -= 2. * (H.block(0, i + 1, M, M - i - 1)
    * ui) * ui.transpose();
159     }
160 
161     return H;
162 }
```

source/linear_least_squares.hpp

```cpp
#pragma once

#include "qr_factorization.hpp"
#include "slae.hpp"



// Linear Least Squares problem. O(N^3) complexity.
//
// LLS has a following solution:
//    x = A^+ b
//    where A^+ = R^-1 * Q^T
//
// We can rewrite it as a SLAE:
//    R x = Q^t b
//
// since 'R' is upper-triangular, we only need to do the backwards gaussian elimination, which is O(N^2).
//
Vector linear_least_squares(const Matrix& A, const Matrix& b) {
    // Computing QR the usual way
    // const auto [Q, R] = qr_factorize(A);
    // const auto x      = backwards_gaussian_elimination(R, Q.transpose() * b);

    // Computing QR with (Q^T * b) directly
    const auto [QTb, R] = qr_factorize_lls(A, b);
    const auto x        = backwards_gaussian_elimination(R, QTb);

    return x;
}
```

source/eigenvalues.hpp

```cpp
#pragma once

#include "qr_factorization.hpp"
#include "slae.hpp"
#include "thirdparty/Eigen/src/Core/util/Constants.h"
#include "utils.hpp"
#include <cassert>
#include <cstddef>
#include <cstdio>
#include <limits>
#include <vector>

#include "thirdparty/Eigen/Core"

// QR-method for eigenvalues with NO shift and NO Hessenberg form optimization.
//
// Used as a reference. O(N^3) single iteration complexity.
//
Matrix eigenvalues_prototype(const Matrix& A) {
    assert(A.rows() == A.cols());

    Matrix T_shur = A;

    for (Idx i = 0; i < A.rows() * 100; ++i) {
        const auto [Q, R] = qr_factorize(T_shur); // O(N^3)
        T_shur            = R * Q;                // O(N^3)
        // no stop condition, just do a ton of iterations
    }

    return T_shur;
}

// QR-method for eigenvalues with shifts and Hessenberg form optimization.
//
// Requires 'A' to be in upper-Hessenber form (!).
// Using Hessenberg form brings complexity down to O(N^2) per iteration.
//
// Algorithm:
// ---------------------------------------------------------------------------
// -   while (N >= 2 && iteration++ < max_iterations) {                      -
// -      sigma = T_shur[N, N]                                  // O(1)     -
// -      [ Q, R, RQ ] = qr_factorize_hessenberg(T_shur[1:N, 1:N]) // O(N^2) -
// -      T_shur[0:N, 0:N] = RQ + sigma I                       // O(N^2)   -
// -      if (|T_shur[N, N-1]| < eps) --N                       // O(1)     -
// -   }                                                                     -
// ---------------------------------------------------------------------------
//
// Note that matrix multiplication here is O(N^2) because 'R' is tridiagonal.
//
// As a stop-condition for deflating the block we use last row element under the
diagonal,
// as soon as it becomes "small enough" the block can deflate.
//
// 'Q' and 'R' matrices aren't directly used anywhere, but still computed for
debugging purposes.
//
// Returns { T_shur, numer_of_iteration_for_each_eigenvalue }
```

```cpp
//
std::pair<Matrix, std::vector<std::size_t>> qr_algorithm(const Matrix& A) {
    assert(A.rows() == A.cols());

    const std::size_t max_iterations = 500 * A.rows();
    std::size_t       iteration      = 0;
    Idx               N              = A.rows(); // mutable here since we shrink the working block (!)

