Учебный курс "Введение в методы программирования"

Лабораторный практикум

Лабораторная работа 2: Суммирование рядов

1. Постановка учебно-практической задачи

При выполнении лабораторной работы требуется разработать программы численного *вычисления значений основных аналитических функций* (типа `sin`, `cos`, `exp` и др.) путем суммирования имеющихся представлений для этих функций в виде *рядов Тейлора* [1]. Получаемый в результате разработки программный комплекс должен обеспечивать вычисление значений для некоторого определенного набора функций и позволять выполнение вычислительного эксперимента для изучения и исследования скорости *сходимости рядов* и *точности* получаемых значений функций. При проведении экспериментов необходимо иметь возможность:

- выбора функции (из числа реализованных) для исследования;
- задания расчетных параметров (значения аргумента функции и количества суммируемых членов ряда);
- выполнения расчетов (суммирования указанного отрезка ряда Тейлора);
- вывода результатов вычислений (приближенного и точного значений функции, погрешности расчетов).

Пусть, например, выбрана функция `sin`, и требуется вычислить ее приближенное значение в точке `x=1.57` за `N=4` шага. Результаты расчетов должны быть представлены в следующем виде:

```
Выбранная функция:              SIN
Аргумент:                       1.57
Количество членов ряда:         4
Приближённое значение:          0.9998
Точное значение функции:        1.0000
Погрешность вычисления:         0.0002
```

Рис. 2.1. Результаты выполнения расчетов

Разработанный программный комплекс должен обеспечивать также и возможность проведения серийных экспериментов, когда вычисление значения выбранной функции осуществляется несколько раз с последовательно увеличивающимся количеством членов суммируемого ряда. Результаты вычислений в этом случае должны быть представлены в табличной форме; погрешность получаемых значений для лучшей наглядности целесообразно представить в виде столбиковой диаграммы. В качестве возможного варианта вывода можно использовать экранную форму следующего вида:

Кол-во итераций	Частичная сумма	Абсолютная погрешность: SIN (логарифмическая шкала)
1	1.570000	
2	0.925018	
3	1.004509	
4	0.999843	
5	1.000003	
6	1.000000	
7	1.000000	
Точное значение	1.000000	

Рис. 2.2. Результаты выполнения последовательности расчетов

2. Учебно-методические цели работы

Выполнение данной работы, помимо рассмотрения на примере суммирования рядов принципов программирования итерационных алгоритмов, направлено на достижение следующих учебно-методических целей:

- закрепление навыков программирования циклических алгоритмов;
- получение практических навыков использования процедурного типа в алгоритмическом языке Zonnon;
- практическое освоение методов реализации итерационных алгоритмов, использующих рекурсивно описанное решающее правило;
- получение общего представления о методах визуализации результатов вычислений;
- начальное знакомство с принципами проведения вычислительных экспериментов с целью изучения и исследования применяемых алгоритмов.

3. Рекомендации по выполнению работы

3.1. Представление аналитических функций в виде рядов Тейлора

Для вычисления значений аналитической функции можно воспользоваться ее представлением в виде степенного ряда

$$f(x)=f(\xi)+f'(\xi)(x-\xi)+\frac{f''(\xi)}{2!}(x-\xi)^2+\ldots+\frac{f^{(n)}(\xi)}{n!}(x-\xi)^n+R_n(x),$$

обычно называемого *рядом Тейлора*. Необходимым и достаточным условием существования такого разложения функции $f(x)$ в некоторой окрестности $|x-\xi|<r$ точки ξ является существование для $f(x)$ всех производных в этой окрестности и сходимость *остаточного члена* $R_n(x)$ к нулю при $n\to\infty$ [1].

В случае элементарных функций ряды Тейлора имеют вид (степенные ряды при $\xi=0$ представлены в *форме Маклорена*):

$$\sin x = x-\frac{x^3}{3!}+\frac{x^5}{5!}-\ldots+(-1)^n\frac{x^{2n+1}}{(2n+1)!}+\ldots|x|<\infty$$

$$\cos x = 1-\frac{x^2}{2!}+\frac{x^4}{4!}-\ldots+(-1)^n\frac{x^{2n}}{(2n)!}+\ldots\ |x|<\infty$$

$$e^x=1+x+\frac{x^2}{2!}+\ldots+\frac{x^n}{n!}+\ldots\ |x|<\infty$$

и т.д. (дополнительная информация по рядам Тейлора представлена, например, в [1]).

