МИНИСТЕРСТВО ЦИФРОВОГО РАЗВИТИЯ, СВЯЗИ И МАССОВЫХ КОММУНИКАЦИЙ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ТЕЛЕКОММУНИКАЦИЙ ИМ. ПРОФ. М.А. БОНЧ-БРУЕВИЧА»
(СПбГУТ)

Лабораторная работа №8
по курсу
«Логическое и функциональное программирование»

Выполнил:
студент группы ИКПИ-11
Дунаев В.Е.

Принял:
доцент кафедры ПИиВТ
Ерофеев С.А.

Санкт-Петербург
2024 г.

Цель работы

Разработать программу на языке Haskell, построения сплайна заданного пользователем порядка (от 1 до 10).

Описание алгоритма

Сплайн - гладкая функция, область определения которой разбита на конечное число отрезков, на каждом из которых она совпадает с некоторым многочленом (полиномом) заданной степени.

Сначала пользователь вводит порядок сплайна (степень полинома), шаг построения итогового графика, кол-во начальных точек, а потом и сами точки. Все эти данные проверяются на корректность, в случае ошибки в терминал выведется сообщение с объяснением.

Если все правильно, то приступаем к составлению уравнений для сплайнов. Кол-во уравнений зависит от кол-ва неизвестных. Например у нас есть 3 отрезка с полиномами 7 порядка. Умножив 3 на 7+1 (кол-во неизвестных коэффициентов в одном полиноме) получаем 24 неизвестных, т. е. 24 условия.

Первый полином определен на первой и второй точке — это 2 условия. Второй полином определен на второй и третьей точках — еще 2 условия. Третий полином, четвертый, пятый, и т.д. — каждый из них определен на 2-х точках. Суммарно в нашем случае выходит 6 условий

Для кажной промежуточной точки из множества введенных точек должно выполнятся условие, что все производные ниже 7 для левого и правого полиномов должны совпадать : S’1(x = 1) = S`2(x = 1) S``1(x = 1) = S``2(x = 1) и т. д. В нашем случае это еще 2 * 6 = 12 условий.

Еще есть «граничные условия», от из задания зависит, какой именно сплайн получится. Обычно задают вторые производные на концах интервала равными 0. Если так сделать, то мы получим «естественный сплайн». Плюс 2 условия в копилку.

Для того чтобы однозначно задать полином на этом интервале, нам не хватает еще 4 условий. Но мы можем из просто придумать. Например найдем все производные до 7 степени в начальной точке. Т.е. S```(x0) = 0; S````(x0) = 0; S`````(x0) = 0; S``````(x0) = 0

В сумме у нас вышло 24 условия, а это значит что мы сможем найти этот сплайн, например методом Гаусса решения СЛАУ.

Все найденные условия записываем в качестве коэффициентов системы уравнения и решаем методом Гаусса. В итоге мы получаем все коэффициенты описывающие сплайны на каждом заданном отрезке.

В конце мы выводим с заданным пользователем шагом точки результирующего графика в .csv файл.

Используемые функции

formatList - вывод точек в файл .csv

checkDuplicates - проверяет список точек на дубликаты

insertPoints - ввод количества точек

points - ввод точек

point - ввод одной точки

order - ввод порядка сплайна

precision - ввод шага для отрисовки итогового графика

interpolation - возвращает список точек получившегося полинома с заданным шагом

segment - возвращает список точек получившегося полинома на отрезке [x0:x1]

ans - решает полином от x, выводит y

main - решает полином от x, выводит y

polinom - Коэфициенты полинома

polinomDerivative - Коэфициенты производных полинома

extendCoef - коэффициенты полинома превращает в коэффициенты итогового уравнения

derivativeEquals - приравнивает коэффициенты двух производных

derivatives - находит все необходимые производные в точке, кроме краевых точек

additionalEquations - добирает дополнительные уравнения в начальной точке так, чтобы их число равнялось числу неизвестных

