МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
имени М. В. ЛОМОНОСОВА
ФАКУЛЬТЕТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И КИБЕРНЕТИКИ

Отчет по заданию №6

«Сборка многомодульных программ. Вычисление корней уравнений и определенных интегралов.»

Вариант 4 / 2 / 1

Исполнитель:
студент 106 группы
Панькин М. М.

Преподаватель:
Корухова Л. С.

Москва
2019

Содержание

1 Постановка задачи

Требуется реализовать две программы. Первая программа является вспомогательной и создает ассемблерный листинг на основе файла с записанными выражениями . Вторая программа является основной и вычисляет площадь фигуры ограниченной графиками трех функций с заданной точностью, используя численные методы: метод секущих для нахождения точек пересечения кривых и метод прямогульников для вычисления интегралов функций. Отрезок, на котором производится поиск корней вычисляется аналитически и записан в файле выражения `func.expr` из которого происходит компиляция ассемблерного файла `func.nasm`.

Компилирующая программа должна выполнять следующие задачи:

- Парсить файл выражения, записанный в определенном формате и строить на его основе дерево разбора
- По дереву разбора строить ассемблерный файл с четыремя фукнциями.
- В качестве дополнительной функции, программа оптимизирует выражения, вычислимые на этапе комплияции.
- Также программа строит выражения в инфиксной форме и записывает в `expr.js`, что используется для построения графика этих функций с отмеченными точками пересечения кривых.

Основная программа должна выполнять следующие задачи:

- Выводить список возможных аргументов командной строки.
- Выводить результат работы программы в виде одного числа, либо в читаемом формате вместе с абсциссами точек пересечения. Также должен быть возможен вывод количества итераций, выполненых для нахождения точек пересечения.
- Находить абсциссы точек пересечения на заданном отрезке любых двух кривых из списка заранее заданных тестировочных функций.
- Находить значение интеграла на заданном отрезке любой функции из списка заранее заданных.

Пример файла выражения `func.expr`:

```
-0.5 3.0
0
x sin
x x * 2.0 -
```

Listing 1: func.expr

Первая строка файла содержит границы поиска корней, записанные в виде десятичных числел. В следующих трех строках записаны выражения в обратной польской нотации, содержащие константы, константы π и e, а также операции sin, cos, tan, ctg, $+$, -, * и $/$.

Исходный код доступен в репозитории [1].

2 Тестируемые наборы

Для тестирования выберем функции, такие что $|f(x)| \leq 10$ и $|f''(x)| \leq 10$ для $\forall x \in [a, b]$. При таких параметрах погрешность интегрирования одной функции, полученная из-за погрешности дифференцирования $E_\delta \leq 2 \cdot 10\varepsilon_1$. Суммарная погрешность $E \leq 3(E_I + E_\delta) = 60\varepsilon_1 + 3\varepsilon_2$. Значит точности $\varepsilon_1 = \varepsilon_2 = 10^{-5}$ будет достаточно для любой фигуры и при этом не будет накапливаться большая вычислительная ошибка.

Будем тестировать программу на следующих наборах функций.

$$\begin{cases} f_1(x) = 0 \\ f_2(x) = \sin(x) \\ f_3(x) = x^2 - 2 \\ a = -0.5 \\ b = 3 \end{cases} \tag{1}$$

$$\begin{cases} f_1(x) = \sin^2(x) \\ f_2(x) = \frac{1}{x+1} \\ f_3(x) = x \\ a = -0.5 \\ b = 1.5 \end{cases} \tag{2}$$

$$\begin{cases} f_1(x) = 6\tan(\frac{x}{2}) \\ f_2(x) = \pi - x^3 \\ f_3(x) = \frac{e}{x+1} \\ a = -0.5 \\ b = 1 \end{cases} \tag{3}$$

Численные методы доказываются в учебнике по математическому анализу [2].

