

**САРОВСКИЙ ФИЗИКО-ТЕХНИЧЕСКИЙ ИНСТИТУТ
ФГАОУ ВО «НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ЯДЕРНЫЙ
УНИВЕРСИТЕТ «МИФИ»**

ФАКУЛЬТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ И ЭЛЕКТРОНИКИ

КАФЕДРА ВЫЧИСЛИТЕЛЬНОЙ И ИНФОРМАЦИОННОЙ ТЕХНИКИ

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К выпускной магистерской диссертации

На тему:

**«Разработка технологии подготовки и подключения пользовательских функций,
написанных на языке Python, в ЛОГОС-МИП»**

Студент Будникова Ирина Валерьевна

РУКОВОДИТЕЛЬ РАБОТЫ к.ф.-м.н. Холушкин Владимир Семенович

СОРУКОВОДИТЕЛЬ РАБОТЫ Надеев Александр Геннадиевич

РЕЦЕНЗЕНТ РАБОТЫ Зеленский Дмитрий Константинович

КОНСУЛЬТАНТ РАБОТЫ:

По экономическим вопросам Беляева Галина Дмитриевна

ЗАВ. КАФЕДРОЙ к.ф.-м.н. Холушкин Владимир Семенович

г. Саров
2020 г.



**САРОВСКИЙ ФИЗИКО-ТЕХНИЧЕСКИЙ ИНСТИТУТ
ФГАОУ ВО «НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ЯДЕРНЫЙ
УНИВЕРСИТЕТ «МИФИ»**

ФАКУЛЬТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ И ЭЛЕКТРОНИКИ

КАФЕДРА ВЫЧИСЛИТЕЛЬНОЙ И ИНФОРМАЦИОННОЙ ТЕХНИКИ

**УТВЕРЖДАЮ:
Зав. кафедрой ВИТ**

_____ 2020г.
" ____ " _____

ЗАДАНИЕ НА МАГИСТЕРСКУЮ ДИССЕРТАЦИЮ

Студент Будникова Ирина Валерьевна.
фамилия, имя, отчество

Направление подготовки 09.04.02. «Информационные системы и технологии»
наименование специальности

Профиль «Информационные системы и технологии в науке и приборостроении»
наименование специальности

Руководитель работы Холушкин Владимир Семенович, к.ф.-м.н., доцент
должность, уч. степень и звание, фамилия, имя, отчество

Соруководитель работы Надеев Александр Геннадиевич, начальник лаборатории
должность, уч. степень и звание, фамилия, имя, отчество

г. Саров

Наименование темы «Разработка технологии подготовки и подключения пользовательских функций, написанных на языке Python, в ЛОГОС-МИП».

Место выполнения ФГУП «РФЯЦ-ВНИИЭФ» ИТМФ отдел 0822

Исходные данные к проекту Научно-технические отчеты по модульной интеграционной платформе ЛОГОС-МИП.

Содержание магистерской диссертации Изучение предметной области, разработка технологии подготовки и подключения пользовательских функций, написанных на языке Python, в ЛОГОС-МИП в составе динамических библиотек.

Экспериментальная часть работы Реализация разработанной технологии на примере пользовательской функции, ранее написанной на языке C++.

Допуск к защите

К защите представляется:

пояснительная записка _____ страницы

Руководитель магистерской диссертации
Семенович

Холушкин Владимир

подпись

фамилия, имя, отчество

Соруководитель магистерской диссертации
Геннадиевич

Надуев Александр

подпись

фамилия, имя, отчество

Рецензент магистерской диссертации
Константинович

Зеленский Дмитрий

подпись

фамилия, имя, отчество

Студент группы ИТМ-28 Будникова И.В. допущен к защите магистерской диссертации по направлению подготовки 09.04.02 «Информационные системы и технологии», профиль «Информационные системы и технологии в науке и приборостроении»

Дата защиты .07.2020.

Заведующий кафедрой ВИТ _____

В.С. Холушкин

РЕФЕРАТ

Отчет стр., 43 рис., 28 ист.

ПОЛЬЗОВАТЕЛЬСКАЯ ФУНКЦИЯ, МОДУЛЬНАЯ
ИНТЕГРАЦИОННАЯ ПЛАТФОРМА ЛОГОС, ДИНАМИЧЕСКАЯ
БИБЛИОТЕКА, PYTHON

Пояснительная записка к диссертационной работе на степень магистра по специальности «Информационные системы и технологии» на тему: «Разработка технологии подготовки и подключения пользовательских функций, написанных на языке Python, в ЛОГОС-МИП».

Магистерская диссертация посвящена разработке технологии, позволяющей вызывать и исполнять пользовательские функции (ПФ) на языке Python внутри подключаемых к модульно-интеграционной платформе ЛОГОС-МИП динамических библиотек.

Цель работы – разработать технологию подготовки и подключения динамических библиотек с ПФ, написанными на языке Python, в ЛОГОС-МИП.

Первая глава пояснительной записки содержит теоретические разделы, в которых изучается понятие пользовательской функции и подробно описывается существующий механизм подключения ПФ, написанных на языке C++, к модульно-интеграционной платформе ЛОГОС-МИП. Также в этой главе рассматривается необходимость расширения функционала для подключения ПФ на языке Python.

Во второй главе представлено исследование существующих методов решения задачи и описана реализованная на практике технология подключения и исполнения, написанных на языке Python.

В третьей главе рассчитана себестоимость разработки данной методики и представлены результаты технико-экономической оценки.

Результатом работы стала разработанная технология создания подключаемых к ЛОГОС-МИП динамических библиотек, предоставляющих возможность вызова и исполнения ПФ на языке Python.

ОГЛАВЛЕНИЕ

ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ.....	7
ВВЕДЕНИЕ.....	9
ГЛАВА 1. ТЕОРЕТИЧЕСКАЯ ЧАСТЬ.....	12
1.1 Модульная интеграционная платформа ЛОГОС-МИП.....	12
1.1.1 Пакет программ ЛОГОС	12
1.1.2 ЛОГОС-МИП	15
1.2 Механизм подключения пользовательской функции в ЛОГОС-МИП..	16
1.2.1 Функция обратного отклика	16
1.2.2 Пользовательская функция в ЛОГОС-МИП.....	18
1.2.3 Подготовка конфигурационных файлов.....	21
1.2.4 Модуль сопряжения CCF_Bridge	25
1.2.4 Последовательность подключения ПФ в ЛОГОС-МИП	28
1.3 Пользовательские функции на языке Python	30
1.3.1 Особенности языка Python.....	30
1.3.2 Функции на языке Python.....	32
1.3.3 Использование пользовательских функций Python в ЛОГОС-МИП	35
ГЛАВА 2. ПРАКТИЧЕСКАЯ ЧАСТЬ	37
2.1 Требования к разрабатываемой технологии подключения ПФ на языке Python	37
2.2 Исследование существующих технологий.....	38
2.2.1 Создание dll-библиотеки средствами языка Python	38
2.2.2 Использование CPython или Cython	39
2.2.3 Использование Python C API.....	41
2.2.4 Обоснование выбора решения.....	41
2.3 Python C API	43
2.4 Практическая реализация технологии	47
2.4.1 Связь с интегратором	48
2.4.2 С-оболочка.....	49
2.4.3 Библиотека PyToCpr.h	51

2.4.4 Сборка dll-библиотеки.....	55
2.5 Пример реализации технологии	56
ЗАКЛЮЧЕНИЕ	63
СПИСОК ИСПОЛЬЗУЕМОЙ ЛИТЕРАТУРЫ	65
ПРИЛОЖЕНИЕ 1. ЛИСТИНГ ПРОЕКТА.....	69

ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

API (Application Programming Interface)	– интерфейс программирования приложения
Callback	– функция обратного вызова
SPEC-файл	– файл спецификации XML формата, используемый Интегратором для получения информации о различных модулях
Интегратор	– программа, имеющая модульную структуру и обеспечивающая сквозное согласованное управление объектами пакета программ «ЛОГОС» для обеспечения расчетов задач оптимизации и задач параметрических исследований
Интерпретатор	– программа, выполняющая построчный анализ, обработку и исполнение исходного кода программы или запроса
Коннектор пользовательской функции	– указатель на функцию, с помощью которого производится вызов пользовательской функции, в которой может быть сохранен адрес любой пользовательской функции с соответствующим интерфейсом
МИП (ЛОГОС-МИП)	– модульная интеграционная платформа «ЛОГОС Платформа»
Модуль сопряжения	– модуль, обеспечивающий загрузку динамической библиотеки ПФ и подключение ПФ к коннектору в авторской методике в момент инициализации расчета задачи
ОПК	– оборонно-промышленный комплекс

ПО

– программное обеспечение

Пользовательская функция (ПФ)

– callback функция разработанная пользователем, вызываемая авторской методикой в процессе расчета физической задачи, служит для передачи методике измененных пользователем параметров расчета задачи

Словарь Python

– неупорядоченная коллекция произвольных объектов с доступом по ключу

ВВЕДЕНИЕ

В современном цифровом мире математическое моделирование является активно развивающимся методом научного познания, который широко применяется в исследованиях в различных областях науки и в решении прикладных задач. Эта методология основана на изучении свойств и характеристик объектов посредством исследования их математических аналогов (моделей). Данный этап развития математического моделирования характеризуется использованием компьютеров и суперкомпьютеров, специального программного обеспечения и новых методов вычислительной математики.

В РФЯЦ-ВНИИЭФ разрабатывается многофункциональный пакет программ инженерного анализа и суперкомпьютерного моделирования ЛОГОС [1], который позволяет решать задачи математического 3D-моделирования в разных сферах науки и техники. В ЛОГОС реализованы передовые технологии математического моделирования; пакет предоставляет широкий круг численных методов и физико-математических моделей для расчета процессов.

В отличие от многих пакетов программ, разрабатываемых на предприятиях Росатома, ЛОГОС предназначен не только для решения внутренних задач Госкорпорации, но и готовится к выходу на коммерческий рынок Российской Федерации, а в будущем, и всего мира [2]. Поэтому разработка пакета ЛОГОС ведется с учетом нужд различных пользователей, работающих в государственных и частных организациях, использующих в процессе моделирования разные операционные системы и языки программирования.

В состав пакета программ ЛОГОС входит модульная интеграционная платформа ЛОГОС-Платформа (далее ЛОГОС-МИП). Эта уникальная разработка РФЯЦ-ВНИИЭФ обеспечивает взаимодействие функциональных блоков ЛОГОС и специализированных автономных математических методик

[3], расчетных модулей, модулей тестирования и верификации, написанных пользователями пакета для решения каких-либо конкретных задач.

Одной из особенной ЛОГОС-МИП является реализация механизма подключения к математическим методикам дополнительных модулей, представленных в виде пользовательских функций (ПФ) и позволяющих изменять данные в процессе моделирования. Таким образом, каждый пользователь пакета программ ЛОГОС может расширять существующие математические методики дополнительным функционалом для собственных нужд.

В настоящий момент механизм подготовки и подключения ПФ позволяет подключать ПФ, написанные исключительно на языках C/C++. Данный подход связан с рядом трудностей, т.к. исполняемый код на C++ требует процесса компиляции и зависит от ОС пользователя, что затрудняет удаленную поддержку в случае возникновения ошибок в работе программы. Кроме того, в последнее время активно развиваются и применяются в математическом моделировании другие языки программирования, наиболее популярным из которых является Python. Таким образом, существующий механизм подключения ПФ к ЛОГОС-МИП ограничивает сегмент разработчиков, которые могли бы выбрать и использовать пакет программ ЛОГОС в своей работе.

Тема моей исследовательской работы является актуальной, так как разрабатываемая мной технология подключения ПФ на языке Python расширит функционал ЛОГОС-МИП, а также позволит устранить зависимость библиотек ПФ от особенностей ОС и привлечь широкий круг разработчиков на набирающем популярность языке Python.

Объект исследования – пользовательские функции, подключаемые к ЛОГОС-МИП.

Предмет исследования – исследование технологии вызова пользовательских функций.

Цель данной работы – разработать технологию подготовки и подключения динамических библиотек, содержащих ПФ, написанные на языке Python, в ЛОГОС-МИП.

Исходя из поставленной цели, необходимо решить следующие задачи:

- 1) изучить понятие пользовательских функций;
- 2) изучить пакет программ ЛОГОС и модульную интеграционную платформу ЛОГОС-МИП;
- 3) изучить научно-технические отчеты по ЛОГОС-МИП;
- 4) изучить существующий механизм подключения ПФ к ЛОГОС-МИП;
- 5) исследовать пути и средства решения поставленной задачи;
- 6) выбрать средства решения, разрешенные для использования в ИТМФ РФЯЦ-ВНИИЭФ;
- 7) разработать технологию вызова и исполнения ПФ на языке Python внутри подключаемой к ЛОГОС-МИП dll-библиотеки;
- 8) на примере одной из существующих динамических библиотек с подключаемыми пользовательскими функциями на языке C++ реализовать подключение аналогичной по функционалу ПФ на языке Python.

ГЛАВА 1. ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

1.1 Модульная интеграционная платформа ЛОГОС-МИП

1.1.1 Пакет программ ЛОГОС

В настоящее время в РФЯЦ-ВНИИЭФ разрабатывается многофункциональный пакет программ инженерного анализа и суперкомпьютерного моделирования «ЛОГОС», который предназначен для решения широкого круга задач математического 3D-моделирования аэро-, гидро-, газодинамики, тепломассопереноса, турбулентного перемешивания, излучения, статической и динамической прочности, разрушения и деформации [1].

В «ЛОГОС» реализованы передовые физико-математические модели для расчета процессов. Отличительной чертой пакета является использование неструктурированных сеток, состоящих из произвольных многогранников. Функционал «ЛОГОС» охватывает более 70% основных задач промышленности, решаемых при проектировании высокотехнологичных изделий [4]. В настоящий момент пакет применяется во многих наукоемких отраслях, таких как авиастроение, ракетно-космическая отрасль, атомная энергетика, автомобилестроение, судостроение и т.д. По оценкам компаний, которые используют в своих расчетах пакет «ЛОГОС», результаты, полученные при моделировании инструментами отечественного пакета и его зарубежных коммерческих конкурентов, таких как ANSYS [5] и Abaqus, хорошо согласуются между собой и находятся в допустимом диапазоне отклонений.

Разработка пакета программ «ЛОГОС» началась в 2005 году для внутренних нужд Росатома и подведомственных ему предприятий. В 2009 году Комиссия при президенте по модернизации и технологическому развитию утвердила программу «Развитие суперкомпьютеров и грид-технологий» [6], в состав которой вошли проекты по созданию отечественных супер-ЭВМ и отечественного ПО для имитационного моделирования процессов на супер-ЭВМ. Проект «ЛОГОС» стал частью

правительственной программы, и подразделения РФЯЦ-ВНИИЭФ начали работы по расширению функционала для различных предприятий и отраслей российской науки. В 2010 году была представлена первая версия отечественного пакета программ «ЛОГОС» [2], которая позволила в несколько сотен раз ускорить время проведения отдельного расчета и расширила возможности проведения многовариантных расчетов.

С 2012 года пакет программ «ЛОГОС» находится в стадии опытной эксплуатации. В 2016 году на предприятия ОПК и других высокотехнологичных отраслей промышленности были проданы первые лицензионные копии пакета «ЛОГОС». В декабре 2018 года был представлен первый коммерческий цифровой продукт «ЛОГОС Аэро-Гидро», выведенный Росатомом на российский рынок [4]. В 2019 году состоялся второй коммерческий релиз продукта «ЛОГОС Тепло».

На рисунке 1 представлен классический пакет программ ЛОГОС [3].





 ЛОГОС АЭРО-ГИДРО	АЭРОДИНАМИКА, ГИДРОДИНАМИКА, ТУРБУЛЕНТНОЕ ПЕРЕМЕШИВАНИЕ
 ЛОГОС ТЕПЛО	ТЕПЛОМАССОПЕРЕНОС, ИЗЛУЧЕНИЕ
 ЛОГОС ПРОЧНОСТЬ	СТАТИЧЕСКАЯ И ДИНАМИЧЕСКАЯ ПРОЧНОСТЬ, РАЗРУШЕНИЕ И ДЕФОРМАЦИЯ
 ЛОГОС ПРЕПОСТ	ПОДГОТОВКА НАЧАЛЬНЫХ ДАННЫХ, ОБРАБОТКА РЕЗУЛЬТАТОВ МОДЕЛИРОВАНИЯ

Рисунок 1 – Продукты классического пакета программ «ЛОГОС»

1) ЛОГОС Аэро-Гидро - высокоточный отечественный инструмент для решения задач течения жидкости и газа, многофазных и реагирующих

потоков, а также акустики при проектировании высокотехнологичных промышленных изделий [1].

