МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ «НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ЯДЕРНЫЙ УНИВЕРСИТЕТ «МИФИ» (НИЯУ МИФИ)
КАФЕДРА ИНФОРМАЦИОННЫХ СИСТЕМ И ТЕХНОЛОГИЙ

КУРСОВАЯ РАБОТА

по дисциплине «Алгоритмы и структуры данных»
на тему «Расширение языка стекового калькулятора операцией возведения в степень, которая является правоассоциативной и имеющей максимальный приоритет. Компилирование формул, содержащих данную операцию. Вычисление значений выражений, содержащих операцию возведения в степень, которая считается левоассоциативной и имеющей минимальный приоритет. Вычисление расстояния от заданной прямой до выпуклой оболочки. Вычисление максимального радиуса круга, целиком содержащегося внутри выпуклой оболочки. Вычисление суммы длин рёбер полиэдра, середина и оба из концов которых — «хорошие» точки. »

Группа	К04-361
Студент	Е.А. Пономарёв
Руководитель работы к.ф.-м.н., доцент	Е.А. Роганов

Москва 2016

Аннотация

Работа посвящена модификации проектов «Компилятор формул», «Интерпретатор арифметических выражений» и«Выпуклая оболочка». В первом из этих проектов решалась задача расширения грамматики языка стекового компилятора определённой операцией, а также компиляции формул, содержащих эту операцию. Суть модификации второго проекта заключалась в вычислении значения выражений, содержащих операцию возведения в степень, которая считается левоассоциативной и имеющей минимальный приоритет. В проекте «Выпуклая оболочка» вычислялось расстояние от заданной прямой до выпуклой оболочки и максимальный радиус круга, содержащегося внутри выпуклой оболочки. В последнем из проектов вычилялась сумма рёбер, середина и оба из концов которых — «хорошие» точки, т.е. точки, проекция которых находится строго вне квадрата единичной площади с центром в начале координат и сторонами, параллельными координатным осям.

Содержание

1. Введение . . . 3
2. Модификация проекта «Компилятор формул» . . . 3
3. Модификация проекта «Интерпретатор арифметических выражений» . 5
4. Модификация проекта «Выпуклая оболочка» . . . 7
5. Модификация проекта «Изображение проекции полиэдра» . . . 15
6. Приложение к пункту 4.2 . . . 20

1. Введение

В проектах «Компилятор формул» и «Интерпретатор арифметических выражений» необходимо было расширить входную грамматику стекового компилятора операцией возведения в степень с указанными приоритетом и ассоциативностью, а также компилировать полученные формулы и вычислять соответствующие численные выражения. Реализация указанной модификации требует представления о формальных языках и грамматиках, об индуктивных функциях и простейших контейнерах данных. Так как в рамках учебного курса «Алгоритмы и структуры данных» проект должен быть реализован на языке Ruby, необходимы также базовые знания данного языка, как и знание основ объектно-ориентированного программирования в целом [1].

Задачей проекта «Выпуклая оболочка»[2] является индуктивное перевычисление выпуклой оболочки последовательно поступающих точек плоскости и таких её характеристик, как периметр и площадь. Целью данной работы является определение расстояния от заданной прямой до выпуклой оболочки; вычисление радиуса максимального круга с центром в заданной точке, содержащегося в выпуклой оболочке. Решение этой задачи требует знания теории индуктивных функций, основ аналитической геометрии и векторной алгебры, а также языка Ruby [1]. Для наглядного изображения реализации задачи необходимы навыки работы с библиотекой графического интерфейса Tk [3].

Проект «Изображение проекции полиэдра» [4] — пример классической задачи, для успешного решения которой необходимо знакомство с основами вычислительной геометрии. Задачей, решаемой в данной работе, является модификация эталонного проекта с целью определения суммы длин рёбер, середина и оба из концов которых — точки, проекция которых находится строго вне квадрата единичной площади с центром в начале координат и сторонами, параллельными координатным осям. Для этого необходимы хорошее понимание ряда разделов аналитической геометрии и векторной алгебры, основ объектно-ориентированного программирования и языка Ruby.

2. Модификация проекта «Компилятор формул»

Постановка задачи

В проекте «Компилятор формул» была поставлена задача модифицировать эталонный проект следующим образом: «В предположении, что язык стекового калькулятора расширен правоассоциативной и имеющей максимальный приоритет операцией ^ возведения в степень, компилировать формулы, содержащие эту операцию».

С формальной точки зрения компилятор представляет собой программную реализацию некоторой функции: $\tau\colon L_1 \to L_2$, действующей из множества цепочек одного языка L_1 в множество цепочек другого L_2 таким образом, что $\forall \omega \in L_1$ семантика цепочек ω и $\tau(\omega) \in L_2$ совпадает. Нужный нам компилятор представляет собой программную реализацию отображения из множества цепочек языка $L(G_0)$ в множество цепочек языка $L(G_S)$ По этой причине его можно рассматривать, как функцию на пространстве последовательностей [5].