3 Результаты экспериментов

Для тестируемых наборов были получены следующие результаты:

3.1 Набор 1

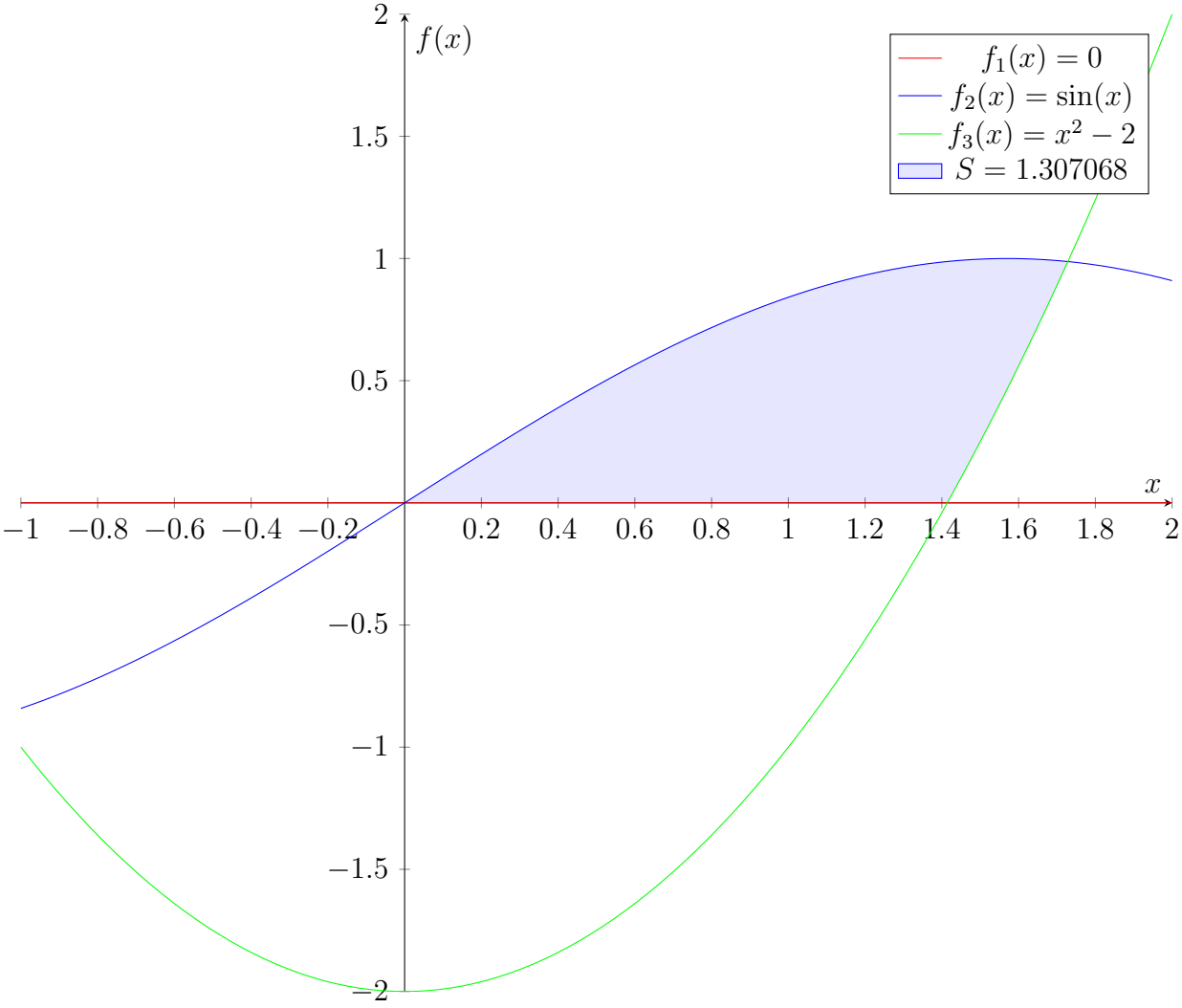

Рис. 1: График для набора 1

	x_{12}	x_{23}	x_{31}	S
Реальное значение	0.000000	1.728466	1.414213	1.0070
Полученное значение	0.000006	1.728470	1.414206	1.0070
$\|\Delta\|$	0.000006	0.000006	0.000007	0

