

# HarrixMathLibrary v.3.33

А. Б. Сергиенко

14 декабря 2013 г.

## Аннотация

Библиотека HarrixMathLibrary — это сборник различных математических функций и функций-шаблонов с открытым кодом на языке C++.

## Содержание

<b>1 Описание</b>	<b>11</b>
<b>2 Установка</b>	<b>12</b>
2.1 Общий алгоритм подключения . . . . .	12
2.2 Подключение к Qt на примере Qt 5.1.1 . . . . .	12
2.3 Подключение к C++ Builder на примере C++ Builder 6.0 . . . . .	13
2.4 Подключение к C++ Builder на примере C++Builder XE4 . . . . .	14
2.5 Подключение к Microsoft Visual Studio на примере Visual Studio 2012 . . . . .	15
<b>3 О случайных числах в библиотеке HarrixMathLibrary</b>	<b>17</b>
<b>4 Как добавлять новые функции в библиотеку</b>	<b>21</b>
<b>5 Сторонние библиотеки, используемые в HarrixMathLibrary</b>	<b>35</b>
<b>6 Список функций</b>	<b>36</b>
<b>7 Функции</b>	<b>52</b>
7.1 Вектора (Одномерные массивы) . . . . .	52
7.1.1 MHL_DependentNoiseInVector . . . . .	52
7.1.2 MHL_EuclidNorma . . . . .	53
7.1.3 MHL_NoiseInVector . . . . .	54

7.1.4	TMHL_AcceptanceLimits . . . . .	55
7.1.5	TMHL_CheckElementInVector . . . . .	57
7.1.6	TMHL_CompareMeanOfVectors . . . . .	58
7.1.7	TMHL_EqualityOfVectors . . . . .	59
7.1.8	TMHL_FibonacciNumbersVector . . . . .	60
7.1.9	TMHL_FillVector . . . . .	61
7.1.10	TMHL_MaximumOfVector . . . . .	62
7.1.11	TMHL_MinimumOfVector . . . . .	63
7.1.12	TMHL_MixingVector . . . . .	64
7.1.13	TMHL_MixingVectorWithConjugateVector . . . . .	65
7.1.14	TMHL_NumberOfDifferentValuesInVector . . . . .	66
7.1.15	TMHL_NumberOfMaximumOfVector . . . . .	67
7.1.16	TMHL_NumberOfMinimumOfVector . . . . .	68
7.1.17	TMHL_NumberOfNegativeValues . . . . .	68
7.1.18	TMHL_NumberOfPositiveValues . . . . .	69
7.1.19	TMHL_NumberOfZeroValues . . . . .	70
7.1.20	TMHL_OrdinalVector . . . . .	71
7.1.21	TMHL_OrdinalVectorZero . . . . .	72
7.1.22	TMHL_ProductOfElementsOfVector . . . . .	73
7.1.23	TMHL_ReverseVector . . . . .	73
7.1.24	TMHL_SearchElementInVector . . . . .	74
7.1.25	TMHL_SearchFirstNotZero . . . . .	75
7.1.26	TMHL_SearchFirstZero . . . . .	76
7.1.27	TMHL_SumSquareVector . . . . .	77
7.1.28	TMHL_SumVector . . . . .	78
7.1.29	TMHL_VectorMinusVector . . . . .	79
7.1.30	TMHL_VectorMultiplyNumber . . . . .	81
7.1.31	TMHL_VectorPlusVector . . . . .	82
7.1.32	TMHL_VectorToVector . . . . .	84
7.1.33	TMHL_ZeroVector . . . . .	85
7.2	Генетические алгоритмы . . . . .	86
7.2.1	MHL_BinaryFitnessFunction . . . . .	86

7.2.2	<i>MHL_MakeVectorOfProbabilityForProportionalSelectionV2</i>	87
7.2.3	<i>MHL_MakeVectorOfProbabilityForRankSelection</i>	88
7.2.4	<i>MHL_MakeVectorOfRankForRankSelection</i>	90
7.2.5	<i>MHL_MakeVectorOfRankZeroForRankSelection</i>	92
7.2.6	<i>MHL_NormalizationVectorAll</i>	93
7.2.7	<i>MHL_NormalizationVectorMaxMin</i>	94
7.2.8	<i>MHL_NormalizationVectorOne</i>	95
7.2.9	<i>MHL_ProbabilityOfTournamentSelection</i>	96
7.2.10	<i>MHL_ProportionalSelection</i>	98
7.2.11	<i>MHL_ProportionalSelectionV2</i>	100
7.2.12	<i>MHL_ProportionalSelectionV3</i>	102
7.2.13	<i>MHL_RankSelection</i>	104
7.2.14	<i>MHL_SelectItemOnProbability</i>	106
7.2.15	<i>MHL_StandartBinaryGeneticAlgorithm</i>	107
7.2.16	<i>MHL_StandartGeneticAlgorithm</i>	110
7.2.17	<i>MHL_StandartRealGeneticAlgorithm</i>	117
7.2.18	<i>MHL_TournamentSelection</i>	121
7.2.19	<i>MHL_TournamentSelectionWithReturn</i>	123
7.2.20	<i>TMHL_MutationBinaryMatrix</i>	124
7.2.21	<i>TMHL_SinglepointCrossover</i>	126
7.2.22	<i>TMHL_SinglepointCrossoverWithCopying</i>	127
7.2.23	<i>TMHL_TwoPointCrossover</i>	129
7.2.24	<i>TMHL_TwoPointCrossoverWithCopying</i>	131
7.2.25	<i>TMHL_UniformCrossover</i>	133
7.3	Геометрия	135
7.3.1	<i>MHL_LineGeneralForm</i>	135
7.3.2	<i>MHL_LineSlopeInterceptForm</i>	136
7.3.3	<i>MHL_LineTwoPoint</i>	136
7.3.4	<i>MHL_Parabola</i>	137
7.3.5	<i>TMHL_BoolCrossingTwoSegment</i>	138
7.4	Гиперболические функции	139
7.4.1	<i>MHL_Cosech</i>	139

7.4.2	<i>MHL_Cosh</i>	140
7.4.3	<i>MHL_Cotanh</i>	140
7.4.4	<i>MHL_Sech</i>	141
7.4.5	<i>MHL_Sinh</i>	141
7.4.6	<i>MHL_Tanh</i>	142
7.5	Дифференцирование	142
7.5.1	<i>MHL_CenterDerivative</i>	142
7.5.2	<i>MHL_LeftDerivative</i>	143
7.5.3	<i>MHL_RightDerivative</i>	144
7.6	Для тестовых функций	145
7.6.1	<i>MHL_ClassOfTestFunction</i>	145
7.6.2	<i>MHL_CountOfFitnessOfTestFunction_Binary</i>	146
7.6.3	<i>MHL_CountOfFitnessOfTestFunction_Real</i>	147
7.6.4	<i>MHL_DefineTestFunction</i>	147
7.6.5	<i>MHL_DimensionTestFunction_Binary</i>	148
7.6.6	<i>MHL_DimensionTestFunction_Real</i>	149
7.6.7	<i>MHL_ErrorExOfTestFunction_Binary</i>	150
7.6.8	<i>MHL_ErrorExOfTestFunction_Real</i>	151
7.6.9	<i>MHL_ErrorEyOfTestFunction_Binary</i>	152
7.6.10	<i>MHL_ErrorEyOfTestFunction_Real</i>	153
7.6.11	<i>MHL_ErrorROfTestFunction_Binary</i>	154
7.6.12	<i>MHL_ErrorROfTestFunction_Real</i>	155
7.6.13	<i>MHL_FitnessOfOptimumOfTestFunction_Binary</i>	156
7.6.14	<i>MHL_FitnessOfOptimumOfTestFunction_Real</i>	157
7.6.15	<i>MHL_GetCountOfFitness</i>	158
7.6.16	<i>MHL_GetCountOfSubProblems_Binary</i>	159
7.6.17	<i>MHL_GetCountOfSubProblems_Real</i>	160
7.6.18	<i>MHL_LeftBorderOfTestFunction_Real</i>	160
7.6.19	<i>MHL_MaximumOrMinimumOfTestFunction_Binary</i>	161
7.6.20	<i>MHL_MaximumOrMinimumOfTestFunction_Real</i>	162
7.6.21	<i>MHL_NumberOfPartsOfTestFunction_Real</i>	163
7.6.22	<i>MHL_OptimumOfTestFunction_Binary</i>	164

7.6.23	<i>MHL_OptimumOfTestFunction_Real</i>	165
7.6.24	<i>MHL_PrecisionOfCalculationsOfTestFunction_Real</i>	166
7.6.25	<i>MHL_SetToZeroCountOfFitness</i>	167
7.6.26	<i>MHL_TestFunction_Binary</i>	168
7.6.27	<i>MHL_TestFunction_Real</i>	169
7.7	Интегрирование	170
7.7.1	<i>MHL_IntegralOfRectangle</i>	170
7.7.2	<i>MHL_IntegralOfSimpson</i>	171
7.7.3	<i>MHL_IntegralOfTrapezium</i>	172
7.8	Кодирование и декодирование	173
7.8.1	<i>MHL_BinaryGrayVectorToRealVector</i>	173
7.8.2	<i>MHL_BinaryVectorToRealVector</i>	175
7.8.3	<i>TMHL_BinaryToDecimal</i>	177
7.8.4	<i>TMHL_BinaryToDecimalFromPart</i>	177
7.8.5	<i>TMHL_GrayCodeToBinary</i>	179
7.8.6	<i>TMHL_GrayCodeToBinaryFromPart</i>	180
7.9	Комбинаторика	181
7.9.1	<i>TMHL_KCombinations</i>	181
7.10	Математические функции	182
7.10.1	<i>MHL_ArithmeticalProgression</i>	182
7.10.2	<i>MHL_ExpMSxD2</i>	183
7.10.3	<i>MHL_GeometricSeries</i>	184
7.10.4	<i>MHL_GreatestCommonDivisorEuclid</i>	185
7.10.5	<i>MHL_HowManyPowersOfTwo</i>	185
7.10.6	<i>MHL_InverseNormalizationNumberAll</i>	186
7.10.7	<i>MHL_LeastCommonMultipleEuclid</i>	186
7.10.8	<i>MHL_MixedMultiLogicVectorOfFullSearch</i>	187
7.10.9	<i>MHL_NormalizationNumberAll</i>	188
7.10.10	<i>MHL_Parity</i>	189
7.10.11	<i>MHL_ProbabilityDensityFunctionOfInverseGaussianDistribution</i>	190
7.10.12	<i>MHL_SumGeometricSeries</i>	191
7.10.13	<i>MHL_SumOfArithmeticalProgression</i>	192

7.10.14 MHL_SumOfDigits . . . . .	192
7.10.15 TMHL_Abs . . . . .	193
7.10.16 TMHL_Factorial . . . . .	193
7.10.17 TMHL_FibonacciNumber . . . . .	194
7.10.18 TMHL_HeavisideFunction . . . . .	194
7.10.19 TMHL_Max . . . . .	195
7.10.20 TMHL_Min . . . . .	196
7.10.21 TMHL_NumberInterchange . . . . .	196
7.10.22 TMHL_PowerOf . . . . .	197
7.10.23 TMHL_Sign . . . . .	198
7.10.24 TMHL_SignNull . . . . .	198
<b>7.11 Матрицы . . . . .</b>	<b>199</b>
7.11.1 TMHL_CheckForIdenticalColsInMatrix . . . . .	199
7.11.2 TMHL_CheckForIdenticalRowsInMatrix . . . . .	200
7.11.3 TMHL_ColInterchange . . . . .	201
7.11.4 TMHL_ColToMatrix . . . . .	202
7.11.5 TMHL_DeleteColInMatrix . . . . .	203
7.11.6 TMHL_DeleteRowInMatrix . . . . .	205
7.11.7 TMHL_EqualityOfMatrixes . . . . .	206
7.11.8 TMHL_FillMatrix . . . . .	207
7.11.9 TMHL_IdentityMatrix . . . . .	208
7.11.10 TMHL_MatrixMinusMatrix . . . . .	209
7.11.11 TMHL_MatrixMultiplyMatrix . . . . .	211
7.11.12 TMHL_MatrixMultiplyMatrixT . . . . .	212
7.11.13 TMHL_MatrixMultiplyNumber . . . . .	214
7.11.14 TMHL_MatrixPlusMatrix . . . . .	215
7.11.15 TMHL_MatrixT . . . . .	217
7.11.16 TMHL_MatrixTMultiplyMatrix . . . . .	219
7.11.17 TMHL_MatrixToCol . . . . .	220
7.11.18 TMHL_MatrixToMatrix . . . . .	221
7.11.19 TMHL_MatrixToRow . . . . .	223
7.11.20 TMHL_MaximumOfMatrix . . . . .	224

7.11.21	<code>TMHL_MinimumOfMatrix</code>	225
7.11.22	<code>TMHL_MixingRowsInOrder</code>	226
7.11.23	<code>TMHL_NumberOfDifferentValuesInMatrix</code>	227
7.11.24	<code>TMHL_RowInterchange</code>	228
7.11.25	<code>TMHL_RowToMatrix</code>	229
7.11.26	<code>TMHL_SumMatrix</code>	231
7.11.27	<code>TMHL_ZeroMatrix</code>	232
7.12	<code>Метрика</code>	233
7.12.1	<code>TMHL_Chebychev</code>	233
7.12.2	<code>TMHL_CityBlock</code>	234
7.12.3	<code>TMHL_Euclid</code>	235
7.12.4	<code>TMHL_Minkovski</code>	236
7.13	<code>Непараметрика</code>	237
7.13.1	<code>MHL_BellShapedKernelExp</code>	237
7.13.2	<code>MHL_BellShapedKernelParabola</code>	238
7.13.3	<code>MHL_BellShapedKernelRectangle</code>	239
7.13.4	<code>MHL_BellShapedKernelTriangle</code>	240
7.13.5	<code>MHL_DerivativeOfBellShapedKernelExp</code>	241
7.13.6	<code>MHL_DerivativeOfBellShapedKernelParabola</code>	243
7.13.7	<code>MHL_DerivativeOfBellShapedKernelRectangle</code>	244
7.13.8	<code>MHL_DerivativeOfBellShapedKernelTriangle</code>	244
7.14	<code>Нечеткие системы</code>	245
7.14.1	<code>MHL_TrapeziformFuzzyNumber</code>	245
7.15	<code>Оптимизация</code>	247
7.15.1	<code>MHL_BinaryMonteCarloAlgorithm</code>	247
7.15.2	<code>MHL_DichotomyOptimization</code>	249
7.15.3	<code>MHL_FibonacciOptimization</code>	250
7.15.4	<code>MHL_GoldenSectionOptimization</code>	250
7.15.5	<code>MHL_QuadraticFitOptimization</code>	251
7.15.6	<code>MHL_RealMonteCarloAlgorithm</code>	252
7.15.7	<code>MHL_RealMonteCarloOptimization</code>	254
7.15.8	<code>MHL_UniformSearchOptimization</code>	255

7.15.9 <i>MHL_UniformSearchOptimizationN</i>	256
7.16 Оптимизация - свалка алгоритмов	257
7.16.1 <i>MHL_BinaryGeneticAlgorithmWCC</i>	257
7.16.2 <i>MHL_BinaryGeneticAlgorithmWDPOfNOfGPS</i>	260
7.16.3 <i>MHL_BinaryGeneticAlgorithmWDTS</i>	263
7.16.4 <i>MHL_RealGeneticAlgorithmWCC</i>	266
7.16.5 <i>MHL_RealGeneticAlgorithmWDPOfNOfGPS</i>	269
7.16.6 <i>MHL_RealGeneticAlgorithmWDTS</i>	272
7.17 Перевод единиц измерений	275
7.17.1 <i>MHL_DegToRad</i>	275
7.17.2 <i>MHL_RadToDeg</i>	276
7.18 Случайные объекты	276
7.18.1 <i>MHL_BitNumber</i>	276
7.18.2 <i>MHL_RandomRealMatrix</i>	277
7.18.3 <i>MHL_RandomRealMatrixInCols</i>	278
7.18.4 <i>MHL_RandomRealMatrixInElements</i>	279
7.18.5 <i>MHL_RandomRealMatrixInRows</i>	280
7.18.6 <i>MHL_RandomRealVector</i>	282
7.18.7 <i>MHL_RandomRealVectorInElements</i>	282
7.18.8 <i>MHL_RandomVectorOfProbability</i>	284
7.18.9 <i>TMHL_BernulliVector</i>	284
7.18.10 <i>TMHL_RandomArrangingObjectsIntoBaskets</i>	285
7.18.11 <i>TMHL_RandomBinaryMatrix</i>	286
7.18.12 <i>TMHL_RandomBinaryVector</i>	287
7.18.13 <i>TMHL_RandomIntMatrix</i>	287
7.18.14 <i>TMHL_RandomIntMatrixInCols</i>	288
7.18.15 <i>TMHL_RandomIntMatrixInElements</i>	289
7.18.16 <i>TMHL_RandomIntMatrixInRows</i>	291
7.18.17 <i>TMHL_RandomIntVector</i>	292
7.18.18 <i>TMHL_RandomIntVectorInElements</i>	293
7.18.19 <i>TMHL_RandomMatrixOfPermutation</i>	294
7.18.20 <i>TMHL_RandomVectorOfPermutation</i>	295

7.19 Случайные числа . . . . .	296
7.19.1 MHL_RandomNormal . . . . .	296
7.19.2 MHL_RandomUniform . . . . .	296
7.19.3 MHL_RandomUniformInt . . . . .	297
7.19.4 MHL_RandomUniformIntIncluding . . . . .	298
7.20 Сортировка . . . . .	299
7.20.1 TMHL_BubbleDescendingSort . . . . .	299
7.20.2 TMHL_BubbleSort . . . . .	300
7.20.3 TMHL_BubbleSortColWithOtherConjugateColsInMatrix . . . . .	301
7.20.4 TMHL_BubbleSortEveryColInMatrix . . . . .	302
7.20.5 TMHL_BubbleSortEveryRowInMatrix . . . . .	303
7.20.6 TMHL_BubbleSortInGroups . . . . .	304
7.20.7 TMHL_BubbleSortRowWithOtherConjugateRowsInMatrix . . . . .	305
7.20.8 TMHL_BubbleSortWithConjugateVector . . . . .	306
7.20.9 TMHL_BubbleSortWithTwoConjugateVectors . . . . .	307
7.21 Статистика и теория вероятности . . . . .	309
7.21.1 MHL_DensityOfDistributionOfNormalizedCenteredNormalDistribution . . . . .	309
7.21.2 MHL_DistributionFunctionOfNormalDistribution . . . . .	310
7.21.3 MHL_DistributionFunctionOfNormalizedCenteredNormalDistribution . . . . .	310
7.21.4 MHL_LeftBorderOfWilcoxonWFromTable . . . . .	312
7.21.5 MHL_RightBorderOfWilcoxonWFromTable . . . . .	313
7.21.6 MHL_StdDevToVariance . . . . .	314
7.21.7 MHL_VarianceToStdDev . . . . .	314
7.21.8 MHL_WilcoxonW . . . . .	315
7.21.9 TMHL_Mean . . . . .	316
7.21.10 TMHL_Median . . . . .	317
7.21.11 TMHL_SampleCovariance . . . . .	318
7.21.12 TMHL_Variance . . . . .	319
7.22 Тестовые функции для оптимизации . . . . .	320
7.22.1 MHL_TestFunction_Ackley . . . . .	320
7.22.2 MHL_TestFunction_AdditivePotential . . . . .	323
7.23 Основная задача и подзадачи . . . . .	325

7.24 Нахождение ошибки оптимизации . . . . .	325
7.25 Свойства задачи . . . . .	326
7.25.1 MHL_TestFunction_MultiplicativePotential . . . . .	326
7.26 Параметры для алгоритмов оптимизации . . . . .	328
7.27 Основная задача и подзадачи . . . . .	328
7.28 Нахождение ошибки оптимизации . . . . .	328
7.29 Свойства задачи . . . . .	329
7.29.1 MHL_TestFunction_ParaboloidOfRevolution . . . . .	329
7.29.2 MHL_TestFunction_Rastrigin . . . . .	332
7.29.3 MHL_TestFunction_Rosenbrock . . . . .	335
7.29.4 MHL_TestFunction_SumVector . . . . .	338
7.30 Тригонометрические функции . . . . .	341
7.30.1 MHL_Cos . . . . .	341
7.30.2 MHL_CosDeg . . . . .	341
7.30.3 MHL_Cosec . . . . .	342
7.30.4 MHL_CosecDeg . . . . .	342
7.30.5 MHL_Cotan . . . . .	343
7.30.6 MHL_CotanDeg . . . . .	343
7.30.7 MHL_Sec . . . . .	344
7.30.8 MHL_SecDeg . . . . .	344
7.30.9 MHL_Sin . . . . .	345
7.30.10 MHL_SinDeg . . . . .	345
7.30.11 MHL_Tan . . . . .	346
7.30.12 MHL_TanDeg . . . . .	346
7.31 Уравнения . . . . .	347
7.31.1 MHL_QuadraticEquation . . . . .	347

**Список литературы** **348**

# 1 Описание

**Сайт:** <https://github.com/Harrix/HarrixMathLibrary>.

**Что это такое?** Сборник различных математических функций и шаблонов с открытым кодом на языке C++. Упор делается на алгоритмы искусственного интеллекта. Используется только C++.

**Что из себя это представляет?** Фактически это .cpp и .h файл с исходниками функций и шаблонов, который можно прикрепить к любому проекту на C++. В качестве подключаемых модулей используется только: stdlib.h, time.h, math.h. Также используются файлы сторонней библиотеки в виде файлов mtrand.cpp и mtrand.h, для генерации псевдослучайных чисел авторства Takuji Nishimura, Makoto Matsumoto, Jasper Bedaux.

**Сколько?** На данный момент опубликовано функций: **292**.

В подсчете участвуют только функции из файлов HarrixMathLibrary.cpp и HarrixMathLibrary.h. Функции из сторонней библиотеки mtrand.cpp, в которой реализован генератор случайных псевдослучайных чисел, не учитываются.

**На какие алгоритмы делается упор?** Генетические алгоритмы, алгоритмы оптимизации первого порядка и другие системы искусственного интеллекта.

**По какой лицензии выпускается?** Библиотека распространяется по лицензии Apache License, Version 2.0.

**Как найти автора?** С автором можно связаться по адресу [sergienkoanton@mail.ru](mailto:sergienkoanton@mail.ru) или <http://vk.com/harrix>. Сайт автора, где публикуются последние новости: <http://blog.harrix.org>, а проекты располагаются по адресу <http://harrix.org>.

## Ваши действия:

- [Как установить](#) и пользоваться библиотекой.
- [Посмотреть](#) все функции библиотеки. Все функции рассортированы по категориям.
- [Читать](#) о случайных числах в библиотеке.
- [Как добавить](#) свои новые функции в библиотеку.

## 2 Установка

Если вы хотите только пользоваться библиотекой, то вам нужна из всего проекта только папки **\_library**, в которой располагается собранная библиотека и справка по ней, и папка **demo**, в которой находится программа с демонстрацией работы функций. Все остальные папки вам потребуются, если вы хотите добавлять новые функции.

### 2.1 Общий алгоритм подключения

- Скопируем себе папку **\_library** с готовой последней версией библиотеки на сайте проекта <https://github.com/Harrix/HarrixMathLibrary>.
- Скопируем файлы **HarrixMathLibrary.cpp**, **HarrixMathLibrary.h**, **mtrand.cpp**, **mtrand.h** в папку с Вашим проектом на C++.
- Пропишем в проекте:

Код 1. Подключение библиотеки

```
#include "HarrixMathLibrary.h"
```

- Если планируем использовать функции, использующие случайные числа (если не знаем, то тоже сделаем), то в начале программы вызовем:

Код 2. Инициализация генератора случайных чисел

```
MHL_SeedRandom(); //Инициализировали генератор случайных чисел
```

- Теперь библиотека готова к работе, и можем ее использовать. Например:

Код 3. Пример использования

```
double x;  
x=MHL_RandomNumber();  
double degree=MHL_DegToRad(60);
```

### 2.2 Подключение к Qt на примере Qt 5.1.1

Рассматривается на примере создания Qt Gui Application в Qt 5.1.1 for Desktop (MinGW 4.8) с использованием Qt Creator 2.8.1.

- Скопируем себе папку **\_library** с готовой последней версией библиотеки на сайте проекта <https://github.com/Harrix/HarrixMathLibrary>.
- Скопируем файлы **HarrixMathLibrary.cpp**, **HarrixMathLibrary.h**, **mtrand.cpp**, **mtrand.h** в папку с Вашим проектом на C++ там, где находится файл проекта \*.pro.
- Добавим к проекту файлы **HarrixMathLibrary.cpp**, **HarrixMathLibrary.h**, **mtrand.cpp**, **mtrand.h**. Для этого по проекту в Qt Creator щелкнем правой кнопкой и вызовем команду **Add Existing Files...**, где выберем наши файлы.
- Пропишем в главном файле исходников проекта **mainwindow.cpp**:

Код 4. Подключение библиотеки

```
#include "HarrixMathLibrary.h"
```

- Если планируем использовать функции, использующие случайные числа (если не знаем, то тоже сделаем), то в начале программы в конструкторе **MainWindow::MainWindow** вызовем:

Код 5. Инициализация генератора случайных чисел

```
MHL_SeedRandom(); //Инициализировали генератор случайных чисел
```

То есть получится код:

Код 6. Пример файла mainwindow.cpp с подключенной библиотекой

```
#include "mainwindow.h"
#include "ui_mainwindow.h"

#include "HarrixMathLibrary.h"

MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);
    MHL_SeedRandom(); //Инициализировали генератор случайных чисел
}

MainWindow::~MainWindow()
{
    delete ui;
}
```

- Теперь библиотека готова к работе, и можем ее использовать. Например, добавим **textEdit**, **pushButton** и напишем слот кнопки:

Код 7. Пример использования

```
void MainWindow::on_pushButton_clicked()
{
    double x;
    x=MHL_RandomNumber();
    double degree=MHL_DegToRad(60);
}
```

## 2.3 Подключение к C++ Builder на примере C++ Builder 6.0

- Скопируем себе папку **\_library** с готовой последней версией библиотеки на сайте проекта <https://github.com/Harrix/HarrixMathLibrary>.
- Скопируем файлы **HarrixMathLibrary.cpp**, **HarrixMathLibrary.h**, **mtrand.cpp**, **mtrand.h** в папку с проектом на C++.
- Пропишем проекте в файле **.cpp** главной формы (часто это **Unit1.cpp**) строчку **#include "HarrixMathLibrary.h"**:

#### Код 8. Подключение библиотеки

```
//-----  
  
#include <vcl.h>  
#pragma hdrstop  
  
#include "Unit1.h"  
#include "HarrixMathLibrary.h"  
//-----  
#pragma package(smart_init)  
#pragma resource "* .dfm"  
TForm1 *Form1;  
...
```

- Добавим в проект файлы **HarrixMathLibrary.cpp** и **mtrand.cpp** через команду: **Project → Add to Project...**
- Если планируем использовать функции, использующие случайные числа (если не знаем, то тоже сделаем), то в конструкторе главной формы инициализируем генератор случайных чисел:

#### Код 9. Инициализация генератора случайных чисел

```
__fastcall TForm1::TForm1(TComponent* Owner)  
    : TForm(Owner)  
{  
    MHL_SeedRandom(); //Инициализировали генератор случайных чисел  
    ...  
}  
//-----
```

- Теперь библиотека готова к работе, и можем ее использовать. Например, создадим кнопку Button1, текстовое поле Memo1 и в клике на Button1 пропишем:

#### Код 10. Пример использования

```
void __fastcall TForm1::Button1Click(TObject *Sender)  
{  
    double x=MHL_RandomNumber(); //получим случайное число  
    Memo1->Lines->Add("x = "+AnsiString(x)); //выведем его  
}  
//-----
```

## 2.4 Подключение к C++ Builder на примере C++Builder XE4

- Скопирую себе папку **\_library** с готовой последней версией библиотеки на сайте проекта <https://github.com/Harrix/HarrixMathLibrary>.
- Скопирую файлы **HarrixMathLibrary.cpp**, **HarrixMathLibrary.h**, **mtrand.cpp**, **mtrand.h** в папку с проектом на C++.
- Пропишу в файле **.cpp** главной формы (часто это **Unit1.cpp**) строку **#include "HarrixMathLibrary.h"**:

### Код 11. Подключение библиотеки

```
//-----  
  
#include <vcl.h>  
#pragma hdrstop  
  
#include "Unit1.h"  
#include "HarrixMathLibrary.h"  
//-----  
#pragma package(smart_init)  
#pragma resource "*.*"  
TForm1 *Form1;  
//-----  
...
```

- Добавим в проект файлы **HarrixMathLibrary.cpp** и **mtrand.cpp** через команду: **Project → Add to Project...**
- Если планируем использовать функции, использующие случайные числа (если не знаем, то тоже сделаем), то в конструкторе главной формы инициализируем генератор случайных чисел:

### Код 12. Инициализация генератора случайных чисел

```
__fastcall TForm1::TForm1(TComponent* Owner)  
    : TForm(Owner)  
{  
    MHL_SeedRandom(); //Инициализировали генератор случайных чисел  
    ...  
}  
//-----
```

- Теперь библиотека готова к работе, и можем ее использовать. Например, создадим кнопку Button1, текстовое поле Memo1 и в клике на Button1 пропишем:

### Код 13. Пример использования

```
void __fastcall TForm1::Button1Click(TObject *Sender)  
{  
    double x=MHL_RandomNumber(); //получим случайное число  
    Memo1->Lines->Add("x = "+AnsiString(x)); //выведем его  
}  
//-----
```

## 2.5 Подключение к Microsoft Visual Studio на примере Visual Studio 2012

Используется CLR приложение Windows Forms Application (точнее пустой проект, к которому прикреплена форма) на Visual C++.

- Скопируем себе папку **\_library** с готовой последней версией библиотеки на сайте проекта <https://github.com/Harrix/HarrixMathLibrary>.
- Скопируем файлы **HarrixMathLibrary.cpp**, **HarrixMathLibrary.h**, **mtrand.cpp**, **mtrand.h** в папку с проектом \*.vcxproj на C++.

- Пропишем проекте в файле **.h** главной формы (у меня это **MyForm.h**) строчку **#include "HarrixMathLibrary.h"**:

Код 14. Подключение библиотеки

```
#pragma once
#include "HarrixMathLibrary.h"
...
```

- Добавим в проект файлы **HarrixMathLibrary.cpp** и **mtrand.cpp** через правый клик по проекту: **Добавить → Существующий элемент Shift+Alt+A**.
- Если планируем использовать функции, использующие случайные числа (если не знаем, то тоже сделаем), то в конструкторе главной формы инициализируем генератор случайных чисел:

Код 15. Инициализация генератора случайных чисел

```
public ref class MyForm : public System::Windows::Forms::Form
{
public:
    MyForm(void)
    {
        MHL_SeedRandom(); //Инициализировали генератор случайных чисел
        InitializeComponent();
        //
        //TODO: добавьте код конструктора
        //
    }
    ...
}
```

- Теперь библиотека готова к работе, и можем ее использовать. Например, создадим кнопку button1 и listBox1 и в клике на button1 пропишем:

Код 16. Пример использования

```
private: System::Void button1_Click(System::Object^ sender, System::EventArgs^ e) {
    double x=MHL_RandomNumber(); //получим случайное число
    listBox1->Items->Add("x = " + x.ToString()); //выведем его
}
```

Как видите, алгоритм подключения почти одинаков.

### 3 О случайных числах в библиотеке HarrixMathLibrary

**Генератор случайных чисел (ГСЧ)** — очень важная и нужная функция в программировании. При этом необходим лишь первичный генератор — генератор случайных вещественных чисел в интервале  $(0; 1)$  по равномерному закону распределения (везде, где пишется о генераторе случайных чисел, надо понимать, что говорится о генераторе псевдослучайных чисел). Все остальные случайные числа с другими законами распределения можно получить из равномерного.

По умолчанию в библиотеке используется генератор случайных чисел Mersenne Twister. Также есть стандартный генератор случайных чисел.

Итак, что есть в библиотеке? Есть три функции и несколько переменных. Рассмотрим функции:

- **MHL\_SeedRandom()** — инициализатор генератора случайных чисел. Нужно вызвать один раз за всё время запуска программы, в которой используется библиотека.
- **MHL\_RandomNumber()** — непосредственно генератор случайных чисел.
- **MHL\_SetRandomNumberGenerator(TypeOfRandomNumberGenerator T)** — функция позволяет переназначить генератор случайных чисел на другой.

В файле **HarrixMathLibrary.h** (после объявления констант в начале файла) есть строчки, которые объявляют эти вещи:

Код 17. Объявление функций в HarrixMathLibrary.h

```
enum TypeOfRandomNumberGenerator { StandardRandomNumberGenerator,
    MersenneTwisterRandomNumberGenerator };//тип генератора случайных чисел: стандартный или MersenneTwister:
void MHL_SeedRandom(void); //Инициализатор генератора случайных чисел
double MHL_RandomNumber(void); //Генерирует вещественное случайное число из интервала (0,1)
void MHL_SetRandomNumberGenerator(TypeOfRandomNumberGenerator T); //Переназначить генератор случайных чисел на другой
```

Код 18. Объявление переменной в HarrixMathLibrary.cpp

```
//ДЛЯ ГЕНЕРАТОРОВ СЛУЧАЙНЫХ ЧИСЕЛ
unsigned int MHL_Dummy; //Результат инициализации генератора случайных чисел
TypeOfRandomNumberGenerator MHL_TypeOfRandomNumberGenerator; //тип генератора случайных чисел
MTRand mt((unsigned)time(NULL)); //Инициализатор генератора случайных чисел Mersenne Twister
MTRand drand; //Для генерирования случайного числа в диапозоне [0,1].
```

В случае своего желания Вы можете заменить тело функций **MHL\_SeedRandom()** и **MHL\_RandomNumber()** на свои собственные. Ниже представлены варианты, которые предлагаются автором.

Код 19. Стандартный вариант по умолчанию

```
void MHL_SeedRandom(void)
{
/*
Инициализатор генератора случайных чисел.
```

*В данном случае используется самый простой его вариант со всеми его недостатками.*

*Входные параметры:*

*Отсутствуют.*

*Возвращаемое значение:*

*Отсутствует.*

*\*/*

*//StandardRandomNumberGenerator*

*//Инициализатор стандартного генератора случайных чисел*

*//В качестве начального значения для ГСЧ используем текущее время*

*MHL\_Dummy=(unsigned)time(NULL);*

*srand(MHL\_Dummy); //Стандартная инициализация*

*//rand(); //первый вызов для контроля*

*//MersenneTwisterRandomNumberGenerator*

*//Инициализатор генератора случайных чисел Mersenne Twister*

*//В качестве начального значения для ГСЧ используем текущее время*

*//Инициализация происходит еще при подключении данного файла*

*//Назначаем генератор по умолчанию как Mersenne Twister*

*MHL\_TypeOfRandomNumberGenerator = MersenneTwisterRandomNumberGenerator;*

*}*

*-----*

**double** MHL\_RandomNumber(**void**)

*{*

*/\**

*Генератор случайных чисел (ГСЧ).*

*Есть два варианта генератора случайных чисел, который можно переключать функцией MHL\_SetRandomNumberGenerator.*

*Входные параметры:*

*Отсутствуют.*

*Возвращаемое значение:*

*Случайное вещественное число из интервала (0;1) или [0;1) по равномерному закону распределения.*

*\*/*

**if** (MHL\_TypeOfRandomNumberGenerator==StandardRandomNumberGenerator)  
    **return** (**double**)rand()/(RAND\_MAX+1);

**if** (MHL\_TypeOfRandomNumberGenerator==MersenneTwisterRandomNumberGenerator)  
    **return** drand();

**return** 0;

*}*

*-----*

**void** MHL\_SetRandomNumberGenerator(**TypeOfRandomNumberGenerator** T)

*{*

*/\**

*Функция переназначает генератор случайных чисел.*

*Входные параметры:*

*TypeOfRandomNumberGenerator - тип генератора случайных чисел:*

*StandardRandomNumberGenerator - стандартный генератор случайных чисел;*

*MersenneTwisterRandomNumberGenerator - генератор случайных чисел Mersenne Twister.*

*Возвращаемое значение:*

*Отсутствует.*

*\*/*

**MHL\_TypeOfRandomNumberGenerator = T;**

*}*

*-----*

Теперь разберем, как применять данные функции.

- Подключаем библиотеку к Вашему проекту на C++ (об этом читайте в главе об установке).
- В начале программы **один** раз вызываем функцию `MHL_SeedRandom()`. Ниже приведены примеры, где обычно стоит вызывать эту функцию.

Код 20. Применение `MHL_SeedRandom` для консольного приложения

```
int main(void)
{
    MHL_SeedRandom(); //Инициализировали генератор случайных чисел
    ...
}
```

Код 21. Применение `MHL_SeedRandom` для C++Builder

```
__fastcall TForm1::TForm1(TComponent* Owner)
    : TForm(Owner)
{
    MHL_SeedRandom(); //Инициализировали генератор случайных чисел
    ...
}
```

Код 22. Применение `MHL_SeedRandom` для Qt

```
MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);
    MHL_SeedRandom(); //Инициализировали генератор случайных чисел
    ...
}
```

- Теперь в любом месте программы мы можем получить случайное число из интервала  $(0; 1]$ . Например:

Код 23. Применение ГСЧ

```
double x;
x=MHL_RandomNumber();
```

Результат вызова функции, например:  $x = 0,420933187007904$ .

А с помощью функции **`MHL_SetRandomNumberGenerator`** можно поменять генератор случайных чисел. С помощью данного вызова меняем на стандартный генератор случайных чисел.

Код 24. Меняем на стандартный генератор случайных чисел

```
MHL_SetRandomNumberGenerator(StandardRandomNumberGenerator);
```

А так меняем на генератор Mersenne Twister (он стоит по умолчанию).

Код 25. Меняем на генератор случайных чисел Mersenne Twister

```
MHL_SetRandomNumberGenerator(MersenneTwisterRandomNumberGenerator);
```

В библиотеке используется сторонняя реализация генератора случайных чисел Mersenne Twister, и эта реализация располагается в файлах mtrand.cpp и mtrand.h.

Вы можете заменить код функций (`MHL_SeedRandom`, `MHL_RandomNumber`) на свой генератор случайных чисел в интервале (0; 1). При этом работоспособность библиотеки не нарушится.

## 4 Как добавлять новые функции в библиотеку

Данная глава предназначена для тех, кто хочет добавлять в библиотеку новые функции и развивать данный продукт.

### Ваши действия:

- **Шаг 0.** Прочитать некоторую справочную информацию.
- **Шаг 1.** Написать и проверить свою функцию в папке **source\_demo**.
- **Шаг 2.** Раскидать в функцию по файлам в папке исходников **source\_library**.
- **Шаг 3.** Собрать библиотеку в папке **make**.
- **Шаг 4.** Раскидать файлы собранной библиотеки из папки **temp\_library** по папкам библиотеки и перекомпилировать некоторые программы и справки.

### Шаг 0. Справочная информация.

Вначале надо сориентироваться в структуре библиотеки:

- **\_library** — основная папка, в которой располагается готовая библиотека и данная справка;
- **demo** — папка, в которой находится программа DemoHarrixMathLibrary.exe с демонстрацией работы функций;
- **make** — в этой папке находится программа MakeHarrixMathLibrary.exe для сборки готовых файлов библиотеки из исходных материалов из папки source\_library. Также там находится справка по этой программе;
- **source\_demo** — папка с исходными кодами DemoHarrixMathLibrary.exe из папки demo;
- **source\_library** — папка исходных материалов библиотеки. Сами эти файлы библиотекой не являются, так как они потом собираются MakeHarrixMathLibrary.exe;
- **source\_make** — папка с исходными кодами MakeHarrixMathLibrary.exe из папки make;
- **LICENSE.txt** и **NOTICE.txt** — файлы Apache лицензии;
- **README.md** — файл информации о проекте в системе GitHub.

Для полноценной работы по добавлению функций вам потребуются:

- программа для проверки работоспособности новых функций и компиляции DemoHarrixMathLibrary.exe (например, Qt 5.1.1 с Qt Creator 2.8.1 или любая другая версия Qt с пятой версии). Для проверки работоспособности библиотеки без компиляции DemoHarrixMathLibrary.exe подойдет любой другой C++ компилятор;
- программа для компиляции \*.tex документов в \*.pdf для формирования справочных материалов. Автор использует для этого связку MiKTeX и TeXstudio (версии MiKTeX 2.9 и TeXstudio 2.6.2).

В варианте, который использует автор, в \*.tex файлах справок для отображения русских букв используется модуль **pscyr**. Об его установке (там можно и скачать) можно прочитать в статье [http://blog.harxit.org/?p=444](http://blog.harrix.org/?p=444).

Далее приведены некоторые спецификации, принятые в данной библиотеке.

- Основу библиотеки составляют функции и шаблоны функций. Имена функций начинаются с **MHL\_**, например:

Код 26. Пример названия функции

```
void MHL_NormalizationVectorOne(double *VMHL_ResultVector, int VMHL_N);
```

Имена же шаблонов начинаются с **TMHL\_**, например:

Код 27. Пример названия шаблона функции

```
template <class T> int TMHL_SearchFirstZero(T *VMHL_Vector, int VMHL_N);
```

Код функций в итоге будет располагаться в HarrixMathLibrary.cpp, а реализация шаблонов будет располагаться в HarrixMathLibrary.h. Содержимое файлов mtrand.cpp и mtrand.h трогать не нужно.

- Количество элементов в одномерном массиве обозначается стандартной переменной **int VMHL\_N**.
- Количество элементов в двумерном массиве обозначается стандартными переменными **int VMHL\_N** и **int VMHL\_M**.
- Возвращаемое значение функций обозначается переменной **VMHL\_Result**.
- Возвращаемый вектор (над которым производятся вычисления) обозначается указателем **\*VMHL\_ResultVector**.
- Возвращаемая матрица (над которой производятся вычисления) обозначается указателем **\*\*VMHL\_ResultMatrix**.
- Если функция в качестве параметра имеет одну числовую переменную, то она обозначается **VMHL\_X** или **VMHL\_X1**. Если есть однотипные переменные, то обозначаются **VMHL\_X2** или **VMHL\_Y** и так далее (данная рекомендация даже автором редко соблюдается).
- Если функция в качестве параметра имеет некий вектор, то он обозначается **VMHL\_Vector** (данная рекомендация даже автором редко соблюдается).
- Если функция в качестве параметра имеет некую матрицу, то она обозначается **VMHL\_Matrix** (данная рекомендация даже автором редко соблюдается).
- То есть если входные переменные не имеют какой-то особый смысл, то название переменных стандартно, но в тоже время все входные и выходные переменные могут начинаться с **VMHL\_**, чтобы различать их от внутренних, но во отличии от выходных значений это есть **не обязательное условие**.

Далее приведена последовательность действий, которую надо выполнить для добавления новой функции. Допустим мы хотим добавить функцию **double MHL\_Func(double VMHL\_X)**.

**Шаг 1.** Вначале нам нужно реализовать саму функцию и проверить ее работоспособность. Если вы хотите работать не через средства, предоставляемые библиотекой, то этот шаг можно пропустить.

- Заходим в папку **source\_demo** и открываем проект **DemoHarrixMathLibrary.pro** в Qt Creator.
- Компилируем проект и в папку с скомпилированным приложением (в режиме Release) из папки **demo** скопируем файлы:
  - папка images;
  - index.html;
  - jquery.js;
  - jsxgraphcore.js;
  - style.css.
- Добавляем в конец файлов **HarrixMathLibrary.cpp** и **HarrixMathLibrary.h** функцию, которую хотим добавить. Например, в HarrixMathLibrary.cpp добавляем:

Код 28. Что добавляем в HarrixMathLibrary.cpp

```
int MHL_Func(int VMHL_X)
{
/*
Умножает число на 2.
Входные параметры:
  x - число, которое будет умножаться.
Возвращаемое значение:
  Число, умноженное на 2.
*/
    return 2*VMHL_X;
}
```

А в HarrixMathLibrary.h добавляем:

Код 29. Что добавляем в HarrixMathLibrary.h

```
int MHL_Func(int VMHL_X);
```

**Замечание.** В .h файл добавляем до строчки «#endif // HARRIXMATHLIBRARY\_H».

**Замечание.** Если вы добавляете шаблон функции, то его реализацию надо добавлять в HarrixMathLibrary.h.

- Теперь перейдем в проекте DemoHarrixMathLibrary.pro в файл **mainwindow.cpp**.
- Вначале этого файла идет следующий код:

Код 30. mainwindow.cpp

```
#include "mainwindow.h"
#include "ui_mainwindow.h"
#include <QDebug>
#include <QFile>
#include <QDesktopServices>
#include <QUrl>
#include <QDir>
#include <QStandardItemModel>

#include "HarrixMathLibrary.h"

#include "HarrixQtLibrary.h"
```

```

MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);

    DS=QDir::separator();
    path=QGuiApplication::applicationDirPath() + DS; //путь к папке

    MHL_SeedRandom(); //Инициализация датчика случайных чисел

    model = new QStandardItemModel(this);
    model->setObjectName(QString::fromUtf8("model"));

    QStandardItem *item; //элемент списка

    //добавление новых элементов
    item = new QStandardItem(QString("TMHL_FillVector"));
    model->appendRow(item);

    //Сюда нужно добавить код

    ...

    //соединение модели списка с конкретным списком
    ui->listView->setModel(model);

    ui->listView->setEditTriggers(QAbstractItemView::NoEditTriggers);
}

```

- Там, где написан комментарий «//**Сюда нужно добавить код**» необходимо добавить две строчки:

Код 31. Что добавить в mainwindow.cpp

```

item = new QStandardItem(QString("[Имя вашей функции]));
model->appendRow(item);

```

То есть в рассматриваемом примере вы должны добавить:

Код 32. Что добавить в mainwindow.cpp в примере

```

item = new QStandardItem(QString("MHL_Func"));
model->appendRow(item);

```

Добавление данного кода добавить вашу функцию в список, которые будут отображаться в программе при запуске. По сути, удобнее было бы извлекать из обычного текстового файла. Может в будущих версиях так и сделаю, но все равно вам нужно потом писать код демонстрации функций, поэтому занесение в текстовой файл не предусмотрел.

- Далее найдем функцию **MainWindow::on\_listView\_clicked**:

Код 33. MainWindow::on\_listView\_clicked

```

void MainWindow::on_listView_clicked(const QModelIndex &index)
{

```

```

Html=<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN" "http://www.w3.org/TR/REC-html40/strict.dtd"><html><head><meta name="qrichtext" content="1" /><meta http-equiv="Content-Type" content="text/html; charset=utf-8" /><style type="text/css">np, li { white-space: pre-wrap; }</style></head><body style=" font-family:'MS Shell Dlg 2'; font-size:8.25pt; font-weight:400; font-style:normal ;">\n";

QString NameFunction;//Какая функция вызывается

//выдергиваем текст
NameFunction=index.data(Qt::DisplayRole).toString();

//Сюда нужно добавить код

...

//Показ итогового результата
Html+="</body></html>";
HQt_SaveFile(Html, path+"temp.html");
ui->webView->setUrl(QUrl::fromLocalFile(path+"temp.html"));
}

```

- Там, где написан комментарий «**//Сюда нужно добавить код**» добавляете код следующего типа:

#### Код 34. Добавление демонстрации работы функции

```

if (NameFunction=="[Имя вашей функции]")
{
//Реализация демонстрации функции
}

```

Вместо **«[Имя вашей функции]»** пишите название вашей функции, такое же, что добавляли выше. Вместо комментария **«//Сюда нужно добавить код»** добавьте реализацию демонстрации вашей функции. Например, для рассматриваемого примера код будет выглядеть так:

#### Код 35. Добавление демонстрации работы функции на примере

```

if (NameFunction=="MHL_Func")
{
    int x=5;

    //Вызов функции
    int y=MHL_Func(x);

    //Используем полученный результат
    MHL_ShowNumber (x, "Первоначальное число", "x");
    MHL_ShowNumber (y, "Умноженное число", "y");
    //Первоначальное число:
    //x=5
    //Умноженное число:
    //y=10
}

```

- Рассмотрим немного этот код. После вызова функции идет комментарий **«//Используем полученный результат»**. После него надо вывести в webView нуж-

ную информацию. Для этого лучше использовать стандартные функции, список которых написан ниже.

- После вывода функций в виде комментариев показывается тот текст, который может продемонстрироваться при вызове функции. У нас это код:

Код 36. Закомментированный результат работы функции

```
//Первоначальное число:  
//x=5  
//Умноженное число:  
//y=10
```

Теперь рассмотрим какие функции используются для вывода результата. Типичными объектами, над которыми выполняются действия по выводу, являются: числа, вектора, матрицы. Их мы стандартизовано и выводим, используя некоторые функции. Так как библиотека HarrixMathLibrary может использоваться на различных системах C++, а вывод информации в каждой системе может быть разным, то функции вывода строились таким образом, чтобы внешне выглядели однотипно в любой системе C++, так как в справке к функциям из библиотеки функции вывода также будут присутствовать. Итак, использование функций внешне должно быть везде одинаковым для всех систем C++. Поэтому вы можете их переписать под свои нужды.

- **MHL\_NumberToText** — функция перевода числа в строку;
- **MHL\_ShowNumber** — функция вывода числа;
- **MHL\_ShowVector** — функция вывода вектора (одномерного массива);
- **MHL\_ShowVectorT** — функция вывода вектора (одномерного массива) в строку одну, то есть это транспонированный вектор;
- **MHL\_ShowMatrix** — функция вывода матрицы.
- **MHL\_ShowText** — функция вывода просто текста.

Далее функции рассмотрены подробнее.

- **MHL\_ShowNumber** — функция вывода числа.

Код 37. Синтаксис функции MHL\_ShowNumber

```
template <class T> void MHL_ShowNumber (T VMHL_X, QString TitleX, QString  
NameX);
```

Входные параметры:

- VMHL\_X — выводимое число;
- TitleX — заголовок выводимого числа;
- NameX — обозначение числа.

Пример использования функции:

Код 38. Пример использования MHL\_ShowNumber

```
MHL_ShowNumber (x, "Первоначальное число", "x");  
//Первоначальное число:  
//x=5
```

И для этой функции покажем исходный код:

Код 39. Реализация функции MHL\_ShowNumber

```
//mainwindow.cpp
template <class T> void MainWindow::MHL_ShowNumber (T VMHL_X, QString TitleX,
    QString NameX)
{
    /*
    Функция выводит число VMHL_X в textEdit.
    Входные параметры:
        VMHL_X - выводимое число;
        TitleX - заголовок выводимого числа;
        NameX - обозначение числа.
    Возвращаемое значение:
        Отсутствует.
    */
    QString VMHL_Result;
    VMHL_Result=THQt_ShowNumber (VMHL_X, TitleX, NameX); // из HarrixQtLibrary.
    h
    Html+=VMHL_Result;
}
//-----

//HarrixQtLibrary.h
template <class T> QString THQt_ShowNumber (T VMHL_X, QString TitleX, QString
NameX)
{
    /*
    Функция возвращает строку с выводом некоторого числа VMHL_X с HTML кодами.
    Для добавление в html файл.
    Входные параметры:
        VMHL_X - выводимое число;
        TitleX - заголовок выводимого числа;
        NameX - обозначение числа.
    Возвращаемое значение:
        Стока с HTML кодами с выводимым числом.
    */
    QString VMHL_Result;

    VMHL_Result=<p><b>" +TitleX+" :</b><br>" ;

    VMHL_Result+=NameX+"=<b><font color=\\"#4200ff\\">" +QString::number(VMHL_X)+"
    "</font></b></p>\n";

    return VMHL_Result;
}
//-----
```

В функции MainWindow::on\_listView\_clicked() есть еще код для сохранения и вывода значения переменной Html в виде \*.html файла.

В предыдущей версии библиотеки для программы демонстрации работы функций использовалась система C++Builder 6. Там эта функции реализовывалась так:

Код 40. Реализация функции MHL\_ShowNumber в C++Builder 6

```
template <class T> void MHL_ShowNumber (T X, AnsiString A, AnsiString B)
{
Form1->Memo1->Lines->Add(A+":");
Form1->Memo1->Lines->Add(B+" = "+AnsiString(X));
Form1->Memo1->Lines->Add("");
```

```
}
```

```
//-----
```

Как видим, вид функций по внешнему виду однотипен — различается только тип строк, который используется.

- **MHL\_NumberToText** — выводит число в строку.

Код 41. Синтаксис функции MHL\_NumberToText

```
template <class T> QString MainWindow::MHL_NumberToText (T VMHL_X);
```

Входные параметры:

- VMHL\_X — выводимое число.

Пример использования функции:

Код 42. Пример использования MHL\_NumberToText

```
MHL_ShowNumber(Deg, "Угол "+MHL_NumberToText(Rad)+" радиан", "равен в градусах")
;
//Угол 3.14159 радиан:
//равен в градусах=180
```

- **MHL\_ShowVector** — функция вывода вектора (одномерного массива).

Код 43. Синтаксис функции MHL\_ShowVector

```
template <class T> void MHL_ShowVector (T *VMHL_Vector, int VMHL_N, QString
TitleVector, QString NameVector);
```

Входные параметры:

- Vector — указатель на выводимый вектор;
- VMHL\_N — количество элементов вектора a;
- TitleVector — заголовок выводимого вектора;
- NameVector — обозначение вектора.

Пример использования функции:

Код 44. Пример использования MHL\_ShowVector

```
MHL_ShowVector (a,VMHL_N,"Заполненный вектор", "a");
//Заполненный вектор:
//a =
//5
//5
//5
//5
//5
//5
//5
//5
//5
//5
```

- **MHL\_ShowVectorT** — функция вывода вектора (одномерного массива) в транспонированном виде, то есть в одну строку.

Код 45. Синтаксис функции MHL\_ShowVectorT

```
template <class T> void MHL_ShowVectorT (T *VMHL_Vector, int VMHL_N, QString TitleVector, QString NameVector);
```

Входные параметры:

- Vector — указатель на выводимый вектор;
- VMHL\_N — количество элементов вектора a;
- TitleVector — заголовок выводимого вектора;
- NameVector — обозначение вектора.

Пример использования функции:

Код 46. Пример использования MHL\_ShowVectorT

```
MHL_ShowVector (a,VMHL_N,"Заполненный вектор", "a");  
//Заполненный вектор:  
//a = 5 5 5 5 5 5 5 5 5 5
```

- **MHL\_ShowMatrix** — функция вывода матрицы.

Код 47. Синтаксис функции MHL\_ShowMatrix

```
template <class T> void MHL_ShowMatrix (T **VMHL_Matrix, int VMHL_N, int VMHL_M, QString TitleMatrix, QString NameMatrix);
```

Входные параметры:

- VMHL\_Matrix — указатель на выводимую матрицу;
- VMHL\_N — количество строк в матрице;
- VMHL\_M — количество столбцов в матрице;
- TitleMatrix — заголовок выводимой матрицы;
- NameMatrix — обозначение матрицы.

Пример использования функции:

Код 48. Пример использования MHL\_ShowMatrix

```
MHL_ShowMatrix (Matrix,VMHL_N,VMHL_M,"Матрица", "x");  
//Матрица:  
//x =  
//0 1 2 3 4  
//1 2 3 4 5  
//2 3 4 5 6  
//3 4 5 6 7  
//4 5 6 7 8  
//5 6 7 8 9  
//6 7 8 9 10
```

- **MHL\_ShowText** — функция вывода просто текста.

Код 49. Синтаксис функции MHL\_ShowText

```
void MHL_ShowText (QString TitleX);
```

Входные параметры:

- TitleX - непосредственно выводимая строка.

Пример использования функции:

Код 50. Пример использования MHL\_ShowText

```
MHL_ShowText ("Выvodimyj tekstu");
//Выводимый текст
```

Итак, мы добавили в DemoHarrixMathLibrary.pro нашу функцию и проверили ее работоспособность.

**Шаг 2.** Теперь нам нужно добавить нашу функцию в исходники. Все исходные материалы располагаются в папке **source\_library**. В ней располагаются некоторые файлы, которые нам не особы интересны (подробнее в файле справке к программе MakeHarrixMathLibrary.exe в файле **make\MakeHarrixMathLibrary\_Help.pdf**) и папки (например, **Вектора (Одномерные массивы)**). Каждая такая папка является разделом функций в библиотеке. Вам нужно выбрать папку, в которую вы будете добавлять свою функцию или создать свою собственную, если ничего не подходит по смыслу.

Каждая функция или шаблон функции в разделе (выбранной вами папке) предоставляется следующими файлами:

- <File>.cpp или <File>.tpp — код функции;
- <File>.h — заголовочный файл функции;
- <File>.tex — справка по функции;
- <File>.desc — описание функции;
- <File>.use — пример использования функции;
- <File>\_<name>.pdf — множество рисунков, необходимых для справки по функции (необязательные файлы);
- <File>\_<name>.png — множество рисунков, необходимых для справки по функции (необязательные файлы);

Без файлов <File>.cpp (или <File>.tpp), <File>.h, <File>.tex, <File>.desc, <File>.use библиотека соберется, но с ошибками, то есть каждая функция должна быть представима минимум 5 файлами (могут быть дополнительно рисунки).

Считаем далее, что вы выбрали папку <Dir> в папке source\_library.