equations - возвращает систему уравнений для дальнейшего решения

gauss - метод Гауса решения слау

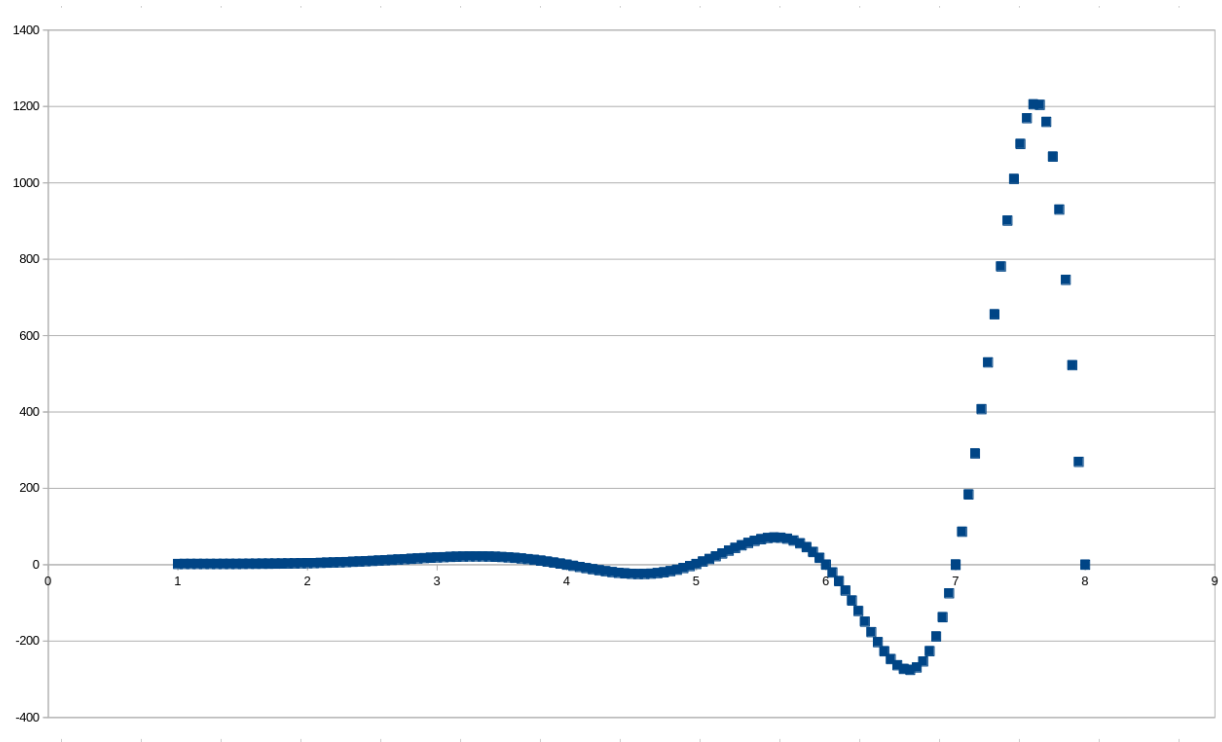
substitute - решает матрицу (должна уже быть в треугольной форме) для метода Гаусса обратной заменой

