Унифицированный программный интерфейс для численной оптимизации на C++

А. Ю. Савинков, email: savinkov_a_yu@sc.vsu.ru

Федеральное государственное бюджетное образовательное учреждение высшего образования «Воронежский государственный университет» (ФГБОУ ВО «ВГУ»)

Аннотация. *В данной работе предложена реализация абстрактного унифицированного интерфейса для объекта численной оптимизации на основе класса C++. Иинтерфейс совместим практически с любым алгоритмом численной оптимизации, позволяет скрыть детали реализации алгоритма оптимизации и предоставить интерфейс к нему в стиле функции или объекта-функции.*

Ключевые слова: *язык программирования C++, объектно-ориентированное программирование, численная оптимизация.*

Введение

Оптимизационные задачи достаточно часто возникают в практике цифровой обработки сигналов и при проектировании телекоммуникационных систем. Эти задачи как правило требуют достаточно больших объемов вычислений и их реализацию целесообразно было бы выполнять на языке C++.

Процедуры численной оптимизации требуют определения множества сопутствующих параметров, от размерности задачи и ограничения на число итераций до определения функции стоимости в виде некоторой вычислительной процедуры. В результате создается специализированный программный код, ориентированный на решение одной частой задачи, в котором реализация алгоритма оптимизации смешивается с реализацией алгоритмов прикладной задачи. Повторное использование такого кода в новых проектах затруднено, так как требует глубокой переработки реализации.

В докладе предлагается реализация на языке C++ обобщенного абстрактного программного интерфейса для произвольной процедуры оптимизации, позволяющая обеспечить переносимость и удобство использования процедур численной оптимизации в широком классе оптимизационных задач.

1. Требования к интерфейсу процедуры численной оптимизации

При решении оптимизационных задач в области цифровой обработки сигнала (ЦОС) или проектирования телекоммуникационных систем чаще всего возникает два сценария применения численной оптимизации:

1. непрерывная оптимизация – регулярное выполнение одного или нескольких шагов численной оптимизационной процедуры на протяжении всего времени работы системы ЦОС для постоянной подстройки параметров под изменение внешних условий (например, подстройка эквалайзера);
2. однократная оптимизация – выполнение полной оптимизационной процедуры для получения оценки параметров сигнала (например, оценка направления прихода сигнала) или оптимизации телекоммуникационной системы на этапе проектирования (например, оптимизация разрядности представления данных).

Обобщенный интерфейс должен быть одинаково удобен для применения в обоих сценариях. При решении реальных задач в рассматриваемой предметной области чаще находят применение методы прямой численной оптимизации (методы поиска), так как явная запись функции стоимости в аналитической форме обычно неизвестна. Поэтому будем строить интерфейс применительно к прямым методам оптимизации. Сформулируем требования к обобщенному интерфейсу процедуры численной оптимизации:

– интерфейс должен устанавливать общие параметры оптимизационной задачи – размерность, шаг оптимизации, требуемая точность, ограничение на число итераций, координаты начальной точки в пространстве оптимизации;

– должны быть предусмотрены различные варианты установки начальной точки, от простого перечисления координат до использования произвольных контейнеров, включая списки и массивы;

– интерфейс должен предусматривать разные варианты задания функции стоимости: указатель на функцию, объект-функция или λ-выражение;

– для поддержки режима непрерывной оптимизации удобно реализовать объект (класс), реализующий процедуру численной оптимизации, чтобы один раз установить параметры, включая функцию стоимости, и сохранять результаты предыдущего шага оптимизации для использования их в качестве начальных условий на следующем шаге;

– для поддержки режима однократной оптимизации более удобен процедурный интерфейс (просто вызов функции без каких-либо

дополнительных действий по созданию и конфигурированию экземпляра класса).

2. Реализация абстрактного интерфейса для произвольной процедуры численной оптимизации

Предлагаемая реализация интерфейса, удовлетворяющая предъявленным требованиям, приведена в листинге 1.