Рассмотрим в качестве входного языка $L(G_0)$ язык правильных арифметических формул, дополненный операцией возведения в степень, являющейся правоассоциативной и имеющей максимальный приоритет. При этом грамматика G_0 вышеуказанного языка будет задаваться следующей НФБН:

$$\begin{array}{rlll} F \to & T & \mid F+T \mid & F-T \\ T \to & M & \mid T*M \mid & T/M \\ M \to & F & \mid M \mathbin{\hat{}} F \mid & V \\ V \to & a & \mid b \mid c \mid ... \mid & z \end{array}$$

Выходным языком будем считать язык $L(G_S)$ стекового калькулятора, грамматика G_S которого также расширена указанной операцией и задаётся такой НФБН:

$$e \to \; ee+ \;\mid ee- \;\mid ee* \;\mid ee/ \mid ee \;\hat{}\; \;\mid a \mid b \mid ... \mid z$$

Теоретические аспекты

Стековый компилятор τ представляет собой программную реализацию отображения из множества цепочек языка $L(G_0)$ в множество цепочек языка $L(G_S)$. По этой причине его можно рассматривать, как функцию на пространстве последовательностей. Данная функция не является индуктивной, так как результатом компиляции двух произвольных цепочек $w_1 = a-b$ и $w_1 = (a-b)$ является одна и та же цепочка $a\ b\ -$. Если же дописать к обеим цепочкам $x = *c$, результаты компиляции соответствующих цепочек буду выглядеть следующим образом: $a\ b\ c\ *\ -$ и $a\ b\ -\ c\ *$. А это значит, что $\tau(w_1 \circ x) \neq \tau(w_2 \circ x)$.

Поэтому возникает необходимость построить индуктивное расширение T функции τ для реализации однопроходного алгоритма. Очевидно, что рекурсивный компилятор формул не подходит для решения данной задачи, а значит, необходимо применить другой алгоритм, основанный на использовании тривиального стека. Для его реализации необходимо определить ряд условий:

- левая скобка (`SYM_LEFT`) не предшествует ничему;
- правая скобка (`SYM_RIGHT`) предшествует всему;
- одна арифметическая операция предшествует другой, если её приоритет не ниже, чем у второй;
- если арифметическая операция является операцией возведения в степень — её приоритет выше, чем у какой-либо другой, и операции справа от неё выполняются согласно правилам правой ассоциативности.

Для автоматической обработки конца формулы можно просто взять в скобки всю исходную формулу.

Описание используемых структур и применяемых алгоритмов

Для решения задачи прежде всего необходимо реализовать тривиальный стек на базе вектора, чтобы размещать в нём отложенные операции. В языке Ruby массив (экземпляр класса Array) имеет все необходимые для этого методы, что только упрощает написание программного кода. Также введём некоторые константы, задающие тип символа:

- SYM_LEFT = 0 — «(»
- SYM_RIGHT = 1 — «)»
- SYM_OPER = 2 — «+», «-», «*», «/»
- SYM_OTHER = 3 — добавленная операция ^

Модификация эталонного проекта заключается в повышении приоритета добавленной операции (возведения в степень) по отношению к другим операциям. Для этого в классе Compf стекового компилятора, выведенного из уже реализованного класса Stack, был изменён метод priority:

```
def priority(c)
  return 3 if (c =='^')

  (c == '+' or c == '-') ? 1 : 2
end
```

Далее требуется изменить ассоциативность арифметической операции. Для этого необходимо в методе precedes?, определяющем отношение предшествования, заменить >= на >:

```
def precedes?(a, b)
  return false if sym_type(a) == SYM_LEFT
  return true  if sym_type(b) == SYM_RIGHT
  if (priority(a)==3 && priority(b)==3)
  priority(a) > priority(b)
  else
  priority(a) >= priority(b)
  end
end
```

Многие значительно более сложные задачи на модификацию также сводятся к минимальным изменениям в тексте программы и не требует изменения структуры всей реализации в целом [5].

Ниже представлен снимок экрана, демонстрирующий работу программы: compf.rb:

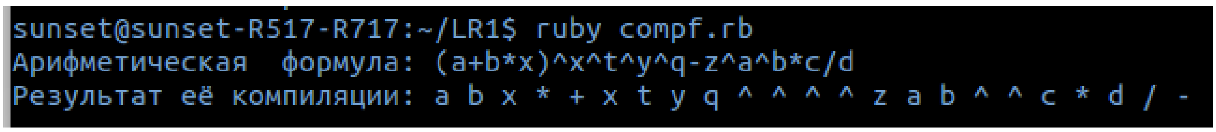

Рис. 1. Вывод программы

3. Модификация проекта «Интерпретатор арифметических выражений»

Постановка задачи

В данном проекте задача была поставлена следующим образом: «Вычисляются значения выражений, содержащих операцию возведения в степень, обозначаемую символом ^, которая считается левоассоциативной и имеет минимальный приоритет».

Прежде всего, необходимо учесть, что в отличии от компилятора формул интерпретатор выражнений подразумевает наличие во входной формуле чисел (вместо идентификаторов переменных), а также необходимость выполнения получаемой выходной формулы (результата выражения) вместо её печати. По сути, к действиям, посредством которых был выполнен предыдущий проект, необходимо добавить класс `Calc`, также выведенный из класса `Stack`, и переопределить метод `compile`, который будет вычислять значение входного арифметического выражения после работы соответствующего метода класса `Compf`.

Рассмотрим в качестве входного языка $L(G_0)$ язык правильных арифметических формул, дополненный операцией возведения в степень, являющейся левоассоциативной и имеющей минимальный приоритет. При этом грамматика G_0 вышеуказанного языка будет задаваться следующей НФБН:

$$\begin{array}{rlll} F \to & T & \mid F+T \mid & F-T \\ T \to & M & \mid T*M \mid & T/M \\ M \to & F & \mid M \mathbin{\hat{}} F \mid & V \\ V \to & 0 & \mid 1 \mid 2 \mid ... \mid 9 & \end{array}$$

Грамматика G_S выходного языка $L(G_S)$ задаётся такой НФБН:

$$e \to \; ee \mid 0 \mid 1 \mid 2 \mid 3 \mid ... \mid 9$$

Теоретические аспекты

Класс `Calc` эталонного проекта был создан таким образом, чтобы добавлять новые числа в стек, а при появлении арифметической операции извлекать два числа из стека. Далее, после выполнения требуемого действия результат необходимо класть обратно в стек. При таком подходе, после завершения компиляции формулы, на вершине стекового калькулятора окажется результат вычисления арифметического выражения.

