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 11 / 5 / 3

Выполнил:
студент 106 группы
Макаров С. С.

Преподаватель:
Корухова Л. С.

Москва
2018

Содержание

Постановка задачи

Требуется реализовать программу, позволяющую вычислять площадь плоской фигуры, ограниченной тремя кривыми, используя метод Симпсона. Для нахождения вершин фигуры использовать два метода, выбор между которыми должен происходить на этапе сборки программы. Рассчитать оценку точности для вычислительных методов, достаточной для достижения необходимой точности площади ε. Отрезок, на котором необходимо искать вершины, а также входные кривые, задаётся во входном файле, имя которого задаётся также на этапе компиляции. Используется следующий формат файла:

Первая строка содержит два вещественных числа, разделённые пробелами. Числа задают границы отрезка, на котором следует искать точки пересечения кривых. Следующие три строки описывают функции, используя для этого обратную польскую нотацию. Каждая строка содержит разделённые пробелами термы, которыми могут быть: переменная величина x, вещественное число либо операция.

Переменная задаётся символом ’x’. Вещественные числа задаются в формате, поддерживаемом функцией scanf. Также возможны константные значения ’e’ и ’pi’. Поддерживаемые операции:

- Бинарные: ’+’, ’-’, ’*’, ’/’
- Унарные: ’sin’, ’cos’, ’tan’, ’ctg’

Математическое обоснование

Поскольку был взят усложнённый вариант задания, рассматривались два набора кривых. Первый взят из варианта 1, а второй - из примера к описанию усложнённого варианта. Рассмотрим эти наборы и приведём рассуждения по выбору отрезков поиска корней и метода поиска корней.

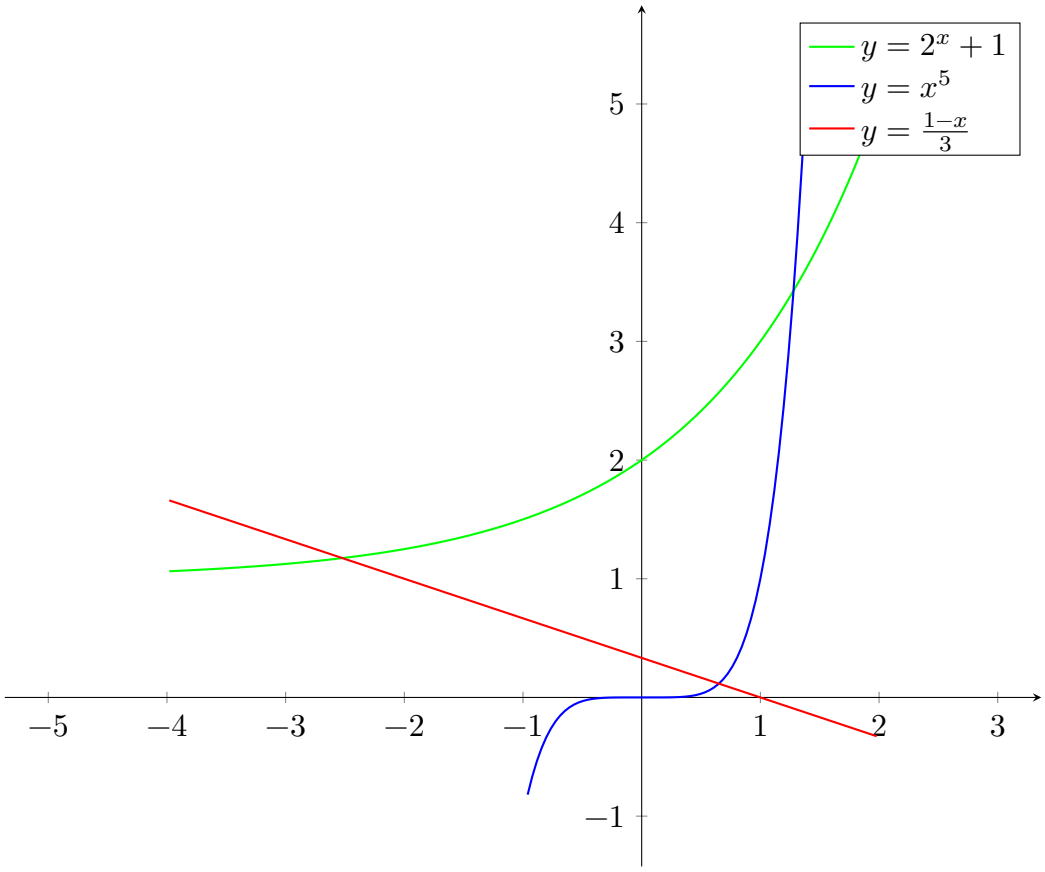


Рис. 1: Плоская фигура, ограниченная графиками заданных уравнений

Несложно проверить, что все три функции имеют четвёртую непрерывную производную на всей области определения, поэтому метод Симпсона для них применим[1]. Из графика можно видеть, что все точки пересечения лежат в отрезке $[-3; 2]$, поэтому достаточно рассматривать функции на нём. Вспомним условия применимости используемых методов[1]:

1. Метод «вилки»:
    - Непрерывность функции;
    - Разные знаки значений функций на концах отрезка;
2. Метод хорд и касательных:
    - Функция должна иметь монотонную непрерывную производную, не обращающуюся на отрезке в ноль;

Рассмотрим функции $f(x) = 2^x + 1 - x^5$, $g(x) = x^5 - \frac{1-x}{3}$, $h(x) = \frac{1-x}{3} - (2^x + 1)$. Все три функции, очевидно, дважды дифференцируемы на рассматриваемом отрезке. Найдём их значения в крайних точках:

$$f(-3) = 244.125, f(2) = -27$$

$$g(-3) \approx -244.33, g(2) \approx 32.33$$

$$h(-3) \approx 0.21, h(2) \approx -5.33$$

Таким образом, требования метода «вилки» на рассматриваемом отрезке выполняются.

Для проверки выполнения требований метода хорд и касательных посчитаем производные рассматриваемых функций:

$$f'(x) = \ln(2) * 2^x - 5x^4$$

$$g'(x) = 5x^4 + \frac{1}{3}$$

$$h'(x) = -\frac{1}{3} - \ln(2) * 2^x$$

На следующем рисунке приведены графики производных. Сопоставляя этот рисунок с предыдущим, получаем, что на любом отрезке, содержащем все точки пересечения, найдутся корни производной $\dot{g}(x)$. Это означает, что мы не можем описать входной файл в заданном формате так, чтобы точки пересечения могли быть найдены с помощью метода хорд и касательных, поэтому нужно использовать метод ”вилки”.