Листинг 1

```
template <typename unit_t>
class optimizer_interface
{
protected:

  optimizer_interface():
  m_dimension(0),
  m_initial_step(std::numeric_limits<double>::quiet_NaN()),
  m_target_accuracy(std::numeric_limits<double>::
    quiet_NaN()),
  m_iteration_limit(0),
  m_continuous_optimization(false),
  m_optimum_cost(std::numeric_limits<double>::quiet_NaN())
  {}

public:

  /*
  установка параметров оптимизации
  */

  // установка параметров с возможностью установки
  // координат начальной точки через список инициализации
  void setup(
    size_t dimension,
    double initial_step,
    double target_accuracy,
    size_t iteration_limit,
    std::initializer_list<unit_t> starting_point =
       std::initializer_list<unit_t>());

  // установка параметров с установкой координат
  // начальной точки из массива (по указателю)
  void setup(
    size_t dimension,
    double initial_step,
    double target_accuracy,
    size_t iteration_limit,
    const unit_t* starting_point);

  // установка параметров оптимизации с установкой
  // координат начальной точки из произвольного контейнера
```

```
template <typename starting_point_t,
typename = std::enable_if_t<
  !std::is_pointer<starting_point_t>::value,
  starting_point_t>>
void setup(
  size_t dimension,
  double initial_step,
  double target_accuracy,
  size_t iteration_limit,
  const starting_point_t& starting_point);

// начальная точка через список инициализации
void set_starting_point(
  std::initializer_list<unit_t> starting_point);

// установка координат начальной точки из массива,
// заданного указателем на первый элемент
void set_starting_point(const unit_t* starting_point);

// установка начальной точки из произвольного контейнера
template <typename starting_point_t,
typename = std::enable_if_t<
  !std::is_pointer<starting_point_t>::value,
  starting_point_t>>
void set_starting_point(
  const starting_point_t& starting_point);

// установка функции стоимости
template <typename proc_t>
void set_proc(proc_t proc);

// результат оптимизации в качестве новой начальной точки
void move_starting_point_to_end_point();

// режим автоматической установки результата оптимизации
// в качестве новой начальной точки
void set_continuous_optimization(bool mode);

/*
запуск оптимизации
*/

// выполняет процедуру оптимизации для предварительно
// установленных начальной точки и функции стоимости
const std::vector<unit_t>& operator()();

// выполняет оптимизацию с указанной функцией стоимости
// для предварительно установленной начальной точки
template <typename proc_t>
const std::vector<unit_t>& operator()(proc_t proc);

// выполняет оптимизацию с указанной функцией стоимости
```

```
  // и координатами начальной точки из списка инициализации
  template <typename proc_t>
  const std::vector<unit_t>& operator()(proc_t proc,
    std::initializer_list<unit_t> starting_point);

  // выполняет оптимизацию с указанной функцией стоимости и
  // начальной точкой из массива
  template <typename proc_t>
  const std::vector<unit_t>& operator()(proc_t proc,
     const unit_t* starting_point);

  // выполняет оптимизацию с указанной функцией стоимости
  // и начальной точкой из произвольного контейнера
  template <typename proc_t, typename starting_point_t,
  typename = std::enable_if_t<
    !std::is_pointer<starting_point_t>::value,
    starting_point_t>>
  const std::vector<unit_t>& operator()(proc_t proc,
    const starting_point_t& starting_point);

  /*
  результаты оптимизации
  */

  // возвращает координаты точки - результата оптимизации
  const std::vector<unit_t>& get_point();

  // возвращает результирующее значение функции стоимости
  double get_cost();

  // возвращает результат оптимизации через преобразование
  // типа для поддержки вызовов в стиле функции
  operator const std::vector<unit_t> ();

  // преобразование типа для одномерной задачи
  operator const unit_t ();

  // возвращает результаты оптимизации и значение
   // функции стоимости вызовов в стиле функции
  operator const std::pair<std::vector<unit_t>, double> ();

protected:

  // предварительно установленная функция стоимости
  std::function<double(const std::vector<unit_t>&)>
    m_proc_wrapper;

  // паремтры оптимизации
  size_t m_dimension;
  double m_initial_step;
  double m_target_accuracy;
  size_t m_iteration_limit;
```

```
  std::vector<unit_t> m_starting_point;
  bool m_continuous_optimization;

  // результаты оптимизации
  std::vector<unit_t> m_optimum_point;
  double m_optimum_cost;

  /*
  реализация процедуры оптимизации
  */

  virtual void optimize(
    std::function<double(const std::vector<unit_t>&)>
      proc,
    const std::vector<unit_t>& starting_point) = NULL;
};
```

Некоторые пояснения к реализации интерфейса.