    Matrix                   T_schur = A;
    std::vector<std::size_t> iteration_counts;
    iteration_counts.reserve(N);

    while (N >= 2 && iteration++ < max_iterations) {
        const double sigma = T_schur(N - 1, N - 1); // O(1)
        [[maybe_unused]] const auto [Q, R, RQ] =
            qr_factorize_hessenberg(T_schur.block(0, 0, N, N) - sigma * Matrix::Identity(N, N)); // (N^2)
        T_schur.block(0, 0, N, N) = RQ + sigma * Matrix::Identity(N, N); // (N^2)
        if (std::abs(T_schur(N - 1, N - 2)) < std::numeric_limits<double>::epsilon()) {          // O(1)
            --N;
            if (iteration_counts.empty()) iteration_counts.push_back(iteration);
            else iteration_counts.push_back(iteration - iteration_counts.back());
        }
    }

    return {T_schur, iteration_counts};
}

// Reverse iteration for computing eigenvectors and (possible) imroving eigenvalues.
//
// Returns { eigenvalue, eigenvector, number_of_iterations }
//
std::tuple<double, Vector, std::size_t> reverse_iteration(const Matrix& A, double lambda_0) {
    assert(A.rows() == A.cols());

    const std::size_t max_iterations = 1 * A.rows();
    std::size_t       iteration      = 0;
    const Idx         N              = A.rows();

    double lambda;
    Vector x = Vector::Ones(N) / N; // ||x0||_2 should be 1

    while (iteration++ < max_iterations) {
        x      = partial_piv_gaussian_elimination(A - lambda_0 * Matrix::Identity(N, N), x).normalized();
        lambda = x.transpose() * A * x;
        if (std::abs(lambda - lambda_0) < 1e-12) break;
        lambda_0 = lambda;
    }

    if (x(0) < 0) x *= -1; // "standardize" eigenvec signs
    return {lambda, x, iteration};
}
```

source/main.cpp

```cpp
#include "eigenvalues.hpp"
#include "linear_least_squares.hpp"
#include "utils.hpp"
#include <cmath>
#include <tuple>



int main() {
    using namespace utl;

    // ===============
    // --- Problem ---
    // ===============

    constexpr double      Nvar    = 1;
    constexpr double      epsilon = 1e-6; // 1e-1, 1e-3, 1e-6
    constexpr double      c       = Nvar / (Nvar + 1.) * epsilon;
    constexpr std::size_t N       = 4;

    // A0 = {  2, if (j == j)
    //      { -1, if (i == j - 1 || i == j + 1)
    //      {  0, else
    Matrix A0(N, N);
    for (Idx i = 0; i < A0.rows(); ++i)
        for (Idx j = 0; j < A0.cols(); ++j) A0(i, j) = (i == j) ? 2. :
(std::abs(i - j) == 1) ? -1. : 0.;

    // deltaA = { c / (i + j), if (i != j)
    //          {           0, else
    Matrix deltaA(N, N);
    for (Idx i = 0; i < deltaA.rows(); ++i)
        for (Idx j = 0; j < deltaA.cols(); ++j) deltaA(i, j) = (i != j) ? c / (i
+ j + 2) : 0;

    // A     = A0 + deltaA
    const Matrix A = A0 + deltaA;

    // A_hat = <A without the last column>
    const Matrix A_hat = A.block(0, 0, A.rows(), A.cols() - 1);

    log::println("---------------");
    log::println("--- Problem ---");
    log::println("---------------");
    log::println();
    log::println("epsilon -> ", epsilon);
    log::println("N       -> ", N);
    log::println("A0      -> ", stringify_matrix(A0));
    log::println("deltaA  -> ", stringify_matrix(deltaA));
    log::println("A       -> ", stringify_matrix(A));
    log::println("A_hat   -> ", stringify_matrix(A_hat));

    // ==============
    // --- Task 1 ---
    // ==============
    //
    // Solving LLS (Linear Least Squares) with QR factorization method.
```

```cpp
    //

    // Try QR decomposition to verify that it works
    const auto [Q, R] = qr_factorize(A_hat);

    log::println("------------------------");
    log::println("--- QR factorization ---");
    log::println("------------------------");
    log::println();
    log::println("Q             -> ", stringify_matrix(Q));
    log::println("R             -> ", stringify_matrix(R));
    log::println("Verification:");
    log::println();
    log::println("Q^T * Q       -> ", stringify_matrix(Q.transpose() * Q));
    log::println("Q * R - A_hat -> ", stringify_matrix(Q * R - A_hat));