3.2. Общая схема алгоритма суммирования

При наличии представления функции $f(x)$ в виде ряда Тейлора проблема вычисления значений $f(x)$ в точках интервала $(-r,r)$ может быть сведена к задаче нахождения *частичных* сумм числового ряда общего вида:

$$Sum=\sum_{i=1}^{n}a_i$$

Общая схема *алгоритма суммирования* может быть представлена в виде:

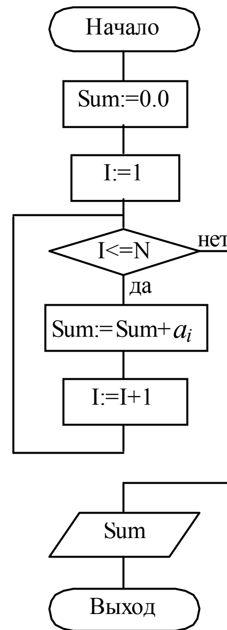

Рис. 2.3. Общая схема алгоритма

Как следует из приведенной блок-схемы, алгоритм суммирования представляет собой *итерационную процедуру*, на каждой итерации которой к переменной *Sum* прибавляется очередной член ряда a_i. Как результат, в переменной *Sum* (называемой далее *сумматором*) обеспечивается накопление частичных сумм числового ряда. Величина N, используемая в общей схеме алгоритма для задания момента завершения цикла суммирования, определяет количество суммируемых членов ряда и может рассматриваться как *показатель требуемой точности* вычисления значения функции.

3.3. Учет рекуррентной зависимости членов ряда

Выполнение итерации алгоритма суммирования включает вычисление значения очередного члена числового ряда. Данные вычисления могут оказаться достаточно трудоемкими. Так, например, общий член ряда при разложении функции `sin(x)` имеет вид:

$$a_i = (-1)^i \frac{x^{2i+1}}{(2i+1)!}$$

и для его вычисления необходим подсчет значений степенной функции и функции факториала. С другой стороны, во многих случаях суммирования рядов соседние пары элементов имеют общую вычислительную часть и, как результат, каждый последующий член ряда может быть получен быстрее и с меньшими затратами при учете имеющихся значений предыдущих элементов. Так, для функции `sin(x)` подобная *рекуррентная формула* имеет вид:

$$a_i = a_{i-1} \cdot q_i, \; где \; q_i = \frac{-x^2}{2i(2i+1)}.$$

При наличии подобного рекуррентного соотношения между членами ряда генерация суммы числового ряда может быть обеспечена при использовании только начального члена ряда a_0 и рекуррентного множителя q_i. Уточненный вариант алгоритма имеет вид:

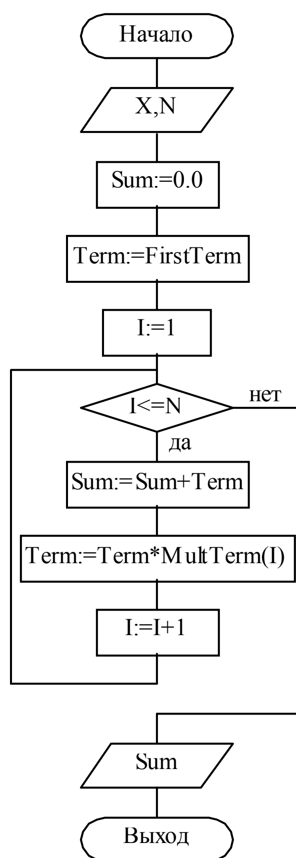

Рис. 2.4. Уточненная блок-схема алгоритма суммирования
(учет рекуррентности),

где `FirstTerm` - начальный член ряда a_0,
`Term` - очередной вычисляемый член ряда a_i,
`MultTerm` - рекуррентный множитель q_i,
`Sum` - искомая частичная сумма.