Описание используемых структур и применяемых алгоритмов

В данном случае суть модификации состояла в добавлении нового метода `involution`, определяющего операцию. Так как дополнительными условиями данной задачи являются минимальный приоритет и левоассоциативность операции, необходимо для правильной обработки случая, когда нам встречается данная операция, переопределить метод `process_oper`:

```
def involution(a,n)
  result =1
  if n>1
    n.times do
      result*=a
    end
```

```
    elsif n==1
      result=a
    elsif n==0
      result=1
    end
    return result
  end

  def process_oper(c)
      if c =='^'
        second,first = @s.pop,@s.pop
        @s.push(involution(first,second))
      else
      second, first = @s.pop, @s.pop
      @s.push(first.method(c).call(second))
      end
  end
```

Ниже проиллюстрирован результат работы программы:

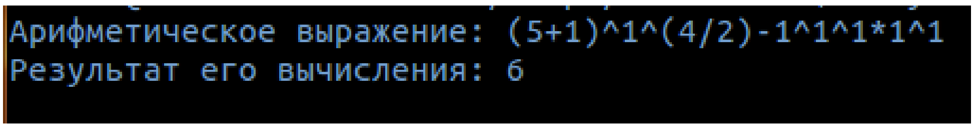

Рис. 2. Вывод программы

4. Модификация проекта «Выпуклая оболочка»

4.1 Постановка задачи

В следующем проекте решалась задача вычисления расстояния от заданной прямой до выпуклой оболочки. Кроме того, необходимо определять выпуклую оболочку и вычислять её характеристики сразу после поступления очередной точки. Для наглядного отображения работы программы необходимо графически иллюстрировать каждый этап, используя графический интерфейс Tk.

4.1 Теоретические аспекты

Пусть X — множество $\mathbb{R}^2$ точек плоскости, а P — совокупность всех выпуклых фигур на плоскости. Тогда тройка функций $f\colon X^* \to \mathcal{P}$ — выпуклая оболочка последовательности точек, $g\colon X^* \to \mathbb{R}$ — её периметр и $h\colon X^* \to \mathbb{R}$ — её площадь, задаёт индуктивную функцию $F\colon X^* \to \mathcal{P} \times \mathbb{R} \times \mathbb{R}$, $F = \begin{pmatrix} f \\ g \\ h \end{pmatrix}$.

Пусть для некоторой последовательности точек плоскости выпуклая оболочка, а также её периметр и площадь, уже известны. Тогда после добавления новой точки $x \in X$ возможны две ситуации: либо точка x попадает внутрь оболочки, либо вне её.

Для получения новой оболочки необходимо, как это хорошо видно из следующего рисунка, удалить все освещённые рёбра, а концы оставшейся ломаной соединить двумя новыми рёбрами с добавляемой точкой x:

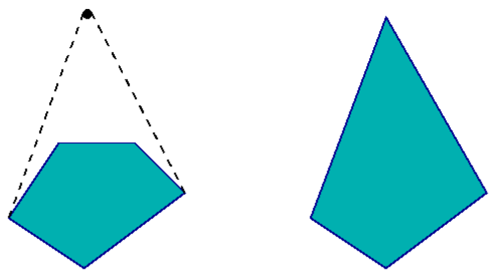

Рис. 3. Удаление освещённых рёбер

Если добавляемая точка лежит на продолжении одного из рёбер, то оболочка должна измениться, поэтому ребро, на продолжении которого лежит точка мы также будем считать освещённым:

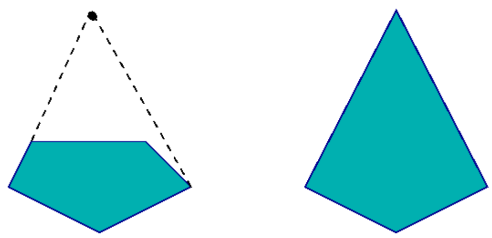

Рис. 4. Удаление освещённых рёбер

Для работы с точками на плоскости создаётся класс `R2Point`. В нём необходим ряд методов, которые позволяли бы вычислять `расстояния` между точками, сравнивать их на `совпадение`, выяснять, лежат ли три точки на одной прямой, находится ли некоторая точка на прямой `между` двумя другими, пересекаются ли отрезки, а также вычислять расстояние от заданной прямой до выпуклой оболочки, сравнивая `расстояние до отдельных рёбер`. Кроме того, необходимо уметь вычислять `площадь треугольника` и находить все `освещённые` ребра многоугольника.

Для вычисления площади треугольника в данном случае разумнее всего будет воспользоваться векторным произведением $S = \frac{1}{2}|\vec{a} \times \vec{b}|$. Тогда сократится количество необходимых вычислительных операций и результат будет более точным, по сравнению с аналогичными способами вычисления.

