Лабораторная работа №2

> Солодуха Дмитрий, 13 группа

1. Генерацию матриц провожу аналогично прошлой лабораторной работе. Использую вот этот вот генератор:

   ```
   std::uniform_real_distribution<double>(a, b);
   ```

   Генерируемые матрицы являются полностью случайными, поэтому ожидать от них можно чего угодно (разбросы во времени работы QR-алгоритма и неправильные ответы в степенном методе).

2. Метод выглядел гораздо проще QR-алгоритма, но в реализации получилось наоборот. Основная проблема состоит в том, чтобы с наименьшими затратами различать следующие случаи:

   - Есть доминирующее собственное значение и $\lambda_d \in \mathbb{R}$
   - Есть два доминирующих собственных значения различных по знаку, то есть $\lambda_1 = -\lambda_2$, $\lambda_1, \lambda_2 \in \mathbb{R}$
   - Доминирующая комплексно-сопряженная пара
     Мне больше понравились методы, которые предложены в книге Репникова, потому что там эти случаи легче связать и способ решения последнего случая там легче.

   Если проанализировать алгоритм, то можно заметить, что основная сложность заключается в умножении на матрицу и именно её нам нужно минимизировать при попытках определить отдельный случай. Я делал следующим образом:

   - Сразу же делаю все необходимые умножения
   - Пробую найти собственные значения и вектора, начиная с самого простого случая
   - Если случай использует только часть векторов, то беру последние из них (все равно мы уже сделали итерации, поэтому логично брать последние)
   - Если на каком-то случае получаю подходящее значение — останавливаюсь, иначе — начинаю заново.

   Единственный минус такого подхода состоит в том, что я могу сразу же сделать лишние итерации, но не более 2.
   К сожалению у меня не сложилось с комплексными векторами в `C++` (мне было слишком лень переписывать кучу методов написанных под `double`. Всё-таки иногда лучше сразу писать в шаблонах), поэтому я не реализовал комплексный случай, но суть описанного процесса от этого не меняется.
   На матрицах без комплексных собственных значениях все работало хорошо (я генерировал вещественные собственные значения и преобразованием подобия создавал матрицу). А вот со случайными матрицами из условия не совсем сложилось. В итоге именно они сильно подпортили оцениваемую точность и время.
   Время получилось достаточно большим для такого простого метода, а норма — огромной: $t_{avg} = 62$, $\langle Ax - \lambda x \rangle = 4509$. Судить о чем-то по этим значениям нельзя, потому что они учитывают комплексные случаи, которые упирались в максимальное число итераций и портили время с нормой. Хотя в теории метод должен быть быстрее QR-алгоритма.
   При чем откинуть данные случаи на ранних этапах тоже сложно из-за того, что мы ничего не знаем о диагонализируемости матрицы.
   Кстати, нормы я писал в файл `norms.txt`, а не `report.txt`

3. Реализуется проще. Приходится сталкиваться с комплексными числами только на последнем этапе итерации, то есть при непосредственном вычислении собственных значений.
   Единственный не очевидный момент, с которым я столкнулся при реализации – определение окончание итераций, потому что как-то хорошо определить точность текущего приближения без знания собственных векторов не получится. Я же определял точность текущей итерации следующим образом:

   $$\epsilon = \sum_{i=1}^{n} |x_i - x_{i-1}| + |y_i - y_{i-1}|, \quad \lambda_i = x_i + iy_i$$

   Помимо этого нужно было как-то определять элементы матрицы, которые при подсчете собственных значений будут считаться нулями. Тут просто достаточно выбрать константу, чтобы числа по модулю меньше этой константы считать нулями. Я, кстати, попытался её эмпирически оптимизировать и получил, что быстрее всего алгоритм работает, когда мы считаем числа меньше 10^{-3} нулями.
   Если говорить о времени работы алгоритма, то тут получаем достаточно большой разброс $t_{min} = 7$, $t_{max} = 356$. То есть время работы достаточно сильно зависит от начальной матрицы, что и логично. Матрица может иметь удобный для итераций вид, содержать поменьше комплексных значений, которые вычисляются достаточно долго. Но все равно они не полностью отображают суть происходящего, поэтому нужно принимать во внимание среднее время $t_{avg} = 129$. Видно, что QR-алгоритм в сравнении с методами из предыдущей лабораторной работает достаточно долго, но тут и задача гораздо сложнее.

