Министерство науки и высшего образования Российской Федерации
Национальный научно-исследовательский университет ИТМО
Факультет программной инженерии и компьютерной техники

Лабораторная работа №3

по дисциплине

«Функциональное программирование».

Линейная интерполяция функции.

Работу выполнил:
Афанасьев Кирилл Александрович,
Студент группы P3306.
Преподаватели:
Новоселов Борис Сергеевич,
Пенской Александр Сергеевич.

Санкт-Петербург, 2024

Оглавление

Задание

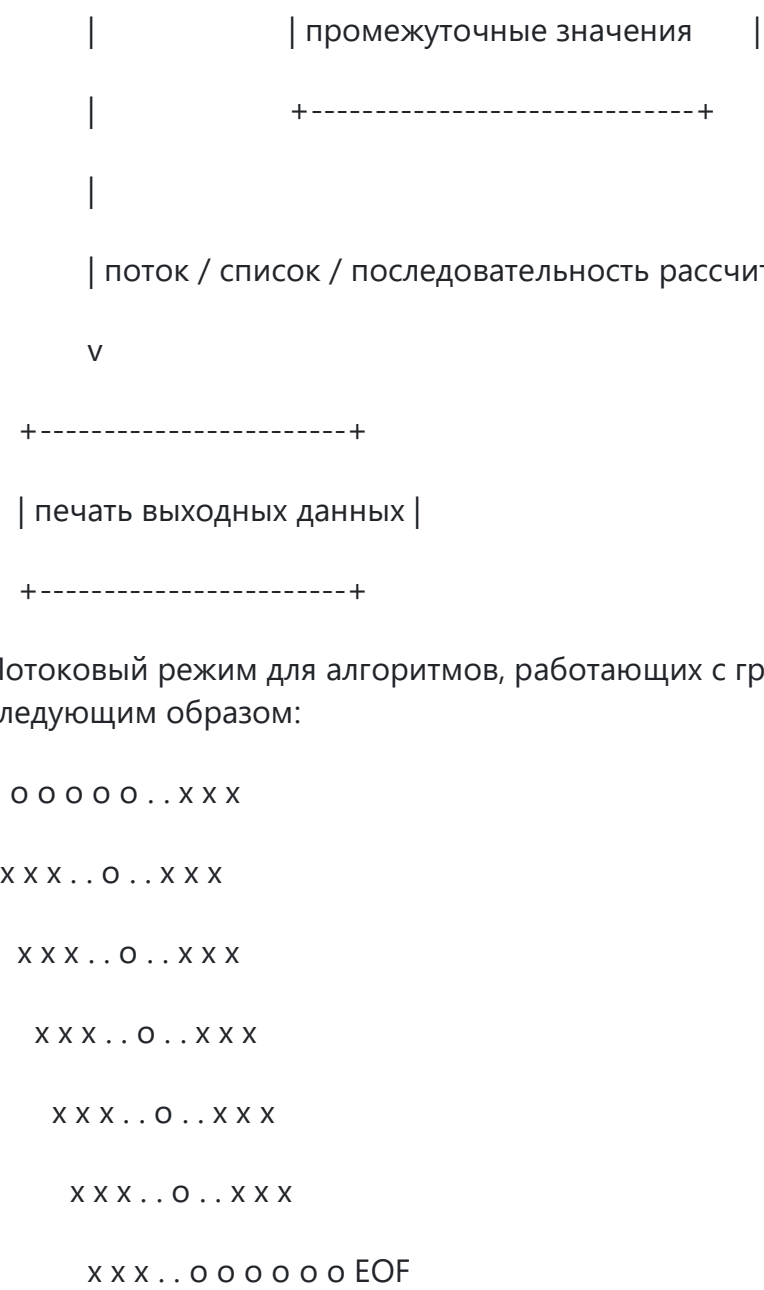
Цель: получить навыки работы с вводом/выводом, потоковой обработкой данных, командной строкой.