Для векторов $\vec{a} = (a_x, a_y, a_z)$ и $\vec{b} = (b_x, b_y, b_z)$ координаты вектора $\vec{c}$ находятся по формуле

$$\vec{c} = \begin{vmatrix} \vec{i} & \vec{j} & \vec{k} \\ a_x & a_y & a_z \\ b_x & b_y & b_z \end{vmatrix} = = (a_y b_z - a_z b_y)\vec{i} + (a_z b_x - a_x b_z)\vec{j} + (a_x b_y - a_y b_x)\vec{k}.$$

Однако в том случае, когда векторы $\vec{a}$ и $\vec{b}$ расположены на плоскости, их векторное произведение имеет единственную отличную от нуля компоненту, вычисление которой и должно быть реализовано в методе `area`. Также важно то, что вычисляемая площадь является `ориентированной`, то есть может быть отрицательной.

Для вычисления расстояния от точки до прямой используется метод `dist_to_line`. В его основу положена хорошо известная формула:

$$d = \frac{|A \times M_x + B \times M_y + C|}{\sqrt{A^2 + B^2}}$$

. Коэффициенты `A`, `B` и `C` мы получаем из общего вида уравнения прямой, проходящей через две точки: $(y_1 - y_2)\,x + (x_2 - x_1)\,y + (x_1 y_2 - x_2 y_1) = 0$.

Ниже приведена реализация метода `dist_to_line`:

```
def dist_to_line(m,n)
    a = (n.y-m.y)
    b = (m.x-n.x)
    c = m.x*(m.y-n.y)+m.y*(n.x-m.x)
    ((a*@x+b*@y+c).abs/Math.sqrt(a*a+b*b))
end
```

4.1 Описание используемых структур и применяемых алгоритмов

Определение того факта, лежат ли три точки на одной прямой (это делает метод `triangle?`), сводится к вычислению площади треугольника и сравнению её с нулём; а для того, чтобы точка T прямой находилась на отрезке $[A, B]$ этой же прямой (проверяет справедливость этого факта метод экземпляра `inside?`), необходимо и достаточно принадлежности проекций этой точки проекциям на оси координат отрезка $[A, B]$.

Для того чтобы реализовать метод `light?`, позволяющий выяснить, освещено ли ребро $[A, B]$ выпуклой оболочки из данной точки, дадим строгое определение освещённости. Ребро [A,B] называется освещённым из точки T, если ориентированная площадь треугольника, образованного точками A, B и T отрицательна, либо если точка T расположена на прямой, проходящей через точки A и B вне отрезка [A,B].

Искомое расстояние от заданной прямой до выпуклой оболочки будет отлично от нуля в том случае, если прямая не пересекает ни одно ребро данного многоугольника. Метод `intersect?` будет вычислять взаимное расположение отрезков, заданных точками класса `R2Point`, при помощи векторных произведений.

Пусть даны два отрезка. Первый задан точками $P_1(x_1; y_1)$ и $P_2(x_2; y_2)$. Второй задан точками $P_3(x_3; y_3)$ и $P_4(x_4; y_4)$.

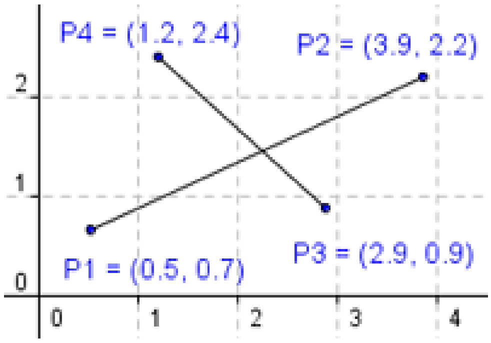

Рис. 5. Взаимное расположение отрезков

Тогда их взаимное расположение проверяется с помощью векторных произведений (каждое из них реализуется методом `cross`):

$$v_1 = |\vec{P_3P_4} \times \vec{P_3P_1}|,\ v_2 = |\vec{P_3P_4} \times \vec{P_3P_2}|$$
$$v_3 = |\vec{P_1P_2} \times \vec{P_1P_3}|,\ v_4 = |\vec{P_1P_2} \times \vec{P_1P_4}|.$$

Рассмотрим отрезок P_3P_4 и точки P_1 и P_2.

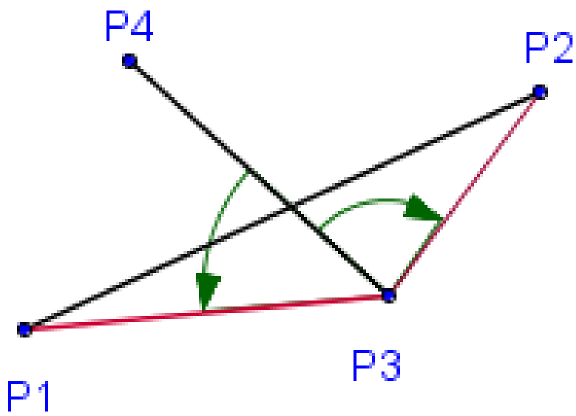

Рис. 6.

Точка P_1 лежит слева от прямой P_3P_4. Для неё векторное произведение $v_1 = |\vec{P_3P_4} \times \vec{P_3P_1}| > 0$, так как векторы положительно ориентированы. Точка P_2 расположена справа от прямой P_3P_4. Для неё векторное произведение $v_2 = |\vec{P_3P_4} \times \vec{P_3P_2}| < 0$, так как векторы отрицательно ориентированы.

Для того чтобы точки P_1 и P_2 лежали по разные стороны от прямой P_3P_4, достаточно, чтобы выполнялось условие $v_1v_2 < 0$ (векторные произведения имели противоположные знаки).