- Создайте в папке <Dir> текстовой файл <File>.h, где <File> — это имя функции, то есть в рассматриваемом примере мы должны создать файл **MHL\_Func.h**.
- В файл <File>.h мы добавляем объявление нашей функции, например:

Код 51. Содержимое MHL\_Func.h

```
int MHL_Func(int VMHL_X);
```

- В файл <File>.cpp мы добавляем код нашей функции, например:

Код 52. Содержимое MHL\_Func.cpp

```
int MHL_Func(int VMHL_X)
{
```

```

/*
Умножает число на 2.
Входные параметры:
    x - число, которое будет умножаться.
Возвращаемое значение:
    Число, умноженное на 2.
*/
return 2*VMHL_X;
}

```

Если у нас не функция, а шаблон функции, то мы создаем файл <File>.tpp ( обратите внимание на расширение файла), например:

Код 53. Содержимое TMHL\_FillVector.tpp

```

template <class T> void TMHL_FillVector(T *VMHL_ResultVector, int VMHL_N, T x)
{
/*
Функция заполняет вектор значениями, равных x.
Входные параметры:
    VMHL_ResultVector - указатель на преобразуемый массив;
    VMHL_N - количество элементов в массиве;
    x - число, которым заполняется вектор.
Возвращаемое значение:
    Отсутствует.
*/
for (int i=0;i<VMHL_N;i++) VMHL_ResultVector[i]=x;
}

```

- В файл <File>.desc мы добавляем описание нашей функции, например:

Код 54. Содержимое MHL\_Func.desc

Умножает число на 2.

- В файл <File>.tex мы добавляем справку к нашей функции в виде куска tex кода, например:

Код 55. Содержимое MHL\_Func.tex

```

\textbf{Входные параметры:}

x --- входной параметр.

\textbf{Возвращаемое значение:}
Число умноженное на 2.

```

- В файл <File>.use мы добавляем код примера использования функции, например:

Код 56. Содержимое MHL\_Func.use

```

int x=5;

//Вызов функции
int y=MHL_Func(x);

//Используем полученный результат
MHL_ShowNumber (x,"Первоначальное число", "x");
MHL_ShowNumber (y,"Умноженное число", "y");
//Первоначальное число:

```

```
//x=5  
//Умноженное число:  
//y=10
```

- Если хотите использовать рисунки в tex справке к функции, то в папку <Dir> скопируйте рисунки вида <File>\_<name>.pdf и <File>\_<name>.png
- Если мы используем дополнительную переменную перечисляемого типа, то добавляем ее в файл **Enum.h** в папке **source\_library**.
- Если мы хотим использовать глобальную константу, то добавляем ее в файл **Const.h** в папке **source\_library**.
- Если мы хотим использовать глобальную переменную, то добавляем ее в файл **AdditionalVariables.cpp** в папке **source\_library**.

**Замечание.** Если вы хотите переопределить функцию какую-нибудь, то вы добавляете переопределенные функции, их объявления в уже существующие файлы, а не создаете новые.

**Замечание.** Класс и его методы нужно оформлять в одном файле \*.cpp, \*.h и др., а не разбивать на несколько и прописывать каждый метод в отдельном.

Итак, мы добавили в папку source\_library нашу функцию. Теперь нужно перестроить библиотеку и провести замену файлов.

**Шаг 3.** Сборка библиотеки. Перейдем в папку **make** в корне файлов библиотеки. В ней есть программа MakeHarrixMathLibrary.exe и справка к ней MakeHarrixMathLibrary\_Help.pdf.

- Включим программу **MakeHarrixMathLibrary.exe**.
- Нажмем кнопку **Собрать библиотеку**.
- В окне программы будет отчет об сборки библиотеки, например:

#### Код 57. Пример отчета о сборке библиотеки

```
Начало формирования файлов библиотеки...  
Загрузили файл Header.cpp  
Загрузили файл AdditionalVariables.cpp  
Загрузили файл Random.cpp  
Загрузили файл Const.h  
Загрузили файл Random.cpp  
Загрузили файл Enum.h  
Загрузили файл Install.tex  
Загрузили файл Random.tex  
Загрузили файл Addnew.tex
```

Было найдено 1 папок - разделов библиотеки

Рассматриваем папку Вектора (Одномерные массивы)  
Было найдено 15 файлов в папке

```
Загрузили файл FuncF.cpp  
Загрузили файл FuncF.desc  
Загрузили файл FuncF.h  
Загрузили файл FuncF.tex  
Загрузили файл FuncF.use  
Загрузили файл MHL_Func.cpp  
Загрузили файл MHL_Func.desc
```

```
Загрузили файл MHL_Func.h
Загрузили файл MHL_Func.tex
Загрузили файл MHL_Func.use
Загрузили файл TMHL_FillVector.desc
Загрузили файл TMHL_FillVector.h
Загрузили файл TMHL_FillVector.tex
Загрузили файл TMHL_FillVector.tpp
Загрузили файл TMHL_FillVector.use
Из 15 файлов нужными нам оказалось 15 файлов в папке

Загрузили файл Description_part2.tex
Загрузили файл Description_part1.tex
Загрузили файл Title.tex

Сохранили файл HarrixMathLibrary.cpp
Сохранили файл HarrixMathLibrary.h
Сохранили файл HarrixMathLibrary_Help.tex

Скопировали файл names.tex
Скопировали файл packages.tex
Скопировали файл styles.tex

Ошибки не были зафиксированы.
Конец формирования файлов библиотеки.
Потребовалось времени: 1 мин. 9 сек. 550 миллисек.
```

Если ошибок нет, то все прошло нормально.

- Также нам будет продемонстрирована папка **temp\_library** с сформированными файлами библиотеки.

Итак, мы собрали файлы библиотеки.

#### Шаг 4. Разберем файлы из папки **temp\_library**.

- Скопирем файлы **HarrixMathLibrary.cpp**, **HarrixMathLibrary.h**, **mtrand.cpp**, **mtrand.h** в папку **\_library**.
- Откройте файл **HarrixMathLibrary\_Help.tex** в L<sup>A</sup>T<sub>E</sub>X программе (автор использует TeXstudio) и скомпилируйте его.

В итоге в папке **temp\_library** появится файл **HarrixMathLibrary\_Help.pdf**. Скопируйте этот файл в папку **\_library**.

- Теперь разберемся с программой для демонстрации. Как мы помним, в ней в самом начале мы проверяли свою функцию.
  - Скопирем файлы **HarrixMathLibrary.cpp**, **HarrixMathLibrary.h**, **mtrand.cpp**, **mtrand.h** в папку **source\_demo**.
  - Откройте **DemoHarrixMathLibrary.pro** из папки **source\_demo** в Qt Creator и скомпилируйте приложение (в режиме Release).
  - Найдите папку, в которую скомпилировался проект. Это может быть папка проектов Qt, или папка появится в корневой папке библиотеки **HarrixMathLibrary**.
  - Скопируйте файл **DemoHarrixMathLibrary.exe** в папку **demo**.
  - В папке **demo** возможно придется обновить набор dll, если Вы используете версию Qt гораздо более новую, чем в данной сборке библиотеке.

- Удалим папку **temp\_library** после всех наших действий.
- Если папка с скомпилированным файлом DemoHarrixMathLibrary.exe появилась в корневой папке библиотеки, то удалите ее (например, build-DemoHarrixMathLibrary-Desktop\_Qt\_5\_1\_1\_MinGW\_32bit-Release).
- Отредактируйте на своё усмотрение файл CHANGELOG.md, где напишите о новых изменениях.
- В файлах **README.md**, **source\_library\Header.cpp**, **source\_library\Title.tex** поменяйте номер версии библиотеки.

Вот, вроде и всё. Мы добавили новую функцию и обновили все файлы и папки библиотеки.

## 5 Сторонние библиотеки, используемые в HarrixMathLibrary

В библиотеке HarrixMathLibrary в различных частях используются сторонние библиотеки (кроме того, что в Qt включены), о которых будет написано ниже.

- **mtrand** — генератор случайных чисел Mersenne Twister, который используется непосредственно в коде библиотеки. Авторы библиотеки: Takuji Nishimura, Makoto Matsumoto, Jasper Bedaux.

<http://www.bedaux.net/mtrand/>

- **jQuery** — библиотека для javascript. Используется в DemoHarrixMathLibrary.exe для отображения информации.

<http://jquery.com/>

- **JSXGraph** — библиотека для отображения графиков в html страницах. Используется в DemoHarrixMathLibrary.exe для отображения информации.

<https://github.com/jsxgraph/jsxgraph>

## 6 Список функций

### Вектора (Одномерные массивы)

1. **MHL\_DependentNoiseInVector** — Функция добавляет к элементам выборки помеху, зависящую от значения элемента выборки (плюс-минус сколько-то процентов модуля разности минимального и максимального элемента выборки, умноженного на значение элемента).
2. **MHL\_EuclidNorma** — Функция вычисляет евклидовую норму вектора.
3. **MHL\_NoiseInVector** — Функция добавляет к элементам выборки аддитивную помеху (плюс-минус сколько-то процентов модуля разности минимального и максимального элемента выборки).
4. **TMHL\_AcceptanceLimits** — Функция вмещает вектор VMHL\_ResultVector в прямоугольную многомерной области, определяемой левыми границами и правыми границами. Если какая-то координата вектора выходит за границу, то значение этой координаты принимает граничное значение.
5. **TMHL\_CheckElementInVector** — Функция проверяет наличие элемента a в векторе x.
6. **TMHL\_CompareMeanOfVectors** — Функция проверяет, какой вектор по среднему арифметическому больше.
7. **TMHL\_EqualityOfVectors** — Функция проверяет равенство векторов.
8. **TMHL\_FibonacciNumbersVector** — Функция заполняет массив числами Фибоначчи.
9. **TMHL\_FillVector** — Функция заполняет вектор значениями, равных x.
10. **TMHL\_MaximumOfVector** — Функция ищет максимальный элемент в векторе (одномерном массиве).
11. **TMHL\_MinimumOfVector** — Функция ищет минимальный элемент в векторе (одномерном массиве).
12. **TMHL\_MixingVector** — Функция перемешивает массив. Поочередно рассматриваютсѧ номера элементов массивов. С некоторой вероятностью рассматриваемый элемент массива меняется местами со случайным элементом массива.
13. **TMHL\_MixingVectorWithConjugateVector** — Функция перемешивает массив вместе со сопряженным массивом. Поочередно рассматриваются номера элементов массивов. С некоторой вероятностью рассматриваемый элемент массива меняется местами со случайным элементом массива. Пары элементов первого массива и сопряженного остаются без изменения.
14. **TMHL\_NumberOfDifferentValuesInVector** — Функция подсчитывает число различных значений в векторе (одномерном массиве).
15. **TMHL\_NumberOfMaximumOfVector** — Функция ищет номер максимального элемента в векторе (одномерном массиве).
16. **TMHL\_NumberOfMinimumOfVector** — Функция ищет номер минимального элемента в векторе (одномерном массиве).

17. **TMHL\_NumberOfNegativeValues** — Функция подсчитывает число отрицательных значений в векторе (одномерном массиве).
18. **TMHL\_NumberOfPositiveValues** — Функция подсчитывает число положительных значений в векторе (одномерном массиве).
19. **TMHL\_NumberOfZeroValues** — Функция подсчитывает число нулевых значений в векторе (одномерном массиве).
20. **TMHL\_OrdinalVector** — Функция заполняет вектор значениями, равные номеру элемента, начиная с единицы.
21. **TMHL\_OrdinalVectorZero** — Функция заполняет вектор значениями, равные номеру элемента, начиная с нуля.
22. **TMHL\_ProductOfElementsOfVector** — Функция вычисляет произведение элементов вектора.
23. **TMHL\_ReverseVector** — Функция меняет порядок элементов в массиве на обратный. Преобразуется подаваемый массив.
24. **TMHL\_SearchElementInVector** — Функция находит номер первого элемента в массиве, равного данному.
25. **TMHL\_SearchFirstNotZero** — Функция возвращает номер первого ненулевого элемента массива.
26. **TMHL\_SearchFirstZero** — Функция возвращает номер первого нулевого элемента массива.
27. **TMHL\_SumSquareVector** — Функция вычисляет сумму квадратов элементов вектора.
28. **TMHL\_SumVector** — Функция вычисляет сумму элементов вектора.
29. **TMHL\_VectorMinusVector** — Функция вычитает поэлементно из одного массива другой и записывает результат в третий массив. Или в переопределенном виде функция вычитает поэлементно из одного массива другой и записывает результат в первый массив.
30. **TMHL\_VectorMultiplyNumber** — Функция умножает вектор на число.
31. **TMHL\_VectorPlusVector** — Функция складывает поэлементно из одного массива другой и записывает результат в третий массив. Или в переопределенном виде функция складывает поэлементно из одного массива другой и записывает результат в первый массив.
32. **TMHL\_VectorToVector** — Функция копирует содержимое вектора (одномерного массива) в другой.
33. **TMHL\_ZeroVector** — Функция зануляет массив.

## Генетические алгоритмы

1. **MHL\_BinaryFitnessFunction** — Служебная функция. Функция вычисляет целевую функцию бинарного вектора, в котором закодирован вещественный вектор. Использует внутренние служебные переменные. Функция для MHL\_StandartRealGeneticAlgorithm. Использовать для своих целей не рекомендуется.

2. **MHL\_MakeVectorOfProbabilityForProportionalSelectionV2** — Функция формирует вектор вероятностей выбора индивидов из вектора значений функции пригодности. Формирование вектора происходит согласно правилам пропорционально селекции из ГА. Это служебная функция для использования функции пропорциональной селекции MHL\_ProportionalSelectionV2.
3. **MHL\_MakeVectorOfProbabilityForRankSelection** — Функция формирует вектор вероятностей выбора индивидов из вектора рангов для ранговой селекции. Это служебная функция для использования функции ранговой селекции MHL\_RankSelection.
4. **MHL\_MakeVectorOfRankForRankSelection** — Проставляет ранги для элементов не сортированного массива, то есть номера, начиная с 1, в отсортированном массиве. Если в массиве есть несколько одинаковых элементов, то ранги им присуждаются как среднеарифметические. Это служебная функция для функции MHL\_RankSelection.
5. **MHL\_MakeVectorOfRankZeroForRankSelection** — Проставляет ранги для элементов не сортированного массива, то есть номера, начиная с 0 (а не 1), в отсортированном массиве. Если в массиве есть несколько одинаковых элементов, то ранги им присуждаются как среднеарифметические.
6. **MHL\_NormalizationVectorAll** — Нормировка вектора чисел в отрезок  $[0; 1]$  посредством функции MHL\_NormalizationNumberAll.
7. **MHL\_NormalizationVectorMaxMin** — Нормировка вектора чисел так, чтобы максимальный элемент имел значение 1, а минимальный 0.
8. **MHL\_NormalizationVectorOne** — Нормировка вектора чисел в отрезок  $[0, 1]$  так, чтобы сумма всех элементов была равна 1.
9. **MHL\_ProbabilityOfTournamentSelection** — Функция вычисляет вероятности выбора индивидов из популяции с помощью турнирной селекции..
10. **MHL\_ProportionalSelection** — Пропорциональная селекция. Оператор генетического алгоритма. Работает с массивом пригодностей.
11. **MHL\_ProportionalSelectionV2** — Пропорциональная селекция. Оператор генетического алгоритма. Работает с вектором вероятностей выбора индивидов, который можно получить из вектора пригодностей индивидов посредством функции MHL\_MakeVectorOfProbabilityForProportionalSelectionV2.
12. **MHL\_ProportionalSelectionV3** — Пропорциональная селекция. Оператор генетического алгоритма. Работает с массивом пригодностей (обязательно не отрицательными).
13. **MHL\_RankSelection** — Ранговая селекция. Оператор генетического алгоритма. Работает с вектором вероятностей выбора индивидов, который можно получить из вектора пригодностей индивидов посредством функции MHL\_MakeVectorOfRankForRankSelection (для получения массива рангов) и потом функции MHL\_MakeVectorOfProbabilityForProportionalSelectionV2 (для получения массива вероятностей выбора индивидов по рангам).
14. **MHL\_SelectItemOnProbability** — Функция выбирает случайно номер элемента из вектора, где вероятность выбора каждого элемента определяется значением в векторе P.

15. **MHL\_StandartBinaryGeneticAlgorithm** — Стандартный генетический алгоритм для решения задач на бинарных строках. Реализация алгоритма из документа «Генетический алгоритм. Стандарт. v.3.0».
16. **MHL\_StandartGeneticAlgorithm** — Стандартный генетический алгоритм для решения задач на бинарных и вещественных строках. Реализация алгоритма из документа «Генетический алгоритм. Стандарт. v.3.0».
17. **MHL\_StandartRealGeneticAlgorithm** — Стандартный генетический алгоритм для решения задач на вещественных строках. Реализация алгоритма из документа «Генетический алгоритм. Стандарт. v.3.0».
18. **MHL\_TournamentSelection** — Турнирная селекция. Оператор генетического алгоритма. Работает с массивом пригодностей индивидов. В переопределенной функции используется во входных параметрах дополнительный массив, так как функция часто вызывается, а постоянно создавать массив накладно.
19. **MHL\_TournamentSelectionWithReturn** — Турнирная селекция с возвращением. Оператор генетического алгоритма. Работает с массивом пригодностей индивидов.
20. **TMHL\_MutationBinaryMatrix** — Мутация для бинарной матрицы. Оператор генетического алгоритма.
21. **TMHL\_SinglepointCrossover** — Одноточечное скрещивание. Оператор генетического алгоритма.
22. **TMHL\_SinglepointCrossoverWithCopying** — Одноточечное скрещивание с возможностью полного копирования одного из родителей. Оператор генетического алгоритма. Отличается от стандартного одноточечного скрещивания тем, что точки разрыва могут происходить по краям родителей, что может привести к полному копированию родителя.
23. **TMHL\_TwoPointCrossover** — Двухточечное скрещивание. Оператор генетического алгоритма.
24. **TMHL\_TwoPointCrossoverWithCopying** — Двухточечное скрещивание с возможностью полного копирования одного из родителей. Оператор генетического алгоритма. Отличается от стандартного двухточечного скрещивания тем, что точки разрыва могут происходить по краям родителей, что может привести к полному копированию родителя.
25. **TMHL\_UniformCrossover** — Равномерное скрещивание. Оператор генетического алгоритма.

## Геометрия

1. **MHL\_LineGeneralForm** — Функция представляет собой уравнение прямой по общему уравнению прямой вида  $Ax + By + C = 0$ .
2. **MHL\_LineSlopeInterceptForm** — Функция представляет собой уравнение прямой с угловым коэффициентом вида  $y = kx + b$ .
3. **MHL\_LineTwoPoint** — Функция представляет собой уравнение прямой по двум точкам.

4. **MHL\_Parabola** — Функция представляет собой уравнение параболы вида:  $y = ax^2 + bx + c$ .
5. **TMHL\_BoolCrossingTwoSegment** — Функция определяет наличие пересечения двух отрезков. Координаты отрезков могут быть перепутаны по порядку в каждом отрезке.

## Гиперболические функции

1. **MHL\_Cosech** — Функция возвращает гиперболический косеканс.
2. **MHL\_Cosh** — Функция возвращает гиперболический косинус.
3. **MHL\_Cotanh** — Функция возвращает гиперболический котангент.
4. **MHL\_Sech** — Функция возвращает гиперболический секанс.
5. **MHL\_Sinh** — Функция возвращает гиперболический синус.
6. **MHL\_Tanh** — Функция возвращает гиперболический тангенс.

## Дифференцирование

1. **MHL\_CenterDerivative** — Численное значение производной в точке (центральной разностной производной с шагом  $2h$ ).
2. **MHL\_LeftDerivative** — Численное значение производной в точке (разностная производная влево).
3. **MHL\_RightDerivative** — Численное значение производной в точке (разностная производная вправо).

## Для тестовых функций

1. **MHL\_ClassOfTestFunction** — Функция выдает принадлежность тестовой функции к классу функций: бинарной, вещественной или иной оптимизации.
2. **MHL\_CountOfFitnessOfTestFunction\_Binary** — Функция определяет количество вычислений целевой функции для тестовых задач для единообразного сравнения алгоритмов. Включает в себя все тестовые функции вещественной оптимизации.
3. **MHL\_CountOfFitnessOfTestFunction\_Real** — Функция определяет количество вычислений целевой функции для тестовых задач для единообразного сравнения алгоритмов. Включает в себя все тестовые функции вещественной оптимизации.
4. **MHL\_DefineTestFunction** — Служебная функция определяет тестовую функцию для других функций по работе с тестовыми функциями.
5. **MHL\_DimensionTestFunction\_Binary** — Функция определяет размерность тестовой задачи для тестовой функции бинарной оптимизации по номеру подзадачи (число подзадач по функции `MHL_GetCountOfSubProblems_Binary`).
6. **MHL\_DimensionTestFunction\_Real** — Функция определяет размерность тестовой задачи для тестовой функции вещественной оптимизации по номеру подзадачи (число подзадач по функции `MHL_GetCountOfSubProblems_Binary`).

7. **MHL\_ErrorExOfTestFunction\_Binary** — Функция определяет значение ошибки по входным параметрам найденного решения в задаче оптимизации для тестовой функции. Включает в себя все тестовые функции бинарной оптимизации. Есть функция-переопределение, где пользователь может сам указать тип тестовой функции.
8. **MHL\_ErrorExOfTestFunction\_Real** — Функция определяет значение ошибки по входным параметрам найденного решения в задаче оптимизации для тестовой функции вещественной оптимизации. Включает в себя все тестовые функции вещественной оптимизации. Есть функция-переопределение, где пользователь может сам указать тип тестовой функции.
9. **MHL\_ErrorEyOfTestFunction\_Binary** — Функция определяет значение ошибки по значениям целевой функции найденного решения в задаче оптимизации для тестовой функции. Включает в себя все тестовые функции бинарной оптимизации. Есть функция-переопределение, где пользователь может сам указать тип тестовой функции.
10. **MHL\_ErrorEyOfTestFunction\_Real** — Функция определяет значение ошибки по значениям целевой функции найденного решения в задаче оптимизации для тестовой функции вещественной оптимизации. Включает в себя все тестовые функции вещественной оптимизации. Есть функция-переопределение, где пользователь может сам указать тип тестовой функции.
11. **MHL\_ErrorROfTestFunction\_Binary** — Функция определяет значение надежности найденного решения в задаче оптимизации для тестовой функции. Включает в себя все тестовые функции бинарной оптимизации. Есть функция-переопределение, где пользователь может сам указать тип тестовой функции.
12. **MHL\_ErrorROfTestFunction\_Real** — Функция определяет значение надежности найденного решения в задаче оптимизации для тестовой функции вещественной оптимизации. Включает в себя все тестовые функции вещественной оптимизации. Есть функция-переопределение, где пользователь может сам указать тип тестовой функции.
13. **MHL\_FitnessOfOptimumOfTestFunction\_Binary** — Функция определяет значение целевой функции в оптимуме для тестовой функции бинарной оптимизации. Включает в себя все тестовые функции бинарной оптимизации. Есть функция-переопределение, где пользователь может сам указать тип тестовой функции.
14. **MHL\_FitnessOfOptimumOfTestFunction\_Real** — Функция определяет значение целевой функции в оптимуме для тестовой функции вещественной оптимизации. Включает в себя все тестовые функции вещественной оптимизации. Есть функция-переопределение, где пользователь может сам указать тип тестовой функции.
15. **MHL\_GetCountOfFitness** — Функция выдает количество вызовов целевой функции.
16. **MHL\_GetCountOfSubProblems\_Binary** — Функция определяет число подзадач (включая основную задачу) для тестовой функции бинарной оптимизации. Включает в себя все тестовые функции бинарной оптимизации.
17. **MHL\_GetCountOfSubProblems\_Real** — Функция определяет число подзадач (включая основную задачу) для тестовой функции вещественной оптимизации. Включает в себя все тестовые функции вещественной оптимизации.

18. **MHL\_LeftBorderOfTestFunction\_Real** — Функция определяет левые и правые границы допустимой области для тестовой функции вещественной оптимизации. Включает в себя все тестовые функции вещественной оптимизации. Есть функция-переопределение, где пользователь может сам указать тип тестовой функции.
19. **MHL\_MaximumOrMinimumOfTestFunction\_Binary** — Функция сообщает - ищется максимум или минимум в задаче оптимизации для тестовой функции бинарной оптимизации.
20. **MHL\_MaximumOrMinimumOfTestFunction\_Real** — Функция сообщает - ищется максимум или минимум в задаче оптимизации для тестовой функции вещественной оптимизации.
21. **MHL\_NumberOfPartsOfTestFunction\_Real** — Функция определяет на сколько частей нужно делить каждую координату в задаче оптимизации для тестовой функции вещественной оптимизации для алгоритма дискретной оптимизации и какая при этом требуется точность для подсчета надежности. Включает в себя все тестовые функции вещественной оптимизации. Есть функция-переопределение, где пользователь может сам указать тип тестовой функции.
22. **MHL\_OptimumOfTestFunction\_Binary** — Функция определяет значение оптимума для тестовой функции. Включает в себя все тестовые функции бинарной оптимизации. Есть функция-переопределение, где пользователь может сам указать тип тестовой функции.
23. **MHL\_OptimumOfTestFunction\_Real** — Функция определяет значение оптимума для тестовой функции вещественной оптимизации. Включает в себя все тестовые функции вещественной оптимизации. Есть функция-переопределение, где пользователь может сам указать тип тестовой функции.
24. **MHL\_PrecisionOfCalculationsOfTestFunction\_Real** — Функция определяет точность для подсчета надежности в задаче оптимизации для тестовой функции вещественной оптимизации для алгоритма дискретной оптимизации.
25. **MHL\_SetToZeroCountOfFitness** — Функция обнуляет количество вызовов целевой функции. Обязательно вызвать один раз перед вызовом алгоритмов оптимизации при исследовании эффективности алгоритмов оптимизации, где требуется контроль числа вызовов целевой функции.
26. **MHL\_TestFunction\_Binary** — Общая тестовая функция для задач бинарной оптимизации. Есть функция-переопределение, где пользователь может сам указать тип тестовой функции.
27. **MHL\_TestFunction\_Real** — Общая тестовая функция для задач вещественной оптимизации. Есть функция-переопределение, где пользователь может сам указать тип тестовой функции.

## Интегрирование

1. **MHL\_IntegralOfRectangle** — Интегрирование по формуле прямоугольников с оценкой точности по правилу Рунге. Считается интеграл функции на отрезке  $[a,b]$  с погрешностью порядка Epsilon.

2. **MHL\_IntegralOfSimpson** — Интегрирование по формуле Симпсона с оценкой точности по правилу Рунге. Считается интеграл функции на отрезке  $[a,b]$  с погрешностью порядка Epsilon.
3. **MHL\_IntegralOfTrapezium** — Интегрирование по формуле трапеции с оценкой точности по правилу Рунге. Считается интеграл функции на отрезке  $[a,b]$  с погрешностью порядка Epsilon.

### Кодирование и декодирование

1. **MHL\_BinaryGrayVectorToRealVector** — Функция декодирует бинарную строку в действительный вектор, который и был закодирован методом «Стандартный рефлексивный Грей-код» (без использования временного массива). Перегруженная функция делает тоже самое, но с использованием временного массива (это позволяет не создавать каждый раз временный массив, что ускоряет работу).
2. **MHL\_BinaryVectorToRealVector** — Функция декодирует бинарную строку в действительный вектор, который и был закодирован методом «Стандартное представление целого числа — номер узла в сетке дискретизации».
3. **TMHL\_BinaryToDecimal** — Функция декодирует двоичное число в десятичное целое неотрицательное.
4. **TMHL\_BinaryToDecimalFromPart** — Функция декодирует двоичное число в десятичное целое неотрицательное. При этом двоичное число длины берется как часть некой бинарной строки а.
5. **TMHL\_GrayCodeToBinary** — Функция переводит бинарный код Грея в бинарный код.
6. **TMHL\_GrayCodeToBinaryFromPart** — Функция переводит бинарный код Грея в бинарный код. При этом бинарный код Грея берется как часть некой строки а, заполненной 0 и 1.

### Комбинаторика

1. **TMHL\_KCombinations** — Функция возвращает число сочетаний из  $n$  по  $m$  (без возвращения).

### Математические функции

1. **MHL\_ArithmeticalProgression** — Арифметическая прогрессия.  $n$ -ый член последовательности.
2. **MHL\_ExpMSxD2** — Функция вычисляет выражение  $\exp(-x * x/2)$ .
3. **MHL\_GeometricSeries** — Геометрическая прогрессия.  $n$ -ый член последовательности.
4. **MHL\_GreatestCommonDivisorEuclid** — Функция находит наибольший общий делитель двух чисел по алгоритму Евклида.
5. **MHL\_HowManyPowersOfTwo** — Функция вычисляет, какой минимальной степенью двойки можно покрыть целое положительное число.
6. **MHL\_InverseNormalizationNumberAll** — Функция осуществляет обратную нормировку числа из интервала  $[0; 1]$  в интервал  $[-\infty; \infty]$ , которое было осуществлено функцией **MHL\_NormalizationNumberAll**.

7. **MHL\_LeastCommonMultipleEuclid** — Функция находит наименьшее общее кратное двух чисел по алгоритму Евклида.
8. **MHL\_MixedMultiLogicVectorOfFullSearch** — Функция генерирует определенный вектор k-значной логики, где каждый элемент может принимать разное максимальное значение, в полном переборе вариантов. Генерируется I вектор в этом полном переборе.
9. **MHL\_NormalizationNumberAll** — Функция нормирует число из интервала  $[-\infty; \infty]$  в интервал  $[0; 1]$ . При этом в нуле возвращает 0.5, в  $-\infty$  возвращает 0, в  $\infty$  возвращает 1. Если  $x < y$ , то  $MHL_NormalizationNumberAll(x) < MHL_NormalizationNumberAll(y)$ . Под бесконечностью принимается машинная бесконечность.
10. **MHL\_Parity** — Функция проверяет четность целого числа.
11. **MHL\_ProbabilityDensityFunctionOfInverseGaussianDistribution** — Функция вычисляет плотность вероятности распределения обратного гауссовскому распределению..
12. **MHL\_SumGeometricSeries** — Геометрическая прогрессия. Сумма первых n членов.
13. **MHL\_SumOfArithmeticalProgression** — Арифметическая прогрессия. Сумма первых n членов.
14. **MHL\_SumOfDigits** — Функция подсчитывает сумму цифр любого целого числа.
15. **TMHL\_Abs** — Функция возвращает модуль числа.
16. **TMHL\_Factorial** — Функция вычисляет факториал числа.
17. **TMHL\_FibonacciNumber** — Функция вычисляет число Фибоначчи, заданного номера.
18. **TMHL\_HeavisideFunction** — Функция Хевисайда (функция одной переменной).
19. **TMHL\_Max** — Функция возвращает максимальный элемент из двух.
20. **TMHL\_Min** — Функция возвращает минимальный элемент из двух.
21. **TMHL\_NumberInterchange** — Функция меняет местами значения двух чисел.
22. **TMHL\_PowerOf** — Функция возводит произвольное число в целую степень.
23. **TMHL\_Sign** — Функция вычисляет знака числа.
24. **TMHL\_SignNull** — Функция вычисляет знака числа. При нуле возвращает 1.

## Матрицы

1. **TMHL\_CheckForIdenticalColsInMatrix** — Функция проверяет наличие одинаковых столбцов в матрице.
2. **TMHL\_CheckForIdenticalRowsInMatrix** — Функция проверяет наличие одинаковых строк в матрице.
3. **TMHL\_ColInterchange** — Функция переставляет столбцы матрицы.
4. **TMHL\_ColToMatrix** — Функция копирует в матрицу (двумерный массив) из вектора столбец.

5. **TMHL\_DeleteColInMatrix** — Функция удаляет k столбец из матрицы (начиная с нуля). Все правостоящие столбцы сдвигаются влево на единицу. Последний столбец зануляется.
6. **TMHL\_DeleteRowInMatrix** — Функция удаляет k строку из матрицы (начиная с нуля). Все нижестоящие строки поднимаются на единицу. Последняя строка зануляется.
7. **TMHL\_EqualityOfMatrixes** — Функция проверяет равенство матриц.
8. **TMHL\_FillMatrix** — Функция заполняет матрицу значениями, равных x.
9. **TMHL\_IdentityMatrix** — Функция формирует единичную квадратную матрицу.
10. **TMHL\_MatrixMinusMatrix** — Функция вычитает две матрицы. Или для переопределенной варианта функция вычитает два матрицы и результат записывает в первую матрицу.
11. **TMHL\_MatrixMultiplyMatrix** — Функция перемножает матрицы.
12. **TMHL\_MatrixMultiplyMatrixT** — Функция умножает матрицу на транспонированную матрицу.
13. **TMHL\_MatrixMultiplyNumber** — Функция умножает матрицу на число.
14. **TMHL\_MatrixPlusMatrix** — Функция суммирует две матрицы. Или для переопределенной варианта функция суммирует два матрицы и результат записывает в первую матрицу.
15. **TMHL\_MatrixT** — Функция транспонирует матрицу.
16. **TMHL\_MatrixTMultiplyMatrix** — Функция умножает транспонированную матрицу на матрицу.
17. **TMHL\_MatrixToCol** — Функция копирует из матрицы (двумерного массива) в вектор столбец.
18. **TMHL\_MatrixToMatrix** — Функция копирует содержимое матрицы (двумерного массива) а в массив VMHL\_ResultMatrix.
19. **TMHL\_MatrixToRow** — Функция копирует из матрицы (двумерного массива) в вектор строку.
20. **TMHL\_MaximumOfMatrix** — Функция ищет максимальный элемент в матрице (двумерном массиве).
21. **TMHL\_MinimumOfMatrix** — Функция ищет минимальный элемент в матрице (двумерном массиве).
22. **TMHL\_MixingRowsInOrder** — Функция меняет строки матрицы в порядке, указанным в массиве b.
23. **TMHL\_NumberOfDifferentValuesInMatrix** — Функция подсчитывает число различных значений в матрице.
24. **TMHL\_RowInterchange** — Функция переставляет строки матрицы.

25. **TMHL\_RowToMatrix** — Функция копирует в матрицу (двумерный массив) из вектора строку.
26. **TMHL\_SumMatrix** — Функция вычисляет сумму элементов матрицы.
27. **TMHL\_ZeroMatrix** — Функция зануляет матрицу.

## Метрика

1. **TMHL\_Chebychev** — Функция вычисляет расстояние Чебышева.
2. **TMHL\_CityBlock** — Функция вычисляет манхэттенское расстояние между двумя массивами.
3. **TMHL\_Euclid** — Функция вычисляет евклидово расстояние.
4. **TMHL\_Minkovski** — Функция вычисляет метрику Минковского.

## Непараметрика

1. **MHL\_BellShapedKernelExp** — Колоколообразное экспоненциальное ядро.
2. **MHL\_BellShapedKernelParabola** — Колоколообразное параболическое ядро.
3. **MHL\_BellShapedKernelRectangle** — Колоколообразное прямоугольное ядро.
4. **MHL\_BellShapedKernelTriangle** — Колоколообразное треугольное ядро.
5. **MHL\_DerivativeOfBellShapedKernelExp** — Производная колоколообразного экспоненциального ядра.
6. **MHL\_DerivativeOfBellShapedKernelParabola** — Производная колоколообразного параболического ядра.
7. **MHL\_DerivativeOfBellShapedKernelRectangle** — Производная колоколообразного прямоугольного ядра.
8. **MHL\_DerivativeOfBellShapedKernelTriangle** — Производная колоколообразного треугольного ядра.

## Нечеткие системы

1. **MHL\_TrapeziformFuzzyNumber** — Трапециевидное нечеткое число. Точнее его функция принадлежности.

## Оптимизация

1. **MHL\_BinaryMonteCarloAlgorithm** — Метод Монте-Карло (Monte-Carlo). Простейший метод оптимизации на бинарных строках. В простонародье его называют «методом научного тыка». Алгоритм оптимизации. Ищет максимум функции пригодности FitnessFunction.
2. **MHL\_DichotomyOptimization** — Метод дихотомии. Метод одномерной оптимизации унимодальной функции на интервале. Ищет минимум.
3. **MHL\_FibonacciOptimization** — Метод Фибоначчи. Метод одномерной оптимизации унимодальной функции на интервале. Ищет минимум.

4. **MHL\_GoldenSectionOptimization** — Метод золотого сечения. Метод одномерной оптимизации унимодальной функции на интервале. Ищет минимум.
5. **MHL\_QuadraticFitOptimization** — Метод квадратичной интерполяции. Метод одномерной оптимизации унимодальной функции на интервале. Ищет минимум.
6. **MHL\_RealMonteCarloAlgorithm** — Метод Монте-Карло (Monte-Carlo). Простейший метод оптимизации на вещественных строках. В простонародье его называют "методом научного тыка". Алгоритм оптимизации. Ищет максимум функции пригодности FitnessFunction.
7. **MHL\_RealMonteCarloOptimization** — Метод Монте-Карло (Monte-Carlo). Простейший метод оптимизации на вещественных строках. Ищет минимум. От функции MHL\_RealMonteCarloAlgorithm отличается тем, что ищет минимум, а не максимум, и не у многомерной функции, а одномерной. Вводится, чтобы было продолжением однотипных методов оптимизации одномерных унимодальных функций.
8. **MHL\_UniformSearchOptimization** — Метод равномерного поиска. Метод одномерной оптимизации функции на интервале. Ищет минимум.
9. **MHL\_UniformSearchOptimizationN** — Метод равномерного поиска. Метод одномерной оптимизации функции на интервале. Ищет минимум. От MHL\_UniformSearchOptimization отличается тем, что вместо параметра шага равномерного прохода используется число вычислений целевой функции, но они взаимозаменяемы.

## Оптимизация - свалка алгоритмов

1. **MHL\_BinaryGeneticAlgorithmWCC** — Генетический алгоритм для решения задач на бинарных строках, в котором есть только два вида скрещивания: одноточечное и двухточечное скрещивание с возможностью полного копирования одного из родителей. Равномерное скрещивание отсутствует.
2. **MHL\_BinaryGeneticAlgorithmWDPOfNOfGPS** — Генетический алгоритм для решения задач на бинарных строках с изменяющимся соотношением числа поколений и размера популяции. Отличается от стандартного генетического алгоритма, тем, что размер популяции и число поколений рассчитывается из числа вычислений целевой функции не как одинаковые величины (извлечение квадратного корня), а через некоторое соотношение.
3. **MHL\_BinaryGeneticAlgorithmWDTS** — Генетический алгоритм для решения задач на бинарных строках с турнирной селекцией, где размер турнира изменяется от 2 до размера популяции. Отличается от стандартного генетического алгоритма, тем, что присутствует только турнирная селекция, но размер турнира может изменяться.
4. **MHL\_RealGeneticAlgorithmWCC** — Генетический алгоритм для решения задач на вещественных строках, в котором есть только два вида скрещивания: одноточечное и двухточечное скрещивание с возможностью полного копирования одного из родителей. Равномерное скрещивание отсутствует.
5. **MHL\_RealGeneticAlgorithmWDPOfNOfGPS** — Генетический алгоритм для решения задач на вещественных строках с изменяющимся соотношением числа поколений и размера популяции. Отличается от стандартного генетического алгоритма, тем, что размер популяции и число поколений рассчитывается из числа вычислений целевой функции

не как одинаковые величины (извлечение квадратного корня), а через некоторое соотношение.

6. **MHL\_RealGeneticAlgorithmWDTs** — Генетический алгоритм для решения задач на вещественных строках с турнирной селекцией, где размер турнира изменяется от 2 до размера популяции. Отличается от стандартного генетического алгоритма, тем, что присутствует только турнирная селекция, но размер турнира может изменяться.

## Перевод единиц измерений

1. **MHL\_DegToRad** — Функция переводит угол из градусной меры в радианную.
2. **MHL\_RadToDeg** — Функция переводит угол из радианной меры в градусную.

## Случайные объекты

1. **MHL\_BitNumber** — Функция с вероятностью P (или 0.5 в переопределенной функции) возвращает 1. В противном случае возвращает 0.
2. **MHL\_RandomRealMatrix** — Функция заполняет матрицу случайными вещественными числами из определенного интервала [Left;Right].
3. **MHL\_RandomRealMatrixInCols** — Функция заполняет матрицу случайными вещественными числами из определенного интервала. При этом элементы каждого столбца изменяются в своих пределах.
4. **MHL\_RandomRealMatrixInElements** — Функция заполняет матрицу случайными вещественными числами из определенного интервала. При этом каждый элемент изменяется в своих пределах.
5. **MHL\_RandomRealMatrixInRows** — Функция заполняет матрицу случайными вещественными числами из определенного интервала. При этом элементы каждой строки изменяются в своих пределах.
6. **MHL\_RandomRealVector** — Функция заполняет массив случайными вещественными числами из определенного интервала [Left;Right].
7. **MHL\_RandomRealVectorInElements** — Функция заполняет массив случайными вещественными числами из определенного интервала, где на каждую координату свои границы изменения.
8. **MHL\_RandomVectorOfProbability** — Функция заполняет вектор случайными значениями вероятностей. Сумма всех элементов вектора равна 1.
9. **TMHL\_BernulliVector** — Функция формирует случайный вектор Бернулли.
10. **TMHL\_RandomArrangingObjectsIntoBaskets** — Функция предлагает случайный способ расставить N объектов в VMHL\_N корзин при условии, что в каждой корзине может располагаться только один предмет.
11. **TMHL\_RandomBinaryMatrix** — Функция заполняет матрицу случайно нулями и единицами.
12. **TMHL\_RandomBinaryVector** — Функция заполняет вектор (одномерный массив) случайно нулями и единицами.

13. **TMHL\_RandomIntMatrix** — Функция заполняет матрицу случайными целыми числами из определенного интервала [n;m).
14. **TMHL\_RandomIntMatrixInCols** — Функция заполняет матрицу случайными целыми числами из определенного интервала [n;m). При этом элементы каждого столбца изменяются в своих пределах.
15. **TMHL\_RandomIntMatrixInElements** — Функция заполняет матрицу случайными целыми числами из определенного интервала [n;m). При этом каждый элемент изменяется в своих пределах.
16. **TMHL\_RandomIntMatrixInRows** — Функция заполняет матрицу случайными целыми числами из определенного интервала [n;m). При этом элементы каждой строки изменяются в своих пределах.
17. **TMHL\_RandomIntVector** — Функция заполняет массив случайными целыми числами из определенного интервала [n,m).
18. **TMHL\_RandomIntVectorInElements** — Функция заполняет массив случайными целыми числами из определенного интервала [n\_i,m\_i). При этом для каждого элемента массива свой интервал изменения.
19. **TMHL\_RandomMatrixOfPermutation** — Функция создает случайный массив строк-перестановок чисел от 1 до VMHL\_M.
20. **TMHL\_RandomVectorOfPermutation** — Функция создает случайную строку-перестановку чисел от 1 до VMHL\_N (включительно).

## Случайные числа

1. **MHL\_RandomNormal** — Случайное число по нормальному закону распределения.
2. **MHL\_RandomUniform** — Случайное вещественное число в интервале [a;b] по равномерному закону распределения.
3. **MHL\_RandomUniformInt** — Случайное целое число в интервале [n,m) по равномерному закону распределения.
4. **MHL\_RandomUniformIntIncluding** — Случайное целое число в интервале [n,m] по равномерному закону распределения.

## Сортировка

1. **TMHL\_BubbleDescendingSort** — Функция сортирует массив в порядке убывания методом «Сортировка пузырьком».
2. **TMHL\_BubbleSort** — Функция сортирует массив в порядке возрастания методом «Сортировка пузырьком».
3. **TMHL\_BubbleSortColWithOtherConjugateColsInMatrix** — Функция сортирует матрицу по какому-то столбцу под номером в порядке возрастания методом «Сортировка пузырьком». При этом все остальные столбцы являются как бы сопряженным с данным столбцом. То есть элементы в этом столбце сортируются, а все строки остаются прежними.
4. **TMHL\_BubbleSortEveryColInMatrix** — Функция сортирует каждый столбец матрицы в отдельности.

5. **TMHL\_BubbleSortEveryRowInMatrix** — Функция сортирует каждую строку матрицы в отдельности.
6. **TMHL\_BubbleSortInGroups** — Функция сортирует массив в порядке возрастания методом «Сортировка пузырьком» в группах данного массива. Имеется массив. Он делится на группы элементов по  $m$  элементов. Первые  $m$  элементов принадлежат первой группе, следующие  $m$  элементов — следующей и т.д. (Разумеется, в последней группе может и не оказаться  $m$  элементов). Потом в каждой группе элементы сортируются по возрастанию.
7. **TMHL\_BubbleSortRowWithOtherConjugateRowsInMatrix** — Функция сортирует матрицу по какой-то строке под номером в порядке возрастания методом «Сортировка пузырьком». При этом все остальные строки являются как бы сопряженными с данной строкой. То есть элементы в этой строке сортируются, а все столбцы остаются прежними.
8. **TMHL\_BubbleSortWithConjugateVector** — Функция сортирует массив вместе с сопряженным массивом в порядке возрастания методом «Сортировка пузырьком». Пары элементов первого массива и сопряженного остаются без изменения.
9. **TMHL\_BubbleSortWithTwoConjugateVectors** — Функция сортирует массив вместе с двумя сопряженными массивами в порядке возрастания методом «Сортировка пузырьком». Пары элементов первого массива и сопряженного остаются без изменения.

## Статистика и теория вероятности

1. **MHL\_DensityOfDistributionOfNormalizedCenteredNormalDistribution** — Плотность распределения вероятности нормированного и центрированного нормального распределения.
2. **MHL\_DistributionFunctionOfNormalDistribution** — Функция распределения нормального распределения.
3. **MHL\_DistributionFunctionOfNormalizedCenteredNormalDistribution** — Функция распределения нормированного и центрированного нормального распределения.
4. **MHL\_LeftBorderOfWilcoxonWFromTable** — Функция возвращает левую границу интервала критический значений статистики  $W$  для критерия Вилкоксена по табличным данным.
5. **MHL\_RightBorderOfWilcoxonWFromTable** — Функция возвращает правую границу интервала критический значений статистики  $W$  для критерия Вилкоксена по табличным данным.
6. **MHL\_StdDevToVariance** — Функция переводит среднеквадратичное уклонение в значение дисперсии случайной величины.
7. **MHL\_VarianceToStdDev** — Функция переводит значение дисперсии случайной величины в среднеквадратичное уклонение.
8. **MHL\_WilcoxonW** — Функция проверяет однородность выборок по критерию Вилкосена  $W$ .
9. **TMHL\_Mean** — Функция вычисляет среднее арифметическое массива.

10. **TMHL\_Median** — Функция вычисляет медиану выборки.
11. **TMHL\_SampleCovariance** — Функция вычисляет выборочную ковариацию выборки (несмещенная, исправленная).
12. **TMHL\_Variance** — Функция вычисляет выборочную дисперсию выборки (несмещенная, исправленная).

## Тестовые функции для оптимизации

1. **MHL\_TestFunction\_Ackley** — Функция многих переменных: Ackley. Тестовая функция вещественной оптимизации.
2. **MHL\_TestFunction\_AdditivePotential** — Функция двух переменных: аддитивная потенциальная функция. Тестовая функция вещественной оптимизации.
3. **MHL\_TestFunction\_MultiplicativePotential** — Функция двух переменных: мультиплексивная потенциальная функция. Тестовая функция вещественной оптимизации.
4. **MHL\_TestFunction\_ParaboloidOfRevolution** — Функция многих переменных: Эллиптический параболоид. Тестовая функция вещественной оптимизации.
5. **MHL\_TestFunction\_Rastrigin** — Функция многих переменных: функция Растригина. Тестовая функция вещественной оптимизации.
6. **MHL\_TestFunction\_Rosenbrock** — Функция многих переменных: функция Розенброка. Тестовая функция вещественной оптимизации.
7. **MHL\_TestFunction\_SumVector** — Сумма всех элементов бинарного вектора. Тестовая функция бинарной оптимизации.

## Тригонометрические функции

1. **MHL\_Cos** — Функция возвращает косинус угла в радианах.
2. **MHL\_CosDeg** — Функция возвращает косинус угла в градусах.
3. **MHL\_Cosec** — Функция возвращает косеканс угла в радианах.
4. **MHL\_CosecDeg** — Функция возвращает косеканс угла в градусах.
5. **MHL\_Cotan** — Функция возвращает котангенс угла в радианах.
6. **MHL\_CotanDeg** — Функция возвращает котангенс угла в градусах.
7. **MHL\_Sec** — Функция возвращает секанс угла в радианах.
8. **MHL\_SecDeg** — Функция возвращает секанс угла в градусах.
9. **MHL\_Sin** — Функция возвращает синус угла в радианах.
10. **MHL\_SinDeg** — Функция возвращает синус угла в градусах.
11. **MHL\_Tan** — Функция возвращает тангенс угла в радианах.
12. **MHL\_TanDeg** — Функция возвращает тангенс угла в градусах.

## Уравнения

1. **MHL\_QuadraticEquation** — Функция решает квадратное уравнение вида:  $a \cdot x^2 + b \cdot x + c = 0$ . Ответ представляет собой два действительных числа.

## 7 Функции

### 7.1 Вектора (Одномерные массивы)

#### 7.1.1 MHL\_DependentNoiseInVector

Функция добавляет к элементам выборки помеху, зависящую от значения элемента выборки (плюс-минус сколько-то процентов модуля разности минимального и максимального элемента выборки, умноженного на значение элемента).

Код 58. Синтаксис

```
void MHL_DependentNoiseInVector(double *VMHL_ResultVector, double percent, int VMHL_N );
```

#### Входные параметры:

VMHL\_ResultVector — указатель на массив;

percent — процент шума;

VMHL\_N — количество элементов в массивах.

#### Возвращаемое значение:

Отсутствует.

#### Формула:

$$b = \frac{percent \cdot (\max x_i - \min x_i)}{100};$$
$$x_i = x_i + x_i \cdot \text{random} \left( -\frac{b}{2}, \frac{b}{2} \right),$$

где  $x_i \in VMHL\_ResultVector$ ,  $i = \overline{1, VMHL\_N}$ .

Код 59. Пример использования

```
int VMHL_N=10; //Размер массива
double *x;
x=new double[VMHL_N];
//Заполним массив номерами от 1
TMHL_OrdinalVector(x,VMHL_N);
MHL_ShowVector (x,VMHL_N, "Вектор", "x");
//Вектор:
//x =
//1
//2
//3
//4
//5
//6
//7
//8
//9
//10
```

```

double percent=double(MHL_RandomUniformInt(0,100)); //Процент помехи

//Вызов функции
MHL_DependentNoiseInVector(x,percent,VMHL_N);

//Используем полученный результат

MHL_ShowNumber (percent,"Процент помехи", "percent");
//Процент помехи:
//percent=6
MHL_ShowVector (x,VMHL_N,"Вектор с зависимой помехой", "x");
//Вектор с помехой:
//Вектор с помехой:
//x =
//0.865099
//2.50058
//2.43314
//4.38595
//3.98511
//5.36837
//8.42834
//7.18024
//9.33134
//10.5783

delete [] x;

```

### 7.1.2 MHL\_EuclidNorma

Функция вычисляет евклидовую норму вектора.

Код 60. Синтаксис

```
double MHL_EuclidNorma(double *a,int VMHL_N);
```

**Входные параметры:**

a — указатель на вектор;

VMHL\_N — размер массива.

**Возвращаемое значение:**

Значение евклидовой нормы вектора.

**Формула:**

$$EuclidNormaVector = \sqrt{\sum_{i=1}^n (a_i)^2}.$$

Код 61. Пример использования

```
int VMHL_N=5; //Размер массива
double *x;
x=new double[VMHL_N];
//Заполним случайными числами
```

```

MHL_RandomRealVector (x, 0, 10, VMHL_N);

//Вызов функции
double a=MHL_EuclidNorma(x,VMHL_N);

//Используем полученный результат
MHL_ShowVector (x,VMHL_N, "Вектор", "x");
// Вектор:
//x =
//2.22504
//5.2655
//5.00092
//5.7428
//9.11682

MHL_ShowNumber (a, "Значение евклидовой нормы вектора", "a");
// Значение евклидовой нормы вектора:
// a=13.1826

delete [] x;

```

### 7.1.3 MHL\_NoiseInVector

Функция добавляет к элементам выборки аддитивную помеху (плюс-минус сколько-то процентов модуля разности минимального и максимального элемента выборки).

Код 62. Синтаксис

```
void MHL_NoiseInVector(double *VMHL_ResultVector, double percent, int VMHL_N);
```

**Входные параметры:**

VMHL\_ResultVector — указатель на массив;

percent — процент шума;

VMHL\_N — количество элементов в массивах.

**Возвращаемое значение:**

Отсутствует.

**Формула:**

$$b = \frac{\text{percent} \cdot (\max x_i - \min x_i)}{100},$$

$$x_i = x_i + \text{random} \left( -\frac{b}{2}, \frac{b}{2} \right),$$

где  $x_i \in VMHL\_ResultVector$ ,  $i = \overline{1, VMHL\_N}$ .

Код 63. Пример использования

```

int VMHL_N=10; //Размер массива
double *x;
x=new double[VMHL_N];
//Заполним массив номерами от 1

```

```

TMHL_OrdinalVector(x,VMHL_N);
MHL_ShowVector (x,VMHL_N,"Вектор", "x");
//Вектор:
//x =
//1
//2
//3
//4
//5
//6
//7
//8
//9
//10

double percent=double(MHL_RandomUniformInt(0,100)); //Процент помехи

//Вызов функции
MHL_NoiseInVector(x,percent,VMHL_N);

//Используем полученный результат

MHL_ShowNumber (percent,"Процент помехи", "percent");
//Процент помехи:
//percent=89
MHL_ShowVector (x,VMHL_N,"Вектор с помехой", "x");
//Вектор с помехой:
//x =
//-1.95828
//2.17942
//1.76139
//4.45956
//3.82128
//8.0003
//6.80982
//5.94739
//9.03153
//8.59053

delete [] x;

```

#### 7.1.4 TMHL\_AcceptanceLimits

Функция вмещает вектор VMHL\_ResultVector в прямоугольную многомерной области, определяемой левыми границами и правыми границами. Если какая-то координата вектора выходит за границу, то значение этой координаты принимает граничное значение.

Код 64. Синтаксис

```

template <class T> void TMHL_AcceptanceLimits(T *VMHL_ResultVector, T *Left, T *Right
, int VMHL_N);

```

#### Входные параметры:

VMHL\_ResultVector — указатель на вектор (в него же записывается исправленный вектор);  
 Left — вектор левых границ;

Right — вектор правых границ;  
VMHL\_N — размерность вектора.

**Возвращаемое значение:**

Отсутствует.

Код 65. Пример использования

```
int VMHL_N=10; //Размер массива
double *a;
a=new double[VMHL_N];
double *Left;
Left=new double[VMHL_N];
double *Right;
Right=new double[VMHL_N];
TMHL_FillVector(Left,VMHL_N,-1.); //Левая граница
TMHL_FillVector(Right,VMHL_N,1.); //Правая граница

for (int i=0;i<VMHL_N;i++) a[i]=MHL_RandomUniform(-1.1,1.1);
MHL_ShowVector (a,VMHL_N, "Вектор", "a");
//Вектор:
//a =
// -0.199268
// -1.07664
// -0.395917
// 0.170935
// -0.720935
// -1.07878
// 1.01608
// -0.594714
// -1.09678
// 0.2513

//Вызов функции
TMHL_AcceptanceLimits(a,Left,Right,VMHL_N);

//Используем полученный результат
MHL_ShowVector (Left,VMHL_N, "Левые границы", "Left");
//Левые границы:
//Left =
//-1
//-1
//-1
//-1
//-1
//-1
//-1
//-1
//-1
//-1
//-1
MHL_ShowVector (Right,VMHL_N, "Правые границы", "Right");
// Правые границы:
//Right =
//1
//1
//1
//1
//1
```

```

//1
//1
//1
//1
//1

MHL_ShowVector (a,VMHL_N,"Отредактированный вектор", "a");
//Отредактированный вектор:
//a =
// -0.199268
// -0.395917
// 0.170935
// -0.720935
// -1
// 1
// -0.594714
// -1
// 0.2513

delete [] a;
delete [] Left;
delete [] Right;

```

### 7.1.5 TMHL\_CheckElementInVector

Функция проверяет наличие элемента а в векторе х.

Код 66. Синтаксис

```
template <class T> int TMHL_CheckElementInVector(T *x, int VMHL_N, T a);
```

**Входные параметры:**

х — указатель на вектор;

VMHL\_N — размер массива;

а — проверяемый элемент.

**Возвращаемое значение:**

Номер (начиная с нуля) первого элемента, равного искомому. Если такого элемента нет, то возвращается -1.

Код 67. Пример использования

```

int i;
int VMHL_N=10; //Размер массива (число строк)
int *a;
a=new int[VMHL_N];
//Заполним случайными числами
for (i=0;i<VMHL_N;i++)
    a[i]=MHL_RandomUniformInt(0,5);
int k=MHL_RandomUniformInt(0,5); //искомое число

//Вызов функции
int Search=TMHL_CheckElementInVector(a,VMHL_N,k);

```

```

//Используем полученный результат
MHL_ShowVector (a, VMHL_N, "Вектор", "a");
//Вектор:
//a =
//2
//1
//2
//1
//0
//1
//0
//3
//0
//0

MHL_ShowNumber (k, "Искомое число", "k");
//Искомое число:
//k=3

MHL_ShowNumber (Search, "находится в векторе a под номером", "Search");
//находится в векторе a под номером:
//Search=7
delete [] a;

```

### 7.1.6 TMHL\_CompareMeanOfVectors

Функция проверяет, какой вектор по среднему арифметическому больше.

