**МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ**

**БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ**

**ФАКУЛЬТЕТ ПРИКЛАДНОЙ МАТЕМАТИКИ И ИНФОРМАТИКИ**

**Кафедра информационных система управления**

Лабораторная работа № 1

**Приближение функции**

Вариант 1

**Выполнил**

Веренич Владислав Николаевич

3 курс 12 группа

**Преподаватель**

Будник А.М.

# **Задание 1.**

**Постановка задачи**

Даны значения функции в узлах интерполирования , , порядковый номер в списке группы, , , ,

.

1. Построить многочлен Лагранжа(1-й вариант)/Ньютона(2-й вариант) по значениям функции в заданных узлах интерполирования.
2. Вычислить значение в точках восстановления , , , .
3. Оценить погрешность в точках восстановления, используя многочлен Ньютона(1-й вариант)/Лагранжа(2-й вариант).

В моём случае(вариант 1, 4-й в списке):

,

,

,

Узлы интерполирования:

|  | 0.3 | 0.4 | 0.5 | 0.6 | 0.7 | 0.8 | 0.9 | 1.0 | 1.1 | 1.2 | 1.3 |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0.890981 | 0.845839 | 0.796267 | 0.742378 | 0.684365 | 0.622493 | 0.557098 | 0.488575 | 0.417379 | 0.344009 | 0.269009 |

**Теория**

*Интерполяционный многочлен*

Интерполяционный многочлен в форме Лагранжа имеет следующий вид:

,

где .

Очевидно, с его помощью удобно интерполировать различные функции по одной и той же таблице узлов, так как коэффициенты влияния( называют функцией влияния i-го узла) в этом случае могут быть вычислены заранее и не требуют пересчета.

*Разделенные разности*

Разделенные разности первого порядка определяются равенствами

разности второго порядка – равенствами

и вообще, разности (𝑘 + 1)-го порядка определяются через разности 𝑘-го порядка по формуле

*Остаток интерполирования*

Под остатком интерполирования понимают разность:

Представление остатка интерполирования в форме Ньютона:

.

Для оценки погрешности я использовал абсолютные значения остатков интерполирования.

**Листинг**

Вычисление функции

| double f (double x)  {  return 0.3 \* std::exp(-x) + 0.7 \* std::cos(x);  } |
| --- |

Формирование таблицы разделённых разностей:

| void createMatrix (int n, double\*\* rr, double\* x, double\* fx)  {  for (int i = 0; i <= n; ++i) {  rr[i][0] = fx[i];  }  for (int j = 1; j <= n; ++j) {  for (int i = 0; i <= n - j; ++i) {  rr[i][j] = (rr[i+1][j-1] - rr[i][j-1]) / (x[i+j] - x[i]);  }  }  } |
| --- |

Вычисление многочлена Лагранжа по узлам интерполирования:

| double lagrangePolynomial (double x, int n, double\* xn, double\* fxn)  {  double result = 0;  double numerator = 1;  double denominator;  for (int i = 0; i <= n; ++i) {  if (x == xn[i]) {  return fxn[i];  }  numerator \*= (x - xn[i]);  }  for (int i = 0; i <= n; ++i) {  denominator = 1;  for (int j = 0; j < i; ++j) {  denominator \*= (xn[i] - xn[j]);  }  for (int j = i + 1; j <= n; ++j) {  denominator \*= (xn[i] - xn[j]);  }  result += fxn[i] \* numerator / (x - xn[i]) / denominator;  }  return result;  } |
| --- |

Вычисление истинной погрешности:

| double getRealResidual (double x, double px)  {  return std::abs(f(x) - px);  } |
| --- |

Вычисление теоретической погрешности:

| double getTheoreticalResidualNewton (double x, int n, double\* xn, double\*\* rrn)  {  rrn[n+1][0] = f(x);  for (int i = 1; i <= n+1; ++i) {  rrn[n-i+1][i] = (rrn[n-i+2][i-1] - rrn[n-i+1][i-1]) / (x - xn[n-i+1]);  }  double w = x - xn[0];  for (int i = 1; i <= n; ++i) {  w \*= (x - xn[i]);  }  return std::abs(w \* rrn[0][n+1]);  } |
| --- |

**Результаты**

Таблица разделённых разностей.

| Divided difference table:  0.890981 -0.451423 -0.221471 0.0187613 0.0331478 -0.00448805 -0.000571162 5.26439e-005 1.69415e-005 -1.70197e-006 -9.64912e-008  0.845839 -0.495717 -0.215843 0.0320204 0.0309038 -0.00483074 -0.000534311 6.61971e-005 1.54097e-005 -1.79846e-006  0.796267 -0.538886 -0.206236 0.0443819 0.0284884 -0.00515133 -0.000487973 7.85249e-005 1.37911e-005  0.742378 -0.580133 -0.192922 0.0557773 0.0259128 -0.00544411 -0.000433006 8.95578e-005  0.684365 -0.618717 -0.176189 0.0661424 0.0231907 -0.00570392 -0.000370315  0.622493 -0.653955 -0.156346 0.0754187 0.0203387 -0.00592611  0.557098 -0.685224 -0.13372 0.0835542 0.0173757  0.488575 -0.711968 -0.108654 0.0905044  0.417379 -0.733699 -0.0815028  0.344009 -0.75  0.269009 |
| --- |

Можем видеть, что теоретическая погрешность практически не отличается от истинной. Это связано с тем, что

Так как в последнем соотношении первое слагаемое есть не что иное, как интерполяционный многочлен в форме Лагранжа, мы и получаем практически одинаковые значения погрешности.