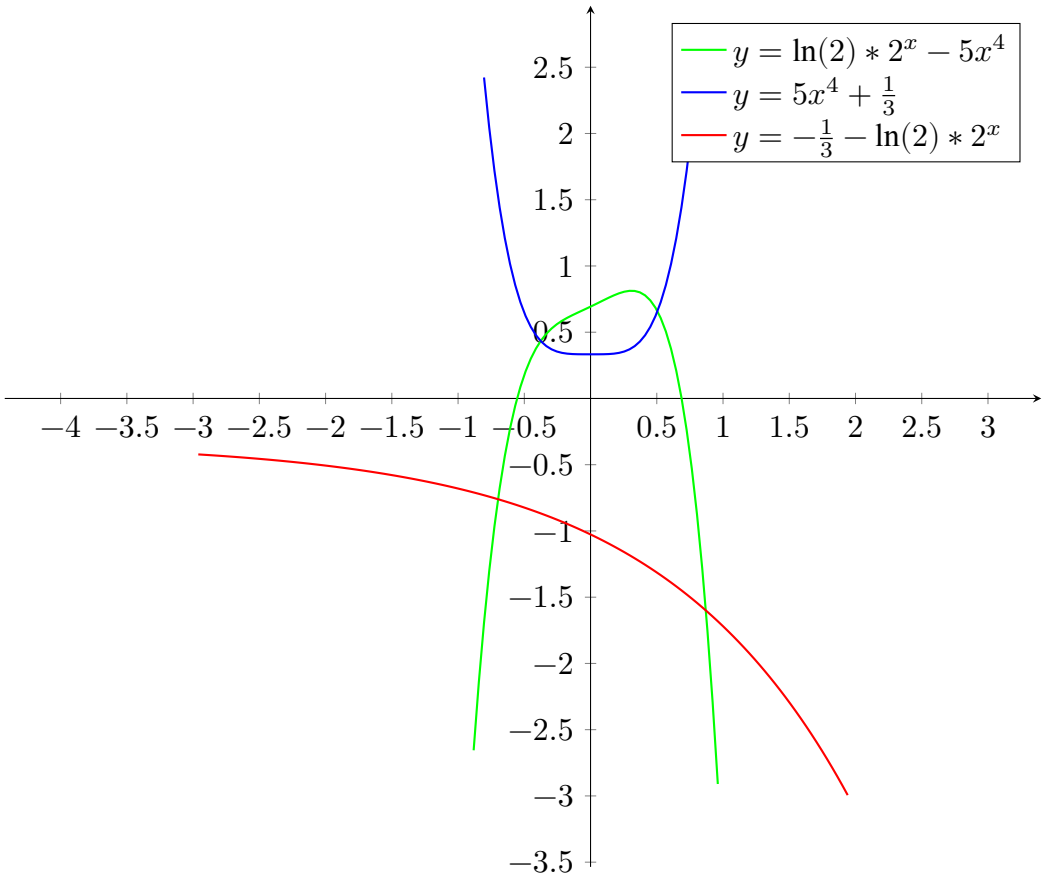


Рис. 2: Графики производных $f(x), g(x)h(x)$

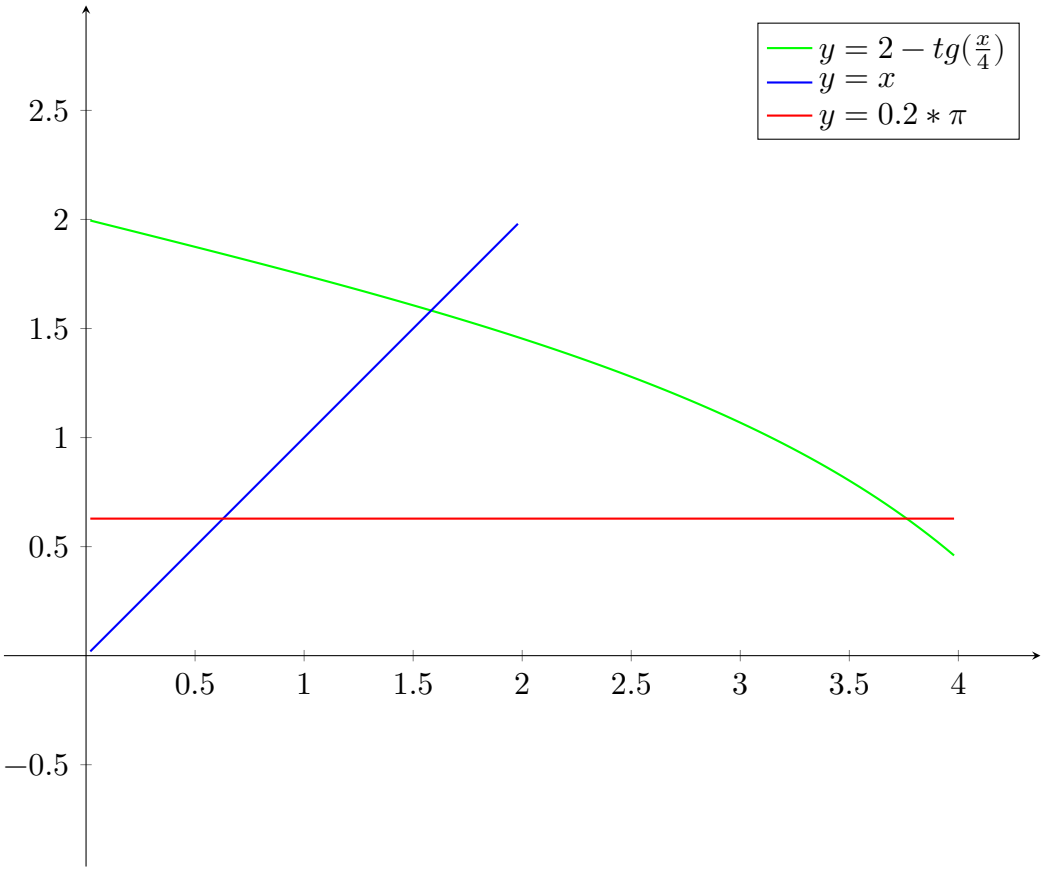


Рис. 3: Плоская фигура, ограниченная графиками заданных уравнений

Теперь рассмотрим второй набор кривых. Отрезок уже дан в исходном файле: [0; 4], остаётся только проверить применимость метода хорд и касательных на нём. Рассмотрим функции $f(x) = 2 - \text{tg}(\frac{x}{4}) - x$, $g(x) = x - 0.2 * \pi$ и $h(x) = 0.2 * \pi - (2 - \text{tg}(\frac{x}{4}))$. Точно так же рассчитаем их производные:

$$f'(x) = -\frac{1}{4 * \cos^2(\frac{x}{4})} - 1$$

$$g'(x) = 1$$

$$h'(x) = \frac{1}{4 * \cos^2(\frac{x}{4})}$$

На рисунке 4 представлены графики этих производных. Как видно из графиков, все три производные монотонны на $[0; 4]$ и не обращаются на нём в ноль, так что метод хорд и касательных в этом случае применим.