Аналогичные рассуждения можно провести для отрезка P_1P_2 и точек P_3 и P_4.

Итак, если $v_2 = |\vec{P_3P_4} \times \vec{P_3P_2}| < 0$, то отрезки пересекаются.

Ниже представлена реализация указанных методов:

```
def cross(w)
  @x*w.y-w.x*@y
end

def R2Point.intersect?(a,b,c,d)
  h = R2Point.new((d.x-c.x),(d.y-c.y))
  j = R2Point.new((a.x-c.x),(a.y-c.y))
  k = R2Point.new((b.x-c.x),(b.y-c.y))
  return true if h.cross(j)*h.cross(k) < 0
  false
end
```

Далее необходимо определить контейнер для хранения точек выпуклой оболочки. Наиболее подходящим в данном случае является дек, так как его можно свернуть в кольцо и считать «текущим» то ребро оболочки, которое соединяет начало и конец дека. Дальнейшие операции проводятся применительно к этому ребру:

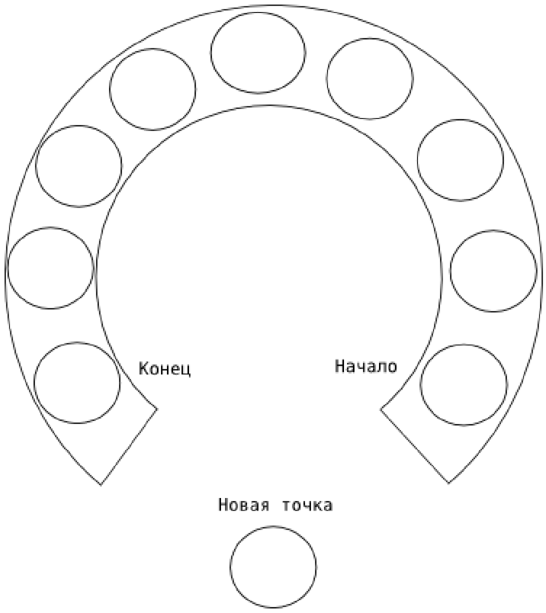

Рис. 7. Дек, содержащий точки

Для того чтобы удалить все освещенные ребра необходимо, «повернуть» дек таким образом, чтобы концы одного из освещённых рёбер находились в конце и в начале дека соответственно (если только освещённые рёбра вообще существуют). После этого нужно удалить найденное ребро.

Проектирование основных классов выполнено следующим образом: создаётся базовый класс `Figure` (фигура), и из него выводятся необходимые нам классы: `Void` (нульугольник), `Point` (одноугольник), `Segment` (двуугольник) и `Polygon` (многоугольник).

Каждый из последних четырёх классов должен реализовывать методы `add` (добавить новую точку), `perimeter` (получить периметр выпуклой оболочки) и `area` (получить площадь выпуклой оболочки). При этом объект типа `Point` должен со-

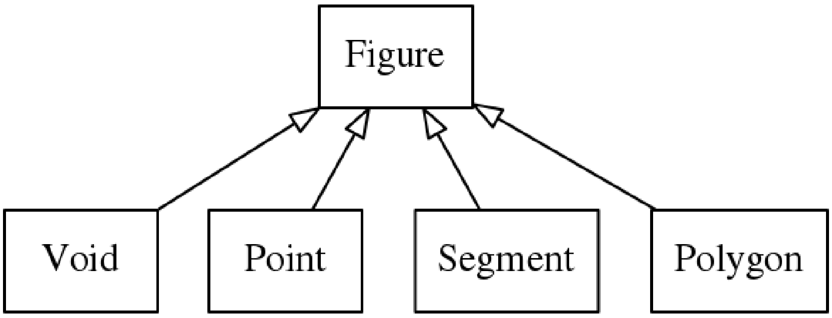

Рис. 8. Иерархия классов

держать внутри себя одну точку (компоненту типа `R2Point`), объект типа `Segment` — две, а типа `Polygon` — целый дек точек.

Вернемся к цели данной модификации, а именно — вычислению расстояния от заданной прямой до выпуклой оболочки. Прежде всего, мы будем хранить 2 точки, задающие прямую, как экземпляры класса `Figure`: @@apoint, @@bpoint. Нужный параметр @@d(расстояние) следует переопределять для каждой фигуры, начиная с одноугольника. Ниже приведен листинг, демонстрирующий модификацию конкретных классов программы. Для `Segment`:

```
def initialize(p, q)
  @p, @q = p, q
  @@intersect = R2Point.intersect?(@p,@q,@@apoint,@@bpoint)
  @@d = 0.0 if @@intersect
  if @q.dist_to_line(@@apoint,@@bpoint)<@@d && !@@intersect
    @@d = @q.dist_to_line(@@apoint,@@bpoint)
  end
...
end
```

Для `Polygon`:

```
...
if c.dist_to_line(@@apoint,@@bpoint)<@@d && !@@intersect
@@d = c.dist_to_line(@@apoint,@@bpoint)
end
...
```

При этом, в случае добавления двух новых рёбер в дек искомое расстояние изменится, а следовательно, его необходимо перевычислить. В случае пересечения прямой и выпуклой оболочки расстояние необходимо приравнять к нулю.

```
...
@intersect=R2Point.intersect?(t,@points.first ,@@apoint,@@bpoint)

if !@@intersect
 @@intersect=R2Point.intersect?(t,@points.last ,@@apoint,@@bpoint)
end

@@d = 0.0 if @@intersect
```

```
if t.dist_to_line(@@apoint,@@bpoint)<@@d && !@@intersect
  @@d = t.dist_to_line(@@aline,@@bline)
end

@points.push_first(t)
...
```

4.2 Постановка задачи

В следующем проекте решалась задача вычисления радиуса максимального круга с центром в заданной точке, содержащегося в выпуклой оболочке. Кроме того, необходимо определять выпуклую оболочку и вычислять её характеристики сразу после поступления очередной точки. Для наглядного отображения работы программы необходимо графически иллюстрировать каждый этап, используя графический интерфейс Tk.