2) ЛОГОС Тепло - высокоточный отечественный инструмент для решения задач теплопроводности, излучения и фазовых переходов в твердых телах и неподвижных средах при проектировании высокотехнологичных промышленных изделий [1].

3) ЛОГОС Прочность – высокоточный отечественный инструмент для решения задач статической, динамической и вибрационной прочности [7].

4) ЛОГОС Препост - препостпроцессор, который предназначен для импорта и обработки CAD/FEM/CFD-моделей и генерации поверхностных и объемных сеток; имеет визуальную среду для подготовки расчетной модели и интерактивную систему инженерной визуализации [8].

На рисунке 2 представлен дополнительный пакет программ ЛОГОС [3].

	МОДЕЛИРОВАНИЕ РАЗЛИЧНЫХ ФИЗИЧЕСКИХ ПРОЦЕССОВ В РЕАКТОРНЫХ УСТАНОВКАХ ПРИ НОРМАЛЬНЫХ РЕЖИМАХ ЭКСПЛУАТАЦИИ И ПРИ АВАРИЙНЫХ СИТУАЦИЯХ
	3D-МОДЕЛИРОВАНИЕ ФИЛЬТРАЦИИ ЖИДКОСТЕЙ И ГАЗОВ В ГЕОЛОГИЧЕСКИХ ПЛАСТАХ
	ГАЗОДИНАМИЧЕСКИЕ ТЕЧЕНИЯ МНОГОКОМПОНЕНТНОЙ СРЕДЫ С СОПУТСТВУЮЩИМИ ПРОЦЕССАМИ (ДЕТОНАЦИЯ, УПРУГОПЛАСТИКА И Т.Д.)
	ПРОГРАММНО-МЕТОДИЧЕСКИЙ ИНСТРУМЕНТ ПОДДЕРЖКИ ПРИНЯТИЯ РЕШЕНИЙ ПО ПОРТФЕЛЮ ПРОЕКТОВ И МЕР ГОСУДАРСТВЕННОЙ ПОДДЕРЖКИ

Рисунок 2 – Продукты дополнительного пакета программ «ЛОГОС»

Важным компонентом пакета программ «ЛОГОС» является модульная интеграционная платформа для обеспечения возможности проведения связанных и сопряженных расчетов «ЛОГОС Платформа» (ЛОГОС-МИП).

1.1.2 ЛОГОС-МИП

Модульная интеграционная платформа (МИП), входящая в состав пакета программ «ЛОГОС», предназначена для решения ряда задач математического моделирования [9]., в том числе для подготовки и проведения:

- 1) связанных расчетов комплексных мультидисциплинарных задач математического моделирования;
- 2) задач оптимизации и параметрических исследований.

Эта уникальная отечественная разработка позволяет расширять функционал пакета программ ЛОГОС за счет подключения сторонних специализированных модулей [3], написанных пользователями для собственных нужд (рисунок 3).

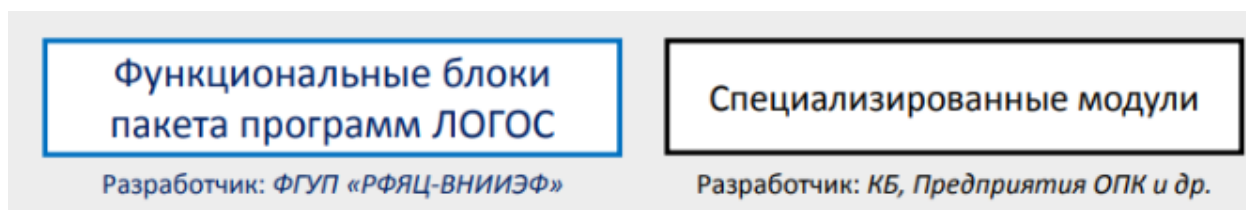


Рисунок 3 - Интерфейсное программное обеспечение пакета программ ЛОГОС

ЛОГОС-МИП позволяет объединить уникальные знания, накопленные в рамках ОПК и частных фирм, работающих в научной сфере, в единую систему в рамках многофункционального пакета программ ЛОГОС. Платформа обеспечивает поддержку и сопровождения широкого круга авторских методик математического моделирования.

ЛОГОС-МИП позволяет интегрировать три типа специализированных авторских модулей [3], которые представлены на рисунке 4.

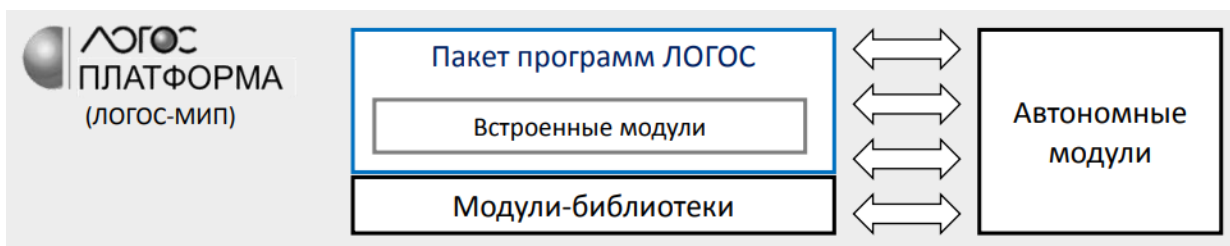


Рисунок 4 – Типы специализированных модулей в ЛОГОС-МИП

- 1) Встроенные модули – авторская методика реализуется в рамках исходных текстов ЛОГОС.
- 2) Модули-библиотеки – интегрируются как внешние библиотеки, использующие структуры данных ЛОГОС.
- 3) Автономные модули – связываются с пакетом программ ЛОГОС через единые форматы файлов и среду обмена данными.

1.2 Механизм подключения пользовательской функции в ЛОГОС-МИП

1.2.1 Функция обратного отклика

Обратный вызов – это ссылка на исполняемый код или фрагмент исполняемого кода, который передается в качестве аргумента другому коду [10].

Функция обратного вызова, или *callback* – это функция, которая передается другой функции (высшего порядка) в качестве параметра для ее вызова или исполнения. В тот момент, когда некоторая функция высшего порядка, получившая на вход *callback*, завершила свою работу или произошло определенное событие, остановившее ее исполнение, она вызывает функцию обратного вызова.

Таким образом, *callback* может быть вызвана в определенный момент времени при выполнении некоторых заданных условий – в точке останова. Например, таким условием может быть щелчок по определенной кнопке графического интерфейса или выполнение действия по таймеру.

Существуют два типа обратных вызовов [11]:

- 1) Синхронный (блокирующий) – останавливает исполнение вызывающей функции до завершения работы *callback* и возвращения управления функции высшего порядка. Например, синхронным является

обратный вызов формы, которая позволяет выбрать рисунки для загрузки в приложение, но при этом блокирует его работу до момента закрытия этой формы.

2) Асинхронный (отложенный) – позволяет вызывать callback и возвращать управление функции высшего порядка, не дожидаясь завершения работы callback. Асинхронным обратным вызовом может быть загрузка указанного изображения, разрешающая продолжать параллельную работу с приложением.

Функция обратного вызова может быть описана как библиотечная функция и подключаться к основной программе в составе статических или динамических библиотек (рисунок 5). При достижении определенного состояния функция высшего порядка обращается к библиотеке для вызова определенной callback, причем некоторые языки поддерживают конструкции с выбором аргументов callback в зависимости от состояния вызывающей функции. Например, один обратный вызов срабатывает в том случае, если эта функция выполняется успешно, другой - в том случае, если она выдает определенную ошибку.

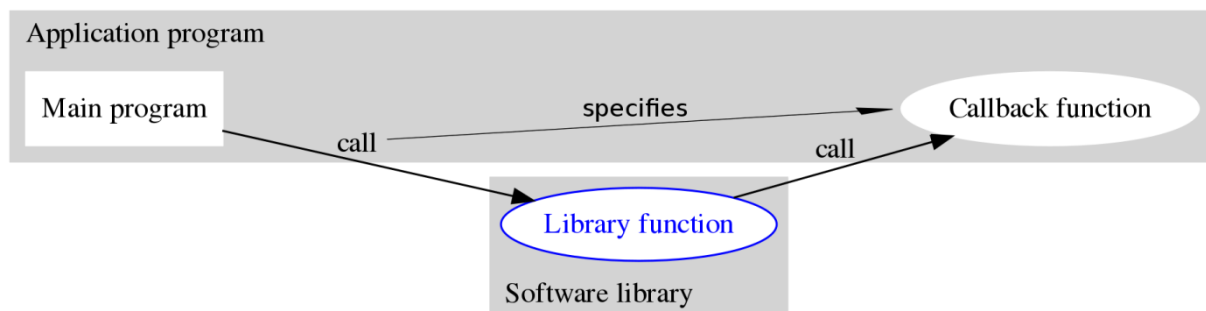


Рисунок 5 – Вызов callback из библиотеки

В языке C обратный вызов реализуется с помощью указателя на функцию, который может быть передан в качестве аргумента другой функции.

C++ позволяет объектам предоставлять собственную реализацию операции вызова функции. Standard Template Library принимает эти объекты

(называемые функторы), а также указатели на функции, в качестве параметров различных полиморфных алгоритмов.

Многие динамические языки, такие как JavaScript , Lua , Python , Perl и PHP, позволяют передавать функциональный объект [12].

Пример реализации обратных вызовов на языке C представлен на рисунке 6.

```
#include <stdio.h>
#include <stdlib.h>

void PrintTwoNumbers ( int ( * numberSource ) ( void ) ) {
    int val1 = numberSource ();
    int val2 = numberSource ();
    printf ( "% d и % d \ n " , val1 , val2 );
}

int overNineThousand ( void ) {
    return ( rand () % 1000 ) + 9001 ;
}

int valueOfLife ( void ) {
    return 42 ;
}

// PrintTwoNumbers () с двумя различными обратными вызовами.
int main ( void ) {
    PrintTwoNumbers ( & rand );
    PrintTwoNumbers ( & overNineThousand );
    вернуть 0 ;
}
```

Рисунок 6 – Пример функций обратного вызова на языке C

Функция высшего порядка PrintTwoNumbers(...) принимает в качестве аргумента указатель на некоторую функцию и вызывает ее исполнение. Функции overNineThousand() и valueOfLife() являются callback – возможными обратными вызовами для получения числовых параметров на печать в функции PrintToNumbers(...).

1.2.2 Пользовательская функция в ЛОГОС-МИП

Пользовательской функцией называется функция обратного вызова (callback), разработанная пользователем с целью расширения стандартных возможностей математической методики [13].

Механизм ПФ предоставляет возможность изменить поведение расчетного модуля путем передачи новых значений параметров в процессе расчета задачи.

Существуют два способа подключения библиотек пользовательских функций [14]:

1) Статическое связывание – связь устанавливается при сборке приложения, содержащего код методики и библиотеку ПФ.

2) Динамическое связывание – связь устанавливается непосредственно в момент выполнения расчета задачи.

Одной из особенностей модульной интеграционной платформы ЛОГОС является поддержка механизма для динамического подключения ПФ и внесения изменений в процесс моделирования при решении ряда задач в рамках пакета программ ЛОГОС [13]. На текущий момент реализация динамического связывания библиотеки ПФ и методики позволяет подключать к расчету ПФ с произвольными именами, расположенные в разных динамических библиотеках, а также использовать графический интерфейс для настройки подключения ПФ при решении конкретной задачи.

Применение механизма ПФ ЛОГОС-МИП позволяет расширить возможности расчетного модуля, например: позволяет инициализировать различные параметры в момент начала расчета или изменять значения различных параметров в процессе расчета.

В основе технологии подключения пользовательских функций ЛОГОС-МИП лежит механизм callback. В системе реализованы коннекторы ПФ - указатели на функцию, с помощью которых производится вызов ПФ. По умолчанию они имеют значение нулевых указателей, но в случае подключения ПФ, инициализируются как указатели на конкретные ПФ с соответствующим интерфейсом. Обобщенная схема механизма callback при подключении ПФ в ЛОГОС-МИП представлена на рисунке 7.

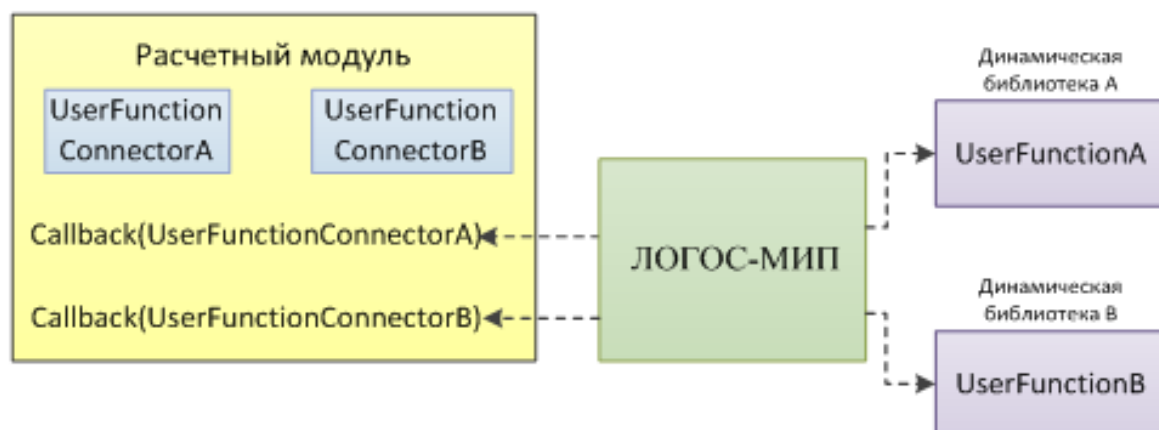


Рисунок 7 – Обобщенная схема механизма callback в ЛОГОС-МИП

В определенных точках расчета задач инициализированные коннекторы могут вызвать на исполнение требуемые пользовательские функции, а затем вернуть управление расчетному модулю. Также предусмотрены возможности ветвления задач, реализующих обратный вызов в зависимости от наличия или отсутствия подключения конкретных ПФ.

Связь библиотек ПФ и математических методик обеспечивает модуль сопряжения CCF_Bridge [14]. Он представляет собой отдельную статическую библиотеку, которая связывается с исполняемыми файлами расчетных модулей на стадии сборки приложения. Модуль сопряжения загружает динамические библиотеки с ПФ, информацию о подключаемых к приложению ПФ и связывает ПФ с соответствующими коннекторами.

ЛОГОС-МИП предоставляет возможность настройки подключения ПФ в единой графической оболочке Интегратор. Он позволяет связывать ПФ с коннекторами и готовить файл инициализации коннекторов, который потом передается модулю сопряжения.

Общая схема подготовки и подключения библиотек ПФ к математическим методикам представлен на рисунке 8.