Таблица 1: Таблица для набора 1

3.2 Набор 2

	x_{12}	x_{23}	x_{31}	S
Реальное значение	0.831472	0.618033	0.000000	0.1480
Полученное значение	0.831474	0.618037	0.000007	0.1480
$\|\Delta\|$	0.000002	0.000004	0.000007	0

Таблица 2: Таблица для набора 2

3.3 Набор 3

	x_{12}	x_{23}	x_{31}	S
Реальное значение	0.816849	-0.135428	0.563939	0.4953
Полученное значение	0.816847	-0.135424	0.563935	0.7227
$\|\Delta\|$	0.000002	0.000004	0.000004	0

Таблица 3: Таблица для набора 3

В таблице результаты округлены до 6 знаков после запятой для координат точек пересечения и до 4 знаков для итогового результата. В результате проведения эксперимента на тестовых наборах, были получены результаты входящие в пределы погрешности $\varepsilon < 10^{-3}$.

4 Структура программы и спецификация функций

4.1 Структура проекта

Струтуру проекта можно представить в виде следующей диаграмы:

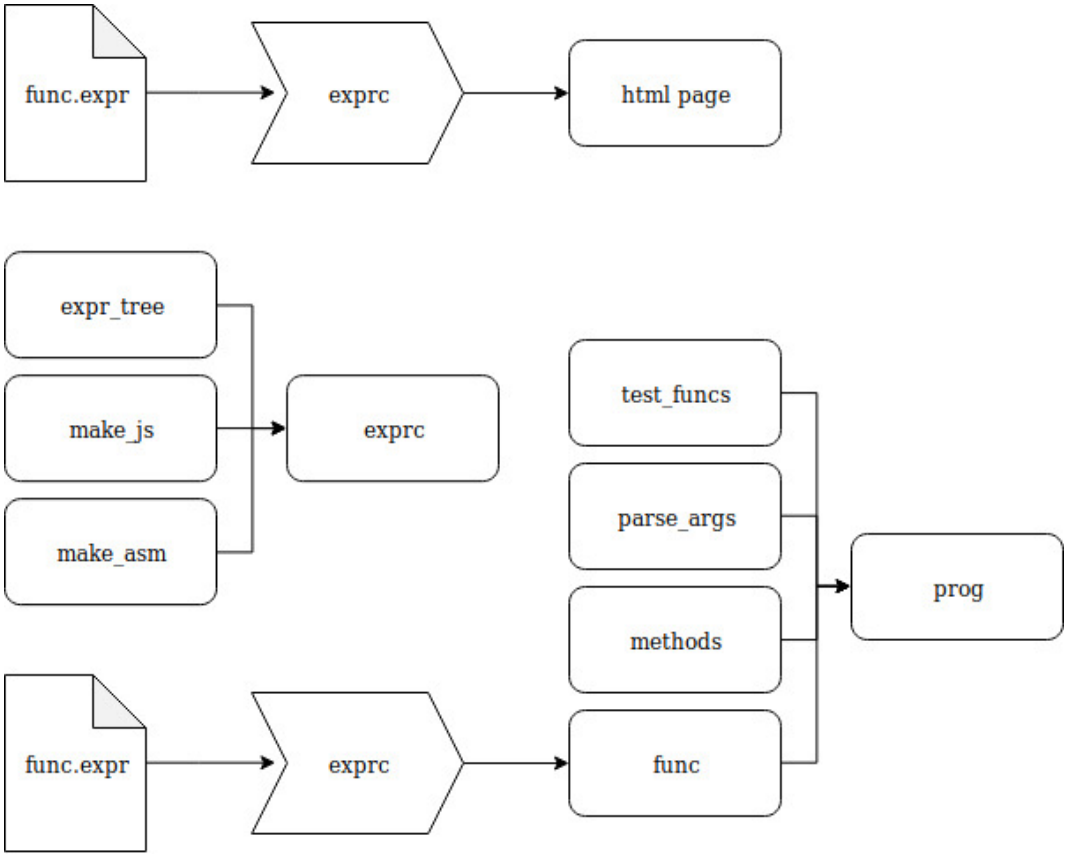

Рис. 2: Структура проекта

4.2 Спецификации функций

Далее приведены спецификации всех функций проекта по модулям.

4.2.1 main.c

```c
/// Выводит информацию о программе и принимаемые аргументы командной строки.
void help(char *current_dir);

/// Точка входу в программу.
/// Разбирает аргументы командной строки и в зависимости от них
/// запускает соответствующий режим работы.
int main(int argc, char** argv);
```

4.2.2 methods.h

```c
/// Находит абсциссу точки пересечения графиков функций f и g.
/// Использует метод секущих.
/// В it_count пишет количество итераций, после которых корень был найден
/// с точностью eps.
/// a и b задают границы поиска, для работы функции необходимо, чтобы
/// на них функция принимала разные по знаку значения.
/// Если a или b - корень, то он возвращается.
double root(func_t f, func_t g,  double a, double b, double eps, int *it_count);

/// Находит интеграл функции f на отрезке [a, b]
/// Использует метод прямоугольников.
/// Точность - eps.
/// Изначально шаг разбиения не больше eps.
/// Увеличивает количество отрезков разбиения до тех пор,
/// пока разность предыдущего и текущего значения превышает eps.
double integral(func_t f, double a, double b, double eps);
```

4.2.3 func.h

```c
/// Инициализирует FPU
void INIT_FPU(void);

// Функции, чьи графики ограничивают фигуру.