# **Задание 2.**

**Постановка задачи**

Интерполирование при равноотстоящих узлах.

1. Используя полином 3-ей степени проинтерполировать функцию в начале таблицы.
2. Вычислить значение в точках восстановления , , , .
3. Провести анализ результатов.

Начало таблицы:

|  | 0.3 | 0.4 | 0.5 | 0.6 |
| --- | --- | --- | --- | --- |
|  | 0.890981 | 0.845839 | 0.796267 | 0.742378 |

**Теория**

*Конечные разности*

Итак, пусть 𝑓 (𝑥) задана таблично в точках, отстоящих друг от друга на равных расстояниях:

Конечными разностями первого порядка назовем числа

По конечным разностям первого порядка рекурсивно строятся конечные разности второго порядка, по разностям второго порядка – разности третьего порядка и т.д.:

…

*Интерполирование в начале таблицы*

Считая, что для достижения целей достаточно интерполирования с помощью многочлена 𝑘-й степени, привлекаем узлы . Запишем интерполяционный многочлен в форме Ньютона:

Произведем замену переменной по формуле . Так как по предположению , то . Тогда, учитывая, что после несложных преобразований получим:

.

Это интерполяционный многочлен Ньютона для начала таблицы.

**Листинг**

Формирование таблицы конечных разностей:

| void createMatrix (int n, double\*\* kr, double\* fx)  {  for (int i = 0; i <= n; ++i) {  kr[i][0] = fx[i];  }  for (int j = 1; j <= n; ++j) {  for (int i = 0; i <= n - j; ++i) {  kr[i][j] = (kr[i+1][j-1] - kr[i][j-1]);  }  }  } |
| --- |

Вычисление многочлена Ньютона для начала таблицы:

| double beginning (double x, int n, double\*\* kr, double xf, double h)  {  double t = (x - xf) / h;  double res = kr[0][0];  double mul = t;  for (int i = 1; i <= n; ++i) {  res += (kr[0][i] \* mul);  mul \*= ((t - i) / (i + 1));  }  return res;  } |
| --- |

Вычисление истинной погрешности.

| double findRealResidual (double x, double px)  {  return f(x) - px;  } |
| --- |

**Результаты**

Таблица конечных разностей:

| KR matrix:  0.890981 -0.0451423 -0.00442942 0.000112568  0.845839 -0.0495717 -0.00431685  0.796267 -0.0538886  0.742378 |
| --- |

= 0.301 0.890552

= 0.333

Для сравнения теоретического значения и точного были взяты точки и . Заметим, что располагается ближе одной из точек интерполирования, поэтому и остаток интерполирования для этой точки меньше чем для точки .

# **Задание 3.**

**Постановка задачи**

Дана функция и значения параметра .

1. Вычислить значения узлов, такие, чтобы при интерполировании по отрезку многочленом степени погрешность была минимальной. По значениям функции в этих узлах построить интерполяционный многочлен (1-й вариант Лагранжа, 2-й вариант Ньютона).
2. Вычислить значение в точках , , , .
3. Провести анализ результатов(сравнить погрешности при интерполировании на чебышевских и равноправных узлах).

**Теория**

*Узлы Чебышёва*

Если интерполирование производится на произвольном отрезке [𝑎, 𝑏], то узлами интерполирования будут:

*Интерполяционный многочлен*

Интерполяционный многочлен в форме Лагранжа имеет следующий вид:

,

где .

*Оценка погрешности*

Соответственно, для оценки погрешности используем неравенство:

**Листинг**

Вычисление узлов интерполирования и значений функции в этих узлах.

| void createNodes (int n, double\* x, double\* fx, double a, double b)  {  double add = (a + b) / 2, mul = (b - a) / 2;  for (int i = 0; i <= n; ++i)  {  x[i] = add + mul \* std::cos (M\_PI \* (2 \* (n - i) + 1) / 2 / (n + 1));  fx[i] = f (x[i]);  }  } |
| --- |