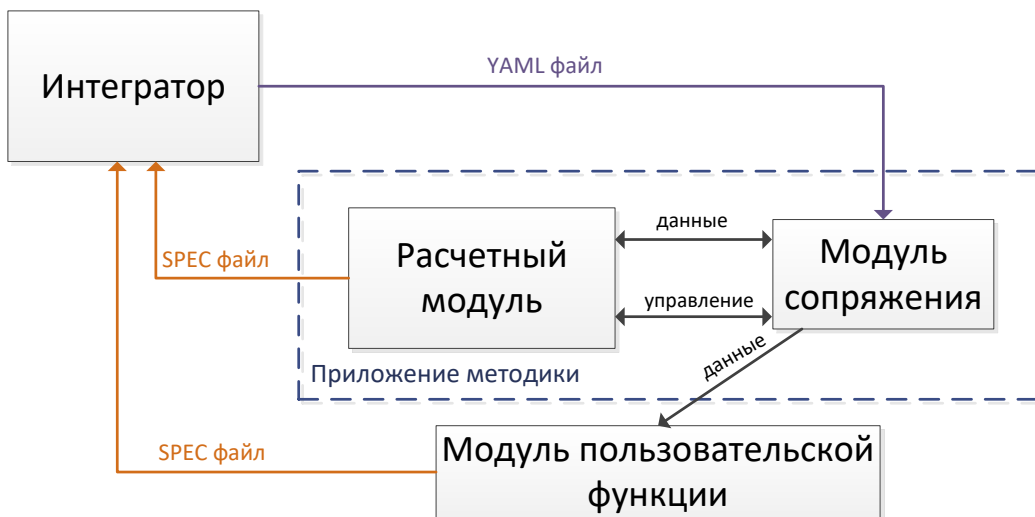


Рисунок 8 – Общая схема взаимодействия математической методики и модуля сопряжения при реализации механизма ПФ

1.2.3 Подготовка конфигурационных файлов

Для возможности настройки подключения ПФ к подготовленной задаче, Интегратору необходимо предоставить информацию о поддерживаемых методикой ПФ, а так же о доступных для подключения библиотек ПФ. Данная информация оформляется в виде файлов спецификаций (SPEC-файлов) XML-формата с описаниями API коннекторов ПФ (рисунок 9).

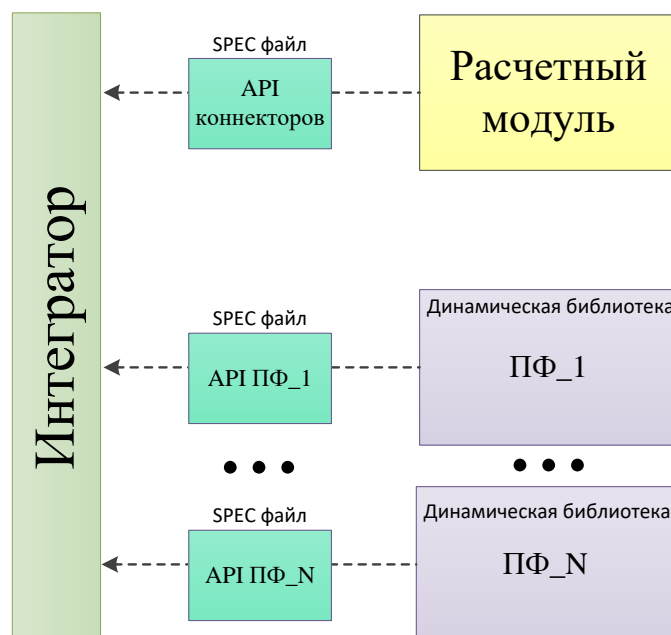


Рисунок 9 - Передача SPEC-файлов в Интегратор

SPEC-файлы могут быть подготовлены при помощи специальной утилиты `spesgen`, входящей в состав ЛОГОС-МИП и основанной на использовании специальных описательных меток системы документирования исходных текстов Doxygen [15]. Алгоритм создания SPEC-файлов представлен на рисунке 10.

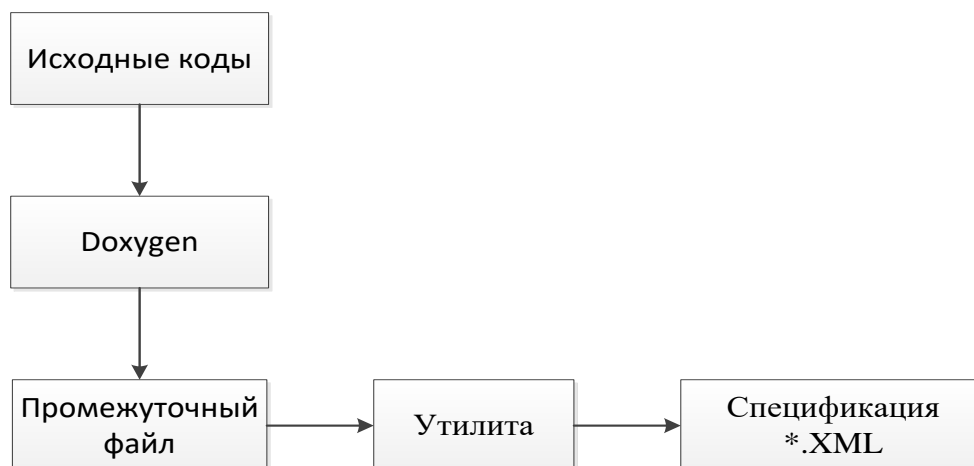


Рисунок 10 – Алгоритм создания SPEC-файлов

Рисунок 11 иллюстрирует пример использования описательных меток системы Doxygen при разработке ПФ.

```

/** @brief Скорость массового уноса
 * @param fT значение температуры в центре ячейки
 * @param centerX координаты центра ячейки
 * @param centerY координаты центра ячейки
 * @param centerZ координаты центра ячейки
 * @param normalX координаты проекции центра ячейки на нормаль
 * @param normalY координаты проекции центра ячейки на нормаль
 * @param normalZ координаты проекции центра ячейки на нормаль
 * @user_function
 */
double uf_ablation_rate( double fT,
                        double centerX,
                        double centerY,
                        double centerZ,
                        double normalX,
                        double normalY,
                        double normalZ)
{

```

Рисунок 11 – Теги Doxygen ПФ `ablationRate`

В SPEC-файл ПФ содержит следующую информацию:

- 1) Имя библиотеки, в которой находится ПФ.
- 2) Имя ПФ;

- 3) Описание ПФ;
- 4) Интерфейс ПФ и описание к нему.

В SPEC-файл расчетного модуля записывается информация о коннекторах ПФ:

- 1) Имя коннектора ПФ.
- 2) Описание коннектора ПФ.
- 3) Интерфейс коннектора ПФ и описание к нему.

На основе SPEC-файлов Интегратор предоставляет пользователю возможность конфигурировать соединение расчетного модуля и ПФ. Результатом является YAML-файл конфигурации `scf.yaml` с описанием связей коннекторов и соответствующих ПФ.

Файл `scf.yaml` содержит следующие поля для каждой пары соответствия коннектора и ПФ [14]:

- 1) `dll_name` – имя динамической библиотеки, в которой расположена ПФ;
- 2) `connector_name` – имя коннектора ПФ;
- 3) `function_name` – имя ПФ.

Пример файла конфигурации для связи расчетного модуля с ПФ `ablationRate` представлен на рисунке 12.

```
bridges:
- user_defined_function_connectors:
  - dll_name: ablationRateLib
    connector_name: ablation_rate
    function_name: uf_ablation_rate
  interfaces: ~
  lightweight_solvers: ~
  invoke_points: ~
```

Рисунок 12 – `scf.yaml` для `ablationRate`

Конфигурирование соединения коннекторов ПФ расчетного модуля и ПФ в Интеграторе происходит в специальном диалоге настройки ПФ, в котором пользователь из предложенного списка может выбрать соответствие коннектора ПФ расчетного модуля и доступной ПФ.

Список доступных коннекторов ПФ и ПФ формируется исходя из информации, полученной из SPEC-файлов доступных расчетных модулей и ПФ.

Для каждой задачи пользователь выбирает соответствие коннекторов ПФ и ПФ [13]. Пример графического интерфейса настройки подключения пользовательских функций к расчетному модулю показан на рисунке 13.

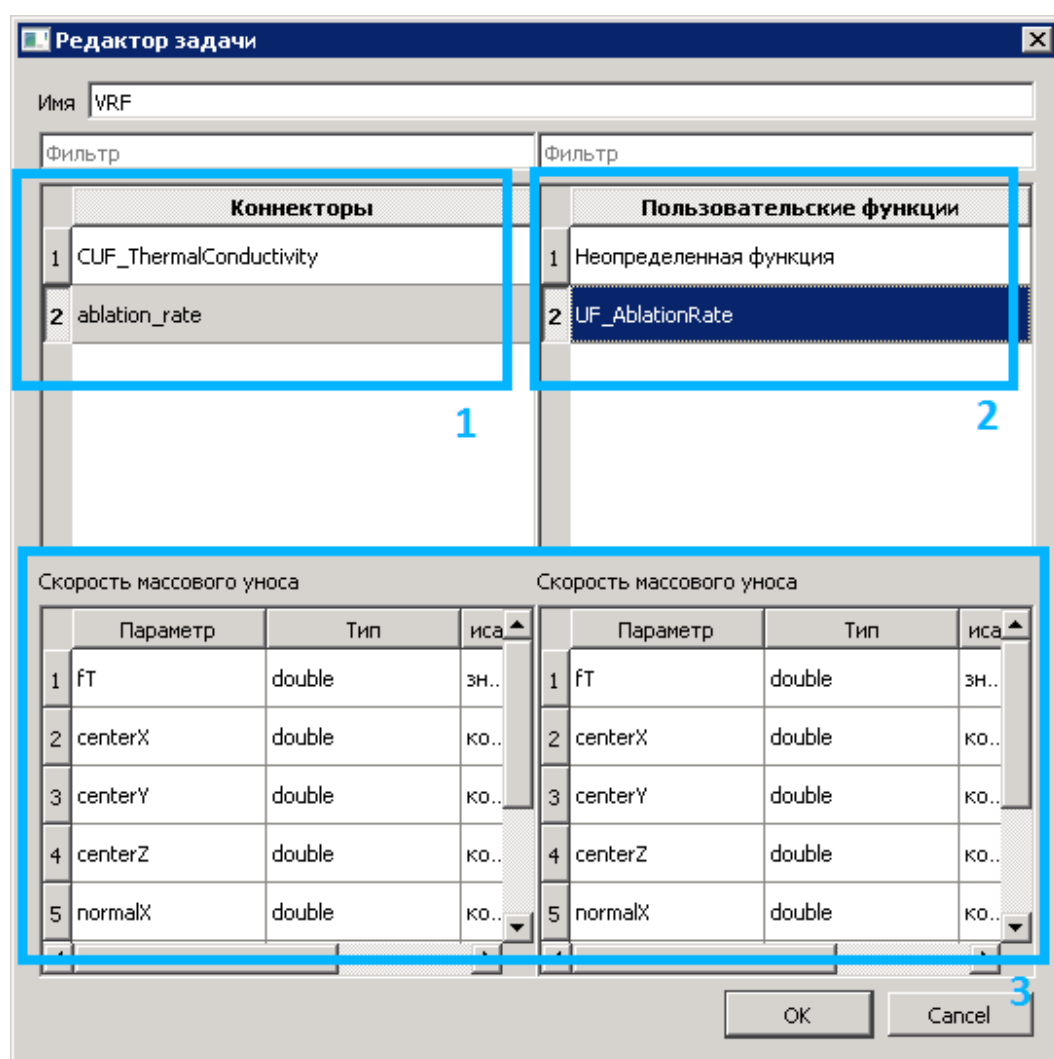


Рисунок 13 – Внешний вид диалога настройки пользовательских функций

Таблицы коннектора (1) формируются Интегратором на основе SPEC-файлов расчетных модулей методики.

Таблицы подключаемых к коннекторам ПФ (2) определяются Интегратором для каждого коннектора на основе SPEC-файлов ПФ.

В таблицах (3) отражены описания ПФ и коннекторов.

Итоговый YAML-файл передается в модуль сопряжения для дальнейшей инициализации коннекторов и связывания ПФ с расчетным модулем.

1.2.4 Модуль сопряжения CCF_Bridge

Модуль сопряжения CCF_Bridge выполняет следующие задачи по использованию ПФ в математических методиках:

- 1) Загружает подготовленный в Интеграторе YAML-файл с информацией о динамических библиотеках ПФ и связях коннекторов с соответствующими ПФ.
- 2) Загружает динамические библиотеки с подключаемыми ПФ.
- 3) Получает указатели на коннекторы из математической методики и связывает их с соответствующими ПФ.

Хранение указателя на ПФ реализовано как шаблонный класс Connector. В качестве параметра шаблона задается тип ПФ [14]. Connector хранит указатель на ПФ в приватном атрибуте m_pointer, а публичные методы pointer() и pointer2pointer() предоставляют доступ к указателю и позволяют изменять его значение соответственно.

Класс ConnectorPool является реализацией контейнера коннекторов ПФ. Его исходный код представлен на рисунке 14.

```

class ConnectorPool
{
public:
    Connector<void(unsigned long long, unsigned int, unsigned int, double,
                  double, double, double, double*, double*, double*)>
        initVelocity;
    Connector<void(unsigned long long, unsigned int, unsigned int, double,
                  double, double, double, double*)>
        initPressure;
    Connector<void(unsigned long long, unsigned int, unsigned int, double,
                  double, double, double, double*)>
        initTemperature;
    Connector<void(unsigned long long, unsigned int, unsigned int, double,
                  double, double, double, unsigned int, double*)> initCk;
    Connector<void(unsigned long long, unsigned int, unsigned int, double,
                  double, double, double, double*)>
        sourceEnergy;
    Connector<void(unsigned long long, unsigned int, unsigned int, double,
                  double, double, double, double*, double*, double*)>
        sourceMomentum;
    Connector<void(double, double, unsigned int, double*, double*)>
        dynamicViscosity;
    Connector<void(double, double, unsigned int, double*, double*)>
        thermalConductivity;
    Connector<void(double, double, unsigned int, double*, double*)>
        specificHeat;
    Connector<void(unsigned long long, unsigned int, unsigned int, double,
                  double, double, double, double*)>
        boundaryConditionParameters;

public:
    static ConnectorPool* instance();

private:
    ConnectorPool();
    ~ConnectorPool();
    ConnectorPool(const ConnectorPool&);
    ConnectorPool& operator=(const ConnectorPool&);
};

```

Рисунок 14 – Исходный код класса ConnectorPool

Функции доступа к коннекторам ПФ имеют двухуровневую структуру:

- 1) Функция первого уровня `root_service_function()`, предоставляющая модулю сопряжения доступ к различным сервисным функциям математической методики по их именам, в том числе, к функции второго уровня `callbackConnectorPointer()`.
- 2) Функция второго уровня `callbackConnectorPointer()` предоставляет модулю сопряжения доступ к коннектору каждой ПФ по указанному имени.

Инициализация модуля сопряжения осуществляется в момент инициализации методики путем вызова функции `scf_bridge_init()` модуля сопряжения и передачи в качестве параметра указателя на функцию `root_service_function()`, с помощью которой модуль сопряжения получает доступ к коннекторам ПФ.

Загрузку YAML-файла конфигурации модуля сопряжения осуществляет класс `configuration_t`, находящийся в динамически подключаемой библиотеке `scf_configuration`.

Полученная из YAML-файла информация хранится в виде дерева, а экземпляр класса `configuration_t` предоставляет доступ к узлам этого дерева.

В ходе работы модуль сопряжения получает информацию о связях ПФ из YAML-файла с использованием объекта `m_helper` вспомогательного класса `configuration_helper_t`. Интерфейс этого класса представлен на рисунках 15 и 16.

```
class configuration_helper_t
{
public:
    // коннекторы для работы с функциями динамической библиотеки connection
    function_t<int()>·current_type;
    function_t<const·config_node_t*>·root;
    function_t<const·config_node_t*(const·config_node_t*,·const·char*)>·node;
    function_t<const·config_node_t*(const·config_node_t*,·int)>·list_subnode;
    function_t<const·config_node_t*(const·config_node_t*,·const·char*)>·map_subnode;
    function_t<int(const·config_node_t*)>·node_size;
    function_t<const·char**(const·config_node_t*)>·map_node_keys;

public:
    template<typename·T>
    void·init(T*·shared_library)
    {
        // инициализация коннекторов указателями функций из динамической библиотеки connection
        checker(0·!=·shared_library,·"the·shared·library·isn't·valid");
        current_type.set_pointer(shared_library->void_function_pointer("config_current_type"));
        root.set_pointer(shared_library->void_function_pointer("config_root"));
        node.set_pointer(shared_library->void_function_pointer("config_node"));
        list_subnode.set_pointer(shared_library->void_function_pointer("config_list_subnode"));
        map_subnode.set_pointer(shared_library->void_function_pointer("config_map_subnode"));
        node_size.set_pointer(shared_library->void_function_pointer("config_node_size"));
        map_node_keys.set_pointer(shared_library->void_function_pointer("config_map_node_keys"));
    }

    const·config_node_t*·bridges()
    {
        return·node(node_checker(root()),·"bridges");
    }

    const·config_node_t*·bridge(const·config_node_t*·bridges,·type_t·type)
    {
        return·internal_list_subnode(bridges,·type,·"the·bridge·type·isn't·valid");
    }
}
```

Рисунок 15 – Интерфейс класса `configuration_helper_t` (часть 1)

```

const·config_node_t*·bridges()
{
    return·node(node_checker(root()),·"bridges");
}

const·config_node_t*·bridge(const·config_node_t*·bridges,·type_t·type)
{
    return·internal_list_subnode(bridges,·type,·"the·bridge·type·isn't·valid");
}

const·char*·dll_name(const·config_node_t*·map_node)
{
    return·internal_map_value(map_node,·"dll_name");
}

const·config_node_t*·user_defined_function_connectors(const·config_node_t*·bridge)
{
    return·node(node_checker(bridge),·"user_defined_function_connectors");
}

const·config_node_t*·user_defined_function_connector(const·config_node_t*·connectors,·int·index)
{
    return·internal_list_subnode
        (connectors,·index,·"the·user·defined·function·connector·isn't·valid");
}

```

Рисунок 16 – Интерфейс класса configuration_helper_t (часть 2)

Связывание коннекторов с соответствующими ПФ осуществляет класс linker_t. Для этого экземпляр этого класса в конструкторе получает указатель на функцию root_service_function() из математической методики и сохраняет его в приватный атрибут m_root_service_function. Далее метод link_user_defined_function() получает из входных параметров имя коннектора ПФ, имя динамической библиотеки, содержащей ПФ, имя подключаемой ПФ и связывает их между собой [14]. Таким образом, модуль сопряжения обеспечивает исполнение ПФ в ходе расчетов математических методик.