В рамках лабораторной работы вам предлагается повторно реализовать лабораторную работу по предмету "Вычислительная математика" посвящённую интерполяции (в разные годы это лабораторная работа 3 или 4) со следующими дополнениями:

- обязательно должна быть реализована линейная интерполяция (отрезками, link);
- настройки алгоритма интерполяции и выводимых данных должны задаваться через аргументы командной строки:
  - какие алгоритмы использовать (в том числе два сразу);
  - частота дискретизации результирующих данных;
  - и т.п.;
- входные данные должны задаваться в текстовом формате на подобии ".csv" (к примеру x;y\n или x\ty\n) и подаваться на стандартный ввод, входные данные должны быть отсортированы по возрастанию x;
- выходные данные должны подаваться на стандартный вывод;
- программа должна работать в потоковом режиме (пример -- cat | grep 11), это значит, что при запуске программы она должна ожидать получения данных на стандартный ввод, и, по мере получения достаточного количества данных, должна выводить рассчитанные точки в стандартный вывод;

Приложение должно быть организовано следующим образом:

```
+--------------------------+

| обработка входного потока |

+--------------------------+

        |

        | поток / список / последовательность точек

        v

+----------------------+      +-----------------------------+

| алгоритм интерполяции |<-----| генератор точек, для которых |

+----------------------+      | необходимо вычислить         |
```



Потоковый режим для алгоритмов, работающих с группой точек должен работать следующим образом:

o o o o o o . . x x x

x x x . . o . . x x x

x x x . . o . . x x x

x x x . . o . . x x x

x x x . . o . . x x x

x x x . . o . . x x x

x x x . . o o o o o o EOF

где:

- каждая строка -- окно данных, на основании которых производится расчёт алгоритма;
- строки сменяются по мере поступления в систему новых данных (старые данные удаляются из окна, новые -- добавляются);
- o -- рассчитанные данные, можно видеть:
  - большинство окон используется для расчёта всего одной точки, так как именно в "центре" результат наиболее точен;
  - первое и последнее окно используются для расчёта большого количества точек, так лучших данных для расчёта у нас не будет.
- . -- точки, задействованные в рассчете значения o.
- x -- точки, расчёт которых для "окон" не требуется.

Пример вычислений для шага 1.0 и функции sin(x):

Ввод первых двух точек (в данном примере X Y через пробел):

0 0.00

1.571 1

Вывод:

Линейная (идем от первой точки из введенных (0.00) с шагом 1, покрывая все введенные X (1.571 < 2)):

0.00 1.00 2.00

0.00 0.64 1.27

Ввод третьей точки:

3.142 0

Следующий вывод:

Линейная (идем от второй точки из введенных (1.571) с шагом 1, покрывая все введенные X (3.142 < 3.57)):

1.57 2.57 3.57

1.00 0.36 -0.27

Ввод четвертой точки:

4.712 -1

Следующий вывод:

Линейная (идем от третьей точки из введенных (3.142) с шагом 1, покрывая все введенные X (4.712 < 5.14)):

3.14 4.14 5.14

0.00 -0.64 -1.27

Лагранж (теперь количество введенных точек повзоляет его рассчитать, идем от первой точки (0.00) из введенных с шагом 1, покрывая все введенные X (4.712 < 5)):

0.00 1.00 2.00 3.00 4.00 5.00

0.00 0.97 0.84 0.12 -0.67 -1.03

Ввод пятой точки:

12.568 0

Следующий вывод:

Линейная (идем от четвертой точки из введенных (4.712) с шагом 1, покрывая все введенные X (12.568 < 12.71))):

4.71 5.71 6.71 7.71 8.71 9.71 10.71 11.71 12.71

-1.00 -0.87 -0.75 -0.62 -0.49 -0.36 -0.24 -0.11 0.02

Лагранж (идем от второй точки из введенных (1.571) с шагом 1, покрывая все введенные X (12.568 < 12.57))):

1.57 2.57 3.57 4.57 5.57 6.57 7.57 8.57 9.57 10.57 11.57 12.57

1.00 0.37 -0.28 -0.91 -1.49 -1.95 -2.26 -2.38 -2.25 -1.84 -1.11 0.00

И т.д.

Как видно из примера выше, окна для каждого метода двигаются по-разному. Для линейной окно начало сдвигаться уже при вводе третьей точки (т.к. для вычисления нужно всего две), в то время как для Лагранжа окно начало двигаться только когда была введена пятая точка (т.к. здесь для вычислений нужно больше точек).