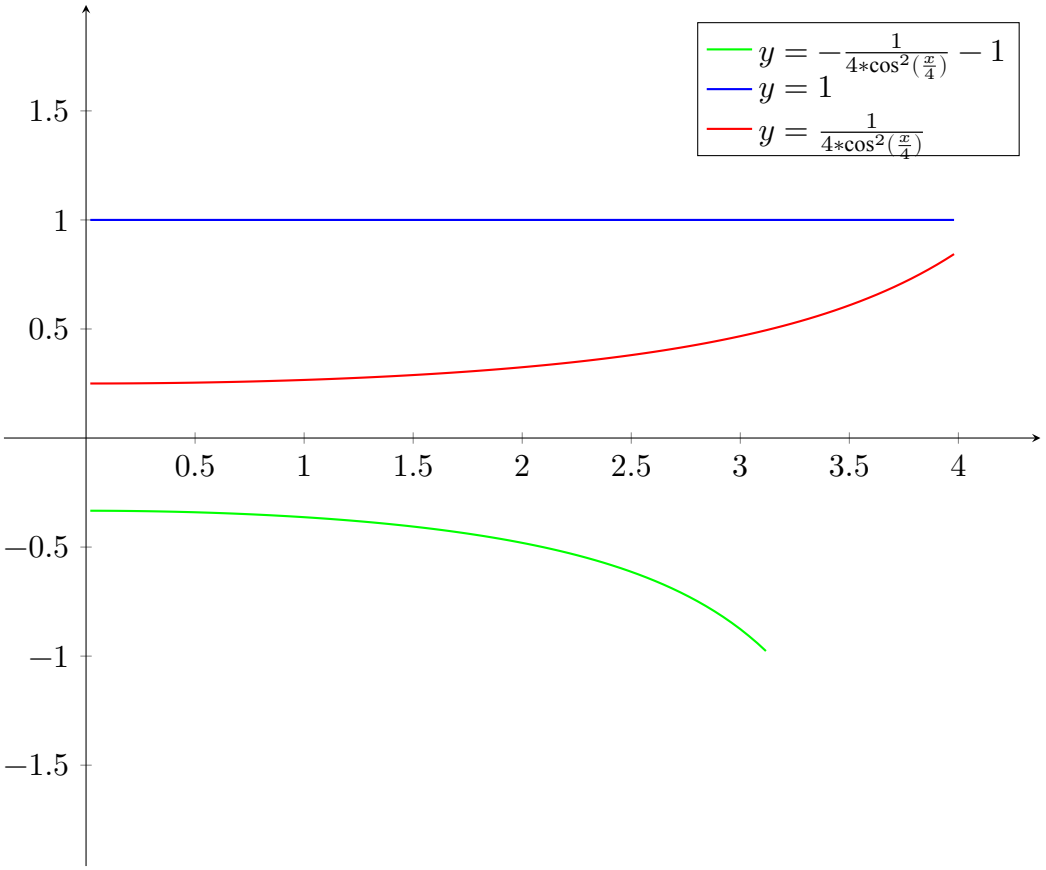


Рис. 4: Графики производных $f(x)$, $g(x)$ и $h(x)$

Ещё не был рассмотрен вопрос выбора погрешностей ε_1 и ε_2 для достижения требуемой точности ε. Рассмотрим этот вопрос в общем случае. Пусть нам даны кривые, заданные уравнениями $y = f(x)$, $y = g(x)$ и $y = h(x)$, имеющие попарные пересечения на отрезке $[a; b]$. Обозначим за x_1 корень уравнения $f(x) = g(x)$, за x_2 - корень уравнения $g(x) = h(x)$, за x_3 - корень уравнения $h(x) = f(x)$, за S - искомую площадь. Введём также обозначения $S_1 = \int_{x3}^{x1} f(x)dx$, $S_2 = \int_{x1}^{x2} g(x)dx$, $S_3 = \int_{x2}^{x3} h(x)dx$. Из геометрического смысла определённого интеграла следует, что $S = S_1 + S_2 + S_3$.

Для оценки погрешностей применим следующую формулу:

Пусть $F = F(x_1, x_2, ..., x_n)$ - дифференцируемая функция от n переменных. Тогда для погрешности величины $F(x_1, x_2, ..., x_n)$ справедлива формула[2]:

$$\Delta F = \sqrt{\sum_{i=1}^{n} (\frac{\partial F}{\partial x_i} * \Delta x_i)^2}$$

В этой формуле $\Delta x_1, \Delta x_2, ..., \Delta x_n$ - погрешности величин x_1, x_2, ..., x_n соответственно. В частности, положив в этой формуле $n = 3$, $F = S$, $x_1 = S_1$, $x_2 = S_2$, $x_3 = S_3$, получим:

$$\Delta S = \sqrt{(\frac{\partial S}{\partial S_1} * \Delta S_1)^2 + (\frac{\partial S}{\partial S_2} * \Delta S_2)^2 + (\frac{\partial S}{\partial S_3} * \Delta S_3)^2}$$

Поскольку $\frac{\partial S}{\partial S_1} = \frac{\partial S}{\partial S_2} = \frac{\partial S}{\partial S_3} = 1$ и $\Delta S_1 = \Delta S_2 = \Delta S_3 = \varepsilon_2$, а $\Delta S = \varepsilon$, то:

$$\varepsilon^2 = 3\varepsilon_2^2$$

$$\varepsilon_2 = \frac{\varepsilon}{\sqrt{3}}$$

Для нахождения погрешности ε_1 нужно рассмотреть S как функцию от x_1, x_2 и x_3:

$$\Delta S = \sqrt{(\frac{\partial S}{\partial x_1} * \Delta x_1)^2 + (\frac{\partial S}{\partial x_2} * \Delta x_2)^2 + (\frac{\partial S}{\partial x_3} * \Delta x_3)^2}$$

В качестве примера найдём $\frac{\partial S}{\partial x_1}$, остальные производные находятся аналогично:

$$\frac{\partial S}{\partial x_1} = \frac{\partial}{\partial x_1}\int_{x_3}^{x_1} f(x)dx + \frac{\partial}{\partial x_1}\int_{x_1}^{x_2} g(x)dx + \frac{\partial}{\partial x_1}\int_{x_2}^{x_3} h(x)dx = f(x_1) - g(x_1)$$

Найдя и подставив оставшиеся две производные получим:

$$\varepsilon = \varepsilon_1\sqrt{(f(x_1) - g(x_1))^2 + (g(x_2) - h(x_2))^2 + (h(x_3) - f(x_3))^2}$$

Или:

$$\varepsilon_1 = \frac{\varepsilon}{\sqrt{(f(x_1) - g(x_1))^2 + (g(x_2) - h(x_2))^2 + (h(x_3) - f(x_3))^2}}$$

Таким образом, значение погрешности в общем случае зависит от выбора x_1, x_2 и x_3. Поэтому процесс нахождение ε_1 состоит из двух частей:

1. Найти абсциссы точек пересечения с точностью ε.
2. Подставив полученные значения в формулу, получить ε_1.