Код 68. Синтаксис

```

template <class T> int TMHL_CompareMeanOfVectors(T *a, T *b, int VMHL_N);
template <class T> int TMHL_CompareMeanOfVectors(T *a, T *b, int VMHL_N1, int VMHL_N2
    );

```

**Входные параметры:**

a — первый вектор;  
b — второй вектор;  
VMHL\_N — размер векторов.

**Входные параметры функции—перезагрузки:**

a — первый вектор;  
b — второй вектор;  
VMHL\_N1 — размер вектора a;  
VMHL\_N2 — размер вектора b.

**Возвращаемое значение:**

0 — вектора совпадают или совпадает среднее арифметическое;  
1 — первый вектор имеет большее значение арифметического среднего;  
2 — второй вектор имеет большее значение арифметического среднего.

Код 69. Пример использования

```
int VMHL_Result;

int VMHL_N1=10;
int VMHL_N2=10;

double *a = new double[VMHL_N1];
double *b = new double[VMHL_N2];
TMHL_RandomIntVector(a,0.,100.,VMHL_N1);
TMHL_RandomIntVector(b,0.,20.,VMHL_N2);
MHL_ShowVectorT(a,VMHL_N1,"Первая выборка","a");
//Первая выборка:
//a =
//15 91 38 78 68 80 68 83 37 97

MHL_ShowVectorT(b,VMHL_N2,"Вторая выборка","b");
//Вторая выборка:
//b =
//7 10 0 18 11 4 18 3 12 13

//Вызов функции
VMHL_Result = TMHL_CompareMeanOfVectors(a, b, VMHL_N1, VMHL_N2);

//Используем результат
MHL_ShowNumber(VMHL_Result,"У какой выборки больше среднеарифметическое","VMHL_Result
");
//у какой выборки больше среднеарифметическое:
//VMHL_Result=1

delete [] a;
delete [] b;
```

### 7.1.7 TMHL\_EqualityOfVectors

Функция проверяет равенство векторов.

Код 70. Синтаксис

```
template <class T> bool TMHL_EqualityOfVectors(T *a, T *b, int VMHL_N);
```

**Входные параметры:**

a — первый вектор;

b — второй вектор;

VMHL\_N — размер векторов.

**Возвращаемое значение:**

true — вектора совпадают;

false — вектора не совпадают.

Код 71. Пример использования

```
int VMHL_N=5;//Размер массива (число строк)
int *a;
```

```

a=new int[VMHL_N];
int *b;
b=new int[VMHL_N];

int x=MHL_RandomUniformInt(0,2); //заполнитель для вектора a
int y=MHL_RandomUniformInt(0,2); //заполнитель для вектора b
TMHL_FillVector (a, VMHL_N, x);
TMHL_FillVector (b, VMHL_N, y);

//Вызов функции
int Q=TMHL_EqualityOfVectors(a,b,VMHL_N);

//Используем полученный результат
MHL_ShowVector (a,VMHL_N,"Вектор", "a");
//Вектор:
//a =
//1
//1
//1
//1
//1

MHL_ShowVector (b,VMHL_N,"Вектор", "b");
//Вектор:
//b =
//0
//0
//0
//0
//0

MHL_ShowNumber (Q,"Равны ли вектора", "Q");
// Равны ли вектора:
//Q=0

delete [] a;
delete [] b;

```

### 7.1.8 TMHL\_FibonacciNumbersVector

Функция заполняет массив числами Фибоначчи.

Код 72. Синтаксис

```

template <class T> void TMHL_FibonacciNumbersVector(T *VMHL_ResultVector, int VMHL_N)
;
```

**Входные параметры:**

VMHL\_ResultVector — указатель на массив, в который записывается результат;

VMHL\_N — размер массива.

**Возвращаемое значение:**

Отсутствует.

Код 73. Пример использования

```
int VMHL_N=MHL_RandomUniformInt(5,15); //Размер массива
double *x;
x=new double[VMHL_N];

//Вызов функции
TMHL_FibonacciNumbersVector(x,VMHL_N);

//Используем полученный результат
MHL_ShowVector (x,VMHL_N,"Вектор, заполненый числами Фибоначи", "x");
//Вектор, заполненый числами Фибоначи:
//x =
//1
//1
//2
//3
//5
//8
//13
//21
//34
//55
//89
//144

delete [] x;
```

### 7.1.9 TMHL\_FillVector

Функция заполняет вектор значениями, равных x.

Код 74. Синтаксис

```
template <class T> void TMHL_FillVector(T *VMHL_ResultVector, int VMHL_N, T x);
```

**Входные параметры:**

VMHL\_ResultVector — указатель на преобразуемый массив;

VMHL\_N — количество элементов в массиве;

x — число, которым заполняется вектор.

**Возвращаемое значение:** Отсутствует.

Код 75. Пример использования

```
int VMHL_N=10; //Размер массива
int *a;
a=new int[VMHL_N];

int x=5; //заполнитель

//Вызов функции
TMHL_FillVector(a,VMHL_N,x);

//Используем полученный результат
// MHL_ShowVector (a,VMHL_N,"Заполненный вектор", "a");
```

```
//Заполненный вектор:
// a[0] = 5
// a[1] = 5
// a[2] = 5
// a[3] = 5
// a[4] = 5
// a[5] = 5
// a[6] = 5
// a[7] = 5
// a[8] = 5
// a[9] = 5
delete [] a;
```

### 7.1.10 TMHL\_MaximumOfVector

Функция ищет максимальный элемент в векторе (одномерном массиве).

Код 76. Синтаксис

```
template <class T> T TMHL_MaximumOfVector(T *VMHL_Vector, int VMHL_N);
```

**Входные параметры:**

VMHL\_Vector — указатель на вектор (одномерный массив);

VMHL\_N — количество элементов в массиве.

**Возвращаемое значение:** Максимальный элемент.

Код 77. Пример использования

```
int VMHL_N=10; //Размер массива
double max;
double *a;
a=new double[VMHL_N];

for (int i=0;i<VMHL_N;i++) a[i]=MHL_RandomNumber(); //Заполняем случайными значениями

//Вызов функции
max=TMHL_MaximumOfVector(a,VMHL_N);

//Используем полученный результат
MHL_ShowVector (a,VMHL_N, "Заполненный вектор", "a");
//Заполненный вектор:
//a =
//0.0988159
//0.61557
//0.674866
//0.937286
//0.521759
//0.074585
//0.733337
//0.5979
//0.604309
//0.917114

MHL_ShowNumber (max, "Максимальное значение в векторе", "max");
//Максимальное значение в векторе:
//max=0.937286
```

```
delete [] a;
```

### 7.1.11 TMHL\_MinimumOfVector

Функция ищет минимальный элемент в векторе (одномерном массиве).

Код 78. Синтаксис

```
template <class T> T TMHL_MinimumOfVector(T *VMHL_Vector, int VMHL_N);
```

**Входные параметры:**

VMHL\_Vector — указатель на вектор (одномерный массив);

VMHL\_N — количество элементов в массиве.

**Возвращаемое значение:** Минимальный элемент.

Код 79. Пример использования

```
int VMHL_N=10; //Размер массива
double min;
double *a;
a=new double[VMHL_N];

for (int i=0;i<VMHL_N;i++) a[i]=MHL_RandomNumber(); //Заполняем случайными значениями

//Вызов функции
min=TMHL_MinimumOfVector(a,VMHL_N);

//Используем полученный результат
MHL_ShowVector (a,VMHL_N, "Заполненный вектор", "a");
//Заполненный вектор:
//a =
//0.777496
//0.446411
//0.14621
//0.938232
//0.354156
//0.831604
//0.420349
//0.50061
//0.491394
//0.0112305

MHL_ShowNumber (min, "Минимальное значение в векторе", "min");
//Максимальное значение в векторе:
//max=0.0112305

delete [] a;
```

### 7.1.12 TMHL\_MixingVector

Функция перемешивает массив. Поочередно рассматриваются номера элементов массивов. С некоторой вероятностью рассматриваемый элемент массива меняется местами со случайным элементом массива.

Код 80. Синтаксис

```
template <class T> void TMHL_MixingVector(T *VMHL_ResultVector, double P, int VMHL_N)  
;
```

**Входные параметры:**

VMHL\_ResultVector — указатель на исходный массив;

P — вероятность того, что элемент массива под рассматриваемым номером поменяется местами с каким-нибудь другим элементов (не включая самого себя);

VMHL\_N — размер массива.

**Возвращаемое значение:**

Отсутствует.

Код 81. Пример использования

```
int VMHL_N=10; //Размер массива  
int *x;  
x=new int[VMHL_N];  
//Заполним массив номерами от 1  
TMHL_OrdinalVector(x,VMHL_N);  
MHL_ShowVector (x,VMHL_N,"Вектор", "x");  
//Вектор:  
//x =  
//1  
//2  
//3  
//4  
//5  
//6  
//7  
//8  
//9  
//10  
  
double P=0.4; //Вероятность перемешивания  
  
//Вызов функции  
TMHL_MixingVector(x,P,VMHL_N); //Перемешаем массив  
  
//Используем полученный результат  
MHL_ShowVector (x,VMHL_N,"Перемешанный вектор", "x");  
//Перемешанный вектор:  
//x =  
//4  
//2  
//1  
//3  
//5  
//6
```

```
//7
//8
//9
//10

delete [] x;
```

### 7.1.13 TMHL\_MixingVectorWithConjugateVector

Функция перемешивает массив вместе со сопряженным массивом. Поочередно рассматриваются номера элементов массивов. С некоторой вероятностью рассматриваемый элемент массива меняется местами со случайным элементом массива. Пары элементов первого массива и сопряженного остаются без изменения.

Код 82. Синтаксис

```
template <class T, class T2> void TMHL_MixingVectorWithConjugateVector(T *
VMHL_ResultVector, T2 *VMHL_ResultVector2, double P, int VMHL_N);
```

**Входные параметры:**

VMHL\_ResultVector — указатель на исходный массив;

VMHL\_ResultVector2 — указатель на сопряженный массив;

P — вероятность того, что элемент массива под рассматриваемым номером поменяется местами с каким-нибудь другим элементов (не включая самого себя);

VMHL\_N — количество элементов в массивах.

**Возвращаемое значение:**

Отсутствует.

Код 83. Пример использования

```
int VMHL_N=10; //Размер массива
int *x;
x=new int[VMHL_N];
int *y;
y=new int[VMHL_N];
//Заполним массив номерами от 1
TMHL_OrdinalVector(x,VMHL_N);
//А сопряженный заполним номерами с нуля
TMHL_OrdinalVectorZero(y,VMHL_N);
MHL_ShowVectorT (x,VMHL_N,"Вектор", "x");
//Вектор:
//x =
//1 2 3 4 5 6 7 8 9 10

MHL_ShowVectorT (y,VMHL_N,"Вектор", "y");
//Вектор:
//y =
//0 1 2 3 4 5 6 7 8 9

double P=0.4; //Вероятность перемешивания

//Вызов функции
```

```

TMHL_MixingVectorWithConjugateVector(x,y,P,VMHL_N); //Перемешаем массив

//Используем полученный результат
MHL_ShowVectorT (x,VMHL_N,"Перемешанный вектор", "x");
// Перемешанный вектор:
//x =
//9   1   4   8   10  5   7   3   6   2

MHL_ShowVectorT (y,VMHL_N,"Сопряженный перемешанный вектор", "y");
//Сопряженный перемешанный вектор:
//y =
//8   0   3   7   9   4   6   2   5   1

delete [] x;
delete [] y;

```

### 7.1.14 TMHL\_NumberOfDifferentValuesInVector

Функция подсчитывает число различных значений в векторе (одномерном массиве).

Код 84. Синтаксис

```
template <class T> int TMHL_NumberOfDifferentValuesInVector(T *a, int VMHL_N);
```

**Входные параметры:**

a — указатель на вектор;

VMHL\_N — размер массива a.

**Возвращаемое значение:**

Отсутствует.

**Примечание:** Алгоритм очень тупоры и медленный.

Код 85. Пример использования

```

int i;
int VMHL_N=10; //Размер массива (число строк)
int *a;
a=new int[VMHL_N];
//Заполним случайными числами
for (i=0;i<VMHL_N;i++)
    a[i]=MHL_RandomUniformInt(0,5);

//Вызов функции
int NumberOfDifferent=TMHL_NumberOfDifferentValuesInVector(a,VMHL_N);

//Используем полученный результат
MHL_ShowVector (a,VMHL_N,"Случайный вектор", "a");
//Случайный вектор:
//a =
//2
//1
//1
//4
//0
//2

```

```

//1
//1
//2
//2

MHL_ShowNumber (NumberOfDifferent, "Число различных значений в векторе", "
    NumberOfDifferent");
//Число различных значений в векторе:
//NumberOfDifferent=4
delete [] a;

```

### 7.1.15 TMHL\_NumberOfMaximumOfVector

Функция ищет номер максимального элемента в векторе (одномерном массиве).

Код 86. Синтаксис

```
template <class T> int TMHL_NumberOfMaximumOfVector(T *a, int VMHL_N);
```

**Входные параметры:**

a — указатель на вектор (одномерный массив);

VMHL\_N — размер массива.

**Возвращаемое значение:**

Номер максимального элемента.

Код 87. Пример использования

```

int i;
int VMHL_N=10; //Размер массива
double *Vector=new double[VMHL_N];
//Заполним случайными числами
for (i=0;i<VMHL_N;i++) Vector[i]=MHL_RandomNumber();

//Вызов функции
double Number=TMHL_NumberOfMaximumOfVector(Vector,VMHL_N);

//Используем полученный результат
MHL_ShowVector (Vector,VMHL_N,"Случайный массив", "Vector");
//Случайный массив:
//Vector =
//0.9245
//0.221466
//0.301544
//0.643951
//0.881958
//0.832764
//0.104462
//0.0611267
//0.943604
//0.335205

MHL_ShowNumber(Number, "Номер максимального элемента", "Number"); //Например, выводим ре
зультат
// Номер максимального элемента:
//Number=8

```

```
delete [] Vector;
```

### 7.1.16 TMHL\_NumberOfMinimumOfVector

Функция ищет номер минимального элемента в векторе (одномерном массиве).

Код 88. Синтаксис

```
template <class T> int TMHL_NumberOfMinimumOfVector(T *a, int VMHL_N);
```

**Входные параметры:**

a — указатель на вектор (одномерный массив);

VMHL\_N — размер массива.

**Возвращаемое значение:**

Номер минимального элемента.

Код 89. Пример использования

```
int i;
int VMHL_N=10; //Размер массива
double *Vector=new double[VMHL_N];
//Заполним случайными числами
for (i=0;i<VMHL_N;i++) Vector[i]=MHL_RandomNumber();

//Вызов функции
double Number=TMHL_NumberOfMinimumOfVector(Vector,VMHL_N);

//Используем полученный результат
MHL_ShowVector (Vector,VMHL_N,"Случайный массив", "Vector");
//Случайный массив:
//Vector =
//0.958344
//0.0968323
//0.689697
//0.102264
//0.142242
//0.135925
//0.473816
//0.0245056
//0.616333
//0.798065

MHL_ShowNumber(Number, "Номер минимального элемента", "Number"); //Например, выводим результат
//Номер минимального элемента:
//Number=7
delete [] Vector;
```

### 7.1.17 TMHL\_NumberOfNegativeValues

Функция подсчитывает число отрицательных значений в векторе (одномерном массиве).

Код 90. Синтаксис

```
template <class T> int TMHL_NumberOfNegativeValues(T *a, int VMHL_N);
```

**Входные параметры:**

a — указатель на вектор (одномерный массив);

VMHL\_N — размер массива.

**Возвращаемое значение:**

Число отрицательных значений в массиве.

Код 91. Пример использования

```
int i;
int VMHL_N=10; //Размер массива (число строк)
int *a;
a=new int[VMHL_N];
//Заполним случайными числами
for (i=0;i<VMHL_N;i++)
    a[i]=MHL_RandomUniformInt(-20,20);

//Вызов функции
int NumberOfNegative=TMHL_NumberOfNegativeValues(a,VMHL_N);

//Используем полученный результат
MHL_ShowVector (a,VMHL_N,"Случайный вектор", "a");
//Случайный вектор:
//a =
//12
//19
//11
//20
//13
//4
//6
//1
//1
//8

MHL_ShowNumber (NumberOfNegative, "Число отрицательных значений в векторе", "NumberOfNegative");
//Число отрицательных значений в векторе:
//NumberOfNegative=5

delete [] a;
```

### 7.1.18 TMHL\_NumberOfPositiveValues

Функция подсчитывает число положительных значений в векторе (одномерном массиве).

Код 92. Синтаксис

```
template <class T> int TMHL_NumberOfPositiveValues(T *a, int VMHL_N);
```

**Входные параметры:**

*a* — указатель на вектор (одномерный массив);

**VMHL\_N** — размер массива.

## Возвращаемое значение:

## Число положительных значений в массиве.

### Код 93. Пример использования

```
int i;
int VMHL_N=10; //Размер массива (число строк)
int *a;
a=new int[VMHL_N];
//Заполним случайными числами
for (i=0;i<VMHL_N;i++)
a[i]=MHL_RandomUniformInt(-20,20);

//Вызов функции
int NumberOfNegative=TMHL_NumberOfPositiveValues(a,VMHL_N);

//Используем полученный результат
MHL_ShowVector (a,VMHL_N,"Случайный вектор", "a");
//Случайный вектор:
//a =
//6
//14
//14
//13
//13
//19
//18
//11
//11
//18
//20
//5

MHL_ShowNumber (NumberOfNegative,"Число положительных значений в векторе",
    NumberOfNegative);
//Число положительных значений в векторе:
//NumberOfNegative=6

delete [] a;
```

### 7.1.19 TMHL NumberOfZeroValues

Функция подсчитывает число нулевых значений в векторе (одномерном массиве).

## Код 94. Синтаксис

```
template <class T> int TMHL_NumberOfZeroValues(T *a, int VMHL_N);
```

## Входные параметры:

*a* — указатель на вектор (одномерный массив);

**VMHL\_N** — размер массива.

## Возвращаемое значение:

Число нулевых значений в массиве.

Код 95. Пример использования

```
int i;
int VMHL_N=10; //Размер массива (число строк)
int *a;
a=new int[VMHL_N];
//Заполним случайными числами
for (i=0;i<VMHL_N;i++)
a[i]=MHL_RandomUniformInt(-2,2);

//Вызов функции
int NumberOfNegative=TMHL_NumberOfZeroValues(a,VMHL_N);

//Используем полученный результат
MHL_ShowVector (a,VMHL_N,"Случайный вектор", "a");
//Случайный вектор:
//a =
//1
//0
//0
//0
//0
//1
//1
//0
//1

MHL_ShowNumber (NumberOfNegative,"Число нулевых значений в векторе", "NumberOfNegative");
//Число нулевых значений в векторе:
//NumberOfNegative=4

delete [] a;
```

### 7.1.20 TMHL\_OrdinalVector

Функция заполняет вектор значениями, равные номеру элемента, начиная с единицы.

Код 96. Синтаксис

```
template <class T> void TMHL_OrdinalVector(T *VMHL_ResultVector, int VMHL_N);
```

**Входные параметры:**

VMHL\_ResultVector — указатель на вектор (одномерный массив), который и заполняется;

VMHL\_N — размер массива.

**Возвращаемое значение:**

Отсутствует.

Код 97. Пример использования

```
int VMHL_N=10; //Размер массива (число строк)
double *a;
```

```

a=new double[VMHL_N];

//Вызов функции
TMHL.OrdinalVector(a,VMHL_N);
//Вектор:
//a =
//1
//2
//3
//4
//5
//6
//7
//8
//9
//10

//Используем полученный результат
MHL>ShowVector (a,VMHL_N, "Вектор", "a");

delete [] a;

```

### 7.1.21 TMHL.OrdinalVectorZero

Функция заполняет вектор значениями, равные номеру элемента, начиная с нуля.

Код 98. Синтаксис

```
template <class T> void TMHL.OrdinalVectorZero(T *VMHL_ResultVector, int VMHL_N);
```

**Входные параметры:**

VMHL\_ResultVector — указатель на вектор (одномерный массив), который и заполняется;

VMHL\_N — размер массива.

**Возвращаемое значение:**

Отсутствует.

Код 99. Пример использования

```

int VMHL_N=10;//Размер массива (число строк)
double *a;
a=new double[VMHL_N];

//Вызов функции
TMHL.OrdinalVectorZero(a,VMHL_N);
//Вектор:
//a =
//0
//1
//2
//3
//4
//5
//6
//7
//8

```

```
//9

//Используем полученный результат
MHL_ShowVector (a,VMHL_N,"Вектор", "a");

delete [] a;
```

### 7.1.22 TMHL\_ProductOfElementsOfVector

Функция вычисляет произведение элементов вектора.

Код 100. Синтаксис

```
template <class T> T TMHL_ProductOfElementsOfVector(T *VMHL_Vector,int VMHL_N);
```

**Входные параметры:**

VMHL\_Vector — указатель на вектор (одномерный массив);

VMHL\_N — количество элементов в массиве.

**Возвращаемое значение:** Произведение элементов массива.

Код 101. Пример использования

```
int VMHL_N=5;//Размер массива
double p;
double *a;
a=new double[VMHL_N];

for (int i=0;i<VMHL_N;i++) a[i]=MHL_RandomUniformIntIncluding(1, 4);//Заполняем случай
ными значениями

//Вызов функции
p=TMHL_ProductOfElementsOfVector(a,VMHL_N);

//Используем полученный результат
MHL_ShowVector (a,VMHL_N,"Заполненный вектор", "a");
//Заполненный вектор:
//a =
//4
//3
//1
//3
//2

MHL_ShowNumber (p,"Произведение элементов массива", "p");
//Произведение элементов массива:
//p=72

delete [] a;
```

### 7.1.23 TMHL\_ReverseVector

Функция меняет порядок элементов в массиве на обратный. Преобразуется подаваемый массив.

Код 102. Синтаксис

```
template <class T> void TMHL_ReverseVector(T *VMHL_ResultVector, int VMHL_N);
```

**Входные параметры:**

VMHL\_ResultVector — указатель на преобразуемый массив;

VMHL\_N — размер массива.

**Возвращаемое значение:**

Отсутствует.

Код 103. Пример использования

```
int i;
int VMHL_N=MHL_RandomUniformInt(2,10); //Размер массива (число строк)
double *a;
a=new double[VMHL_N];
//Заполним случайными числами
for (i=0;i<VMHL_N;i++)
    a[i]=MHL_RandomUniformInt(10,100);

MHL_ShowVector (a,VMHL_N,"Вектор равен", "a");
//Вектор равен:
//a =
//83
//57
//55
//52
//70
//73

//Вызов функции
TMHL_ReverseVector(a,VMHL_N);

//Используем полученный результатом
MHL_ShowVector (a,VMHL_N,"Теперь вектор равен", "a");
//Теперь вектор равен:
//a =
//73
//70
//52
//55
//57
//83

delete [] a;
```

### 7.1.24 TMHL\_SearchElementInVector

Функция находит номер первого элемента в массиве, равного данному.

Код 104. Синтаксис

```
template <class T> int TMHL_SearchElementInVector (T *x, T x, int VMHL_N);
```

**Входные параметры:**

X — исходный массив;

x — данный элемент;

VMHL\_N — размер массива.

**Возвращаемое значение:**

Номер элемента. Если такого элемента нет, то возвращается —1.

Код 105. Пример использования

```
int i;
int VMHL_N=10; //Размер массива (число строк)
int *a;
a=new int[VMHL_N];
//Заполним случайными числами
for (i=0;i<VMHL_N;i++)
a[i]=MHL_RandomUniformInt(0, 4);

int x=2;

//Вызов функции
int Number=TMHL_SearchElementInVector(a,x,VMHL_N);

//Используем полученный результат
MHL_ShowVector (a,VMHL_N,"Случайный вектор", "a");
//Случайный вектор:
//a =
//3
//3
//0
//1
//0
//2
//2
//1
//3
//1

MHL_ShowNumber (x,"Искомое число", "x");
//Искомое число:
//x=2

MHL_ShowNumber (Number,"Номер первого элемента, равного искомому", "Number");
//Номер первого элемента, равного искомому:
//Number=5

delete [] a;
```

### 7.1.25 TMHL\_SearchFirstNotZero

Функция возвращает номер первого ненулевого элемента массива.

Код 106. Синтаксис

```
template <class T> int TMHL_SearchFirstNotZero(T *x, int VMHL_N);
```

**Входные параметры:**

*x* — указатель на вектор (одномерный массив);

*VMHL\_N* — размер массива.

**Возвращаемое значение:**

Номер первого ненулевого элемента массива (начиная с нуля). Если такого элемента нет, то возвращается -1.

Код 107. Пример использования

```
int i;
int VMHL_N=10; //Размер массива (число строк)
int *a;
a=new int[VMHL_N];
//Заполним случайными числами
for (i=0;i<VMHL_N;i++)
    a[i]=MHL_RandomUniformInt(0,2);

//Вызов функции
int Number=TMHL_SearchFirstNotZero(a,VMHL_N);

//Используем полученный результат
MHL_ShowVector (a,VMHL_N,"Случайный вектор", "a");
//Случайный вектор:
//a =
//0
//0
//0
//1
//0
//0
//0
//1
//1
//1
//0

MHL_ShowNumber (Number,"Номер первого ненулевого элемента", "Number");
//Номер первого ненулевого элемента:
//Number=3

delete [] a;
```

### 7.1.26 TMHL\_SearchFirstZero

Функция возвращает номер первого нулевого элемента массива.

Код 108. Синтаксис

```
template <class T> int TMHL_SearchFirstZero(T *x, int VMHL_N);
```

**Входные параметры:**

*x* — указатель на вектор (одномерный массив);

*VMHL\_N* — размер массива.

**Возвращаемое значение:**

Номер первого нулевого элемента массива (начиная с нуля). Если такого элемента нет, то возвращается -1.

Код 109. Пример использования

```
int i;
int VMHL_N=10; //Размер массива (число строк)
int *a;
a=new int[VMHL_N];
//Заполним случайными числами
for (i=0;i<VMHL_N;i++)
    a[i]=MHL_RandomUniformInt(0,2);

//Вызов функции
int Number=TMHL_SearchFirstZero(a,VMHL_N);

//Используем полученный результат
MHL_ShowVector (a,VMHL_N, "Случайный вектор", "a");
//Случайный вектор:
//a =
//1
//1
//1
//0
//0
//1
//0
//0
//0
//1

MHL_ShowNumber (Number, "Номер первого нулевого элемента", "Number");
//Номер первого нулевого элемента:
//Number=3

delete [] a;
```

### 7.1.27 TMHL\_SumSquareVector

Функция вычисляет сумму квадратов элементов вектора.

Код 110. Синтаксис

```
template <class T> T TMHL_SumSquareVector(T *VMHL_Vector, int VMHL_N);
```

**Входные параметры:**

VMHL\_Vector — указатель на вектор (одномерный массив);

VMHL\_N — количество элементов в массиве.

**Возвращаемое значение:** Сумма квадратов элементов массива.

Код 111. Пример использования

```
int VMHL_N=10; //Размер массива
double sum;
double *a;
a=new double[VMHL_N];
```

```

for (int i=0;i<VMHL_N;i++) a[i]=i;//Заполняем значениями

//Вызов функции
sum=TMHL_SumSquareVector(a,VMHL_N);

//Используем полученный результат
MHL_ShowVector (a,VMHL_N, "Заполненный вектор", "a");
//Заполненный вектор:
//a =
//0
//1
//2
//3
//4
//5
//6
//7
//8
//9

MHL_ShowNumber (sum, "Сумма квадратов элементов массива", "sum");
//Сумма квадратов элементов массива:
//sum=285

delete [] a;

```

### 7.1.28 TMHL\_SumVector

Функция вычисляет сумму элементов вектора.

Код 112. Синтаксис

```
template <class T> T TMHL_SumVector(T *VMHL_Vector, int VMHL_N);
```

**Входные параметры:**

VMHL\_Vector — указатель на вектор (одномерный массив);

VMHL\_N — количество элементов в массиве.

**Возвращаемое значение:** Сумма элементов вектора.

Код 113. Пример использования

```

int VMHL_N=10;//Размер массива
double sum;
double *a;
a=new double[VMHL_N];

for (int i=0;i<VMHL_N;i++) a[i]=MHL_RandomNumber();//Заполняем случайными значениями

//Вызов функции
sum=TMHL_SumVector(a,VMHL_N);

//Используем полученный результат
MHL_ShowVector (a,VMHL_N, "Заполненный вектор", "a");
//Заполненный вектор:
//a =

```

```

//0.886475
//0.998413
//0.242859
//0.221405
//0.292175
//0.134247
//0.723846
//0.271393
//0.188904
//0.727936

MHL_ShowNumber (sum, "Сумма элементов массива", "sum");
//Сумма элементов массива:
//sum=4.68765

delete [] a;

```

### 7.1.29 TMHL\_VectorMinusVector

Функция вычитает поэлементно из одного массива другой и записывает результат в третий массив. Или в переопределенном виде функция вычитает поэлементно из одного массива другой и записывает результат в первый массив.

Код 114. Синтаксис

```

template <class T> void TMHL_VectorMinusVector(T *a, T *b, T *VMHL_ResultVector, int
    VMHL_N);
template <class T> void TMHL_VectorMinusVector(T *VMHL_ResultVector, T *b, int VMHL_N
    );

```

**Входные параметры:**

a — первый вектор;

b — второй вектор;

VMHL\_ResultVector — вектор разности;

VMHL\_N — размер векторов.

**Возвращаемое значение:**

Отсутствует.

Для переопределенной функции:

**Входные параметры:**

VMHL\_ResultVector — первый вектор, из которого вычитают второй вектор;

b — второй вектор;

VMHL\_N — размер векторов.

**Возвращаемое значение:**

Отсутствует.

Код 115. Пример использования

```

int i;
int VMHL_N=10; //Размер массива (число строк)
int *a;
a=new int[VMHL_N];
int *b;
b=new int[VMHL_N];
int *c;
c=new int[VMHL_N];
//Заполним случайными числами
for (i=0;i<VMHL_N;i++)
    a[i]=MHL_RandomUniformInt(0,10);
for (i=0;i<VMHL_N;i++)
    b[i]=MHL_RandomUniformInt(0,10);

//Вызов функции
TMHL_VectorMinusVector(a,b,c,VMHL_N);

//Используем полученный результат
MHL_ShowVectorT (a,VMHL_N, "Случайный вектор", "a");
//Случайный вектор:
//a =
//0    7   0   0   8   5   0   4   8   2

MHL_ShowVectorT (b,VMHL_N, "Случайный вектор", "b");
//Случайный вектор:
//b =
//6    1   3   1   2   7   2   6   1   4

MHL_ShowVectorT (c,VMHL_N, "Их разница", "c");
//Их разница:
//c =
// -6   6   -3  -1  6   -2  -2  -2  7   -2

delete [] a;
delete [] b;
delete [] c;

//Для переопределенной функции
VMHL_N=10; //Размер массива (число строк)
a=new int[VMHL_N];
b=new int[VMHL_N];
//Заполним случайными числами
for (i=0;i<VMHL_N;i++)
    a[i]=MHL_RandomUniformInt(0,10);
for (i=0;i<VMHL_N;i++)
    b[i]=MHL_RandomUniformInt(0,10);

MHL_ShowVectorT (a,VMHL_N, "Случайный вектор", "a");
//Случайный вектор:
//a =
//6    9   3   0   2   9   4   2   3   7

//Вызов функции
TMHL_VectorMinusVector(a,b,VMHL_N);

//Используем полученный результат
MHL_ShowVectorT (b,VMHL_N, "Случайный вектор", "b");
//Случайный вектор:

```

```

//b =
//5   6   3   8   5   0   7   6   4   4

MHL_ShowVectorT (a,VMHL_N, "Из первого вычли второй", "a");
//Из первого вычли второй:
//a =
//1   3   0   -8  -3  9   -3  -4  -1  3

delete [] a;
delete [] b;

```

### 7.1.30 TMHL\_VectorMultiplyNumber

Функция умножает вектор на число.

Код 116. Синтаксис

```

template <class T> void TMHL_VectorMultiplyNumber(T *VMHL_ResultVector, int VMHL_N, T Number);

```

**Входные параметры:**

VMHL\_ResultVector — вектор (в нем и сохраняется результат);

VMHL\_N — размер вектора;

Number — число, на которое умножается вектор.

**Возвращаемое значение:**

Отсутствует.

Код 117. Пример использования

```

int i;
int VMHL_N=10; //Размер массива (число строк)
double *a;
a=new double[VMHL_N];
//Заполним случайными числами
for (i=0;i<VMHL_N;i++)
a[i]=MHL_RandomUniformInt(0,10);

MHL_ShowVector (a,VMHL_N, "Случайный вектор", "a");
//Случайный вектор:
//a =
//4
//6
//3
//5
//4
//7
//8
//2
//1
//0

double Number=MHL_RandomUniform(0,10);

//Вызов функции

```

```

TMHL_VectorMultiplyNumber(a, VMHL_N, Number);

//Используем полученный результат
MHL_ShowNumber (Number, "Случайный множитель", "Number");
//Случайный множитель:
//Number=3.57941

MHL_ShowVector (a, VMHL_N, "Умножили на число Number", "a");
//Умножили на число Number:
//a =
//14.3176
//21.4764
//10.7382
//17.897
//14.3176
//25.0558
//28.6353
//7.15881
//3.57941
//0

delete [] a;

```

### 7.1.31 TMHL\_VectorPlusVector

Функция складывает поэлементно из одного массива другой и записывает результат в третий массив. Или в переопределенном виде функция складывает поэлементно из одного массива другой и записывает результат в первый массив.

Код 118. Синтаксис

```

template <class T> void TMHL_VectorPlusVector(T *a, T *b, T *VMHL_ResultVector, int
    VMHL_N);
template <class T> void TMHL_VectorPlusVector(T *VMHL_ResultVector, T *b, int VMHL_N)
    ;

```

#### **Входные параметры:**

a — первый вектор;  
 b — второй вектор;  
 VMHL\_ResultVector — вектор суммы;  
 VMHL\_N — размер векторов.

#### **Возвращаемое значение:**

Отсутствует.

Для переопределенной функции:

#### **Входные параметры:**

VMHL\_ResultVector — первый вектор, к которому прибавляют второй вектор;  
 b — второй вектор;  
 VMHL\_N — размер векторов.

## **Возвращаемое значение:**

Отсутствует.

Код 119. Пример использования

```
int i;
int VMHL_N=10; //Размер массива (число строк)
int *a;
a=new int[VMHL_N];
int *b;
b=new int[VMHL_N];
int *c;
c=new int[VMHL_N];
//Заполним случайными числами
for (i=0;i<VMHL_N;i++)
    a[i]=MHL_RandomUniformInt(0,10);
for (i=0;i<VMHL_N;i++)
    b[i]=MHL_RandomUniformInt(0,10);

//Вызов функции
TMHL_VectorPlusVector(a,b,c,VMHL_N);

//Используем полученный результатом
MHL_ShowVectorT (a,VMHL_N,"Случайный вектор", "a");
//Случайный вектор:
//a =
//2    7   9   2   3   3   3   2   8   8

MHL_ShowVectorT (b,VMHL_N,"Случайный вектор", "b");
//Случайный вектор:
//b =
//3    7   2   9   5   3   2   7   2   7

MHL_ShowVectorT (c,VMHL_N,"Их сумма", "c");
//Их сумма:
//c =
//5    14  11  11  8   6   5   9   10  15

delete [] a;
delete [] b;
delete [] c;

//Для переопределенной функции
VMHL_N=10; //Размер массива (число строк)
a=new int[VMHL_N];
b=new int[VMHL_N];
//Заполним случайными числами
for (i=0;i<VMHL_N;i++)
    a[i]=MHL_RandomUniformInt(0,10);
for (i=0;i<VMHL_N;i++)
    b[i]=MHL_RandomUniformInt(0,10);

MHL_ShowVectorT (a,VMHL_N,"Случайный вектор", "a");
//Случайный вектор:
//a =
//0    6   7   4   9   3   9   8   5   6

//Вызов функции
TMHL_VectorPlusVector(a,b,VMHL_N);
```

```

//Используем полученный результат
MHL_ShowVectorT (b,VMHL_N,"Случайный вектор", "b");
//Случайный вектор:
//b =
//1 7 0 5 4 0 9 5 7 7

MHL_ShowVectorT (a,VMHL_N,"К первому прибавили второй", "a");
//К первому прибавили второй:
//a =
//1 13 7 9 13 3 18 13 12 13

delete [] a;
delete [] b;

```

### 7.1.32 TMHL\_VectorToVector

Функция копирует содержимое вектора (одномерного массива) в другой.

Код 120. Синтаксис

```
template <class T> void TMHL_VectorToVector(T *VMHL_Vector, T *VMHL_ResultVector, int VMHL_N);
```

**Входные параметры:**

VMHL\_Vector — указатель на исходный массив;

VMHL\_ResultVector — указатель на массив в который производится запись;

VMHL\_N — размер массивов.

**Возвращаемое значение:** Отсутствует.

Код 121. Пример использования

```

int VMHL_N=10;//Размер массива

double *a;
a=new double[VMHL_N];
for (int i=0;i<VMHL_N;i++) a[i]=MHL_RandomNumber();//Заполняем случайными значениями

double *b;
b=new double[VMHL_N];

//Вызов функции
TMHL_VectorToVector(a,b,VMHL_N);

//Используем полученный результат
MHL_ShowVector (a,VMHL_N,"Первоначальный вектор", "a");
//Первоначальный вектор:
//a =
//0.874634
//0.28656
//0.676056
//0.861755
//0.0521851
//0.308319
//0.348267

```

```
//0.431671
//0.186462
//0.562805

MHL_ShowVector (b,VMHL_N,"Вектор, в который скопировали первый", "b");
//Вектор, в который скопировали первый:
//b =
//0.874634
//0.28656
//0.676056
//0.861755
//0.0521851
//0.308319
//0.348267
//0.431671
//0.186462
//0.562805

delete [] a;
delete [] b;
```

### 7.1.33 TMHL ZeroVector

Функция зануляет массив.

## Код 122. Синтаксис

```
template <class T> void TMHL_ZeroVector(T *VMHL_ResultVector, int VMHL_N);
```

### **Входные параметры:**

**VMHL\_Vector** — указатель на вектор (одномерный массив);

VMHL N – количество элементов в массиве.

**Возвращаемое значение:** Отсутствует.

Код 123. Пример использования

```
int VMHL_N=10; //Размер массива
double *a;
a=new double[VMHL_N];

//Вызов функции
TMHL_ZeroVector(a,VMHL_N);

//Используем полученный результат
MHL_ShowVector (a,VMHL_N, "Занулленный вектор", "a");
//Занулленный вектор:
//a =
//0
//0
//0
//0
//0
//0
//0
//0
//0
//0
```

```
//0  
delete [] a;
```

## 7.2 Генетические алгоритмы

### 7.2.1 MHL\_BinaryFitnessFunction

Служебная функция. Функция вычисляет целевую функцию бинарного вектора, в котором закодирован вещественный вектор. Использует внутренние служебные переменные. Функция для MHL\_StandartRealGeneticAlgorithm. Использовать для своих целей не рекомендуется.

Код 124. Синтаксис

```
double MHL_BinaryFitnessFunction(int*x, int VMHL_N);
```

**Входные параметры:**

x — бинарный вектор;

VMHL\_N — количество элементов в векторе.

**Возвращаемое значение:**

Значение целевой функции бинарного вектора.

**Примечание:**

Используемые переменные, переодеваемые из MHL\_StandartRealGeneticAlgorithm:

VMHL\_TempFunction — указатель на целевая функция для вещественного решения;

VMHL\_TempInt1 — указатель на массив, сколько бит приходится в бинарной хромосоме на кодирование ;

VMHL\_TempDouble1 — указатель на массив левых границ изменения вещественной переменной;

VMHL\_TempDouble2 — указатель на массив правых границ изменения вещественной переменной;

VMHL\_TempDouble3 — указатель на массив, в котором можно сохранить вещественный индивид при его раскодировании из бинарной строки;

VMHL\_TempInt2 — указатель на размерность вещественного вектора;

VMHL\_TempInt3 — указатель на тип преобразования вещественной задачи оптимизации в бинарное.

Код 125. Пример использования

```
//Служебная функция. Не предназначена для самостоятельного использования.
```

## 7.2.2 MHL\_MakeVectorOfProbabilityForProportionalSelectionV2

Функция формирует вектор вероятностей выбора индивидов из вектора значений функции пригодности. Формирование вектора происходит согласно правилам пропорционально селекции из ГА. Это служебная функция для использования функции пропорциональной селекции MHL\_ProportionalSelectionV2.

Код 126. Синтаксис

```
void MHL_MakeVectorOfProbabilityForProportionalSelectionV2(double *Fitness, double *
VMHL_ResultVector, int VMHL_N);
```

### Входные параметры:

Fitness — массив пригодностей (можно подавать не массив пригодностей, а массив значений целевой функции, но только для задач безусловной оптимизации);

VMHL\_ResultVector — вектор вероятностей выбора индивидов из популяции, который мы и формируем;

VMHL\_N — размер массива пригодностей.

### Возвращаемое значение:

Отсутствует.

### О функции:

Это служебная функция для использования функции пропорциональной селекции MHL\_SelectionProportionalV2. Формирование вектора происходит согласно правилам пропорциональной селекции из ГА. Работает в связке с функцией MHL\_SelectionProportionalV2. Оператор селекции работает с массивом пригодностей индивидов, но непосредственно пропорциональная селекция выбирает индивида исходя из вероятностей выбора индивидов. Каждый раз для выбора индивида создавать массив вероятностей затратно, поэтому для каждой популяции на каждом поколении вначале вызывается функция MHL\_MakeVectorProbabilityForSelectionProportionalV2 для генерации вектора вероятностей выбора индивида, а затем этот массив и подставляется в пропорциональную селекцию.

### Примечание:

Под массивом пригодностей понимается специально преобразованный массив значений целевой функции. Процесс подробно описан в стандарте генетического алгоритма. Смотреть здесь. Но это если Вы используете в алгоритмах оптимизации подобных генетическому. А так, если будете использовать, то учитывайте, что массив пригодностей — это массив вещественных чисел из отрезка [0; 1].

### Примечание:

Не используйте эту функцию над векторами целых типов int, long. Вектор обнулится кроме одно какого-нибудь элемента, так как нормировка вектора предполагает числа из интервала [0; 1].

Код 127. Пример использования

```
int i;
int VMHL_N=10; //Размер массива (число строк)
double *Fitness;
Fitness=new double[VMHL_N];
```

```

//Заполним вектор случайными значениями пригодностей индивидов
//на практике, конечно, пригодности вычисляются, например, в
//процессе работы ГА
for (i=0;i<VMHL_N;i++) Fitness[i]=MHL_RandomNumber();

//Для работы этого варианта пропорциональной селекции нужен
//массив вероятностей выбора индивидов для порциональной селекции;
double *VectorProbability;
VectorProbability=new double[VMHL_N];

//Вызов функции
MHL_MakeVectorOfProbabilityForProportionalSelectionV2(Fitness,VectorProbability,
VMHL_N);

//Используем полученный результат
MHL_ShowVector (Fitness,VMHL_N,"Вектор пригодностей индивидов", "a");
//Вектор пригодностей индивидов:
//a =
//0.902191
//0.804932
//0.0402527
//0.344849
//0.375427
//0.0223999
//0.650024
//0.207642
//0.275391
//0.164215

MHL_ShowVector (VectorProbability,VMHL_N,"Вектор вероятностей выбора индивидов", "VectorProbability");
// Вектор вероятностей выбора индивидов:
//VectorProbability =
//0.246902
//0.219607
//0.00501015
//0.090491
//0.0990725
//0
//0.176135
//0.0519856
//0.0709985
//0.0397986

MHL_ShowNumber (TMHL_SumVector(VectorProbability,VMHL_N), "Его сумма", "Sum");
//Его сумма:
//Sum=1

delete [] Fitness;
delete [] VectorProbability;

```

### 7.2.3 MHL\_MakeVectorOfProbabilityForRankSelection

Функция формирует вектор вероятностей выбора индивидов из вектора рангов для ранговой селекции. Это служебная функция для использования функции ранговой селекции MHL\_RankSelection.

Код 128. Синтаксис

```
void MHL_MakeVectorOfProbabilityForRankSelection(double *Rank, double *
VMHL_ResultVector, int VMHL_N);
```

### Входные параметры:

Rank — массив рангов, которые были посчитаны функцией MHL\_MakeVectorOfRankForRankSelection;

VMHL\_ResultVector — вектор вероятностей выбора индивидов из популяции, который мы и формируем;

VMHL\_N — размер массива пригодностей.

### Возвращаемое значение:

Отсутствует.

Код 129. Пример использования

```
int i;
int VMHL_N=7; //Размер массива (число строк)
double *Fitness;
Fitness=new double[VMHL_N];
for (i=0;i<VMHL_N;i++)
    Fitness[i]=MHL_RandomUniformInt(1,10)/10.;

double *Rank;
Rank=new double[VMHL_N];

double *VectorProbability;
VectorProbability=new double[VMHL_N];

//Получаем массив рангов
MHL_MakeVectorOfRankForRankSelection(Fitness,Rank,VMHL_N);

//Вызов функции
MHL_MakeVectorOfProbabilityForRankSelection(Rank,VectorProbability,VMHL_N);

//Используем полученный результат

MHL_ShowVector (Fitness,VMHL_N,"Массив пригодностей", "Fitness");
//Массив пригодностей:
//Fitness =
//0.5
//0.7
//0.3
//0.9
//0.8
//0.7
//0.9

MHL_ShowVector (Rank,VMHL_N,"Массив рангов", "Rank");
//Массив рангов:
//Rank =
//2
//3.5
//1
//6.5
//5
//3.5
//6.5
```

```

MHL_ShowVector (VectorProbability, VMHL_N, "Вектор вероятностей выбора", "VectorProbability");
//Вектор вероятностей выбора:
//VectorProbability =
//0.0714286
//0.125
//0.0357143
//0.232143
//0.178571
//0.125
//0.232143

delete [] Fitness;
delete [] Rank;
delete [] VectorProbability;

```

#### 7.2.4 MHL\_MakeVectorOfRankForRankSelection

Проставляет ранги для элементов не сортированного массива, то есть номера, начиная с 1, в отсортированном массиве. Если в массиве есть несколько одинаковых элементов, то ранги им присуждаются как среднеарифметические. Это служебная функция для функции MHL\_RankSelection.

Код 130. Синтаксис

```
void MHL_MakeVectorOfRankForRankSelection(double *Fitness, double *VMHL_ResultVector,
                                          int VMHL_N);
```

##### Входные параметры:

Fitness — массив пригодностей (можно подавать не массив пригодностей, а массив значений целевой функции, но только для задач безусловной оптимизации);

VMHL\_ResultVector — массив рангов, который мы и формируем;

VMHL\_N — размер массива пригодностей.

##### Возвращаемое значение:

Отсутствует.

##### О функции:

Это служебная функция для использования функции ранговой селекции MHL\_RankSelection. Формирование вектора происходит согласно правилам ранговой селекции из ГА. Проставляет ранги для элементов несортированного массива, то есть номера, начиная с 1, в отсортированном массиве. Если в массиве есть несколько одинаковых элементов, то ранги им присуждаются как среднеарифметические.

Работает в связке с функциями MHL\_RankSelection и MHL\_MakeVectorOfProbabilityForProportionalSelectionV2. Оператор селекции работает с массивом пригодностей индивидов, но непосредственно ранговая селекция выбирает индивида исходя из рангов индивидов, преобразованных в вероятности выбора. Каждый раз для выбора индивида создавать массив вероятностей и рангов затратно, поэтому для каждой популяции на каждом поколении вначале вызывается функция MHL\_MakeVectorOfRankForRankSelection для

генерации вектора рангов, а затем MHL\_MakeVectorOfProbabilityForProportionalSelectionV2 для генерации вектора вероятностей выбора индивида, а затем этот массив и подставляется в ранговую селекцию.

### Примечание:

Под массивом пригодностей понимается специально преобразованный массив значений целевой функции. Процесс подробно описан в стандарте генетического алгоритма. Смотреть здесь. Но это если Вы используете в алгоритмах оптимизации подобных генетическому. а так, если будете использовать, то учитывайте, что массив пригодностей — это массив вещественных чисел из отрезка [0; 1].

Код 131. Пример использования

```
int i;
int VMHL_N=7; //Размер массива (число строк)
double *Fitness;
Fitness=new double[VMHL_N];
for (i=0;i<VMHL_N;i++)
    Fitness[i]=MHL_RandomUniformInt(1,10)/10.;

double *Rank;
Rank=new double[VMHL_N];

//Вызов функции
MHL_MakeVectorOfRankForRankSelection(Fitness,Rank,VMHL_N);

//Используем полученный результат

MHL_ShowVector (Fitness,VMHL_N,"Массив пригодностей", "Fitness");
//Массив пригодностей:
//Fitness =
//0.3
//0.5
//0.7
//0.5
//0.8
//0.1
//0.6

MHL_ShowVector (Rank,VMHL_N,"Массив рангов", "Rank");
//Массив рангов:
//Rank =
//2
//3.5
//6
//3.5
//7
//1
//5

delete [] Fitness;
delete [] Rank;
```

## 7.2.5 MHL\_MakeVectorOfRankZeroForRankSelection

Проставляет ранги для элементов не сортированного массива, то есть номера, начиная с 0 (а не 1), в отсортированном массиве. Если в массиве есть несколько одинаковых элементов, то ранги им присуждаются как среднеарифметические.

Код 132. Синтаксис

```
void MHL_MakeVectorOfRankZeroForRankSelection(double *Fitness, double *
VMHL_ResultVector, int VMHL_N);
```

### Входные параметры:

Fitness — массив пригодностей (можно подавать не массив пригодностей, а массив значений целевой функции, но только для задач безусловной оптимизации);

VMHL\_ResultVector — массив рангов, который мы и формируем;

VMHL\_N — размер массива пригодностей.

### Возвращаемое значение:

Отсутствует.

### О функции:

Это модифицированная функция. Оригинальная функция MHL\_MakeVectorOfRankForRankSelection проставляет ранги с 1.

### Примечание:

Под массивом пригодностей понимается специально преобразованный массив значений целевой функции. Процесс подробно описан в стандарте генетического алгоритма. Смотреть здесь. Но это если Вы используете в алгоритмах оптимизации подобных генетическому, а так, если будете использовать, то учитывайте, что массив пригодностей — это массив вещественных чисел из отрезка [0; 1].

Код 133. Пример использования

```
int i;
int VMHL_N=7; //Размер массива (число строк)
double *Fitness;
Fitness=new double[VMHL_N];
for (i=0;i<VMHL_N;i++)
    Fitness[i]=MHL_RandomUniformInt(1,10)/10.;

double *Rank;
Rank=new double[VMHL_N];

//Вызов функции
MHL_MakeVectorOfRankZeroForRankSelection(Fitness,Rank,VMHL_N);

//Используем полученный результат

MHL_ShowVector (Fitness,VMHL_N,"Массив пригодностей", "Fitness");
//Массив пригодностей:
//Fitness =
//0.3
//0.8
//0.2
```

```

//0.9
//0.1
//0.9
//0.4

MHL_ShowVector (Rank, VMHL_N, "Массив рангов", "Rank");
//Массив рангов:
//Rank =
//2
//4
//1
//5.5
//0
//5.5
//3

delete [] Fitness;
delete [] Rank;

```

## 7.2.6 MHL\_NormalizationVectorAll

Нормировка вектора чисел в отрезок  $[0; 1]$  посредством функции MHL\_NormalizationNumberAll.

Код 134. Синтаксис

```
void MHL_NormalizationVectorAll(double *x, int VMHL_N);
```

### Входные параметры:

VMHL\_ResultVector — указатель на вектор (одномерный массив);

VMHL\_N — размер массива.

### Возвращаемое значение:

Отсутствует.

### Примечание:

Не используйте эту функцию над векторами целых типов int, long. Вектор обнуляется кроме одно какого-нибудь элемента, так как нормировка вектора предполагает числа из интервала  $[0; 1]$ .

Код 135. Пример использования

```

int VMHL_N=10; //Размер массива (число строк)
double *a;
a=new double[VMHL_N];
//Заполним случайными числами
MHL_RandomRealVector(a, -100, 100, VMHL_N);

MHL_ShowVector (a, VMHL_N, "Случайный вектор", "a");
//Случайный вектор:
//a =
// -41.1987
// 81.2317
// -64.386

```

```

//58.4839
//78.5706
//11.8958
//52.179
//47.5952
//−98.4924
//−47.6013

//Вызов функции
MHL_NormalizationVectorAll(a,VMHL_N);

//Используем полученный результат
MHL_ShowVector (a,VMHL_N,"Нормализованный вектор", "а");
// Нормализованный вектор:
//а =
//0.0118487
//0.99392
//0.0076469
//0.991594
//0.993716
//0.961228
//0.990598
//0.989711
//0.00502551
//0.0102878

delete [] a;

```

### 7.2.7 MHL\_NormalizationVectorMaxMin

Нормировка вектора чисел так, чтобы максимальный элемент имел значение 1, а минимальный 0.

Код 136. Синтаксис

```
void MHL_NormalizationVectorMaxMin(double *VMHL_ResultVector,int VMHL_N);
```

**Входные параметры:**

VMHL\_ResultVector — указатель на вектор (одномерный массив);

VMHL\_N — размер массива.

**Возвращаемое значение:**

Отсутствует.

**Примечание:**

Не используйте эту функцию над векторами целых типов int, long. Вектор обнуляется кроме одно какого-нибудь элемента, так как нормировка вектора предполагает числа из интервала [0; 1].

Код 137. Пример использования

```

int i;
int VMHL_N=10; //Размер массива (число строк)
double *a;
```

```

a=new double[VMHL_N];
//Заполним случайными числами
for (i=0;i<VMHL_N;i++)
a[i]=MHL_RandomUniformInt(-100,100);

MHL_ShowVector (a,VMHL_N,"Случайный вектор", "a");
//Случайный вектор:
//a =
//38
// -35
// -59
// -15
// -98
// 24
// 83
// -16
// 73
// 45

//Вызов функции
MHL_NormalizationVectorMaxMin(a,VMHL_N);

//Используем полученный результат
MHL_ShowVector (a,VMHL_N,"Нормализованный вектор", "a");
//Нормализованный вектор:
//a =
//0.751381
//0.348066
//0.21547
//0.458564
//0
//0.674033
//1
//0.453039
//0.944751
//0.790055

delete [] a;

```

### 7.2.8 MHL\_NormalizationVectorOne

Нормировка вектора чисел в отрезок  $[0, 1]$  так, чтобы сумма всех элементов была равна 1.

Код 138. Синтаксис

```
void MHL_NormalizationVectorOne(double *VMHL_ResultVector,int VMHL_N);
```

**Входные параметры:**

VMHL\_ResultVector — указатель на вектор (одномерный массив), который и будет преобразовываться;

VMHL\_N — размер массива.

**Возвращаемое значение:**

Отсутствует.

### Примечание:

Не используйте эту функцию над векторами целых типов int, long. Вектор обнуляется кроме одно какого-нибудь элемента, так как нормировка вектора предполагает числа из интервала [0; 1].

Код 139. Пример использования

```
int i;
int VMHL_N=10; //Размер массива (число строк)
double *a;
a=new double[VMHL_N];
//Заполним случайными числами
for (i=0;i<VMHL_N;i++)
    a[i]=MHL_RandomUniformInt(-100,100);

MHL_ShowVector (a,VMHL_N, "Случайный вектор", "a");
//Случайный вектор:
//a =
//-76
//-100
/-2
//97
//99
//49
//87
// -49
// -13
//7

//Вызов функции
MHL_NormalizationVectorOne(a,VMHL_N);

//Используем полученный результат
MHL_ShowVector (a,VMHL_N, "Нормализованный вектор", "a");
//Нормализованный вектор:
//a =
//0.021838
//0
//0.089172
//0.179254
//0.181074
//0.135578
//0.170155
//0.0464058
//0.0791629
//0.0973612

MHL_ShowNumber (TMHL_SumVector(a,VMHL_N), "Его сумма", "Sum");
//Его сумма:
//Sum=1

delete [] a;
```

### 7.2.9 MHL\_ProbabilityOfTournamentSelection

Функция вычисляет вероятности выбора индивидов из популяции с помощью турнирной селекции..

#### Код 140. Синтаксис

```
double MHL_ProbabilityOfTournamentSelection(double *Fitness, double *
VMHL_ResultVector_Probability, int T, int VMHL_N);
```

#### Входные параметры:

Fitness — указатель на вектор значений целевой функции (не пригодности) индивидов;

VMHL\_ResultVector\_Probability — указатель на вектор, в который будет проводиться запись;

T — размер турнира;

VMHL\_N — размер массивов.

#### Возвращаемое значение:

Сумму вектора вероятностей Probability.

#### Примечание:

Данная функция не нужна для работы турнирной селекции через функцию MHL\_TournamentSelection в генетическом алгоритме. Функция предназначена для научных изысканий по исследованию работы различных видов селекций.