1.2.4 Последовательность подключения ПФ в ЛОГОС-МИП

Таким образом, обобщенная схема взаимодействия ПФ с математическими методиками в ЛОГОС-МИП изображена на рисунке 17.

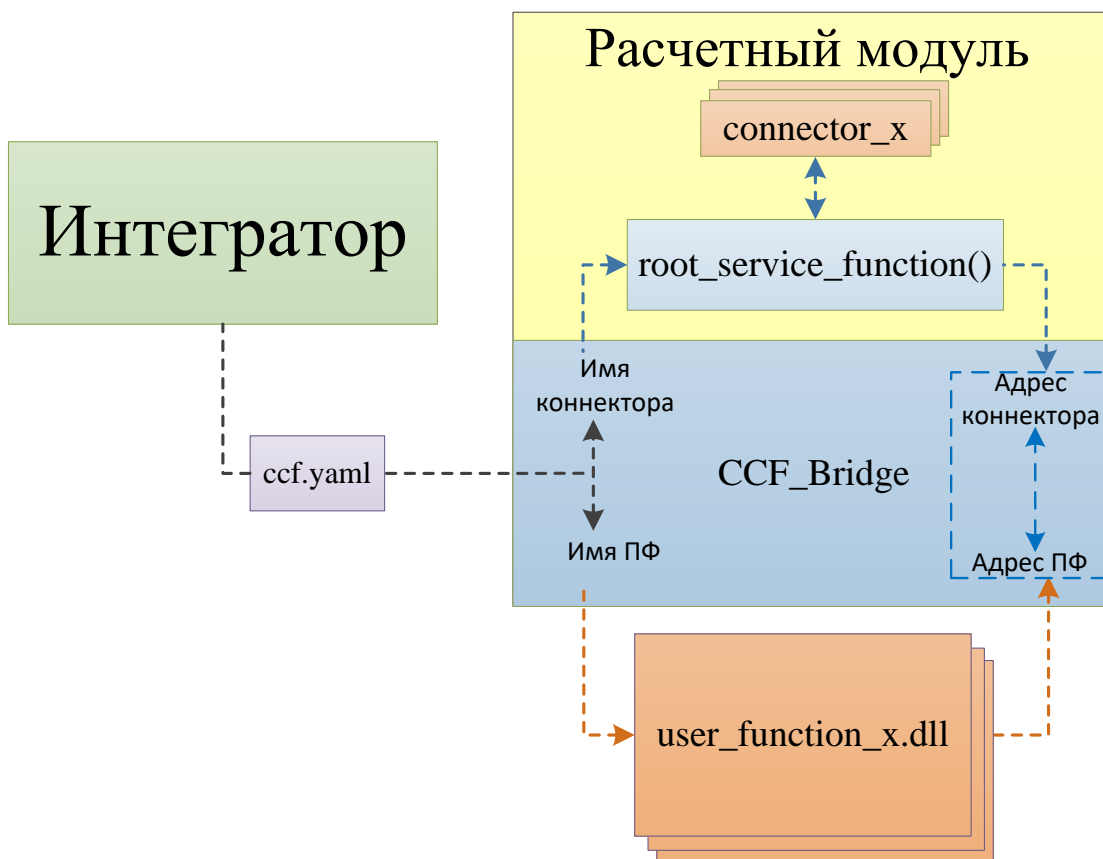


Рисунок 17 – Схема взаимодействия ПФ с математическими методиками в ЛОГОС-МИП

Механизм подключения ПФ подразумевает следующую последовательность действий:

- 1) Создание коннекторов ПФ.
- 2) Определение точек вызова ПФ в расчете.
- 3) Описание API коннекторов с использованием специальных описательных меток системы Doxygen в коде расчетных модулей методики.
- 4) Создание SPEC-файл с описаниями API коннекторов ПФ с помощью специальной утилиты spresgen в ЛОГОС-МИП.
- 5) Разработку ПФ (программный интерфейс ПФ должен соответствовать API ПФ математической методики).
- 6) Описание API ПФ с помощью тегов Doxygen в исходных кодах ПФ.

7) Создание динамической библиотеки подключаемой ПФ (dll-библиотека должна быть объявлена экспортируемой на платформах, которые этого требуют, например, Windows).

8) Создание SPEC-файл с описаниями API ПФ с помощью специальной утилиты `spesgen` в ЛОГОС-МИП.

9) Передачу подготовленных SPEC-файлов в Интегратор.

10) Определение связей ПФ с соответствующими коннекторами в диалоге настройки интегратора.

11) Создание YAML-файла с описанием связей в Интеграторе.

12) Запуск приложения математической методики с инициализацией модуля сопряжения.

13) Обработку YAML-файла модулем сопряжения.

14) Формирование связей между коннекторами и ПФ через указатели (рисунок 18).

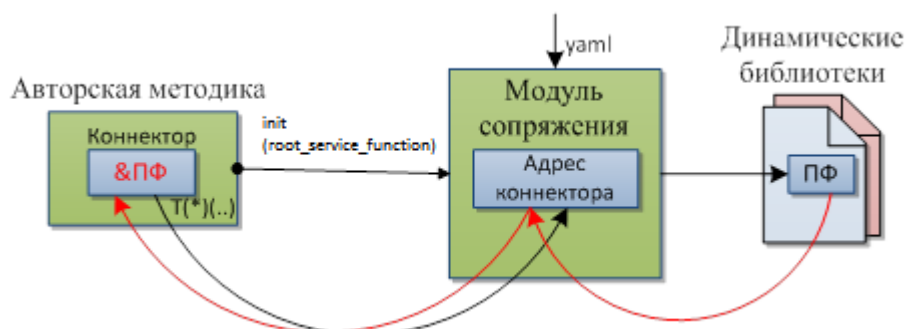


Рисунок 18 – Инициализация коннекторов ПФ

15) Вызов ПФ в ходе расчета в указанное время остановки.

1.3 Пользовательские функции на языке Python

1.3.1 Особенности языка Python

Python – это универсальный современный язык программирования высокого уровня, к преимуществам которого относят высокую производительность программных решений и структурированный, хорошо читаемый код. Это скриптовый интерпретируемый язык, который применяется для решения самого широкого спектра задач [16].

Среди прочих языков программирования его выделяет ряд особенностей:

- 1) Простой и минималистичный синтаксис.
- 2) Свободное и открытое программное обеспечение.
- 3) Расширяемость и встраиваемость. Python можно встраивать в программы на C/C++, чтобы предоставлять возможности написания сценариев или для ускорения работы программы.
- 4) Интерпретируемость. Исходный код не требует компиляции.
- 5) Автоматическое управление памятью. Данная функция позволяет программистам избежать проблем с необходимостью распределять или освобождать память.
- 6) Активное развитие. Примерно раз в 2 года выходят обновления.
- 7) Отсутствие стандартов кодировки.
- 8) Поддержка практически всех операционных систем. В случае значительного устаревания платформы можно использовать ранние версии языка.
- 9) Важность форматирования текста кода.
- 10) Любой описанный класс представляет из себя и объект.
- 11) Динамическая типизация.
- 12) Постоянное расширение языка разрабатываемыми модулями.
- 13) Доступность огромного количества сервисов, сред разработки, и фреймворков.

По статистике Tapatvise [17] на конец 2018 года Python входил в топ-3 самых востребованных языков разработки (рисунок 19).

Dec 2018	Dec 2017	Change	Programming Language	Ratings	Change
1	1		Java	15.932%	+2.66%
2	2		C	14.282%	+4.12%
3	4	▲	Python	8.376%	+4.60%
4	3	▼	C++	7.562%	+2.84%
5	7	▲	Visual Basic .NET	7.127%	+4.66%
6	5	▼	C#	3.455%	+0.63%
7	6	▼	JavaScript	3.063%	+0.59%
8	9	▲	PHP	2.442%	+0.85%
9	-	▲▲	SQL	2.184%	+2.18%
10	12	▲	Objective-C	1.477%	-0.02%
11	16	▲▲	Delphi/Object Pascal	1.396%	+0.00%
12	13	▲	Assembly language	1.371%	-0.10%
13	10	▼	MATLAB	1.283%	-0.29%
14	11	▼	Swift	1.220%	-0.35%
15	17	▲	Go	1.189%	-0.20%
16	8	▼▼	R	1.111%	-0.80%
17	15	▼	Ruby	1.109%	-0.32%
18	14	▼▼	Perl	1.013%	-0.42%
19	20	▲	Visual Basic	0.979%	-0.37%
20	19	▼	PL/SQL	0.844%	-0.52%

Рисунок 19 – Популярность языков программирования

Главными недостатками языка Python является скорость исполнения кода из-за его интерпретируемости, динамической типизации и автоматического управления памятью.

1.3.2 Функции на языке Python

Подпрограмма – это средство языка программирования, фрагмент программного кода, к которому можно обратиться из другого места программы. Подпрограмма должна быть объявлена и в общем случае содержать:

- 1) имя;
- 2) список имен и типов передаваемых параметров (необязательно);
- 3) тип возвращаемого значения (необязательно).

В большинстве языков программирования, если подпрограмма возвращает значение вызывающему коду (одно или несколько), она называется функцией, иначе - процедурой.

Для того, чтобы использовать ранее определенную подпрограмму, необходимо в требуемом месте кода произвести ее *вызов*, указав имя подпрограммы и передав требуемые аргументы (значения параметров).

Код, вызвавший подпрограмму, передает ей управление и ожидает завершения выполнения.

В настоящее время наиболее часто встречаются следующие способы передачи аргументов:

1) По значению. Для переменной, переданной по значению, создается локальная копия и любые изменения, которые происходят в теле подпрограммы с переданной переменной, на самом деле, происходят с локальной копией и никак не сказываются на самой переменной.

2) По ссылке. Изменения, которые происходят в теле подпрограммы с переменной, переданной по ссылке, происходят с самой переданной переменной.

В языке Python нет формального разделения подпрограмм на функции и процедуры – во всех случаях допустимо использования термина «функции» [18].

Для объявления функции в Python используется ключевое слово «def». Обобщенный синтаксис функции показан на рисунке 20.

```
def function_name([parameters]): # `parameters`: параметры функции (через запятую)
    suite                        # Тело функции
```

Рисунок 20 – Синтаксис функций на языке Python

Т.к. Python все данные – это объекты, то при вызове в функцию передается ссылка на этот объект. При этом «мутирующие» объекты передаются по ссылке, «немутирующие» - по значению.

Функция в Python может возвращать результат своего выполнения, используя оператор `return` (рисунок 21). В случае если он не был указан или указан пустой оператор `return`, возвращается специальное значение `None`.

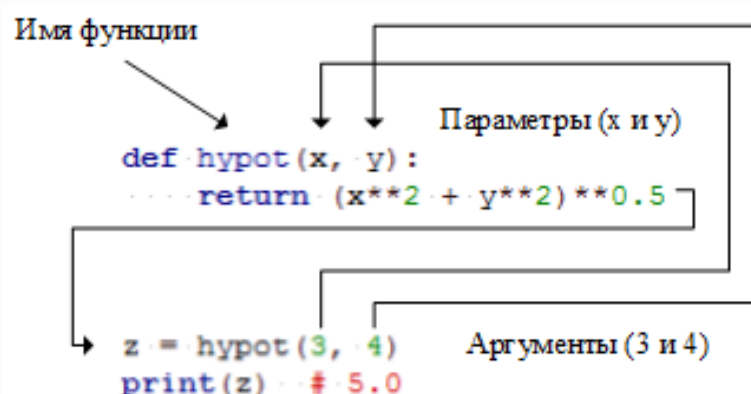


Рисунок 21 – Передача параметров и возврат в функции Python

Python позволяет создавать пользовательские функции, которые можно разделить на 4 типа [19]:

- 1) Глобальные. Доступны из любой точки программного кода в том же модуле или из других модулей.
- 2) Локальные. Объявляются внутри других функций и видны только внутри них: используются для создания вспомогательных функций, которые нигде больше не используются.
- 3) Анонимные. Не имеют имени и объявляются в месте использования. В Python представлены лямбда-выражениями.
- 4) Методы. Функции, ассоциированные с каким-либо объектом (например, `list.append()`, где `append()` - метод объекта `list`).

Все параметры в Python, используемые при объявлении и вызове функции, делятся на позиционные (указываются простым перечислением) и ключевые (комбинация «ключ=значение»).

В ряде случаев возникает необходимость в функции, способной принимать любое число аргументов. Достичь такого поведения можно, используя механизм упаковки аргументов, указав при объявлении параметра в функции один из двух символов:

1) * - все позиционные аргументы начиная с этой позиции и до конца будут собраны в кортеж;

2) ** - все ключевые аргументы начиная с этой позиции и до конца будут собраны в словарь.

Для возвращаемых переменных действительны обратные операции распаковки.

1.3.3 Использование пользовательских функций Python в ЛОГОС-МИП

В данный момент модульная интеграционная платформа ЛОГОС поддерживает подключение пользовательских функций, написанных исключительно на языках C/C++. Однако, как отмечалось выше, в сфере разработки приложений, в том числе, математического моделирования становятся популярными другие языки – в особенности, Python.

Поэтому перед разработчиками ЛОГОС-МИП поставлена востребованная задача расширения функциональности платформы с возможностью подключения в расчетные модули математических методик динамических библиотек ПФ, написанных на языке Python.

Использование скриптового языка Python ускоряет разработку необходимого функционала, так как интерпретируемый код сразу же готов к использованию, исключая временные затраты на его компиляцию и линковку. Также Python является свободно распространяемым и открытым ПО, поэтому не требует дополнительных финансовых затрат на приобретение необходимых компонентов, например, компилятора.

Кроме того, Python устраняет зависимость языка C++ от специфики ОС. Исполняемый файл с пользовательской функцией на C++ должен быть скопирован на целевой удаленный компьютер и скомпилирован для определенной ОС. Такой подход усложняет систему взаимодействия разработчиков пользовательских функций, расчетных модулей и ЛОГОС-МИП, т.к. приводит к сложностям компиляции, удаленной отладки и связи с разработчиком ПФ в случае возникновения ошибок.

Код скрипта Python может быть интерпретирован и использован на любых ОС, за исключением ряда системных функций, использование которых в сфере математического моделирования является маловероятным.

Исходя из вышеперечисленных аргументов, возникла потребность в разработке технологии подключения пользовательских функций, написанных на языке Python, в ЛОГОС-МИП в дополнение к существующим на данный момент механизмам.

ГЛАВА 2. ПРАКТИЧЕСКАЯ ЧАСТЬ

2.1 Требования к разрабатываемой технологии подключения ПФ на языке Python

Результатом практической работы должна являться разработанная технология, которая позволяет подключать и исполнять пользовательские функции на языке Python в составе динамических библиотек формата .dll в модульной интеграционной платформе ЛОГОС-МИП.