solve - находит решение системы уравнений методом Гауса

Тестирование

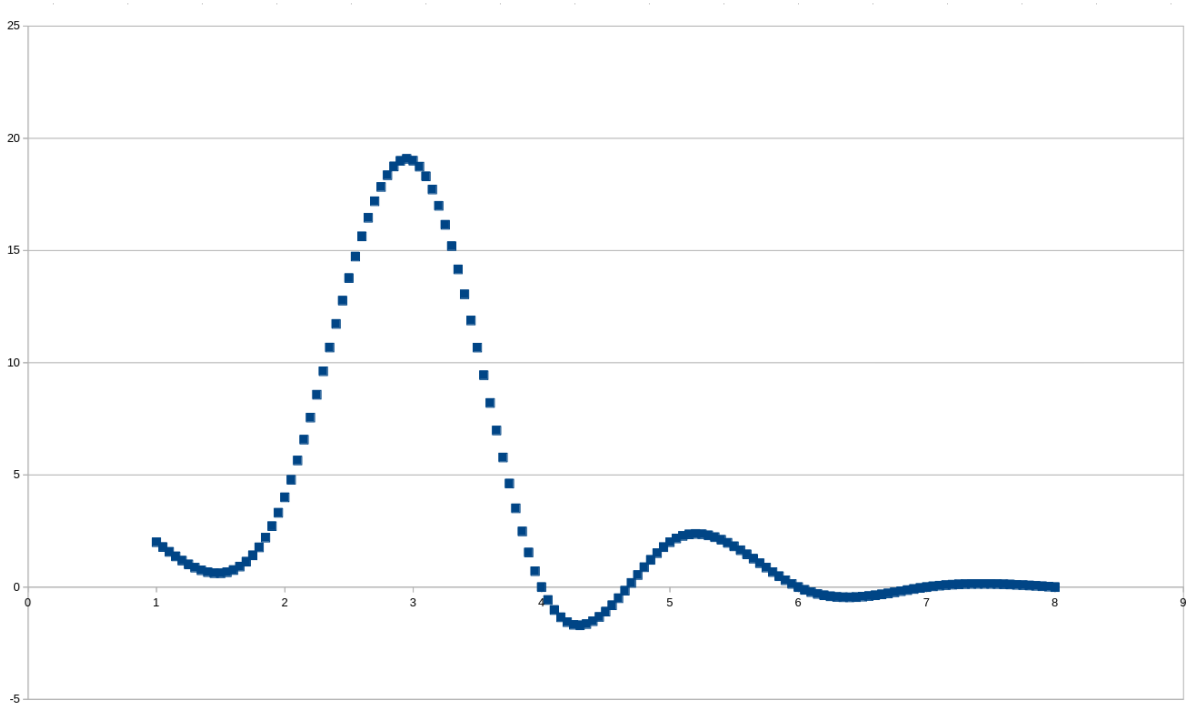
1) Порядок 10

Точки : [(1.0,2.0),(2.0,4.0),(3.0,19.0),(4.0,0.0),(5.0,2.0),(6.0,0.0),(7.0,0.0),(8.0,0.0)]



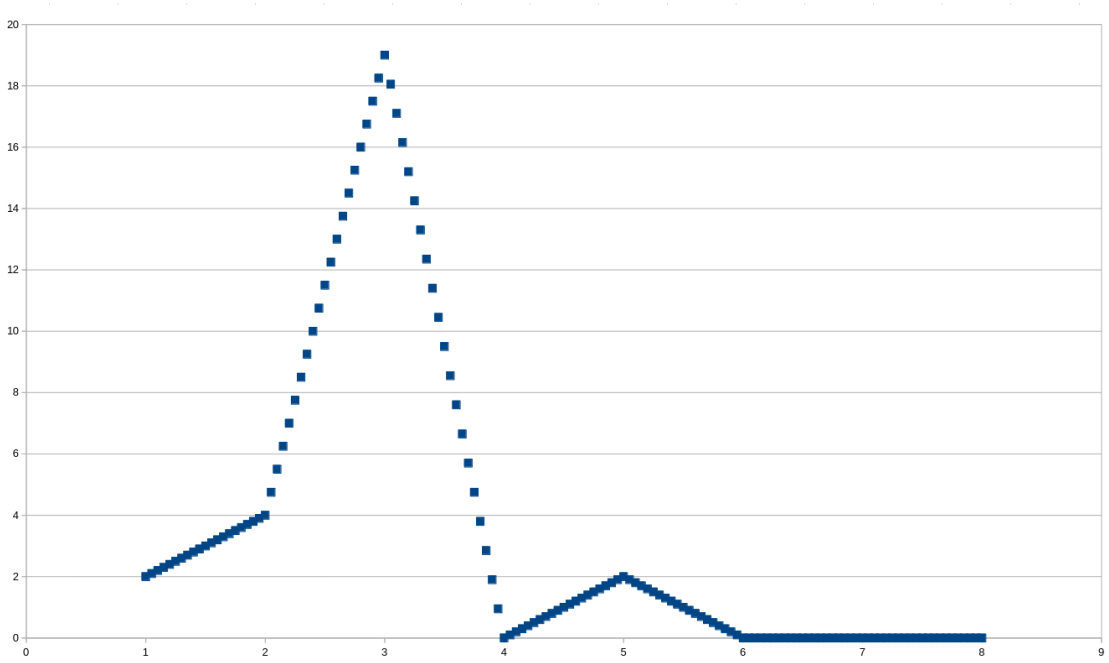
2) Порядок 3

Точки : [(1.0,2.0),(2.0,4.0),(3.0,19.0),(4.0,0.0),(5.0,2.0),(6.0,0.0),(7.0,0.0),(8.0,0.0)]



3) Порядок 1

Точки : [(1.0,2.0),(2.0,4.0),(3.0,19.0),(4.0,0.0),(5.0,2.0),(6.0,0.0),(7.0,0.0),(8.0,0.0)]



Выводы

В ходе проведенной лабораторной работы я разработал программу на языке Haskell для построения сплайна заданного пользователем порядка. Было проведено тестирование отдельных функций и всей программы в целом. Программа полностью удовлетворяет заданным требованиям.