4.2 Теоретические аспекты

Помимо вышеописанных методов и алгоритмов, используемых в выполнении предыдущей модификации, необходимо индуктивно вычислять максимальный радиус вписанной окружности, т.е. минимальное из расстояний от заданного центра до каждого из рёбер выпуклой оболочки.

4.2 Описание используемых структур и применяемых алгоритмов

В общем случае необходимо искать минимальное из расстояний от заданного центра до каждого ребра выпуклой оболочки (если центр лежит внутри многоугольника). Если же центр окружности находится вне оболочки — радиус вписанной окружности будет равен нулю.

В первую очередь, необходимо объявить экземпляр класса `Figure` для центра окружности, и метод, который будет вычислять искомый радиус:

```
@@center = nil
def radius; @@R end
```

Далее, при первом же вычислении радиуса многоугольника необходимо проверить, лежит ли центр внутри выпуклой оболочки:

```
def outside?
    @points.size.times do
      break if @@center.light?(@points.last, @points.first)
      @points.push_last(@points.pop_first)
    end
    @@center.light?(@points.last, @points.first)
end
```

Для общего случая будем сразу выполнять эту проверку и в случае надобности обнулять переменную @@R, отвечающую за радиус:

```
outside?() ? @@R = 0 : @@R = [@@center.dist_to_line(a,b),
@@center.dist_to_line(b,c),@@center.dist_to_line(a,c)].min
```

Ниже приведён графический вывод работы программы для заданных точек:

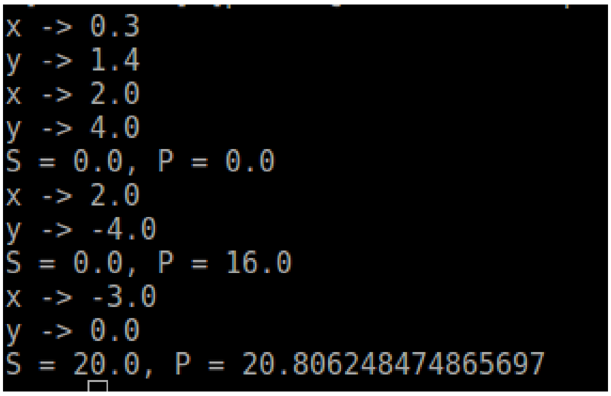

Рис. 9. Ввод данных

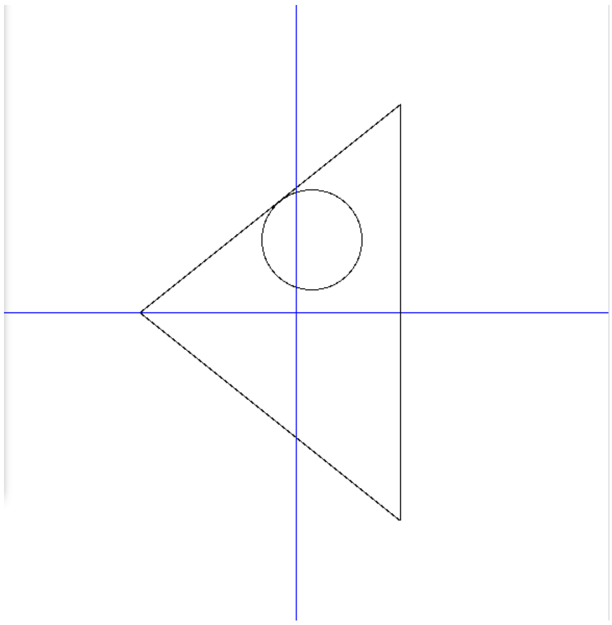

Рис. 10. Вывод в Tk

В приложении приведены реализации некоторых технических моментов, связанных с изменением выпуклой оболочки.

5. Модификация проекта «Изображение проекции полиэдра»

Постановка задачи

Пусть точку в пространстве назывется «хорошей», если её проекция находится строго вне квадрата единичной площади с центром в начале координат и сторонами, параллельными координатным осям. В данном прокте необходимо было модифицировать эталонный проект таким образом, чтобы определялась и печаталась следующая характеристика полиэдра: сумма длин рёбер, середина и оба из концов которых — «хорошие» точки.

В общем случае, для успешного решения задачи необходимо знакомство с основами вычислительной геометрии.

Теоретические аспекты

Для начала приведём некоторые понятия.

`Полиэдр (polyedr)` — Множество выпуклых плоских многоугольников.

`Грань полиэдра (facet)` — Выпуклый плоский многоугольник.

`Ребро полиэдра (edge)` — Любая из сторон одной из граней.

`Вершина полиэдра (vertex)` — Любая из вершин произвольной грани.