double f1(double);
double f2(double);
double f3(double);
```

4.2.4 parse_args.h

```c
/// Проверяет наличие флага flag среди аргументов командной строки.
int has_flag(int argc, char **argv, const char *flag);

/// Находит первый аргумент командной строки, совпадающий с flag.
/// Если после него хотя бы count аргументов, то возвращает указатель
/// на следующий за ним аргумент (первый аргумент флага). Иначе NULL.
char** get_flag(int argc, char **argv, const char *flag, int count);
```

4.2.5 test_funcs.h

```c
/// Тестовые функции.
double t_f1(double);
double t_f2(double);
double t_f3(double);
double t_f4(double);
double t_f5(double);

/// Выводит на консоль результаты тестирования метода нахождения
/// точек пересечения.
/// Если human, то выводит в человеко-читаемом формате.
/// Если iter, то выводит количесто итераций до нахождения ответа.
void test_root(char **arguments, int human, double EPS, int iter);

/// Выводит на консоль результаты тестирования метода интегрирования.
/// Если human, то выводит в человеко-читаемом формате.
void test_integral(char **arguments, int human, double EPS);
```

4.2.6 exprc.c

```c
/// Считывает fin и строит файлы f_asm и f_js.
void compile(FILE *fin, FILE *f_asm, FILE *f_js);

/// Точка входа. Разбирает аргументы командной строки и запускает compile.
int main(int argv, char** argc);
```

4.2.7 make_asm.h

```
/// Строит ассемлерный файл fout.
/// В секции .rodata размещены переменные a и b.
/// В секции .text функции f1, f2, f3.
void make_asm(double a, double b, Node *tree1, Node *tree2, Node *tree3, FILE *fout);

/// Заполняет массив label_arr значениями констант в дереве tree
/// в порядке обхода в глубину (текущая вершина, левое поддерево, правое).
/// Значение token константной вершины становится равным T_LABEL.
/// Значение label_num становится равным ее номеру в массиве label_arr.
int fill_label_arr(int i, double *label_arr, Node *tree);

/// Добавляет в конец fout ассемблерный код функции с названием name
/// и вычисляющую выражение tree.
void print_asm_func(const char *name, Node *tree, FILE *fout);

/// Печатает в конец fout ассемблерный код, вычисляющий выражение tree.
void print_asm_node(Node *tree, FILE *fout);
```