Исходный код программы

18.hs:

```
import Data.List

-- вывод точек в файл .csv
formatList [] = ""
formatList ((x, y):xs) = show x ++ ";" ++ show y ++ "\n" ++ formatList xs

-- проверяет список точек на дубликаты
checkDuplicates xs = length (nubBy (\(x0, y0) (x1, y1) -> x0 == x1 ) xs) /= length xs

-- ввод количества точек
insertPoints = do
    putStrLn "!!! Ввод опорных точек !!! "
    putStr " Введите кол-во точек : "
    cnt <- readLn :: IO Int
    if cnt < 2 then
        error "Неправильное число точек"
    else do
        result <- points cnt cnt
        putStr " Введенные точки : "
        print $ sort $ result
        return $ (sort $ result, cnt)

-- ввод точек
points len 0 = do return []
points len cnt  | cnt < 0   = error " Кол-во не может быть отрицательным"
                | otherwise = do
    putStr " Точка "
    print (len - cnt + 1)
    (x,y) <- point
    temp <- points len (cnt - 1)
    if null temp then
        return $ [(x,y)]
    else
        return $ [(x,y)] ++ temp

-- ввод одной точки
point = do
    putStr "  X = "
    x <- readLn :: IO Double
    putStr "  Y = "
    y <- readLn :: IO Double
    return (x, y)

-- ввод порядка сплайна
order = do
    putStrLn "!!! Порядок сплайна !!!"
    putStr " Введите число от 1 до 10 : "
    someNum <- readLn :: IO Int
    if someNum > 0 && someNum < 11 then
        return someNum
    else
        error "Неправильно введен порядок сплайна"

-- ввод шага для отрисовки итогового графика
precision = do
    putStrLn "!!! Шаг итогового графика !!!"
    putStr " Введите положительное число : "
    someNum <- readLn :: IO Double
    if someNum > 0 then
        return someNum
    else
        error "Шаг должет быть положительным"


-- возвращает список точек получившегося полинома с заданным шагом
interpolation equation precision list order | null equation = []
                                            | otherwise = let
    koef = take (order + 1) equation
    temp = segment x0 x1 koef order precision x0
    newEquation = drop (order + 1) equation
    newList = tail list
    in temp ++ interpolation newEquation precision newList order
    where   (x0,y0) = list !! 0
            (x1,y1) = list !! 1

-- возвращает список точек получившегося полинома на отрезке [x0:x1]
segment x0 x1 koef order precision i | i > x1 = []
                                     | otherwise =
    [(i,y)] ++ segment x0 x1 koef order precision (i + precision)
    where   y = ans x0 i koef order

-- решает полином от x, выводит y
ans start x koef 0 = head koef
ans start x koef order = term + ans start x tl (order - 1)
    where   term = ((x - start) ^ order) * head koef
            tl = tail koef

-- точка входа в программу
main = do
    putStrLn "  --- Построение сплайна ---"
    order <- order
    precision <- precision
    (list, cnt) <- insertPoints
    if not $ checkDuplicates list then do
        let i = cnt
        let equation = equations list (cnt - 1) order 1
        let answer = solve equation
        let temp = interpolation answer precision list order
        writeFile "output.csv" $ formatList temp
        putStrLn "Результат записан в файл output.csv"
```

```
    else
        error " Повторяющиеся точки недопустимы, одному значению x соответствует одно значение y"


-- Коэфициенты полинома
polinom 1 start (x, y) = [(x - start), 1, y]
polinom order start (x, y) = [(x - start) ^ order] ++ polinom (order - 1) start (x, y)

-- Коэфициенты производных полинома
polinomDerivative 1 order start x = drop (10 - order) koef
    where koef =    [10 * ((x - start) ^ 9)] ++ [9 * ((x - start) ^ 8)] ++ [8 * ((x - start) ^ 7)] ++