На рисунке 22 представлена схема подключения динамических библиотек ПФ к математическим методикам с учетом нового функционала.

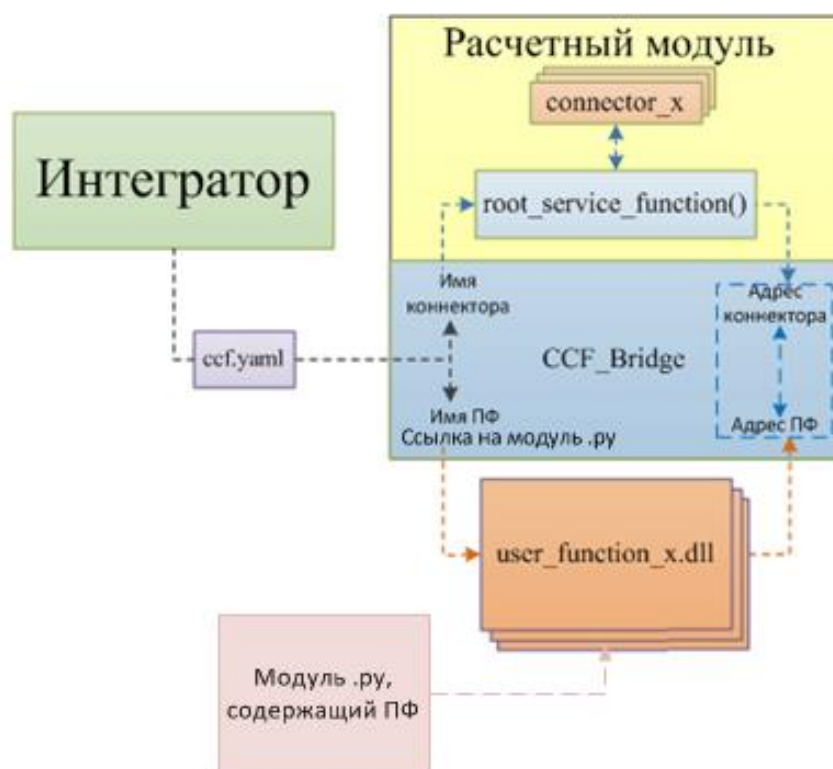


Рисунок 22 – Схема подключения динамических библиотек ПФ, дополненная новым функционалом

Технология должна обеспечивать подключение и исполнение ПФ, написанных на языках Python 2 и Python 3.

В ходе разработки технологии должны применяться исключительно программные продукты, инструменты и среда разработки, разрешенные в ИТМФ РФЯЦ-ВНИИЭФ.

2.2 Исследование существующих технологий

В ходе исследования было выявлено три пути решения поставленной задачи:

- 1) Создание dll-библиотеки из модуля .py, содержащего требуемую ПФ, средствами языка Python.
- 2) Использование CPython, Cython - специальных реализаций языка Python.
- 3) Использование Python C/C++ API.

2.2.1 Создание dll-библиотеки средствами языка Python

Первый вариант представляет собой конвертацию.pyd (динамической библиотеки, подготовленной для импорта в Python) в библиотеку .dll. Этот механизм предполагает несколько операций: получение исходных файлов на языке C++ в качестве источника для скрипта Python, создание библиотеки .pyd средствами Python и подготовку дополнительных файлов для дальнейшего преобразования в .dll. Чаще всего для подключения источников используют стандартный модуль distutils или setuptools [20], а для создания .pyd – скрипт установки setup.py (рисунок 23).

```
from distutils.core import setup, Extension
module1 = Extension('brisk',
    include_dirs = ['include', 'C:/opencv2.4/build/include',
    'C:/brisk/thirdparty/agast/include'],
    #libraries = ['agast_static', 'brisk_static'],
    #library_dirs = ['win32/lib'],
    sources = ['src/brisk.cpp'])
setup (name = 'BriskPackage',
    ext_modules = [module1])
```

Рисунок 23 – Создание .pyd.

Дальнейшая конвертация .pyd в .dll возможна при помощи дополнительных модулей или программ. Свободно распространяемым модулем является py2exe [21] – расширение distutils, позволяющее

преобразовать .pyd в .exe без дополнительной подготовки или в .dll с файлами экспорта и переменных окружения.

Данный вариант создания dll-библиотеки требует установки двух дополнительных модулей py2exe, отдельно для Python2 и Python 3. В ходе конвертации py2exe вносит изменения в модуль .py, что может породить различные ошибки. Существует вероятность, что в случае ошибки конвертации или недостаточной подготовки дополнительных файлов интегратор будет воспринимать строку экспорта как одну из подключаемых пользовательских функций.

2.2.2 Использование CPython или Cython

Второй путь решения поставленной задачи заключается в использовании CPython или Cython. CPython является одной из реализацией языка Python. Он совмещает в себе функции компилятора и интерпретатора, т.к. компилирует код Python в байт-код перед его интерпретацией [22]. Схема преобразований представлена на рисунке 24.

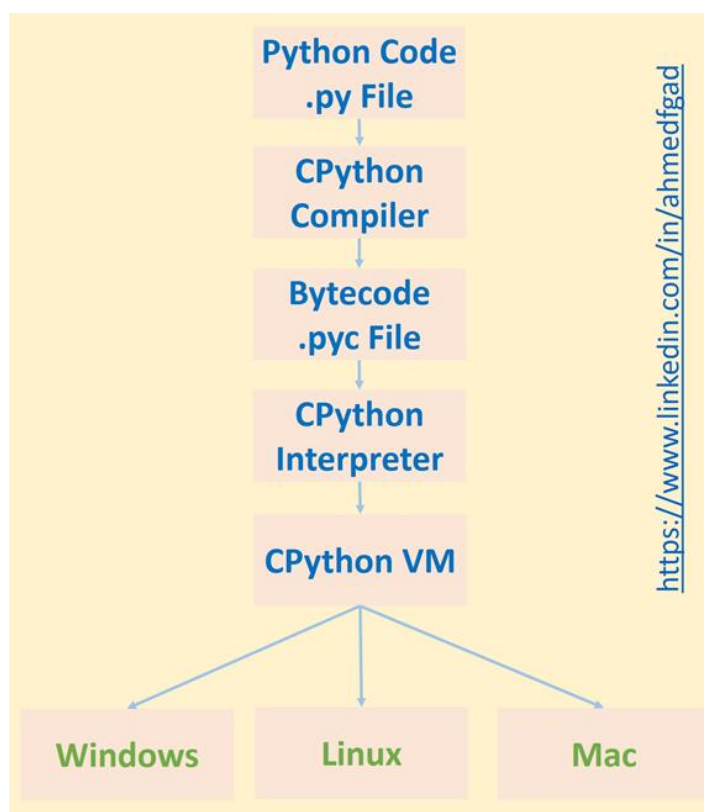


Рисунок 24 – Схема преобразований кода в CPython.

CPython работает быстрее классического языка Python, т.к. позволяет разработчику один раз скомпилировать байт-код, вместо длительной интерпретации исходного кода на Python при каждом запуске. Также CPython использует статическую типизацию, приближенную к языку C, в отличие от динамической в Python. На рисунке 25 представлена разница в реализации типов на двух языках [23].



Рисунок 25 – Типизация в CPython и Python

Недостатками CPython является технология Global Interpreter Lock, из-за которой язык не поддерживает многопоточность. Также не реализованы многие алгоритмы оптимизации, например, рекурсий, которые используются в Python.

Cython - это язык программирования, основанный на Python, с дополнительным синтаксисом для статической типизации. Он позволяет совмещать код на C++ и Python в одном файле. Исходный код компилируется в эквивалентный C-код [24]. Разница в синтаксисе языка Python и Cython представлена на рисунке 26.

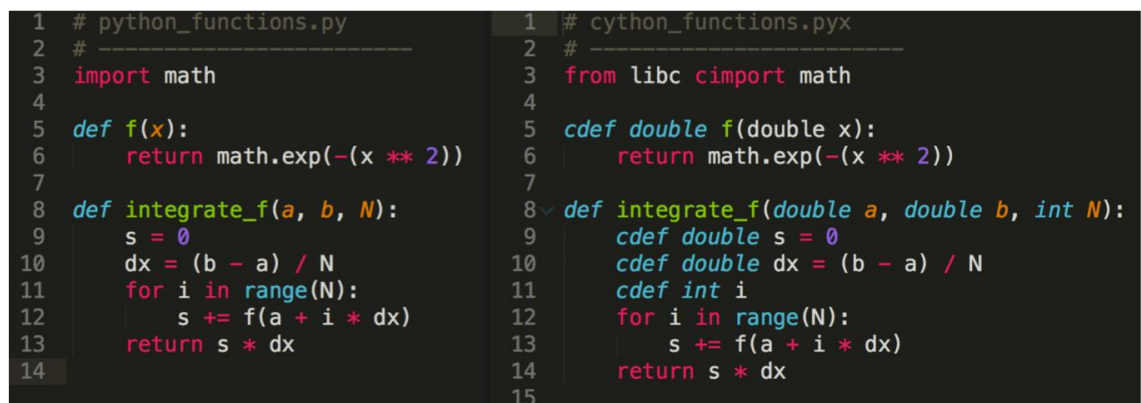


Рисунок 26 – Разница синтаксиса Python и Cython

В языке Cython не ограничивается многопоточность и используется весь функционал языка Python.

2.2.3 Использование Python C API

Python C/C++ API (далее Python C API) предоставляет программистам C и C++ доступ к интерпретатору Python и позволяет встраивать компонент на языке Python в приложения и библиотеки на языке C или C++. Связь между языками реализуется через подключение в проект библиотеки Python.h, которая находится в папке include любой установленной версии Python и не требует загрузки дополнительного ПО.

Этот метод позволяет оперировать объектами Python из C-кода, т.е. разработчик может обрабатывать переменные и функции модулей .py на языке C, используя функции из библиотеки Python.h [25]. Пример функции с использованием Python C API представлен на рисунке 27.

```
#include <Python.h>

static PyObject*
arithmetic_add(PyObject* self, PyObject* args)
{
    int i, j;
    PyArg_ParseTuple(args, "ii", &i, &j);
    PyObject* sum = PyInt_FromLong(i + j);
    return sum;
}
```



Рисунок 27 – Пример кода с использованием Python C API

Таким образом, dll-библиотека может быть собрана из C-файлов с подключением модулей .py в качестве внешних источников.

2.2.4 Обоснование выбора решения

Из предыдущей главы следуют следующие плюсы и минусы каждого из возможных путей решения.

1) Конвертация .pyd предполагает наличие у каждого разработчика динамических библиотек модуля py2exe для Python 2 и/или Python 3. Кроме того, разработчик должен уметь создавать библиотеки .pyd и конвертировать

их при помощи этого модуля, специальным образом подготавливая дополнительные файлы. В ходе конвертации вероятно нарушение целостности исходных файлов проекта и внесения изменений в модуль .ру, что в будущем может негативно сказываться на работе интегратора.

2) Использование CPython предполагает установку этого модуля и знание особенностей его синтаксиса. Разработчик библиотеки вынужден заниматься и написанием ПФ на языке CPython, и подготовкой оболочки для ее взаимодействия с интегратором, без возможности разделить обязанности на специалистов в Python и C++. Также технология GIL, используемая в CPython, блокирует многопоточность. В этой реализации языка не поддерживаются многие методы оптимизации Python. Главным достоинством этого пути решения является скорость исполнения ПФ. Для сбора библиотек нужны специальные программы, например, CMake.

3) Использование Cython также предполагает установку пакетов этого языка. В Cython устранены многие недостатки CPython, поддерживается многопоточность. Этот метод решения требует использования полноценного языка со своими особенностями синтаксиса (в отличие от CPython, где достаточно знать статические типы данных), что подходит не каждого разработчику ПФ на Python. Для сбора библиотек нужны специальные программы, например, CMake.

4) Использование Python C API не требует установки дополнительных расширений и модулей, необходимая библиотека Python.h для любой версии языка находится в папке Python<Version>/include. Этот путь позволяет разделить разработчиков ПФ на языке Python и оболочек на языке C++. Существует возможность создать универсальную библиотеку, реализующую подключение и исполнение ПФ без необходимости знать ее содержимое – определяющими являются типы и количество входящих и возвращаемых переменных. Работу по совмещению языков выполняет только разработчик на C++, который при помощи стандартных функций Python C API, являющимися функциями языка C++, должен подготовить преобразование

типов переменных. Для сбора библиотек нужны специальные программы, например, CMake.

Я считаю Python C API наиболее подходящим для решения поставленной задачи, поскольку он не требует установки дополнительных модулей и расширений, а использует стандартную библиотеку языка Python, разрешенного на предприятиях Росатома. Также этот путь позволяет разделить разработчиков на C++ и Python и создать универсальную библиотеку для исполнения ПФ, не зависящую от ее скрипта, кроме входящих аргументов.

2.3 Python C API

API для Python предоставляет программистам C и C++ доступ к интерпретатору Python на разных уровнях. API можно использовать как в языке C, так и в C ++, но для краткости его обычно называют Python C API [26].

Существуют две главные причины использования Python C API. Первая заключается в написании модулей расширения; это модули C/C++, расширяющие интерпретатор Python. Вторая причина заключается в использовании Python в качестве компонента в более крупном приложении; этот метод обычно называется встраиванием Python в приложение, что соответствует поставленной передо мной задаче.

Для работы с Python C API достаточно включить в проект стандартные файлы языка Python – Python.h в папке include и Python<Version>.lib, Python<Version>_d.lib в папке libs. Версии этих файлов должны совпадать с используемой версией языка, и при переходе на другой Python пути к файлам должны быть изменены на корректные.

Включение <Python.h> подразумевает так же включение следующих стандартных заголовков: <stdio.h>, <string.h>, <errno.h>, <limits.h>, <assert.h>и <stdlib.h>.

Все видимые пользователю имена, определенные Python.h имеют один из префиксов Py или _Py. Имена, начинающиеся с Py, _Py не должны использоваться авторами расширений.

Большинство функций Python C API имеют один или несколько аргументов, а также возвращаемое значение типа PyObject*. Этот тип является указателем на непрозрачный тип данных, представляющий произвольный объект Python. Поскольку все типы объектов Python обрабатываются языком Python одинаково в большинстве ситуаций, вполне уместно, чтобы они были представлены одним типом в языках C/C++.

Например, PyObject* может представлять собой кортеж (1, 2, "three"), который можно заполнить разными способами – рисунки 28 и 29.

```
PyObject *t;  
  
t = PyTuple_New(3);  
PyTuple_SetItem(t, 0, PyLong_FromLong(1L));  
PyTuple_SetItem(t, 1, PyLong_FromLong(2L));  
PyTuple_SetItem(t, 2, PyUnicode_FromString("three"));
```

Рисунок 28 – Последовательное заполнение кортежа

```
PyObject *tuple, *list;  
  
tuple = Py_BuildValue("(iis)", 1, 2, "three");
```

Рисунок 29 – Заполнение кортежа при помощи строки формата

Все объекты Python в языке C имеют тип и счетчик ссылок. Тип объекта определяет, что он представляет собой в языке Python, например, целое число, список или пользовательскую функцию. Для каждого стандартного типа существует макрос, который проверяет, принадлежит ли объект этому типу; например, PyList_Check(a) возвращает true, если объект, на который указывает a, является списком языка Python. Счетчик ссылок важен, потому что современные компьютеры имеют ограниченный (и часто строго ограниченный) объем памяти; он подсчитывает, сколько разных ссылок имеется на этот объект. Ссылаться другой объект, глобальная или статическая переменная языка C или локальная переменная в некоторой

функции C. Когда счетчик ссылок объекта становится равным нулю, объект уничтожается. Если он содержит ссылки на другие объекты, их счетчик ссылок уменьшается. Каждый объект PyObject* имеет ссылку на самого себя.

Существует очевидная проблема с объектами, которые ссылаются здесь друг на друга; на данный момент, согласно документации, существует единственное решение - «не допускать этого» [26].

Кроме главенствующего типа PyObject*, есть несколько других типов данных, которые играют важную роль в Python C API; большинство из них совпадает с простыми типами языка C, такими как int, long, double и char*.