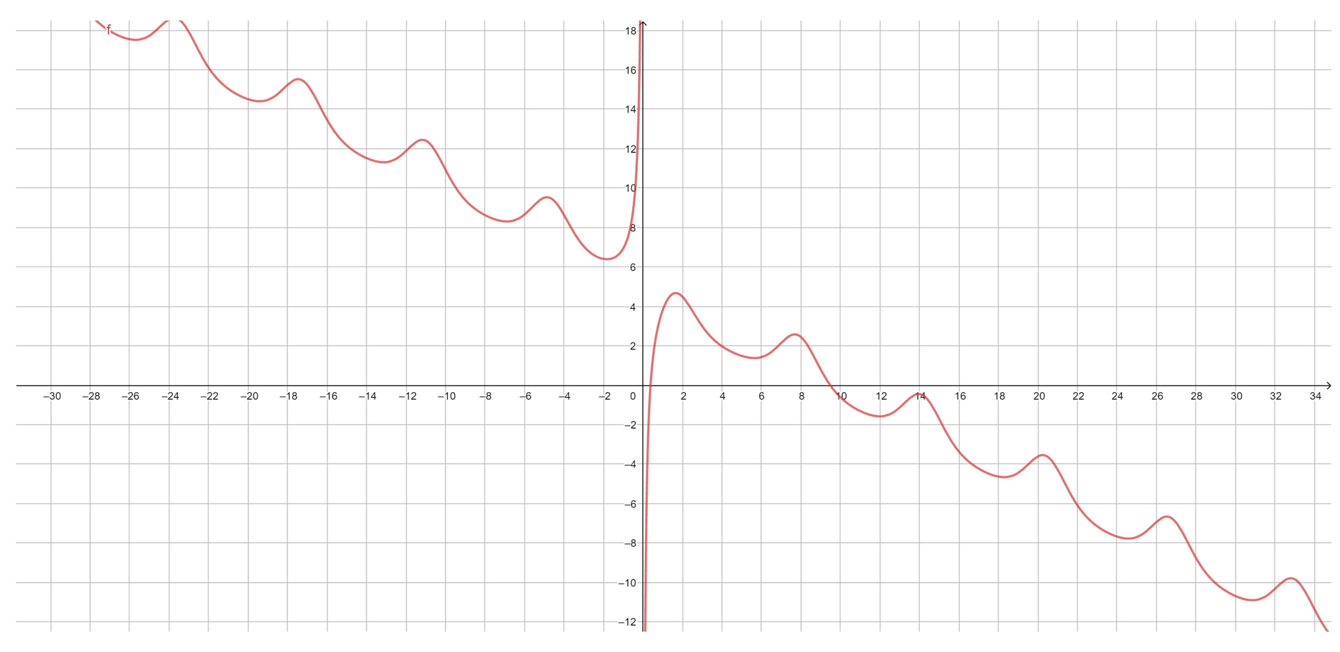
4. Моя функция:

   $$f(x) = e^{\sin x} - \frac{x^2 - 8x + 4}{2x}$$

   Производная для неё:

   $$f'(x) = \cos x \, e^{\sin x} + \frac{2}{x^2} - \frac{1}{2}$$

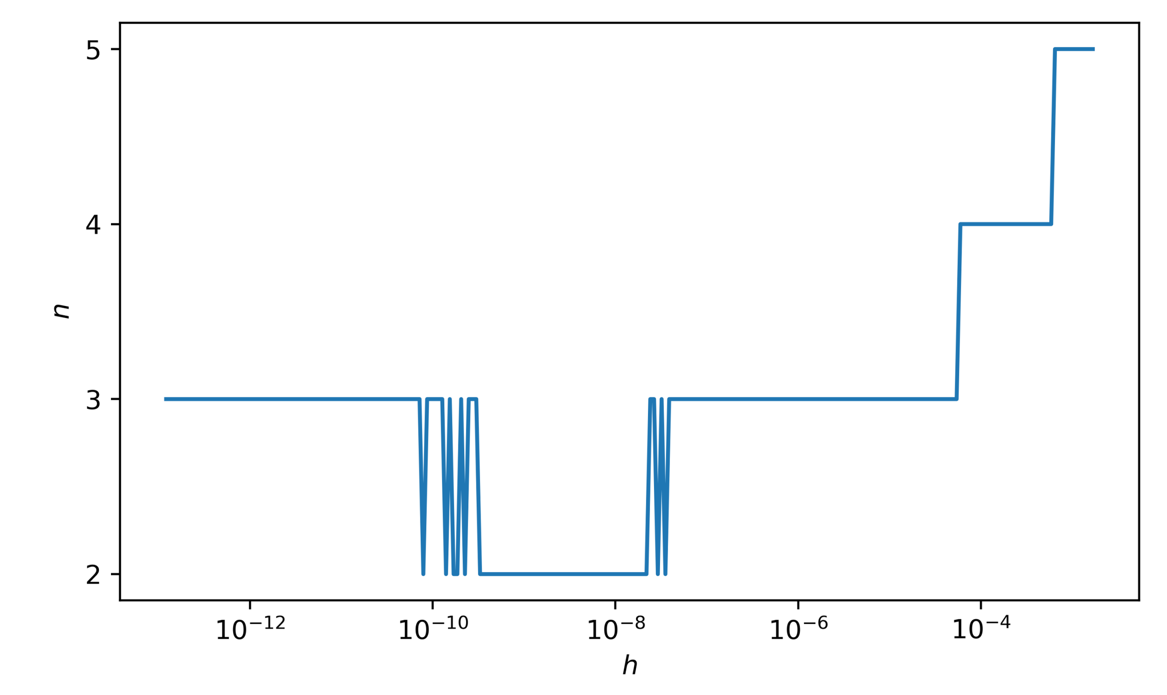
5. Сразу видно, что $e^{\sin x}$ не сильно будет влиять на поведение нашей функции в предельных случаях. Основная вклад в ее поведение будет вносить второе слагаемое: $\frac{x^2 - 8x + 4}{2x}$. На бесконечности функция будет вести себя как линейная с небольшими колебаниями, а вблизи нуля как гипербола, то есть каких-то непонятных нулей не будет.