#### Формула:

$$P(X_j) = \frac{\sum_{j=\max(1, T-(N-n_1-n_0))}^{\min(T, n_1)} C_{n_1}^j \cdot C_{N-n_1-n_0}^{T-j}}{n_1 \cdot C_N^T}, \text{ где}$$
$$n_0 = \sum_{j=1}^N S_0(X_j), S_0(X_j) = \begin{cases} 1, & \text{если } f(X_j) > f(X_i); \\ 0, & \text{если } f(X_j) \leq f(X_i). \end{cases}$$
$$n_1 = \sum_{j=1}^N S_1(X_j), S_1(X_j) = \begin{cases} 1, & \text{если } f(X_j) = f(X_i); \\ 0, & \text{если } f(X_j) \neq f(X_i). \end{cases}$$

#### Код 141. Пример использования

```
int i;
int VMHL_N=10; //размер популяции
double *f=new double[VMHL_N]; //массив значений целевой функции
double *p=new double[VMHL_N]; //массив значений вероятностей выбора индивидов
int T=3; // размер турнира

for (i=0;i<VMHL_N;i++) f[i]=MHL_RandomUniformInt(0,11)/10.; //заполним случайными значениями целевой функции

MHL_ShowVector (f,VMHL_N,"Вектор значений целевой функции", "f");
//Вектор значений целевой функции:
//f =
//0.9
//0.4
//0.4
//0.5
//1
//0.8
```

```

//0.5
//0.8
//0.8
//0.4

//Вызов функции
double sum = MHL_ProbabilityOfTournamentSelection(f, p, T, VMHL_N);

//Используем результат
MHL_ShowVector (p, VMHL_N, "Вектор значений вероятностей выбора", "p");
//Вектор значений вероятностей выбора:
//p =
//0.233333
//0.00277778
//0.00277778
//0.0375
//0.3
//0.127778
//0.0375
//0.127778
//0.127778
//0.00277778

MHL_ShowNumber(sum, "Сумма вектора значений вероятностей выбора", "sum");
//Сумма вектора значений вероятностей выбора:
//sum=1

delete[] f;
delete[] p;

```

## 7.2.10 MHL\_ProportionalSelection

Пропорциональная селекция. Оператор генетического алгоритма. Работает с массивом пригодностей.

Код 142. Синтаксис

```
int MHL_ProportionalSelection(double *Fitness, int VMHL_N);
```

**Входные параметры:**

Fitness — массив пригодностей (можно подавать не массив пригодностей, а массив значений целевой функции, но только для задач безусловной оптимизации);

VMHL\_N — размер массива пригодностей.

**Возвращаемое значение:**

Номер выбранной пригодности, а, соответственно, номер индивида популяции.

**Принцип работы:**

**Примечание:**

Использовать реализацию оператора ГА в виде этой функции нецелесообразно ввиду того, что при каждом запуске создается дополнительный массив. Данная функция аналогична по действию (результат действия аналогичен):

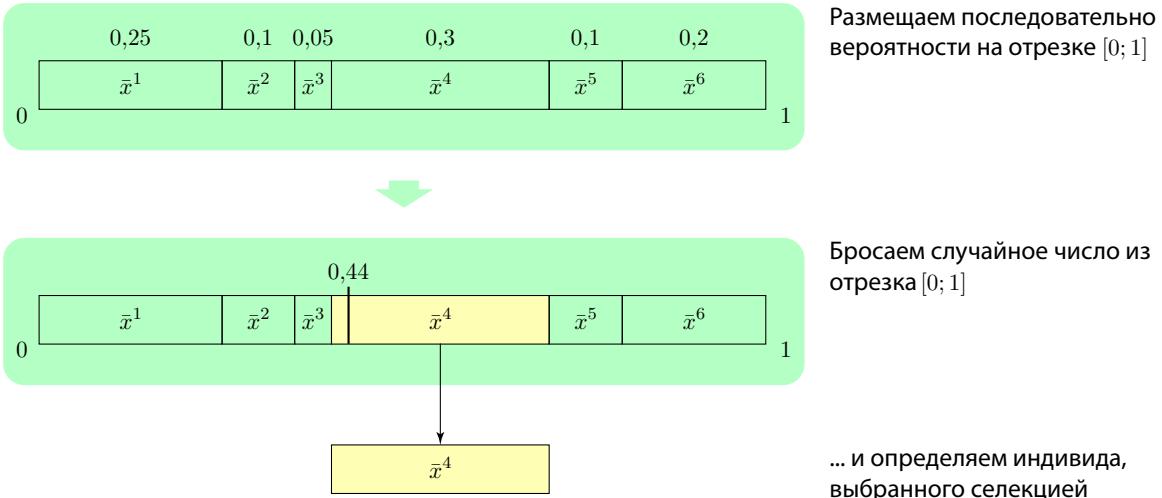


Рисунок 1. Механизм работы пропорциональной селекции

- Связке функций `MHL_MakeVectorOfProbabilityForProportionalSelectionV2` и `MHL_ProportionalSelectionV2`;
- Функции `MHL_ProportionalSelectionV3`.

Различия по временным затратам на выполнение. У этой реализации самое большое время выполнения.

#### Примечание:

Под массивом пригодностей понимается специально преобразованный массив значений целевой функции. Процесс подробно описан в стандарте генетического алгоритма. Смотреть здесь. Но это если Вы используете в алгоритмах оптимизации подобных генетическому. а так, если будете использовать, то учитывайте, что массив пригодностей — это массив вещественных чисел из отрезка [0; 1].

Код 143. Пример использования

```

int i;
int VMHL_N=10; //Размер массива (число строк)
double *Fitness;
Fitness=new double[VMHL_N];
//Заполним вектор случайными значениями пригодностей индивидов
//на практике, конечно, пригодности вычисляются, например, в
//процессе работы ГА
for (i=0;i<VMHL_N;i++) Fitness[i]=MHL_RandomNumber();

//Вызов функции
int Number=MHL_ProportionalSelection(Fitness,VMHL_N);

//Используем полученный результат

//Например:
MHL_ShowVector (Fitness,VMHL_N,"Вектор пригодностей индивидов", "a");
// Вектор пригодностей индивидов:
//a =
//0.368073
//0.474609
//0.297089
//0.373474

```

```

//0.102203
//0.774292
//0.487335
//0.747742
//0.505646
//0.901184

MHL_ShowNumber (Number, "Номер выбранного индивида", "Number");
//Номер выбранного индивида:
//Number=5

delete [] Fitness;

```

### 7.2.11 MHL\_ProportionalSelectionV2

Пропорциональная селекция. Оператор генетического алгоритма. Работает с вектором вероятностей выбора индивидов, который можно получить из вектора пригодностей индивидов посредством функции MHL\_MakeVectorOfProbabilityForProportionalSelectionV2.

Код 144. Синтаксис

```
int MHL_ProportionalSelectionV2(double *VectorOfProbability, int VMHL_N);
```

**Входные параметры:**

VectorOfProbability — массив вероятностей выбора индивидов для порциональной селекции;

VMHL\_N — размер массива пригодностей.

**Возвращаемое значение:**

Номер выбранной пригодности, а, соответственно, номер индивида популяции.

**Принцип работы:**

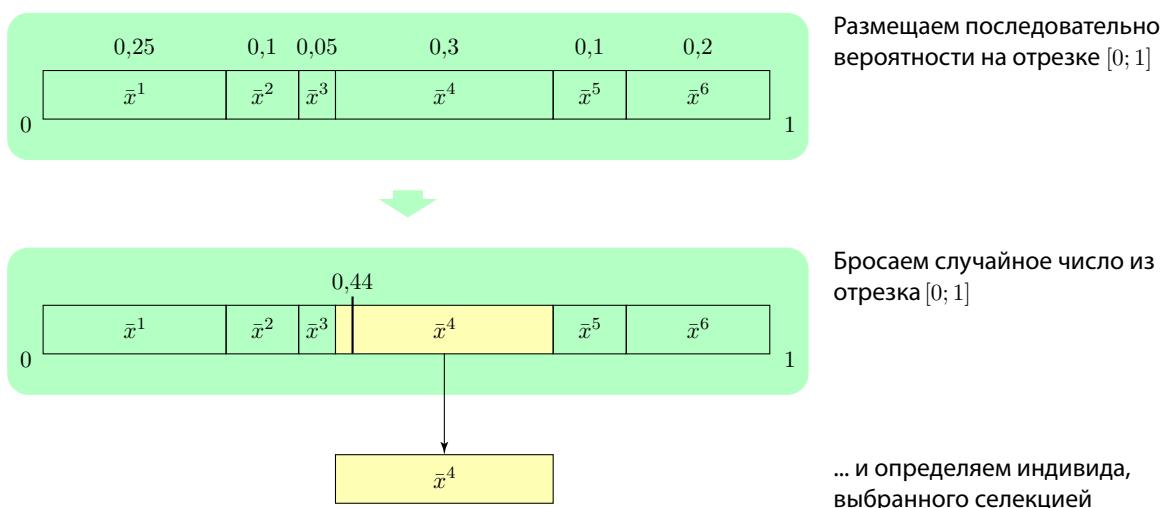


Рисунок 2. Механизм работы пропорциональной селекции

**Примечание:**

Связка данной функции и MHL\_MakeVectorOfProbabilityForProportionalSelectionV2 аналогична по действию (результат действия аналогичен):

1. Функции MHL\_ProportionalSelection;
2. Функции MHL\_ProportionalSelectionV3.

Различия по временным затратам на выполнение. У этой связки выполнение быстрее, чем у MHL\_ProportionalSelection.

### О функции:

Данная функция используется в стандартном генетическом алгоритме, реализованным в виде функции MHL\_StandartGeneticAlgorithm. Работает в связке с функцией MHL\_MakeVectorOfProbabilityForProportionalSelectionV2. Оператор селекции работает с массивом пригодностей индивидов, но непосредственно пропорциональная селекция выбирает индивида исходя из вероятностей выбора индивидов. Каждый раз для выбора индивида создавать массив вероятностей затратно, поэтому для каждой популяции на каждом поколении вначале вызывается функция MHL\_MakeVectorProbabilityForSelectionProportionalV2 для генерации вектора вероятностей выбора индивида, а затем этот массив и подставляется в пропорциональную селекцию.

Код 145. Пример использования

```
int i;
int VMHL_N=10; //Размер массива (число строк)
double *Fitness;
Fitness=new double[VMHL_N];
//Заполним вектор случайными значениями пригодностей индивидов
//на практике, конечно, пригодности вычисляются, например, в
//процессе работы ГА
for (i=0;i<VMHL_N; i++) Fitness[i]=MHL_RandomNumber();

//Для работы этого варианта пропорциональной селекции нужен
//массив вероятностей выбора индивидов для порпциональной селекции;
double *VectorProbability;
VectorProbability=new double[VMHL_N];
//Сформируем этот массив
MHL_MakeVectorOfProbabilityForProportionalSelectionV2(Fitness,VectorProbability,
VMHL_N);

//Вызов функции
int Number=MHL_ProportionalSelectionV2(VectorProbability,VMHL_N);

//Используем полученный результат
MHL_ShowVector (Fitness,VMHL_N,"Вектор пригодностей индивидов", "a");
// Вектор пригодностей индивидов:
//a =
//0.681061
//0.629517
//0.697021
//0.140045
//0.221649
//0.203461
//0.702576
//0.998077
//0.853607
//0.19928
```

```

MHL_ShowVector (VectorProbability, VMHL_N, "Вектор вероятностей выбора индивидов", "VectorProbability");
// Вектор вероятностей выбора индивидов:
//VectorProbability =
//0.137809
//0.124679
//0.141874
//0
//0.0207864
//0.0161534
//0.143289
//0.21856
//0.18176
//0.0150884

MHL_ShowNumber (TMHL_SumVector(VectorProbability, VMHL_N), "Его сумма", "Sum");
// Его сумма:
//Sum=1

MHL_ShowNumber (Number, "Номер выбранного индивида", "Number");

// Номер выбранного индивида:
//Number=6

delete [] Fitness;
delete [] VectorProbability;

```

### 7.2.12 MHL\_ProportionalSelectionV3

Пропорциональная селекция. Оператор генетического алгоритма. Работает с массивом пригодностей (обязательно не отрицательными).

Код 146. Синтаксис

```
int MHL_ProportionalSelectionV3(double *Fitness, int VMHL_N);
```

**Входные параметры:**

Fitness — массив пригодностей (В отличии от MHL\_ProportionalSelection вектор пригодностей должен быть именно вектором пригодностей, то есть все элементы Fitness должны быть больше нуля);

VMHL\_N — размер массива пригодностей.

**Возвращаемое значение:**

Номер выбранной пригодности, а, соответственно, номер индивида популяции.

**Принцип работы:**

**Примечание:**

Данная функция аналогична по действию (результат действия аналогичен):

1. Связке функций MHL\_MakeVectorOfProbabilityForProportionalSelectionV2 и MHL\_ProportionalSelectionV2;
2. Функции MHL\_ProportionalSelection.

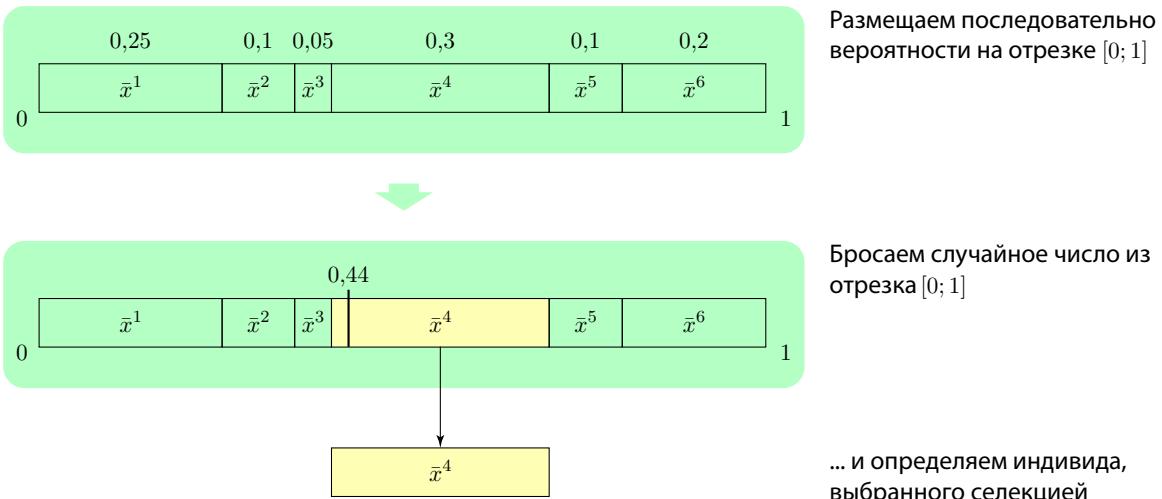


Рисунок 3. Механизм работы пропорциональной селекции

Различия по времененным затратам на выполнение. Эта реализация быстрее, чем `MHL_SelectionProportional` и почти равна связке функций `MHL_MakeVectorOfProbabilityForProportionalSelectionV2` и `MHL_ProportionalSelectionV2`, но реализация отличается от формульной записи в угоду более простой записи в программировании, но ей тождественна.

#### Примечание:

Под массивом пригодностей понимается специально преобразованный массив значений целевой функции. Процесс подробно описан в стандарте генетического алгоритма. Смотреть здесь. Но это если Вы используете в алгоритмах оптимизации подобных генетическому. а так, если будете использовать, то учитывайте, что массив пригодностей — это массив вещественных чисел из отрезка  $[0; 1]$ .

Код 147. Пример использования

```

int i;
int VMHL_N=10; //Размер массива (число строк)
double *Fitness;
Fitness=new double[VMHL_N];
//Заполним вектор случайными значениями пригодностей индивидов
//на практике, конечно, пригодности вычисляются, например, в
//процессе работы ГА
for (i=0;i<VMHL_N;i++) Fitness[i]=MHL_RandomNumber();

//Вызов функции
int Number=MHL_ProportionalSelectionV3(Fitness,VMHL_N);

//Используем полученный результат
MHL_ShowVector (Fitness,VMHL_N,"Вектор пригодностей индивидов", "a");
// Вектор пригодностей индивидов:
//a =
//0.774231
//0.15918
//0.671448
//0.0546265
//0.881012
//0.766541
//0.638275
//0.0705261

```

```

//0.234528
//0.0510559

MHL_ShowNumber (Number, "Номер выбранного индивида", "Number");
// Номер выбранного индивида:
//Number=0

delete [] Fitness;

```

### 7.2.13 MHL\_RankSelection

Ранговая селекция. Оператор генетического алгоритма. Работает с вектором вероятностей выбора индивидов, который можно получить из вектора пригодностей индивидов посредством функции *MHL\_MakeVectorOfRankForRankSelection* (для получения массива рангов) и потом функции *MHL\_MakeVectorOfProbabilityForProportionalSelectionV2* (для получения массива вероятностей выбора индивидов по рангам).

Код 148. Синтаксис

```
int MHL_RankSelection(double *VectorOfProbability, int VMHL_N);
```

**Входные параметры:**

VectorOfProbability — массив вероятностей выбора индивидов для ранговой селекции;

VMHL\_N — размер массива VectorProbability.

**Возвращаемое значение:**

Номер выбранного индивида популяции.

**Принцип работы:**

**О функции:**

Данная функция используется в стандартном генетическом алгоритме, реализованным в виде функции *MHL\_StandartGeneticAlgorithm*. Работает в связке с функциями *MHL\_MakeVectorOfRankForRankSelection* и *MHL\_MakeVectorOfProbabilityForProportionalSelectionV2*. Оператор селекции работает с массивом пригодностей индивидов, но непосредственно ранговая селекция выбирает индивида исходя из рангов индивидов, преобразованных в вероятности выбора. Каждый раз для выбора индивида создавать массив вероятностей и рангов затратно, поэтому для каждой популяции на каждом поколении вначале вызывается функция *MHL\_MakeVectorOfRankForRankSelection* для генерации вектора рангов, а затем *MHL\_MakeVectorOfProbabilityForProportionalSelectionV2* для генерации вектора вероятностей выбора индивида, а затем этот массив и подставляется в ранговую селекцию.

**Примечание:**

На рисунке показано, что ранги подаются в пропорциональную селекцию, но из кода этого не видно. Но по своей сути код данной функции повторяет код функции *MHL\_ProportionalSelectionV2* и также требует вектор вероятностей выбора. Так что всё соответствует рисунку.

Код 149. Пример использования

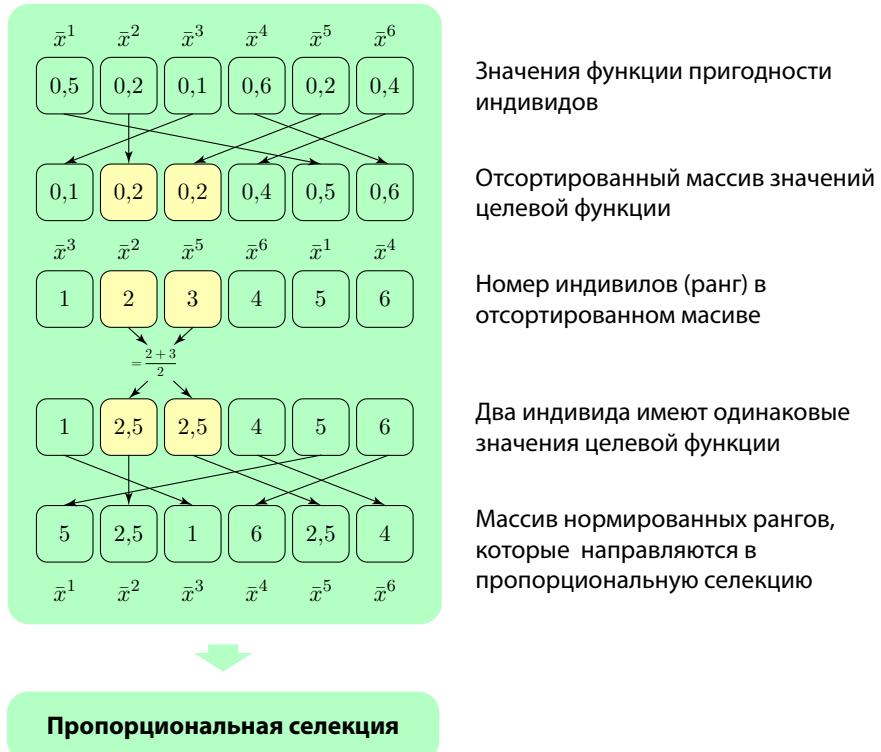


Рисунок 4. Механизм работы ранговой селекции

```

int i;
int VMHL_N=7; //Размер массива
double *Fitness;
Fitness=new double[VMHL_N];
for (i=0;i<VMHL_N;i++)
    Fitness[i]=MHL_RandomUniformInt(1,10)/10.;

double *Rank;
Rank=new double[VMHL_N];

double *VectorProbability;
VectorProbability=new double[VMHL_N];

//Сформируем вектор рангов
MHL_MakeVectorOfRankForRankSelection(Fitness,Rank,VMHL_N);
//Из вектора рангов получим вектор вероятностей выбора
MHL_MakeVectorOfProbabilityForProportionalSelectionV2(Rank,VectorProbability,VMHL_N);

//Вызов функции
int Number=MHL_RankSelection(VectorProbability,VMHL_N);

//Используем полученный результат

MHL_ShowVector (Fitness,VMHL_N,"Массив пригодностей", "Fitness");
// Массив пригодностей:
//Fitness =
//0.2
//0.2
//0.6
//0.8
//0.4
//0.3

```

```

//0.2

MHL_ShowVector (Rank, VMHL_N, "Массив рангов", "Rank");
// Массив рангов:
//Rank =
//2
//2
//6
//7
//5
//4
//4
//2

MHL_ShowVector (VectorProbability, VMHL_N, "Массив вероятностей выбора", "VectorProbability");
//Массив вероятностей выбора:
//VectorProbability =
//0
//0
//0.285714
//0.357143
//0.214286
//0.142857
//0

MHL_ShowNumber (Number, "Номер выбранного индивида", "Number");
// Номер выбранного индивида:
//Number=3

delete [] Fitness;
delete [] Rank;
delete [] VectorProbability;

```

### 7.2.14 MHL\_SelectItemOnProbability

Функция выбирает случайно номер элемента из вектора, где вероятность выбора каждого элемента определяется значением в векторе Р.

Код 150. Синтаксис

```
int MHL_SelectItemOnProbability(double *P, int VMHL_N);
```

**Входные параметры:**

P — вектор вероятностей выбора каждого элемента, то есть его компоненты должны быть из отрезка [0; 1], а сумма их равна 1;

VMHL\_N — размер вектора.

**Возвращаемое значение:**

Номер выбранного элемента.

**Примечание:**

Проверка на правильность вектора Р не проводится, так как функция обычно вызывается многократно, а проводить постоянно проверку накладно. Всё на Вашей совести.

Код 151. Пример использования

```
int VMHL_N=10; //Размер массива (число строк)
double *a;
a=new double[VMHL_N];
//Заполним вектор случайными значениями вероятностей
MHL_RandomVectorOfProbability(a, VMHL_N);

//Вызов функции
int Number=MHL_SelectItemOnProbability(a,VMHL_N);

//Используем полученный результат
MHL_ShowVector (a,VMHL_N,"Вектор вероятностей выбора", "a");
// Вектор вероятностей выбора:
//Вектор вероятностей выбора:
//a =
//0.0701006
//0.190423
//0.0231631
//0.160255
//0.0983935
//0.038739
//0.166252
//0.105259
//0.0621408
//0.0852747

MHL_ShowNumber (Number,"Номер выбранного элемента", "Number");
// Номер выбранного элемента:
//Number=6

delete [] a;
```

### 7.2.15 MHL\_StandartBinaryGeneticAlgorithm

Стандартный генетический алгоритм для решения задач на бинарных строках. Реализация алгоритма из документа «Генетический алгоритм. Стандарт. v.3.0».

Код 152. Синтаксис

```
int MHL_StandartBinaryGeneticAlgorithm(int *Parameters, double (*FitnessFunction)(int *,int), int *VMHL_ResultVector, double *VMHL_Result);
```

#### Входные параметры:

Parameters — Вектор параметров генетического алгоритма. Каждый элемент обозначает свой параметр:

- 0 — длина бинарной хромосомы (определяется задачей оптимизации, что мы решаем);
- 1 — число вычислений целевой функции (CountOfFitness);
- 2 — тип селекции (TypeOfSel):
  - 0 — ProportionalSelection (Пропорциональная селекция);
  - 1 — RankSelection (Ранговая селекция);
  - 2 — TournamentSelection (Турнирная селекция).

3 — тип скрещивания (TypeOfCros):

- 0 — SinglepointCrossover (Одноточечное скрещивание);
- 1 — TwopointCrossover (Двухточечное скрещивание);
- 2 — UniformCrossover (Равномерное скрещивание).

4 — тип мутации (TypeOfMutation):

- 0 — Weak (Слабая мутация);
- 1 — Average (Средняя мутация);
- 2 — Strong (Сильная мутация).

5 — тип формирования нового поколения (TypeOfForm):

- 0 — OnlyOffspringGenerationForming (Только потомки);
- 1 — OnlyOffspringWithBestGenerationForming (Только потомки и копия лучшего индивида).

FitnessFunction — указатель на целевую функцию (если решается задача условной оптимизации, то учет ограничений должен быть включен в эту функцию);

VMHL\_ResultVector — найденное решение (бинарный вектор);

VMHL\_Result — значение целевой функции в точке, определенной вектором VMHL\_ResultVector.

#### **Возвращаемое значение:**

1 — завершил работу без ошибок. Всё хорошо.

0 — возникли при работе ошибки. Скорее всего в этом случае в VMHL\_ResultVector и в VMHL\_Result не содержится решение задачи.

**Принцип работы** смотрите ниже на рисунке 5 на странице 109.

#### **О функции:**

Реализация алгоритма из документа «Генетический алгоритм. Стандарт. v.3.0».

<https://github.com/Harrix/Standard-Genetic-Algorithm>

Алгоритм бинарной оптимизации. Ищет максимум целевой функции FitnessFunction.

Решением является бинарная строка, то есть вектор, состоящий из 0 и 1.

**Примерный настройки** (для примера Вы можете поставить такие рабочие настройки):

Parameters[0]=50;

Parameters[1]=100\*100;

Parameters[2]=2;

Parameters[3]=2;

Parameters[4]=1;

Parameters[5]=1;

#### **Примечание:**

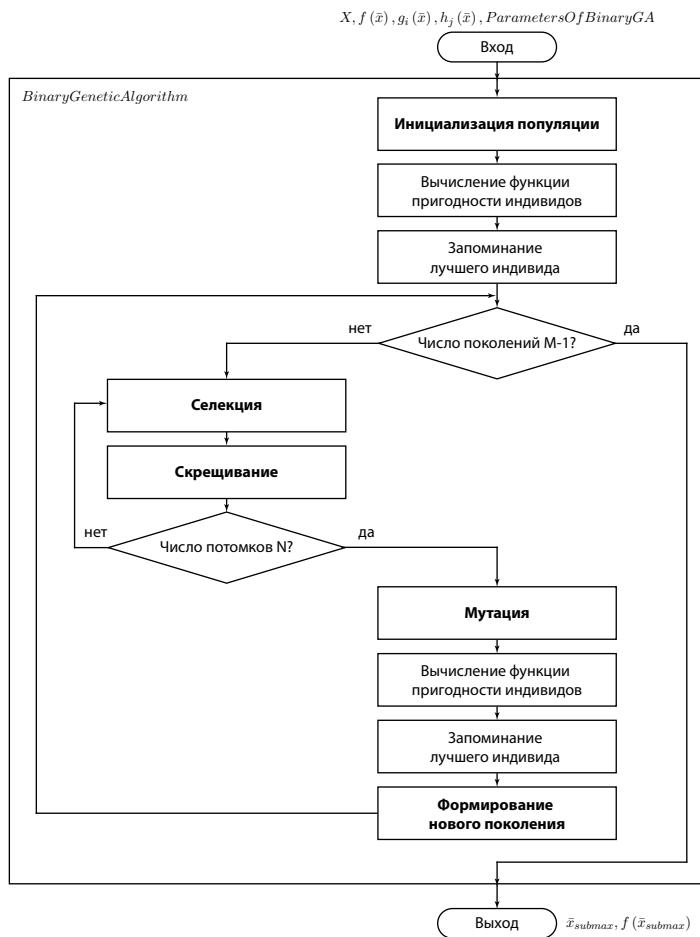


Рисунок 5. Механизм работы генетического алгоритма

На рисунке блок-схемы сГА на бинарных строках число поколений обозначено буквой  **$M$** , в коде функции же обозначается переменной **NumberOfGenerations**.

#### Примечание:

В сГА на бинарных строках не нужно задавать в параметрах число поколений и размер популяции, а только число вычислений целевой функции. Почему? Алгоритм сам определит число поколений и размер популяции, исходя из принципа, что число поколений и размер популяции должны быть примерно равны. Поэтому выбирайте значение **Parameters[1]** в виде:

```
int K=100;
```

```
Parameters[1]=K*K;
```

То есть в виде квадрата целого числа. В противном случае реальное число вычислений целевой функции и значение **Parameters[1]** будут не совпадать.

Код целевой функции:

Код 153. Оптимизируемая функция

```
double Func(int *x, int VMHL_N)
{
    //Сумма всех элементов массива
    return TMHL_SumVector(x, VMHL_N);
}
```

### Код 154. Пример использования

## 7.2.16 MHL StandartGeneticAlgorithm

Стандартный генетический алгоритм для решения задач на бинарных и вещественных строках. Реализация алгоритма из документа «Генетический алгоритм. Стандарт. v.3.0».

## Код 155. Синтаксис

```

int MHL_StandartGeneticAlgorithm(int *Parameters, int *NumberOfParts, double *Left,
    double *Right, double (*FitnessFunction)(double*, int), double *VMHL_ResultVector,
    double *VMHL_Result);
int MHL_StandartGeneticAlgorithm(int *Parameters, double (*FitnessFunction)(int*, int),
    , int *VMHL_ResultVector, double *VMHL_Result);

```

Функция и функция перегрузка вызывают функции MHL\_StandartBinaryGeneticAlgorithm и MHL\_StandartRealGeneticAlgorithm.

### Принцип работы:

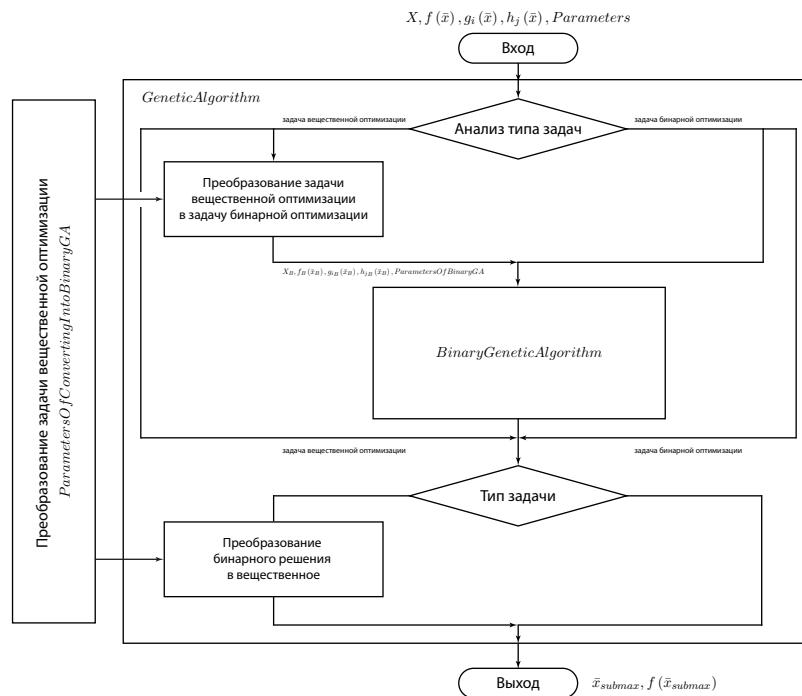


Рисунок 6. Механизм работы генетического алгоритма

### Входные параметры:

Parameters — Вектор параметров генетического алгоритма. Каждый элемент обозначает свой параметр:

- 0 — длина бинарной хромосомы (определяется задачей оптимизации, что мы решаем);
- 1 — число вычислений целевой функции (CountOfFitness);
- 2 — тип селекции (TypeOfSel):
  - 0 — ProportionalSelection (Пропорциональная селекция);
  - 1 — RankSelection (Ранговая селекция);
  - 2 — TournamentSelection (Турнирная селекция).
- 3 — тип скрещивания (TypeOfCros):
  - 0 — SinglepointCrossover (Одноточечное скрещивание);
  - 1 — TwopointCrossover (Двухточечное скрещивание);
  - 2 — UniformCrossover (Равномерное скрещивание).

4 — тип мутации (TypeOfMutation):

- 0 — Weak (Слабая мутация);
- 1 — Average (Средняя мутация);
- 2 — Strong (Сильная мутация).

5 — тип формирования нового поколения (TypeOfForm):

- 0 — OnlyOffspringGenerationForming (Только потомки);
- 1 — OnlyOffspringWithBestGenerationForming (Только потомки и копия лучшего индивида).

FitnessFunction — указатель на целевую функцию (если решается задача условной оптимизации, то учет ограничений должен быть включен в эту функцию);

VMHL\_ResultVector — найденное решение (бинарный вектор);

VMHL\_Result — значение целевой функции в точке, определенной вектором VMHL\_ResultVector.

#### **Возвращаемое значение:**

1 — завершил работу без ошибок. Всё хорошо.

0 — возникли при работе ошибки. Скорее всего в этом случае в VMHL\_ResultVector и в VMHL\_Result не содержится решение задачи.

**Принцип работы** смотрите ниже на рисунке 5 на странице 109.

#### **О функции:**

Реализация алгоритма из документа «Генетический алгоритм. Стандарт. v.3.0».

<https://github.com/Harrix/Standard-Genetic-Algorithm>

Алгоритм бинарной оптимизации. Ищет максимум целевой функции FitnessFunction.

Решением является бинарная строка, то есть вектор, состоящий из 0 и 1.

**Примерный настройки** (для примера Вы можете поставить такие рабочие настройки):

Parameters[0]=50;

Parameters[1]=100\*100;

Parameters[2]=2;

Parameters[3]=2;

Parameters[4]=1;

Parameters[5]=1;

#### **Примечание:**

На рисунке блок-схемы сГА на бинарных строках число поколений обозначено буквой **M**, в коде функции же обозначается переменной **NumberOfGenerations**.

#### **Примечание:**

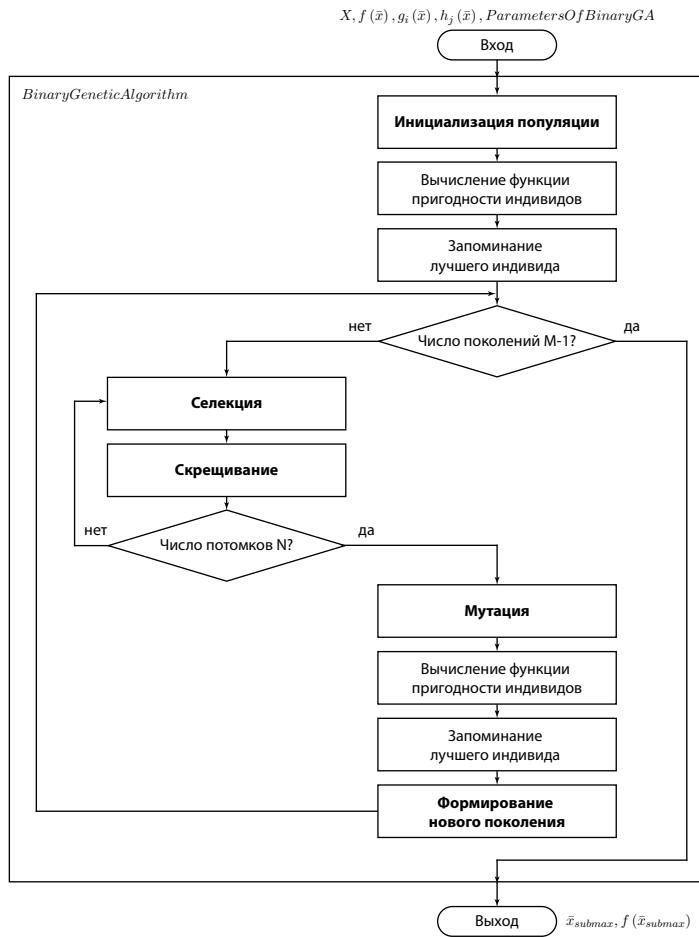


Рисунок 7. Механизм работы генетического алгоритма

В сГА на бинарных строках не нужно задавать в параметрах число поколений и размер популяции, а только число вычислений целевой функции. Почему? Алгоритм сам определит число поколений и размер популяции, исходя из принципа, что число поколений и размер популяции должны быть примерно равны. Поэтому выбирайте значение `Parameters[1]` в виде:

```
int K=100;
```

```
Parameters[1]=K*K;
```

То есть в виде квадрата целого числа. В противном случае реальное число вычислений целевой функции и значение `Parameters[1]` будут не совпадать.

Код целевой функции:

Код 156. Оптимизируемая функция

```
double Func(int *x, int VMHL_N)
{
//Сумма всех элементов массива
return TMHL_SumVector(x, VMHL_N);
}
//-----
```

**Для переопределенной функции.**

**Входные параметры:**

Parameters — Вектор параметров генетического алгоритма. Каждый элемент обозначает свой параметр:

- 0 — длина вещественной хромосомы (определяется задачей оптимизации, что мы решаем);
- 1 — число вычислений целевой функции (CountOfFitness);
- 2 — тип селекции (TypeOfSel):
  - 0 — ProportionalSelection (Пропорциональная селекция);
  - 1 — RankSelection (Ранговая селекция);
  - 2 — TournamentSelection (Турнирная селекция).
- 3 — тип скрещивания (TypeOfCros):
  - 0 — SinglepointCrossover (Одноточечное скрещивание);
  - 1 — TwopointCrossover (Двухточечное скрещивание);
  - 2 — UniformCrossover (Равномерное скрещивание).
- 4 — тип мутации (TypeOfMutation):
  - 0 — Weak (Слабая мутация);
  - 1 — Average (Средняя мутация);
  - 2 — Strong (Сильная мутация).
- 5 — тип формирования нового поколения (TypeOfForm):
  - 0 — OnlyOffspringGenerationForming (Только потомки);
  - 1 — OnlyOffspringWithBestGenerationForming (Только потомки и копия лучшего индивида).
- 6 — тип преобразования задачи вещественной оптимизации в задачу бинарной оптимизации (TypeOfConverting):
  - 0 — IntConverting (Стандартное представление целого числа — номер узла в сетке дискретизации);
  - 1 — GrayCodeConverting (Стандартный рефлексивный Грей-код).

NumberOfParts — указатель на массив: на сколько частей делить каждую вещественную координату при дискретизации (размерность Parameters[0]);

Желательно брать по формуле  $NumberOfParts[i] = 2^k - 1$ , где  $k$  — натуральное число, например, 12.

Left — массив левых границ изменения каждой вещественной координаты (размерность Parameters[0]);

Right — массив правых границ изменения каждой вещественной координаты (размерность Parameters[0]);

FitnessFunction — указатель на целевую функцию (если решается задача условной оптимизации, то учет ограничений должен быть включен в эту функцию);

VMHL\_ResultVector — найденное решение (вещественный вектор);

`VMHL_Result` — значение целевой функции в точке, определенной вектором `VMHL_ResultVector`.

### **Возвращаемое значение:**

1 — завершил работу без ошибок. Всё хорошо.

0 — возникли при работе ошибки. Скорее всего в этом случае в `VMHL_ResultVector` и в `VMHL_Result` не содержится решение задачи.

### **О функции:**

Реализация алгоритма из документа «Генетический алгоритм. Стандарт. v.3.0».

<https://github.com/Harrix/Standard-Genetic-Algorithm>

Алгоритм вещественной оптимизации. Ищет максимум целевой функции `FitnessFunction`.

Решением является вещественная строка.

**Примерный настройки** (для примера Вы можете поставить такие рабочие настройки):

`Parameters[0]=50;`

`Parameters[1]=100*100;`

`Parameters[2]=2;`

`Parameters[3]=2;`

`Parameters[4]=1;`

`Parameters[5]=1;`

`Parameters[5]=0;`

### **Примечание:**

В сГА на вещественных строках не нужно задавать в параметрах число поколений и размер популяции, а только число вычислений целевой функции. Почему? Алгоритм сам определит число поколений и размер популяции, исходя из принципа, что число поколений и размер популяции должны быть примерно равны. Поэтому выбирайте значение `Parameters[1]` в виде:

`int K=100;`

`Parameters[1]=K*K;`

То есть в виде квадрата целого числа. В противном случае реальное число вычислений целевой функции и значение `Parameters[1]` будут не совпадать.

Код целевой функции:

Код 157. Оптимизируемая функция

```
double Func2(double *x, int VMHL_N)
{
    return -((x[0]-2)*(x[0]-2)+(x[1]-2)*(x[1]-2));
}
```

Код 158. Пример использования

```
int ChromosomeLength=50; //Длина хромосомы
```



```

ParametersOfStandartRealGeneticAlgorithm[1]=CountOfFitness; //Число вычислений целевой функции
ParametersOfStandartRealGeneticAlgorithm[2]=TypeOfSel; //Тип селекции
ParametersOfStandartRealGeneticAlgorithm[3]=TypeOfCros; //Тип скрещивания
ParametersOfStandartRealGeneticAlgorithm[4]=TypeOfMutation; //Тип мутации
ParametersOfStandartRealGeneticAlgorithm[5]=TypeOfForm; //Тип формирования нового поколения
ParametersOfStandartRealGeneticAlgorithm[6]=0; //Тип формирования нового поколения

double *Left; //массив левых границ изменения каждой вещественной координаты
Left=new double[ChromosomeLength];
double *Right; //массив правых границ изменения каждой вещественной координаты
Right=new double[ChromosomeLength];
int *NumberOfParts; //на сколько делить каждую координату
NumberOfParts=new int[ChromosomeLength];

//Заполним массивы
//Причем по каждой координате одинаковые значения выставим
TMHL_FillVector(Left,ChromosomeLength,-5.); //Пусть будет интервал [-3;3]
TMHL_FillVector(Right,ChromosomeLength,5.);
TMHL_FillVector(NumberOfParts,ChromosomeLength,TMHL_PowerOf(2,15)-1); //Делим на 32768-1 частей каждую вещественную координату

double *Decision; //вещественное решение
Decision=new double[ChromosomeLength];
double ValueOfFitnessFunction; //значение целевой функции в точке Decision
int VMHL_Success=0; //Успешен ли будет запуск СГА

//Запуск алгоритма
VMHL_Success=MHL_StandartGeneticAlgorithm (
    ParametersOfStandartRealGeneticAlgorithm, NumberOfParts, Left, Right, Func2,
    Decision, &ValueOfFitnessFunction);

//Используем полученный результат
MHL_ShowNumber(VMHL_Success, "Как прошел запуск", "VMHL_Success");
if (VMHL_Success==1)
{
    MHL_ShowVectorT(Decision,ChromosomeLength, "Найденное решение", "Decision");
    //Найденное решение:
    //Decision =
    //2.00348 2.00226
    MHL_ShowNumber(ValueOfFitnessFunction, "Значение целевой функции", "ValueOfFitnessFunction");
    //Значение целевой функции:
    //ValueOfFitnessFunction=-1.72034e-05
}

delete [] ParametersOfStandartRealGeneticAlgorithm;
delete [] Decision;
delete [] Left;
delete [] Right;
delete [] NumberOfParts;
} //чтобы не удалять объявления переменных, заключим в скобки

```

## 7.2.17 MHL\_StandartRealGeneticAlgorithm

Стандартный генетический алгоритм для решения задач на вещественных строках. Реализация алгоритма из документа «Генетический алгоритм. Стандарт. v.3.0».

### Код 159. Синтаксис

```
int MHL_StandartRealGeneticAlgorithm(int *Parameters, int *NumberOfParts, double *
Left, double *Right, double (*FitnessFunction)(double*, int), double *
VMHL_ResultVector, double *VMHL_Result);
```

#### Входные параметры:

Parameters — Вектор параметров генетического алгоритма. Каждый элемент обозначает свой параметр:

0 — длина вещественной хромосомы (определяется задачей оптимизации, что мы решаем);

1 — число вычислений целевой функции (CountOfFitness);

2 — тип селекции (TypeOfSel):

- 0 — ProportionalSelection (Пропорциональная селекция);
- 1 — RankSelection (Ранговая селекция);
- 2 — TournamentSelection (Турнирная селекция).

3 — тип скрещивания (TypeOfCros):

- 0 — SinglepointCrossover (Одноточечное скрещивание);
- 1 — TwopointCrossover (Двухточечное скрещивание);
- 2 — UniformCrossover (Равномерное скрещивание).

4 — тип мутации (TypeOfMutation):

- 0 — Weak (Слабая мутация);
- 1 — Average (Средняя мутация);
- 2 — Strong (Сильная мутация).

5 — тип формирования нового поколения (TypeOfForm):

- 0 — OnlyOffspringGenerationForming (Только потомки);
- 1 — OnlyOffspringWithBestGenerationForming (Только потомки и копия лучшего индивида).

6 — тип преобразования задачи вещественной оптимизации в задачу бинарной оптимизации (TypeOfConverting):

- 0 — IntConverting (Стандартное представление целого числа — номер узла в сетке дискретизации);
- 1 — GrayCodeConverting (Стандартный рефлексивный Грей-код).

NumberOfParts — указатель на массив: на сколько частей делить каждую вещественную координату при дискретизации (размерность Parameters[0]);

Желательно брать по формуле  $NumberOfParts[i] = 2^k - 1$ , где  $k$  — натуральное число, например, 12.

Left — массив левых границ изменения каждой вещественной координаты (размерность Parameters[0]);

Right — массив правых границ изменения каждой вещественной координаты (размерность Parameters[0]);

FitnessFunction — указатель на целевую функцию (если решается задача условной оптимизации, то учет ограничений должен быть включен в эту функцию);

VMHL\_ResultVector — найденное решение (вещественный вектор);

VMHL\_Result — значение целевой функции в точке, определенной вектором VMHL\_ResultVector.

### **Возвращаемое значение:**

1 — завершил работу без ошибок. Всё хорошо.

0 — возникли при работе ошибки. Скорее всего в этом случае в VMHL\_ResultVector и в VMHL\_Result не содержится решение задачи.

### **О функции:**

Реализация алгоритма из документа «Генетический алгоритм. Стандарт. v.3.0».

<https://github.com/Harrix/Standard-Genetic-Algorithm>

Алгоритм вещественной оптимизации. Ищет максимум целевой функции FitnessFunction.

Решением является вещественная строка.

**Примерный настройки** (для примера Вы можете поставить такие рабочие настройки):

Parameters[0]=50;

Parameters[1]=100\*100;

Parameters[2]=2;

Parameters[3]=2;

Parameters[4]=1;

Parameters[5]=1;

Parameters[6]=0;

### **Примечание:**

В сГА на вещественных строках не нужно задавать в параметрах число поколений и размер популяции, а только число вычислений целевой функции. Почему? Алгоритм сам определит число поколений и размер популяции, исходя из принципа, что число поколений и размер популяции должны быть примерно равны. Поэтому выбирайте значение Parameters[1] в виде:

int K=100;

Parameters[1]=K\*K;

То есть в виде квадрата целого числа. В противном случае реальное число вычислений целевой функции и значение Parameters[1] будут не совпадать.

Код целевой функции:

Код 160. Оптимизируемая функция

```

double Func2(double *x, int VMHL_N)
{
return -((x[0]-2)*(x[0]-2)+(x[1]-2)*(x[1]-2));
}

```

Код 161. Пример использования

```

int ChromosomeLength=2; //Длина хромосомы
int CountOfFitness=50*50; //Число вычислений целевой функции
int TypeOfSel=1; //Тип селекции
int TypeOfCros=0; //Тип скрещивания
int TypeOfMutation=1; //Тип мутации
int TypeOfForm=0; //Тип формирования нового поколения

int *ParametersOfStandartRealGeneticAlgorithm;
ParametersOfStandartRealGeneticAlgorithm=new int[7];
ParametersOfStandartRealGeneticAlgorithm[0]=ChromosomeLength; //Длина хромосомы
ParametersOfStandartRealGeneticAlgorithm[1]=CountOfFitness; //Число вычислений целевой
    функции
ParametersOfStandartRealGeneticAlgorithm[2]=TypeOfSel; //Тип селекции
ParametersOfStandartRealGeneticAlgorithm[3]=TypeOfCros; //Тип скрещивания
ParametersOfStandartRealGeneticAlgorithm[4]=TypeOfMutation; //Тип мутации
ParametersOfStandartRealGeneticAlgorithm[5]=TypeOfForm; //Тип формирования нового поко
    ления
ParametersOfStandartRealGeneticAlgorithm[6]=0; //Тип преобразование задачи веществен
        нной оптимизации в задачу бинарной оптимизации

double *Left; //массив левых границ изменения каждой вещественной координаты
Left=new double[ChromosomeLength];
double *Right; //массив правых границ изменения каждой вещественной координаты
Right=new double[ChromosomeLength];
int *NumberOfParts; //на сколько делить каждую координату
NumberOfParts=new int[ChromosomeLength];

//Заполним массивы
//Причем по каждой координате одинаковые значения выставим
TMHL_FillVector(Left,ChromosomeLength,-5.); //Пусть будет интервал [-3;3]
TMHL_FillVector(Right,ChromosomeLength,5.);
TMHL_FillVector(NumberOfParts,ChromosomeLength,TMHL_PowerOf(2,15)-1); //Делим на
    32768-1 частей каждую вещественную координату

double *Decision; //вещественное решение
Decision=new double[ChromosomeLength];
double ValueOfFitnessFunction; //значение целевой функции в точке Decision
int VMHL_Success=0; //Успешен ли будет запуск СГА

//Запуск алгоритма
VMHL_Success=MHL_StandartRealGeneticAlgorithm (
    ParametersOfStandartRealGeneticAlgorithm, NumberOfParts, Left, Right, Func2, Decision,
    &ValueOfFitnessFunction);

//Используем полученный результат
MHL_ShowNumber(VMHL_Success, "Как прошел запуск", "VMHL_Success");
if (VMHL_Success==1)
{
    MHL_ShowVectorT(Decision,ChromosomeLength, "Найденное решение", "Decision");
    //Найденное решение:
    //Decision =
    //2.00348 2.00226
    MHL_ShowNumber(ValueOfFitnessFunction, "Значение целевой функции", "
        ValueOfFitnessFunction");
}

```

```

//Значение целевой функции:
//ValueOfFitnessFunction=-1.72034e-05
}

delete [] ParametersOfStandartRealGeneticAlgorithm;
delete [] Decision;
delete [] Left;
delete [] Right;
delete [] NumberOfParts;

```

### 7.2.18 MHL\_TournamentSelection

Турнирная селекция. Оператор генетического алгоритма. Работает с массивом пригодностей индивидов. В переопределенной функции используется во входных параметрах дополнительный массив, так как функция часто вызывается, а постоянно создавать массив накладно.

Код 162. Синтаксис

```

int MHL_TournamentSelection(double *Fitness, int SizeTournament, int VMHL_N);
int MHL_TournamentSelection(double *Fitness, int SizeTournament, int *Taken, int
VMHL_N);

```

**Входные параметры:**

Fitness — массив пригодностей индивидов;

SizeTournament — размер турнира;

VMHL\_N — размер массива.

**Возвращаемое значение:**

Номер выбранного индивида популяции.

Для переопределенной функции.

**Входные параметры:**

Fitness — массив пригодностей индивидов;

SizeTournament — размер турнира;

Taken — Информация о том, в турнире или нет индивид (служебный массив);

VMHL\_N — размер массива.

**Возвращаемое значение:**

Номер выбранного индивида популяции.

**Примечание:**

Является стандартной реализацией турнирной селекции. Это турнирная селекция без возврата.

Код 163. Пример использования

```

int i;
int VMHL_N=7; //Размер массива

```

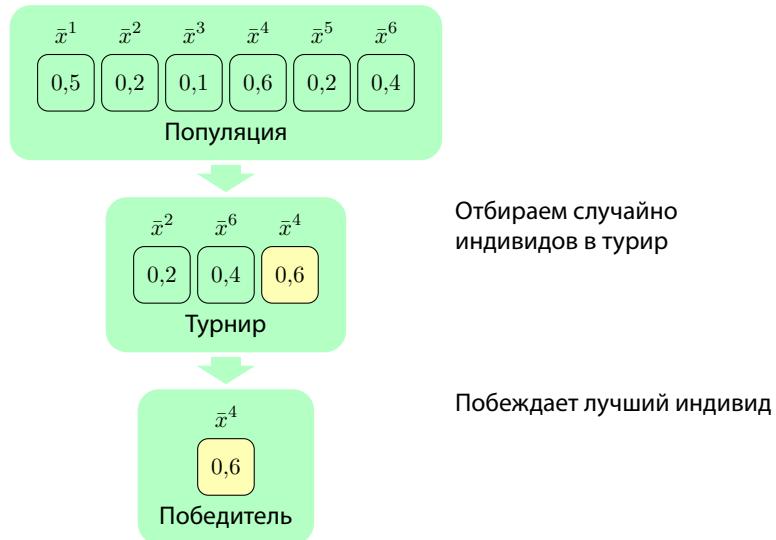


Рисунок 8. Механизм работы турнирной селекции

```

double *Fitness;
Fitness=new double[VMHL_N];
for (i=0;i<VMHL_N;i++)
Fitness[i]=MHL_RandomNumber();

int SizeTournament=3;// Размер турнира

// Вызов функции
int Number=MHL_TournamentSelection(Fitness,SizeTournament,VMHL_N);

// Используем полученный результатом

MHL_ShowVector (Fitness,VMHL_N,"Массив пригодностей", "Fitness");
// Массив пригодностей:
// Fitness =
// 0.858643
// 0.460541
// 0.469696
// 0.454315
// 0.594543
// 0.000457764
// 0.476135

MHL_ShowNumber (SizeTournament,"Размер турнираа", "SizeTournament");
// Размер турнираа:
// SizeTournament=3

MHL_ShowNumber (Number,"Номер выбранного индивида", "Number");
// Номер выбранного индивида:
// Number=6

delete [] Fitness;

// Для переопределенной функции
VMHL_N=7;// Размер массива
Fitness=new double[VMHL_N];
for (i=0;i<VMHL_N;i++)
Fitness[i]=MHL_RandomNumber();

```

```

int *Taken; //Информация о том, в турнире или нет индивид (Служебный массив)
Taken=new int[VMHL_N];

SizeTournament=3; // Размер турнира

//Вызов функции
Number=MHL_TournamentSelection(Fitness,SizeTournament,Taken,VMHL_N);

//Используем полученный результат

MHL_ShowVector (Fitness,VMHL_N,"Массив пригодностей", "Fitness");
//Массив пригодностей:
//Fitness =
//0.598633
//0.396423
//0.756683
//0.123505
//0.0546875
//0.542511
//0.605499

MHL_ShowNumber (SizeTournament,"Размер турнира", "SizeTournament");
//Размер турнира:
//SizeTournament=3

MHL_ShowNumber (Number,"Номер выбранного индивида", "Number");
//Номер выбранного индивида:
//Number=2

delete [] Fitness;
delete [] Taken;

```

### 7.2.19 MHL\_TournamentSelectionWithReturn

Турнирная селекция с возвращением. Оператор генетического алгоритма. Работает с массивом пригодностей индивидов.

Код 164. Синтаксис

```
int MHL_TournamentSelectionWithReturn(double *Fitness, int SizeTournament, int VMHL_N
);
```

**Входные параметры:**

Fitness — массив пригодностей индивидов;

VMHL\_N — размер массива VectorProbability;

SizeTournament — размер турнира.

**Возвращаемое значение:**

Номер выбранного индивида популяции.

**Примечание:**

Не является стандартной реализацией турнирной селекции, так как в классической турнирной селекции в один турнир один и тот же индивид может попасть только один раз.

Код 165. Пример использования

```
int i;
int VMHL_N=7; //Размер массива
double *Fitness;
Fitness=new double[VMHL_N];
for (i=0;i<VMHL_N;i++)
    Fitness[i]=MHL_RandomNumber();

int SizeTournament=3; // Размер турнира

//Вызов функции
int Number=MHL_TournamentSelectionWithReturn(Fitness,SizeTournament,VMHL_N);

//Используем полученный результатом

MHL_ShowVector (Fitness,VMHL_N, "Массив пригодностей", "Fitness");
//Массив пригодностей:
//Fitness =
//0.883148
//0.370209
//0.0719604
//0.311371
//0.558594
//0.42215
//0.011322

MHL_ShowNumber (Number, "Номер выбранного индивида", "Number");
//Номер выбранного индивида:
//Number=4

delete [] Fitness;
```

## 7.2.20 TMHL\_MutationBinaryMatrix

Мутация для бинарной матрицы. Оператор генетического алгоритма.

Код 166. Синтаксис

```
template <class T> void TMHL_MutationBinaryMatrix(T **VMHL_ResultMatrix, double
ProbabilityOfMutation, int VMHL_N,int VMHL_M);
```

**Входные параметры:**

VMHL\_ResultMatrix — указатель на преобразуемый массив;

ProbabilityOfMutation — вероятность мутации;

VMHL\_N — размер массива VMHL\_ResultMatrix (число строк);

VMHL\_M — размер массива VMHL\_ResultMatrix (число столбцов).

**Возвращаемое значение:**

Отсутствует.