Результаты экспериментов

Координаты точек пересечения кривых и площадь полученных фигур для обоих рассмотренных случаев изображены на графиках.

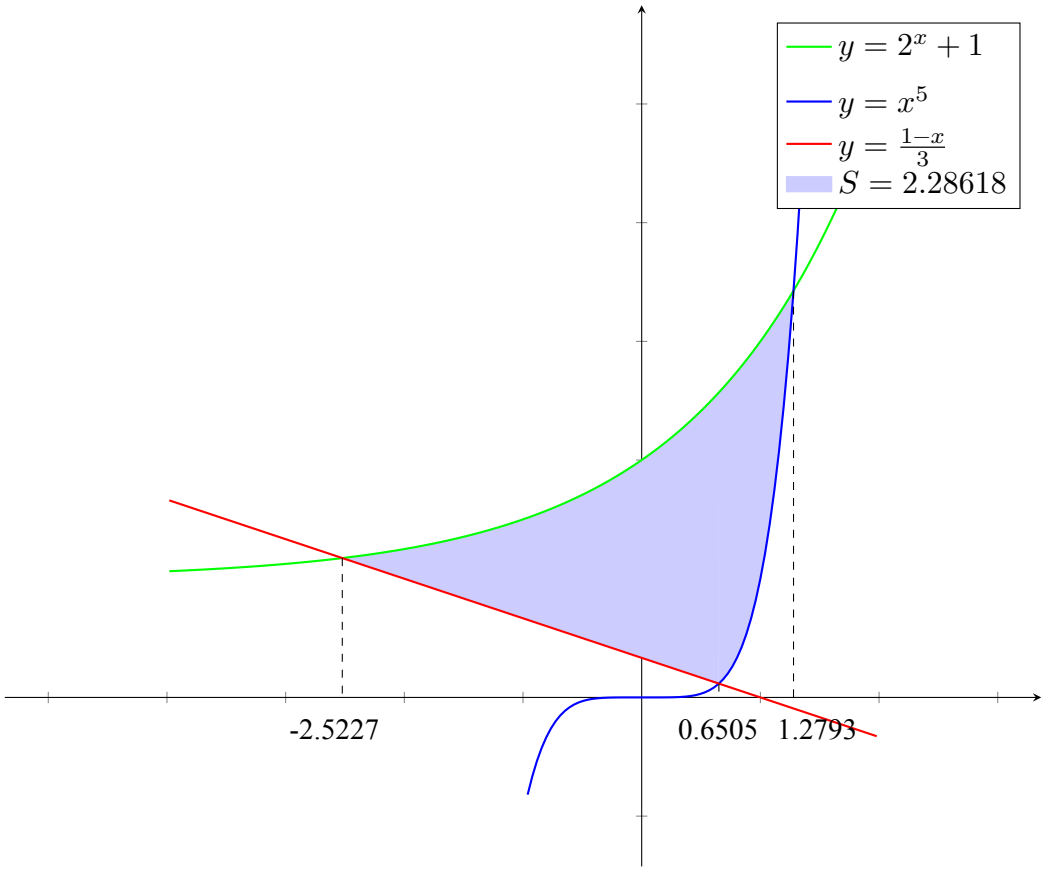


Рис. 5: Плоская фигура, ограниченная графиками заданных уравнений

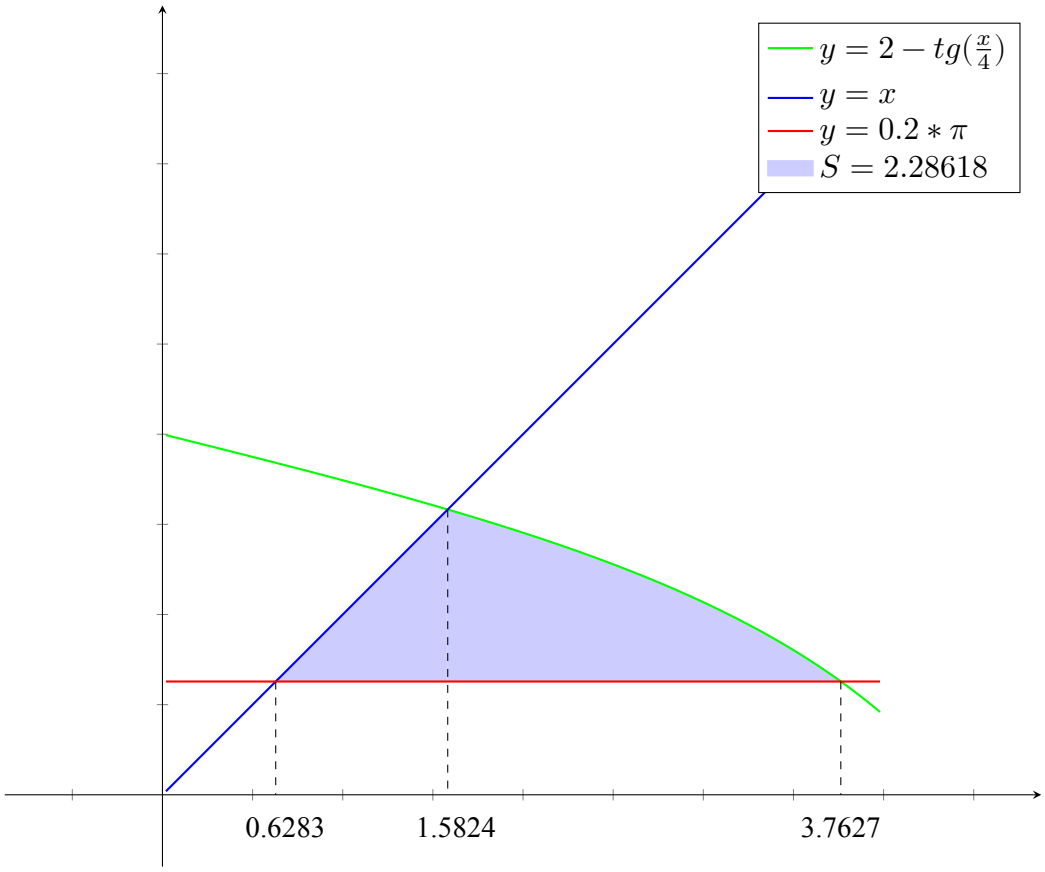
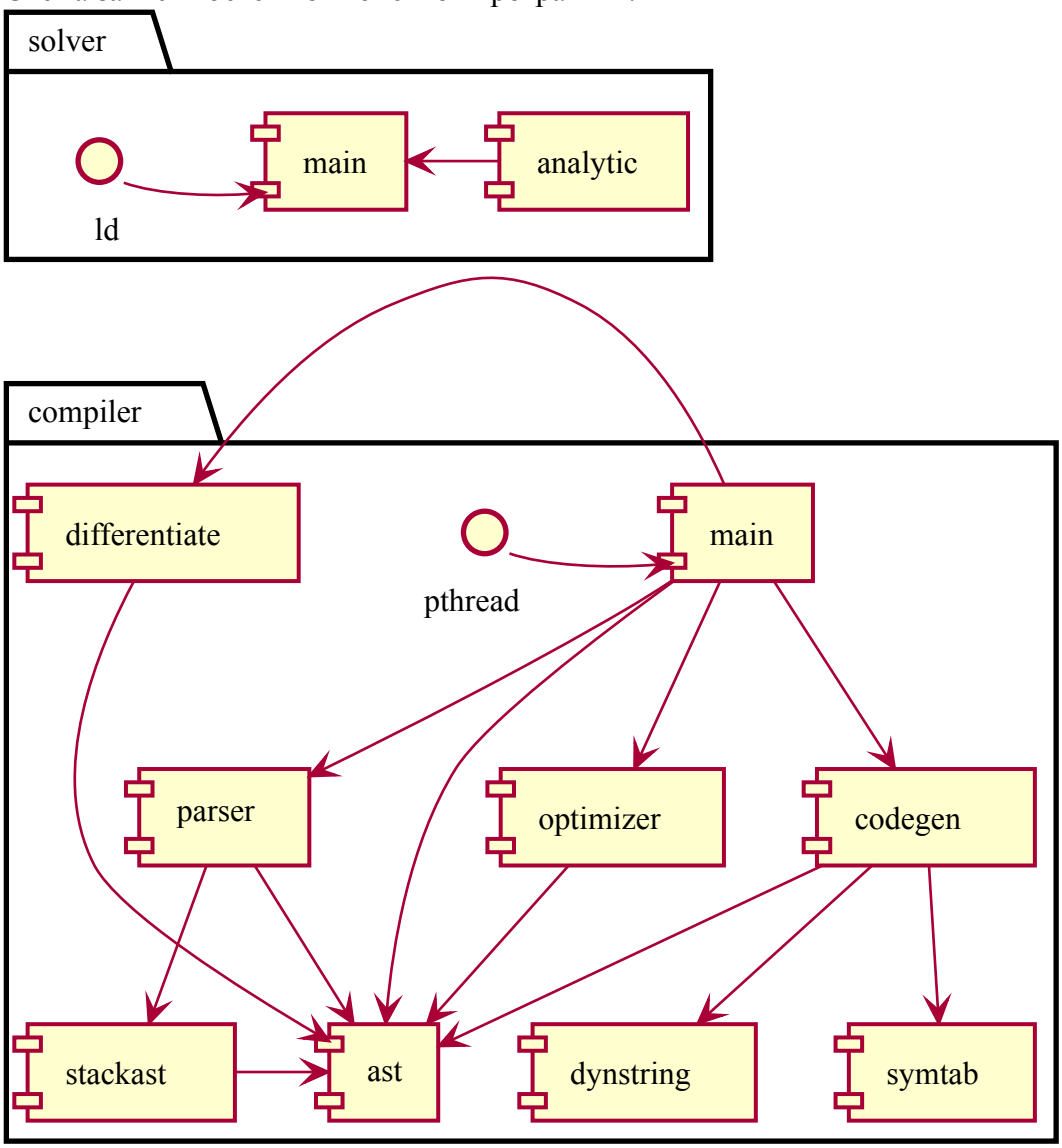


Рис. 6: Плоская фигура, ограниченная графиками заданных уравнений

Структура программы и спецификация функций

Схема зависимостей компонентов программы:



Публичный интерфейс

Решатель

Модуль analytic:
Предоставляет вычислительные методы.

```
double integrate(double (*f)(double), double a, double b, double eps);
```

Метод нахождения площади, ограниченной графиком функции $y = f(x)$, осью Ox и прямыми $x = a$ и $x = b$.

Аргументы:

- f - интегрируемая функция.

- a, b - границы отрезка интегрирования.
- eps - точность интегрирования.

```
double root(double (*f)(double), double (*g)(double),
                    double (*df)(double), double (*dg)(double),
                    double a, double b, double eps, bool iterations);
```

Метод нахождения точек пересечения кривых. Находит точку пересечения кривых, описываемых уравнениями $y = f(x)$ и $y = g(x)$ при условии, что для них выполнены на отрезке [a; b] условия применимости метода хорд и касательных.

Аргументы:

- f, g - функции, описывающие кривые, для которых нужно найти точки пересечения.
- df, dg - их производные.