    // Generate some 'x0',
    // set b = A_hat * x0

    Vector x0(N - 1);
    for (Idx i = 0; i < x0.rows(); ++i) x0(i) = math::sqr(i + 1);
    const Vector b = A_hat * x0;

    // Solve LLS
    const Vector x_lls = linear_least_squares(A_hat, b);

    // Relative error estimate ||x_lls - x0||_2 / ||x0||_2
    const double lls_error_estimate = (x_lls - x0).norm() / x0.norm();

    log::println("-------------------------------------");
    log::println("--- Linear Least Squares solution ---");
    log::println("-------------------------------------");
    log::println();
    log::println("x0                -> ", stringify_matrix(x0));
    log::println("b                 -> ", stringify_matrix(b));
    log::println("x_lls             -> ", stringify_matrix(x_lls));
    log::println("lls_error_estimate -> ", lls_error_estimate);
    log::println();

    // ==============
    // --- Task 2 ---
    // ==============
    //
    // Computing eigenvalues of the matrix using QR method with a shift.
    //

    // Compute analythical eigenvalues
    Vector lambda0(N);
    for (Idx j = 0; j < lambda0.size(); ++j) lambda0(j) = 2. * (1. -
std::cos(math::PI * (j + 1) / (N + 1)));
    std::sort(lambda0.begin(), lambda0.end());

    // Compute analythical eigenvectors (columns of the matrix store vectors)
    Matrix z0(N, N);
    for (Idx k = 0; k < z0.cols(); ++k)
        for (Idx i = 0; i < z0.rows(); ++i)
            z0(i, k) = std::sqrt(2. / (N + 1)) * std::sin(math::PI * (i + 1) * (k
+ 1) / (N + 1));

```

```cpp
    // Compute 'H' from Hessenberg decomposition 'A = P H P^*'
    Matrix H_hessenberg = hessenberg_reduce(A);

    // Compute numeric eigenvalues
    const auto [ T_shur, lambda_iteration_counts ] = qr_algorithm(H_hessenberg);

    // Extract numeric eigenvalues as a sorted vector for comparison
    Vector lambda = T_shur.diagonal();
    std::sort(lambda.begin(), lambda.end());

    // Compute numeric eigenvecs
    Matrix                   z(N, N);
    std::vector<std::size_t> z_iteration_counts(N);
    for (Idx k = 0; k < z0.cols(); ++k) {
        const auto res        = reverse_iteration(A, lambda(k));
        lambda(k)             = std::get<0>(res);
        z.col(k)              = std::get<1>(res); // Eigen doesn't like '
std::tie()'
        z_iteration_counts[k] = std::get<2>(res);
    }

    log::println("---------------------------");
    log::println("--- Eigenvalue solution ---");
    log::println("---------------------------");
    log::println();
    log::println("H_hessenberg                  -> ",
stringify_matrix(H_hessenberg));
    log::println("T_shur                        -> ", stringify_matrix(T_shur));
    log::println("lambda0 (analythic eigenvals) -> ", stringify_matrix(lambda0));
    log::println("lambda    (numeric eigenvals) -> ", stringify_matrix(lambda));
    log::println("z0      (analythic eigenvecs) -> ", stringify_matrix(z0));
    log::println("z       (analythic eigenvecs) -> ", stringify_matrix(z));

    table::set_latex_mode(true); // generate tables in export format

    table::create({4, 25, 21, 18, 19});
    table::hline();
    table::cell(" j ", " |lambda_j^0 - lambda_j| ", "Reduction iterations", " ||
z0_j - z-j||_2 ", "Reverse iterations");
    table::hline();
    for (std::size_t j = 0; j < N; ++j) {
        table::cell(j + 1, std::abs(lambda0(j) - lambda(j)),
lambda_iteration_counts[j], (z0.col(j) - z.col(j)).norm(), z_iteration_counts[j])
;
    }
    table::hline();

    return 0;
}
```

5. Листинг проверочного скрипта на Wolfram Mathematica

In[1234]:=