                    [7 * ((x - start) ^ 6)] ++ [6 * ((x - start) ^ 5)] ++ [5 * ((x - start) ^ 4)] ++
                    [4 * ((x - start) ^ 3)] ++ [3 * ((x - start) ^ 2)] ++ [2 * (x - start)] ++
                    [1, 0, 0]

polinomDerivative 2 order start x = drop (10 - order) koef
    where koef =    [90 * ((x - start) ^ 8)] ++ [72 * ((x - start) ^ 7)] ++ [56 * ((x - start) ^ 6)] ++
                    [42 * ((x - start) ^ 5)] ++ [30 * ((x - start) ^ 4)] ++ [20 * ((x - start) ^ 3)] ++
                    [12 * ((x - start) ^ 2)] ++ [6 * (x - start)] ++ [2, 0, 0, 0]

polinomDerivative 3 order start x = drop (10 - order) koef
    where koef =    [720 * ((x - start) ^ 7)] ++ [504 * ((x - start) ^ 6)] ++ [336 * ((x - start) ^ 5)] ++
                    [210 * ((x - start) ^ 4)] ++ [120 * ((x - start) ^ 3)] ++ [60 * ((x - start) ^ 2)] ++
                    [24 * (x - start)] ++ [6, 0, 0, 0, 0]

polinomDerivative 4 order start x = drop (10 - order) koef
    where koef =    [5040 * ((x - start) ^ 6)] ++ [3024 * ((x - start) ^ 5)] ++ [1680 * ((x - start) ^ 4)] ++
                    [840 * ((x - start) ^ 3)] ++ [360 * ((x - start) ^ 2)] ++ [120 * (x - start)] ++
                    [24, 0, 0, 0, 0, 0]

polinomDerivative 5 order start x = drop (10 - order) koef
    where koef =    [10080 * ((x - start) ^ 5)] ++ [7560 * ((x - start) ^ 4)] ++ [5040 * ((x - start) ^ 3)] ++
                    [2520 * ((x - start) ^ 2)] ++ [720 * (x - start)] ++ [120, 0, 0, 0, 0, 0, 0]

polinomDerivative 6 order start x = drop (10 - order) koef
    where koef =    [45360 * ((x - start) ^ 4)] ++ [30240 * ((x - start) ^ 3)] ++ [15120 * ((x - start) ^ 2)] ++
                    [5040 * (x - start)] ++ [720, 0, 0, 0, 0, 0, 0, 0]

polinomDerivative 7 order start x = drop (10 - order) koef
    where koef =    [241920 * ((x - start) ^ 3)] ++ [120960 * ((x - start) ^ 2)] ++ [40320 * (x - start)] ++
                    [5040, 0, 0, 0, 0, 0, 0, 0, 0]

polinomDerivative 8 order start x = drop (10 - order) koef
    where koef =    [1209600 * ((x - start) ^ 2)] ++ [362880 * (x - start)] ++
                    [40320, 0, 0, 0, 0, 0, 0, 0, 0, 0]

polinomDerivative 9 order start x = drop (10 - order) koef
    where koef =    [3628800 * (x - start) ] ++ [362880, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

-- коэффициенты полинома превращает в коэффициенты итогового уравнения
extendCoef list interval order i = let
    a1 = take (((fromIntegral order) + 1) * i) (repeat 0.0)
    a2 = take (((fromIntegral order) + 1) * (interval - i - 1)) (repeat 0.0)
    in  a1 ++ init list ++ a2 ++ [last list]

-- приравнивает коэффициенты двух производных
derivativeEquals = zipWith (\a b -> a - b)


-- находит все необходимые производные в точке, кроме краевых точек
derivatives x0 x1 order i cnt iter  | iter == order = []
                                    | otherwise = let
    pol0 = extendCoef (polinomDerivative iter order x0 x1) cnt order (i - 1)
    pol1 = extendCoef (polinomDerivative iter order x1 x1) cnt order i
    in [derivativeEquals pol0 pol1] ++ derivatives x0 x1 order i cnt (iter + 1)

-- добирает дополнительные уравнения в начальной точке так, чтобы их число равнялось числу неизвестных
additionalEquations betterOrder order cnt start_x   | order < 4 = []
                                                    | order > 10 = error "че?"
                                                    | otherwise = let
    temp = extendCoef (polinomDerivative (order - 1) betterOrder start_x start_x) cnt betterOrder 0
    in [temp] ++ additionalEquations betterOrder (order - 1) cnt start_x

-- возвращает систему уравнений для дальнейшего решения
equations list cnt order i  | cnt == i =