Класс optimizer_interface является абстрактным классом [1], у него нет public-конструктора и отсутствует реализация виртуальной функции optimize. Для реализации класса-оптимизатора необходимо использовать наследование. В дочернем классе необходимо определить реализацию виртуальной функции optimize в соответствии с выбранным алгоритмом оптимизации. Параметры оптимизационной задачи и координаты начальной точки реализация функции optimize должна брать из protected-параметров класса optimizer_interface. Другим обязательным требованием к реализации функции optimize является установка значений m_optimum_point и m_optimum_cost в соответствии с результатом оптимизации.

Чтобы координаты начальной точки можно было установить как через массив, заданный указателем на первый элемент, так и через стандартный контейнер STL [2], например, vector или list, в интерфейсе определены две функции (листинг 2):

Листинг 2

```
void set_starting_point(const unit_t* starting_point);

template <typename starting_point_t>
void set_starting_point(
  const starting_point_t& starting_point);
```

Но при генерации кода из шаблона, если в качестве аргумента функции использовать указатель, шаблон второй функции также подходит, что приводит к неверному выбору и ошибке при компиляции. Чтобы запретить компилятору выбирать вторую функцию для указателей, используется шаблон enable_if, появившийся впервые в C++11 и использующий механизм SFINAE (Substitution Failure Is Not An

Error) [3] для подавления генерации шаблонной функции при выполнении определенных условий, в нашем случае, если тип starting_point_t является указателем (см. листинг 3):

Листинг 3

```
template <typename starting_point_t
typename = std::enable_if_t<
  !std::is_pointer<starting_point_t>::value,
  starting_point_t>>
void set_starting_point(
  const starting_point_t& starting_point);
```

Этот же подход используется при определении функций setup и операторов круглая скобка. Таким образом реализуется требование к интерфейсу о множестве вариантов установки координат начальной точки в пространстве оптимизации.

Другим требованием к интерфейсу было обеспечить различны варианты задания функции стоимости. Для реализации этого требования используется стандартная обертка функции из STL (листинг 4)

Листинг 4

```
std::function<double(
  const std::vector<unit_t>&)> m_proc_wrapper;
```

При этом требования к функции стоимости ограничиваются возвращаемый тип double и наличием единственный входного аргумента типа const std::vector<unit_t>& с координатами точки, для которой нужно вычислить функцию стоимости. В качестве функции стоимости может использоваться указатель на функцию, объект-функция или λ-выражение.

Для использования экземпляра класса оптимизатора в стиле функции необходимо, во-первых, реализовать оператор преобразования типа, чтобы можно было использовать экземпляр класса оптимизатора в правой части оператора присваивания. В нашем случае реализован оператор приведения типа optimizer_interface к типу vector и при попытке присваивания экземпляра класса оптимизатора вектору в векторе будут автоматически сохранены координаты найденной оптимальной точки.

В предлагаемой реализации предусмотрено еще два оператора приведения типа (листинг 5):

Листинг 5

```
operator const unit_t ();
operator const std::pair<std::vector<unit_t>, double> ();
```

Первый оператор приведения типа возвращает первый элемент вектора координат оптимальной точки (результата оптимизации), что может быть полезно для одномерных задач, чтобы избежать использование вектора из одного элемента для представления результата оптимизации. Достаточно использовать числовой тип, например, double.

Второй оператор приведения типа возвращает пару: координаты оптимальной точки как вектор и значение функции стоимости в оптимальной точке, что может быть полезно в задачах ЦОС.

Во-вторых, для использования экземпляра класса оптимизатора в стиле функции необходимо предусмотреть конструктор, который задаст все необходимые параметры, включая функцию стоимости, и запустит процедуру оптимизации. Поскольку класс optimizer_interface является абстрактным, конструкторы должны определяться в производных классах. Для упрощения этой задачи определен макрос __prepare_optimizer_interface_constructors, показанный в листинге 6:

Листинг 6

```
#define __prepare_optimizer_interface_constructors(name, type) \
name##(): \
      optimizer_interface<##type##>() \
{} \
template <class proc_t> \
name##(proc_t proc, size_t dimension, double initial_step, \
      double target_accuracy, size_t iteration_limit, \
      std::initializer_list<##type##> starting_point = \
            std::initializer_list<##type##>(), \
      bool run_optimization = true): \
      optimizer_interface<##type##>() \
{ \
      setup(dimension, initial_step, target_accuracy, \
            iteration_limit, starting_point); \
      set_proc(proc); \
      if (run_optimization) { \
            optimize(m_proc_wrapper, m_starting_point); } \
} \
template <class proc_t, class starting_point_t> \
name##(proc_t proc, size_t dimension, double initial_step, \
      double target_accuracy, size_t iteration_limit, \
      const starting_point_t& starting_point, \
      bool run_optimization = true): \
      optimizer_interface<##type##>() \
{ \
      setup(dimension, initial_step, target_accuracy, \
            iteration_limit, starting_point); \
      set_proc(proc); \
      if (run_optimization) { \
            optimize(m_proc_wrapper, m_starting_point); } \
```

```
}
```

Макрос получает два параметра: имя производного класса и тип шаблона класса optimizer_interface. Макрос генерирует конструктор по умолчанию, конструктор с возможностью указания координат начальной точки через список инициализации и конструктор с инициализацией координат начальной точки через массив, заданный указателем на первый элемент, или через любой стандартный контейнер STL. Последний параметр конструктора, run_optimization, имеет значение по умолчанию true, и если этот параметр при вызове конструктора не указывать, то процедура оптимизации будет запущена автоматически, что и требуется для использования объекта оптимизатора в стиле функции.

Заметим также, что предлагаемый базовый интерфейс предназначен для поддержки безусловной оптимизации, ограничения для условной оптимизации или дополнительные параметры должны определяться в произвольном классе.

Для использования предложенного интерфейса требуется версия компилятора с поддержкой не ниже C++14.

Заключение

Описанный программный интерфейс для численной оптимизации реализован в среде Visual Studio 2019 и хорошо показал себя при моделировании различных систем цифровой обработки сигнала.

В качестве примера использования интерфейса рассмотрим поиск минимума функции Розенброка [4] с использованием алгоритма численной оптимизации Нелдера-Мида [5]. Будем использовать интерфейс в стиле вызова функции. Функция стоимости определена через λ-выражение, размерность задачи 2, начальный шаг оптимизации 0.1, требуемая точность 1e-6, предельное число итераций 1000, начальная точка (-1, -1), см. листинг 7.

Листинг 7

```
class nelder_mead_optimizer:
  public optimizer_interface<double>
{
public:

  __prepare_optimizer_interface_constructors
  (nelder_mead_optimizer, double)

protected:

  virtual void optimize(
    std::function<double(const std::vector<double>&)>
```

```
      proc,
    const std::vector<double>& starting_point)
  {
    // реализация алгоритма Нелдера-Мида [4]
  }
};

int main()
{
  try
  {
    auto rozenbrok_func =
      [](const std::vector<double>& x)
    {
      return 100.0 * pow(x[1] - x[0] * x[0], 2.0)
        + pow(1 - x[0], 2.0);
    };

    std::vector<double>
      opt = nelder_mead_optimizer(rozenbrok_func,
        2, 0.1, 1e-6, 1000, { -1, -1 });
  }
  catch (std::exception& e)
  {
    puts(e.what());
  }

  return 0;
}
```

Список литературы

1. Страуструп Б. Программирование: принципы и практика с использованием C++ / Бьярне Страуструп – 2-е изд.: Пер. с англ. – М.: Вильямс, 2016. – 1328 с.
2. Джосьютис Н. C++. Стандартная библиотека / Н. Джосьютис – Пер. с англ. – СПб.: Питер, 2004. – 730 с.
3. Вандевурд Д. Шаблоны C++. Справочник разработчика / Дэвид Вандевурд, Николай М. Джосаттис – Пер. с англ. – М.: Вильямс, 2016, 544 с.
4. Rosenbrock H.H. An Automatic Method for Finding the Greatest or Least Value of a Function / H.H. Rosenbrock // The Computer Journal – 1960 – Vol #3, Iss. 3, pp. 175–184.
5. Nelder J.A. A Simplex Method for Function Minimization / J.A. Nelder, R. Mead // The Computer Journal – 1965 – Vol #7, Iss. 4, pp. 308–313.