```
Nvar = 1;
ε = 0.1;
c = Nvar / (Nvar + 1) ε;
N = 4;

A0 = Table[Piecewise[{{2, i == j}, {-1, (i == j - 1) || (i == j + 1)}}, 0], {i, 1, N}, {j, 1, N}];
δA = Table[Piecewise[{{c / (i + j), i ≠ j}}, 0], {i, 1, N}, {j, 1, N}];
A = A0 + δA;
Ahat = A[[All, 1 ;; -2]];

{Q, R} = QRDecomposition[Ahat];
Q = Transpose@Q; (* for some reason Mathematica returns Q as transposed *)

x0 = Table[i * i, {i, 1, N - 1}];
b = Ahat.x0;

LLSx = Inverse[R].Transpose[Q].b;
LLSx2 = LeastSquares[Ahat, b]; (* Should give the same result as formula above *)

eigenvals = Reverse@Eigenvalues[A];
{ShurQ, ShurT} = SchurDecomposition[A];
(* Should give the same eigenvalues as method above *)
(* Eigenvalues will be stored on the main diagonal of 'T' *)
{HessP, HessH} = HessenbergDecomposition[A];

Framed@"Problem"
Row@{"A₀ = ", A0 // MatrixForm}
Row@{"δA = ", δA // MatrixForm}
Row@{"A = ", A // MatrixForm}
Row@{"Â = ", Ahat // MatrixForm}
Framed@"QR decomposition"
Row@{"Q = ", Q // N // MatrixForm}
Row@{"R = ", R // N // MatrixForm}
Row@{"Q^TQ = ", Transpose[Q].Q // MatrixForm}
Row@{"QR = ", Q.R // MatrixForm}
Framed@"LLS"
Row@{"x₀ = ", x0 // MatrixForm}
Row@{"b = ", b // MatrixForm}
Row@{"x_LLS (formula) = ", LLSx // MatrixForm}
Row@{"x_LLS (built-in) = ", LLSx2 // MatrixForm}
Framed@"Eigenvalue problem"
Row@{"{λi}_{i=1}^N = ", eigenvals // MatrixForm}
```

6. Приложение. Пример сводки результатов расчетной программы

```
---------------
--- Problem ---
---------------

epsilon -> 1e-06
N       -> 4
A0      -> Dense matrix [size = 16] (4 x 4):
  [  2 -1  0  0 ]
  [ -1  2 -1  0 ]
  [  0 -1  2 -1 ]
  [  0  0 -1  2 ]

deltaA  -> Dense matrix [size = 16] (4 x 4):
  [          0 1.6667e-07   1.25e-07      1e-07 ]
  [ 1.6667e-07          0      1e-07 8.3333e-08 ]
  [   1.25e-07      1e-07          0 7.1429e-08 ]
  [      1e-07 8.3333e-08 7.1429e-08          0 ]

A       -> Dense matrix [size = 16] (4 x 4):
  [        2         -1 1.25e-07      1e-07 ]
  [       -1          2       -1 8.3333e-08 ]
  [ 1.25e-07         -1        2         -1 ]
  [    1e-07 8.3333e-08       -1          2 ]

A_hat   -> Dense matrix [size = 12] (4 x 3):
  [        2         -1 1.25e-07 ]
  [       -1          2       -1 ]
  [ 1.25e-07         -1        2 ]
  [    1e-07 8.3333e-08       -1 ]

------------------------
--- QR factorization ---
------------------------

Q             -> Dense matrix [size = 12] (4 x 3):
  [    -0.89443   -0.35857 -0.19518 ]
```

```
  [     0.44721   -0.71714 -0.39036 ]
  [ -5.5902e-08    0.59761 -0.58554 ]
  [ -4.4721e-08 -9.761e-08  0.68313 ]