- a, b - границы отрезка, на котором необходимо искать корни.
- eps - точность вычисления корня.
- iterations - отладочный аргумент, флаг, обозначающий, нужно ли выводить количество итераций метода.

Модуль main:

Главный модуль программы.

```
int main(int argc, char** argv);
```

Точка входа в программу.

Компилятор

Модуль ast:

Предоставляет основные структуры данных и функции для работы с абстрактным синтаксическим деревом(AST).

```
AST* create_tree();
```

Функция создания дерева. Возвращает указатель на созданное дерево, который должен быть удалён с помощью destroy_tree().

```
void destroy_tree(AST* tree);
```

Функция удаления дерева со всеми его потомками.

Аргумент:

- tree - указатель на дерево, выделенное с помощью create_tree(), которое нужно удалить.

```
AST *copy_ast(AST* tree);
```

Функция глубокого копирования дерева, т. е. копируется не только сам узел, но и его сыновья, внуки и т. д. Возвращает указатель на копию.

Аргумент:

- tree - указатель на дерево, которое нужно скопировать.

```
void move_ast(AST* dest, AST* src);
```

Функция перемещения узла. Содержимое узла src переносится в узел dest, при этом узел src освобождается. В отличие от предыдущей функции, ссылки на сыновей не изменяются.

Аргументы:

- dest - указатель на узел, в котоый нужно переместить.
- src - указатель на исходный узел.

```
void print_ast(AST* tree, int spacing, FILE* outfile);
```

Функция вывода дерева.