3.4. Программная реализация алгоритма суммирования

Алгоритм суммирования, приведенный на рис. 2.4, может быть реализован в виде подпрограммы-функции, имеющей следующий возможный формат строки заголовка:

```
    {вычисление частичной суммы ряда}
procedure sumSeries( ns:integer{8}; x:real; n:integer{16} ) : real;
```

параметры подпрограммы задают необходимые данные для работы алгоритма суммирования:

`ns` - номер ряда для суммирования;
`x` - значение аргумента;
`n` - количество членов ряда, которые нужно просуммировать.

В качестве выходного значения функции `sumSeries` должен возвращаться результат вычисления суммы указанного количества членов ряда (*значение частичной суммы ряда*).

При выполнении лабораторной работы можно предполагать, что ряды, подлежащие суммированию, перенумерованы и, тем самым, порядковый *номер ряда* может быть использован для однозначного указания необходимого для суммирования ряда.

3.5. Создание функциональных обработчиков рядов

Основная проблема при реализации программы `sumSeries` состоит в том, что, несмотря на единство общей схемы суммирования, числовые ряды содержат и различающиеся (*функционально-зависимые*) части, среди которых начальный член ряда a_0 и

рекуррентный множитель q_i - см. рис. 2.4. Возможный подход к программированию в таких условиях может состоять в подготовке набора функций суммирования для каждого числового ряда в отдельности (например, для вычисления значений функции `sin(x)` реализуется программа `sumSinSeries`, для функции `cos(x)` - программа `sumCosSeries` и т.д.). Такой вариант разработки является достаточно простым, но приводит к многократному дублированию алгоритма суммирования в создаваемых программах.

При выполнении лабораторной работы предлагается использовать более эффективный способ построения программ, при котором:

- алгоритм суммирования реализуется однократно в виде подпрограммы-функции `sumSeries`;
- функционально-зависимые части рядов выносятся из состава программы суммирования и оформляются в виде отдельных вспомогательных подпрограмм;
- для получения необходимых характеристик числового ряда функция суммирования использует реализованные для рядов вспомогательные подпрограммы.

Возможный подход для реализации рассмотренной схемы может состоять в следующем:

– для определения начального члена ряда a_0 разрабатывается функция

```
procedure firstTerm(x:real):real;
```

– вычисление значения рекуррентного множителя обеспечивается при помощи подпрограммы-функции

```
procedure multTerm(x:real; num:integer):real;
```

– для определения предельного значения суммируемого ряда (точного значения функции) подготавливается функция

```
procedure sumLimit(x:real):real;
```

Функция `sumLimit` введена в число описывающих числовой ряд программ для возможности определения *погрешности* проводимых вычислений с целью изучения и исследования скорости сходимости суммируемых рядов. Для получения точного значения вычисляемых функций можно использовать, например, стандартные функции алгоритмического языка Zonnon.

Перечисленный набор функций `firstTerm`, `multTerm` и `sumLimit` после своей реализации полностью характеризуют конкретный числовой ряд; каждый суммируемый ряд должен иметь свой соответствующий комплект этих функций. Для более простого использования функции `firstTerm`, `multTerm` и `sumLimit`, обеспечивающие вычисление характеристик одного и того же числового ряда, целесообразно объединить в рамках единой подпрограммы, возможный вид которой может быть следующим:

```
procedure funcSeries (x:real; num:integer):real;
  procedure sumLimit (x:real):real;
  procedure firstTerm(x:real):real;
  procedure multTerm (x:real; num:integer):real;
  begin  {тело обработчика}
    case num of
      SUM_LIMIT:   return sumLimit(x);
      | FIRST_TERM:  return firstTerm(x);
    else           return multTerm(x,num);
    end; {case}
end funcSeries;
```

В результате подобного укрупнения программ каждому отдельному числовому ряду будет соответствовать единая функция типа `funcSeries` (будем называть далее эти функции *функциональными обработчиками рядов*). С другой стороны, функции `firstTerm`, `multTerm` и `sumLimit` становятся локальными подпрограммами объединенной программы, и получение характеристик ряда оказывается возможным только через обращение общего вида к подпрограмме - функциональному обработчику ряда. Различение ситуаций вызова можно выполнить при помощи параметра `num` функции `funcSeries`. В общем случае этот параметр должен содержать номер суммируемого элемента ряда (номер итерации

суммирования); для выделения моментов обращения к функциям firstTerm и sumLimit можно использовать для параметра num некоторые исключительные (например, отрицательные) значения. Тогда алгоритм выполнения функции funcSeries может быть следующим (см. текст программы):

– при значении параметра num, равном SUM_LIMIT, выполняется вызов функции sumLimit;

– при значении параметра num, равном FIRST_TERM, выполняется вызов функции firstTerm;

– при всех остальных значениях параметра num выполняется вызов функции multTerm.