Вычисление многочлена Лагранжа по узлам интерполирования:

| double P (double x, int n, double\* xn, double\* fxn)  {  double res = 0;  double num = 1, denom;  for (int i = 0; i <= n; ++i)  {  if (x == xn[i])  return fxn[i];  num \*= (x - xn[i]);  }  for (int i = 0; i <= n; ++i)  {  denom = 1;  for (int j = 0; j < i; ++j)  denom \*= (xn[i] - xn[j]);  for (int j = i + 1; j <= n; ++j)  denom \*= (xn[i] - xn[j]);  res += fxn[i] \* num / (x - xn[i]) / denom;  }  return res;  } |
| --- |

Вычисление теоретической погрешности.

| double findTheoreticalResidual (int n, double a, double b)  {  double mul = (b - a) / 2;  for (int i = 1; i <= n; ++i)  mul \*= ((b - a) / (i + 1) / 4);  switch (n % 4)  {  case 0:  return (0.3 \* std::exp(-1.3) - 0.7 \* std::sin(1.3)) \* mul;  case 1:  return (0.3 \* std::exp(-1.3) - 0.7 \* std::cos(1.3)) \* mul;  case 2:  return (0.3 \* std::exp(-1.3) + 0.7 \* std::sin(1.3)) \* mul;  case 3:  return (0.3 \* std::exp(-1.3) + 0.7 \* std::cos(1.3)) \* mul;  }  } |
| --- |

Вычисление истинной погрешности.

| double findRealResidual (double x, double px)  {  return f(x) - px;  } |
| --- |

**Результаты**

Узлы чебышева

|  | 0.305089 | 0.345184 | 0.422125 | 0.52968 | 0.659134 | 0.8 | 0.940866 | 1.07032 | 1.17787 | 1.25482 | 1.29491 |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0.888791 | 0.871136 | 0.83525 | 0.780716 | 0.708556 | 0.622493 | 0.529449 | 0.43876 | 0.360402 | 0.303062 | 0.272856 |

, ,

Теоретическая погрешность интерполирования при чебышевских узлах:

Истинная погрешность интерполирования при равноправных узлах(только в точке ):

Истинная погрешность интерполирования при использовании многочлена Лагранжа:

Как видно, использование чебышевских узлов в качестве узлов интерполирования привело к тому, что истинная погрешность интерполирования для точки уменьшилась по сравнению с истинной погрешностью интерполирования, где в качестве узлов интерполирования использовались равноотстоящие узлы, и стала практически такой же, как и при интерполировании с использованием многочлена Лагранжа, то есть, как это было доказано выше, стала сравнима с истинной погрешностью. Однако, для точки погрешность стала в 8.85714 раз больше, а для точки в 12.786 меньше по сравнению с интерполированием при использовании многочлена Лагранжа.

# **Задание 4.**

**Постановка задачи**

Метод наименьших квадратов.

Построить аппроксимирующий многочлен степени и вычислить приближение функции в точках

|  | 0.3 | 0.5 | 0.7 | 0.9 | 1.1 | 1.3 |
| --- | --- | --- | --- | --- | --- | --- |
|  | 0.890981 | 0.796267 | 0.684365 | 0.557098 | 0.417379 | 0.269009 |

**Теория**

Пусть функция задана таблицей своих значений: .

Требуется найти многочлен фиксированной степени , ля которого среднеквадратичное отклонение (СКО) минимально.

Так как многочлен определяется своими коэффициентами, то фактически нужно подобрать набор коэффициентов , минимизирующий функцию

Используя необходимое условие экстремума, получаем так называемую нормальную систему метода наименьших квадратов:

Полученная система есть система алгебраических уравнений относительно неизвестных .

**Листинг**

Вычисление функции

| double f (double x)  {  return 0.3 \* exp(-x) + 0.7 \* cos(x);  } |
| --- |

Создание матрицы линейных коэффициентов и вектора правых частей для нахождения методом Гаусса.

| void createMatrix (int n, int N, double\* x, double\*\* A, double\* b, double\* fx)  {  for (int i = 0; i <= n; ++i)  {  for (int j = 0; j <= n; ++j)  {  A[i][j] = 0;  for (int p = 0; p < N; ++p)  A[i][j] += pow (x[p], i + j);  }  b[i] = 0;  for (int j = 0; j < N; ++j)  b[i] += fx[j] \* pow (x[j], i);  }  } |
| --- |

Вычисление многочлена

| double Polynom (double pt, int n, double\* coef)  {  double res = 0;  for (int i = 0; i <= n; ++i)  res += coef[i] \* pow (pt, i);  return res;  } |
| --- |

Вычисление истинной погрешности.

| double findRealResidual (double x, double px)  {  return f(x) - px;  } |
| --- |

**Результаты**

Коэффициенты

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 1.00007 | -0.300692 | -0.197379 | -0.0550401 | 0.0468974 | -0.00528256 |

, ,

,

,

,

Точность результатов, полученных приближением функции многочленом 3 степени по 6 узлам при помощи метода наименьших квадратов, ниже, чем точность результатов, полученных при интерполировании другими, использованными ранее способами.