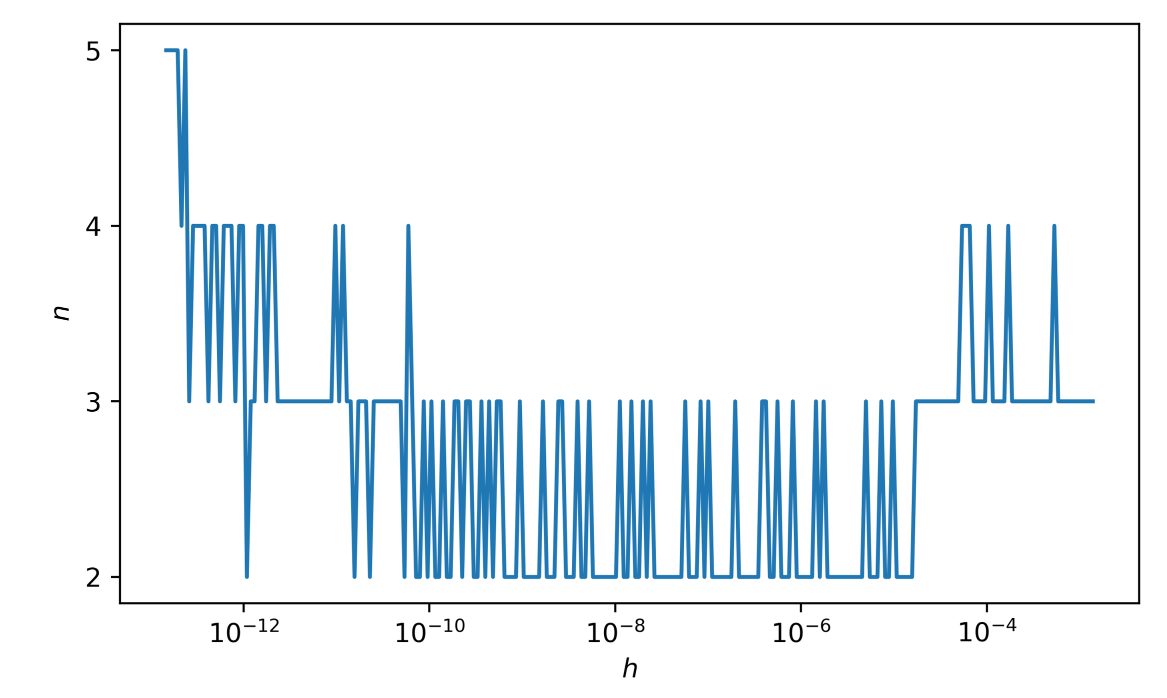
   

   Получаем два следующих промежутка отделенности корней: $x_1 \in (0, 2]$ и $x_2 \in [8, 10]$. Такие отрезки и беру для начального приближения в методе бисекций.

> Далее конкретные значения корней не указываю. Их можно посмотреть здесь: `reports/report.txt`

5. Реализация метода очень простая и не особа интересна. Число итераций можно посчитать заранее $n = \lceil \log_2\left(\frac{b-a}{\epsilon}\right) \rceil$ Мои начальные отрезки имеют одинаковую длину, поэтому оба до необходимой длины сужаются за одинаковое количество итераций $n = 15$. Так как отрезки отделенности корней выбраны правильно, то и алгоритм работает корректно. Количество итераций, конечно, оставляет желать лучшего, но и ожидать от линейной сходимости большего нельзя.
6. Дискретный метод Ньютона тоже реализуется очень просто. Можно даже передавать в обычный метод Ньютона дискретное приближение производной. Работает достаточно быстро, но квадратичную сходимость не гарантирует, хотя при моём случайно выбранном значении шага почти догнал обычный метод Ньютона.
   Точность брал предельную для типа `double`. Полученное число итераций: $n_1 = 5, n_2 = 3$.
   Сразу же видно, что шаг выбирается не очевидным образом и скорее всего зависит от поведения производной в области корня. Предположу, что выбор оптимального шага напрямую зависит от модуля второй производной вблизи корней.
   Так как число итераций очень мало, построение диаграммы сходимости почти не имеет смысла. Для нахождения оптимального шага просто буду сравнивать число шагов для достижения необходимой точности. Полученные графики для первого и второго корня:

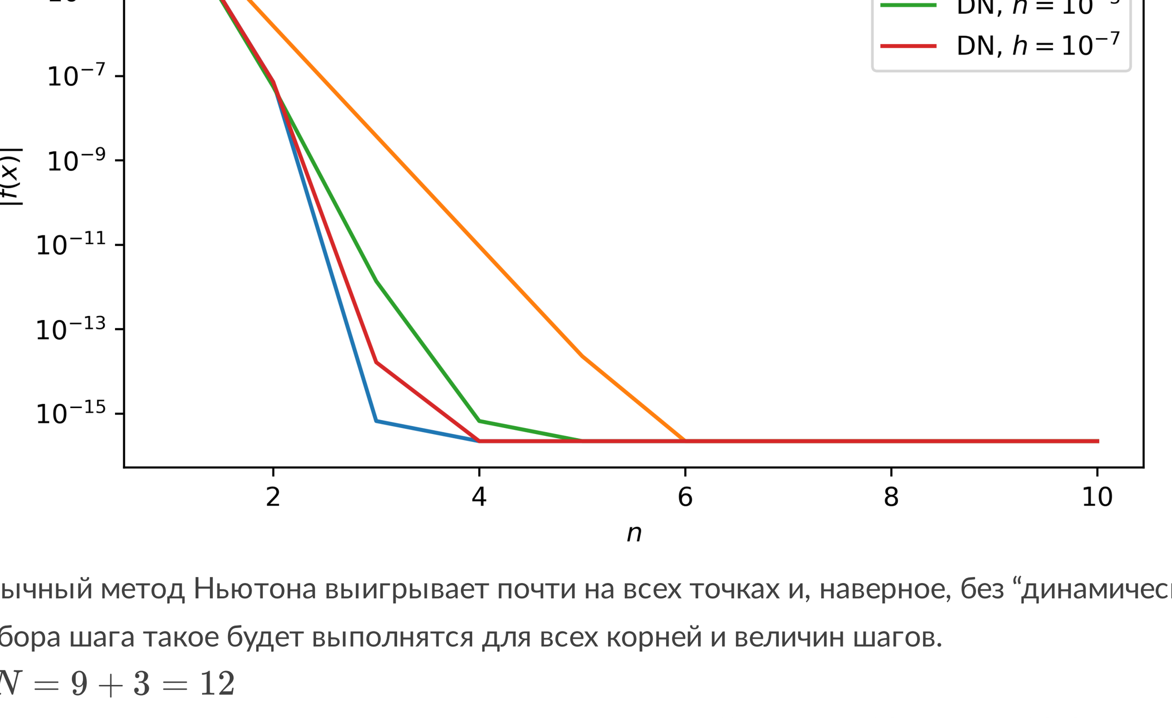
   

   Для второго корня не так красиво получилось, конечно. Из графиков видно, что оптимальное значение шага выбирается сложнее чем просто “ну чем меньше, тем лучше”. Также, как и предполагалось, на некоторых значениях шага дискретный догоняет обычный вариант. Лучше всего для этих корней было брать значение $h = 10^{-8}$.
7. Опять же реализуется просто. Вопрос остается только с выбором оценки для точности текущего приближения. Я выбрал для этого величину текущей невязки $|f(x_i)|$, потому что так писать меньше, а для моих корней еще и почти оптимально получается, так производная в их окрестностях далека от нуля. Сходится, как и ожидалось, быстрее дискретного варианта: $n_1 = 2, n_2 = 2$.
   Проверим условие сходимости Фурье (надеюсь, что вычисления тут писать не надо было). Для второго корня все получается хорошо и условие не выполнять, а для первого — нет. Поэтому проверял следующее условие:

   $$m_1 = \inf_{x \in U} |f'(x)|, \; M_2 = \sup_{x \in U} |f''(x)|$$

   $$\frac{M_2 |x^0 - x^*|}{2m_1} < 1$$

   Данное условие выполняется и для первого корня, следовательно методы сходятся.
   В диаграмме сходимости смысла не вижу, но в условии просилось:

   Обычный метод Ньютона выигрывает почти на всех точках и, наверное, без “динамического” выбора шага такое будет выполнятся для всех корней и величин шагов.
   8. $N = 9 + 3 = 12$
   Результат QR-алгоритма для данной матрицы: $t_{min} = 5, t_{max} = 18, t_{avg} = 12.3$.
   Собственные значения: `(36.00000000000001,3.999999999999972), (36.00000000000001,-3.999999999999972), (24.00000000000002,0), (14.99999999999995,0), (11.99999999999997,0), (-12.00000000000006,0), (-10.99999999999996,0), (10.00000000000045,0), (0.9999999999997269,0), (-0.9999999999999178,0)`
   Со степенным методом очевидные проблемы получаются, так как я не написал на `C++` комплексный случай. Код на `py`, который я писал для ДЗ считает максимальное собственное значение корректно.