Через Python C API доступны все встроенные возможности языка Python[27]:

1) высокоуровневый интерфейс интерпретатора (функции и макросы Py_Main(), PyRun_String(), PyRun_File(), Py_CompileString(), PyCompilerFlags() и т.п.);

2) функции для работы со встроенным интерпретатором и потоками (Py_Initialize(), Py_Finalize(), Py_NewInterpreter(), Py_EndInterpreter(), Py_SetProgramName() и другие);

3) управление подсчетом ссылок (макросы Py_INCREF(), Py_DECREF(), Py_XINCREF(), Py_XDECREF(), Py_CLEAR()), требуется при создании или удалении Python-объектов в C/C++-коде;

4) обработка исключений (PyErr* -функции и PyExc_* -константы, например, PyErr_NoMemory() и PyExc_IOError);

5) управление процессом и сервисы операционной системы (Py_FatalError(), Py_Exit(), Py_AtExit(), PyOS_CheckStack(), и другие функции/макросы PyOS*);

6) импорт модулей (PyImport_Import() и другие);

7) поддержка сериализации объектов (PyMarshal_WriteObjectToFile(), PyMarshal_ReadObjectFromFile() и т.п.);

8) поддержка анализа строки аргументов (PyArg_ParseTuple(), PyArg_VaParse(), PyArg_ParseTupleAndKeywords(),

PyArg_VaParseTupleAndKeywords(), PyArg_UnpackTuple() и Py_BuildValue()). С помощью этих функций облегчается задача получения в коде на C параметров, заданных при вызове функции из Python. Функции PyArg_Parse* принимают в качестве аргумента строку формата полученных аргументов;

9) поддержка протоколов абстрактных объектов:

- Протокол объекта PyObject_Print(), PyObject_HasAttrString(), PyObject_GetAttrString(), PyObject_HasAttr(), PyObject_GetAttr(), PyObject_RichCompare(), ..., PyObject_IsInstance(), PyCallable_Check(), PyObject_Call(), PyObject_Dir() и другие);

- Протокол числа (PyNumber_Check(), PyNumber_Add(), PyNumber_And(), PyNumber_InPlaceAdd(), PyNumber_Coerce(), PyNumber_Int(), ...);

- Протокол последовательности (PySequence_Check(), PySequence_Size(), PySequence_Concat(), PySequence_Repeat(), PySequence_InPlaceConcat(), PySequence_GetItem(), PySequence_GetSlice(), PySequence_Tuple(), PySequence_Count(), ...);

- Протокол отображения (например, словарь является отображением) (PyMapping_Check(), PyMapping_Length(), PyMapping_HasKey(), PyMapping_Keys(), PyMapping_SetItemString(), PyMapping_GetItemString() и др.);

- Протокол итератора (PyIter_Check(), PyIter_Next());

- Протокол буфера (PyObject_AsCharBuffer(), PyObject_AsReadBuffer(), PyObject_AsWriteBuffer(), PyObject_CheckReadBuffer());

10) поддержка встроенных типов данных. Например, булевый объект (PyBool_Check() - проверка принадлежности типу PyBool_Type, Py_False - объект False, Py_True - объект True,

11) управление памятью (то есть интерпретатором Python) (PyMem_Malloc(), PyMem_Realloc(), PyMem_Free(), PyMem_New(), PyMem_Resize(), PyMem_Del()). Разумеется, можно применять и средства

выделения памяти C/C++, однако, в этом случае не будут использоваться преимущества управления памятью интерпретатора Python (сборка мусора и т.п.).

12) структуры для определения объектов встроенных типов (PyObject, PyVarObject и много других).

Разработчики приложений, которые встраивают язык Python, должны инициализировать интерпретатор Python функцией Py_Initialize() и, при необходимости, прерывать доступ к нему функцией Py_Finalize(). Большая часть функций интерпретатора может использоваться только после инициализации интерпретатора.

Py_Initialize() инициализирует таблицу загруженных модулей, и создает основные модули builtins, __main__ и sys. Она также инициализирует путь поиска модуля (sys.path).

В моей работе основными используемыми функциями Python C API являются функции импорта модуля .py, позволяющие разобрать модуль на несколько пользовательских функций и обращаться к ним для исполнения; функции преобразования типов из PyObject* в типы языка C и обратно; функции формирования строки аргументов переменной длины для исполнения ПФ и функции вызова методов ПФ.

2.4 Практическая реализация технологии

Главная идея технологии заключается в инициализации интерпретатора Python в оболочке, написанной на C++ языке. Доступ к интерпретатору позволяет получить библиотека Python.h и ее стандартные зависимости. Функции Python C API являются функциями языка C++, поэтому инициализация интерпретатора не требует знания языка Python, достаточно знания C++.

Модуль .py, хранящий описание пользовательских функций, является внешним источником для библиотеки dll, он может подключаться к проекту или вызываться по ссылке.

Библиотека PyToCcpp.h представляет набор функций, позволяющих исполнить пользовательскую функцию с заданными аргументами и получить возвращаемое ей значение в качестве PyObject*.

С-файл, вместе с библиотеками экспорта, обеспечивает взаимодействие dll-библиотеки с интегратором через configuration_helper.hpp. Также С-файл инициализирует интерпретатор Python и производит конвертацию С-типов в PyObject* и обратно.

Таким образом, в проект для сбора dll-библиотеки с инициализированным интерпретатором Python (рисунок 30) должны быть обязательно включены С-файл, библиотека PyToCcpp.h и Python.h. Модуль .py может быть вызван по ссылке, определенной в файле конфигурации scf.yaml, связь библиотеки с интегратором устанавливается функциями библиотеки configuration_helper.hpp.

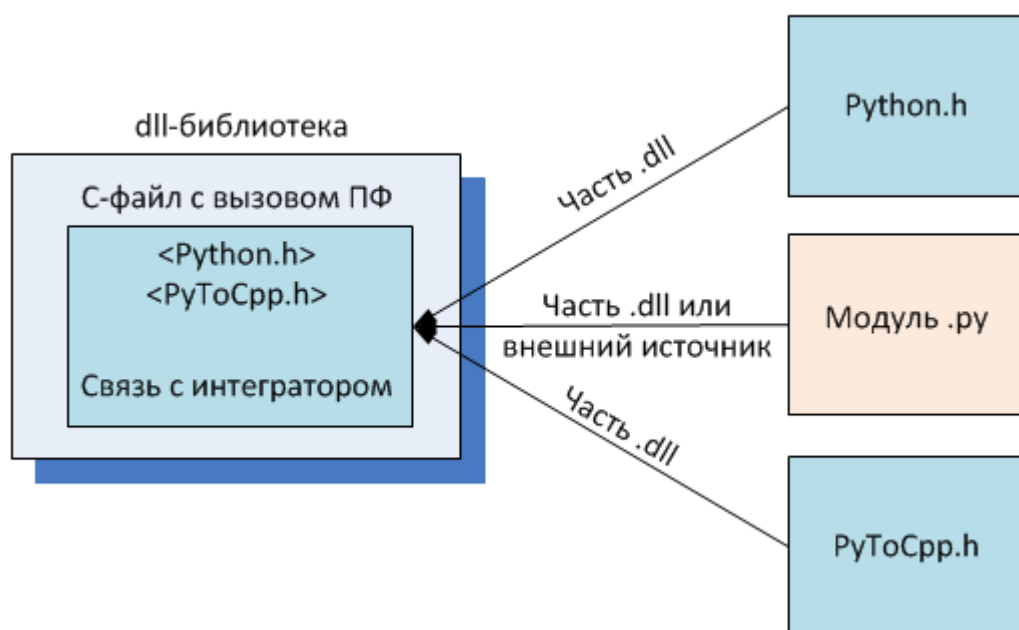


Рисунок 30 – Составные части проекта для сбора dll-библиотеки

2.4.1 Связь с интегратором

Согласно техническому заданию, разрабатываемая технология подключения динамических библиотек ПФ к ЛОГОС-МИП должна предоставлять возможность связи dll-библиотек и .py модулей с Интегратором. Для этого был доработан класс configuration_helper_t.

В него включены две новые функции (рисунок 31):

1) `const char* python_module_name(...)` позволяет получать ссылку на .py модули, указанные в файле `scf.yaml`.

2) `const char* python_def_name(...)` позволяет получить название целевой пользовательской функции этого модуля из `scf.yaml`.

```
const char* python_module_name(const config_node_t* map_node)
{
    return internal_map_value(map_node, "python_module_name");
}

const char* python_def_name(const config_node_t* map_node)
{
    return internal_map_value(map_node, "python_def_name");
}
```

Рисунок 31 – Изменения в классе `configuration_helper_t`

Аналогичным образом была доработана структура конфигурационного файла `scf.yaml`, который, при необходимости, включает поля `python_module_name` и `python_def_name`. Новый вид дерева `scf.yaml` представлен на рисунке 32.

```
bridges:
- user_defined_function_connectors:
  - dll_name: ablationRateLib
    connector_name: ablation_rate
    function_name: uf_ablation_rate
    python_module_name: ablation.py
    python_def_name: ablation
  interfaces: ~
  lightweight_solvers: ~
  invoke_points: ~
```

Рисунок 32 – Файл конфигурации `scf.yaml`

2.4.2 С-оболочка

С-файл является оболочкой для пользовательской функции на языке Python. Он обеспечивает связь с интегратором и получение требуемых данных о .py модуле и целевой ПФ.

Этот компонент реализует инициализацию и прерывание интерпретатора Python функциями `Py_Initialize()` и `Py_Finalize()`. Работающий интерпретатор обрабатывает функцию `python_to_cpp(...)` из библиотеки `PyToCcpp.h`, в которую подаются необходимые имена и аргументы для

исполнения конкретной ПФ, и которая возвращает полученное значение типа PyObject*.

Главная задача разработчика С-оболочки для ПФ заключается в специальном преобразовании типов из языка С++ в типы Python, подаваемые в Python-скрипт, и из возвращенного PyObject* в необходимый тип языка С++ (рисунок 33).

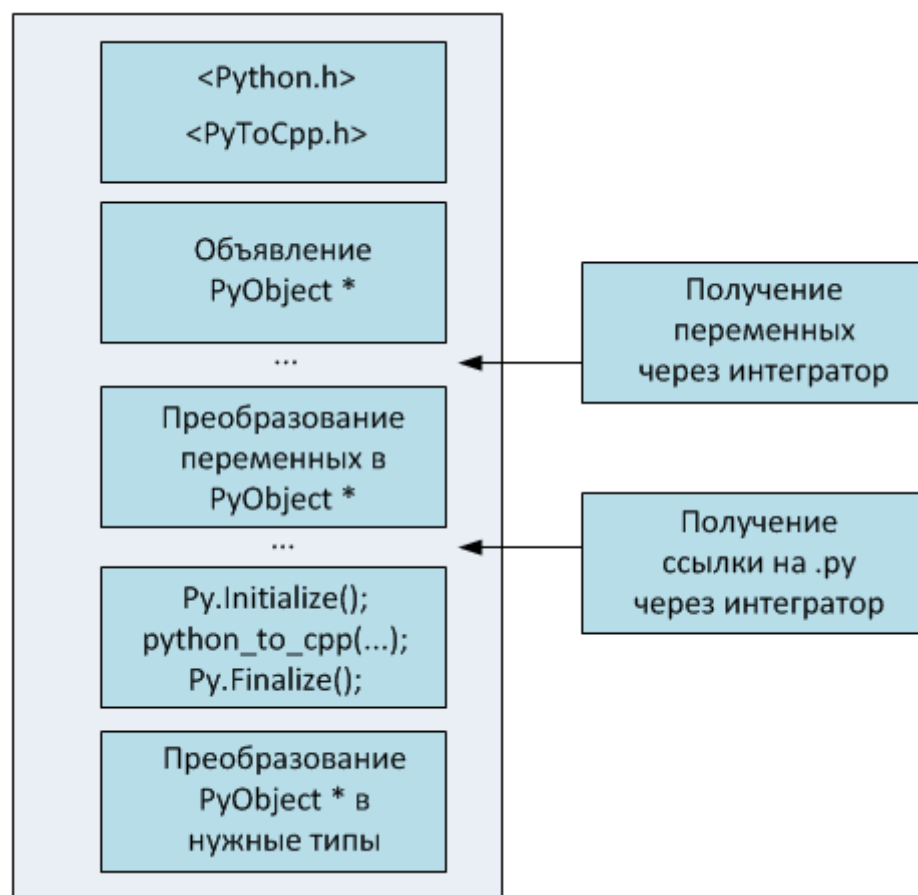


Рисунок 33 – Структура С-оболочки

ПФ Python могут запрашивать разное количество аргументов разных типов. Для того чтобы ПФ смогла понять, какие переменные подают ей на вход, необходимо использовать функции Python C API – Py_BuildValue(...) и подобные. Т.к. количество аргументов является переменной величиной, то выбор функций ограничивается одним вариантом - Py_VaBuildValue(const char* format, va_list argp).

Эта функция принимает на вход некий набор аргументов и строку формата с последовательным описанием типа каждой переменной.

Например, последовательность аргументов типа `char*`, `int`, `int` имеет строку формат “(sii)”.

Однако очень неудобно каждый раз вычислять букву формата каждого аргумента, особенно выходящих за рамки стандартных, например, `vector<>`. Поэтому я выбрала другой способ – передачу всех аргументов как тип `PyObject*`, обозначаемый в строке формата буквой N. Тогда появляется возможность автоматически, циклом, генерировать строку формата, состоящую из букв N по количеству аргументов.

Функции Python C API позволяют превращать типы языка C в `PyObject*`. Например, `PyObject* PyFloat_FromDouble(double x)`. Многие списки, вектора, массивы и прочее могут быть преобразованы в `PyObject*`, передающие в скрипты списки или кортежи.

Аналогично функции Python C API позволяют «распаковывать» `PyObject*` в нужные типы, в том числе, обрабатывать получаемые списки и кортежи.

2.4.3 Библиотека PyToCrr.h

Библиотека `PyToCrr.h` представляет собой набор функций, позволяющих запускать и исполнять пользовательские функции на языке Python. Библиотека может обрабатывать содержимое любого указанного модуля `.py`, находить в модуле требуемую пользовательскую функцию по ее названию и исполнять эту функцию с переданными аргументами, количество которых определено заранее. Библиотека написана в общем виде, она не зависит от конкретных модулей `.py` и `c`-файлов с передаваемыми переменными заданных типов, т.к. `PyToCrr.h` работает с переменным количеством аргументов, приведенных к типу `PyObject*`.

Структура библиотеки `PyToCrr.h` представлена на рисунке 34.