Общие требования:

- программа должна быть реализована в функциональном стиле;
- ввод/вывод должен быть отделён от алгоритмов интерполяции;
- требуется использовать идиоматичный для технологии стиль программирования.

Содержание отчёта:

- титульный лист;
- требования к разработанному ПО, включая описание алгоритма;
- ключевые элементы реализации с минимальными комментариями;
- ввод/вывод программы;
- выводы (отзыв об использованных приёмах программирования).

Общие рекомендации по реализации. Не стоит писать большие и страшные автоматы, управляющие поведением приложения в целом. Если у вас:

- Язык с ленью -- используйте лень.
- Языки с параллельным программированием и акторами -- используйте их.
- Язык без всей этой прелести -- используйте генераторы/итераторы/и т.п.

Исходный код программы

GitHub: https://github.com/Zerumi-ITMO-Related/fp3_251024_linear-interpolation
Ключевые элементы реализации:
Линейная интерполяция:

```
linearInterpolationFunc :: Double -> SlidingWindow Point -> Maybe [Point]
linearInterpolationFunc stepSize sw =
  case getElements sw of
    [Just (x0, y0), Just (x1, y1)] -> Just [(x, y0 + (x - x0) * (y1 - y0) / (x1 - x0)) | x <- generateSequence x1 stepSize (x0
+ stepSize)]
    _ -> Nothing
```

Лагранж 4 порядка:

```
lagrangeInterpolationFunc :: Double -> SlidingWindow Point -> Maybe [Point]
lagrangeInterpolationFunc stepSize sw =
  case getElements sw of
    [Just (x0, y0), Just (x1, y1), Just (x2, y2), Just (x3, y3)] ->
      Just
        [ ( x,
            y0 * ((x - x1) * (x - x2) * (x - x3)) / ((x0 - x1) * (x0 - x2) * (x0 - x3))
              + y1 * ((x - x0) * (x - x2) * (x - x3)) / ((x1 - x0) * (x1 - x2) * (x1 - x3))
              + y2 * ((x - x0) * (x - x1) * (x - x3)) / ((x2 - x0) * (x2 - x1) * (x2 - x3))
              + y3 * ((x - x0) * (x - x1) * (x - x2)) / ((x3 - x0) * (x3 - x1) * (x3 - x2))
          )
          | x <- generateSequence x3 stepSize (x0 + stepSize)
        ]
    _ -> Nothing
```

REPL:

```
-- | Read a line from the user
read_ :: IO String
read_ =
  putStr "Enter point (x, y)> "
    >> hFlush stdout
    >> getLine


-- | Validate the input
validate_ :: String -> Bool
validate_ str = case words str of
  [x, y] -> isJust (readMaybe x :: Maybe Double) && isJust (readMaybe y :: Maybe Double)
  _ -> False


validateGrowingX :: String -> Interpolation -> Bool
validateGrowingX input (Interpolation _ window _ _) = case words input of
```

```haskell
  [xStr, _] -> case head (getElements window) of
    Just (x0, _) -> x0 < read xStr
    _ -> True
  _ -> False

-- | Evaluate the input
eval_ :: String -> Interpolation -> (Interpolation, Maybe String)
eval_ = evalPoint

-- | Print the result
print_ :: Maybe String -> IO ()
print_ Nothing = return ()
print_ (Just str) = putStrLn str

-- | Loop the read-eval-print cycle
loop_ :: [Interpolation] -> IO ()
loop_ windows = do
  input <- read_
  unless (input == ":quit") $
    if validate_ input && validateGrowingX input (head windows)
      then do
        let (newWindows, results) = unzip $ map (eval_ input) windows
        mapM_ print_ results
        loop_ newWindows
      else do
        putStrLn "Invalid input. Please enter a valid point (x, y). X should be greater than the previous X."
        loop_ windows
```

Вывод

Во время выполнения данной лабораторной работы я ознакомился с обработкой ввода-вывода в Haskell, а также применил возможности ленивого программирования для реализации нескольких алгоритмов интерполяции с возможностью ввода данных от пользователя.