**Принцип работы:** (на примере одной строки матрицы)

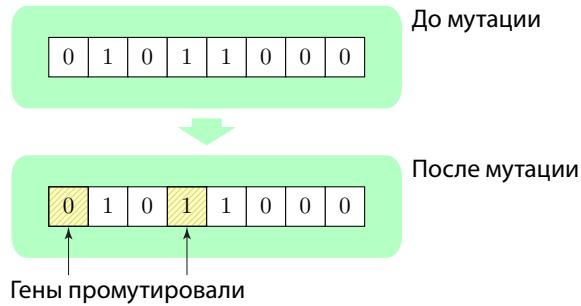


Рисунок 9. Механизм работы мутации

Код 167. Пример использования

```
int i;
int VMHL_N=10; //Размер массива (число строк)
int VMHL_M=3; //Размер массива (число столбцов)
int **a;
a=new int*[VMHL_N];
for (i=0;i<VMHL_N;i++) a[i]=new int[VMHL_M];
TMHL_RandomBinaryMatrix(a,VMHL_N,VMHL_M); //Случайная бинарная матрица
MHL_ShowMatrix (a,VMHL_N,VMHL_M, "Случайная бинарная матрица", "a");
// Случайная бинарная матрица:
//a =
//1 0 1
//0 0 0
//0 1 1
//1 0 1
//1 0 1
//1 0 1
//0 0 1
//1 1 0
//1 0 1
//1 1 0

double ProbabilityOfMutation=0.1; //Вероятность мутации

//Вызов функции
TMHL_MutationBinaryMatrix(a,ProbabilityOfMutation,VMHL_N,VMHL_M);

//Используем полученный результат
MHL_ShowMatrix (a,VMHL_N,VMHL_M, "Мутированная бинарная матрица", "a");
//Мутированная бинарная матрица:
//a =
//1 1 1
//1 0 0
//0 1 1
//1 0 1
//1 0 1
//1 0 1
//0 0 1
//1 1 0
//1 0 1
//1 1 1

for (i=0;i<VMHL_N;i++) delete [] a[i];
delete [] a;
```

### 7.2.21 TMHL\_SinglepointCrossover

Одноточечное скрещивание. Оператор генетического алгоритма.

Код 168. Синтаксис

```
template <class T> void TMHL_SinglepointCrossover(T *Parent1, T *Parent2, T *
VMHL_ResultVector, int VMHL_N);
```

**Входные параметры:**

Parent1 — первый родитель;

Parent2 — второй родитель;

VMHL\_ResultVector — потомок;

VMHL\_N — размер векторов Parent1, Parent2 и VMHL\_ResultVector.

**Возвращаемое значение:**

Отсутствует.

**Примечание:**

Потомок выбирается случайно.

**Формула:**

$$\begin{aligned} \text{Crossover} \left( \overline{\text{Parent}}^1, \overline{\text{Parent}}^2, \text{DataOfCros} \right) &= \text{Random} \left( \left\{ \overline{\text{Offspring}}^1; \overline{\text{Offspring}}^2 \right\} \right), \\ R &= \text{Random} (\{2; 3; \dots; n\}); \\ \overline{\text{Offspring}}_i^1 &= \overline{\text{Parent}}_i^1, i = \overline{1, R-1}; \\ \overline{\text{Offspring}}_i^1 &= \overline{\text{Parent}}_i^2, i = \overline{R, n}; \\ \overline{\text{Offspring}}_i^2 &= \overline{\text{Parent}}_i^2, i = \overline{1, R-1}; \\ \overline{\text{Offspring}}_i^2 &= \overline{\text{Parent}}_i^1, i = \overline{R, n}; \\ \overline{\text{Offspring}}^1 &\in X, \overline{\text{Offspring}}^2 \in X. \end{aligned}$$

DataOfCros не содержит каких-либо параметров относительно данного типа скрещивания.

**Пример.** Для всех видов скрещивания будем использовать двух родителей:  $\overline{\text{Parent}}^1 = (0; 1; 0; 1; 1; 1; 0; 0)^T$  и  $\overline{\text{Parent}}^2 = (1; 1; 0; 0; 1; 0; 1)^T$ . Одноточечное скрещивание показано на рисунке:

Код 169. Пример использования

```
int VMHL_N=10; //Размер массива (число строк)
int *Parent1;
Parent1=new int[VMHL_N];
int *Parent2;
Parent2=new int[VMHL_N];
int *Child;
Child=new int[VMHL_N];
TMHL_RandomBinaryVector(Parent1,VMHL_N);
TMHL_RandomBinaryVector(Parent2,VMHL_N);
```

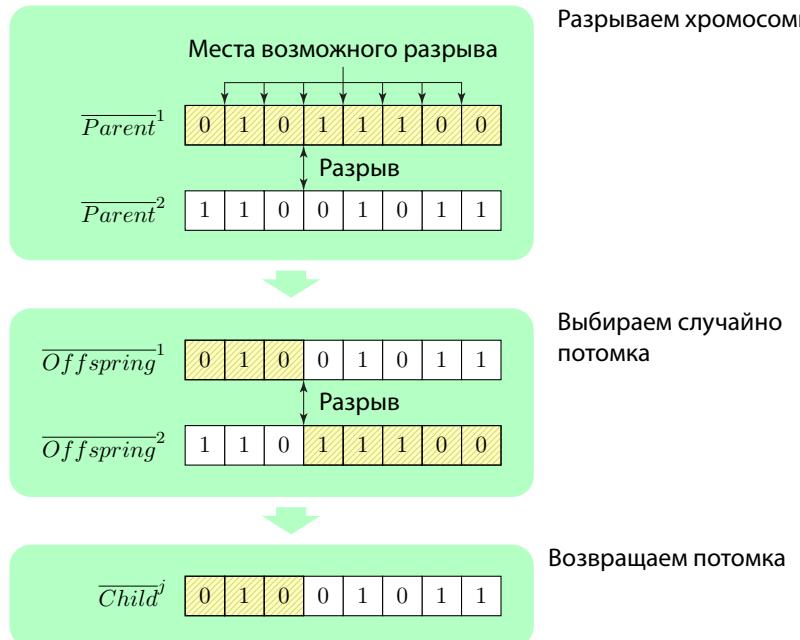


Рисунок 10. Механизм работы одноточечного скрещивания

```
//Получим потомка Child
TMHL_SinglepointCrossover(Parent1,Parent2,Child,VMHL_N);

//Используем полученный результат
MHL_ShowVectorT (Parent1,VMHL_N,"Первый родитель", "Parent1");
//Первый родитель:
//Parent1 =
//0  1  1  0  0  1  0  1  1  1

MHL_ShowVectorT (Parent2,VMHL_N,"Второй родитель", "Parent2");
//Второй родитель:
//Parent2 =
//0  0  0  1  0  1  0  0  0  0

MHL_ShowVectorT (Child,VMHL_N,"Полученный потомок", "Child");
//Полученный потомок:
//Child =
//0  1  1  0  0  1  0  1  1  1

delete [] Parent2;
delete [] Parent1;
delete [] Child;
```

## 7.2.22 TMHL\_SinglepointCrossoverWithCopying

Одноточечное скрещивание с возможностью полного копирования одного из родителей. Оператор генетического алгоритма. Отличается от стандартного одноточечного скрещивания тем, что точки разрыва могут происходить по краям родителей, что может привести к полному копированию родителя.

Код 170. Синтаксис

```
template <class T> void TMHL_SinglepointCrossoverWithCopying(T *Parent1, T *Parent2,
    T *VMHL_ResultVector, int VMHL_N);
```

### **Входные параметры:**

Parent1 — первый родитель;

Parent2 — второй родитель;

VMHL\_ResultVector — потомок;

VMHL\_N — размер векторов Parent1, Parent2 и VMHL\_ResultVector.

### **Возвращаемое значение:**

Отсутствует.

### **Примечание:**

Потомок выбирается случайно.

Отличается от стандартного одноточечного скрещивания тем, что точки разрыва могут проходить по краям родителей, что может привести к полному копированию родителя.

### **Формула:**

$$\text{Crossover} \left( \overline{\text{Parent}}^1, \overline{\text{Parent}}^2, \text{DataOfCros} \right) = \text{Random} \left( \left\{ \overline{\text{Offspring}}^1; \overline{\text{Offspring}}^2 \right\} \right),$$

$$R = \text{Random} (\{1; 3; \dots; n + 1\});$$

$$\overline{\text{Offspring}}_i^1 = \overline{\text{Parent}}_i^1, i = \overline{1, R - 1};$$

$$\overline{\text{Offspring}}_i^1 = \overline{\text{Parent}}_i^2, i = \overline{R, n};$$

$$\overline{\text{Offspring}}_i^2 = \overline{\text{Parent}}_i^2, i = \overline{1, R - 1};$$

$$\overline{\text{Offspring}}_i^2 = \overline{\text{Parent}}_i^1, i = \overline{R, n};$$

$$\overline{\text{Offspring}}^1 \in X, \overline{\text{Offspring}}^2 \in X.$$

*DataOfCros* не содержит каких-либо параметров относительно данного типа скрещивания.

**Пример.** Для всех видов скрещивания будем использовать двух родителей:  $\overline{\text{Parent}}^1 = (0; 1; 0; 1; 1; 1; 0; 0)^T$  и  $\overline{\text{Parent}}^2 = (1; 1; 0; 0; 1; 0; 1)^T$ . Одноточечное скрещивание показано на рисунке:

Код 171. Пример использования

```

int VMHL_N=10; //Размер массива (число строк)
int *Parent1;
Parent1=new int[VMHL_N];
int *Parent2;
Parent2=new int[VMHL_N];
int *Child;
Child=new int[VMHL_N];
TMHL_RandomBinaryVector(Parent1,VMHL_N);
TMHL_RandomBinaryVector(Parent2,VMHL_N);

//Получим потомка Child
TMHL_SinglepointCrossoverWithCopying(Parent1,Parent2,Child,VMHL_N);

//Используем полученный результат

```

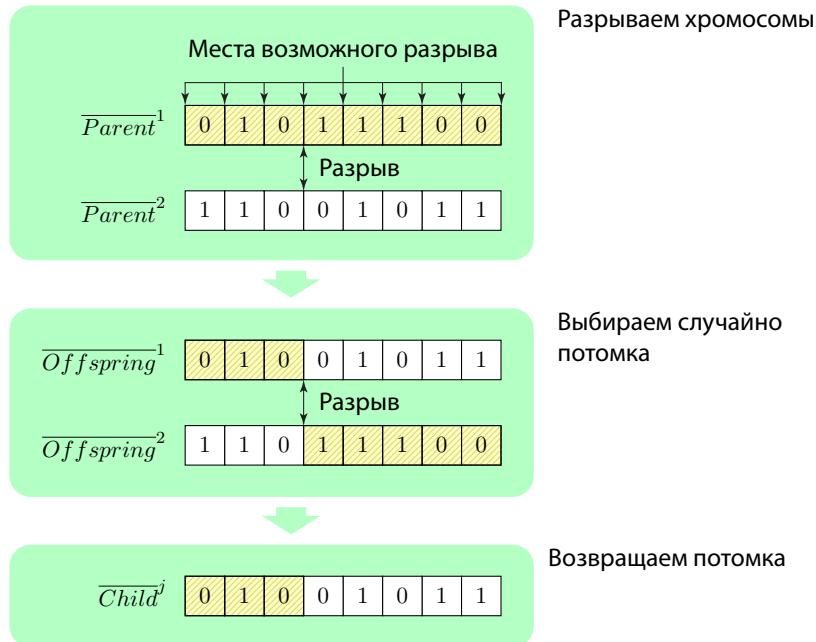


Рисунок 11. Механизм работы одноточечного скрещивания с возможностью полного копирования одного из родителей

```

MHL_ShowVectorT (Parent1,VMHL_N,"Первый родитель", "Parent1");
//Первый родитель:
//Parent1 =
//0 1 1 0 0 1 0 1 1 1

MHL_ShowVectorT (Parent2,VMHL_N,"Второй родитель", "Parent2");
//Второй родитель:
//Parent2 =
//0 0 0 1 0 1 0 0 0 0

MHL_ShowVectorT (Child,VMHL_N,"Полученный потомок", "Child");
//Полученный потомок:
//Child =
//0 1 1 0 0 1 0 1 1 1

delete [] Parent2;
delete [] Parent1;
delete [] Child;

```

### 7.2.23 TMHL\_TwopointCrossover

Двуточечное скрещивание. Оператор генетического алгоритма.

Код 172. Синтаксис

```
template <class T> void TMHL_TwopointCrossover(T *Parent1, T *Parent2, T *
VMHL_ResultVector, int VMHL_N);
```

**Входные параметры:**

Parent1 — первый родитель;

Parent2 — второй родитель;

VMHL\_ResultVector — потомок;

VMHL\_N — размер векторов Parent1, Parent2 и VMHL\_ResultVector.

**Возвращаемое значение:**

Отсутствует.

**Примечание:**

Потомок выбирается случайно.

$$\begin{aligned} \text{Crossover} \left( \overline{\text{Parent}}^1, \overline{\text{Parent}}^2, \text{DataOfCros} \right) &= \text{Random} \left( \left\{ \overline{\text{Offspring}}^1; \overline{\text{Offspring}}^2 \right\} \right), \\ r_1 &= \text{Random} (\{2; 3; \dots; n\}); \\ r_2 &= \text{Random} (\{2; 3; \dots; n\}); \\ R_1 &= \min (r_1, r_2); \\ R_2 &= \max (r_1, r_2); \\ \overline{\text{Offspring}}_i^1 &= \overline{\text{Parent}}_i^1, i = \overline{1, R_1 - 1}; \\ \overline{\text{Offspring}}_i^1 &= \overline{\text{Parent}}_i^2, i = \overline{R_1, R_2 - 1}; \\ \overline{\text{Offspring}}_i^1 &= \overline{\text{Parent}}_i^1, i = \overline{R_2, n}; \\ \overline{\text{Offspring}}_i^2 &= \overline{\text{Parent}}_i^2, i = \overline{1, R_1 - 1}; \\ \overline{\text{Offspring}}_i^2 &= \overline{\text{Parent}}_i^1, i = \overline{R_1, R_2 - 1}; \\ \overline{\text{Offspring}}_i^2 &= \overline{\text{Parent}}_i^2, i = \overline{R_2, n}; \\ \overline{\text{Offspring}}^1 &\in X, \overline{\text{Offspring}}^2 \in X. \end{aligned} \tag{1}$$

DataOfCros не содержит каких-либо параметров относительно данного типа скрещивания.

**Пример.** Двухточечное скрещивание показано на рисунке:

Код 173. Пример использования

```
int VMHL_N=10; //Размер массива (число строк)
int *Parent1;
Parent1=new int[VMHL_N];
int *Parent2;
Parent2=new int[VMHL_N];
int *Child;
Child=new int[VMHL_N];
TMHL_RandomBinaryVector(Parent1,VMHL_N);
TMHL_RandomBinaryVector(Parent2,VMHL_N);

//Получим потомка Child
TMHL_TwoPointCrossover(Parent1,Parent2,Child,VMHL_N);

//Используем полученный результат
MHL_ShowVectorT (Parent1,VMHL_N,"Первый родитель", "Parent1");
//Первый родитель:
//Parent1 =
//1 1 0 0 1 1 0 0 0 0

MHL_ShowVectorT (Parent2,VMHL_N,"Второй родитель", "Parent2");
```

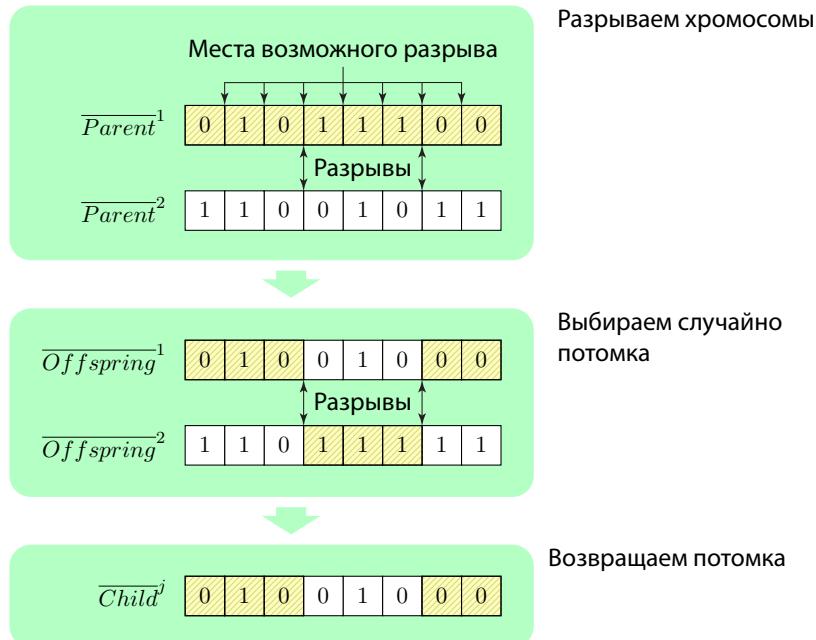


Рисунок 12. Механизм работы двухточечного скрещивания

```
//Второй родитель:  

//Parent2 =  

//0 1 1 0 0 1 0 0 1 0  

MHL_ShowVectorT (Child,VMHL_N,"Полученный потомок", "Child");  

//Полученный потомок:  

//Child =  

//0 1 1 0 0 1 0 0 1 0  

delete [] Parent2;  

delete [] Parent1;  

delete [] Child;
```

### 7.2.24 TMHL\_TwoPointCrossoverWithCopying

Двухточечное скрещивание с возможностью полного копирования одного из родителей. Оператор генетического алгоритма. Отличается от стандартного двухточечного скрещивания тем, что точки разрыва могут происходить по краям родителей, что может привести к полному копированию родителя.

Код 174. Синтаксис

```
template <class T> void TMHL_TwoPointCrossoverWithCopying(T *Parent1, T *Parent2, T *  

VMHL_ResultVector, int VMHL_N);
```

#### Входные параметры:

Parent1 — первый родитель;

Parent2 — второй родитель;

VMHL\_ResultVector — потомок;

VMHL\_N — размер векторов Parent1, Parent2 и VMHL\_ResultVector.

**Возвращаемое значение:**

Отсутствует.

**Примечание:**

Потомок выбирается случайно.

Отличается от стандартного двухточечного скрещивания тем, что точки разрыва могут проходить по краям родителей, что может привести к полному копированию родителя.

$$\begin{aligned}
 & \text{Crossover} \left( \overline{\text{Parent}}^1, \overline{\text{Parent}}^2, \text{DataOfCros} \right) = \text{Random} \left( \left\{ \overline{\text{Offspring}}^1; \overline{\text{Offspring}}^2 \right\} \right), \\
 & r_1 = \text{Random} (\{1; 2; \dots; n + 1\}); \\
 & r_2 = \text{Random} (\{1; 2; \dots; n + 1\}); \\
 & R_1 = \min (r_1, r_2); \\
 & R_2 = \max (r_1, r_2); \\
 & \overline{\text{Offspring}}_i^1 = \overline{\text{Parent}}_i^1, i = \overline{1, R_1 - 1}; \\
 & \overline{\text{Offspring}}_i^1 = \overline{\text{Parent}}_i^2, i = \overline{R_1, R_2 - 1}; \\
 & \overline{\text{Offspring}}_i^1 = \overline{\text{Parent}}_i^1, i = \overline{R_2, n}; \\
 & \overline{\text{Offspring}}_i^2 = \overline{\text{Parent}}_i^2, i = \overline{1, R_1 - 1}; \\
 & \overline{\text{Offspring}}_i^2 = \overline{\text{Parent}}_i^1, i = \overline{R_1, R_2 - 1}; \\
 & \overline{\text{Offspring}}_i^2 = \overline{\text{Parent}}_i^2, i = \overline{R_2, n}; \\
 & \overline{\text{Offspring}}^1 \in X, \overline{\text{Offspring}}^2 \in X. \tag{2}
 \end{aligned}$$

*DataOfCros* не содержит каких-либо параметров относительно данного типа скрещивания.

**Пример.** Двухточечное скрещивание показано на рисунке:

Код 175. Пример использования

```

int VMHL_N=10; //Размер массива (число строк)
int *Parent1;
Parent1=new int[VMHL_N];
int *Parent2;
Parent2=new int[VMHL_N];
int *Child;
Child=new int[VMHL_N];
TMHL_RandomBinaryVector(Parent1,VMHL_N);
TMHL_RandomBinaryVector(Parent2,VMHL_N);

//Получим потомка Child
TMHL_TwopointCrossoverWithCopying(Parent1,Parent2,Child,VMHL_N);

//Используем полученный результат
MHL_ShowVectorT (Parent1,VMHL_N, "Первый родитель", "Parent1");
//Первый родитель:
//Parent1 =
//1 1 0 0 1 1 0 0 0 0

MHL_ShowVectorT (Parent2,VMHL_N, "Второй родитель", "Parent2");
//Второй родитель:

```

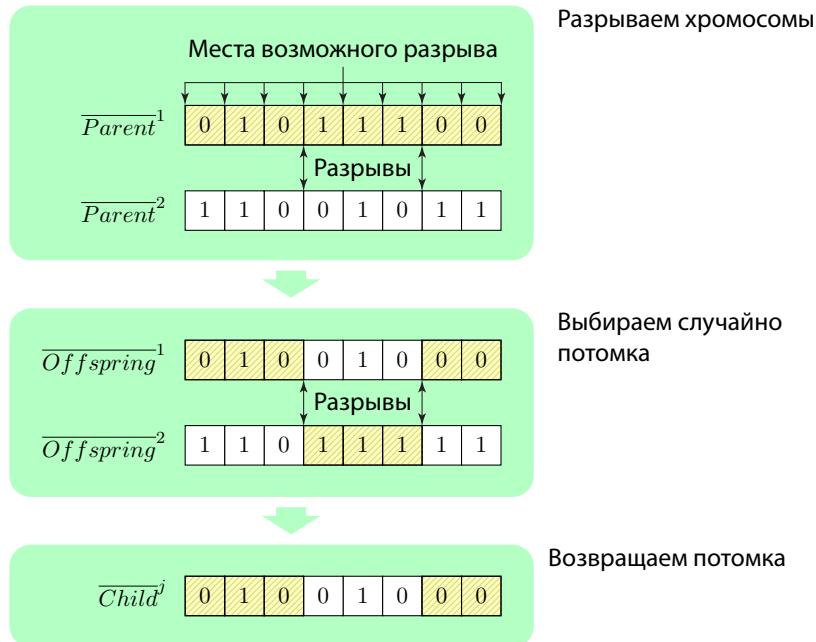


Рисунок 13. Механизм работы двухточечного скрещивания с возможностью полного копирования одного из родителей

```
//Parent2 =
//0 1 1 0 0 1 0 0 1 0

MHL_ShowVectorT (Child,VMHL_N,"Полученный потомок", "Child");
//Полученный потомок:
//Child =
//0 1 1 0 0 1 0 0 1 0

delete [] Parent2;
delete [] Parent1;
delete [] Child;
```

## 7.2.25 TMHL\_UniformCrossover

Равномерное скрещивание. Оператор генетического алгоритма.

Код 176. Синтаксис

```
template <class T> void TMHL_UniformCrossover(T *Parent1, T *Parent2, T *
VMHL_ResultVector, int VMHL_N);
```

**Входные параметры:**

Parent1 — первый родитель;

Parent2 — второй родитель;

VMHL\_ResultVector — потомок;

VMHL\_N — размер векторов Parent1, Parent2 и VMHL\_ResultVector.

**Возвращаемое значение:**

Отсутствует.

$$\begin{aligned} Crossover \left( \overline{Parent}^1, \overline{Parent}^2, DataOfCros \right) &= \overline{Offspring}; \\ \overline{Offspring}_i &= Random \left( \left\{ \overline{Parent}_i^1; \overline{Parent}_i^2 \right\} \right), i = \overline{1, n}; \\ \overline{Offspring} &\in X. \end{aligned}$$

*DataOfCros* не содержит каких-либо параметров относительно данного типа скрещивания.

**Пример.** Двухточечное скрещивание показано на рисунке:

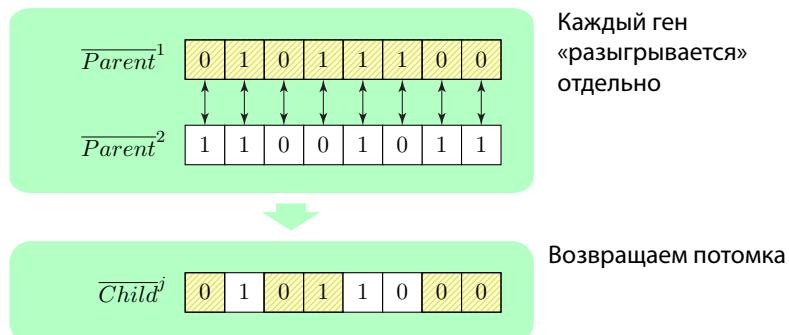


Рисунок 14. Механизм работы равномерного скрещивания

Код 177. Пример использования

```

int VMHL_N=10; //Размер массива (число строк)
int *Parent1;
Parent1=new int[VMHL_N];
int *Parent2;
Parent2=new int[VMHL_N];
int *Child;
Child=new int[VMHL_N];
TMHL_RandomBinaryVector(Parent1,VMHL_N);
TMHL_RandomBinaryVector(Parent2,VMHL_N);

//Получим потомка Child
TMHL_UniformCrossover(Parent1,Parent2,Child,VMHL_N);

//Используем полученный результат
MHL_ShowVectorT (Parent1,VMHL_N, "Первый родитель", "Parent1");
//Первый родитель:
//Parent1 =
//1 1 0 1 1 1 1 1 0

MHL_ShowVectorT (Parent2,VMHL_N, "Второй родитель", "Parent2");
//Второй родитель:
//Parent2 =
//0 0 0 0 1 0 1 1 0

MHL_ShowVectorT (Child,VMHL_N, "Полученный потомок", "Child");
//Полученный потомок:
//Child =
//1 1 0 0 1 0 1 1 1 0

delete [] Parent2;
    
```

```
delete [] Parent1;
delete [] Child;
```

## 7.3 Геометрия

### 7.3.1 MHL\_LineGeneralForm

Функция представляет собой уравнение прямой по общему уравнению прямой вида  $Ax + By + C = 0$ .

Код 178. Синтаксис

```
double MHL_LineGeneralForm(double x, double A, double B, double C, int *solutionis);
double MHL_LineGeneralForm(double x, double A, double B, double C);
```

#### Входные параметры:

$x$  — значение точки для которой считаем значение прямой;

$A$  — множитель из уравнения прямой  $Ax + By + C = 0$ ;

$B$  — множитель из уравнения прямой  $Ax + By + C = 0$ ;

$C$  — слагаемое из уравнения прямой  $Ax + By + C = 0$ ;

$solutionis$  — сюда возвращается результат решения задачи:

0 — решений нет;

1 — решение есть;

2 — любое число является решением (прямая параллельна оси  $Oy$ )

В функции перезагрузки нет параметра  $solutionis$ .

#### Возвращаемое значение:

Значение  $y$  прямой для данного  $x$ .

Код 179. Пример использования

```
double A=MHL_RandomUniformInt(1,10);
double B=MHL_RandomUniformInt(1,10);
double C=MHL_RandomUniformInt(1,10);

double x=5;

int solutionis;

//Вызов функции
double y=MHL_LineGeneralForm(x,A,B,C,&solutionis);

//Используем полученный результат
MHL_ShowText("Уравнение прямой: "+MHL_NumberToText(A)+"*x+"+MHL_NumberToText(B)+"*y"+
    "+MHL_NumberToText(C)+"=0");
//Уравнение прямой: 7*x+1*y+7=0.
MHL_ShowNumber(solutionis,"Найдено ли решение","solutionis");
//Найдено ли решение:
```

```

//solutionis=1
if (solutionis!=0)
{
MHL_ShowNumber(y, "Значение функции прямой в точке x=5", "y");
//Значение функции прямой в точке x=5:
//y=-42
}

```

### 7.3.2 MHL\_LineSlopeInterceptForm

Функция представляет собой уравнение прямой с угловым коэффициентом вида  $y = kx + b$ .

Код 180. Синтаксис

```
double MHL_LineSlopeInterceptForm(double x, double k, double b);
```

**Входные параметры:**

x — значение точки для которой считаем значение прямой;

k — угловой коэффициент из уравнения прямой  $y = kx + b$ ;

b — слагаемое из уравнения прямой  $y = kx + b$ ;

**Возвращаемое значение:**

Значение у прямой для данного x.

Код 181. Пример использования

```

double k=MHL_RandomUniformInt(1,10);
double b=MHL_RandomUniformInt(1,10);

double x=5;

//Вызов функции
double y=MHL_LineSlopeInterceptForm(x,k,b);

//Используем полученный результат
MHL_ShowText("Уравнение прямой: y="+MHL_NumberToText(k)+"*x+"+MHL_NumberToText(b));
//Уравнение прямой: y=4*x+1.
MHL_ShowNumber(y, "Значение функции прямой в точке x=5", "y");
//Значение функции прямой в точке x=5:
//y=21

```

### 7.3.3 MHL\_LineTwoPoint

Функция представляет собой уравнение прямой по двум точкам.

Код 182. Синтаксис

```

double MHL_LineTwoPoint(double x, double x1, double y1, double x2, double y2, int *
    solutionis);
double MHL_LineTwoPoint(double x, double x1, double y1, double x2, double y2);

```

**Входные параметры:**

$x$  — значение точки для которой считаем значение прямой;

$x_1$  — абсцисса первой точки;

$y_1$  — ордината первой точки;

$x_2$  — абсцисса второй точки;

$y_2$  — ордината второй точки;

`solutionis` — сюда возвращается результат решения задачи:

0 — решения нет;

1 — решение есть;

2 — любое число является решением (прямая параллельна оси  $Oy$ ).

В функции перезагрузки нет параметра `solutionis`.

#### **Возвращаемое значение:**

Значение у прямой для данного  $x$ .

Код 183. Пример использования

```
double x1=MHL_RandomUniformInt(1,10);
double y1=MHL_RandomUniformInt(1,10);
double x2=MHL_RandomUniformInt(1,10);
double y2=MHL_RandomUniformInt(1,10);

double x=5;

int solutionis;

//Вызов функции
double y=MHL_LineTwoPoint(x,x1,y1,x2,y2,&solutionis);

//Используем полученный результат
MHL_ShowText("Первая точка: ("+MHL_NumberToText(x1)+";" "+MHL_NumberToText(y1)+")");
//Первая точка: (6; 3).
MHL_ShowText("Вторая точка: ("+MHL_NumberToText(x2)+";" "+MHL_NumberToText(y2)+")");
//Вторая точка: (3; 3).
MHL_ShowNumber(solutionis,"Найдено ли решение","solutionis");
//Найдено ли решение:
//solutionis=1
if (solutionis!=0)
{
MHL_ShowNumber(y,"Значение прямой, проходящей через две указанные точки, в точке x=5",
"y");
//Значение прямой, проходящей через две указанные точки, в точке x=5:
//y=32
}
```

#### **7.3.4 MHL\_Parabola**

Функция представляет собой уравнение параболы вида:  $y = ax^2 + bx + c$ .

Код 184. Синтаксис

```
double MHL_Parabola(double x, double a, double b, double c);
```

#### Входные параметры:

x — значение точки для которой считаем значение параболы;

a — множитель из уравнения  $y = ax^2 + bx + c$ ;

b — множитель из уравнения  $y = ax^2 + bx + c$ ;

c — слагаемое из уравнения  $y = ax^2 + bx + c$ .

#### Возвращаемое значение:

Значение у параболы для данного x.

#### Код 185. Пример использования

```
double a=MHL_RandomUniformInt(1,10);
double b=MHL_RandomUniformInt(1,10);
double c=MHL_RandomUniformInt(1,10);

double x=5;

//Вызов функции
double y=MHL_Parabola(x,a,b,c);

//Используем полученный результат
MHL_ShowText("Уравнение параболы: y="+MHL_NumberToText(a)+"*x^2+"+MHL_NumberToText(b)
    +"*x+"+MHL_NumberToText(c));
//Уравнение параболы: y=4*x^2+5*x+3.
MHL_ShowNumber(y,"Значение функции параболы в точке x=5", "y");
//Значение функции параболы в точке x=5:
//y=128
```

### 7.3.5 TMHL\_BoolCrossingTwoSegment

Функция определяет наличие пересечения двух отрезков. Координаты отрезков могут быть перепутаны по порядку в каждом отрезке.

#### Код 186. Синтаксис

```
template <class T> int TMHL_BoolCrossingTwoSegment(T b1,T c1,T b2,T c2);
```

#### Входные параметры:

b1 — левый конец первого отрезка;

c1 — правый конец первого отрезка;

b2 — левый конец второго отрезка;

c2 — правый конец второго отрезка.

#### Возвращаемое значение:

1 — отрезки пересекаются;

0 — отрезки не пересекаются.

Код 187. Пример использования

```
double b1,c1,b2,c2;
int Result;
//Зададим случайные координаты отрезков
b1=MHL_RandomUniform(-3,5);
c1=MHL_RandomUniform(-3,5);
b2=MHL_RandomUniform(-3,5);
c2=MHL_RandomUniform(-3,5);

//Вызов функции
Result=TMHL_BoolCrossingTwoSegment(b1,c1,b2,c2);

//Используем полученный результат
MHL_ShowNumber (b1,"Левый конец первого отрезка", "b1");
//Левый конец первого отрезка:
//b1=0.773193
MHL_ShowNumber (c1,"Правый конец первого отрезка", "c1");
//Правый конец первого отрезка:
//c1=3.22803
MHL_ShowNumber (b2,"Левый конец второго отрезка", "b2");
//Левый конец второго отрезка:
//b2=4.99121
MHL_ShowNumber (c2,"Правый конец второго отрезка", "c2");
//Правый конец второго отрезка:
//c2=1.43921
MHL_ShowNumber (Result,"Пересекаются ли отрезки", "Result");
//Пересекаются ли отрезки:
//Result=1
```

## 7.4 Гиперболические функции

### 7.4.1 MHL\_Cosech

Функция возвращает гиперболический косеканс.

Код 188. Синтаксис

```
double MHL_Cosech(double x);
```

**Входные параметры:**

x — входная переменная.

**Возвращаемое значение:**

Гиперболический косеканс.

Код 189. Пример использования

```
double x=MHL_RandomUniform(0,10);

//Вызов функции
double Result=MHL_Cosech(x);

//Используем полученный результат
MHL_ShowNumber(Result,"Гиперболический косеканс от x="+MHL_NumberToText(x),"равен");
//Гиперболический косеканс от x=0.571289;
```

```
//равен=1.65872
```

### 7.4.2 MHL\_Cosh

Функция возвращает гиперболический косинус.

Код 190. Синтаксис

```
double MHL_Cosh(double x);
```

**Входные параметры:**

x — входная переменная.

**Возвращаемое значение:**

Гиперболический косинус.

Код 191. Пример использования

```
double x=MHL_RandomUniform(0,10);

//Вызов функции
double Result=MHL_Cosh(x);

//Используем полученный результат
MHL_ShowNumber(Result,"Гиперболический косинус от x="+MHL_NumberToText(x),"равен");
//Гиперболический косинус от x=4.04968:
//равен=28.6983
```

### 7.4.3 MHL\_Cotanh

Функция возвращает гиперболический котангенс.

Код 192. Синтаксис

```
double MHL_Cotanh(double x);
```

**Входные параметры:**

x — входная переменная.

**Возвращаемое значение:**

Гиперболический котангенс.

Код 193. Пример использования

```
double x=MHL_RandomUniform(0,10);

//Вызов функции
double Result=MHL_Cotanh(x);

//Используем полученный результат
MHL_ShowNumber(Result,"Гиперболический котангенс от x="+MHL_NumberToText(x),"равен");
//Гиперболический котангенс от x=1.92505:
//равен=1.04348
```

#### 7.4.4 MHL\_Sech

Функция возвращает гиперболический секанс.

Код 194. Синтаксис

```
double MHL_Sech(double x);
```

**Входные параметры:**

x — входная переменная.

**Возвращаемое значение:**

Гиперболический секанс.

Код 195. Пример использования

```
double x=MHL_RandomUniform(0,10);

//Вызов функции
double Result=MHL_Sech(x);

//Используем полученный результат
MHL_ShowNumber(Result, "Гиперболический секанс от x="+MHL_NumberToText(x), "равен");
//Гиперболический секанс от x=0.679932:
//равен=0.806323
```

#### 7.4.5 MHL\_Sinh

Функция возвращает гиперболический синус.

Код 196. Синтаксис

```
double MHL_Sinh(double x);
```

**Входные параметры:**

x — входная переменная.

**Возвращаемое значение:**

Гиперболический синус.

Код 197. Пример использования

```
double x=MHL_RandomUniform(0,10);

//Вызов функции
double Result=MHL_Sinh(x);

//Используем полученный результат
MHL_ShowNumber(Result, "Гиперболический синус от x="+MHL_NumberToText(x), "равен");
//Гиперболический синус от x=0.166321:
//равен=0.167089
```

#### 7.4.6 MHL\_Tanh

Функция возвращает гиперболический тангенс.

Код 198. Синтаксис

```
double MHL_Tanh(double x);
```

**Входные параметры:**

x — входная переменная.

**Возвращаемое значение:**

Гиперболический тангенс.

Код 199. Пример использования

```
double x=MHL_RandomUniform(0,10);

//Вызов функции
double Result=MHL_Tanh(x);

//Используем полученный результат
MHL>ShowNumber(Result,"Гиперболический тангенс от x="+MHL_NumberToText(x),"равен");
//Гиперболический тангенс от x=4.27643:
//равен=0.999614
```

### 7.5 Дифференцирование

#### 7.5.1 MHL\_CenterDerivative

Численное значение производной в точке (центральной разностной производной с шагом 2h).

Код 200. Синтаксис

```
double MHL_CenterDerivative(double x, double h, double (*Function)(double));
```

**Входные параметры:** x — точка, в которой считается производная;

h — малое приращение x;

Function — функция, производная которой ищется.

**Возвращаемое значение:**

Значение производной в точке.

**Примечание:**

При  $h \leq 0$  возвращается 0.

**Формула:**

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2 \cdot h}$$

Будем использовать в примере использования дополнительную функцию.

#### Код 201. Дополнительная функция

```
double Func3(double x)
{
    return x*x;
}
//-----
```

#### Код 202. Пример использования

```
double x;
double h;
double dfdx;
//Зададим случайные координаты отрезков
x=MHL_RandomUniform(-3,5);
h=0.01; //малое приращение x

//Вызов функции
dfdx=MHL_CenterDerivative(x,h,Func3);

//Используем полученный результат
MHL_ShowNumber (x,"Точка, в которой считается производная", "x");
//Точка, в которой считается производная:
//x=0.843262
MHL_ShowNumber (h,"Малое приращение x", "h");
// Малое приращение x:
//h=0.01
MHL_ShowNumber (dfdx,"Значение производной в точке", "dfdx");
// Значение производной в точке:
//dfdx=1.68652
```

### 7.5.2 MHL\_LeftDerivative

Численное значение производной в точке (разностная производная влево).

#### Код 203. Синтаксис

```
double MHL_LeftDerivative(double x, double h, double (*Function)(double));
```

**Входные параметры:** x — точка, в которой считается производная;

h — малое приращение x;

Function — функция, производная которой ищется.

**Возвращаемое значение:**

Значение производной в точке.

**Примечание:**

При  $h \leq 0$  возвращается 0.

**Формула:**

$$f'(x) \approx \frac{f(x) - f(x-h)}{h}$$

Будем использовать в примере использования дополнительную функцию.

Код 204. Дополнительная функция

```
double Func3(double x)
{
    return x*x;
}
//-----
```

Код 205. Пример использования

```
double x;
double h;
double dfdx;
//Зададим случайные координаты отрезков
x=MHL_RandomUniform(-3,5);
h=0.01; //малое приращение x

//Вызов функции
dfdx=MHL_LeftDerivative(x,h,Func3);

//Используем полученный результат
MHL_ShowNumber (x,"Точка, в которой считается производная", "x");
// Точка, в которой считается производная:
//x=1.87964
MHL_ShowNumber (h,"Малое приращение x", "h");
// Малое приращение x:
//h=0.01
MHL_ShowNumber (dfdx,"Значение производной в точке", "dfdx");
// Значение производной в точке:
//dfdx=3.74928
```

### 7.5.3 MHL\_RightDerivative

Численное значение производной в точке (разностная производная вправо).

Код 206. Синтаксис

```
double MHL_RightDerivative(double x, double h, double (*Function)(double));
```

**Входные параметры:** x — точка, в которой считается производная;

h — малое приращение x;

Function — функция, производная которой ищется.

**Возвращаемое значение:**

Значение производной в точке.

**Примечание:**

При  $h \leq 0$  возвращается 0.

**Формула:**

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$

Будем использовать в примере использования дополнительную функцию.

Код 207. Дополнительная функция

```
double Func3(double x)
{
    return x*x;
}
//-----
```

Код 208. Пример использования

```
double x;
double h;
double dfdx;
//Зададим случайные координаты отрезков
x=MHL_RandomUniform(-3,5);
h=0.01; //малое приращение x

//Вызов функции
dfdx=MHL_RightDerivative(x,h,Func3);

//Используем полученный результат
MHL>ShowNumber (x,"Точка, в которой считается производная", "x");
// Точка, в которой считается производная:
//x=-1.69409
MHL>ShowNumber (h,"Малое приращение x", "h");
// Малое приращение x:
//h=0.01
MHL>ShowNumber (dfdx,"Значение производной в точке", "dfdx");
// Значение производной в точке:
//dfdx=-3.37818
```

## 7.6 Для тестовых функций

### 7.6.1 MHL\_ClassOfTestFunction

Функция выдает принадлежность тестовой функции к классу функций: бинарной, вещественной или иной оптимизации.

Код 209. Синтаксис

```
int MHL_ClassOfTestFunction(TypeOfTestFunction Type);
```

#### Входные параметры:

Type — тип тестовой функции. Смотреть виды в переменных перечисляемого типа в начале HarrixMathLibrary.h файла.

#### Возвращаемое значение:

Класс тестовой функции:

1. 1 - бинарной оптимизации;
2. 2 - вещественной оптимизации.

Код 210. Пример использования

```
TypeOfTestFunction Type=TestFunction_Ackley;

//Вызов функции
int ClassOfTestFunction=MHL_ClassOfTestFunction(Type);

//используем результат
if (ClassOfTestFunction==1)
    MHL_ShowText("Это задача бинарной оптимизации");
if (ClassOfTestFunction==2)
    MHL_ShowText("Это задача вещественной оптимизации");
//Это задача вещественной оптимизации.
```

## 7.6.2 MHL\_CountOfFitnessOfTestFunction\_Binary

Функция определяет количество вычислений целевой функции для тестовых задач для единообразного сравнения алгоритмов. Включает в себя все тестовые функции вещественной оптимизации.

Код 211. Синтаксис

```
int MHL_CountOfFitnessOfTestFunction_Binary(int Dimension);
int MHL_CountOfFitnessOfTestFunction_Binary(int Dimension, TypeOfTestFunction Type);
```

### Входные параметры:

Dimension — размерность тестовой задачи. Может принимать значения: 20; 30; 40; 50; 60; 70; 80; 90; 100; 200.

В переопределяемой функции также есть параметр:

Type — обозначение тестовой функции, которую вызываем.

Смотреть виды в переменных перечисляемого типа в начале HarrixMathLibrary.h файла: TestFunction\_Ackley, TestFunction\_ParaboloidOfRevolution, TestFunction\_Rastrigin и др. Они совпадают с названиями одноименных тестовых функций, но без приставки **MHL\_**.

### Возвращаемое значение:

Количество вычислений целевой функции для тестовых задач.

Итак, для обычного использования (без параметра Type) нужно вызвать функцию MHL\_DefineTestFunction. Иначе использовать переопределенную функцию и самому указать тип тестовой функции.

Код 212. Пример использования

```
MHL_DefineTestFunction(TestFunction_SumVector);

int Dimension = 30;

//Вызов функции
int N=MHL_CountOfFitnessOfTestFunction_Binary(Dimension);

//Использование результата
MHL>ShowNumber(N,"Количество вычислений целевой функции для TestFunction_SumVector пр
и размерности 30","N");
```

```
//Количество вычислений целевой функции для TestFunction_SumVector при размерности  
30:  
//N=400
```

### 7.6.3 MHL\_CountOfFitnessOfTestFunction\_Real

Функция определяет количество вычислений целевой функции для тестовых задач для единообразного сравнения алгоритмов. Включает в себя все тестовые функции вещественной оптимизации.

Код 213. Синтаксис

```
int MHL_CountOfFitnessOfTestFunction_Real(int Dimension);  
int MHL_CountOfFitnessOfTestFunction_Real(int Dimension, TypeOfTestFunction Type);
```

#### Входные параметры:

Dimension — размерность тестовой задачи. Может принимать значения: 2; 3; 4; 5; 10; 20; 30.

В переопределяемой функции также есть параметр:

Type — обозначение тестовой функции, которую вызываем.

Смотреть виды в переменных перечисляемого типа в начале HarrixMathLibrary.h файла: TestFunction\_Ackley, TestFunction\_ParaboloidOfRevolution, TestFunction\_Rastrigin и др. Они совпадают с названиями одноименных тестовых функций, но без приставки **MHL\_**.

#### Возвращаемое значение:

Количество вычислений целевой функции для тестовых задач.

Итак, для обычного использования (без параметра Type) нужно вызвать функцию MHL\_DefineTestFunction. Иначе использовать переопределенную функцию и самому указать тип тестовой функции.

Код 214. Пример использования

```
MHL_DefineTestFunction(TestFunction_Ackley);  
  
int Dimension = 3;  
  
//Вызов функции  
int N=MHL_CountOfFitnessOfTestFunction_Real(Dimension);  
  
//Использование результата  
MHL_ShowNumber(N, "Количество вычислений целевой функции для TestFunction_Ackley при р  
азмерности 3", "N");  
//Количество вычислений целевой функции для TestFunction_Ackley при размерности 3:  
//N=729
```

### 7.6.4 MHL\_DefineTestFunction

Служебная функция определяет тестовую функцию для других функций по работе с тестовыми функциями.

Код 215. Синтаксис

```
void MHL_DefineTestFunction(TypeOfTestFunction Type);
```

#### Входные параметры:

Type — обозначение тестовой функции, которую вызываем. Смотреть виды в переменных перечисляемого типа в начале HarrixMathLibrary.h файла: TestFunction\_Ackley, TestFunction\_ParaboloidOfRevolution, TestFunction\_Rastrigin и др. Они совпадают с названиями одноименных тестовых функций, но без приставки **MHL\_**.

#### Возвращаемое значение:

Отсутствует.

Код 216. Пример использования

```
//Вызов функции
MHL_DefineTestFunction(TestFunction_SumVector);

//Использование результата
int N=5;
int *x=new int[N];
TMHL_RandomBinaryVector(x,N);
double f=MHL_TestFunction_Binary(x,N);

MHL_ShowVectorT(x,N, "Решение", "x");
//Решение:
//x =
//1   1   1   1   0

MHL>ShowNumber(f, "Значение целевой функции", "f");
//Значение целевой функции:
//f=4
```

### 7.6.5 MHL\_DimensionTestFunction\_Binary

Функция определяет размерность тестовой задачи для тестовой функции бинарной оптимизации по номеру подзадачи (число подзадач по функции MHL\_GetCountOfSubProblems\_Binary).

Код 217. Синтаксис

```
int MHL_DimensionTestFunction_Binary(int i);
int MHL_DimensionTestFunction_Binary(int i, TypeOfTestFunction Type);
```

#### Входные параметры:

i - номер подзадачи (начиная с нуля).

В переопределяемой функции также есть параметр:

Type — обозначение тестовой функции, которую вызываем.

Смотреть виды в переменных перечисляемого типа в начале HarrixMathLibrary.h файла: TestFunction\_Ackley, TestFunction\_ParaboloidOfRevolution, TestFunction\_Rastrigin и др. Они совпадают с названиями одноименных тестовых функций, но без приставки **MHL\_**.

### **Возвращаемое значение:**

Размерность тестовой задачи для тестовой функции бинарной оптимизации.

Итак, для обычного использования (без параметра Type) нужно вызвать функцию MHL\_DefineTestFunction. Иначе использовать переопределенную функцию и самому указать тип тестовой функции.

Код 218. Пример использования

```
MHL_DefineTestFunction(TestFunction_SumVector);

//Вызов функции
double N=MHL_DimensionTestFunction_Binary(0);

//Использование результата
MHL_ShowNumber(N, "Размерность тестовой задачи для TestFunction_SumVector при i=0", "N");
//Размерность тестовой задачи для TestFunction_SumVector при i=0:
//N=20
```

### **7.6.6 MHL\_DimensionTestFunction\_Real**

Функция определяет размерность тестовой задачи для тестовой функции вещественной оптимизации по номеру подзадачи (число подзадач по функции MHL\_GetCountOfSubProblems\_Binary).

Код 219. Синтаксис

```
int MHL_DimensionTestFunction_Real(int i);
int MHL_DimensionTestFunction_Real(int i, TypeOfTestFunction Type);
```

### **Входные параметры:**

**i** - номер подзадачи (начиная с нуля).

В переопределяемой функции также есть параметр:

Type — обозначение тестовой функции, которую вызываем.

Смотреть виды в переменных перечисляемого типа в начале HarrixMathLibrary.h файла: TestFunction\_Ackley, TestFunction\_ParaboloidOfRevolution, TestFunction\_Rastrigin и др. Они совпадают с названиями одноименных тестовых функций, но без приставки **MHL\_**.

### **Возвращаемое значение:**

Размерность тестовой задачи для тестовой функции вещественной оптимизации.

Итак, для обычного использования (без параметра Type) нужно вызвать функцию MHL\_DefineTestFunction. Иначе использовать переопределенную функцию и самому указать тип тестовой функции.

Код 220. Пример использования

```
MHL_DefineTestFunction(TestFunction_Ackley);

//Вызов функции
double N=MHL_DimensionTestFunction_Real(0);
```

```

//Использование результата
MHL_ShowNumber(N, "Размерность тестовой задачи для TestFunction_Ackley при i=0", "N");
//Размерность тестовой задачи для TestFunction_Ackley при i=0:
//N=2

```

## 7.6.7 MHL\_ErrorExOfTestFunction\_Binary

Функция определяет значение ошибки по входным параметрам найденного решения в задаче оптимизации для тестовой функции. Включает в себя все тестовые функции бинарной оптимизации. Есть функция-переопределение, где пользователь может сам указать тип тестовой функции.

Код 221. Синтаксис

```

double MHL_ErrorExOfTestFunction_Binary(int *x, int VMHL_N);
double MHL_ErrorExOfTestFunction_Binary(int *x, int VMHL_N, TypeOfTestFunction Type);

```

### Входные параметры:

*x* — указатель на исходный массив (найденное решение алгоритмом);

*VMHL\_N* — размер массива *x*.

В переопределяемой функции также есть параметр:

*Type* — обозначение тестовой функции, которую вызываем. Смотреть виды в переменных перечисляемого типа в начале HarrixMathLibrary.h файла: *TestFunction\_Ackley*, *TestFunction\_ParaboloidOfRevolution*, *TestFunction\_Rastrigin* и др. Они совпадают с названиями одноименных тестовых функций, но без приставки **MHL\_**.

### Возвращаемое значение:

Значение ошибки по входным параметрам *Ex*.

Итак, для обычного использования (без параметра *Type*) нужно вызвать функцию *MHL\_DefineTestFunction*. Иначе использовать переопределенную функцию и самому указать тип тестовой функции.

Конкретную формулу, которые используются для нахождения для каждой тестовой функции, смотрите в функциях этих тестовых функций. Обратите внимание, что данная функция находит ошибку только для одного решения, тогда как по формулам нужно множество решений.

Код 222. Пример использования

```

MHL_DefineTestFunction(TestFunction_SumVector);

int N=5;
int *x=new int[N];
TMHL_RandomBinaryVector(x,N);

//Вызов функции
double Ex=MHL_ErrorExOfTestFunction_Binary(x,N);

//Использование результата
MHL_ShowVectorT(x,N,"Решение", "x");

```

```

//Решение:
//x =
//1 0 1 1 1

MHL_ShowNumber(Ex, "Значение ошибки по входным параметрам", "E<sub>x</sub>");
//Значение ошибки по входным параметрам:
//Ex=1

```

### 7.6.8 MHL\_ErrorExOfTestFunction\_Real

Функция определяет значение ошибки по входным параметрам найденного решения в задаче оптимизации для тестовой функции вещественной оптимизации. Включает в себя все тестовые функции вещественной оптимизации. Есть функция-переопределение, где пользователь может сам указать тип тестовой функции.

Код 223. Синтаксис

```

double MHL_ErrorExOfTestFunction_Real(double *x, int VMHL_N);
double MHL_ErrorExOfTestFunction_Real(double *x, int VMHL_N, TypeOfTestFunction Type)
;
```

#### Входные параметры:

x — указатель на исходный массив (найденное решение алгоритмом);

VMHL\_N — размер массива x.

В переопределяемой функции также есть параметр:

Type — обозначение тестовой функции, которую вызываем. Смотреть виды в переменных перечисляемого типа в начале HarrixMathLibrary.h файла: TestFunction\_Ackley, TestFunction\_ParaboloidOfRevolution, TestFunction\_Rastrigin и др. Они совпадают с названиями одноименных тестовых функций, но без приставки **MHL\_**.

#### Возвращаемое значение:

Значение ошибки по входным параметрам Ex.

Итак, для обычного использования (без параметра Type) нужно вызвать функцию MHL\_DefineTestFunction. Иначе использовать переопределенную функцию и самому указать тип тестовой функции.

Конкретную формулу, которые используются для нахождения для каждой тестовой функции, смотрите в функциях этих тестовых функций. Обратите внимание, что данная функция находит ошибку только для одного решения, тогда как по формулам нужно множество решений.

Код 224. Пример использования

```

MHL_DefineTestFunction(TestFunction_Ackley);

int N=5;
double *x=new double[N];
MHL_RandomRealVector(x, -0.5, 0.05, N);

//Вызов функции
double Ex=MHL_ErrorExOfTestFunction_Real(x,N);

```

```

//Использование результата
MHL_ShowVectorT(x,N,"Решение","x");
//Решение:
//x =
//-0.43694 -0.458693 -0.0266388 0.0117142 -0.136948

MHL_ShowNumber(Ex,"Значение ошибки по входным параметрам","E<sub>x</sub>");
//Значение ошибки по входным параметрам:
//Ex=0.129756

```

### 7.6.9 MHL\_ErrorEyOfTestFunction\_Binary

Функция определяет значение ошибки по значениям целевой функции найденного решения в задаче оптимизации для тестовой функции. Включает в себя все тестовые функции бинарной оптимизации. Есть функция-переопределение, где пользователь может сам указать тип тестовой функции.

Код 225. Синтаксис

```

double MHL_ErrorEyOfTestFunction_Binary(double FitnessOfx, int VMHL_N);
double MHL_ErrorEyOfTestFunction_Binary(double FitnessOfx, int VMHL_N,
                                         TypeOfTestFunction Type);

```

#### Входные параметры:

FitnessOfx — значение целевой функции найденного решения алгоритмом оптимизации;

VMHL\_N — размер массива x.

В переопределяемой функции также есть параметр:

Type — обозначение тестовой функции, которую вызываем. Смотреть виды в переменных перечисляемого типа в начале HarrixMathLibrary.h файла: TestFunction\_Ackley, TestFunction\_ParaboloidOfRevolution, TestFunction\_Rastrigin и др. Они совпадают с названиями одноименных тестовых функций, но без приставки **MHL\_**.

#### Возвращаемое значение:

Значение ошибки по значениям целевой функции Ey.

Итак, для обычного использования (без параметра Type) нужно вызвать функцию MHL\_DefineTestFunction. Иначе использовать переопределенную функцию и самому указать тип тестовой функции.

Конкретную формулу, которые используются для нахождения для каждой тестовой функции, смотрите в функциях этих тестовых функций. Обратите внимание, что данная функция находит ошибку только для одного решения, тогда как по формулам нужно множество решений.

Все функции так высчитываются, чтобы алгоритм решал задачу поиска максимального значения целевой функции, поэтому тестовые функции на минимум умножаются на  $-1$ . Поэтому, фактически алгоритмы оптимизации находят максимум перевернутой функции. А значит, чтобы правильно посчитать ошибку по значениям целевой функции, нужно найденное решение умножить на  $-1$ .

Код 226. Пример использования

```
MHL_DefineTestFunction(TestFunction_SumVector);

int N=5;
int *x=new int[N];
TMHL_RandomBinaryVector(x,N);
double f=MHL_TestFunction_Binary(x,N);

//Вызов функции
double Ey=MHL_ErrorEyOfTestFunction_Binary(f,N);

//Использование результата
MHL_ShowVectorT(x,N,"Решение","x");
//Решение:
//x =
//0   1   1   0   1

MHL>ShowNumber(Ey,"Значение ошибки по значениям целевой функции","E<sub>y</sub>");
//Значение ошибки по значениям целевой функции:
//Ey=2
```

### 7.6.10 MHL\_ErrorEyOfTestFunction\_Real

Функция определяет значение ошибки по значениям целевой функции найденного решения в задаче оптимизации для тестовой функции вещественной оптимизации. Включает в себя все тестовые функции вещественной оптимизации. Есть функция- переопределение, где пользователь может сам указать тип тестовой функции.