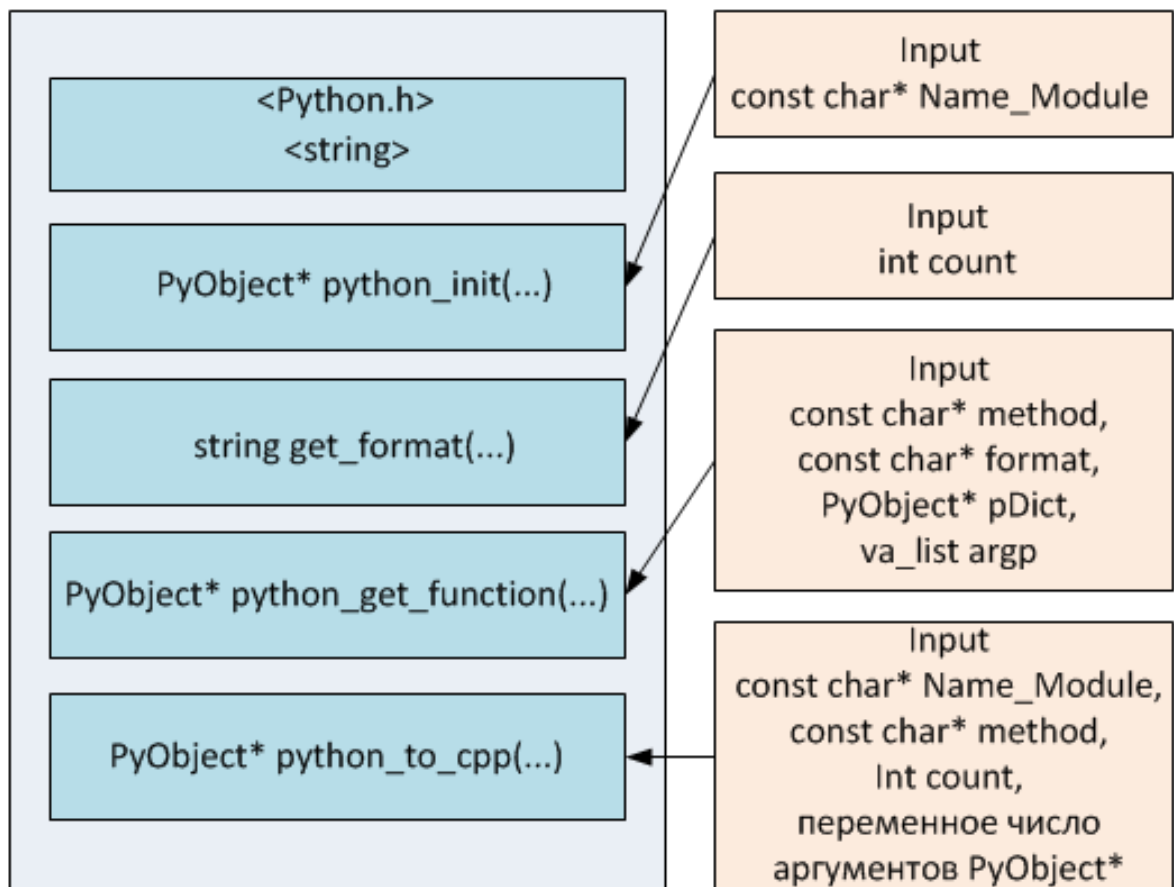


Рисунок 34 – Структура PyToCpp.h

PyToCpp зависит от библиотеки Python.h и класса string.

1) Функция `PyObject* python_init (const char* Name_Module)` подключает и обрабатывает указанный модуль .py, составляя словарь пользовательских функций для быстрого доступа к ним по их именам. Функция принимает символьную переменную, содержащую имя требуемого модуля .py с пользовательскими функциями в случае его подключения на этапе сборки библиотеки или ссылку на этот модуль как внешний источник. Код функции представлен на рисунке 35.

```

PyObject* python_init(const char* Name_Module){
    /*Получение словаря функций pDict*/
    PyObject *pName, *pModule, *pDict;
    pName = PyUnicode_FromString(Name_Module);
    pModule = PyImport_Import(pName);
    if (!pModule) {
        printf("Error_pModule");
        PyErr_Print();
    }
    pDict = PyModule_GetDict(pModule);
    if (!pDict) {
        printf("Error_pDict");
        PyErr_Print();
    }
    return pDict;
}

```

Рисунок 35 – Код функции python_init(...)

Функция возвращает словарь пользовательский функций типа PyObject*. Корректность работы библиотеки с модулем (успешность подключения модуля и создания словаря) проверяется средствами Python C API.

2) Функция string get_format (int count) создает и возвращает символьную строку, в которой перечислены форматы аргументов для исполнения конкретной пользовательской функции. Функция принимает целое число, представляющее количество аргументов пользовательской функции. Код представлен на рисунке 36.

```

string get_format(int count){
    string format = "(";
    for (int i = 0; i < count; i++)
        format = format + "N";
    format = format + ")";
    return format;
}

```

Рисунок 36 – Код функции get_format(...)

Согласно разработанной технологии, аргументами являются переменные типа PyObject*. Поэтому итоговой формат аргументов состоит из соответствующего числа букв N, стандартного символа Python C API,

обозначающего этот тип. Таким образом, если пользовательская функция предполагает использование двух int-аргументов и одного char или трех float, в обоих случаях строка формата будет выглядеть одинаково – «(NNN)».

3) Функция PyObject* python_get_function (const char* method, const char* format, PyObject* pDict, va_list argp) вызывает и исполняет указанную пользовательскую функцию из модуля .py. Входящими переменными является строка с названием ПФ, полученная строка формата, словарь модуля .py и va_list набор аргументов. Функция возвращает результат работы ПФ - переменную типа PyObject*, которая может представлять собой единичный объект PyObject* или кортеж из таких объектов, в зависимости от количества возвращаемых переменных ПФ на языке Python. Код функции представлен на рисунке 37.

```
PyObject* python_get_function(const char* method, const char* format,
                             PyObject* pDict, va_list argp){
    PyObject *pObj, *pVal, *pArgs;
    pObj = PyDict_GetItemString(pDict, method);
    if (!pObj)
        PyErr_Print();
    pArgs = Py_VaBuildValue(format, argp);
    if(pArgs == NULL)
        PyErr_Print();
    pVal = PyObject_CallObject(pObj, pArgs);
    if (pVal == NULL)
        PyErr_Print();
    return pVal;
}
```

Рисунок 37 – Код функции python_get_function(...)

Функция находит в словаре модуля .py нужную ПФ по ее названию, затем составляет перечень аргументов переменной длины, используя составленную ранее строку формата, и исполняет ПФ. Каждый этап контролирует обработчик ошибок, использующий стандартные средства Python C API. В случае невозможности исполнения ПФ (например, из-за несоответствия количества аргументов или стандартных типов, «упакованных» в PyObject*, тем, которые предполагает данная ПФ), функция

python_get_function(...) выводит текст ошибки согласно интерпретатору языка Python.

4) Функция PyObject* python_to_cpp (const char* Name_Module, const char* method, int count, ...) запускает процесс обработки и исполнения ПФ из модуля .py. Входящими переменными является строка с именем модуля .py или со ссылкой на него, строка с названием ПФ, количество передаваемых аргументов и перечень аргументов переменной длины. Функция возвращает переменную типа PyObject* - результат исполнения функции python_get_function(...). Код представлен на рисунке 38.

```
PyObject* python_to_cpp(const char* Name_Module, const char* method,
                        int count, ...){
    PyObject *rez, *pDict;
    const char* format;
    string fString = get_format(count);
    format = fString.c_str();
    va_list argp;
    va_start(argp, count);
    pDict = python_init(Name_Module);
    rez = python_get_function(method, format, pDict, argp);
    va_end(argp);
    return rez;
}
```

Рисунок 38 – Код функции python_to_cpp(...)

Функция вызывает get_format(...), возвращающую строку форматов аргументов, python_init(...), возвращающую словарь модуля .py, при помощи макроса формирует va_list аргументов и исполняет требуемую ПФ, возвращая полученные в результате ее работы значения для дальнейшей обработки.

2.4.4 Сборка dll-библиотеки

Для сборки dll-библиотеки я использовала CMake версии 3.2.3 – кроссплатформенную систему автоматизации сборки программного обеспечения из исходного кода [28].

В состав проекта nameLib вошли следующие файлы:

1) nameLib.cpp, главный файл библиотеки, в котором обеспечивается связь с интегратором, происходит подготовка аргументов для исполнения

пользовательских функций, вызов и исполнение ПФ, обработка полученных результатов.

2) export.h и nameLib.h, библиотеки экспорта.

3) Один или несколько модулей *.py с ПФ (могут быть переданы по ссылке через Интегратор).

4) PyToCpp.h, библиотека, реализующая исполнение ПФ.

5) Библиотека configuration_helper.hpp.

Для подключения dll-библиотеки к интегратору пользователю необходимо установить компоненты «ЛОГОС Платформа» и включить в проект библиотеку configuration_helper.hpp.

Для корректной работы библиотеки PyToCpp.h в файле CmakeLists.txt должны быть указаны пути к Python.h используемой версии языка (стандартный путь Python<Version>/include) и к библиотекам Python<Version>.lib, Python<Version>_d.lib (стандартный путь Python<Version>/libs). Например:

```
set (PY_LIB "C:/Python27/libs")  
file(GLOB PL ${PY_LIB}/python27_d.lib)  
file(GLOB PL ${PY_LIB}/python27.lib)
```

2.5 Пример реализации технологии

Пример существующей пользовательской функции ablationRate.cpp предоставлен научным руководителем. В ходе работы ПФ была переписана на язык Python в модуле ablationRate.py, а файл ablationRate.cpp изменен для подготовки аргументов и вызова ПФ на языке Python. Библиотеки экспорта остались без изменений. Подготовленные файлы, вместе с библиотекой PyToCpp.h, были включены в проект и собраны в dll-библиотеку ablationRateLib.dll.

Разработка велась в среде Visual Studio 2008.

Подготовлено подключение dll-библиотеки к интегратору, но тестирование их взаимодействия на данный момент не проводилось.

Состав проекта представлен на рисунке 39.

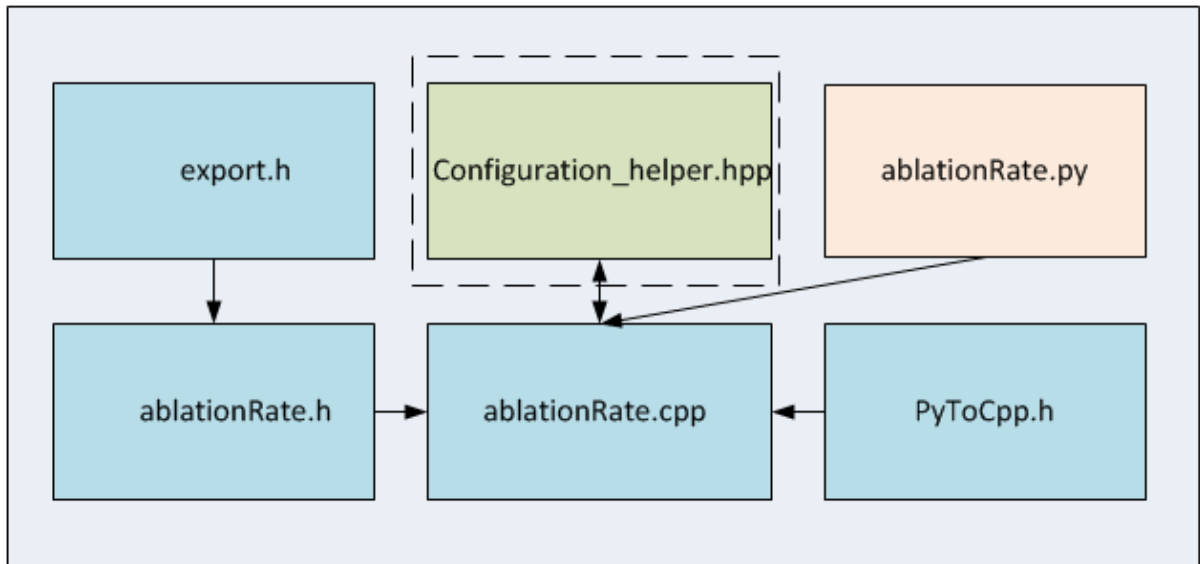


Рисунок 39 – Состав проекта ablationRateLib

В модуль ablationRate.py входят три ПФ (рисунок 40):

```

from math import exp

def ablation(fT):
    return fT*(exp(-fT/680)+0.15)/700

def mathAll(x,y):
    return x+y, x-y, x*y, x/y

def indexString(x):
    return 'file'+x+'.txt'
  
```

Рисунок 40 – Модуль ablationRate.py

1) def ablation (fT), вычисляющая значение абляции. Функция переписана на язык Python из соответствующего примера на C++. В качестве входящей переменной принимает одно числовое значение (int, float), возвращает также одно числовое значение.

2) def mathAll (x, y), функция для тестирования, позволяющая вычислять значения 4 стандартных математических операций над двумя числами. В качестве входящих переменных принимает два числовых значения, возвращает кортеж из 4 числовых значений.

3) def indexString (x), функция для тестирования, позволяющая объединять строку с заранее заданной. В качестве входящей переменной принимает строку или символ, возвращает строку.

Файл ablationRate.cpp содержит функцию uf_ablation_rate(...) с аргументами типа double, получаемыми из внешних источников (рисунок 41).

```
#include "configuration_helper.hpp"

#include "Python.h"
#include "PyToCpp.h"
#include "ablationRate.h"

/** @brief Скорость массового уноса
 * @param fT значение температуры в центре ячейки
 * @param centerX координаты центра ячейки
 * @param centerY координаты центра ячейки
 * @param centerZ координаты центра ячейки
 * @param normalX координаты проекции центра ячейки на нормаль
 * @param normalY координаты проекции центра ячейки на нормаль
 * @param normalZ координаты проекции центра ячейки на нормаль
 * @user_function
 */
double uf_ablation_rate( double fT,
                        double centerX,
                        double centerY,
                        double centerZ,
                        double normalX,
                        double normalY,
                        double normalZ)
{
    double result = 0.0;
    PyObject *argObj1, *rez;

    ccf::configuration_helper_t helper;
    const config_node_t* udf_connectors = helper.user_defined_function_connectors(
        helper.bridge(helper.bridges(), helper.config_current_type()));
    const config_node_t* udf_connector = helper.user_defined_function_connector(udf_connectors, 0);
    const char* python_module = helper.python_module_name(udf_connector);
    const char* python_def = helper.python_def_name(udf_connector);

    argObj1 = PyFloat_FromDouble(fT);

    Py_Initialize();
    rez = python_to_cpp(python_module, python_def, 1, argObj1);
    result = PyFloat_AsDouble(rez);
    Py_Finalize();

    return result;
}
// UF_AblationRate
```

Рисунок 41 – Листинг ablationRate.cpp

Ссылка на модуль .py и название требуемой ПФ – в данном случае, def ablation из модуля ablationRate.py - получены из файла ccf.yaml через функции библиотеки configuration_helper.hpp.

Также объявлены два объекта PyObject*: argObj1 предназначен для хранения передаваемого в ПФ аргумента, rez – для хранения результата исполнения функции.

Переменная double fT преобразована в PyObject* стандартной функцией Python C API – PyFloat_FromDouble(double x).

Объявлена инициализация интерпретатора языка Python функцией Py_Initialize().

В PyObject* rez записан результат исполнения функций python_to_cpp(python_module, python_def, 1, argObj1). Т.е. входящие переменные принимают следующие значения:

- 1) (const char*) “ablationRate” – название модуля .py.
- 2) (const char*) “ablation” – название ПФ.
- 3) (int) 1 – число аргументов ПФ.
- 4) (PyObject*) argObj1 – «упакованную» переменную языка Python типа float, значение которой равно fT.

Функция python_to_cpp(...) вызывает другие функции из библиотеки PyToCpp.h в указанной на рисунке 42 последовательности.

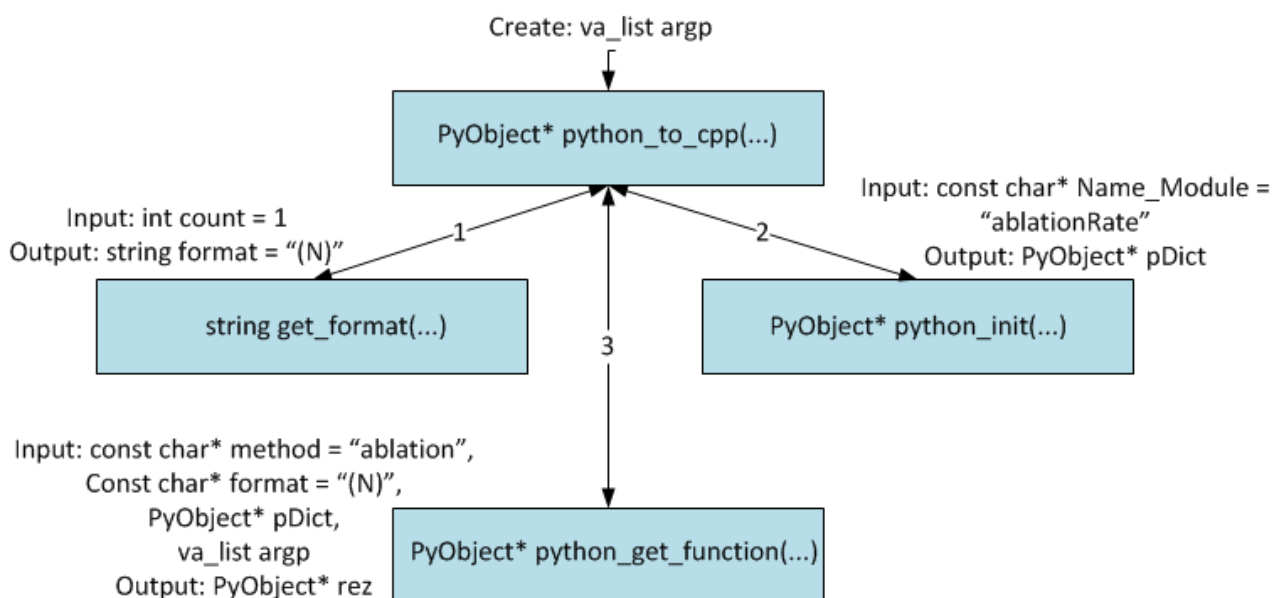


Рисунок 42 – Порядок вызова функций и обмен переменными

Функция get_format(1) формирует строку «(N)». Создается va_list argp, содержащий один аргумент; python_init(“ablationRate”) возвращает словарь

модуля `ablationRate.py`; `python_get_function("ablation", "(N)", pDict, argp)` ищет функцию `ablation` в словаре, проверяет соответствие аргумента и его формата этой ПФ и исполняет ее, возвращая полученное значение в «упаковке» `PyObject*` - в данном случае, «упакована» одна переменная типа `float`.