    -- Если прошли все центральные точки, то не забываем о крайних
    -- если порядок = 1, то добавляем просто начальные условия
    if order == 1 then
        nach
    -- если порядок = 2, то добавляем начальные условия + первая производная в крайней точке
    else if order == 2 then
        nach ++
        [extendCoef (polinomDerivative 1 order start_x start_x) cnt order 0]

    -- если порядок >= 3, то добавляем начальные условия + вторые производные в крайних точках +
    -- + дополнительные производные в первой точке, для добора общего числа уравений
    else
        nach ++ [extendCoef (polinomDerivative 2 order x0 x1) cnt order (i - 1)] ++
        [extendCoef (polinomDerivative 2 order start_x start_x) cnt order 0] ++
        additionalEquations order order cnt start_x

    -- Проходимся по точкам, добавляем уравнения, производные и т.д
                            | otherwise = let
    pol0 = extendCoef (polinom order x0 (x0, y0)) cnt order (i - 1)
    pol1 = extendCoef (polinom order x0 (x1, y1)) cnt order (i - 1)
    derivativess = derivatives x0 x1 order i cnt 1
    in  [pol0] ++ [pol1]++
        derivativess ++
        equations list cnt order (i + 1)
    where
        nach =  [extendCoef (polinom order x0 (x0, y0)) cnt order (i - 1)] ++
                [extendCoef (polinom order x0 (x1, y1)) cnt order (i - 1)]
        (x0, y0) = list !! (i - 1)
        (x1, y1) = list !! i
        (start_x, start_y) = list !! 0


-- Метод Гауса решения слау
gauss matrix = fixlastrow $ foldl reduceRow matrix [0..length matrix-1] where
    -- swap меняет местами строки
    swap xs a b
        | a > b = swap xs b a
        | a == b = xs
        | a < b = let
        (p1,p2) = splitAt a xs
        (p3,p4) = splitAt (b-a-1) (tail p2)
        in p1 ++ [xs!!b] ++ p3 ++ [xs!!a] ++ (tail p4)

    reduceRow matrix1 r = let
        -- ищем первый не нулевой элемен на или ниже элемента (r, r)
        firstnonzero = head $ filter (\x -> matrix1 !! x !! r /= 0) [r..length matrix1-1]

        -- матрица с замененными рядами (если это требуется)
        matrix2 = swap matrix1 r firstnonzero

        -- ряд с которым сейчас работаем
```

```
        row = matrix2 !! r

        --make it have 1 as the leading coefficient
        -- делаем ведущий коэффициент равным 1
        row1 = map (\x -> x / (row !! r)) row

        --subtract nr from row1 while multiplying
        -- вычитаем nr из row1 во время умножения
        subrow nr = let k = nr!!r in zipWith (\a b -> k*a - b) row1 nr

        --apply subrow to all rows below
        -- применяем подстроку ко всем строкам ниже
        nextrows = map subrow $ drop (r+1) matrix2

        -- проверка: если найдется столбец в котором все элементы равны нулю, то система не решаема
        in  if (not $ null $ filter (\x -> matrix1 !! x !! r /= 0) [r..length matrix1-1]) then
                -- конкатинируем список и продолжаем
                take r matrix2 ++ [row1] ++ nextrows
            else
                error "\n Проверьте правильность вводных данных\n Система уравнений не решаемая"

    fixlastrow matrix' = let
        a = init matrix'; row = last matrix'; z = last row; nz = last (init row)
        in a ++ [init (init row) ++ [1, z / nz]]

-- решает матрицу (должна уже быть в треугольной форме) для метода гаусса обратной заменой
substitute matrix = foldr next [last (last matrix)] (init matrix) where
    next row found = let
        subpart = init $ drop (length matrix - length found) row
        solution = last row - sum (zipWith (*) found subpart)
        in solution : found

-- находит решение системы уравнений методом Гауса
solve = substitute . gauss
```