В качестве возможных значений констант SUM_LIMIT и FIRST_TERM могут быть выбраны величины:

```
const
  FIRST_TERM = 0;
  SUM_LIMIT  = max (integer);
```

Приведем для примера реализацию программы функционального обработчика ряда для функции sin(x).

```
procedure sinSeries(x:real; num:integer):real;
  procedure firstTerm(x:real):real;
    begin
      return x;
  end firstTerm;
  procedure multTerm(x:real; num:integer):real;
    begin
      return -(x*x)/(2*num*(2*num+1));
  end multTerm;
  procedure sumLimit(x:real):real;
    begin
      return sin(x);
  end sumLimit;
  begin
   {тело обработчика}
    case num of
      SUM_LIMIT:  return sumLimit(x);
      | FIRST_TERM: return firstTerm(x);
      else        return multTerm(x,num);
    end; {case}
end sinSeries;
```

Подводя итог обсуждению, сведем воедино **правила разработки** программ - функциональных обработчиков рядов:

- для вычисления характеристик для числового ряда разрабатываются подпрограммы-функции firstTerm, multTerm и sumLimit;
- разработанные функции характеристик объединяются в рамках единой подпрограммы типа funcSeries;
- использование функционального обработчика ряда выполняется следующим образом:
  - z:=funcSeries(x,FIRST_TERM) - для вычисления значения начального члена ряда a_0;
  - z:=funcSeries(x,SUM_LIMIT) - для вычисления предельного значения суммы ряда;
  - z:=funcSeries(x,i) - для вычисления значения рекуррентного множителя q_i (i - номер суммируемого элемента ряда).

3.6. Организация доступа к обработчикам рядов

Проблема организации доступа к обработчикам рядов состоит в том, что в зависимости от числовых рядов, подлежащих обработке, программе суммирования могут потребоваться различные обработчики. Непосредственная реализация схемы переключения операторов вызова обработчиков рядов усложняет программу суммирования и требует ее корректировки после каждого расширения набора обрабатываемых рядов.

Возможное решение данной проблемы может состоять в разработке дополнительной функции, которая обеспечивала бы взаимодействие программы суммирования и обработчиков рядов. Строка-заголовок такой функции может иметь вид:

```
procedure anySeries(ns:integer{8}; x:real; num:integer):real;
```

При наличии такого переходника программа суммирования `sumSeries` может обращаться только к этой единственной функции; для организации доступа программа-переходник `anySeries` должна обеспечивать вызов необходимого обработчика ряда (определение требуемого обработчика может быть выполнено по номеру обрабатываемого ряда `ns`). Возможная схема реализации функции `anySeries` может состоять в следующем:

```
procedure anySeries(ns:integer{8}; x:real; num:integer):real;
  begin
    case ns OF
      0: return sinSeries(x,num);
      | 1: return cosSeries(x,num);
   .   .   .
    end; {case}
end anySeries;
```

Введение переходника позволяет обеспечить полную независимость функции суммирования `sumSeries` от набора числовых рядов, подлежащих реализации. Структура программы-переходника является достаточно простой, однако подлежит корректировке всякий раз при изменении состава или порядка расположения рядов для суммирования. В лабораторной работе предлагается обеспечить неизменность и функции `anySeries` за счет использования *процедурных типов*, поддерживаемых в алгоритмическом языке Zonnon (см. п. 3.7.).