Полученный результат «распаковывается» стандартной функцией Python C API – `PyFloat_AsDouble` – в переменную типа `double`, которую возвращает функция `uf_ablation_rate(...)`.

Функция `Py_Finalize()` останавливает работу с интерпретатором Python и освобождает память.

Файл содержит специальные описательные метки системы Doxygen.

Библиотека `ablationRateLib.dll` была собрана со следующими настройками `CMakeLists.txt` для версии Python 3.6.5 (рисунок 43)

```
cmake_minimum_required(VERSION 3.2.3)

set(LIB_NAME ablationRateLib)
project(${LIB_NAME} C CXX)
set(CMAKE_CXX_STANDARD 98)
set (PY_PATH "C:/Python36/include")
set (PY_LIB "C:/Python36/libs")

find_package(Boost 1.61.0)
find_package(CCF REQUIRED COMPONENTS configuration)

include_directories (
    ${Boost_INCLUDE_DIR}
    ${CCF_INCLUDE_DIR}
    ${CMAKE_CURRENT_SOURCE_DIR}
    ${PY_PATH}
    ${PY_LIB}
)

file(GLOB SRCS ${CMAKE_CURRENT_SOURCE_DIR}/*.cpp)
file(GLOB HDRS ${CMAKE_CURRENT_SOURCE_DIR}/*.h)
file(GLOB PY ${CMAKE_CURRENT_SOURCE_DIR}/*.py)
file(GLOB PL ${PY_LIB}/python36_d.lib)
file(GLOB PL ${PY_LIB}/python36.lib)

add_library(${LIB_NAME} SHARED ${SRCS} ${HDRS} ${PL})

target_link_libraries(${LIB_NAME} ${CCF_LIBRARIES} ${Boost_LIBRARIES})
```

Рисунок 43 – Листинг файла `CmakeLists.txt`

В дальнейшем планируется подключение полученной библиотеки к Интегратору ЛОГОС-МИП и создание графического интерфейса для обработчика ошибок при исполнении ПФ на языке Python.

ГЛАВА 3. ЭКОНОМИЧЕСКАЯ ЧАСТЬ

ЗАКЛЮЧЕНИЕ

В ходе прохождения производственной практики в отделе 0822 ИТМФ РФЯЦ-ВНИИЭФ был изучен теоретический материал о текущем состоянии пакета программ ЛОГОС и его составной части – модульной интеграционной платформы ЛОГОС-МИП. Также был рассмотрен существующий механизм подключения динамических библиотек ПФ, написанных на языке C++, к математическим методикам.

Была обоснована необходимость расширения функционала ПФ в ЛОГОС-МИП и реализации подготовки и подключения ПФ на языке Python. Главными аргументами, доказывающими актуальность поставленной задачи, являются:

1) Возможность ускорить разработку модулей с ПФ, т.к. код на Python является интерпретируемым и не требует компиляции.

2) Переносимость модулей на языке Python и возможность их использования на любой целевой ОС без необходимости переноса модуля на удаленный компьютер для компиляции под конкретную ОС. Это позволяет упростить поддержку и сопровождение пользователя в случае возникновения ошибок в подключении и работе модулей.

3) Возможность привлечения к использованию пакета программ ЛОГОС широкого круга разработчиков на популярном языке Python.

В процессе исследования было рассмотрено несколько путей решения поставленной задачи и было произведено обоснование выбора одного из них, как наиболее подходящего к использованию в условиях РФЯЦ-ВНИИЭФ и дополняющего существующий на данный момент механизм подключения ПФ в ЛОГОС-МИП.

Был изучен механизм Python C API, позволяющий связывать между собой отдельные модули на языках C++ и Python.

Результатом работы стала разработанная технология подключения динамических библиотек пользовательских функций, написанных на языке Python, к ЛОГОС-МИП. Ее реализация была продемонстрирована на примере

ПФ `ablationRate`, изначально написанной на языке C++ и измененной соответствующим образом под язык Python.

Таким образом, можно сделать вывод, что поставленные задачи были решены, а цель данной работы полностью достигнута.

СПИСОК ИСПОЛЬЗУЕМОЙ ЛИТЕРАТУРЫ

1. Логос [Электронный ресурс]. – Режим доступа: <http://logos.vniief.ru/products/>
2. Атомная энергия. Эксперты об итогах реализации проекта «Развитие суперкомпьютеров и грид-технологий» [Электронный ресурс]. – Режим доступа: <https://www.atomic-energy.ru/statements/2011/01/31/18182>
3. Доклад «Актуальные задачи развития и внедрения технологий высокопроизводительных вычислений в ОПК и высокотехнологичные отрасли промышленности» / Гребенников А.Н. [Электронный ресурс]. – Режим доступа: <http://итопк.рф/wp-content/uploads/2019/04/Doklad--Aktualnye-zadachi-Grebennikov.pdf>
4. Страна Росатом. Корпоративная газета РФЯЦ-ВНИИЭФ №47 (189) декабрь 2018 [Электронный ресурс]. – Режим доступа: http://www.vniief.ru/resources/2688e480483504bca92fb9049005f695/47_2018.pdf
5. Rational Enterprise Management. Автоматизация проектирования № 2/2014. Применение суперкомпьютерных технологий для решения актуальных задач проектирования новых образцов авиационной техники [Электронный ресурс]. – Режим доступа: http://www.remmag.ru/upload_data/files/2-2014/Sukhoi.pdf
6. Атомная энергия. Развитие суперкомпьютеров и грид-технологий [Электронный ресурс]. – Режим доступа: <https://www.atomic-energy.ru/keywords/razvitie-superkompyutero-v-i-grid-tekhnologii>
7. Пакет программ ЛОГОС-Прочность. Реализация модели Джонсона-Холмквиста для моделирования бетонных конструкций / А. М. Гельберг, Д. Ю. Дьянов [Электронный ресурс]. – Режим доступа: <http://book.sarov.ru/wp-content/uploads/2018/12/16-molodej-v1-2018-12.pdf>
8. TAdviser. РФЯЦ-ВНИИЭФ: Пакет программ ЛОГОС [Электронный ресурс]. – Режим доступа: <https://www.tadviser.ru/index.php/Продукт:РФЯЦ-ВНИИЭФ:ПакетПрограммЛогос>

9. Подсистема управления расчетными данными Интегратора ЛОГОС МИП / Е. А. Ескова, А. Д. Черевань, А. Г. Надуев, Д. А. Кожаяев [Электронный ресурс]. – Режим доступа: <http://book.sarov.ru/wp-content/uploads/2020/02/17-molodej-2019-86.pdf>

10. CodeRoad. Что такое функция обратного вызова? [Электронный ресурс]. – Режим доступа: <https://coderoad.ru/824234/Что-такое-функция-обратного-вызова>

11. Ад обратных вызовов [Электронный ресурс]. – Режим доступа: <http://callbackhell.ru/>

12. Хабр. Ликбез по типизации в языках программирования [Электронный ресурс]. – Режим доступа: <https://habr.com/ru/post/161205/>

13. Механизм пользовательских функций модульной интеграционной платформы ЛОГОС / Тюндина А.А., Побуринная Н.А., Кожаяев Д.А.

14. Реализация функционального блока пользовательских функций для задач аэрогидродинамики, позволяющих динамическое подключение и внесение изменений в стандартные параметры задачи в процессе моделирования в программном модуле ЛОГОС-МИП: научно-технический отчет / Кожаяев Д.А., Надуев А.Г., Шемякин А.В., Черевань А.Д., Побуринная Н.А., Вараксин Г.В., Киселева Е.Б., Дьяков А.В., Тюндина А.А., Кочкина О.Ю., Ескова Е.А. - Инв.№ 8/28008 нс. – Саров, 2018. 37 с.

15. Реализация функционального блока пользовательских функций для задач прочностного анализа, позволяющих динамическое подключение и внесение изменений в стандартные параметры задачи в процессе моделирования в программном модуле ЛОГОС-МИП: научно-технический отчет / Надуев А.Г., Шемякин А.В., Жуков Д.А., Черевань А.В., Дьяков А.В. - Инв.№ 8/28698 нс. – Саров, 2020. 64 с.

16. Институт прикладной автоматизации и программирования. Язык программирования Python: особенности и преимущества [Электронный ресурс]. – Режим доступа: <https://ipap.ru/poleznye-stati/4-useful/yazyk-programmirovaniya-python-osobennosti-i-preimushchestva>

17. TAdviser. Рейтинг востребованности языков программирования – Tiobe [Электронный ресурс]. – Режим доступа: https://www.tadviser.ru/index.php/Статья:Рейтинг_востребованности_языков_п_рограммирования#2018

18. Программирование на языке высокого уровня Python / Петров Ю. [Электронный ресурс]. – Режим доступа: https://www.yuripetrov.ru/edu/python/ch_05_01.html#python

19. OTUS: Онлайн-образование. Функции Python: 7 примеров. Базовые, встроенные и пользовательские функции [Электронный ресурс]. – Режим доступа: <https://otus.ru/nest/post/1107/>

20. Документация Python 3.8.3. distutils – Сборка и установка модулей Python [Электронный ресурс]. – Режим доступа: <https://docs.python.org/3/library/distutils.html>

21. Python Package Index. Py2exe 0.9.2.2 [Электронный ресурс]. – Режим доступа: <https://pypi.org/project/py2exe/>

22. Cython 3.0a5 documentation [Электронный ресурс]. – Режим доступа: <https://cython.readthedocs.io/en/latest/>

23. Paperspace Blog. Boosting Python Scripts With Cython [Электронный ресурс]. – Режим доступа: <https://blog.paperspace.com/boosting-python-scripts-cython/>

24. Cython: C-Extensions for Python [Электронный ресурс]. – Режим доступа: <https://cython.org/>

25. Intermediate Python. Python C расширения / Каратаев П. [Электронный ресурс]. – Режим доступа: https://pavel-karateev.gitbook.io/intermediate-python/raznoe/python_c_extension

26. Документация Python 3.8.3. Справочное руководство по Python C API [Электронный ресурс]. – Режим доступа: <https://docs.python.org/3/c-api/intro.html>

27. Интуит. Лекция 13: Интеграция Python с другими языками программирования [Электронный ресурс]. – Режим доступа: <https://www.intuit.ru/studies/courses/49/49/lecture/27082>

28. Хабр. Введение в CMake. [Электронный ресурс]. – Режим доступа: <https://habr.com/ru/post/155467/>

ПРИЛОЖЕНИЕ 1. ЛИСТИНГ ПРОЕКТА

ablationRate.py

```
from math import exp

def ablation(fT):
    return fT*(exp(-fT/680)+0.15)/700

def mathAll(x,y):
    return x+y, x-y, x*y, x/y

def indexString(x):
    return 'file'+x+'.txt'
```

ablationRate.cpp

```
#include "configuration_helper.hpp"
#include "Python.h"
#include "PyToCpp.h"
#include «ablationRate.h»

/** @brief Скорость массового уноса
 * @param fT значение температуры в центре ячейки
 * @param centerX координаты центра ячейки
 * @param centerY координаты центра ячейки
 * @param centerZ координаты центра ячейки
 * @param normalX координаты проекции центра ячейки на нормаль
 * @param normalY координаты проекции центра ячейки на нормаль
 * @param normalZ координаты проекции центра ячейки на нормаль
 * @user_function
 */

double uf_ablation_rate( double fT,
```

```

        double centerX,
        double centerY,
        double centerZ,
        double normalX,
        double normalY,
        double normalZ)
    {
        double result = 0.0;
        PyObject *argObj1, *rez;

        ccf::configuration_helper_t helper;
        const          config_node_t*          udf_connectors          =
helper.user_defined_function_connectors(helper.bridge(helper.bridges(),
helper.config_current_type()));
        const          config_node_t*          udf_connector          =
helper.user_defined_function_connector(udf_connectors, 0);
        const          char*                   python_module          =
helper.python_module_name(udf_connector);
        const char* python_def = helper.python_def_name(udf_connector);

        argObj1 = PyFloat_FromDouble(fT);

        Py_Initialize();
        rez = python_to_cpp(python_module, python_def, 1, argObj1);
        result = PyFloat_AsDouble(rez);
        Py_Finalize();

        return result;
    }

```

```
#include <Python.h>
#include <string>
using std::string;

PyObject* python_init(const char* Name_Module){
    /*Получение словаря функций pDict*/
    PyObject *pName, *pModule, *pDict;
    pName = PyUnicode_FromString(Name_Module);
    pModule = PyImport_Import(pName);
    if (!pModule) {
        printf("Error_pModule");
        PyErr_Print();
    }
    pDict = PyModule_GetDict(pModule);
    if (!pDict) {
        printf("Error_pDict");
        PyErr_Print();
    }
    return pDict;
}

string get_format(int count){
    string format = "(";
    for (int i = 0; i < count; i++)
        format = format + « N » ;
    format = format + « ) » ;
    return format;
}
```

```

PyObject* python_get_function(const char* method, const char* format,
                               PyObject* pDict, va_list argp){
    PyObject *pObj, *pVal, *pArgs;
    pObj = PyDict_GetItemString(pDict, method);
    if (!pObj)
        PyErr_Print();
    pArgs = Py_VaBuildValue(format, argp);
    if(pArgs == NULL)
        PyErr_Print();
    pVal = PyObject_CallObject(pObj, pArgs);
    if (pVal == NULL)
        PyErr_Print();
    return pVal;
}

```

```

PyObject* python_to_cpp(const char* Name_Module, const char* method,
                        int count, ...){
    PyObject *rez, *pDict;
    const char* format;
    string fString = get_format(count);
    format = fString.c_str();
    va_list argp;
    va_start(argp, count);
    pDict = python_init(Name_Module);
    rez = python_get_function(method, format, pDict, argp);
    va_end(argp);
    return rez;
}

```



```
cmake_minimum_required(VERSION 3.2.3)
```

```
set(LIB_NAME ablationRateLib)
```

```
project(${LIB_NAME} C CXX)
```

```
set(CMAKE_CXX_STANDARD 98)
```

```
set (PY_PATH "C:/Python36/include")
```

```
set (PY_LIB "C:/Python36/libs")
```

```
find_package(Boost 1.61.0)
```

```
find_package(CCF REQUIRED COMPONENTS configuration)
```

```
include_directories (
```

```
    ${Boost_INCLUDE_DIR}
```

```
    ${CCF_INCLUDE_DIR}
```

```
    ${CMAKE_CURRENT_SOURCE_DIR}
```

```
    ${PY_PATH}
```

```
    ${PY_LIB}
```

```
)
```

```
file(GLOB SRCS ${CMAKE_CURRENT_SOURCE_DIR}/*.cpp)
```

```
file(GLOB HDRS ${CMAKE_CURRENT_SOURCE_DIR}/*.h)
```

```
file(GLOB HDRS ${CMAKE_CURRENT_SOURCE_DIR}/*.py)
```

```
file(GLOB PL ${PY_LIB}/python36_d.lib)
```

```
file(GLOB PL ${PY_LIB}/python36.lib)
```

```
add_library(${LIB_NAME} SHARED ${SRCS} ${HDRS} ${PL})
```

```
target_link_libraries(${LIB_NAME} ${CCF_LIBRARIES}
```

```
    ${Boost_LIBRARIES})
```