Код 227. Синтаксис

```
double MHL_ErrorEyOfTestFunction_Real(double FitnessOfx, int VMHL_N);
double MHL_ErrorEyOfTestFunction_Real(double FitnessOfx, int VMHL_N,
TypeOfTestFunction Type);
```

#### Входные параметры:

FitnessOfx — значение целевой функции найденного решения алгоритмом оптимизации;

VMHL\_N — размер массива x.

В переопределяемой функции также есть параметр:

Type — обозначение тестовой функции, которую вызываем. Смотреть виды в переменных перечисляемого типа в начале HarrixMathLibrary.h файла: TestFunction\_Ackley, TestFunction\_ParaboloidOfRevolution, TestFunction\_Rastrigin и др. Они совпадают с названиями одноименных тестовых функций, но без приставки **MHL\_**.

#### Возвращаемое значение:

Значение ошибки по значениям целевой функции Ey.

Итак, для обычного использования (без параметра Type) нужно вызвать функцию MHL\_DefineTestFunction. Иначе использовать переопределенную функцию и самому указать тип тестовой функции.

Конкретную формулу, которые используются для нахождения для каждой тестовой функции, смотрите в функциях этих тестовых функций. Обратите внимание, что данная функция находит ошибку только для одного решения, тогда как по формулам нужно множество решений.

Все функции так высчитываются, чтобы алгоритм решал задачу поиска максимального значения целевой функции, поэтому тестовые функции на минимум умножаются на  $-1$ . Поэтому, фактически алгоритмы оптимизации находят максимум перевернутой функции. А значит, чтобы правильно посчитать ошибку по значениям целевой функции, нужно найденное решение умножить на  $-1$ .

Код 228. Пример использования

```
MHL_DefineTestFunction(TestFunction_Ackley);

int N=5;
double *x=new double[N];
MHL_RandomRealVector(x,-0.5,0.05,N);
double f=MHL_TestFunction_Real(x,N);

//Вызов функции
double Ey=MHL_ErrorOfTestFunction_Real(f,N);

//Использование результата
MHL_ShowVectorT(x,N,"Решение", "x");
//Решение:
//x =
// -0.0963959   -0.183693   -0.0485428   -0.185757   0.0321075

MHL>ShowNumber(Ey,"Значение ошибки по значениям целевой функции","Ey");
//Значение ошибки по значениям целевой функции:
//Ey=1.18549y
```

### 7.6.11 MHL\_ErrorROfTestFunction\_Binary

Функция определяет значение надежности найденного решения в задаче оптимизации для тестовой функции. Включает в себя все тестовые функции бинарной оптимизации. Есть функция- переопределение, где пользователь может сам указать тип тестовой функции.

Код 229. Синтаксис

```
double MHL_ErrorROfTestFunction_Binary(int *x, int VMHL_N);
double MHL_ErrorROfTestFunction_Binary(int *x, int VMHL_N, TypeOfTestFunction Type);
```

#### Входные параметры:

$x$  — указатель на исходный массив (найденное решение алгоритмом);

$VMHL\_N$  — размер массива  $x$ .

В переопределяемой функции также есть параметр:

$Type$  — обозначение тестовой функции, которую вызываем. Смотреть виды в переменных перечисляемого типа в начале HarrixMathLibrary.h файла: `TestFunction_Ackley`, `TestFunction_ParaboloidOfRevolution`, `TestFunction_Rastrigin` и др. Они совпадают с названиями одноименных тестовых функций, но без приставки **MHL\_**.

### **Возвращаемое значение:**

Значение надежности R.

Итак, для обычного использования (без параметра Type) нужно вызвать функцию MHL\_DefineTestFunction. Иначе использовать переопределенную функцию и самому указать тип тестовой функции.

Конкретную формулу, которые используются для нахождения для каждой тестовой функции, смотрите в функциях этих тестовых функций. Обратите внимание, что данная функция находит ошибку только для одного решения, тогда как по формулам нужно множество решений.

#### Код 230. Пример использования

```
MHL_DefineTestFunction(TestFunction_SumVector);

int N=5;
int *x=new int[N];
TMHL_RandomBinaryVector(x,N);

//Вызов функции
double R=MHL_ErrorROfTestFunction_Binary(x,N);

//Использование результата
MHL_ShowVectorT(x,N,"Решение","x");
//Решение:
//x =
//1   1   1   1

MHL_ShowNumber(R,"Значение надежности","R");
//Значение надежности:
//R=1
```

### **7.6.12 MHL\_ErrorROfTestFunction\_Real**

Функция определяет значение надежности найденного решения в задаче оптимизации для тестовой функции вещественной оптимизации. Включает в себя все тестовые функции вещественной оптимизации. Есть функция-переопределение, где пользователь может сам указать тип тестовой функции.

#### Код 231. Синтаксис

```
double MHL_ErrorROfTestFunction_Real(double *x, int VMHL_N);
double MHL_ErrorROfTestFunction_Real(double *x, int VMHL_N, TypeOfTestFunction Type);
```

### **Входные параметры:**

x — указатель на исходный массив (найденное решение алгоритмом);

VMHL\_N — размер массива x.

В переопределяемой функции также есть параметр:

Type — обозначение тестовой функции, которую вызываем. Смотреть виды в переменных перечисляемого типа в начале HarrixMathLibrary.h файла: TestFunction\_Ackley,

TestFunction\_ParaboloidOfRevolution, TestFunction\_Rastrigin и др. Они совпадают с названиями одноименных тестовых функций, но без приставки **MHL\_**.

#### **Возвращаемое значение:**

Значение надежности R.

Итак, для обычного использования (без параметра Type) нужно вызвать функцию MHL\_DefineTestFunction. Иначе использовать переопределенную функцию и самому указать тип тестовой функции.

Конкретную формулу, которые используются для нахождения для каждой тестовой функции, смотрите в функциях этих тестовых функций. Обратите внимание, что данная функция находит ошибку только для одного решения, тогда как по формулам нужно множество решений.

#### Код 232. Пример использования

```
MHL_DefineTestFunction(TestFunction_Ackley);

int N=5;
double *x=new double[N];
MHL_RandomRealVector(x,0.01,0.02,N);

//Вызов функции
double R=MHL_ErrorROfTestFunction_Real(x,N);

//Использование результата
MHL_ShowVectorT(x,N,"Решение","x");
//Решение:
//x =
//0.0118939 0.0177618 0.0115656 0.0181937 0.0124084

MHL_ShowNumber(R,"Значение надежности","R");
//Значение надежности:
//R=1
```

### 7.6.13 MHL\_FitnessOfOptimumOfTestFunction\_Binary

Функция определяет значение целевой функции в оптимуме для тестовой функции бинарной оптимизации. Включает в себя все тестовые функции бинарной оптимизации. Есть функция-переопределение, где пользователь может сам указать тип тестовой функции.

#### Код 233. Синтаксис

```
double MHL_FitnessOfOptimumOfTestFunction_Binary(int VMHL_N);
double MHL_FitnessOfOptimumOfTestFunction_Binary(int VMHL_N, TypeOfTestFunction Type)
;
```

#### **Входные параметры:**

VMHL\_N — размер массива x.

В переопределяемой функции также есть параметр:

Type — обозначение тестовой функции, которую вызываем. Смотреть виды в переменных перечисляемого типа в начале HarrixMathLibrary.h файла: TestFunction\_Ackley,

TestFunction\_ParaboloidOfRevolution, TestFunction\_Rastrigin и др. Они совпадают с названиями одноименных тестовых функций, но без приставки **MHL\_**.

#### **Возвращаемое значение:**

Значение тестовой функции в оптимальной точке.

Итак, для обычного использования (без параметра Type) нужно вызвать функцию MHL\_DefineTestFunction. Иначе использовать переопределенную функцию и самому указать тип тестовой функции.

Код 234. Пример использования

```
MHL_DefineTestFunction(TestFunction_SumVector);

int N=5;

//Вызов функции
double f=MHL_FitnessOfOptimumOfTestFunction_Binary(N);

//Использование результата
MHL_ShowNumber(f, "Значение целевой функции оптимального решения", "f");
//Значение целевой функции оптимального решения:
//f=5
```

### **7.6.14 MHL\_FitnessOfOptimumOfTestFunction\_Real**

Функция определяет значение целевой функции в оптимуме для тестовой функции вещественной оптимизации. Включает в себя все тестовые функции вещественной оптимизации. Есть функция-переопределение, где пользователь может сам указать тип тестовой функции.

Код 235. Синтаксис

```
double MHL_FitnessOfOptimumOfTestFunction_Real(double VMHL_N);
double MHL_FitnessOfOptimumOfTestFunction_Real(double VMHL_N, TypeOfTestFunction Type
);
```

#### **Входные параметры:**

VMHL\_N — размер массива x.

В переопределяемой функции также есть параметр:

Type — обозначение тестовой функции, которую вызываем. Смотреть виды в переменных перечисляемого типа в начале HarrixMathLibrary.h файла: TestFunction\_Ackley, TestFunction\_ParaboloidOfRevolution, TestFunction\_Rastrigin и др. Они совпадают с названиями одноименных тестовых функций, но без приставки **MHL\_**.

#### **Возвращаемое значение:**

Значение тестовой функции в оптимальной точке.

Итак, для обычного использования (без параметра Type) нужно вызвать функцию MHL\_DefineTestFunction. Иначе использовать переопределенную функцию и самому указать тип тестовой функции.

Код 236. Пример использования

```

MHL_DefineTestFunction(TestFunction_Ackley);

int N=5;

//Вызов функции
double f=MHL_FitnessOfOptimumOfTestFunction_Binary(N);

//Использование результата
MHL_ShowNumber(f,"Значение целевой функции оптимального решения функции
    TestFunction_Ackley","f");
//Значение целевой функции оптимального решения функции TestFunction_Ackley:
//f=0

```

### 7.6.15 MHL\_GetCountOfFitness

Функция выдает количество вызовов целевой функции.

Код 237. Синтаксис

```
int MHL_GetCountOfFitness();
```

**Входные параметры:**

Отсутствуют.

**Возвращаемое значение:**

Количество вызовов целевой функции.

Данную функцию надо использовать в связке с функцией MHL\_GetCountOfFitness(). Для чего использовать эти функции? Дело в том, что для сравнения алгоритмов оптимизации очень критично оценивать вызов целевой функции. И часто многие программисты пишут или не очень аккуратно, или логика алгоритма такая, что заявленное число вычислений функций не совпадает с действительным. Поэтому в общие тестовые функции (например, MHL\_TestFunction\_Binary) вшил подсчет числа вызовов целевой функции.

MHL\_SetToZeroCountOfFitness — эту функцию вызываем перед вызовом какого-то алгоритма оптимизации, а MHL\_GetCountOfFitness — после его работы.

Код 238. Пример использования

```

MHL_DefineTestFunction(TestFunction_SumVector);

MHL_SetToZeroCountOfFitness();

int N=5;
double f=0;
int *x=new int[N];

for (int i=0;i<10;i++)
{
    TMHL_RandomBinaryVector(x,N);
    f+=MHL_TestFunction_Binary(x,N);
}

f/=double(10.);

//Вызов функции

```

```

int M=MHL_GetCountOfFitness();

//Использование результата
MHL_ShowNumber(M, "Количество вызовов целевой функции", "M");
//Количество вызовов целевой функции:
//M=10

MHL_ShowNumber(f, "Среднее значение целевой функции", "f");
//Среднее значение целевой функции:
//f=2.6

```

### 7.6.16 MHL\_GetCountOfSubProblems\_Binary

Функция определяет число подзадач (включая основную задачу) для тестовой функции бинарной оптимизации. Включает в себя все тестовые функции бинарной оптимизации.

Код 239. Синтаксис

```

int MHL_GetCountOfSubProblems_Binary();
int MHL_GetCountOfSubProblems_Binary(TypeOfTestFunction Type);

```

#### Входные параметры:

Отсутствуют.

В переопределяемой функции также есть параметр:

Type — обозначение тестовой функции, которую вызываем.

Смотреть виды в переменных перечисляемого типа в начале HarrixMathLibrary.h файла: TestFunction\_Ackley, TestFunction\_ParaboloidOfRevolution, TestFunction\_Rastrigin и др. Они совпадают с названиями одноименных тестовых функций, но без приставки **MHL\_**.

#### Возвращаемое значение:

Число подзадач (включая основную задачу) для тестовой функции.

Итак, для обычного использования (без параметра Type) нужно вызвать функцию MHL\_DefineTestFunction. Иначе использовать переопределенную функцию и самому указать тип тестовой функции.

Код 240. Пример использования

```

MHL_DefineTestFunction(TestFunction_SumVector);

//Вызов функции
double N=MHL_GetCountOfSubProblems_Binary();

//Использование результата
MHL_ShowNumber(N, "Число подзадач (включая основную задачу) для TestFunction_SumVector
", "N");
//Число подзадач (включая основную задачу) для TestFunction_SumVector:
//N=10

```

## 7.6.17 MHL\_GetCountOfSubProblems\_Real

Функция определяет число подзадач (включая основную задачу) для тестовой функции вещественной оптимизации. Включает в себя все тестовые функции вещественной оптимизации.

Код 241. Синтаксис

```
int MHL_GetCountOfSubProblems_Real();
int MHL_GetCountOfSubProblems_Real(TypeOfTestFunction Type);
```

### Входные параметры:

Отсутствуют.

В переопределяемой функции также есть параметр:

Type — обозначение тестовой функции, которую вызываем.

Смотреть виды в переменных перечисляемого типа в начале HarrixMathLibrary.h файла: TestFunction\_Ackley, TestFunction\_ParaboloidOfRevolution, TestFunction\_Rastrigin и др. Они совпадают с названиями одноименных тестовых функций, но без приставки **MHL\_**.

### Возвращаемое значение:

Число подзадач (включая основную задачу) для тестовой функции.

Итак, для обычного использования (без параметра Type) нужно вызвать функцию MHL\_DefineTestFunction. Иначе использовать переопределенную функцию и самому указать тип тестовой функции.

Код 242. Пример использования

```
MHL_DefineTestFunction(TestFunction_Ackley);

//Вызов функции
double N=MHL_GetCountOfSubProblems_Real();

//Использование результата
MHL_ShowNumber(N, "Число подзадач (включая основную задачу) для TestFunction_Ackley", "N");
//Число подзадач (включая основную задачу) для TestFunction_Ackley:
//N=7
```

## 7.6.18 MHL\_LeftBorderOfTestFunction\_Real

Функция определяет левые и правые границы допустимой области для тестовой функции вещественной оптимизации. Включает в себя все тестовые функции вещественной оптимизации. Есть функция-переопределение, где пользователь может сам указать тип тестовой функции.

Код 243. Синтаксис

```
void MHL_LeftAndRightBorderOfTestFunction_Real(double *Left, double *Right, int VMHL_N );
void MHL_LeftAndRightBorderOfTestFunction_Real(double *Left, double *Right, int VMHL_N, TypeOfTestFunction Type);
```

### Входные параметры:

Left — указатель на массив, куда будет записываться результат левых границ допустимой области;

Right — указатель на массив, куда будет записываться результат левых границ допустимой области;

VMHL\_N — размер массива x.

В переопределяемой функции также есть параметр:

Type — обозначение тестовой функции, которую вызываем. Смотреть виды в переменных перечисляемого типа в начале HarrixMathLibrary.h файла: TestFunction\_Ackley, TestFunction\_ParaboloidOfRevolution, TestFunction\_Rastrigin и др. Они совпадают с названиями одноименных тестовых функций, но без приставки **MHL\_**.

#### **Возвращаемое значение:**

Отсутствует.

Итак, для обычного использования (без параметра Type) нужно вызвать функцию MHL\_DefineTestFunction. Иначе использовать переопределенную функцию и самому указать тип тестовой функции.

Код 244. Пример использования

```
MHL_DefineTestFunction(TestFunction_Ackley);

int N=5;
double *Left=new double[N];
double *Right=new double[N];

//Вызов функции
MHL_LeftAndRightBorderOfTestFunction_Real(Left,Right,N);

//Использование результата
MHL_ShowVectorT(Left,N,"Левые границы допустимой области функции TestFunction_Ackley"
    , "Left");
//Левые границы допустимой области функции TestFunction_Ackley:
//Left =
//-5 -5 -5 -5 -5

MHL_ShowVectorT(Right,N,"Правые границы допустимой области функции
    TestFunction_Ackley", "Right");
//Правые границы допустимой области функции TestFunction_Ackley:
//Right =
/5 5 5 5 5
```

### **7.6.19 MHL\_MaximumOrMinimumOfTestFunction\_Binary**

Функция сообщает - ищется максимум или минимум в задаче оптимизации для тестовой функции бинарной оптимизации.

Код 245. Синтаксис

```
double MHL_MaximumOrMinimumOfTestFunction_Binary();
double MHL_MaximumOrMinimumOfTestFunction_Binary(TypeOfTestFunction Type);
```

#### **Входные параметры:**

Отсутствуют.

В переопределяемой функции также есть параметр:

Type — обозначение тестовой функции, которую вызываем. Смотреть виды в переменных перечисляемого типа в начале HarrixMathLibrary.h файла: TestFunction\_Ackley, TestFunction\_ParaboloidOfRevolution, TestFunction\_Rastrigin и др. Они совпадают с названиями одноименных тестовых функций, но без приставки **MHL\_**.

#### **Возвращаемое значение:**

1 — задача на нахождение максимума;

-1 — задача на нахождение минимума.

Итак, для обычного использования (без параметра Type) нужно вызвать функцию MHL\_DefineTestFunction. Иначе использовать переопределенную функцию и самому указать тип тестовой функции.

Код 246. Пример использования

```
MHL_DefineTestFunction(TestFunction_SumVector);

//Вызов функции
double MorM=MHL_MaximumOrMinimumOfTestFunction_Binary();

//Использование результата
MHL_ShowNumber(MorM, "Максимум или минимум функции находим у TestFunction_SumVector", "
    MorM");
//Максимум или минимум функции находим у TestFunction_SumVector:
//MorM=1
```

### 7.6.20 MHL\_MaximumOrMinimumOfTestFunction\_Real

Функция сообщает - ищется максимум или минимум в задаче оптимизации для тестовой функции вещественной оптимизации.

Код 247. Синтаксис

```
double MHL_MaximumOrMinimumOfTestFunction_Real();
double MHL_MaximumOrMinimumOfTestFunction_Real(TypeOfTestFunction Type);
```

#### **Входные параметры:**

Отсутствуют.

В переопределяемой функции также есть параметр:

Type — обозначение тестовой функции, которую вызываем. Смотреть виды в переменных перечисляемого типа в начале HarrixMathLibrary.h файла: TestFunction\_Ackley, TestFunction\_ParaboloidOfRevolution, TestFunction\_Rastrigin и др. Они совпадают с названиями одноименных тестовых функций, но без приставки **MHL\_**.

#### **Возвращаемое значение:**

1 — задача на нахождение максимума;

-1 — задача на нахождение минимума.

Итак, для обычного использования (без параметра Type) нужно вызвать функцию MHL\_DefineTestFunction. Иначе использовать переопределенную функцию и самому указать тип тестовой функции.

Код 248. Пример использования

```
MHL_DefineTestFunction(TestFunction_Ackley);

//Вызов функции
double MorM=MHL_MaximumOrMinimumOfTestFunction_Real();

//Использование результата
MHL_ShowNumber(MorM, "Максимум или минимум функции находим у TestFunction_Ackley",
    "MorM");
//Максимум или минимум функции находим у TestFunction_Ackley:
//MorM=-1
```

### 7.6.21 MHL\_NumberOfPartsOfTestFunction\_Real

Функция определяет на сколько частей нужно делить каждую координату в задаче оптимизации для тестовой функции вещественной оптимизации для алгоритма дискретной оптимизации и какая при этом требуется точность для подсчета надежности. Включает в себя все тестовые функции вещественной оптимизации. Есть функция-переопределение, где пользователь может сам указать тип тестовой функции.

Код 249. Синтаксис

```
double MHL_NumberOfPartsOfTestFunction_Real(int *NumberOfParts, int VMHL_N);
double MHL_NumberOfPartsOfTestFunction_Real(int *NumberOfParts, int VMHL_N,
    TypeOfTestFunction Type);
```

#### Входные параметры:

NumberOfParts — указатель на массив, куда будет записываться результат;

VMHL\_N — размер массива NumberOfParts.

В переопределяемой функции также есть параметр:

Type — обозначение тестовой функции, которую вызываем. Смотреть виды в переменных перечисляемого типа в начале HarrixMathLibrary.h файла: TestFunction\_Ackley, TestFunction\_ParaboloidOfRevolution, TestFunction\_Rastrigin и др. Они совпадают с названиями одноименных тестовых функций, но без приставки **MHL\_**.

#### Возвращаемое значение:

Точность вычислений.

Итак, для обычного использования (без параметра Type) нужно вызвать функцию MHL\_DefineTestFunction. Иначе использовать переопределенную функцию и самому указать тип тестовой функции.

Код 250. Пример использования

```
MHL_DefineTestFunction(TestFunction_Ackley);

int N=5;
```

```

int *NumberOfParts=new int[N];

//Вызов функции
double e=MHL_NumberOfPartsOfTestFunction_Real(NumberOfParts,N);

//Использование результата
MHL_ShowVectorT(NumberOfParts,N,"На сколько частей нужно делить каждую координату функции TestFunction_Ackley","NumberOfParts");
//На сколько частей нужно делить каждую координату функции TestFunction_Ackley:
//NumberOfParts =
//4095 4095 4095 4095 4095

MHL_ShowNumber(e,"Точность вычислений.", "e");
//Точность вычислений.:
//e=0.025

```

## 7.6.22 MHL\_OptimumOfTestFunction\_Binary

Функция определяет значение оптимума для тестовой функции. Включает в себя все тестовые функции бинарной оптимизации. Есть функция-переопределение, где пользователь может сам указать тип тестовой функции.

Код 251. Синтаксис

```

double MHL_OptimumOfTestFunction_Binary(int *Optimum, int VMHL_N);
double MHL_OptimumOfTestFunction_Binary(int *Optimum, int VMHL_N, TypeOfTestFunction Type);

```

### Входные параметры:

Optimum — указатель на исходный массив, куда будет записываться результат, то есть оптимум тестовой функции (максимум или минимум — это зависит от типа тестовой функции, что расписывается в самих функциях тестовых функций);

VMHL\_N — размер массива x.

В переопределяемой функции также есть параметр:

Type — обозначение тестовой функции, которую вызываем. Смотреть виды в переменных перечисляемого типа в начале HarrixMathLibrary.h файла: TestFunction\_Ackley, TestFunction\_ParaboloidOfRevolution, TestFunction\_Rastrigin и др. Они совпадают с названиями одноименных тестовых функций, но без приставки **MHL\_**.

### Возвращаемое значение:

Значение тестовой функции в оптимальной точке.

Итак, для обычного использования (без параметра Type) нужно вызвать функцию MHL\_DefineTestFunction. Иначе использовать переопределенную функцию и самому указать тип тестовой функции.

Код 252. Пример использования

```

MHL_DefineTestFunction(TestFunction_SumVector);

int N=5;
int *x=new int[N];

```

```

//Вызов функции
double f=MHL_OptimumOfTestFunction_Binary(x,N);

//Использование результата
MHL_ShowVectorT(x,N,"Оптимальное решение тестовой функции TestFunction_SumVector","x"
    );
//Оптимальное решение тестовой функции TestFunction_SumVector:
//x =
//1   1   1   1   1

MHL>ShowNumber(f,"Значение целевой функции оптимального решения","f");
//Значение целевой функции оптимального решения:
//f=5

```

### 7.6.23 MHL\_OptimumOfTestFunction\_Real

Функция определяет значение оптимума для тестовой функции вещественной оптимизации. Включает в себя все тестовые функции вещественной оптимизации. Есть функция-переопределение, где пользователь может сам указать тип тестовой функции.

Код 253. Синтаксис

```

double MHL_OptimumOfTestFunction_Real(double *Optimum, int VMHL_N);
double MHL_OptimumOfTestFunction_Real(double *Optimum, int VMHL_N, TypeOfTestFunction
    Type);

```

#### Входные параметры:

Optimum — указатель на исходный массив, куда будет записываться результат, то есть оптимум тестовой функции (максимум или минимум — это зависит от типа тестовой функции, что расписывается в самих функциях тестовых функций);

VMHL\_N — размер массива x.

В переопределяемой функции также есть параметр:

Type — обозначение тестовой функции, которую вызываем. Смотреть виды в переменных перечисляемого типа в начале HarrixMathLibrary.h файла: TestFunction\_Ackley, TestFunction\_ParaboloidOfRevolution, TestFunction\_Rastrigin и др. Они совпадают с названиями одноименных тестовых функций, но без приставки **MHL\_**.

#### Возвращаемое значение:

Значение тестовой функции в оптимальной точке.

Итак, для обычного использования (без параметра Type) нужно вызвать функцию MHL\_DefineTestFunction. Иначе использовать переопределенную функцию и самому указать тип тестовой функции.

Код 254. Пример использования

```

MHL_DefineTestFunction(TestFunction_Ackley);

int N=5;
double *x=new double[N];

```

```

//Вызов функции
double f=MHL_OptimumOfTestFunction_Real(x,N);

//Использование результата
MHL_ShowVectorT(x,N,"Оптимальное решение тестовой функции TestFunction_Ackley","x");
//Оптимальное решение тестовой функции TestFunction_Ackley:
//x =
//0 0 0 0

MHL>ShowNumber(f,"Значение целевой функции оптимального решения","f");
//Значение целевой функции оптимального решения:
//f=0

```

### 7.6.24 MHL\_PrecisionOfCalculationsOfTestFunction\_Real

Функция определяет точность для подсчета надежности в задаче оптимизации для тестовой функции вещественной оптимизации для алгоритма дискретной оптимизации.

Код 255. Синтаксис

```

double MHL_PrecisionOfCalculationsOfTestFunction_Real();
double MHL_PrecisionOfCalculationsOfTestFunction_Real(TypeOfTestFunction Type);

```

**Входные параметры:**

Отсутствуют.

В переопределяемой функции также есть параметр:

Type — обозначение тестовой функции, которую вызываем. Смотреть виды в переменных перечисляемого типа в начале HarrixMathLibrary.h файла: TestFunction\_Ackley, TestFunction\_ParaboloidOfRevolution, TestFunction\_Rastrigin и др. Они совпадают с названиями одноименных тестовых функций, но без приставки **MHL\_**.

**Возвращаемое значение:**

Точность вычислений.

Итак, для обычного использования (без параметра Type) нужно вызвать функцию MHL\_DefineTestFunction. Иначе использовать переопределенную функцию и самому указать тип тестовой функции.

Код 256. Пример использования

```

MHL_DefineTestFunction(TestFunction_Ackley);

//Вызов функции
double e=MHL_PrecisionOfCalculationsOfTestFunction_Real();

//Использование результата
MHL>ShowNumber(e,"Точность вычислений","e");
//Точность вычислений:
//e=0.025

```

## 7.6.25 MHL\_SetToZeroCountOfFitness

Функция обнуляет количество вызовов целевой функции. Обязательно вызвать один раз перед вызовом алгоритмов оптимизации при исследовании эффективности алгоритмов оптимизации, где требуется контроль числа вызовов целевой функции.

Код 257. Синтаксис

```
void MHL_SetToZeroCountOfFitness();
```

**Входные параметры:**

Отсутствуют.

**Возвращаемое значение:**

Отсутствует.

Данную функцию надо использовать в связке с функцией MHL\_GetCountOfFitness(). Для чего использовать эти функции? Дело в том, что для сравнения алгоритмов оптимизации очень критично оценивать вызов целевой функции. И часто многие программисты пишут или не очень аккуратно, или логика алгоритма такая, что заявленное число вычислений функций не совпадает с действительным. Поэтому в общие тестовые функции (например, MHL\_TestFunction\_Binary) вшил подсчет числа вызовов целевой функции.

MHL\_SetToZeroCountOfFitness — эту функцию вызываем перед вызовом какого-то алгоритма оптимизации, а MHL\_GetCountOfFitness — после его работы.

Код 258. Пример использования

```
MHL_DefineTestFunction(TestFunction_SumVector);

//Вызов функции
MHL_SetToZeroCountOfFitness();

//Использование результата
int N=5;
double f=0;
int *x=new int[N];

for (int i=0;i<10;i++)
{
    TMHL_RandomBinaryVector(x,N);
    f+=MHL_TestFunction_Binary(x,N);
}

f/=double(10.);

int M=MHL_GetCountOfFitness();
MHL_ShowNumber(M,"Количество вызовов целевой функции","M");
//Количество вызовов целевой функции:
//M=10

MHL_ShowNumber(f,"Среднее значение целевой функции","f");
//Среднее значение целевой функции:
//f=2.6
```

## 7.6.26 MHL\_TestFunction\_Binary

Общая тестовая функция для задач бинарной оптимизации. Есть функция-переопределение, где пользователь может сам указать тип тестовой функции.

Код 259. Синтаксис

```
double MHL_TestFunction_Binary(int *x, int VMHL_N);
double MHL_TestFunction_Binary(int *x, int VMHL_N, TypeOfTestFunction Type);
```

### Входные параметры:

x — указатель на исходный массив;

VMHL\_N — размер массива x.

В переопределяемой функции также есть параметр:

Type — обозначение тестовой функции, которую вызываем. Смотреть виды в переменных перечисляемого типа в начале HarrixMathLibrary.h файла: TestFunction\_Ackley, TestFunction\_ParaboloidOfRevolution, TestFunction\_Rastrigin и др. Они совпадают с названиями одноименных тестовых функций, но без приставки **MHL\_**.

### Возвращаемое значение:

Значение тестовой функции в точке x.

Итак, для обычного использования (без параметра Type) нужно вызвать функцию MHL\_DefineTestFunction. Иначе использовать переопределенную функцию и самому указать тип тестовой функции.

Все функции так высчитываются, чтобы алгоритм решал задачу поиска максимального значения целевой функции, поэтому тестовые функции на минимум умножаются на -1.

Код 260. Пример использования

```
MHL_DefineTestFunction(TestFunction_SumVector);

int N=5;
int *x=new int[N];
TMHL_RandomBinaryVector(x,N);

//Вызов функции
double f=MHL_TestFunction_Binary(x,N);

//Использование результата
MHL_ShowVectorT(x,N,"Решение","x");
//Решение:
//x =
//1 1 1 1 0

MHL_ShowNumber(f,"Значение целевой функции","f");
//Значение целевой функции:
//f=4
```

## 7.6.27 MHL\_TestFunction\_Real

Общая тестовая функция для задач вещественной оптимизации. Есть функция-переопределение, где пользователь может сам указать тип тестовой функции.

Код 261. Синтаксис

```
double MHL_TestFunction_Real(double *x, int VMHL_N);
double MHL_TestFunction_Real(double *x, int VMHL_N, TypeOfTestFunction Type);
```

### Входные параметры:

x — указатель на исходный массив;

VMHL\_N — размер массива x.

В переопределяемой функции также есть параметр:

Type — обозначение тестовой функции, которую вызываем. Смотреть виды в переменных перечисляемого типа в начале HarrixMathLibrary.h файла: TestFunction\_Ackley, TestFunction\_ParaboloidOfRevolution, TestFunction\_Rastrigin и др. Они совпадают с названиями одноименных тестовых функций, но без приставки **MHL\_**.

### Возвращаемое значение:

Значение тестовой функции в точке x.

Итак, для обычного использования (без параметра Type) нужно вызвать функцию MHL\_DefineTestFunction. Иначе использовать переопределенную функцию и самому указать тип тестовой функции.

Все функции так высчитываются, чтобы алгоритм решал задачу поиска максимального значения целевой функции, поэтому тестовые функции на минимум умножаются на -1.

Код 262. Пример использования

```
MHL_DefineTestFunction(TestFunction_Ackley);

int N=5;
double *x=new double[N];
MHL_RandomRealVector(x,-1,1,N);

//Вызов функции
double f=MHL_TestFunction_Real(x,N);

//Использование результата
MHL_ShowVectorT(x,N,"Решение","x");
//Решение:
//x =
// -0.391724 0.347656 0.259155 -0.544617 0.116516

MHL_ShowNumber(f,"Значение целевой функции","f");
//Значение целевой функции:
//f=3.38932
```

## 7.7 Интегрирование

### 7.7.1 MHL\_IntegralOfRectangle

Интегрирование по формуле прямоугольников с оценкой точности по правилу Рунге. Считается интеграл функции на отрезке  $[a,b]$  с погрешностью порядка Epsilon.

Код 263. Синтаксис

```
double MHL_IntegralOfRectangle(double a, double b, double Epsilon, double (*Function)(  
    double));
```

#### Входные параметры:

a — начало отрезка интегрирования;

b — конец отрезка интегрирования;

Epsilon — погрешность;

Function — подынтегральная функция.

#### Возвращаемое значение:

Значение определенного интеграла.

#### Примечание:

Значимые цифры в ответе определяются Epsilon.

Будем использовать в примере использования дополнительную подынтегральную функцию:

Код 264. Дополнительная функция

```
double Func3(double x)  
{  
    return x*x;  
}  
//-----
```

Код 265. Пример использования

```
double a=-2;  
double b=2;  
double Epsilon=0.01;  
double S;  
  
//Вызов функции  
S=MHL_IntegralOfRectangle(a,b,Epsilon,Func3);  
  
//Используем полученный результат  
MHL_ShowNumber (a,"Левая граница интегрирования", "a");  
//Левая граница интегрирования:  
//a=-2  
MHL_ShowNumber (b,"Правая граница интегрирования", "b");  
//Правая граница интегрирования:  
//b=2  
MHL_ShowNumber (S,"Интеграл", "S");  
// Интеграл:  
//S=5.32812
```

## 7.7.2 MHL\_IntegralOfSimpson

Интегрирование по формуле Симпсона с оценкой точности по правилу Рунге. Считается интеграл функции на отрезке  $[a,b]$  с погрешностью порядка Epsilon.

Код 266. Синтаксис

```
double MHL_IntegralOfSimpson(double a, double b, double Epsilon, double (*Function)(  
    double));
```

### Входные параметры:

a — начало отрезка интегрирования;

b — конец отрезка интегрирования;

Epsilon — погрешность;

Function — подынтегральная функция.

### Возвращаемое значение:

Значение определенного интеграла.

### Примечание:

Значимые цифры в ответе определяются Epsilon.

Будем использовать в примере использования дополнительную подынтегральную функцию:

Код 267. Дополнительная функция

```
double Func3(double x)  
{  
    return x*x;  
}
```

Код 268. Пример использования

```
double a=-2;  
double b=2;  
double Epsilon=0.01;  
double S;  
  
//Вызов функции  
S=MHL_IntegralOfSimpson(a,b,Epsilon,Func3);  
  
//Используем полученный результат  
MHL_ShowNumber (a,"Левая граница интегрирования", "a");  
// Левая граница интегрирования:  
//a=-2  
MHL_ShowNumber (b,"Правая граница интегрирования", "b");  
// Правая граница интегрирования:  
//b=2  
MHL_ShowNumber (S,"Интеграл", "S");  
// Интеграл:  
//S=5.33333
```

### 7.7.3 MHL\_IntegralOfTrapezium

Интегрирование по формуле трапеции с оценкой точности по правилу Рунге. Считается интеграл функции на отрезке  $[a,b]$  с погрешностью порядка Epsilon.

Код 269. Синтаксис

```
double MHL_IntegralOfTrapezium(double a, double b, double Epsilon, double (*Function)(  
    double));
```

**Входные параметры:**

a — начало отрезка интегрирования;

b — конец отрезка интегрирования;

Epsilon — погрешность;

Function — подынтегральная функция.

**Возвращаемое значение:**

Значение определенного интеграла.

**Примечание:**

Значимые цифры в ответе определяются Epsilon.

Будем использовать в примере использования дополнительную подынтегральную функцию:

Код 270. Дополнительная функция

```
double Func3(double x)  
{  
    return x*x;  
}
```

Код 271. Пример использования

```
double a=-2;  
double b=2;  
double Epsilon=0.01;  
double S;  
  
//Вызов функции  
S=MHL_IntegralOfTrapezium(a,b,Epsilon,Func3);  
  
//Используем полученный результат  
MHL_ShowNumber (a,"Левая граница интегрирования", "a");  
//Левая граница интегрирования:  
//a=-2  
MHL_ShowNumber (b,"Правая граница интегрирования", "b");  
//Правая граница интегрирования:  
//b=2  
MHL_ShowNumber (S,"Интеграл", "S");  
//Интеграл:  
//S=5.33594
```

## 7.8 Кодирование и декодирование

### 7.8.1 MHL\_BinaryGrayVectorToRealVector

Функция декодирует бинарную строку в действительный вектор, который и был закодирован методом «Стандартный рефлексивный Грей-код» (без использования временного массива). Перегруженная функция делает тоже самое, но с использованием временного массива (это позволяет не создавать каждый раз временный массив, что ускоряет работу).

Код 272. Синтаксис

```
void MHL_BinaryGrayVectorToRealVector(int *x, int n, double *VMHL_ResultVector,
    double *Left, double *Right, int *Lengthi, int VMHL_N);
void MHL_BinaryGrayVectorToRealVector(int *x, double *VMHL_ResultVector, int *
    TempBinaryVector, double *Left, double *Right, int *Lengthi, int VMHL_N);
```

#### Входные параметры:

*a* — бинарная строка представляющая собой Грей-код нескольких закодированных вещественных координат;

*n* — длина бинарной строки;

*VMHL\_ResultVector* — вещественный вектор, в который мы и записываем результат, размера *n*;

*Left* — массив левых границ изменения каждой вещественной координаты (размер *VMHL\_N*);

*Right* — массив правых границ изменения каждой вещественной координаты (размер *VMHL\_N*);

*Lengthi* — массив значений, сколько на каждую координату отводится бит в бинарной строке (размер массива *Lengthi* *VMHL\_N*);

*VMHL\_N* — длина вещественного вектора.

#### Возвращаемое значение:

Отсутствует.

Для перегруженной функции

#### Входные параметры:

*a* — бинарная строка представляющая собой Грей-код нескольких закодированных вещественных координат;

*VMHL\_ResultVector* — вещественный вектор, в который мы и записываем результат;

*TempBinaryVector* — указатель на временный массив размера *n*;

*Left* — массив левых границ изменения каждой вещественной координаты размера *VMHL\_N*;

*Right* — массив правых границ изменения каждой вещественной координаты размера *VMHL\_N*;

Lengthi — массив значений, сколько на каждую координату отводится бит в бинарной строке. Размер массива VMHL\_N;

VMHL\_N — длина вещественного вектора.

**Возвращаемое значение:**

Отсутствует.

**Примечание:** К криптографии данная функция не имеет отношения.

Код 273. Пример использования

```
int n=10; //Размер массива
int *BinaryGrayVector;
BinaryGrayVector=new int[n];
//Заполним случайно
TMHL_RandomBinaryVector(BinaryGrayVector,n);

int VMHL_N=2; //Пусть был закодирован двумерный вектор
double *RealVector; //Вещественный вектор
RealVector=new double[VMHL_N];
double *Left; //массив левых границ изменения каждой вещественной координаты
Left=new double[VMHL_N];
double *Right; //массив правых границ изменения каждой вещественной координаты
Right=new double[VMHL_N];
int *Lengthi; //массив значений, сколько на каждую координату отводится бит в бинарной строке;
Lengthi=new int[VMHL_N];

//Заполним массивы
//Причем по каждой координате одинаковые значения выставим
TMHL_FillVector(Left,VMHL_N,0.); //Пусть будет интервал [0;1]
TMHL_FillVector(Right,VMHL_N,1.);
TMHL_FillVector(Lengthi,VMHL_N,5); //По сумме элементов вектор должен равен n (длине бинарной строки)

//Вызов функции
MHL_BinaryGrayVectorToRealVector(BinaryGrayVector,n,RealVector,Left,Right,Lengthi,
VMHL_N);

//Используем полученный результат

MHL_ShowVectorT (BinaryGrayVector,n,"Бинарная строка Грея кода", "BinaryVector");
//Бинарная строка Грея кода:
//BinaryVector =
//1 0 1 0 1 0 0 0 1 0

MHL_ShowVectorT (RealVector,VMHL_N,"Был закодирован вектор", "RealVector");
//Был закодирован вектор:
//RealVector =
//0.78125 0.09375

delete [] BinaryGrayVector;
delete [] RealVector;
delete [] Left;
delete [] Right;
delete [] Lengthi;

//Для перегруженной функции
n=10; //Размер массива
```

```

BinaryGrayVector=new int[n];
//Заполним случайно
TMHL_RandomBinaryVector(BinaryGrayVector,n);

VMHL_N=2; //Пусть был закодирован двумерный вектор
RealVector=new double[VMHL_N];
Left=new double[VMHL_N];
Right=new double[VMHL_N];
Lengthi=new int[VMHL_N];

int *TempBinaryVector;
TempBinaryVector=new int[n];

//Заполним массивы
//Причем по каждой координате одинаковые значения выставим
TMHL_FillVector(Left,VMHL_N,0.); //Пусть будет интервал [0;1]
TMHL_FillVector(Right,VMHL_N,1.);
TMHL_FillVector(Lengthi,VMHL_N,5); //По сумме элементов вектор должен равен n (длине б
иарной строки)

//Вызов функции
MHL_BinaryGrayVectorToRealVector(BinaryGrayVector,RealVector,TempBinaryVector,Left,
    Right,Lengthi,VMHL_N);

//Используем полученный результатом

MHL_ShowVectorT (BinaryGrayVector,n,"Бинарная строка Грея кода", "BinaryVector");
// Бинарная строка Грея кода:
//BinaryVector =
//0 0 1 0 0 1 1 1 0 1

MHL_ShowVectorT (RealVector,VMHL_N,"Был закодирован вектор", "RealVector");
//Был закодирован вектор:
//RealVector =
//0.21875 0.6875

delete [] BinaryGrayVector;
delete [] RealVector;
delete [] Left;
delete [] Right;
delete [] Lengthi;
delete [] TempBinaryVector;

```

## 7.8.2 MHL\_BinaryVectorToRealVector

Функция декодирует бинарную строку в действительный вектор, который и был закодирован методом «Стандартное представление целого числа — номер узла в сетке дискретизации».

Код 274. Синтаксис

```
void MHL_BinaryVectorToRealVector(int *x, double *VMHL_ResultVector, double *Left,
    double *Right, int *Lengthi, int VMHL_N);
```

**Входные параметры:**

а — бинарная строка;

VMHL\_ResultVector — вещественный вектор, в который мы и записываем результат;

Left — массив левых границ изменения каждой вещественной координаты;

Right — массив правых границ изменения каждой вещественной координаты;

Lengthi — массив значений, сколько на каждую координату отводится бит в бинарной строке;

VMHL\_N — длина вещественного вектора.

### **Возвращаемое значение:**

Число в десятичной системе исчисления.

### **Примечание:**

К криптографии данная функция не имеет отношения.

### **Примечание:**

Вектор входных параметров действительно избыточен, но каждый раз пересчитывать затратно, так как функция вызывается в ГА часто.

Код 275. Пример использования

```
int n=10; //Размер массива
int *BinaryVector;
BinaryVector=new int[n];
//Заполним случайно
TMHL_RandomBinaryVector(BinaryVector,n);

int VMHL_N=2;//Пусть был закодирован двумерный вектор
double *RealVector;//Вещественный вектор
RealVector=new double[VMHL_N];
double *Left;//массив левых границ изменения каждой вещественной координаты
Left=new double[VMHL_N];
double *Right;//массив правых границ изменения каждой вещественной координаты
Right=new double[VMHL_N];
int *Lengthi;//массив значений, сколько на каждую координату отводится бит в бинарной строке;
Lengthi=new int[VMHL_N];

//Заполним массивы
//Причем по каждой координате одинаковые значения выставим
TMHL_FillVector(Left,VMHL_N,0.);//Пусть будет интервал [0;1]
TMHL_FillVector(Right,VMHL_N,1.);
TMHL_FillVector(Lengthi,VMHL_N,5);//По сумме элементов вектор должен равен n (длине бинарной строки)

//Вызов функции
MHL_BinaryVectorToRealVector(BinaryVector,RealVector,Left,Right,Lengthi,VMHL_N);

//Используем полученный результат
MHL_ShowVectorT (BinaryVector,n,"Бинарная строка", "BinaryVector");
//Бинарная строка:
//BinaryVector =
//0 1 0 1 1 1 0 1 0 1

MHL_ShowVectorT (RealVector,VMHL_N,"Был закодирован вектор", "RealVector");
//Был закодирован вектор:
//RealVector =
//0.34375 0.65625
```

```
delete [] BinaryVector;
delete [] RealVector;
delete [] Left;
delete [] Right;
delete [] Lengthi;
```

### 7.8.3 TMHL\_BinaryToDecimal

Функция декодирует двоичное число в десятичное целое неотрицательное.

Код 276. Синтаксис

```
template <class T> T TMHL_BinaryToDecimal(T *a, int VMHL_N);
```

**Входные параметры:**

a — двоичное число;

VMHL\_N — длина двоичного числа.

**Возвращаемое значение:**

Число в десятичной системе исчисления.

Код 277. Пример использования

```
int VMHL_N=8; //Размер массива
int *a;
a=new int[VMHL_N];
TMHL_RandomBinaryVector(a,VMHL_N);

//Вызов функции
int x=TMHL_BinaryToDecimal(a,VMHL_N);

//Используем полученный результат
MHL_ShowVectorT (a,VMHL_N,"Двоичное число", "a");
//Двоичное число:
//a =
//0   1   1   1   0   1   0

MHL_ShowNumber (x,"Было закодировано", "x");
//Было закодировано:
//x=122

delete [] a;
```

### 7.8.4 TMHL\_BinaryToDecimalFromPart

Функция декодирует двоичное число в десятичное целое неотрицательное. При этом двоичное число длины берется как часть некой бинарной строки a.

Код 278. Синтаксис

```
template <class T> T TMHL_BinaryToDecimalFromPart(T *a, int Begin, int n);
```

**Входные параметры:**

*a* — бинарная строка;

*Begin* — номер элемента массива *a* как начало двоичного числа (начиная с нуля);

*n* — длина двоичного числа (это не длина вектора *a*).

### **Возвращаемое значение:**

Число в десятичной системе исчисления. Если не удается вычислить, то возвращается -1.

### **О функции:**

Для декодирования из бинарной строки берется только часть:

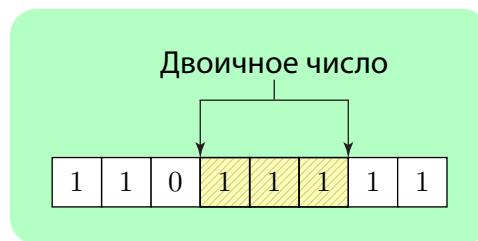


Рисунок 15. Часть бинарной строки

### **Примечание:**

Для перевода всей бинарной строки в число лучше воспользоваться функцией TMHL\_BinaryToDecimal.

### **Примечание:**

Данная функция используется, например, если в бинарной строке закодировано несколько десятичных чисел, каждое из которых закодировано своей подстрокой, а общая строка получена склейкой этих бинарных строк.

Код 279. Пример использования

```
int VMHL_N=8; //Размер массива
int *a;
a=new int[VMHL_N];
TMHL_RandomBinaryVector(a,VMHL_N);

int Begin=2;

//Вызов функции
int x=TMHL_BinaryToDecimalFromPart(a,Begin,5);

//Используем полученный результат
MHL_ShowVectorT (a,VMHL_N,"Бинарная строка", "a");
//Бинарная строка:
//a =
//0 0 1 0 1 0 0 0

MHL_ShowNumber (Begin,"Двоичное число состоит из 5 символов начиная с", "номера");
//Двоичное число состоит из 5 символов начиная с:
//номера=2

MHL_ShowNumber (x,"Было закодировано", "x");
//Было закодировано:
//x=20
```

```
delete [] a;
```

### 7.8.5 TMHL\_GrayCodeToBinary

Функция переводит бинарный код Грея в бинарный код.

Код 280. Синтаксис

```
template <class T> void TMHL_GrayCodeToBinary(T *a, int *VMHL_ResultVector, int VMHL_N  
);
```

**Входные параметры:**

a — код Грея (массив заполнен 0 и 1);

VMHL\_N — длина массива a.

**Возвращаемое значение:**

Отсутствует.

**О функции:**

Бинарная строка не представляет собой двоичный код целого числа, а представляет код Грея. Его отличительной особенностью является то, что если два целых числа отличаются на единицу, то их коды Грея также будут отличаться только одним битом. Двоичный код не обладает данным свойством. Существует метод по переводу кода Грея в двоичный код: старший разряд (крайний левый бит) записывается без изменения, каждый следующий символ кода Грея нужно инвертировать, если в двоичном коде перед этим была получена «1», и оставить без изменения, если в двоичном коде был получен «0».

Код 281. Пример использования

```
int VMHL_N=5;//Размер массива  
int *GrayCode;  
GrayCode=new int[VMHL_N];  
//Получим случайный Грей код  
TMHL_RandomBinaryVector(GrayCode,VMHL_N);  
  
int *BinaryCode;  
BinaryCode=new int[VMHL_N];  
  
//Вызов функции  
TMHL_GrayCodeToBinary(GrayCode,BinaryCode,VMHL_N);  
  
//Используем полученный результат  
MHL_ShowVectorT(GrayCode,VMHL_N,"Грей код", "a");  
//Грей код:  
//a =  
//1 1 0 1 1  
  
MHL_ShowVectorT(BinaryCode,VMHL_N,"Бинарный код, полученный из кода Грея", "a");  
//Бинарный код, полученный из кода Грея:  
//a =  
//1 0 0 1 0  
  
delete [] GrayCode;
```

```
delete [] BinaryCode;
```

### 7.8.6 TMHL\_GrayCodeToBinaryFromPart

Функция переводит бинарный код Грея в бинарный код. При этом бинарный код Грея берется как часть некой строки а, заполненной 0 и 1.

Код 282. Синтаксис

```
template <class T> void TMHL_GrayCodeToBinaryFromPart(T *a, T *VMHL_ResultVector, int Begin, int n);
```

#### Входные параметры:

а — строка, заполненная 0 и 1;

VMHL\_ResultVector — сюда записывается вектор бинарного числа. Причем запись происходит в те же элементы по номерам, что брались из вектора а, то есть в номера элементов от Begin до Begin+n. Остальные элементы в VMHL\_ResultVector не трогаются.

Begin — номер элемента массива а как начало числа в виде кода Грея (начиная с нуля);

n — длина числа в виде кода Грея (это не длина вектора а).

#### Возвращаемое значение:

Отсутствует.

#### О функции:

Для декодирования из строки кода Грея берется только часть:

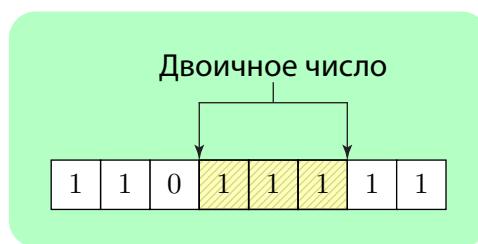


Рисунок 16. Часть бинарной строки

Бинарная подстрока не представляет собой двоичный код целого числа, а представляет код Грея. Его отличительной особенностью является то, что если два целых числа отличаются на единицу, то их коды Грея также будут отличаться только одним битом. Двоичный код не обладает данным свойством.

Существует метод по переводу кода Грея в двоичный код: старший разряд (крайний левый бит) записывается без изменения, каждый следующий символ кода Грея нужно инвертировать, если в двоичном коде перед этим была получена «1», и оставить без изменения, если в двоичном коде был получен «0».

#### Примечание:

Для перевода всей строки кода Грея в бинарную лучше воспользоваться функцией TMHL\_GrayCodeToBinary.

### **Примечание:**

Для перевода полученной бинарной строки в десятичное число воспользуйтесь функцией TMHL\_BinaryToDecimal или TMHL\_BinaryToDecimalFromPart.

### **Примечание:**

Данная функция используется, например, если в строке Грей кода закодировано несколько десятичных чисел, каждое из которых закодировано своей подстрокой, а общая строка получена склейкой этих строк Грей кода. Для того, чтобы получить десятичные числа проще вначале перевести Грей код в бинарный, а потом бинарный код перевести в десятичный.

Код 283. Пример использования

```
int VMHL_N=8; //Размер массива
int *VectorWithGrayCode;
VectorWithGrayCode=new int[VMHL_N];
//Заполним случайно нулями и единицами
TMHL_RandomBinaryVector(VectorWithGrayCode, VMHL_N);

int *VectorWithBinaryCode;
VectorWithBinaryCode=new int[VMHL_N];
TMHL_FillVector(VectorWithBinaryCode, VMHL_N, -1);

int Begin=2;

//Вызов функции
TMHL_GrayCodeToBinaryFromPart(VectorWithGrayCode, VectorWithBinaryCode, Begin, 5);

//Используем полученный результат
MHL_ShowVectorT (VectorWithGrayCode, VMHL_N, "Строка, содержащая код Грея", "a");
//Строка, содержащая код Грея:
//a =
//1 0 0 1 1 1 0 0

MHL_ShowNumber (Begin, "Двоичное число состоит из 5 символов начиная с", "номера");
//Двоичное число состоит из 5 символов начиная с:
//номера=2

MHL_ShowVectorT (VectorWithBinaryCode, VMHL_N, "Строка, содержащая бинарный код, полученный из кода Грея", "a");
//Строка, содержащая бинарный код, полученный из кода Грея:
//a =
//-1 -1 0 1 0 1 1 -1

delete [] VectorWithGrayCode;
delete [] VectorWithBinaryCode;
```

## 7.9 Комбинаторика

### 7.9.1 TMHL\_KCombinations

Функция возвращает число сочетаний из n по m (без возвращения).

Код 284. Синтаксис

```
template <class T> T TMHL_KCombinations(T k, T n);
```

**Входные параметры:**

k — по сколько элементов надо брать в группу;

n — общее число элементов.

**Возвращаемое значение:**

Число сочетаний из n по k.

**Формула:**

В программном коде число сочетаний находится через рекурсивную формулу. А в математике находится через формулу:

$$C_n^k = \frac{n!}{k! (n - k)!}.$$

Код 285. Пример использования

```
int n=10;
int k=MHL_RandomUniformInt(0,10);

int C=TMHL_KCombinations(k,n);

//Используем полученный результат
MHL_ShowNumber(C, "Число сочетаний по "+MHL_NumberToText(k)+" элементов из " +
    MHL_NumberToText(n), "C");
//Число сочетаний по 8 элементов из 10:
//C=45
```

## 7.10 Математические функции

### 7.10.1 MHL\_ArithmeticalProgression

Арифметическая прогрессия. n-ый член последовательности.

Код 286. Синтаксис

```
double MHL_ArithmeticalProgression(double a1,double d,int n);
```

**Входные параметры:**

a1 — начальный член прогрессии;

d — шаг арифметической прогрессии;

n — номер последнего члена.

**Возвращаемое значение:**

n-ый член последовательности.

Код 287. Пример использования

```
double a1=MHL_RandomUniformInt(1,10);
double d=MHL_RandomUniformInt(1,10);
```

```

int n=MHL_RandomUniformInt(1,10);

double an=MHL_ArithmeticalProgression(a1,d,n);

//Используем полученный результат
MHL_ShowNumber(a1,"Первый член последовательности", "a1");
//Первый член последовательности:
//a1=6
MHL_ShowNumber(d,"Шаг арифметической прогрессии", "d");
//Шаг арифметической прогрессии:
//d=9
MHL_ShowNumber(n,"Номер последнего члена", "n");
//Номер последнего члена:
//n=4
MHL_ShowNumber(an,"n-ый член последовательности", "an");
//n-ый член последовательности:
//an=33

```

## 7.10.2 MHL\_ExpMSxD2

Функция вычисляет выражение  $\exp(-x * x/2)$ .

Код 288. Синтаксис

```
double MHL_ExpMSxD2(double x);
```

**Входные параметры:**

$x$  — входная переменная.

**Возвращаемое значение:**

Значение функции в точке.

**Формула:**

$$F(x) = e^{-\frac{x^2}{2}}.$$

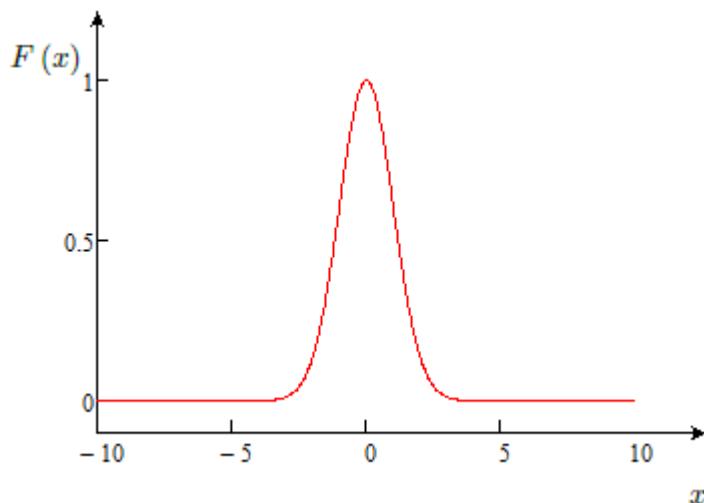


Рисунок 17. График функции

Код 289. Пример использования

```
double t;
double f;
t=MHL_RandomUniform(0,3);

//Вызов функции
f=MHL_ExpMSxD2(t);

//Используем полученный результат

MHL_ShowNumber (t,"Параметр", "t");
//Параметр:
//t=2.06177
MHL_ShowNumber (f,"Значение функции", "f");
//Значение функции:
//f=0.11938
```

### 7.10.3 MHL\_GeometricSeries

Геометрическая прогрессия. n-ый член последовательности.