3.7. Использование процедурного типа

Под *процедурным типом* данных в алгоритмическом языке Zonnon понимается тип, используемый для указания количества и типов формальных параметров функций. Общий формат определения процедурного типа имеет вид:

```
type
 T=procedure(<типы параметров>)[:<тип_результата>],
```

В качестве примера определим процедурный тип, описывающий программы-обработчики рядов:

```
type
  TSeries=procedure(real; integer):real;
```

Основные моменты при использовании процедурного типа состоят в следующем:

– допускается создание переменных процедурного типа

```
var
   someSeries : TSeries;
```

– переменным процедурного типа могут быть присвоены имена имеющихся в программе подпрограмм (тип присваиваемой подпрограммы и ее параметры должны соответствовать описанию процедурного типа)

```
someSeries := sinSeries;
```

Выполнение такой операции можно понимать, как запоминание в переменной процедурного типа адреса расположения подпрограммы в оперативной памяти ЭВМ;

– переменная процедурного типа может быть использована для обращения к подпрограмме, адрес которой в ней запомнен:

```
z := someSeries(x,i);
```

Возможность создания в алгоритмическом языке Zonnon процедурных типов данных может быть использована для построения нового варианта программы-переходника `anySeries`. Зафиксируем максимально возможное количество рядов, которое будет реализовываться в программах (константа `MaxSeriesNum`), и определим массив с элементами процедурного типа:

```
CONST
  MaxSeriesNum = 10;
var
  serArr : ARRAY MaxseriesNum of TSeries;
```

Для начального заполнения элементов массива `serArr` можно разработать процедуру инициализации:

```
procedure initSeries;
  begin
    serArr[0] := sinSeries;
    serArr[1] := cosSeries;
  .  .  .
end initSeries;
```

Процедура `initSeries` должна установить адреса программ-обработчиков для всех реализованных рядов; порядок запоминания адресов должен соответствовать номерам рядов.

С учетом выполненной подготовительной работы программа-переходник `anySeries` может иметь вид:

```
procedure anySeries(ns:integer{8}; x:real; num:integer):real;
  begin
    return serArr[ns](x,num);
end anySeries;
```

Подобная реализация программы-переходника `anySeries` не зависит от набора реализуемых при выполнении лабораторной работы числовых рядов; все необходимые изменения при расширении состава рядов сводятся к соответствующей корректировке процедуры инициализации `initSeries`.

4. Рекомендации по организации программного интерфейса

4.1. Структура программы

Разрабатываемые программы при выполнении лабораторной работы целесообразно разместить в трех макромодулях ***module*** (см. рис. 2.5). В первый из них - в основной модуль - рекомендуется поместить программы, организующие диалог с пользователем и управление процессом вычислений. Во втором модуле `Series` можно расположить программы, связанные с обработкой числовых рядов. Третий модуль `serVis` может быть использован для программ, обеспечивающих визуализацию получаемых результатов. Такое разделение программ уменьшит сложность создаваемого программного обеспечения и позволит организовать поэтапное выполнение лабораторной работы.

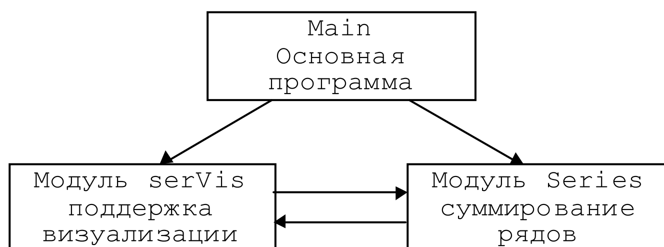

Рис. 2.5. Структура программы

4.2. Управление процессом вычислений

Управление процессом вычислений предлагается реализовать в виде отдельного модуля - *основной программы* (`Main`). Действия, которые должен обеспечивать разрабатываемый в ходе данной лабораторной работы программный комплекс, уже рассматривались в разделе 1. Опишем кратко правила, которых рекомендуется придерживаться при разработке модуля `Main`:

- основная программа должна выполняться в *интерактивном режиме*. Это значит, что ее работа должна зависеть от решений, принимаемых пользователем, и происходить в

форме диалога с ним. Назначение пользовательского интерфейса состоит в том, чтобы сделать работу с программой по возможности проще для человека. Поэтому при разработке основной программы следует обратить внимание на наглядное представление входной и выходной информации, найти простой и удобный способ задания расчетных параметров для проводимых экспериментов;