Аргументы:

- tree - указатель на дерево, которое нужно вывести.
- spacing - количество пробелов, которым выравнивается содержимое узлов.
- outfile - файл, в который выводится дерево.

```
bool equal(AST* first, AST* second);
```

Функция сравнения деревьев. Возвращает true, если деревья одинаковы.

Аргументы:

- first, second - указатели на сравниваемые деревья.

```
bool is_zero(AST *tree);
```

Функция проверки, представляет ли дерево число 0.

Аргумент:

- tree - проверяемое дерево.

```
bool is_one(AST *tree);
```

Функция проверки, представляет ли дерево число 1.

Аргумент:

- tree - проверяемое дерево.

```
bool is_number(AST *tree);
```

Функция проверки, представляет ли дерево число.

Аргумент:

- tree - проверяемое дерево.

```
bool is_variable(AST *tree);
```

Функция проверки, представляет ли дерево переменную.

Аргумент:

- tree - проверяемое дерево.

```
bool is_operator(AST *tree);
```

Функция проверки, представляет ли дерево операцию.

Аргумент:

- tree - проверяемое дерево.

Модуль codegen:

Предоставляет функцию для генерации ассемблерного листинга на основе AST.

```
char *translate(double a, double b, size_t n,
                AST** functions, char** names);
```

Функция для генерации ассемблерного листинга, содержащего константы a и b, а также n функций с заданными именами.

Аргументы:

- a, b - значения констант, которые будут записаны в листинг под именами a и b, границы отрезка для решателя.
- n - количество функций.
- functions - массив из указателей на деревья их разбора. Если деревьев больше чем n, в листинге окажутся первые n функций.
- names - массив из имён, которые будут присвоены этим функциям в листинге. Если их больше, чем n, будут взяты первые n имён, порядок имён должен соответствовать порядку AST.

Модуль differentiate:

Предоставляет функциональность для дифференцирования функции, представленной AST.

```
AST* derivative(AST *source);
```

Расчёт производной функции. Производная возвращается в качестве результата, дерево дифференцируемой функции не изменяется.

Аргумент:

- source - указатель на дерево функции, которую необходимо продифференцировать.

Модуль dynstring:

Предоставляет функции для работы с динамически расширяемыми строками.

```
string* make_string(int initial_capacity);
```

Функция создания строки. Возвращает указатель на созданную строку. Созданная строка впоследствии должна быть освобождена с помощью destroy_string().

Аргумент:

- initial_capacity - начальный размер отведённой под строку памяти.

```
void destroy_string(string* str);
```

Функция удаления строки.

Аргумент:

- str - указатель на строку, выделенную с помощью make_string().

```
string *from_cstring(char* str);
```

Функция перевода из C-строки в динамически расширяемую. Старая строка не изменяется, создаётся её копия. Возвращается указатель на созданную динамически расширяемую строку.

Аргумент:

- str - строка, которую нужно преобразовать.

```
void append(string *source, char *addend);
```

Функция конкатенации строк. К строке source приписывается строка addend с перевыделением памяти при необходимости.

Аргументы:

- source - указатель на динамически расширяемую строку, к которой нужно приписать addend.
- addend - добавляемая C-строка.

```
void append_line(int level, string *source, char *line);
```

Функция для добавления в буфер одной выровненной строки текста.

Аргументы:

- level - количество пробелов, которыми должна быть выровнена строка.
- source - указатель на буфер, в который добавляется строка.
- line - строка текста. Она не должна оканчиваться на символ переноса строки, этот символ дописывается автоматически.

Модуль main:

Главный модуль компилятора.

```
int main(int argc, char** argv);
```

Точка входа в программу. Модуль optimizer:

Предоставляет функцию для уменьшения AST.

```
void optimize(AST *tree);
```

Функция оптимизации дерева разбора. Применяется свёртка констант, а также некоторые арифметические оптимизации: оптимизация прибавления/вычитания нуля, умножения на ноль, деления нуля на функцию, вычитания и деления одинаковых поддеревьев. Модуль stackast:

Предоставляет реализацию стека из синтаксических деревьев.

```
ASTack* create_stack();
```

Функция создания стека. Созданный стек впоследствии должен быть освобождён с помощью destroy_stack().

```
void destroy_stack(ASTack* stack);
```

Функция удаления стека.

Аргумент:

- stack - указатель на удаляемый стек.

```
void clear(ASTack* stack);
```

Функция очистки стека.

Аргумент:

- stack - указатель на стек, который нужно очистить.

```
size_t size(ASTack* stack);
```

Функция, возвращающая количество элементов в стеке.

Аргумент:

- stack - указатель на стек.

```
void push(ASTack *stack, AST* tree);
```

Функция добавления дерева разбора на вершину стека.

Аргументы:

- stack - указатель на модифицируемый стек.
- tree - указатель на добавляемое в стек дерево.

```
AST *pop(ASTack* stack);
```

Функция извлечения дерева разбора с вершины стека. Возвращаемым значением служит извлечённое со стека дерево.

Аргумент:

- stack - указатель на модифицируемый стек.

Модуль symtab:

Предоставляет реализацию ассоциативного массива(используется для таблицы констант).

```
identifiers_table *create_table();
```

Конструктор таблицы идентификаторов. Возвращает указатель на таблицу, которая позже должна быть удалена с помощью destroy_table().

```
void destroy_table(identifiers_table *table);
```

Деструктор таблицы идентификаторов. Аргумент:

- table - указатель на таблицу, выделенную с помощью create_table().

```
char *lookup(identifiers_table *table, double value);
```

Функция поиска идентификатора в таблице. Возвращает имя, отвечающее данной константе или NULL, если данной константы в таблице нет. Аргументы:

- table - указатель на таблицу, в которой производится поиск.
- value - искомое значение.

```
void add_named_identifier(identifiers_table *table, char* name, double
```

Добавление в таблицу идентификатора с заданным именем. Предполагается, что идентификатора с данным значением в таблице ещё нет. Аргументы:

- table - указатель на модифицируемую таблицу.
- name - присваиваемое идентификатору имя.
- value - значение идентификатора.

```
void add_identifier(identifiers_table *table, double value);
```