Код 290. Синтаксис

```
double MHL_GeometricSeries(double u1,double q,int n);
```

**Входные параметры:**

u1 — начальный член прогрессии;

q — шаг прогрессии;

n — номер последнего члена.

**Возвращаемое значение:**

n-ый член последовательности.

Код 291. Пример использования

```
double u1=MHL_RandomUniformInt(1,10);
double q=MHL_RandomUniformInt(1,10);
int n=MHL_RandomUniformInt(1,10);

double qn=MHL_GeometricSeries(u1,q,n);

//Используем полученный результат
MHL_ShowNumber(u1,"Первый член последовательности", "u1");
//Первый член последовательности:
//u1=4
MHL_ShowNumber(q, "Шаг арифметической прогрессии", "q");
//Шаг арифметической прогрессии:
//q=4
MHL_ShowNumber(n, "Номер последнего члена", "n");
//Номер последнего члена:
//n=6
MHL_ShowNumber(qn, "n-ый член последовательности", "qn");
//n-ый член последовательности:
//qn=4096
```

#### 7.10.4 MHL\_GreatestCommonDivisorEuclid

Функция находит наибольший общий делитель двух чисел по алгоритму Евклида.

Код 292. Синтаксис

```
int MHL_GreatestCommonDivisorEuclid(int A,int B);
```

**Входные параметры:**

А — первое число;

В — второе число.

**Возвращаемое значение:**

НОД(А,В)

Код 293. Пример использования

```
int A=MHL_RandomUniformInt(1,100);
int B=MHL_RandomUniformInt(1,100);

double Result=MHL_GreatestCommonDivisorEuclid(A,B);

//Используем полученный результат
MHL_ShowNumber(A, "Число", "A");
//Число:
//A=96
MHL_ShowNumber(B, "Число", "B");
//Число:
//B=18
MHL_ShowNumber(Result, "НОД", "");
//НОД:
//=6
```

#### 7.10.5 MHL\_HowManyPowersOfTwo

Функция вычисляет, какой минимальной степенью двойки можно покрыть целое положительное число.

Код 294. Синтаксис

```
int MHL_HowManyPowersOfTwo(int x);
```

**Входные параметры:**

х — целое число.

**Возвращаемое значение:**

Минимальная степень двойки можно покрыть целое положительное число:  $2^{V_{MHL\_Result}} > x$

Код 295. Пример использования

```
int x=MHL_RandomUniformInt(0,1000);
int Degree;

//Вызываем функцию
```

```

Degree=MHL_HowManyPowersOfTwo(x);

//Используем полученный результат
MHL_ShowNumber(x, "Число", "x");
//Число:
//x=480
MHL_ShowNumber(Degree, "Его покрывает 2 в степени", "Degree");
//Его покрывает 2 в степени:
//Degree=9
MHL_ShowNumber(TMHL_PowerOf(2,Degree), "То есть", "2^"+MHL_NumberToText(Degree));
//То есть:
//2^9=512

```

### 7.10.6 MHL\_InverseNormalizationNumberAll

Функция осуществляет обратную нормировку числа из интервала  $[0; 1]$  в интервал  $[-\infty; \infty]$ , которое было осуществлено функцией MHL\_NormalizationNumberAll.

Код 296. Синтаксис

```
double MHL_InverseNormalizationNumberAll(double x);
```

**Входные параметры:**

x — число в интервале  $[0;1]$ .

**Возвращаемое значение:**

Перенормированное число.

Код 297. Пример использования

```

double x;
double y;
x=MHL_RandomNumber();

//Вызов функции
y=MHL_InverseNormalizationNumberAll(x);

//Используем полученный результат
MHL_ShowNumber (x, "Нормированное число", "x");
// Нормированное число:
//x=0.0491333
MHL_ShowNumber (y, "Перенормированное число", "y");
// Перенормированное число:
//y=-9.1764

```

### 7.10.7 MHL\_LeastCommonMultipleEuclid

Функция находит наименьшее общее кратное двух чисел по алгоритму Евклида.

Код 298. Синтаксис

```
int MHL_LeastCommonMultipleEuclid(int A,int B);
```

**Входные параметры:**

А — первое число;

В — второе число.

#### **Возвращаемое значение:**

НОК(А,В)

Код 299. Пример использования

```
int A=MHL_RandomUniformInt(1,100);
int B=MHL_RandomUniformInt(1,100);

double Result=MHL_LeastCommonMultipleEuclid(A,B);

//Используем полученный результат
MHL_ShowNumber(A, "Число", "A");
//Число:
//A=68
MHL_ShowNumber(B, "Число", "B");
//Число:
//B=67
MHL_ShowNumber(Result, "НОК", "");
//НОК:
//=4556
```

### **7.10.8 MHL\_MixedMultiLogicVectorOfFullSearch**

Функция генерирует определенный вектор k-значной логики, где каждый элемент может принимать разное максимальное значение, в полном переборе вариантов. Генерируется I вектор в этом полном переборе.

Код 300. Синтаксис

```
void MHL_MixedMultiLogicVectorOfFullSearch(int *VMHL_Vector, int I, int *
HowMuchInElements, int VMHL_N);
```

#### **Входные параметры:**

VMHL\_Vector — выходной вектор, в который записывается результат;

I — номер в массиве в полном переборе, начиная с нуля (от 0 и до произведения всех элементов массива HowMuchInElements - 1);

HowMuchInElements — сколько значений может принимать элемент в векторе. То есть элемент может быть 0 и HowMuchInElements[i]-1;

VMHL\_N — количество элементов в массиве.

#### **Возвращаемое значение:**

Отсутствует.

#### **Примечание:**

Где может быть использована эта функция? Допустим, у вас есть десять вложенных циклов, в которых меняется какой-то параметр. Плюс вложенность циклов может еще и варьи-

роваться. И с помощью данной функции все эти вложенные циклы заменяются на один, а значения счетчиков вычисляются с помощью этой функции.

Код 301. Пример использования

```
int i;
int N=3;
int *x;
x=new int[N];
int *y;
y=new int[N];

x[0]=3;
x[1]=2;
x[2]=3;

int P=MHL_ProductOfElementsOfVector(x,N);

MHL_ShowVector(x,N,"Сколько каждого параметра в штуках", "x");
//Сколько каждого параметра в штуках:
//x =
//3
//2
//3

for (i=0;i<P;i++)
{
    MHL_ShowNumber(i,<hr>Номер итерации, "i");
    //Номер итерации:
    //i=0

    //Вызов функции
    MHL_MixedMultiLogicVectorOfFullSearch(y,i,x,N);

    MHL_ShowVectorT(y,N,"Необходимый вектор", "y");
    //Необходимый вектор:
    //y =
    //0  0  0
}

delete [] x;
delete [] y;
```

### 7.10.9 MHL\_NormalizationNumberAll

Функция нормирует число из интервала  $[-\infty; \infty]$  в интервал  $[0; 1]$ . При этом в нуле возвращает 0.5, в  $-\infty$  возвращает 0, в  $\infty$  возвращает 1. Если  $x < y$ , то  $MHL_NormalizationNumberAll(x) < MHL_NormalizationNumberAll(y)$ . Под бесконечностью принимается машинная бесконечность.

Код 302. Синтаксис

```
double MHL_NormalizationNumberAll(double x);
```

**Входные параметры:**

Х — число.

**Возвращаемое значение:**

Нормированное число.

**Формула:**

$$f(x) = \frac{1}{2} \left( \frac{1}{1 + \frac{1}{|x|}} \cdot sign(x) + 1 \right).$$

Код 303. Пример использования

```
double x;
double y;
y=MHL_RandomUniform(-100,100);

//Вызов функции
x=MHL_NormalizationNumberAll(y);

//Используем полученный результат
MHL_ShowNumber (y, "Число", "y");
//Число:
//y=-10.4004
MHL_ShowNumber (x, "Нормированное число", "x");
//Нормированное число:
//x=0.0438581
```

### 7.10.10 MHL\_Parity

Функция проверяет четность целого числа.

Код 304. Синтаксис

```
int MHL_Parity(int a);
```

**Входные параметры:**

a — исходное число.

**Возвращаемое значение:**

1 — четное;

0 — нечетное.

Код 305. Пример использования

```
int a=MHL_RandomUniformInt(-50,50);

double Result=MHL_Parity(a);

//Используем полученный результат
MHL_ShowNumber(Result, "Четность числа "+MHL_NumberToText(a), "равна");
//Четность числа 2:
//равна=1
```

### 7.10.11 MHL\_ProbabilityDensityFunctionOfInverseGaussianDistribution

Функция вычисляет плотность вероятности распределения обратного гауссовскому распределению..

Код 306. Синтаксис

```
double MHL_ProbabilityDensityFunctionOfInverseGaussianDistribution (double x, double mu, double lambda);
```

**Входные параметры:**

x — входная переменная ( $x > 0$ );

mu — первый параметр распределения;

lambda — второй параметр распределения.

**Возвращаемое значение:**

Значение функции в точке.

**Формула:**

$$f(x, \mu, \lambda) = \left[ \frac{\lambda}{2\pi x^3} \right]^{1/2} \exp \frac{-\lambda(x - \mu)^2}{2\mu^2 x}.$$

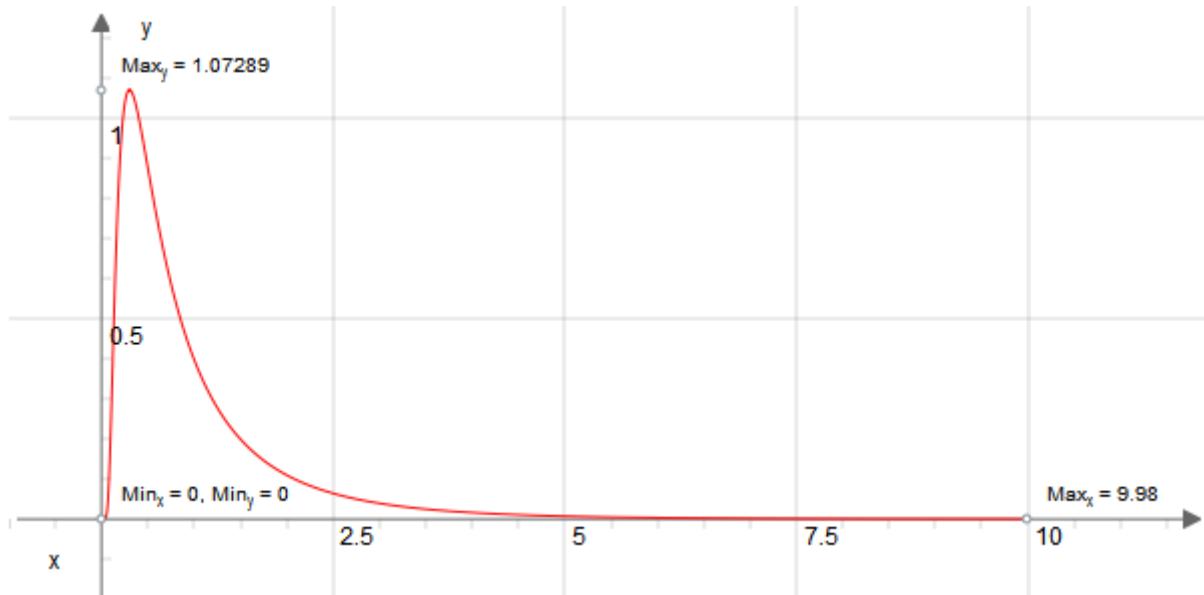


Рисунок 18. График функции

Код 307. Пример использования

```
double x;
double mu=1;
double lambda=1;
double f;
x=MHL_RandomUniform(0,3);

//Вызов функции
f=MHL_ProbabilityDensityFunctionOfInverseGaussianDistribution(x,mu,lambda);
```

```

//Используем полученный результат
MHL_ShowNumber (x, "Входная переменная", "x");
//Входная переменная:
//x=1.10063
MHL_ShowNumber (mu, "Параметр mu", "mu");
//Параметр mu:
//mu=1
MHL_ShowNumber (lambda, "Параметр lambda", "lambda");
//Параметр lambda:
//lambda=1
MHL_ShowNumber (f, "Значение функции", "f");
//Значение функции:
//f=0.343916

```

### 7.10.12 MHL\_SumGeometricSeries

Геометрическая прогрессия. Сумма первых n членов.

Код 308. Синтаксис

```
double MHL_SumGeometricSeries(double u1,double q,int n);
```

**Входные параметры:**

u1 — начальный член прогрессии;

q — шаг прогрессии;

n — номер последнего члена.

**Возвращаемое значение:**

Сумма первых n членов.

Код 309. Пример использования

```

double u1=MHL_RandomUniformInt(1,5);
double q=MHL_RandomUniformInt(1,5);
int n=MHL_RandomUniformInt(1,5);

double Sum=MHL_SumGeometricSeries(u1,q,n);

//Используем полученный результат
MHL_ShowNumber(u1, "Первый член последовательности", "u1");
//Первый член последовательности:
//u1=4
MHL_ShowNumber(q, "Шаг арифметической прогрессии", "q");
//Шаг арифметической прогрессии:
//q=4
MHL_ShowNumber(n, "Номер последнего члена", "n");
//Номер последнего члена:
//n=3
MHL_ShowNumber(Sum, "Сумма первых n членов", "Sum");
//Сумма первых n членов:
//Sum=84

```

### 7.10.13 MHL\_SumOfArithmeticalProgression

Арифметическая прогрессия. Сумма первых n членов.

Код 310. Синтаксис

```
double MHL_SumOfArithmeticalProgression(double a1, double d, int n);
```

**Входные параметры:**

a1 — начальный член прогрессии;  
d — шаг арифметической прогрессии;  
n — номер последнего члена.

**Возвращаемое значение:**

Сумма первых n членов.

Код 311. Пример использования

```
double a1=MHL_RandomUniformInt(1,10);
double d=MHL_RandomUniformInt(1,10);
int n=MHL_RandomUniformInt(1,10);

double Sum=MHL_SumOfArithmeticalProgression(a1,d,n);

//Используем полученный результат
MHL_ShowNumber(a1,"Первый член последовательности","a1");
//Первый член последовательности:
//a1=9
MHL_ShowNumber(d,"Шаг арифметической прогрессии","d");
//Шаг арифметической прогрессии:
//d=6
MHL_ShowNumber(n,"Номер последнего члена","n");
//Номер последнего члена:
//n=9
MHL_ShowNumber(Sum,"Сумма первых n членов","Sum");
//Сумма первых n членов:
//Sum=297
```

### 7.10.14 MHL\_SumOfDigits

Функция подсчитывает сумму цифр любого целого числа.

Код 312. Синтаксис

```
int MHL_SumOfDigits(int a);
```

**Входные параметры:**

a — целое число.

**Возвращаемое значение:**

Сумма цифр.

Код 313. Пример использования

```
int a=MHL_RandomUniformInt(100,30000);

//Вызов функции
int SumOfDigits=MHL_SumOfDigits(a);

//Используем полученный результат
MHL_ShowNumber (SumOfDigits, "Сумма цифр числа a="+MHL_NumberToText(a), "равна");
//Сумма цифр числа a=2069:
//равна=17
```

### 7.10.15 TMHL\_Abs

Функция возвращает модуль числа.

Код 314. Синтаксис

```
template <class T> T TMHL_Abs(T x);
```

**Входные параметры:**

x — число.

**Возвращаемое значение:**

Модуль числа.

Код 315. Пример использования

```
double x;
double abs;

x=MHL_RandomUniform(-10,10);

//Вызов функции
abs=TMHL_Abs(x);

//Используем полученный результат
MHL_ShowNumber (x, "Число", "x");
// Число:
//x=-6.29578
MHL_ShowNumber (abs, "Модуль", "abs");
// Модуль:
//abs=6.29578
```

### 7.10.16 TMHL\_Factorial

Функция вычисляет факториал числа.

Код 316. Синтаксис

```
template <class T> T TMHL_Factorial(T x);
```

**Входные параметры:**

x — число.

### **Возвращаемое значение:**

Факториал числа.

Код 317. Пример использования

```
int a=MHL_RandomUniformInt(0,10);

double Result=TMHL_Factorial(a);

//Используем полученный результат
MHL_ShowNumber(Result, "Факториал числа "+MHL_NumberToText(a), "a!");

//Факториал числа 3:
//a!=6
```

### **7.10.17 TMHL\_FibonacciNumber**

Функция вычисляет число Фибоначчи, заданного номера.

Код 318. Синтаксис

```
template <class T> T TMHL_FibonacciNumber(T n);
```

### **Входные параметры:**

n — номер числа Фибоначчи.

### **Возвращаемое значение:**

Число Фибоначчи, заданного номера.

Код 319. Пример использования

```
int n;
int F;
n=MHL_RandomUniformInt(3,20);

//Вызов функции
F=TMHL_FibonacciNumber(n);

//Используем полученный результат

MHL_ShowNumber (n, "Номер числа", "n");
// Номер числа:
// n=14
MHL_ShowNumber (F, "Число Фибоначчи, заданного номера", "F");
// Число Фибоначчи, заданного номера:
// F=377
```

### **7.10.18 TMHL\_HeavisideFunction**

Функция Хевисайда (функция одной переменной).

Код 320. Синтаксис

```
template <class T> T TMHL_HeavisideFunction(T x);
```

**Входные параметры:**

$x$  — переменная.

**Возвращаемое значение:**

Значение функции Хевисайда.

**Формула:**

$$F(x) = \begin{cases} 1, & \text{если } x > 0; \\ 0, & \text{если } x \leq 0. \end{cases}$$

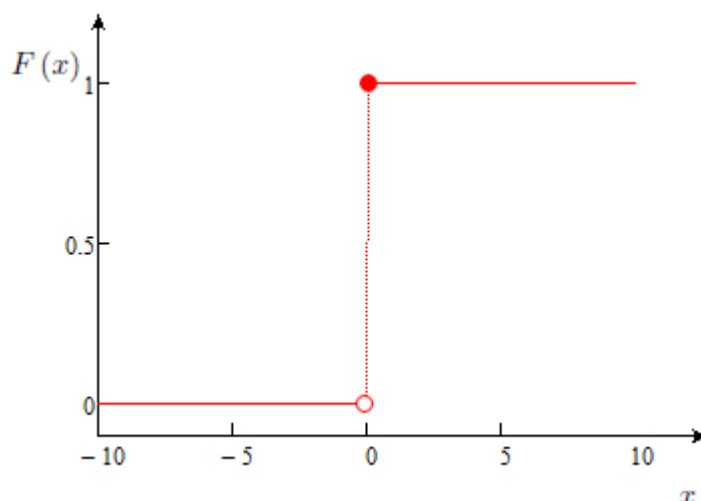


Рисунок 19. График функции

Код 321. Пример использования

```
double x=MHL_RandomUniform(-50,50);  
  
double F=TMHL_HeavisideFunction(x);  
  
//Используем полученный результат  
MHL_ShowNumber(F, "Функция Хевисайда при a = "+MHL_NumberToText(x), "равна");  
//Функция Хевисайда при a = -49.7559:  
//равна=0
```

### 7.10.19 TMHL\_Max

Функция возвращает максимальный элемент из двух.

Код 322. Синтаксис

```
template <class T> T TMHL_Max(T a, T b);
```

**Входные параметры:**

$a$  — первый элемент;

$b$  — второй элемент.

**Возвращаемое значение:**

Максимальный элемент.

Код 323. Пример использования

```
int a=MHL_RandomUniformInt(10,100);
int b=MHL_RandomUniformInt(10,100);

//Вызов функции
int Max=TMHL_Max(a,b);

//Используем полученный результат
MHL_ShowNumber(Max, "Максимальное среди "+MHL_NumberToText(a)+" и "+MHL_NumberToText(b)
    , "равно");
//Максимальное среди 73 и 44:
//равно=73
```

## 7.10.20 TMHL\_Min

Функция возвращает минимальный элемент из двух.

Код 324. Синтаксис

```
template <class T> T TMHL_Min(T a, T b);
```

**Входные параметры:**

a — первый элемент;

b — первый элемент.

**Возвращаемое значение:**

Минимальный элемент.

Код 325. Пример использования

```
int a=MHL_RandomUniformInt(10,100);
int b=MHL_RandomUniformInt(10,100);

//Вызов функции
int Max=TMHL_Min(a,b);

//Используем полученный результат
MHL_ShowNumber(Max, "Минимальное среди "+MHL_NumberToText(a)+" и "+MHL_NumberToText(b)
    , "равно");
//Минимальное среди 79 и 18:
//равно=18
```

## 7.10.21 TMHL\_NumberInterchange

Функция меняет местами значения двух чисел.

Код 326. Синтаксис

```
template <class T> void TMHL_NumberInterchange(T *a, T *b);
```

### **Входные параметры:**

a — первое число;

b — второе число.

### **Возвращаемое значение:**

Отсутствует.

Код 327. Пример использования

```
double a=MHL_RandomUniform(-10,10);
double b=MHL_RandomUniform(-10,10);

MHL_ShowNumber(a, "Было", "a");
//Было:
//a=-3.18237
MHL_ShowNumber(b, "Было", "b");
//Было:
//b=5.36194

//Вызов функции
TMHL_NumberInterchange(&a, &b);

//Используем полученный результат
MHL_ShowNumber(a, "Стало", "a");
//Стало:
//a=5.36194
MHL_ShowNumber(b, "Стало", "b");
//Стало:
//b=-3.18237
```

## **7.10.22 TMHL\_PowerOf**

Функция возводит произвольное число в целую степень.

Код 328. Синтаксис

```
template <class T> T TMHL_PowerOf(T x, int n);
```

### **Входные параметры:**

x — основание степени;

n — показатель степени.

### **Возвращаемое значение:**

Степень числа.

Код 329. Пример использования

```
double a=MHL_RandomUniform(-5,5);
int Degree=MHL_RandomUniformInt(0,20);

double Result=TMHL_PowerOf(a,Degree);

//Используем полученный результат
```

```
MHL_ShowNumber(Result, "Число "+MHL_NumberToText(a)+" в степени "+MHL_NumberToText(Degree), "равно");
//Число 3.9624 в степени 4:
//равно=246.51
```

### 7.10.23 TMHL\_Sign

Функция вычисляет знака числа.

Код 330. Синтаксис

```
template <class T> int TMHL_Sign(T a);
```

**Входные параметры:**

a — исходное число.

**Возвращаемое значение:**

0 — если a==0;

1 — если число положительное;

-1 — если число отрицательное.

Код 331. Пример использования

```
int a=MHL_RandomUniformInt(-5,5);

//Вызов функции
int Result=TMHL_Sign(a);

//Используем полученный результат
MHL_ShowNumber(Result, "Знак числа "+MHL_NumberToText(a), "равен");
//Знак числа -3:
//равен=-1
```

### 7.10.24 TMHL\_SignNull

Функция вычисляет знака числа. При нуле возвращает 1.

Код 332. Синтаксис

```
template <class T> int TMHL_SignNull(T a);
```

**Входные параметры:**

a — исходное число.

**Возвращаемое значение:**

1 — если число неотрицательное;

-1 — если число отрицательное.

Код 333. Пример использования

```

int a=MHL_RandomUniformInt(-5,5);

//Вызов функции
int Result=TMHL_SignNull(a);

//Используем полученный результат
MHL_ShowNumber(Result,"Знак числа "+MHL_NumberToText(a),"равен");
//Знак числа 0:
//равен=1

```

## 7.11 Матрицы

### 7.11.1 TMHL\_CheckForIdenticalColsInMatrix

Функция проверяет наличие одинаковых столбцов в матрице.

Код 334. Синтаксис

```
template <class T> bool TMHL_CheckForIdenticalColsInMatrix(T **VMHL_ResultMatrix, int VMHL_N, int VMHL_M);
```

**Входные параметры:**

VMHL\_ResultMatrix — указатель на массив;

VMHL\_N — размер массива VMHL\_ResultMatrix (число строк);

VMHL\_M — размер массива VMHL\_ResultMatrix (число столбцов);

**Возвращаемое значение:**

true — если есть одинаковые столбцы;

false — если таких столбцов нет.

Код 335. Пример использования

```

int i;
int VMHL_N=2;//Размер массива (число строк)
int VMHL_M=10;//Размер массива (число столбцов)
int **a;
a=new int*[VMHL_N];
for (i=0;i<VMHL_N;i++) a[i]=new int[VMHL_M];

TMHL_RandomIntMatrix(a,0,5,VMHL_N,VMHL_M);//заполним матрицу

//Вызов функции
bool b=TMHL_CheckForIdenticalColsInMatrix(a,VMHL_N,VMHL_M);

//Используем полученный результат
MHL_ShowMatrix (a,VMHL_N,VMHL_M,"Заполненная матрица", "a");
//Заполненная матрица:
//a =
//4   4   0   1   0   1   0   0   2   1
//1   4   4   3   3   4   4   3   0   1

MHL_ShowNumber(b,"Есть ли одинаковые столбцы", "b");
//Есть ли одинаковые столбцы::
```

```
//b=1

for (i=0;i<VMHL_N;i++) delete [] a[i];
delete [] a;
```

### 7.11.2 TMHL\_CheckForIdenticalRowsInMatrix

Функция проверяет наличие одинаковых строк в матрице.

Код 336. Синтаксис

```
template <class T> bool TMHL_CheckForIdenticalRowsInMatrix(T **VMHL_ResultMatrix, int
VMHL_N, int VMHL_M);
```

**Входные параметры:**

VMHL\_ResultMatrix — указатель на массив;

VMHL\_N — размер массива VMHL\_ResultMatrix (число строк);

VMHL\_M — размер массива VMHL\_ResultMatrix (число столбцов);

**Возвращаемое значение:**

true — если есть одинаковые строки;

false — если таких строк нет.

Код 337. Пример использования

```
int i;
int VMHL_N=10; //Размер массива (число строк)
int VMHL_M=2; //Размер массива (число столбцов)
int **a;
a=new int*[VMHL_N];
for (i=0;i<VMHL_N;i++) a[i]=new int[VMHL_M];

TMHL_RandomIntMatrix(a,0,5,VMHL_N,VMHL_M); //заполним матрицу

//Вызов функции
bool b=TMHL_CheckForIdenticalRowsInMatrix(a,VMHL_N,VMHL_M);

//Используем полученный результат
MHL_ShowMatrix (a,VMHL_N,VMHL_M,"Заполненная матрица", "a");
//Заполненная матрица:
//a =
//3 3
//0 0
//3 0
//0 3
//3 1
//3 2
//3 2
//2 1
//0 3
//4 2

MHL_ShowNumber(b,"Есть ли одинаковые строки", "b");
//Есть ли одинаковые строки:
```

```
//b=1

for (i=0;i<VMHL_N;i++) delete [] a[i];
delete [] a;
```

### 7.11.3 TMHL\_ColInterchange

Функция переставляет столбцы матрицы.

Код 338. Синтаксис

```
template <class T> void TMHL_ColInterchange(T **VMHL_ResultMatrix, int VMHL_N, int k,
                                             int l);
```

**Входные параметры:**

VMHL\_ResultMatrix — указатель на исходную матрицу (в ней и сохраняется результат);

VMHL\_N — размер массива (число строки);

k,l — номера переставляемых столбцов (нумерация с нуля).

**Возвращаемое значение:**

Отсутствует.

Код 339. Пример использования

```
int i,j;
int VMHL_N=5; //Размер массива (число строк)
int VMHL_M=5; //Размер массива (число столбцов)
int **Matrix;
Matrix=new int*[VMHL_N];
for (i=0;i<VMHL_N;i++) Matrix[i]=new int[VMHL_M];
//Заполним случайными числами
for (i=0;i<VMHL_N;i++)
    for (j=0;j<VMHL_M;j++)
        Matrix[i][j]=MHL_RandomUniformInt(10,100);

MHL_ShowMatrix (Matrix,VMHL_N,VMHL_M,"Случайная матрица", "Matrix");
// Случайная матрица:
//Matrix =
//46 37 90 95 83
//92 58 48 61 16
//31 92 37 64 56
//20 54 84 90 75
//86 79 20 40 69

// номера переставляемых столбцов
int k=MHL_RandomUniformInt(0,5);
int l=MHL_RandomUniformInt(0,5);

//Вызов функции
TMHL_ColInterchange(Matrix,VMHL_N,k,l);

//Используем полученный результат
MHL_ShowNumber (k,"Номер первого столбца","k");
// Номер первого столбца:
//k=4
```

```

MHL_ShowNumber (1,"Номер второго столбца","1");
// Номер второго столбца:
//l=0
MHL>ShowMatrix (Matrix,VMHL_N,VMHL_M,"Матрица с персетавленными столбцами", "Matrix")
;
// Матрица с персетавленными столбцами:
//Matrix =
//83 37 90 95 46
//16 58 48 61 92
//56 92 37 64 31
//75 54 84 90 20
//69 79 20 40 86

for (i=0;i<VMHL_N;i++)
    delete [] Matrix[i];
delete [] Matrix;

```

#### 7.11.4 TMHL\_ColToMatrix

Функция копирует в матрицу (двумерный массив) из вектора столбец.

Код 340. Синтаксис

```
template <class T> void TMHL_ColToMatrix(T **VMHL_ResultMatrix, T *b, int VMHL_N, int k);
```

**Входные параметры:**

VMHL\_ResultMatrix — указатель на матрицу;

b — указатель на вектор;

VMHL\_N — количество строк в матрице и одновременно размер массива b;

k — номер столбца, в который будет происходить копирование (начиная с 0).

**Возвращаемое значение:**

Отсутствует.

Код 341. Пример использования

```

int i,j;
int VMHL_N=10; //Размер массива (число строк)
int VMHL_M=3; //Размер массива (число столбцов)
int **a;
a=new int*[VMHL_N];
for (i=0;i<VMHL_N;i++) a[i]=new int[VMHL_M];
int *b;
b=new int[VMHL_N];
//Заполним случайными числами
for (i=0;i<VMHL_N;i++)
    for (j=0;j<VMHL_M;j++)
        a[i][j]=MHL_RandomUniformInt(10,100);
MHL>ShowMatrix (a,VMHL_N,VMHL_M,"Случайная матрица", "a");
//Случайная матрица:
//a =
//13 99 23
//69 44 44
//64 70 72

```

```

//14  85 92
//11  40 12
//95  85 81
//82  50 13
//63  82 58
//56  68 89
//51  89 78

for (j=0;j<VMHL_N;j++)
    b[j]=MHL_RandomUniformInt(10,100);

int k=1;//Номер столбца, в который мы копируем

//Вызов функции
TMHL_ColToMatrix(a,b,VMHL_N,k);

//Используем полученный результат
MHL_ShowNumber(k,"Номер столбца, в который мы копируем ", "k");
//Номер столбца, в который мы копируем :
//k=1
MHL_ShowVector (b,VMHL_N,"Вектор", "b");
//Вектор:
//b =
//35
//92
//90
//41
//17
//24
//11
//13
//23
//14

MHL_ShowMatrix (a,VMHL_N,VMHL_M,"Матрица с изменившимся столбцом", "a");
//Матрица с изменившимся столбцом:
//a =
//13  35 23
//69  92 44
//64  90 72
//14  41 92
//11  17 12
//95  24 81
//82  11 13
//63  13 58
//56  23 89
//51  14 78

for (i=0;i<VMHL_N;i++)
    delete [] a[i];
delete [] a;
delete [] b;

```

### 7.11.5 TMHL\_DeleteColInMatrix

Функция удаляет k столбец из матрицы (начиная с нуля). Все правостоящие столбцы сдвигаются влево на единицу. Последний столбец зануляется.

Код 342. Синтаксис

```
template <class T> void TMHL_DeleteColInMatrix(T **VMHL_ResultMatrix, int k, int VMHL_N, int VMHL_M);
```

### **Входные параметры:**

VMHL\_ResultMatrix — указатель на преобразуемый массив;

k — номер удаляемого столбца;

VMHL\_N — размер массива VMHL\_ResultMatrix (число строк);

VMHL\_M — размер массива VMHL\_ResultMatrix (число столбцов).

### **Возвращаемое значение:**

Отсутствует.

Код 343. Пример использования

```
int i,j;
int VMHL_N=6;//Размер массива (число строк)
int VMHL_M=4;//Размер массива (число столбцов)
double **Matrix;
Matrix=new double*[VMHL_N];
for (i=0;i<VMHL_N;i++) Matrix[i]=new double[VMHL_M];
//Заполним случайными числами
for (i=0;i<VMHL_N;i++)
    for (j=0;j<VMHL_M;j++)
        Matrix[i][j]=MHL_RandomUniformInt(10,100);

MHL_ShowMatrix (Matrix,VMHL_N,VMHL_M, "Случайная матрица", "Matrix");
// Случайная матрица:
//Matrix =
//39  52  14  31
//49   49  59  65
//68   15  12  86
//91   73  36  32
//52   31  24  78
//22   20  33  94

int k=2;//удалим второй столбец

//Вызов функции
TMHL_DeleteColInMatrix(Matrix,k,VMHL_N,VMHL_M);

//Используем полученный результат

MHL_ShowMatrix (Matrix,VMHL_N,VMHL_M, "Матрица с удаленным столбцом", "Matrix");
// Матрица с удаленным столбцом:
//Matrix =
//39  52  31  0
//49   49  65  0
//68   15  86  0
//91   73  32  0
//52   31  78  0
//22   20  94  0

for (i=0;i<VMHL_N;i++) delete [] Matrix[i];
delete [] Matrix;
```

### 7.11.6 TMHL\_DeleteRowInMatrix

Функция удаляет k строку из матрицы (начиная с нуля). Все нижестоящие строки поднимаются на единицу. Последняя строка зануляется.

Код 344. Синтаксис

```
template <class T> void TMHL_DeleteRowInMatrix(T **VMHL_ResultMatrix, int k, int  
VMHL_N, int VMHL_M);
```

#### Входные параметры:

VMHL\_ResultMatrix — указатель на преобразуемый массив;

k — номер удаляемой строки;

VMHL\_N — размер массива VMHL\_ResultMatrix (число строк);

VMHL\_M — размер массива VMHL\_ResultMatrix (число столбцов).

#### Возвращаемое значение:

Отсутствует.

Код 345. Пример использования

```
int i,j;  
int VMHL_N=6;//Размер массива (число строк)  
int VMHL_M=4;//Размер массива (число столбцов)  
double **Matrix;  
Matrix=new double*[VMHL_N];  
for (i=0;i<VMHL_N;i++) Matrix[i]=new double[VMHL_M];  
//Заполним случайными числами  
for (i=0;i<VMHL_N;i++)  
    for (j=0;j<VMHL_M;j++)  
        Matrix[i][j]=MHL_RandomUniformInt(10,100);  
  
MHL_ShowMatrix (Matrix,VMHL_N,VMHL_M,"Случайная матрица", "Matrix");  
// Случайная матрица:  
//Matrix =  
//70 57 44 95  
//26 21 60 63  
//61 55 27 95  
//10 10 43 92  
//66 20 51 65  
//32 52 63 78  
  
int k=2;//удалим вторую строку  
  
//Вызов функции  
TMHL_DeleteRowInMatrix(Matrix,k,VMHL_N,VMHL_M);  
  
//Используем полученный результат  
  
MHL_ShowMatrix (Matrix,VMHL_N,VMHL_M,"Матрица с удаленной строкой", "Matrix");  
// Матрица с удаленной строкой:  
//Matrix =  
//70 57 44 95  
//26 21 60 63  
//10 10 43 92  
//66 20 51 65
```

```

//32 52 63 78
//0 0 0 0

for (i=0;i<VMHL_N;i++) delete [] Matrix[i];
delete [] Matrix;

```

### 7.11.7 TMHL\_EqualityOfMatrixes

Функция проверяет равенство матриц.

Код 346. Синтаксис

```
template <class T> bool TMHL_EqualityOfMatrixes (T **a, T **b, int VMHL_N, int VMHL_M);
```

**Входные параметры:**

a — указатель на первую матрицу;

b — указатель на вторую матрицу;

VMHL\_N — размер массива a (число строк);

VMHL\_M — размер массива a (число столбцов).

**Возвращаемое значение:**

true — матрицы совпадают;

false — матрицы не совпадают.

Код 347. Пример использования

```

int i,j;
int VMHL_N=2;//Размер массива (число строк)
int VMHL_M=2;//Размер массива (число столбцов)

int **a;
a=new int*[VMHL_N];
for (i=0;i<VMHL_N;i++) a[i]=new int[VMHL_M];
//Заполним случайными числами
for (i=0;i<VMHL_N;i++)
    for (j=0;j<VMHL_M;j++)
        a[i][j]=MHL_RandomUniformInt(0,2);

int **b;
b=new int*[VMHL_N];
for (i=0;i<VMHL_N;i++) b[i]=new int[VMHL_M];
//Заполним случайными числами
for (i=0;i<VMHL_N;i++)
    for (j=0;j<VMHL_M;j++)
        b[i][j]=MHL_RandomUniformInt(0,2);

//Вызов функции
bool Equality=TMHL_EqualityOfMatrixes(a,b,VMHL_N,VMHL_M);

//Используем полученный результат
MHL_ShowMatrix (a,VMHL_N,VMHL_M,"Случайная матрица", "a");
//Случайная матрица:
//a =

```

```

//1   1
//0   1

MHL_ShowMatrix (b,VMHL_N,VMHL_M, "Случайная матрица", "b");
//Случайная матрица:
//b =
//1   1
//0   1

MHL_ShowNumber (Equality,"Равны ли матрицы", "Equality");
//Равны ли матрицы:
//Equality=1

for (i=0;i<VMHL_N;i++) delete [] a[i];
delete [] a;

for (i=0;i<VMHL_N;i++) delete [] b[i];
delete [] b;

```

### 7.11.8 TMHL\_FillMatrix

Функция заполняет матрицу значениями, равных x.

Код 348. Синтаксис

```

template <class T> void TMHL_FillMatrix(T **VMHL_ResultMatrix, int VMHL_N, int VMHL_M
, T x);

```

**Входные параметры:**

VMHL\_ResultMatrix — указатель на преобразуемый массив;

VMHL\_N — размер массива VMHL\_ResultMatrix (число строк);

VMHL\_M — размер массива VMHL\_ResultMatrix (число столбцов);

**Возвращаемое значение:**

Отсутствует.

Код 349. Пример использования

```

int i;
int VMHL_N=10;//Размер массива (число строк)
int VMHL_M=3;//Размер массива (число столбцов)
int **a;
a=new int*[VMHL_N];
for (i=0;i<VMHL_N;i++) a[i]=new int[VMHL_M];

int x=MHL_RandomUniformInt(0,10);//заполнитель

//Вызов функции
TMHL_FillMatrix(a,VMHL_N,VMHL_M,x);

//Используем полученный результат
MHL_ShowMatrix (a,VMHL_N,VMHL_M, "Заполненная матрица", "a");
//Заполненная матрица:
//a =
//3   3   3

```

```

//3 3 3
//3 3 3
//3 3 3
//3 3 3
//3 3 3
//3 3 3
//3 3 3
//3 3 3
//3 3 3
for (i=0;i<VMHL_N;i++) delete [] a[i];
delete [] a;

```

### 7.11.9 TMHL\_IdentityMatrix

Функция формирует единичную квадратную матрицу.

Код 350. Синтаксис

```
template <class T> void TMHL_IdentityMatrix(T **VMHL_ResultMatrix,int VMHL_N);
```

**Входные параметры:**

VMHL\_ResultMatrix — исходная матрица (в ней и сохраняется результат);

VMHL\_N — размер матрицы (число строк и столбцов).

**Возвращаемое значение:**

Отсутствует.

Код 351. Пример использования

```

int i;
int VMHL_N=5;//Размер массива (число строк и столбцов)
int **Matrix;
Matrix=new int*[VMHL_N];
for (i=0;i<VMHL_N;i++) Matrix[i]=new int[VMHL_N];

//Вызов функции
TMHL_IdentityMatrix(Matrix,VMHL_N);

//Используем полученный результат
MHL_ShowMatrix (Matrix,VMHL_N,VMHL_N,"Единичная матрица", "Matrix");
//Единичная матрица:
//Matrix =
//1 0 0 0 0
//0 1 0 0 0
//0 0 1 0 0
//0 0 0 1 0
//0 0 0 0 1

for (i=0;i<VMHL_N;i++) delete [] Matrix[i];
delete [] Matrix;

```

### 7.11.10 TMHL\_MatrixMinusMatrix

Функция вычитает две матрицы. Или для переопределенной варианта функция вычитает два матрицы и результат записывает в первую матрицу.

Код 352. Синтаксис

```
template <class T> void TMHL_MatrixMinusMatrix(T **a, T **b, T **VMHL_ResultMatrix,  
    int VMHL_N, int VMHL_M);  
template <class T> void TMHL_MatrixMinusMatrix(T **VMHL_ResultMatrix, T **b, int  
    VMHL_N, int VMHL_M);
```

#### Входные параметры:

a — первая матрица;  
b — вторая матрица;  
VMHL\_ResultMatrix — разница матриц;  
VMHL\_N — размер матриц (число строк);  
VMHL\_M — размер матриц (число столбцов).

#### Возвращаемое значение:

Отсутствует.

Для переопределенного варианта.

#### Входные параметры:

VMHL\_ResultMatrix — первая матрица (в ней и сохраняется разница);  
b — вторая матрица;  
VMHL\_N — размер матриц (число строк);  
VMHL\_M — размер матриц (число столбцов).

#### Возвращаемое значение:

Отсутствует.

Код 353. Пример использования

```
int i,j;  
int VMHL_N=5; //Размер массива (число строк)  
int VMHL_M=3; //Размер массива (число столбцов)  
int **a;  
a=new int*[VMHL_N];  
for (i=0;i<VMHL_N;i++) a[i]=new int[VMHL_M];  
int **b;  
b=new int*[VMHL_N];  
for (i=0;i<VMHL_N;i++) b[i]=new int[VMHL_M];  
int **c;  
c=new int*[VMHL_N];  
for (i=0;i<VMHL_N;i++) c[i]=new int[VMHL_M];  
//Заполним случайными числами  
for (i=0;i<VMHL_N;i++)  
    for (j=0;j<VMHL_M;j++)  
    {
```

```

    a[i][j]=MHL_RandomUniformInt(10,20);
    b[i][j]=MHL_RandomUniformInt(10,20);
}

//Вызов функции
TMHL_MatrixMinusMatrix(a,b,c,VMHL_N,VMHL_M);

//Используем полученный результат
MHL_ShowMatrix (a,VMHL_N,VMHL_M,"Матрица", "a");
//Матрица:
//a =
//18 19 17
//14 12 11
//10 16 19
//12 18 16
//12 16 11

MHL_ShowMatrix (b,VMHL_N,VMHL_M,"Матрица", "b");
//Матрица:
//b =
//11 19 18
//12 10 13
//11 14 10
//11 17 15
//12 16 10

MHL_ShowMatrix (c,VMHL_N,VMHL_M,"Их разница", "c");
//Их разница:
//c =
//7 0 -1
//2 2 -2
// -1 2 9
//1 1 1
//0 0 1

for (i=0;i<VMHL_N;i++) delete [] a[i];
delete [] a;
for (i=0;i<VMHL_N;i++) delete [] b[i];
delete [] b;
for (i=0;i<VMHL_N;i++) delete [] c[i];
delete [] c;

//Для переопределенной функции
VMHL_N=5;//Размер массива (число строк)
VMHL_M=3;//Размер массива (число столбцов)
a=new int*[VMHL_N];
for (i=0;i<VMHL_N;i++) a[i]=new int[VMHL_M];
b=new int*[VMHL_N];
for (i=0;i<VMHL_N;i++) b[i]=new int[VMHL_M];
//Заполним случайными числами
for (i=0;i<VMHL_N;i++)
    for (j=0;j<VMHL_M;j++)
    {
        a[i][j]=MHL_RandomUniformInt(10,20);
        b[i][j]=MHL_RandomUniformInt(10,20);
    }

MHL_ShowMatrix (a,VMHL_N,VMHL_M,"Матрица", "a");
//Матрица:
//a =

```

```

//11  18 11
//19  14 15
//14  13 14
//19  13 12
//19  15 10

//Вызов функции
TMHL_MatrixMinusMatrix(a,b,VMHL_N,VMHL_M);

//Используем полученный результат
MHL_ShowMatrix (b,VMHL_N,VMHL_M,"Матрица", "b");
//Матрица:
//b =
//12  13 18
//14  12 14
//12  14 19
//18  16 16
//16  17 19

MHL_ShowMatrix (a,VMHL_N,VMHL_M,"Теперь матрица а", "a");
//Теперь матрица а:
//a =
// -1  5  -7
// 5  2  1
// 2  -1  -5
// 1  -3  -4
// 3  -2  -9

for (i=0;i<VMHL_N;i++) delete [] a[i];
delete [] a;
for (i=0;i<VMHL_N;i++) delete [] b[i];
delete [] b;

```

### 7.11.11 TMHL\_MatrixMultiplyMatrix

Функция перемножает матрицы.

Код 354. Синтаксис

```

template <class T> void TMHL_MatrixMultiplyMatrix(T **a, T **b, T **VMHL_ResultMatrix
, int VMHL_N, int VMHL_M, int VMHL_S);

```

**Входные параметры:**

a — первый сомножитель, VMHL\_N x VMHL\_M;

b — второй сомножитель, VMHL\_M x VMHL\_S;

VMHL\_ResultMatrix — произведение матриц (сюда записывается результат), VMHL\_N x VMHL\_S;

VMHL\_N — число строк в матрице a;

VMHL\_M — число столбцов в матрице a и строк в матрице b;

VMHL\_S — число столбцов в матрице b.

**Возвращаемое значение:**

Отсутствует.

Код 355. Пример использования

```
int i;
int VMHL_N=3;
int VMHL_M=5;
int VMHL_S=4;
int **a;
a=new int*[VMHL_N];
for (i=0;i<VMHL_N;i++) a[i]=new int[VMHL_M];
int **b;
b=new int*[VMHL_M];
for (i=0;i<VMHL_M;i++) b[i]=new int[VMHL_S];
int **c;
c=new int*[VMHL_N];
for (i=0;i<VMHL_N;i++) c[i]=new int[VMHL_S];
TMHL_RandomIntMatrix(a,0,10,VMHL_N,VMHL_M);
TMHL_RandomIntMatrix(b,0,10,VMHL_M,VMHL_S);

//Вызов функции
TMHL_MatrixMultiplyMatrix (a, b, c, VMHL_N, VMHL_M, VMHL_S);

//Используем полученный результат
MHL_ShowMatrix (a,VMHL_N,VMHL_M,"Случайная матрица", "a");
//Случайная матрица:
//a =
//3 0 4 4 5
//9 4 3 4 4
//8 0 1 9 8

MHL_ShowMatrix (b,VMHL_M,VMHL_S,"Случайная матрица", "b");
// Случайная матрица:
//b =
//6 6 3
//4 2 1 2
//6 9 6 3
//1 1 8 2
//6 8 0 9

MHL_ShowMatrix (c,VMHL_N,VMHL_S,"Произведение", "c");
// Произведение:
//c =
//76 98 74 74
//116 125 108 88
//111 130 126 117

for (i=0;i<VMHL_N;i++) delete [] a[i];
delete [] a;
for (i=0;i<VMHL_M;i++) delete [] b[i];
delete [] b;
for (i=0;i<VMHL_N;i++) delete [] c[i];
delete [] c;
```

### 7.11.12 TMHL\_MatrixMultiplyMatrixT

Функция умножает матрицу на транспонированную матрицу.

Код 356. Синтаксис

```
template <class T> void TMHL_MatrixMultiplyMatrixT(T **a, T **b, T **
VMHL_ResultMatrix, int VMHL_N, int VMHL_M, int VMHL_S);
```

### Входные параметры:

a — первый сомножитель, VMHL\_N x VMHL\_M;

b — второй сомножитель (матрица, которую мы транспонируем), VMHL\_S x VMHL\_M;

VMHL\_ResultMatrix — произведение матриц (сюда записывается результат), VMHL\_N x VMHL\_S;

VMHL\_N — число строк в матрице a;

VMHL\_M — число столбцов в матрице a и столбцов в матрице b;

VMHL\_S — число строк в матрице b.

### Возвращаемое значение:

Отсутствует.

#### Код 357. Пример использования

```
int i;
int VMHL_N=3;
int VMHL_M=5;
int VMHL_S=4;
int **a;
a=new int*[VMHL_N];
for (i=0;i<VMHL_N;i++) a[i]=new int[VMHL_M];
int **b;
b=new int*[VMHL_S];
for (i=0;i<VMHL_S;i++) b[i]=new int[VMHL_M];
int **c;
c=new int*[VMHL_N];
for (i=0;i<VMHL_N;i++) c[i]=new int[VMHL_S];
TMHL_RandomIntMatrix(a,0,10,VMHL_N,VMHL_M);
TMHL_RandomIntMatrix(b,0,10,VMHL_S,VMHL_M);

//Вызов функции
TMHL_MatrixMultiplyMatrixT (a, b, c, VMHL_N, VMHL_M, VMHL_S);

//Используем полученный результат
MHL_ShowMatrix (a,VMHL_N,VMHL_M,"Случайная матрица", "a");
//Случайная матрица:
//a =
//9   8   5   2   1
//1   9   3   4   8
//9   9   3   0   3

MHL_ShowMatrix (b,VMHL_S,VMHL_M,"Случайная матрица", "b");
// Случайная матрица:
//b =
//3   7   3   0   8
//9   8   0   6   9
//0   2   5   6   5
//8   7   9   2   3

MHL_ShowMatrix (c,VMHL_N,VMHL_S,"Произведение", "c");
// Произведение:
```

```

//c =
//106 166 58 180
//139 177 97 130
//123 180 48 171

for (i=0;i<VMHL_N;i++) delete [] a[i];
delete [] a;
for (i=0;i<VMHL_S;i++) delete [] b[i];
delete [] b;
for (i=0;i<VMHL_N;i++) delete [] c[i];
delete [] c;

```

### 7.11.13 TMHL\_MatrixMultiplyNumber

Функция умножает матрицу на число.

Код 358. Синтаксис

```
template <class T> void TMHL_MatrixMultiplyNumber(T **VMHL_ResultMatrix, int VMHL_N,
    int VMHL_M, T Number);
```

**Входные параметры:**

VMHL\_ResultMatrix — указатель на исходную матрицу (в ней и сохраняется результат);

VMHL\_N — размер матрицы (число строк);

VMHL\_M — размер матрицы (число столбцов);

Number — число, на которое умножается матрица.

**Возвращаемое значение:**

Отсутствует.

Код 359. Пример использования

```

int i,j;
int VMHL_N=5;//Размер массива (число строк)
int VMHL_M=5;//Размер массива (число столбцов)
int **Matrix;
Matrix=new int*[VMHL_N];
for (i=0;i<VMHL_N;i++) Matrix[i]=new int[VMHL_M];
//Заполним случайными числами
for (i=0;i<VMHL_N;i++)
    for (j=0;j<VMHL_M;j++)
        Matrix[i][j]=MHL_RandomUniformInt(10,100);

MHL_ShowMatrix (Matrix,VMHL_N,VMHL_M,"Случайная матрица", "Matrix");
//Случайная матрица:
//Matrix =
//77 34 14 83 30
//31 15 87 68 20
//52 11 49 92 95
//77 29 96 50 90
//10 47 40 49 20

int Number=MHL_RandomUniformInt(-10,10);

```

```

//Вызов функции
TMHL_MatrixMultiplyNumber(Matrix,VMHL_N,VMHL_M,Number);

//Используем полученный результат
MHL_ShowNumber (Number,"Число, на которое умножается матрица","Number");
//Число, на которое умножается матрица:
//Number=4
MHL_ShowMatrix (Matrix,VMHL_N,VMHL_M,"Матрица умноженное на число", "Matrix");
//Матрица умноженное на число:
//Matrix =
//308 136 56 332 120
//124 60 348 272 80
//208 44 196 368 380
//308 116 384 200 360
//40 188 160 196 80

for (i=0;i<VMHL_N;i++) delete [] Matrix[i];
delete [] Matrix;

```

### 7.11.14 TMHL\_MatrixPlusMatrix

Функция суммирует две матрицы. Или для переопределенной варианта функция суммирует два матрицы и результат записывает в первую матрицу.

Код 360. Синтаксис

```

template <class T> void TMHL_MatrixPlusMatrix(T **a, T **b, T **VMHL_ResultMatrix,
    int VMHL_N, int VMHL_M);
template <class T> void TMHL_MatrixPlusMatrix(T **VMHL_ResultMatrix, T **b, int
    VMHL_N, int VMHL_M);

```

#### **Входные параметры:**

a — первая матрица;

b — вторая матрица;

VMHL\_ResultMatrix — сумма матриц;

VMHL\_N — размер матриц (число строк);

VMHL\_M — размер матриц (число столбцов).

#### **Возвращаемое значение:**

Отсутствует.

Для переопределенного варианта.

#### **Входные параметры:**

VMHL\_ResultMatrix — первая матрица (в ней и сохраняется сумма);

b — вторая матрица;

VMHL\_N — размер матриц (число строк);

VMHL\_M — размер матриц (число столбцов).

## **Возвращаемое значение:**

Отсутствует.

Код 361. Пример использования

```
int i,j;
int VMHL_N=5; //Размер массива (число строк)
int VMHL_M=3; //Размер массива (число столбцов)
int **a;
a=new int*[VMHL_N];
for (i=0;i<VMHL_N;i++) a[i]=new int[VMHL_M];
int **b;
b=new int*[VMHL_N];
for (i=0;i<VMHL_N;i++) b[i]=new int[VMHL_M];
int **c;
c=new int*[VMHL_N];
for (i=0;i<VMHL_N;i++) c[i]=new int[VMHL_M];
//Заполним случайными числами
for (i=0;i<VMHL_N;i++)
    for (j=0;j<VMHL_M;j++)
    {
        a[i][j]=MHL_RandomUniformInt(10,20);
        b[i][j]=MHL_RandomUniformInt(10,20);
    }

//Вызов функции
TMHL_MatrixPlusMatrix(a,b,c,VMHL_N,VMHL_M);

//Используем полученный результат
MHL_ShowMatrix (a,VMHL_N,VMHL_M,"Матрица", "a");
//Матрица:
//a =
//18  15  15
//15  11  17
//19  14  10
//17  18  18
//19  15  16

MHL_ShowMatrix (b,VMHL_N,VMHL_M,"Матрица", "b");
//Матрица:
//b =
//17  15  15
//16  18  10
//17  12  15
//12  16  13
//15  14  10

MHL_ShowMatrix (c,VMHL_N,VMHL_M,"Их сумма", "c");
//Их сумма:
//c =
//35  30  30
//31  29  27
//36  26  25
//29  34  31
//34  29  26

for (i=0;i<VMHL_N;i++) delete [] a[i];
delete [] a;
for (i=0;i<VMHL_N;i++) delete [] b[i];
delete [] b;
```

```

for (i=0;i<VMHL_N;i++) delete [] c[i];
delete [] c;

//Для переопределенной функции
VMHL_N=5; //Размер массива (число строк)
VMHL_M=3; //Размер массива (число столбцов)
a=new int*[VMHL_N];
for (i=0;i<VMHL_N;i++) a[i]=new int[VMHL_M];
b=new int*[VMHL_N];
for (i=0;i<VMHL_N;i++) b[i]=new int[VMHL_M];
//Заполним случайными числами
for (i=0;i<VMHL_N;i++)
    for (j=0;j<VMHL_M;j++)
    {
        a[i][j]=MHL_RandomUniformInt(10,20);
        b[i][j]=MHL_RandomUniformInt(10,20);
    }

MHL_ShowMatrix (a,VMHL_N,VMHL_M, "Матрица", "a");
//Матрица:
//a =
//18 12 12
//19 17 12
//19 17 17
//11 10 17
//11 19 10

//Вызов функции
TMHL_MatrixPlusMatrix(a,b,VMHL_N,VMHL_M);

//Используем полученный результат
MHL_ShowMatrix (b,VMHL_N,VMHL_M, "Матрица", "b");
//Матрица:
//b =
//10 10 16
//10 18 18
//15 13 17
//13 11 14
//16 13 11

MHL_ShowMatrix (a,VMHL_N,VMHL_M, "Теперь матрица а", "a");
//Теперь матрица а:
//a =
//28 22 28
//29 35 30
//34 30 34
//24 21 31
//27 32 21

for (i=0;i<VMHL_N;i++) delete [] a[i];
delete [] a;
for (i=0;i<VMHL_N;i++) delete [] b[i];
delete [] b;

```

### 7.11.15 TMHL\_MatrixT

Функция транспонирует матрицу.

Код 362. Синтаксис

```
template <class T> void TMHL_MatrixT(T **a, T **VMHL_ResultMatrix, int VMHL_N, int VMHL_M);
```

**Входные параметры:**

a — исходная матрица, (VMHL\_N x VMHL\_M);

VMHL\_ResultMatrix — транспонированная матрица, (VMHL\_M x VMHL\_N);

VMHL\_N — размер матрицы (число строк) в матрице a;

VMHL\_M — размер матрицы (число столбцов) в матрице a.

**Возвращаемое значение:**

Отсутствует.