- диалог с пользователем предлагается реализовать в виде *меню,* обеспечивающего управление вычислительным процессом. С его помощью должна обеспечиваться возможность выбора функции для исследования соответствующего ряда, задания расчетных параметров эксперимента, проведения серии испытаний или единичного эксперимента. В числе команд меню должен быть также пункт для получения краткой справки о возможностях программы. При реализации меню могут потребоваться управление поведением курсора и обработка нажатий клавиш;
- при проведении вычислительных экспериментов основная программа использует подпрограммы работы с рядами из модуля `Series` (см. п. 4.3.) и средства визуализации процесса сходимости из модуля `serVis` (см. п. 4.4.). Эти модули должны подключаться в программу в разделе ***import***. Для правильной обработки ошибочных ситуаций, которые могут возникнуть в ходе проведения вычислительных экспериментов или при задании исходных параметров процесса, рекомендуется пользоваться подпрограммами подсистемы обработки программных ошибок (см. в разделе 5.). Код завершения каждой подпрограммы должен контролироваться.

4.3. Описание модуля работы с рядами

Программы, реализующие суммирование числовых рядов, рекомендуется оформить в виде макромодуля `Series`. Интерфейсная секция модуля может иметь следующий вид:

```
module Series; {суммирование рядов}
(* описание символических констант *)
…
                                           {вычисление частичной суммы ряда}
procedure  {public} sumSeries(ns:integer{8}; x:real; n:Word):real;
                                         {генерация <предельной> суммы ряда}
procedure  {public} sumSeriesLimit(ns:integer{8}; x:real):real;
                                                 {получение названия ряда}
procedure  {public} isSeriesName(ns:integer{8}):String;
                                  {получение кода завершения последней операции}
procedure  {public} getRetCode:ShortInt;
                                               {печать сообщения об ошибке}
procedure  {public} printErrorMsg(err:ShortInt);
```

Поясним кратко назначение процедур и функций модуля.

SumSeries - вычисляет частичную сумму заданного ряда;

sumSeriesLimit - определяет точное значение суммы указанного ряда. Данная функция может использоваться другими программными средствами (не входящими в модуль `Series`) при исследовании скорости сходимости ряда. Исходный программный текст для данной функции будет приведен далее в разделе 5;

isSeriesName - генерирует в строковом виде название заданного ряда;

getRetCode - возвращает код завершения последней выполненной пользователем операции. Данная функция входит в подсистему обработки программных ошибок (см. раздел 5);

printErrorMsg - возвращает в строковом виде сообщение об обнаруженной программной ошибке. Данная функция входит в подсистему обработки программных ошибок (см. раздел 5).

В блоке "описание символических констант" рекомендуется поместить символические константы для номеров рядов из базового набора. Вариант его реализации будет рассмотрен в разделе 5.

Рекомендуемая последовательность реализации подпрограмм (включая программные средства, размещенные в секции реализации модуля `Series`) может быть следующей:

```
{подсистема обработки программных ошибок}
   isRangeValid
   isArgValid
   isNumValid
   setRetCode
   getRetCode
   printErrorMsg
{функциональные обработчики рядов}
   initSeries
   anySeries
   sinSeries
   cosSeries
   ...
{программы суммирования числовых рядов}
   sumSeriesLimit
   sumSeries
{функции общего назначения}
   IsSeriesName
```

Приведенную форму интерфейсной секции рекомендуется сохранять при модификации базового набора рядов (или при внесении других дополнений в модуль), чтобы обеспечить совместимость создаваемых версий программы и сократить количество допускаемых при разработке ошибок.

4.4. Средства визуализации процесса сходимости (структура модуля SerVis)

4.4.1. Интерфейсная часть модуля SerVis

Вывод результатов выполняемых экспериментов по суммированию числовых рядов рекомендуется оформить в виде таблицы данных, а значения получаемой погрешности расчетов целесообразно представить в виде гистограммы (для уменьшения сложности разработки вывод результатов может быть выполнен в консольном режиме). Возможный вид интерфейсной секции модуля `serVis` может быть следующим:

```
module serVis;

                                        {визуализация процесса сходимости}
procedure {public}  visualProcess(ns:integer{8};x:real;Nmin,Nmax:Word);
            {получение кода завершения последней операции}
procedure {public} getVisualError:ShortInt;
```

При выполнении серии экспериментов для получения набора частичных сумм и вычисления погрешности результатов расчетов используются процедуры и функции модуля `Series` (модуль суммирования может быть подключен в секции реализации модуля `serVis`). Программные ошибки, возникающие в процессе работы модуля `serVis`, могут быть отслежены при помощи функции `getVisualError` (см. в разделе 5).