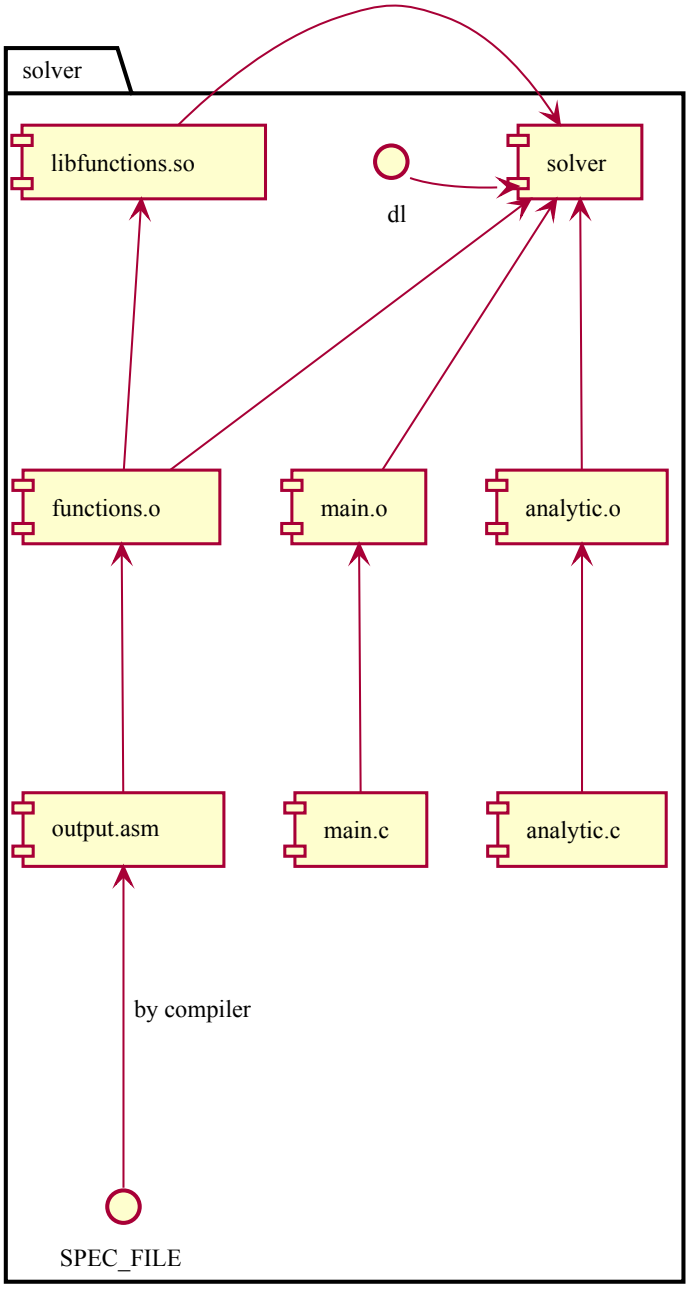
Добавление в таблицу идентификатора без имени. В таблицу он помещается с некоторым сгенерированным уникальным именем, которое можно получить впоследствии с помощью lookup(). Предполагается, что идентификатора с данным значением в таблице ещё нет. Аргументы:

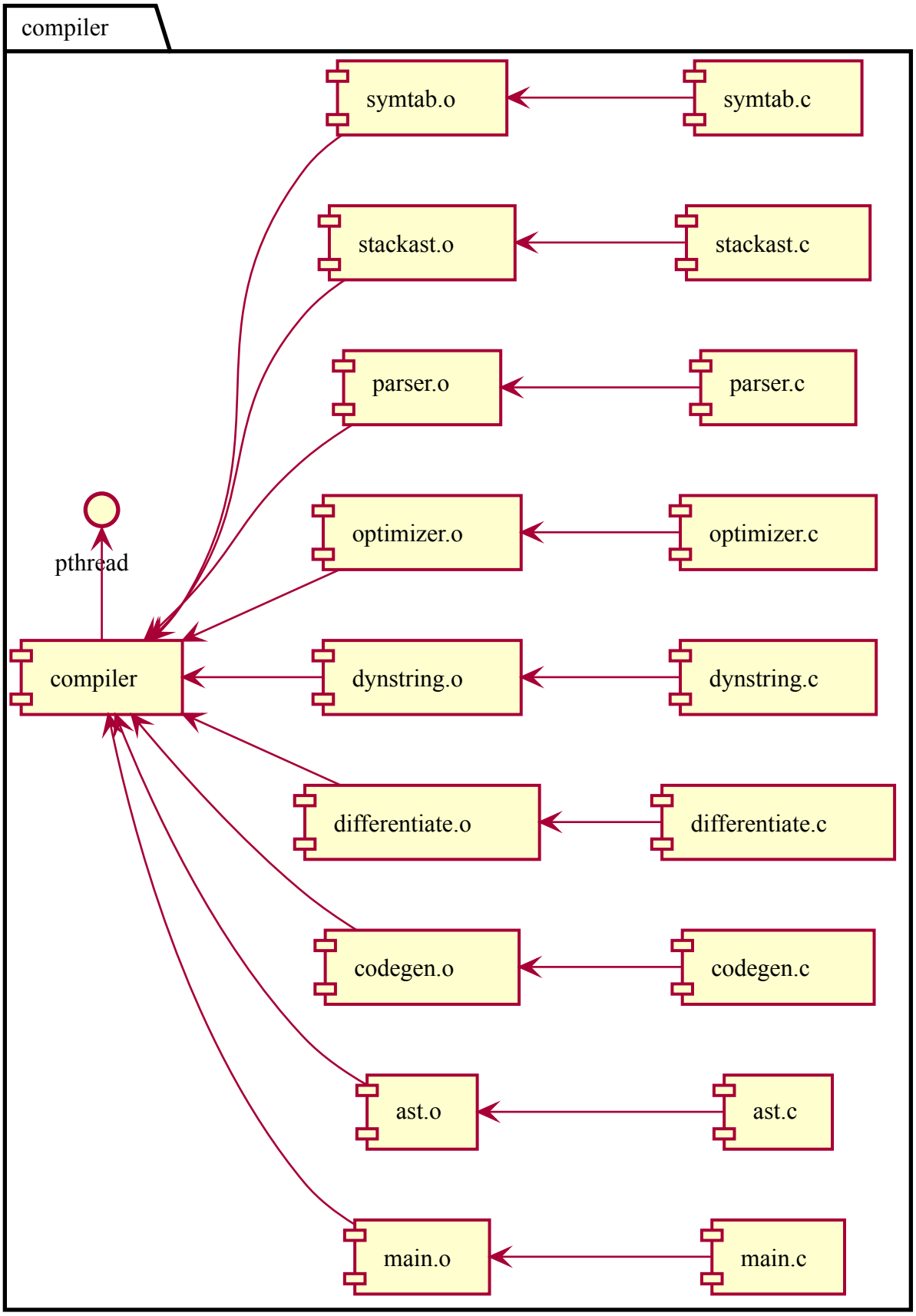
- table - указатель на модифицируемую таблицу.
- value - значение идентификатора.

Описание внутренних интерфейсов не приводится, т. к. займёт слишком много места. Кроме того, в отчёт не включается описание тестирующих функций. Кроме того, есть отдельная иерархия тестовых модулей, она будет описана позже.