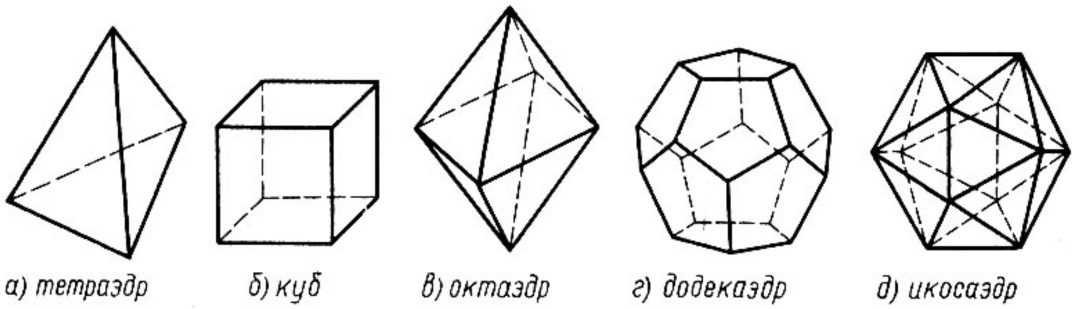

Рис. 11. Некоторые виды полиэдров

В рассматриваемом проекте речь идёт о геометрических объектах, находящихся в трёхмерном пространстве, поэтому необходимо вспомнить некоторые факты из линейной алгебры и аналитической геометрии.

Прежде всего вследует вспомнить, что точку пространства $\mathbb{R}^3$ можно задать с помощью трёх действительных чисел — её координат. Эти же координаты задают и вектор трёхмерного пространства, начало которого совпадает с началом координат, а конец — с рассматриваемой точкой.

Для векторов в пространстве определены операции сложения, вычитания и умножения на число, порождающие новый вектор, координаты которого находятся с помощью сложения, вычитания и умножения на число координат исходных векторов.

Также необходимо знание скалярного, векторного и смешанного произведения векторов, и уметь использовать данные операции.

Помимо прочего, в данном проекте необходимо использовать линейные преобразования (отображения) плоскости и пространства. Любое такое преобразование, как известно, задаётся квадратной матрицей порядка два для плоскости и порядка три для пространства. Напомним, что тождественному преобразованию соответствует единичная матрица E.

Среди всех преобразований в данном случае особенно интересны повороты и гомотетия, задаваемая матрицей вида kE, где k — положительная константа, а E — единичная матрица. Ортогональные преобразования плоскости и пространства, сохраняющие ориентацию, являются поворотами. На плоскости поворот на угол φ задаётся матрицей вида

$$\begin{pmatrix} \cos\varphi & \sin\varphi \\ -\sin\varphi & \cos\varphi \end{pmatrix}.$$

.

В случае пространства повороту на угол φ вокруг оси Oz соответствует следующая матрица:

$$\begin{pmatrix} \cos\varphi & \sin\varphi & 0 \\ -\sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

.

Аналогичный вид имеют и матрицы поворота вокруг осей Ox и Oy, а произвольное вращение пространства может быть представлено в виде композиции трёх последовательных поворотов вокруг осей Oz, Oy и вновь Oz.

В процессе работы над проектом нам потребуется решать следующую задачу: найти пересечение отрезка с полупространством, которое задано точкой на граничной плоскости и вектором внешней нормали к ней. Двумерный аналог этой задачи таков: найти пересечение отрезка на плоскости с полуплоскостью, которая задана точкой на граничной прямой и вектором внешней нормали к этой прямой. Рассмотрим решение последней из задач.

Пусть концы отрезка B (от английского слова `begin`) и F (от `final`) имеют соответственно координаты (x_1, y_1) и (x_2, y_2) точка на граничной прямой — координаты (x_0, y_0), а вектор внешней нормали к полуплоскости — это вектор $\overrightarrow{N}$.

Для этого необходимо и достаточно, чтобы оба конца отрезка лежали с «нужной» стороны от заданной прямой.

Точка (x_1, y_1) будет лежать с «нужной» (внешней по отношению к полуплоскости) стороны от граничной прямой, если угол между векторами внешней нормали и вектором, соединяющим точки (x_0, y_0) и (x_1, y_1) является острым.

Искомое условие имеет следующий вид: $\overrightarrow{N} \cdot \overrightarrow{AB} > 0$. Тогда для того чтобы пересечение отрезка с полуплоскостью представляло собой пустое множество, необходимо чтобы: $\overrightarrow{N} \cdot \overrightarrow{AB} > 0 \wedge \overrightarrow{N} \cdot \overrightarrow{AF} > 0$.

Если знаки указанных скалярных произведений разные, то отрезок пересекает прямую, ограничивающую полуплоскость. В этом случае для определения пересечения отрезка с полуплоскостью прежде всего необходимо найти точку их пересечения. Произвольная точка отрезка BF может быть записана как $(1-t)B+tF$, где $0 \leqslant t \leqslant 1$. При $t = 0$ эта формула задаёт начало отрезка (точку B), а при $t = 1$ — его конец (точку F). Отсюда следует, что искомая точка пересечения в свою очередь соответствует значению

$$t = \frac{\overrightarrow{N} \cdot \overrightarrow{AB}}{\overrightarrow{N} \cdot \overrightarrow{AF} - \overrightarrow{N} \cdot \overrightarrow{AB}},$$

так как для неё $\overrightarrow{N} \cdot \overrightarrow{AC} = 0$.

Описание используемых структур и применяемых алгоритмов

Точку в трёхмерном пространстве ($\mathbb{R}^3$), так же, как и вектор этого пространства, мы будем представлять объектом класса `R3`, имеющем три действительные компоненты (`@x, @y` и `@z`), — координаты точки или вектора. В этом классе реализуем ряд методов, обеспечивающих все необходимые в данном проекте манипуляции над точками (векторами).