4.4.2. Таблица промежуточных результатов

Процесс приближения частичной суммы ряда к своему предельному значению можно проиллюстрировать путем вычисления частичных сумм $S_{n1}, S_{n2}, \dots, S_{nk}$ для последовательно

увеличивающегося количества суммируемых членов ряда ($n_1<n_2<...<n_k$). Получаемые значения целесообразно разместить в таблице, возможный вид которой может быть следующим:

```
Суммирование числового ряда для функции sin(x)
аргумент x=1.57
```

Кол-во итераций	Частичная сумма
1	1.570000
2	0.925018
3	1.004509
4	0.999843
5	1.000003
6	1.000000
7	1.000000
Точное значение	1.000000

Рис. 2.6. Таблица промежуточных результатов

Рисование рамок таблицы в текстовом режиме работы дисплея может быть выполнено при помощи символов псевдографики.

4.4.3. Гистограмма

Для наглядного и простого качественного анализа получаемых результатов числовые данные расчетов обычно дополняются формами визуального (графического) представления информации. Так, например, в рамках данной выполняемой лабораторной работы для оценки скорости сходимости частичных сумм к точному значению вычисляемой функции могло бы быть полезным представление на экране дисплея *гистограммы значений погрешности* получаемых приближений для значений функции. Расположение данной гистограммы целесообразно совместить с таблицей результатов (см. рис.2.2). При таком способе вывода каждой строке таблицы (т.е. каждому выполненному расчету) будет соответствовать строка гистограммы, а результаты последовательно вычисляемых частичных сумм будут располагаться по вертикали.

Длина строки гистограммы должна определяться с учетом имеющейся погрешности вычислений (например, чем меньше погрешность, тем меньше длина строки). Конкретный способ формирования гистограммы определяется применяемой формой вычисления погрешности (*абсолютной* или *относительной*) и зависит от используемого вида шкалы представления значений (*линейная*, *логарифмическая* и др.). Так, при расчетах абсолютной погрешности по логарифмической шкале длина строки гистограммы может вычисляться в соответствии с выражением:

$$w_i = \left[M \Big/ \left(1 + \Delta_{\max} - \log_{10} \left| S^* - S_i + \delta \right| \right) \right],$$

где скобки [] означают операцию взятия целого значения, M есть размер (число позиций) области вывода гистограммы по горизонтали, S^* содержит точное значение вычисляемой функции, S_i обозначает частичную сумму для рассчитываемой строки гистограммы, а $\Delta_{\max}$ есть максимальное значение погрешности в выполненной серии расчетов:

$$\Delta_{\max} = \max \log_{10} \left| S^* - S_i + \delta \right|.$$

Величина δ в выражении для w_i введена для корректного поведения формулы при значениях погрешностей, близких к нулю; в качестве возможного значения δ может быть выбрано, например, $\delta = 1.0e-80$ (в этом случае при вычислении w_i следует использовать вещественный тип `double`).

5. Подсистема обработки программных ошибок

При разработке программ в ходе выполнения лабораторной работы предлагается обратить особое внимание на возможность возникновения ошибочных ситуаций в процессе проводимых вычислений. Ошибки могут возникать как вследствие некорректной реализации программ (*алгоритмические ошибки*), так и в результате неправильного задания исходных данных (*ошибки ввода*). Отладка (*поиск и исправление ошибок*) является одним из наиболее трудоемких этапов при разработке программного обеспечения. Для снижения затрат на отладку при разработке программ обычно осуществляется контроль правильности возникающих ситуаций (например, путем проверки допустимости значений переменных) в процессе решения задач (*программирование с защитой от ошибок*). Результаты контроля могут фиксироваться в некоторой служебной переменной с использованием ранее определенного набора констант (*кодов завершения*), и тогда проверка значения этой переменной может обеспечить учет и обработку всех обнаруженных ошибок в ходе выполнения программ. Данный способ следует рассматривать как предварительный подход, дающий начальное знакомство с проблемами контроля и обработки ошибочных ситуаций при выполнении программ; профессиональное решение, требующего знания объектно-ориентировного программирования, состоит в использовании *исключений* (с., например, [2]).