Сборка программы (Make-файл)

При сборке программы сначала собирается транслятор, затем с его помощью из исходного файла получается ассемблерный листинг. Данный ассемблерный листинг собирается в объектный файл, который используется при сборке расчётной программы и в качестве динамической библиотеки позже подключается к ней во время её работы. Графически процесс сборки программы может быть представлен так:





Запустить сборку можно с помощью

make

или, с указанием входного файла для компилятора:

```
SPEC_FILE=input.txt make
```

Вместо input.txt вписать относительный путь от корня проекта к входному файлу.

Также есть возможность собрать решатель, использующий метод бинарного поиска корней вместо метода хорд и касательных. Сделать это можно с помощью:

```
SOLVE_METHOD=binary make
```

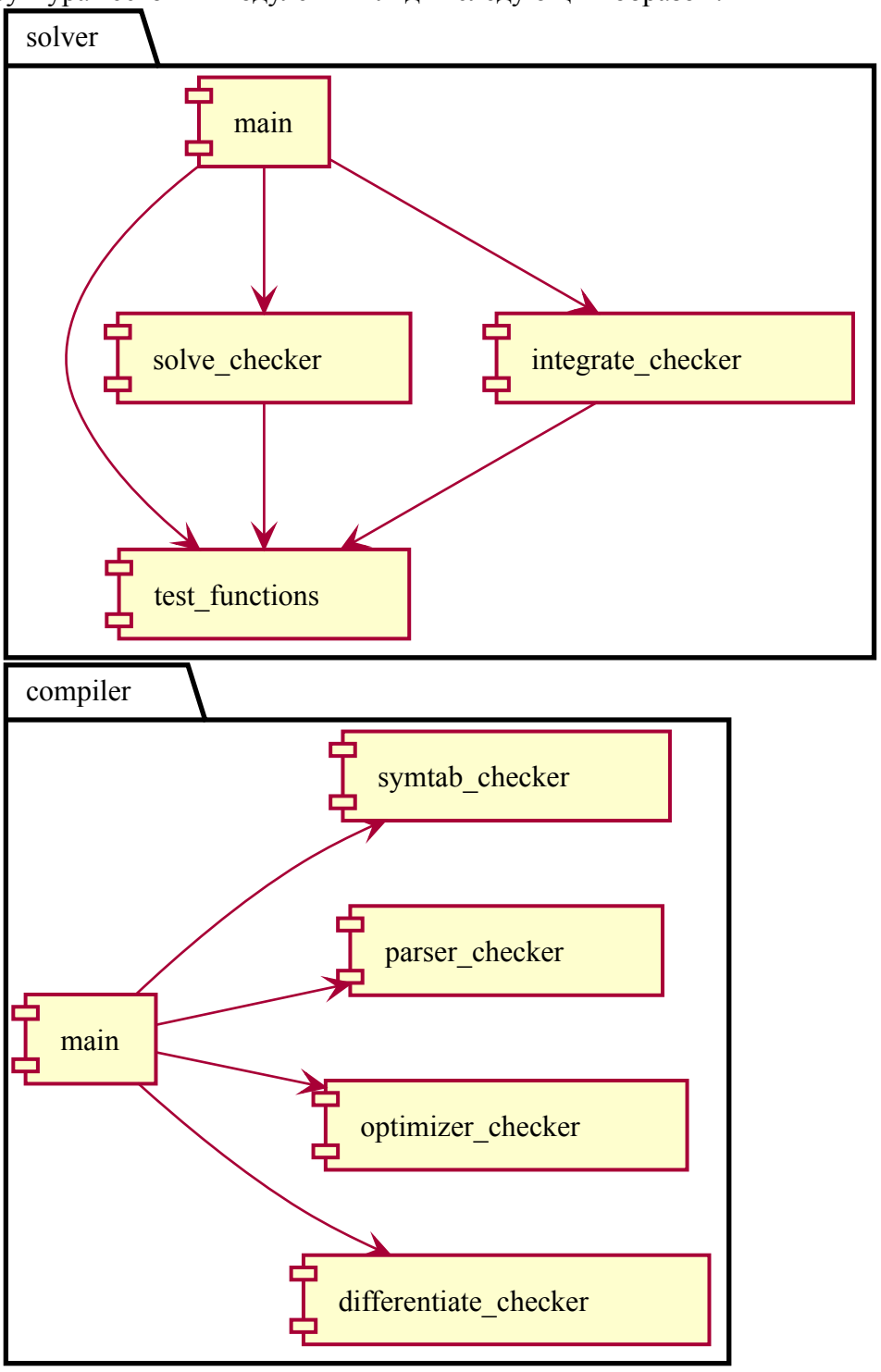
Makefile находятся в приложенном к отчёту архиве.

Отладка программы, тестирование функций

Для тестирования функций использовалась самописная система модульного тестирования. Запустить тестирование можно с помощью

```
make tests
```

Структура тестовых модулей выглядит следующим образом:



Для проверки интегрирования использовались следующие тестовые примеры:

1. $\int_0^{\pi} sin(x) = 2$
2. $\int_0^{3} x^3 = 20,25$
3. $\int_0^{5} min(2^x, x^2) \approx 40,31234$

Для проверки решения уравнений были использованы следующие уравнения:

1. $sin(x) = 0$ на отрезке $[0; 1]$. Уравнение производной $y = cos(x)$. Получен ответ $x = 0$.
2. $x^3 = 2$ на отрезке $[0; 3]$. Уравнение производной $y = 3 * x^2$. Получен ответ $x \approx 1,2599$.
3. $sin(x) = x^3$ на отрезке $[0,1; 1]$. Уравнение производной $y = cos(x) - 3 * x^2$. Получен ответ $x \approx 0.9286$.

Помимо модульного тестирования проводилась проверка программы целиком на некоторых входных данных. Кроме того была осуществлена проверка и исправление утечек с помощью valgrind.

Программа на Си и на Ассемблере

Исходные тексты программы, тестов, сборочные скрипты и пример входного файла находятся в архиве, приложенном к отчёту.

Анализ допущенных ошибок

Допущенные ошибки:

- Неверное деление при ручной трансляции производной: fdivp вместо fdivrp.
- Ошибки при работе с указателями: в функцию передавались указатели на локальные переменные.
- Многочисленные ошибки при управлении памятью: выделенная память не освобождалась.
- Была допущена ошибка при реализации метода Симпсона, выбрана неверная формула.
- В правиле Рунге для метода Симпсона не был учтён коэффициент, благодаря чему производились лишние итерации алгоритма.

Список литературы

[1] Ильин В. А., Садовничий В. А., Сендов Бл. Х. Математический анализ. Т. 1 — Москва: Наука, 1985.

[2] Тейлор Дж. Введение в теорию ошибок. -М., «Мир», 1985