4.2.8 make_js.h

```
/// Печатает в файл fout javascript код, содержащий константы a и b
/// и выражения tree1, tree2, tree3.
void
make_js (double a, double b, Node *tree1, Node *tree2, Node *tree3, FILE *fout);
```

5 Сборка программы (Makefile)

5.1 Структура

Для сборки программы используется утилита make, компилятор gcc и ассемблер nasm для ассемблирования промежуточного файла с функциями.

Проект содержит 8 файлов с исходным кодом C, 8 заголовочных файлов, 1 промежуточный ассемблерный файл создается во время сборки, 1 Makefile, а также отрисовывающий график код, не входящий в дерево сборки.

Дерево сборки можно представить в виде следующей диаграмы:

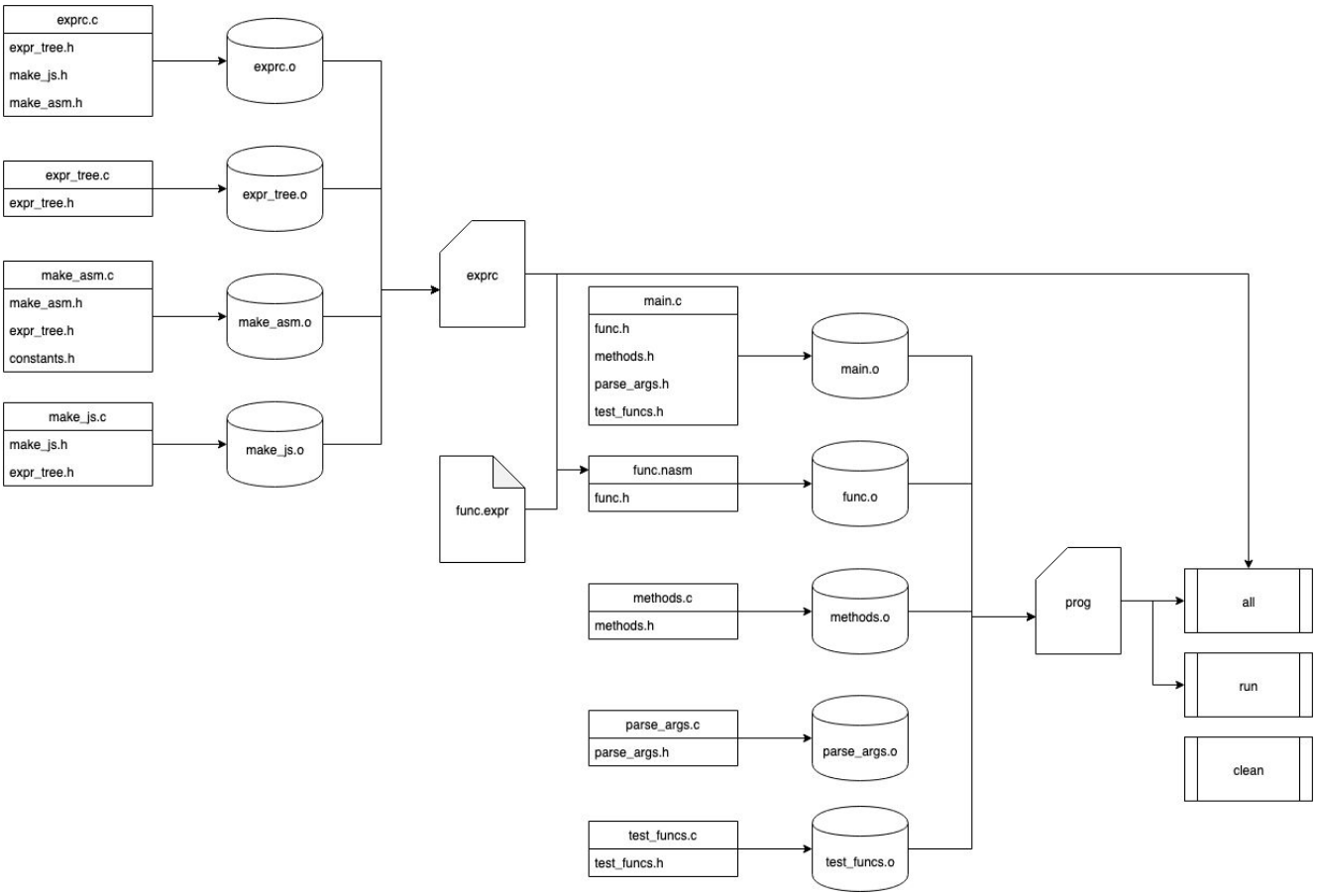

Рис. 3: Граф сборки

В большем разрешении диаграмма находится в репозитории проекта [1].

5.2 Переменные

Могут быть изменены следующие переменный в Makefile:

- `TARGET` - Расположение исполняемого файла основной программы.
- `EXPR_COMPILER` - Расположение исполняемого файла комплиятора выражений.
- `JS_LOC` - Расположение `expr.js`. Для корректной работы в той же директории должны находиться `graph.html`, `make_graph.js` и `style.css`.

6 Отладка программы, тестирование функций

6.1 Тестирование

Для отладки в программе предусмотрен режим тестирования с заранее заданными функциями.

6.1.1 Интегрирование

- $\int_0^\pi \sin(x)dx = 2$
- $\int_1^e \frac{1}{x}dx = 1$
- $\int_2^4 x + 1dx = 8$

6.1.2 Дифференцирование

- $x + 1 = \frac{1}{x}, x = \frac{-1+\sqrt{5}}{2} \approx 0.61803$
- $x + 1 = 0, x = -1$
- $x^2 = x + 1, x = \frac{1+\sqrt{5}}{2} \approx 1.61803$