Для разрабатываемого программного комплекса по суммированию числовых рядов для контроля значений задаваемых пользователем данных (номер ряда, аргумент функции и количество итераций) рекомендуется подготовить программы контроля:

```
                                             {контроль заданного числа шагов}
procedure  {public} isRangeValid(n:Word):boolean;
                                             {контроль заданного аргумента функции}
procedure  {public} isArgValid(x:real): boolean;
                                             {контроль дескриптора ряда}
procedure  {public} isNumValid(ns:integer{8}): boolean;
```

Для контроля и обнаружения ошибок при выполнении лабораторной работы следует соблюдать следующие правила разработки:

- программы должны включать контролирующие действия; вид и объем контроля определяется в зависимости от выполняемых в программах вычислений;
- при завершении работы программы должны устанавливать коды (признаки) успешности своего завершения; ошибочные ситуации могут фиксироваться при помощи различающихся кодов.

В качестве кодов возможных ошибок предлагается использовать отрицательные числа (ситуация отсутствия ошибок соответствует нулевому коду). Целесообразным представляется помещение кодов программных ошибок в отдельный файл. Исходный файл с именем `series.inc` является примером такого описания, в нем также можно расположить необходимые для пользователя (см. п. 4.3.) символические константы для номеров рядов из базового набора. Блок "описание символических констант" может иметь следующий вид:

```
        (* описание символических констант *)
const {коды ошибок}
 OK          =  0;{ошибок нет                     }
 BadTermNum  = -1;{недопустимое кол-во членов ряда  }
 BadArgument = -2;{аргумент- вне области определения}
 BadSeriesNum= -3;{недопустимый номер ряда          }
 BadVisual   = -4;{ошибка визуализации              }

                   {описатели рядов}
  RSIN        = 1; {          синус}
  RCOS        = 2; {        косинус}
  REXP        = 3; {     экспонента}
```

Для запоминания кода завершения может быть разработана программа секции реализации:

```
procedure setRetCode(err:ShortInt);
```

Запоминание кода завершения целесообразно проводить только в случае, если значение служебной переменной является нулевым (признак отсутствия ошибки); такой способ запоминания позволяет зафиксировать код первой обнаруженной ошибки. Ниже приводится пример возможной реализации описанной процедуры.

```
procedure  {public} setRetCode(err:ShortInt);
  begin
    if RetCode=Ok then RetCode := err; end
end setRetCode;
```

Предполагается, что переменная с именем `RetCode` служит для хранения кода завершения, а специальная символьная константа `OK` (равная нулю) соответствует ситуации отсутствия ошибок.

Для получения кода завершения необходима функция интерфейсной секции; возможный вид строки заголовка функции может быть следующим:

```
procedure  {public} getRetCode:ShortInt;
```

При реализации функции `getRetCode` после передачи кода завершения целесообразно установить нулевое значение.

В виде дополнительной программы может быть рекомендована разработка процедуры печати сообщения с описанием ошибки с указанным кодом:

```
procedure  {public} printErrorMsg(err:ShortInt);
```

Завершим описание программ, обрабатывающих и отслеживающих ошибочные ситуации, примером реализации функции `sumSeriesLimit` (из модуля `Series`), использующей соответствующие средства:

```
    {генерация <предельной> суммы ряда}
procedure sumSeriesLimit(ns:integer{8}; x:real):real;
  begin
    if ~ isNumValid(ns)
      then setRetCode(BadSeriesNum)
    elsif ~ isArgValid(ns,x)
      then setRetCode(BadArgument)
    else
       setRetCode(Ok);
       return anySeries(ns,x,SUM_LIMIT);
    end;
end sumSeriesLimit;
```

6. Рекомендуемые темы дополнительных заданий

1). Разработка расширенного набора числовых рядов, (например, для функций `ln`, `sh`, `ch` и др.).

2). Реализация *условия остановки по точности* в алгоритме суммирования, при котором вычисления завершаются при выполнении неравенства:

$|a_i| \le \varepsilon$, для задаваемой величины $\varepsilon > 0$.

7. Литература

1. Демидович Б.П., Марон И.А. Основы вычислительной математики. – М.: Наука, 1966.

2. Рихтер Д. Программирование на платформе Microsoft .NET Framework. – М.: Издательско-торговый дом "Русская Редакция", 2002.