R             -> Dense matrix [size = 9] (3 x 3):
  [    -2.2361      1.7889 -0.44721 ]
  [ 2.2204e-16     -1.6733   1.9124 ]
  [ -2.647e-23 -1.1102e-16  -1.4639 ]

Verification:

Q^T * Q       -> Dense matrix [size = 9] (3 x 3):
  [          1  2.7756e-16  8.3267e-17 ]
  [ 2.7756e-16           1 -2.2204e-16 ]
  [ 8.3267e-17 -2.2204e-16           1 ]

Q * R - A_hat -> Dense matrix [size = 12] (4 x 3):
  [  2.2204e-15 -1.7764e-15  6.8361e-16 ]
  [ -8.8818e-16 -8.8818e-16  1.3323e-15 ]
  [   1.327e-16  3.3307e-16 -4.4409e-16 ]
  [  3.9705e-23 -7.5843e-17  2.2204e-16 ]

--------------------------------------
--- Linear Least Squares solution ---
--------------------------------------

x0                -> Dense matrix [size = 3] (3 x 1):
  [ 1 ]
  [ 4 ]
  [ 9 ]

b                 -> Dense matrix [size = 4] (4 x 1):
  [ -2 ]
  [ -2 ]
  [ 14 ]
  [ -9 ]

x_lls             -> Dense matrix [size = 3] (3 x 1):
  [ 1 ]
  [ 4 ]
```

```
  [ 9 ]

lls_error_estimate -> 1.8496162997539822e-16

---------------------------
--- Eigenvalue solution ---
---------------------------

H_hessenberg                  -> Dense matrix [size = 16] (4 x 4):
  [          2          1 -2.647e-23 1.3235e-23 ]
  [          1          2         -1 -2.647e-23 ]
  [  2.647e-23         -1          2          1 ]
  [ 1.3235e-23 -2.647e-23          1          2 ]

T_shur                        -> Dense matrix [size = 16] (4 x 4):
  [       3.618 -5.9746e-16 -2.7092e-16 -3.8027e-16 ]
  [  2.5614e-27     0.38197  3.9167e-16   1.396e-16 ]
  [ -2.8355e-21  1.6472e-16       2.618  -1.558e-16 ]
  [  6.2793e-18   5.744e-18 -4.3103e-23       1.382 ]

lambda0 (analythic eigenvals) -> Dense matrix [size = 4] (4 x 1):
  [ 0.38197 ]
  [   1.382 ]
  [   2.618 ]
  [   3.618 ]

lambda    (numeric eigenvals) -> Dense matrix [size = 4] (4 x 1):
  [ 0.38197 ]
  [   1.382 ]
  [   2.618 ]
  [   3.618 ]

z0      (analythic eigenvecs) -> Dense matrix [size = 16] (4 x 4):
  [ 0.37175   0.6015   0.6015  0.37175 ]
  [  0.6015  0.37175 -0.37175  -0.6015 ]
  [  0.6015 -0.37175 -0.37175   0.6015 ]
  [ 0.37175  -0.6015   0.6015 -0.37175 ]

z       (analythic eigenvecs) -> Dense matrix [size = 16] (4 x 4):
  [ 0.37175   0.6015   0.6015  0.37175 ]
```

```
  [  0.6015  0.37175 -0.37175  -0.6015 ]
  [  0.6015 -0.37175 -0.37175   0.6015 ]
  [ 0.37175  -0.6015   0.6015 -0.37175 ]

\hline
  j  &  |lambda_j^0 - lambda_j|  &  Reduction
  iterations &  ||z0_j - z-j||_2  &  Reverse iterations \\
\hline
   $1$ &                         $2.9964 \cdot 10^{-7}$ &                      $17$ &
   $2$ &                         $8.6690 \cdot 10^{-8}$ &                      $1$ &
   $3$ &                         $9.9648 \cdot 10^{-8}$ &                      $20$ &
   $4$ &                         $1.1331 \cdot 10^{-7}$ &                      $33$ &
\hline
```