Для всех данных примеров программа выдает с результат с необходимой точностью.

6.2 Отладка

Для отладки численных методов использовался вывод состояния программы внутри кода методов и тестирование на различных примерах.

Для отладки остальной части программы использовался gdb, чтение генерируемого программой ассемблерного кода и специальный график, генерируремый программой, по которому можно узнать, правильно ли считалось выражение и быстро узнать значения корней.

7 Программа на Си и на Ассемблере

Исходные коды проекта приложены в архиве вместе с данным отчетом, а также доступны для скачивания в репозитории проекта на github.com[1].

8 Анализ допущенных ошибок

В ходе написания программы были допущены следующие ошибки:

- Метод интегрирования находил результат не с точностью ε_2, а брал длину шага разбиения равной ε_2.
- В функции `read_node` некорректно определялась ошибка при работе `sscanf`, из-за чего работа функции продолжалась и происходило обращение к элементу за пределом буфера.
- В функции `print_tree` не учитывалась необходимость экранировать '\'.
- В функции `read_node` по ошибке считывались не 'sin', 'cos' и.т.д., а '\\sin', '\\cos' и.т.д.
- В функции `print_tree` был забыл `break` внутри `switch`.
- При вычислении интеграла, у которого левая граница больше правой, границы менялись местами, а значение не домножалось на -1.
- В функции `fill_label_arr` были перепутаны константы pi и e.

Список литературы

[1] Pankin Mikhail. Project repository, 2019. URL `https://github.com/mishapankin/Area`.

[2] Позняк Э.Г. Ильин В.А. *Основы математического анализа. Часть 1.* ФИЗМАТЛИТ, Москва, 2005.