Каждое из рёбер (экземпляров класса `Edge`) полиэдра будем задавать его началом и концом — объектами класса `R3`, а произвольную его грань (экземпляр класса `Facet`) — массивом её вершин. Сам полиэдр будет являться экземпляром класса `Polyedr`, в конструкторе (методе `initialize`) которого будет обрабатываться файл, содержащий всю информацию о полиэдре.

Так как плоскость проектирования горизонтальна, то достаточно «забыть» z-координату.

Теперь рассмотрим некоторые программистские аспекты. Модификация данного проекта довольна проста и, по сути, заключается в добавлении нового параметра — суммы длин рёбер, середина и оба из концов которых — «хорошие» точки. Следовательно вся задача сводится к определению «хороших» точек ребра и умению вычислять середину отрезка.

Найдем обратное условие, когда данная нам точка не является «хорошей». Очевидно, что оно будет выглядеть следующим образом (так как мы рассматриваем проекцию — нет смысла включать в условие координату `z`):

$$x, y < 0.5 \quad x, y > -0.5$$

Учитывая коэффициент гомотетии и данное условие, можно написать необходимый метод:

```
def good?(c)
   return false if ((@x.abs<0.5*c)&&(@y.abs<0.5*c))
   true
end
```

Середина отрезка находится стандартным способом:

Пусть заданы два конца отрезка в пространстве $A(x_a, y_a, z_a)$ и $B(x_b, y_b, z_b)$, тогда середина отрезка имеет следующие координаты:

$$x_c = \frac{x_a + x_b}{2},\ y_c = \frac{y_a + y_b}{2},\ z_c = \frac{z_a + z_b}{2}.$$

Для реализации данного метода у класса `Edge`(Ребро) есть две переменные `@@beg`, `@@fin`, отвечающие за начало и конец отрезка. Учитывая то, что каждая из этих точек уже имеет свои координаты, метод нахождения середины отрезка будет выглядеть следующим образом:

```
def middle
    (@beg+@fin)/2
end
```

Исходя из всего вышеупомянутого, можно написать метод вычисления длины ребра. Чтобы найти длину отрезка можно воспользоваться скалярным произведением векторов. Для этого необходимо ребро превратить в вектор, то есть взять разность `@fin-@beg`, обозначить её как $\vec{A}$, и используя метод класса `R3 dot`, вычислить квадратный корень из скалярного квадрата вектора $\vec{A}$. Ниже представлена реализация методов `dot` и `length`.

```
def dot(other)
   @x*other.x+@y*other.y+@z*other.z
end

def length(c)
    if (@beg.good?(c) && @fin.good?(c) && self.middle)
      a=(@fin-@beg)*(1/c)
      Math.sqrt(a.dot(a))
    else
      0
    end
end
```

Итак, всё что остаётся сделать — при задании нового ребра его вершинами, проверять координаты начала, конца и середины ребра на «хорошие» точки, и суммировать подходящие нам рёбра. Всё это реализуется в классе `Polyedr`, в цикле задания рёбер очередной грани:

```
(0...size).each do|n|
      e = Edge.new(vertexes[n-1],vertexes[n])
      @edges << e
      @length += e.dist(c)
end
```

Список литературы и интернет-ресурсов

[1] http://ru.wikipedia.org/wiki/Ruby — Википедия (свободная энциклопедия) о языке Ruby.

[2] https://home.mephi.ru/files/2373/material_ici_toc.zip/index.html — Описание проекта «Выпуклая оболочка».

[3] http://www.tkdocs.com/tutorial/install.html — Описание проекта «Выпуклая оболочка».

[4] https://home.mephi.ru/files/3214/material_ici_toc.zip/index.html — Описание проекта «Изображение проекции полиэдра».

[5] https://home.mephi.ru/files/2077/material_ici_toc.zip/index.html — Описание проекта Стековый компилятор формул.

6. Приложение к пункту 4.2

В участки кода программы, отвечающие за удаление освещённых рёбер и добавление новых, были внесены следующие изменения:

При удалении или добавлении рёбер с некоторой вероятностью придётся пересчитывать радиус вписанной окружности. На самом этапе удаления ему будет временно присваиваться значение `-1.0`. Далее искомый радиус будет вычисляться в зависимости от ситуации.

Учёт удаления ребра, соединяющего конец и начало дека:

```
if (@@R - @@center.dist_to_line(@points.last,@points.first))<EPS
   @@R = -1.0
end
```

Удаление освещённых рёбер из начала дека:

```
@@R = -1.0 if @@R == @@center.dist_to_line(p, @points.first)
```

Удаление освещённых рёбер из конца дека:

```
@@R = -1.0 if @@R == @@center.dist_to_line(p, @points.last)
```

В случае, если надо переопределить радиус вписанной окружности, мы также ищем минимальное из расстояний от центра до каждого ребра.

```
if @@R == -1.0
  @@R = @@center.dist_to_line(@points.last, @points.first)
  @points.size.times do
    if @@R > @@center.dist_to_line(@points.last, @points.first)
    end
    @@R = @@center.dist_to_line(@points.last, @points.first)
    @points.push_last(@points.pop_first)
  end
else
 @@R=[@@R, @@center.dist_to_line(@points.last, @points.first)].min
 @points.push_last(@points.pop_first)
 @@R=[@@R, @@center.dist_to_line(@points.last, @points.first)].min
end
```