Код 363. Пример использования

```
int i,j;
int VMHL_N=5; //Размер массива (число строк)
int VMHL_M=3; //Размер массива (число столбцов)
int **Matrix;
Matrix=new int*[VMHL_N];
for (i=0;i<VMHL_N;i++) Matrix[i]=new int[VMHL_M];
int **MatrixT;
MatrixT=new int*[VMHL_M];
for (i=0;i<VMHL_M;i++) MatrixT[i]=new int[VMHL_N];
//Заполним случайными числами
for (i=0;i<VMHL_N;i++)
    for (j=0;j<VMHL_M;j++)
        Matrix[i][j]=MHL_RandomUniformInt(10,100);

//Вызов функции
TMHL_MatrixT(Matrix,MatrixT,VMHL_N,VMHL_M);

//Используем полученный результат
MHL_ShowMatrix (Matrix,VMHL_N,VMHL_M,"Матрица", "Matrix");
//Матрица:
//Matrix =
//26  64  62
//70  49  43
//50  41  50
//76  75  81
//26  72  24

MHL_ShowMatrix (MatrixT,VMHL_M,VMHL_N,"Транспонированная матрица", "MatrixT");
// Транспонированная матрица:
//MatrixT =
//26  70  50  76  26
//64  49  41  75  72
//62  43  50  81  24

for (i=0;i<VMHL_N;i++) delete [] Matrix[i];
delete [] Matrix;
for (i=0;i<VMHL_M;i++) delete [] MatrixT[i];
delete [] MatrixT;
```

### 7.11.16 TMHL\_MatrixTMultiplyMatrix

Функция умножает транспонированную матрицу на матрицу.

Код 364. Синтаксис

```
template <class T> void TMHL_MatrixTMultiplyMatrix(T **a, T **b, T **  
VMHL_ResultMatrix, int VMHL_N, int VMHL_M, int VMHL_S);
```

**Входные параметры:**

a — первый сомножитель (матрица, которую мы транспонируем), VMHL\_M x VMHL\_N;

b — второй сомножитель, VMHL\_M x VMHL\_S;

VMHL\_ResultMatrix — произведение матриц (сюда записывается результат), VMHL\_N x VMHL\_S;

VMHL\_N — число столбцов в матрице a;

VMHL\_M — число строк в матрице a и строк в матрице b;

VMHL\_S — число столбцов в матрице b.

**Возвращаемое значение:**

Отсутствует.

Код 365. Пример использования

```
int i;  
int VMHL_N=3;  
int VMHL_M=5;  
int VMHL_S=4;  
int **a;  
a=new int*[VMHL_M];  
for (i=0;i<VMHL_M;i++) a[i]=new int[VMHL_N];  
int **b;  
b=new int*[VMHL_M];  
for (i=0;i<VMHL_M;i++) b[i]=new int[VMHL_S];  
int **c;  
c=new int*[VMHL_N];  
for (i=0;i<VMHL_N;i++) c[i]=new int[VMHL_S];  
TMHL_RandomIntMatrix(a,0,10,VMHL_M,VMHL_N);  
TMHL_RandomIntMatrix(b,0,10,VMHL_M,VMHL_S);  
  
//Вызов функции  
TMHL_MatrixTMultiplyMatrix (a, b, c, VMHL_N, VMHL_M, VMHL_S);  
  
//Используем полученный результат  
MHL_ShowMatrix (a,VMHL_M,VMHL_N,"Случайная матрица", "a");  
// Случайная матрица:  
//a =  
//6 0 1  
//6 5 9  
//7 2 0  
//3 1 5  
//3 8 8  
  
MHL_ShowMatrix (b,VMHL_M,VMHL_S,"Случайная матрица", "b");  
// Случайная матрица:
```

```

//b =
//6   7   1   0
//7   6   0   0
//5   6   0   0
//9   7   9   3
//5   7   0   1

MHL_ShowMatrix (c,VMHL_N,VMHL_S, "Произведение", "c");
// Произведение:
//c =
//155 162    33 12
//94  105    9  11
//154 152    46 23

for (i=0;i<VMHL_M;i++) delete [] a[i];
delete [] a;
for (i=0;i<VMHL_M;i++) delete [] b[i];
delete [] b;
for (i=0;i<VMHL_N;i++) delete [] c[i];
delete [] c;

```

### 7.11.17 TMHL\_MatrixToCol

Функция копирует из матрицы (двумерного массива) в вектор столбец.

Код 366. Синтаксис

```
template <class T> void TMHL_MatrixToCol(T **a, T *VMHL_ResultVector, int VMHL_N, int k);
```

**Входные параметры:**

a — указатель на матрицу;

VMHL\_ResultVector — указатель на вектор;

VMHL\_N — количество строк в матрице и одновременно размер массива b;

k — номер копируемого столбца (начиная с 0).

**Возвращаемое значение:**

Отсутствует.

Код 367. Пример использования

```

int i,j;
int VMHL_N=10;//Размер массива (число строк)
int VMHL_M=3;//Размер массива (число столбцов)
int **a;
a=new int*[VMHL_N];
for (i=0;i<VMHL_N;i++) a[i]=new int[VMHL_M];
int *b;
b=new int[VMHL_N];
//Заполним случайными числами
for (i=0;i<VMHL_N;i++)
  for (j=0;j<VMHL_M;j++)
    a[i][j]=MHL_RandomUniformInt(10,100);

```

```

int k=1; //Номер копируемого столбца

//Вызов функции
TMHL_MatrixToCol(a,b,VMHL_N,k);

//Используем полученный результат
MHL_ShowMatrix (a,VMHL_N,VMHL_M,"Случайная матрица", "a");
//Случайная матрица:
//a =
//98 39 35
//18 30 95
//68 81 59
//43 20 23
//94 40 14
//17 36 84
//98 13 69
//11 94 63
//62 80 22
//27 17 58

MHL_ShowNumber(k,"Номер копируемого столбца ", "k");
//Номер копируемого столбца :
//k=1
MHL_ShowVector (b,VMHL_N,"Вектор, в который скопировали столбец", "b");
//Вектор, в который скопировали столбец:
//b =
//39
//30
//81
//20
//40
//36
//13
//94
//80
//17

for (i=0;i<VMHL_N;i++) delete [] a[i];
delete [] a;
delete [] b;

```

### 7.11.18 TMHL\_MatrixToMatrix

Функция копирует содержимое матрицы (двумерного массива) а в массив VMHL\_ResultMatrix.

Код 368. Синтаксис

```

template <class T> void TMHL_MatrixToMatrix(T **a, T **VMHL_ResultMatrix, int VMHL_N,
                                             int VMHL_M);

```

#### Входные параметры:

а — указатель на исходный массив;

VMHL\_ResultMatrix — указатель на массив в который производится запись;

VMHL\_N — размер массива (число строк);

VMHL\_M — размер массива (число столбцов).

### Возвращаемое значение:

Отсутствует.

Код 369. Пример использования

```
int i,j;
int VMHL_N=10; //Размер массива (число строк)
int VMHL_M=3; //Размер массива (число столбцов)
double **a;
a=new double*[VMHL_N];
for (i=0;i<VMHL_N;i++) a[i]=new double[VMHL_M];
double **b;
b=new double*[VMHL_N];
for (i=0;i<VMHL_N;i++) b[i]=new double[VMHL_M];
//Заполним случайными числами
for (i=0;i<VMHL_N;i++)
    for (j=0;j<VMHL_M;j++)
        a[i][j]=MHL_RandomUniformInt(10,100);

//Вызов функции
TMHL_MatrixToMatrix(a,b,VMHL_N,VMHL_M);

//Используем полученный результат
MHL_ShowMatrix (a,VMHL_N,VMHL_M,"Случайная матрица", "a");
//Случайная матрица:
//a =
//82  55 19
//38  82 91
//68  67 50
//82  62 63
//24  41 69
//16  47 29
//18  92 63
//11  29 30
//71  49 64
//11  95 38

MHL_ShowMatrix (b,VMHL_N,VMHL_M,"Теперь b равна a", "b");
//Теперь b равна a:
//b =
//82  55 19
//38  82 91
//68  67 50
//82  62 63
//24  41 69
//16  47 29
//18  92 63
//11  29 30
//71  49 64
//11  95 38

for (i=0;i<VMHL_N;i++) delete [] a[i];
delete [] a;
for (i=0;i<VMHL_N;i++) delete [] b[i];
delete [] b;
```

### 7.11.19 TMHL\_MatrixToRow

Функция копирует из матрицы (двумерного массива) в вектор строку.

Код 370. Синтаксис

```
template <class T> void TMHL_MatrixToRow(T **a, T *VMHL_ResultVector, int k, int VMHL_M);
```

**Входные параметры:**

a — указатель на матрицу;

VMHL\_ResultVector — указатель на вектор;

k — номер копируемой строки (начиная с 0);

VMHL\_M — количество столбцов в матрице и одновременно размер массива VMHL\_ResultVector.

**Возвращаемое значение:**

Отсутствует.

Код 371. Пример использования

```
int i,j;
int VMHL_N=10; //Размер массива (число строк)
int VMHL_M=3; //Размер массива (число столбцов)
int **a;
a=new int*[VMHL_N];
for (i=0;i<VMHL_N;i++) a[i]=new int[VMHL_M];
int *b;
b=new int[VMHL_M];
//Заполним случайными числами
for (i=0;i<VMHL_N;i++)
    for (j=0;j<VMHL_M;j++)
        a[i][j]=MHL_RandomUniformInt(10,100);

int k=1; //Номер копируемой строки

//Вызов функции
TMHL_MatrixToRow(a,b,k,VMHL_M);

//Используем полученный результат
MHL_ShowMatrix (a,VMHL_N,VMHL_M,"Случайная матрица", "a");
//Случайная матрица:
//a =
//31  57  29
//69  75  13
//85  14  75
//78  91  11
//83  23  94
//79  48  31
//43  18  70
//80  18  15
//38  95  78
//16  90  69

MHL_ShowNumber(k,"Номер копируемой строки ","k");
//Номер копируемой строки :
```

```

//k=1
MHL_ShowVector (b,VMHL_M,"Вектор, в который скопировали строку", "b");
//Вектор, в который скопировали строку:
//b =
//69
//75
//13

for (i=0;i<VMHL_N;i++) delete [] a[i];
delete [] a;
delete [] b;

```

### 7.11.20 TMHL\_MaximumOfMatrix

Функция ищет максимальный элемент в матрице (двумерном массиве).

Код 372. Синтаксис

```
template <class T> T TMHL_MaximumOfMatrix(T **a, int VMHL_N, int VMHL_M);
```

**Входные параметры:**

a — указатель на матрицу;

VMHL\_N — размер массива (число строк);

VMHL\_M — размер массива (число столбцов).

**Возвращаемое значение:**

Максимальный элемент.

Код 373. Пример использования

```

int i,j;
int VMHL_N=10;//Размер массива (число строк)
int VMHL_M=3;//Размер массива (число столбцов)
double **Matrix;
Matrix=new double*[VMHL_N];
for (i=0;i<VMHL_N;i++) Matrix[i]=new double[VMHL_M];
//Заполним случайными числами
for (i=0;i<VMHL_N;i++)
    for (j=0;j<VMHL_M;j++)
        Matrix[i][j]=MHL_RandomUniformInt(10,100);

//Вызов функции
double Maximum=TMHL_MaximumOfMatrix(Matrix,VMHL_N,VMHL_M);

//Используем полученный результат
MHL_ShowMatrix (Matrix,VMHL_N,VMHL_M,"Случайная матрица", "Matrix");
//Случайная матрица:
//Matrix =
//25  42  79
//99  34  34
//16  80  41
//12  95  78
//67  27  14
//29  20  93
//57  66  17

```

```

//52  38  42
//31  96  27
//39  77  50

MHL_ShowNumber(Maximum, "Максимальный элемент", "Maximum");
//Максимальный элемент:
//Maximum=96

for (i=0;i<VMHL_N;i++) delete [] Matrix[i];
delete [] Matrix;

```

### 7.11.21 TMHL\_MinimumOfMatrix

Функция ищет минимальный элемент в матрице (двумерном массиве).

Код 374. Синтаксис

```
template <class T> T TMHL_MinimumOfMatrix(T **a, int VMHL_N, int VMHL_M);
```

**Входные параметры:**

a — указатель на матрицу;

VMHL\_N — размер массива (число строк);

VMHL\_M — размер массива (число столбцов).

**Возвращаемое значение:**

Минимальный элемент.

Код 375. Пример использования

```

int i,j;
int VMHL_N=10;//Размер массива (число строк)
int VMHL_M=3;//Размер массива (число столбцов)
double **Matrix;
Matrix=new double*[VMHL_N];
for (i=0;i<VMHL_N;i++) Matrix[i]=new double[VMHL_M];
//Заполним случайными числами
for (i=0;i<VMHL_N;i++)
    for (j=0;j<VMHL_M;j++)
        Matrix[i][j]=MHL_RandomUniformInt(10,100);

//Вызов функции
double Minimum=TMHL_MinimumOfMatrix(Matrix,VMHL_N,VMHL_M);

//Используем полученный результат
MHL>ShowMatrix (Matrix,VMHL_N,VMHL_M,"Случайная матрица", "Matrix");
//Случайная матрица:
//Matrix =
//29  64  95
//55  25  36
//73  31  62
//54  19  22
//29  78  48
//24  40  46
//82  13  90
//66  23  14

```

```

//44 45 56
//73 92 16

MHL_ShowNumber(Minimum, "Минимальный элемент", "Minimum");
//Минимальный элемент:
//Minimum=13

for (i=0;i<VMHL_N;i++) delete [] Matrix[i];
delete [] Matrix;

```

### 7.11.22 TMHL\_MixingRowsInOrder

Функция меняет строки матрицы в порядке, указанным в массиве b.

Код 376. Синтаксис

```

template <class T> void TMHL_MixingRowsInOrder(T **VMHL_ResultMatrix, int *b, int
VMHL_N, int VMHL_M);

```

**Входные параметры:**

VMHL\_ResultMatrix — указатель на матрицу, в которой меняем порядок строк;

b — вектор, в котором записаны номера, под которыми должны стать строки в матрице (от 0 до VMHL\_N-1);

VMHL\_N — размер массива (число строк);

VMHL\_M — размер массива (число столбцов).

**Возвращаемое значение:**

Отсутствует.

Код 377. Пример использования

```

int i;
int VMHL_N=7;//Размер массива (число строк)
int VMHL_M=3;//Размер массива (число столбцов)

int *b;
b=new int[VMHL_N];

int **a;
a=new int*[VMHL_N];
for (i=0;i<VMHL_N;i++) a[i]=new int[VMHL_M];
//Заполним случайными числами
TMHL_RandomIntMatrix(a,10,100,VMHL_N,VMHL_M);
MHL_ShowMatrix (a,VMHL_N,VMHL_M,"Случайная матрица", "a");
//Случайная матрица:
//a =
//49 65 82
//92 73 27
//10 72 80
//87 62 12
//82 11 75
//15 75 94
//56 96 39

```

```

//Первончальный порядок
TMHL_OrdinalVectorZero(b, VMHL_N);
//Перемешаем
TMHL_MixingVector(b, 0.5, VMHL_N);

//Вызов функции
TMHL_MixingRowsInOrder(a, b, VMHL_N, VMHL_M);

//Используем полученный результат

MHL_ShowVector (b, VMHL_N, "Номера нового порядка", "b");
//Номера нового порядка:
//b =
//5
//0
//1
//4
//6
//2
//3

MHL_ShowMatrix (a, VMHL_N, VMHL_M, "Случайная матрица с новым порядком строк", "a");
//Случайная матрица с новым порядком строк:
//a =
//92  73  27
//10  72  80
//15  75  94
//56  96  39
//87  62  12
//49  65  82
//82  11  75

for (i=0;i<VMHL_N;i++) delete [] a[i];
delete [] a;
delete [] b;

```

### 7.11.23 TMHL\_NumberOfDifferentValuesInMatrix

Функция подсчитывает число различных значений в матрице.

Код 378. Синтаксис

```

template <class T> int TMHL_NumberOfDifferentValuesInMatrix(T **a, int VMHL_N, int
VMHL_M);

```

**Входные параметры:**

a — указатель на матрица;

VMHL\_N — размер массива а (строки);

VMHL\_M — размер массива а (столбцы).

**Возвращаемое значение:**

Отсутствует.

**Примечание:**

Алгоритм очень топорный и медленный.

Код 379. Пример использования

```
int i,j;
int VMHL_N=5; //Размер массива (число строк)
int VMHL_M=3; //Размер массива (число столбцов)
int **a;
a=new int*[VMHL_N];
for (i=0;i<VMHL_N;i++) a[i]=new int[VMHL_M];
//Заполним случайными числами
for (i=0;i<VMHL_N;i++)
    for (j=0;j<VMHL_M;j++)
        a[i][j]=MHL_RandomUniformInt(0,50);

//Вызов функции
int NumberOfDifferent=TMHL_NumberOfDifferentValuesInMatrix(a,VMHL_N,VMHL_M);

//Используем полученный результат
MHL_ShowMatrix (a,VMHL_N,VMHL_M,"Случайная матрица", "a");
//Случайная матрица:
//a =
//7   3   27
//31  30  10
//37  34  49
//45  26  12
//26  28  0

MHL_ShowNumber (NumberOfDifferent,"Число различных значений в матрице", "
    NumberOfDifferent");
//Число различных значений в матрице:
//NumberOfDifferent=14
for (i=0;i<VMHL_N;i++) delete [] a[i];
delete [] a;
```

#### 7.11.24 TMHL\_RowInterchange

Функция переставляет строки матрицы.

Код 380. Синтаксис

```
template <class T> void TMHL_MatrixToRow(T **a, T *VMHL_ResultVector, int k, int
    VMHL_M);
```

**Входные параметры:**

VMHL\_ResultMatrix — указатель на исходную матрицу (в ней и сохраняется результат);

VMHL\_M — размер массива (число столбцов);

k,l — номера переставляемых строк (нумерация с нуля).

**Возвращаемое значение:**

Отсутствует.

Код 381. Пример использования

```
int i,j;
```

```

int VMHL_N=5; //Размер массива (число строк)
int VMHL_M=5; //Размер массива (число столбцов)
int **Matrix;
Matrix=new int*[VMHL_N];
for (i=0;i<VMHL_N;i++) Matrix[i]=new int[VMHL_M];
//Заполним случайными числами
for (i=0;i<VMHL_N;i++)
    for (j=0;j<VMHL_M;j++)
        Matrix[i][j]=MHL_RandomUniformInt(10,100);

MHL_ShowMatrix (Matrix,VMHL_N,VMHL_M, "Случайная матрица", "Matrix");
//Случайная матрица:
//Matrix =
//64 41 93 98 45
//19 55 31 38 44
//38 78 39 44 86
//28 54 39 14 72
//31 99 64 49 63

// номера переставляемых строк
int k=MHL_RandomUniformInt(0,5);
int l=MHL_RandomUniformInt(0,5);

//Вызов функции
TMHL_RowInterchange(Matrix,VMHL_M,k,l);

//Используем полученный результат
MHL_ShowNumber (k,"Номер первой строки","k");
//Номер первой строки:
//k=4
MHL_ShowNumber (l,"Номер второй строки","l");
//Номер второй строки:
//l=3
MHL_ShowMatrix (Matrix,VMHL_N,VMHL_M, "Матрица с переставленными строками", "Matrix");
//Матрица с переставленными строками:
//Matrix =
//64 41 93 98 45
//19 55 31 38 44
//38 78 39 44 86
//31 99 64 49 63
//28 54 39 14 72

for (i=0;i<VMHL_N;i++) delete [] Matrix[i];
delete [] Matrix;

```

### 7.11.25 TMHL\_RowToMatrix

Функция копирует в матрицу (двумерный массив) из вектора строку.

Код 382. Синтаксис

```
template <class T> void TMHL_RowToMatrix(T **VMHL_ResultMatrix, T *b, int k, int VMHL_M);
```

**Входные параметры:**

VMHL\_ResultMatrix — указатель на матрицу;

b — указатель на вектор;

k — номер строки, в которую будет происходить копирование (начиная с 0);

VMHL\_M — количество столбцов в матрице и одновременно размер массива b.

### Возвращаемое значение:

Отсутствует.

Код 383. Пример использования

```
int i,j;
int VMHL_N=10; //Размер массива (число строк)
int VMHL_M=3; //Размер массива (число столбцов)
int **a;
a=new int*[VMHL_N];
for (i=0;i<VMHL_N;i++) a[i]=new int[VMHL_M];
int *b;
b=new int[VMHL_M];
//Заполним случайными числами
for (i=0;i<VMHL_N;i++)
    for (j=0;j<VMHL_M;j++)
        a[i][j]=MHL_RandomUniformInt(10,100);
MHL_ShowMatrix (a,VMHL_N,VMHL_M,"Случайная матрица", "a");
//Случайная матрица:
//a =
//53 15 56
//47 85 84
//82 56 58
//24 34 53
//42 34 20
//76 46 24
//93 17 17
//73 31 26
//29 63 20
//84 83 98

for (j=0;j<VMHL_M;j++)
    b[j]=MHL_RandomUniformInt(10,100);

int k=1; //Номер строки, в которую мы копируем

//Вызов функции
TMHL_RowToMatrix(a,b,k,VMHL_M);

//Используем полученный результат
MHL_ShowNumber(k,"Номер строки, в которую мы копируем ","k");
//Номер строки, в которую мы копируем :
//k=1
MHL_ShowVector (b,VMHL_M,"Вектор", "b");
//Вектор:
//b =
//92
//89
//11

MHL_ShowMatrix (a,VMHL_N,VMHL_M,"Матрица с изменившейся строкой", "a");
//Матрица с изменившейся строкой:
//a =
//53 15 56
//92 89 11
```

```

//82  56 58
//24  34 53
//42  34 20
//76  46 24
//93  17 17
//73  31 26
//29  63 20
//84  83 98

for (i=0;i<VMHL_N;i++) delete [] a[i];
delete [] a;
delete [] b;

```

### 7.11.26 TMHL\_SumMatrix

Функция вычисляет сумму элементов матрицы.

Код 384. Синтаксис

```
template <class T> T TMHL_SumMatrix(T **a,int VMHL_N,int VMHL_M);
```

**Входные параметры:**

a — указатель на исходный массив;

VMHL\_N — размер массива a (число строк);

VMHL\_M — размер массива a (число столбцов).

**Возвращаемое значение:**

Сумма элементов матрицы.

Код 385. Пример использования

```

int i,j;
int VMHL_N=10;//Размер массива (число строк)
int VMHL_M=3;//Размер массива (число столбцов)
int **a;
a=new int*[VMHL_N];
for (i=0;i<VMHL_N;i++) a[i]=new int[VMHL_M];
//Заполним случайными числами
for (i=0;i<VMHL_N;i++)
    for (j=0;j<VMHL_M;j++)
        a[i][j]=MHL_RandomUniformInt(10,100);

//Вызов функции
int SumMatrix=TMHL_SumMatrix(a,VMHL_N,VMHL_M);

//Используем полученный результат
MHL_ShowMatrix (a,VMHL_N,VMHL_M,"Случайная матрица", "a");
//Случайная матрица:
//a =
//93  11 72
//58  74 66
//39  16 46
//87  23 76
//85  60 13
//34  43 63

```

```

//11  99 20
//77  93 70
//68  32 65
//36  74 35

MHL_ShowNumber (SumMatrix, "Её сумма", "SumVector");
//Её сумма:
//SumVector=1639

for (i=0;i<VMHL_N;i++) delete [] a[i];
delete [] a;

```

### 7.11.27 TMHL\_ZeroMatrix

Функция зануляет матрицу.

Код 386. Синтаксис

```
template <class T> void TMHL_ZeroMatrix(T **VMHL_ResultMatrix, int VMHL_N, int VMHL_M);
```

**Входные параметры:**

VMHL\_ResultMatrix — указатель на преобразуемый массив;

VMHL\_N — размер массива VMHL\_ResultMatrix (число строк);

VMHL\_M — размер массива VMHL\_ResultMatrix (число столбцов).

**Возвращаемое значение:**

Отсутствует.

Код 387. Пример использования

```

int i,j;
int VMHL_N=10;//Размер массива (число строк)
int VMHL_M=3;//Размер массива (число столбцов)
int **a;
a=new int*[VMHL_N];
for (i=0;i<VMHL_N;i++) a[i]=new int[VMHL_M];
//Заполним случайными числами
for (i=0;i<VMHL_N;i++)
    for (j=0;j<VMHL_M;j++)
        a[i][j]=MHL_RandomUniformInt(10,100);

//Вызов функции
int SumMatrix=TMHL_SumMatrix(a,VMHL_N,VMHL_M);

//Используем полученный результат
MHL_ShowMatrix (a,VMHL_N,VMHL_M,"Случайная матрица", "a");
//Случайная матрица:
//a =
//93  11 72
//58  74 66
//39  16 46
//87  23 76
//85  60 13
//34  43 63
//11  99 20

```

```

//77  93 70
//68  32 65
//36  74 35

MHL_ShowNumber (SumMatrix, "Её сумма", "SumVector");
//Её сумма:
//SumVector=1639

for (i=0;i<VMHL_N;i++) delete [] a[i];
delete [] a;

```

## 7.12 Метрика

### 7.12.1 TMHL\_Chebychev

Функция вычисляет расстояние Чебышева.

Код 388. Синтаксис

```
template <class T> T TMHL_Chebychev(T *x, T *y, int VMHL_N);
```

**Входные параметры:**

x — указатель на первый вектор;

y — указатель на второй вектор;

VMHL\_N — размер массивов.

**Возвращаемое значение:**

Значение метрики расстояние Чебышева.

**Формула:**

$$S(\bar{x}, \bar{y}) = \max_{i \in \overline{1, n}} (|x_i - y_i|).$$

Код 389. Пример использования

```

int VMHL_N=5; //Размер массива
double *x;
x=new double[VMHL_N];
double *y;
y=new double[VMHL_N];
//Заполним случайными числами
MHL_RandomRealVector (x, 0, 10, VMHL_N);
MHL_RandomRealVector (y, 0, 10, VMHL_N);

//Вызов функции
double metric=TMHL_Chebychev(x, y, VMHL_N);

//Используем полученный результат
MHL_ShowVector (x, VMHL_N, "Первый массив", "x");
//Первый массив:
//x =
//7.9245

```

```

//7.28699
//6.24054
//1.12152
//7.65442

MHL_ShowVector (y,VMHL_N,"Второй массив", "y");
//Второй массив:
//y =
//0.324097
//3.12164
//4.47266
//9.70062
//5.64117

MHL_ShowNumber (metric,"Значение метрики расстояние Чебышева", "metric");
//Значение метрики расстояние Чебышева:
//metric=8.5791

delete [] x;
delete [] y;

```

## 7.12.2 TMHL\_CityBlock

Функция вычисляет манхэттенское расстояние между двумя массивами.

Код 390. Синтаксис

```
template <class T> T TMHL_CityBlock(T *x, T *y, int VMHL_N);
```

**Входные параметры:**

x — указатель на первый вектор;

y — указатель на второй вектор;

VMHL\_N — размер массивов.

**Возвращаемое значение:**

Значение метрики манхэттенское расстояние.

**Формула:**

$$S(\bar{x}, \bar{y}) = \sum_{i=1}^n |x_i - y_i|.$$

Код 391. Пример использования

```

int i;
int VMHL_N=5; //Размер массива
int *x;
x=new int[VMHL_N];
int *y;
y=new int[VMHL_N];
//Заполним случайными числами
for (i=0;i<VMHL_N;i++)
{

```

```

x[i]=MHL_RandomUniformInt(0,5);
y[i]=MHL_RandomUniformInt(0,5);
}

//Вызов функции
int metric=TMHL_CityBlock(x,y,VMHL_N);

//Используем полученный результат
MHL_ShowVector (x,VMHL_N,"Первый массив", "x");
//Первый массив:
//x =
//4
//1
//3
//3
//0

MHL_ShowVector (y,VMHL_N,"Второй массив", "y");
//Второй массив:
//y =
//3
//4
//1
//4
//1

MHL_ShowNumber (metric,"Значение метрики манхэттенское расстояние", "metric");
// Значение метрики манхэттенское расстояние:
//metric=8

delete [] x;
delete [] y;

```

### 7.12.3 TMHL\_Euclid

Функция вычисляет евклидово расстояние.

Код 392. Синтаксис

```
template <class T> T TMHL_Euclid(T *x, T *y, int VMHL_N);
```

**Входные параметры:**

x — указатель на первый вектор;

y — указатель на второй вектор;

VMHL\_N — размер массивов.

**Возвращаемое значение:**

Значение метрики евклидово расстояние.

**Формула:**

$$S(\bar{x}, \bar{y}) = \sqrt{\sum_{i=1}^n (x_i - y_i)^2}.$$

Код 393. Пример использования

```
int VMHL_N=5; //Размер массива
double *x;
x=new double[VMHL_N];
double *y;
y=new double[VMHL_N];
//Заполним случайными числами
MHL_RandomRealVector (x,0,10,VMHL_N);
MHL_RandomRealVector (y,0,10,VMHL_N);

//Вызов функции
double metric=TMHL_Euclid(x,y,VMHL_N);

//Используем полученный результат
MHL_ShowVector (x,VMHL_N,"Первый массив", "x");
//Первый массив:
//x =
//3.15491
//4.04266
//2.63519
//9.94141
//3.2193

MHL_ShowVector (y,VMHL_N,"Второй массив", "y");
//Второй массив:
//y =
//9.4516
//2.59064
//2.56348
//4.78271
//5.78705

    MHL_ShowNumber (metric,"Значение метрики евклидово расстояние", "metric");
//Значение метрики евклидово расстояние:
//metric=8.65837

delete [] x;
delete [] y;
```

#### 7.12.4 TMHL\_Minkovski

Функция вычисляет метрику Минковского.

Код 394. Синтаксис

```
template <class T> T TMHL_Minkovski(T *x, T *y, int r, int VMHL_N);
```

**Входные параметры:**

х — указатель на первый вектор;

у — указатель на второй вектор;

r — порядок метрики;

VMHL\_N — размер массивов.

**Возвращаемое значение:**

Значение метрики Минковского.

**Формула:**

$$S(\bar{x}, \bar{y}) = \sqrt[1/r]{\sum_{i=1}^n |x_i - y_i|^r}.$$

Код 395. Пример использования

```
int VMHL_N=5; //Размер массива
double *x;
x=new double[VMHL_N];
double *y;
y=new double[VMHL_N];
//Заполним случайными числами
MHL_RandomRealVector (x,0,10,VMHL_N);
MHL_RandomRealVector (y,0,10,VMHL_N);

int r=3;

//Вызов функции
double metric=TMHL_Minkovski(x,y,r,VMHL_N);

//Используем полученный результат
MHL_ShowVector (x,VMHL_N,"Первый массив", "x");
//Первый массив:
//x =
//8.2312
//2.74628
//9.36371
//7.31995
//0.139465

MHL_ShowVector (y,VMHL_N,"Второй массив", "y");
//Второй массив:
//y =
//6.2793
//5.07324
//9.01978
//5.29297
//9.84436

MHL_ShowNumber (metric,"Значение метрики Минковского", "metric");
//Значение метрики Минковского:
//metric=9.8044

delete [] x;
delete [] y;
```

## 7.13 Непараметрика

### 7.13.1 MHL\_BellShapedKernelExp

Колоколообразное экспоненциальное ядро.

Код 396. Синтаксис

```
double MHL_BellShapedKernelExp(double z);
```

**Входные параметры:**

$z$  — входная переменная.

**Возвращаемое значение:**

Значение функции в точке.

**Формула:**

$$f(z) = \begin{cases} 0.05968 (e^z + e^{-z}) - 0.154293z^2 + 0.2311, & \text{если } |z| \leq 2.47638181818; \\ 0, & \text{иначе.} \end{cases}$$

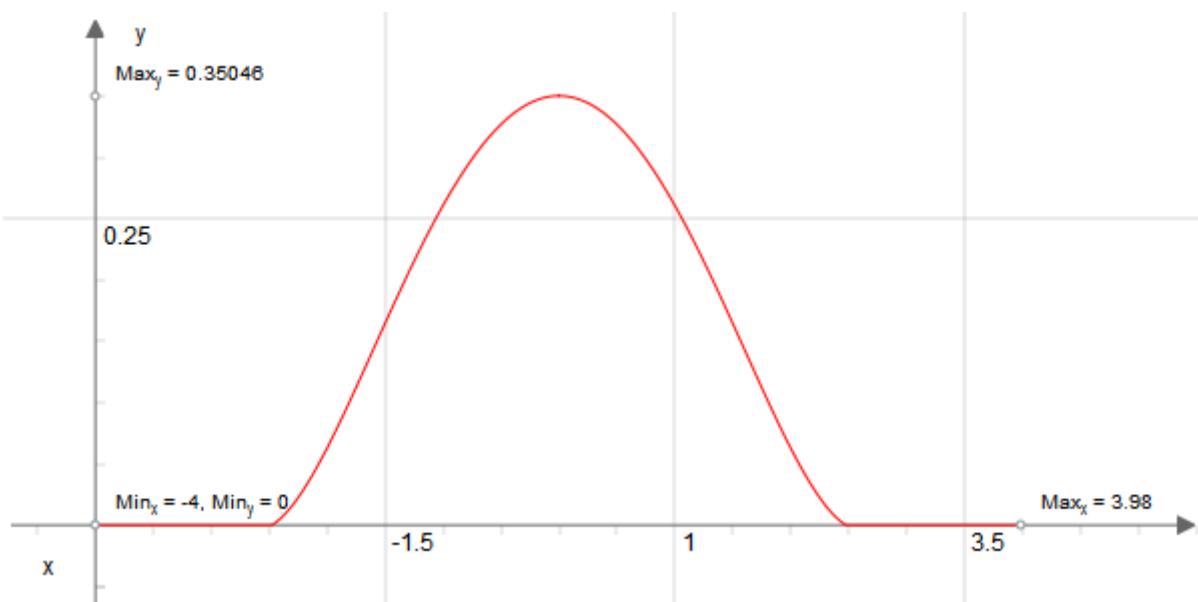


Рисунок 20. График функции

Код 397. Пример использования

```
double z=MHL_RandomUniform(-5, 5);

//Вызов функции
double f=MHL_BellShapedKernelExp(z);

//Используем полученный результат
MHL>ShowNumber(z, "Значение параметра", "z");
//Значение параметра:
//z=0.721436
MHL>ShowNumber(f, "Значение колоколообразного экспоненциального ядра", "f");
//Значение колоколообразного экспоненциального ядра:
//f=0.302588
```

### 7.13.2 MHL\_BellShapedKernelParabola

Колоколообразное параболическое ядро.

Код 398. Синтаксис

```
double MHL_BellShapedKernelParabola(double z);
```

**Входные параметры:**

$z$  — входная переменная.

**Возвращаемое значение:**

Значение функции в точке.

**Формула:**

$$f(z) = \begin{cases} 0.335 - 0.067z^2, & \text{если } z^2 \leq 5; \\ 0, & \text{иначе.} \end{cases}$$



Рисунок 21. График функции

Код 399. Пример использования

```
double z=MHL_RandomUniform(-5,5);

//Вызов функции
double f=MHL_BellShapedKernelParabola(z);

//Используем полученный результат
MHL>ShowNumber(z, "Значение параметра", "z");
//Значение параметра:
//z=0.33905
MHL>ShowNumber(f, "Значение колоколообразного параболического ядра", "f");
//Значение колоколообразного параболического ядра:
//f=0.327298
```

### 7.13.3 MHL\_BellShapedKernelRectangle

Колоколообразное прямоугольное ядро.

Код 400. Синтаксис

```
double MHL_BellShapedKernelRectangle(double z);
```

**Входные параметры:**

$z$  — входная переменная.

**Возвращаемое значение:**

Значение функции в точке.

**Формула:**

$$f(z) = \begin{cases} 0.5, & \text{если } |z| \leq 1; \\ 0, & \text{иначе.} \end{cases}$$

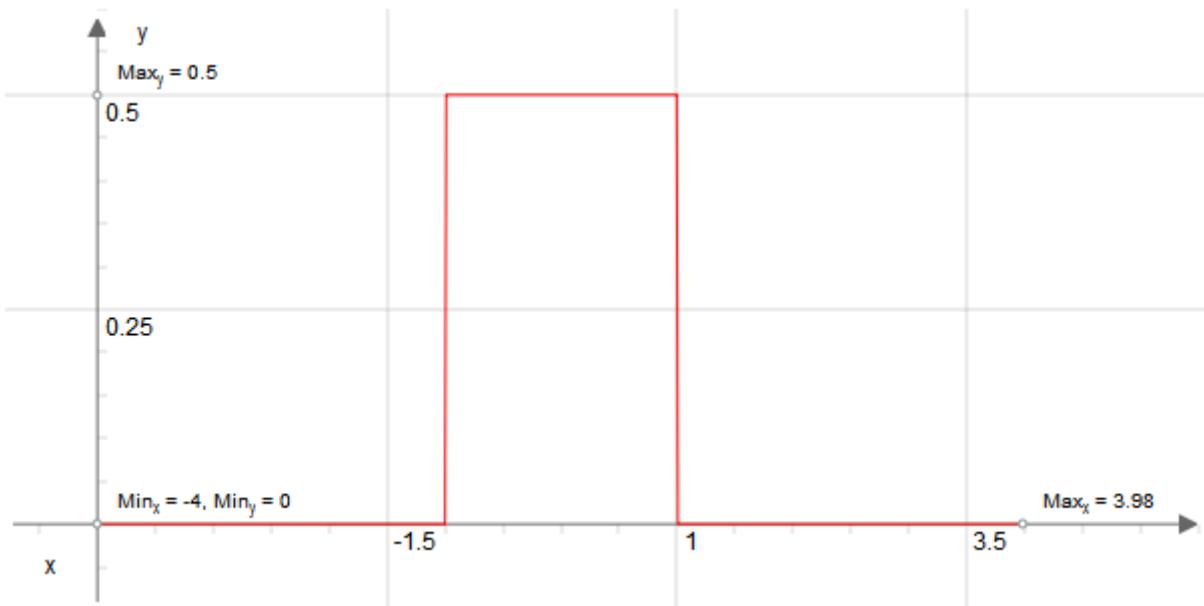


Рисунок 22. График функции

Код 401. Пример использования

```
double z=MHL_RandomUniform(-5, 5);

//Вызов функции
double f=MHL_BellShapedKernelRectangle(z);

//Используем полученный результат
MHL_ShowNumber(z, "Значение параметра", "z");
//Значение параметра:
//z=0.669556
MHL_ShowNumber(f, "Значение колоколообразного прямоугольного ядра", "f");
//Значение колоколообразного прямоугольного ядра:
//f=0.5
```

#### 7.13.4 MHL\_BellShapedKernelTriangle

Колоколообразное треугольное ядро.

Код 402. Синтаксис

```
double MHL_BellShapedKernelTriangle(double z);
```

**Входные параметры:**

$z$  — входная переменная.

**Возвращаемое значение:**

Значение функции в точке.

**Формула:**

$$f(z) = \begin{cases} 1 - z, & \text{если } |z| \leq 1; \\ 0, & \text{иначе.} \end{cases}$$

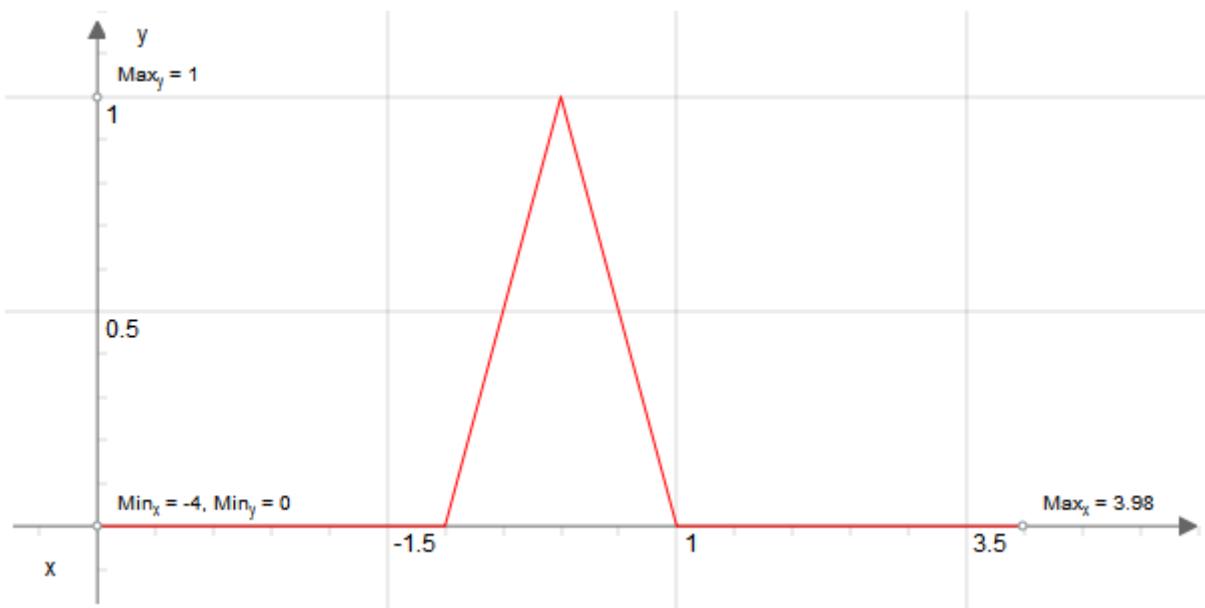


Рисунок 23. График функции

Код 403. Пример использования

```
double z=MHL_RandomUniform(-5,5);

//Вызов функции
double f=MHL_BellShapedKernelTriangle(z);

//Используем полученный результат
MHL_ShowNumber(z, "Значение параметра", "z");
//Значение параметра:
//z=0.362854
MHL_ShowNumber(f, "Значение колоколообразного треугольного ядра", "f");
//Значение колоколообразного треугольного ядра:
//f=0.637146
```

### 7.13.5 MHL\_DerivativeOfBellShapedKernelExp

Производная колоколообразного экспоненциального ядра.

Код 404. Синтаксис

```
double MHL_DerivativeOfBellShapedKernelExp(double z);
```

**Входные параметры:**

$z$  — входная переменная.

**Возвращаемое значение:**

Значение функции в точке.

**Формула:**

$$f(z) = \begin{cases} 0.05968 (e^z + e^{-z}) - 0.308586z, & \text{если } |z| \leq 2.47638181818; \\ 0, & \text{иначе.} \end{cases}$$

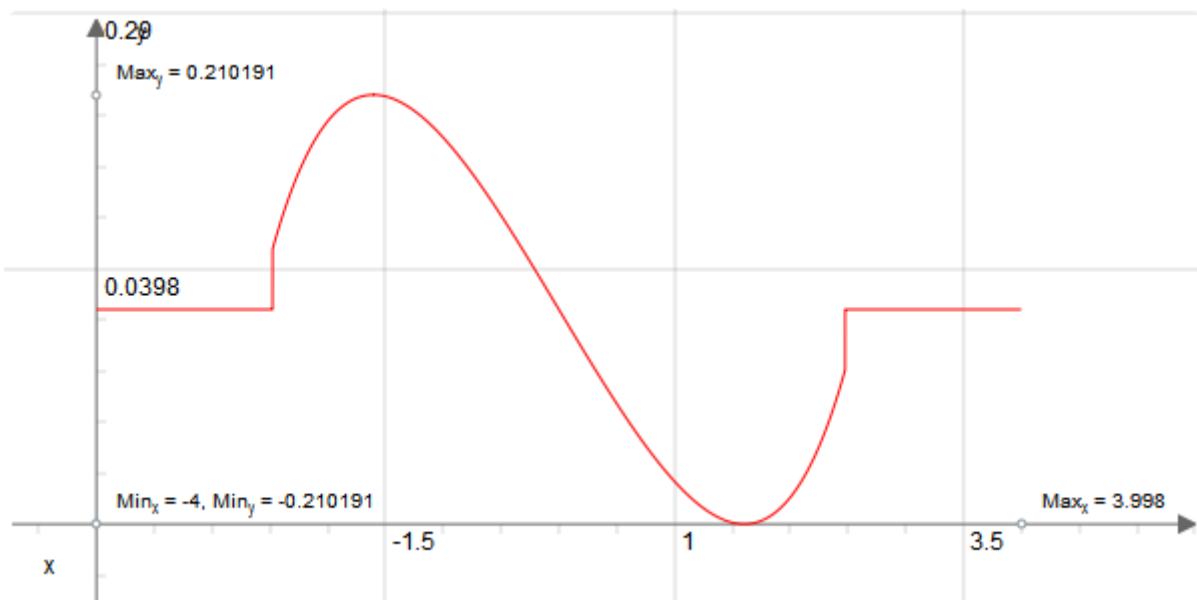


Рисунок 24. График функции

Код 405. Пример использования

```
double z=MHL_RandomUniform(-4, 4);

//Вызов функции
double f=MHL_DerivativeOfBellShapedKernelExp(z);

//Используем полученный результат
MHL>ShowNumber(z, "Значение параметра", "z");
//Значение параметра:
//z=-1.93701
MHL>ShowNumber(f, "Значение производной колоколообразного экспоненциального ядра", "f");
//Значение производной колоколообразного экспоненциального ядра:
//f=0.192278
```

### 7.13.6 MHL\_DerivativeOfBellShapedKernelParabola

Производная колоколообразного параболического ядра.

Код 406. Синтаксис

```
double MHL_DerivativeOfBellShapedKernelParabola(double z);
```

**Входные параметры:**

$z$  — входная переменная.

**Возвращаемое значение:**

Значение функции в точке.

**Формула:**

$$f(z) = \begin{cases} -0.134z, & \text{если } z^2 \leq 5; \\ 0, & \text{иначе.} \end{cases}$$

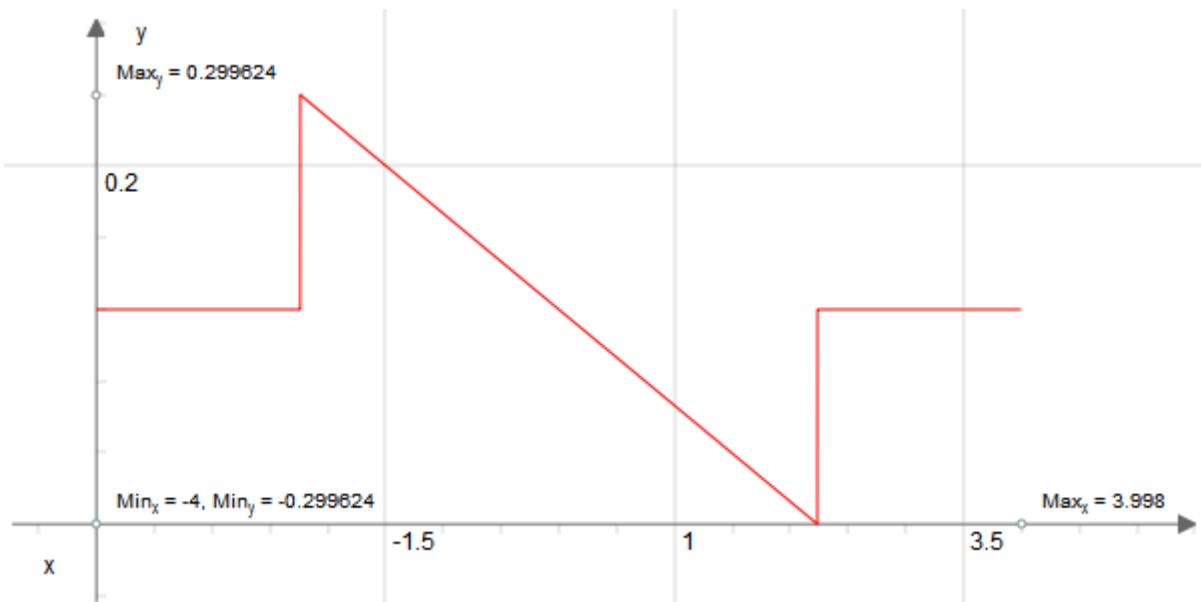


Рисунок 25. График функции

Код 407. Пример использования

```
double z=MHL_RandomUniform(-4, 4);

//Вызов функции
double f=MHL_DerivativeOfBellShapedKernelParabola(z);

//Используем полученный результат
MHL_ShowNumber(z, "Значение параметра", "z");
//Значение параметра:
//z=1.28394
MHL_ShowNumber(f, "Значение производной колоколообразного параболического ядра", "f");
//Значение производной колоколообразного параболического ядра:
//f=-0.172047
```

### 7.13.7 MHL\_DerivativeOfBellShapedKernelRectangle

Производная колоколообразного прямоугольного ядра.

Код 408. Синтаксис

```
double MHL_DerivativeOfBellShapedKernelRectangle(double z);
```

**Входные параметры:**

$z$  — входная переменная.

**Возвращаемое значение:**

Значение функции в точке.

**Формула:**

$$f(z) = \begin{cases} -\infty, & \text{если } z = 1; \\ \infty, & \text{если } z = -1; \\ 0, & \text{иначе.} \end{cases}$$

Код 409. Пример использования

```
double z=MHL_RandomUniform(-4, 4);

//Вызов функции
double f=MHL_DerivativeOfBellShapedKernelRectangle(z);

//Используем полученный результат
MHL_ShowNumber(z, "Значение параметра", "z");
//Значение параметра:
//z=3.146
MHL_ShowNumber(f, "Значение производной колоколообразного прямоугольного ядра", "f");
//Значение производной колоколообразного прямоугольного ядра:
//f=0
```

### 7.13.8 MHL\_DerivativeOfBellShapedKernelTriangle

Производная колоколообразного треугольного ядра.

Код 410. Синтаксис

```
double MHL_DerivativeOfBellShapedKernelTriangle(double z);
```

**Входные параметры:**

$z$  — входная переменная.

**Возвращаемое значение:**

Значение функции в точке.

**Формула:**

$$f(z) = \begin{cases} -1, & \text{если } z \in [0; 1]; \\ 1, & \text{если } z \in [-1; 0); \\ 0, & \text{иначе.} \end{cases}$$

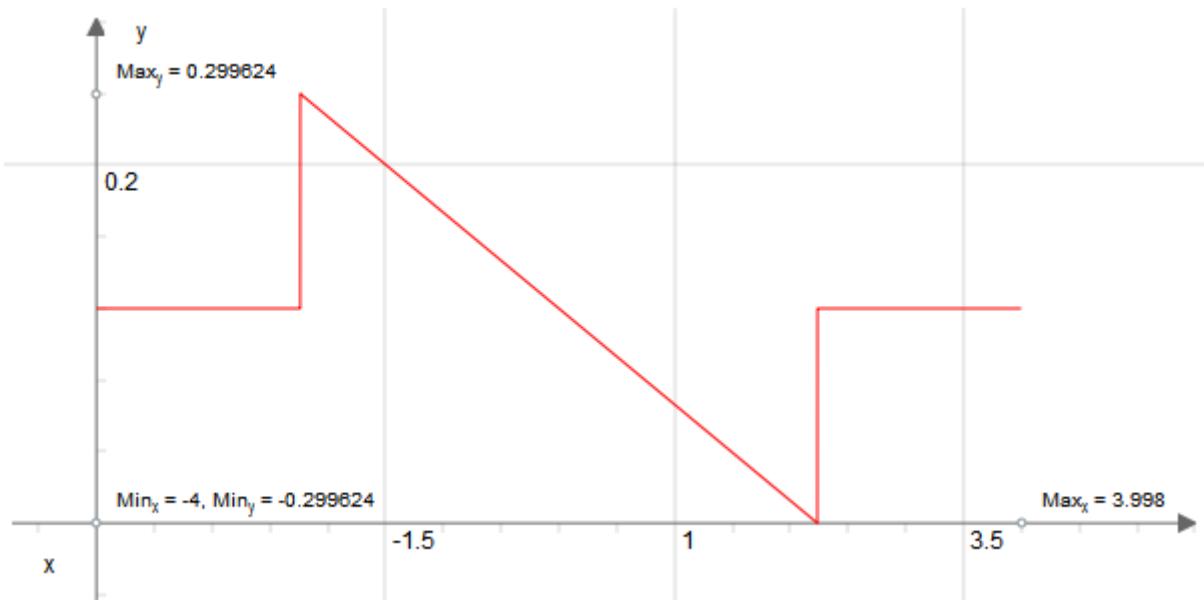


Рисунок 26. График функции

Код 411. Пример использования

```
double z=MHL_RandomUniform(-4, 4);

//Вызов функции
double f=MHL_DerivativeOfBellShapedKernelTriangle(z);

//Используем полученный результат
MHL_ShowNumber(z,"Значение параметра","z");
//Значение параметра:
//z=0.365479
MHL_ShowNumber(f,"Значение производной колоколообразного треугольного ядра","f");
//Значение производной колоколообразного треугольного ядра:
//f=-1
```

## 7.14 Нечеткие системы

### 7.14.1 MHL\_TrapeziformFuzzyNumber

Трапециевидное нечеткое число. Точнее его функция принадлежности.

Код 412. Синтаксис

```
double MHL_TrapeziformFuzzyNumber(double x,double a,double b,double c,double d);
```

**Входные параметры:**

$x$  — действительное число, для которого считаем функцию принадлежности.

$a$  — левая крайняя граница;

$b$  — начало устойчивой области;

$c$  — конец устойчивой области;

$d$  — правая крайняя граница.

### Возвращаемое значение:

Значение функции принадлежности.

### Формула:

$$f(x) = \begin{cases} 0, & \text{если } x < a; \\ \frac{x-a}{b-a}, & \text{если } x \in [a; b); \\ 1, & \text{если } x \in [b; c]; \\ \frac{d-x}{d-c}, & \text{если } x \in (c; d]; \\ 0, & \text{если } x > d. \end{cases}$$

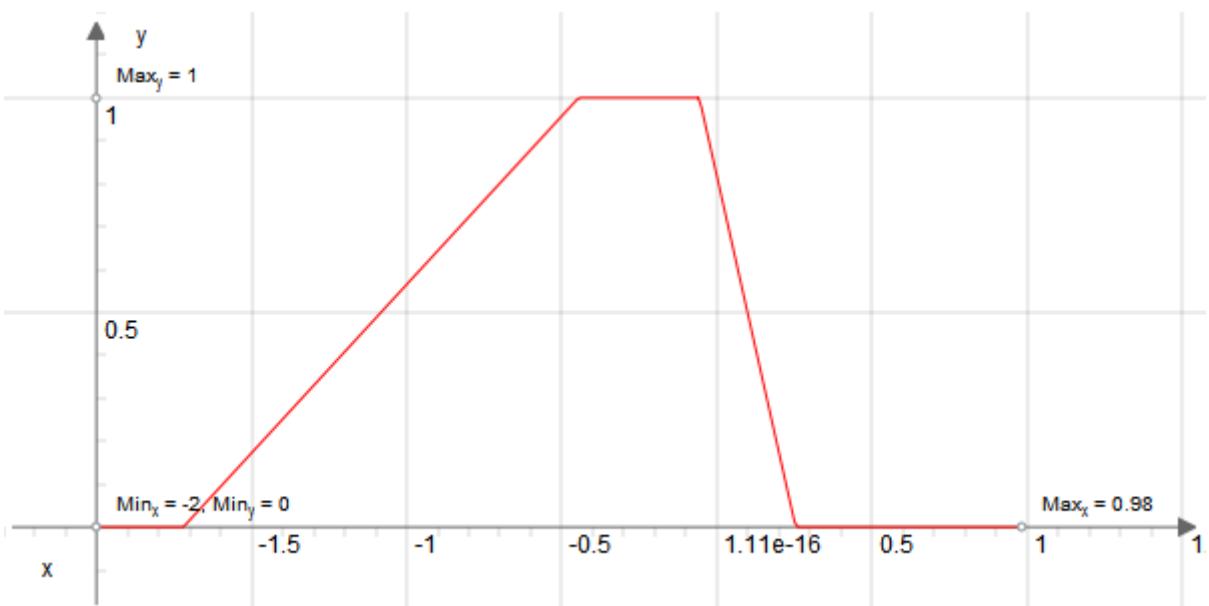


Рисунок 27. График функции

Код 413. Пример использования

```
double a=MHL_RandomUniform(-4, 4);
double b=a+MHL_RandomUniform(0, 2);
double c=b+MHL_RandomUniform(0, 2);
double d=c+MHL_RandomUniform(0, 2);

double x=MHL_RandomUniform(a-1, d+1);

//Вызов функции
double f=MHL_TrapeziformFuzzyNumber (x,a,b,c,d);

//Используем полученный результат
MHL_ShowNumber(x, "Значение параметра", "x");
//Значение параметра:
//x=-0.932339
MHL_ShowNumber(a, "Значение первого параметра трапециевидного нечеткого числа", "a");
//Значение первого параметра трапециевидного нечеткого числа:
//a=-1.71997
MHL_ShowNumber(b, "Значение второго параметра трапециевидного нечеткого числа", "b");
//Значение второго параметра трапециевидного нечеткого числа:
//b=-0.446045
MHL_ShowNumber(c, "Значение третьего параметра трапециевидного нечеткого числа", "c");
```

```

//Значение третьего параметра трапециевидного нечеткого числа:
//c=-0.0568848
MHL_ShowNumber(d, "Значение последнего параметра трапециевидного нечеткого числа", "d");
//Значение последнего параметра трапециевидного нечеткого числа:
//d=0.253784
MHL_ShowNumber(f, "Значение функция принадлежности трапециевидного нечеткого числа", "f
");
//Значение функция принадлежности трапециевидного нечеткого числа:
//f=0.618271

```

## 7.15 Оптимизация

### 7.15.1 MHL\_BinaryMonteCarloAlgorithm

Метод Монте-Карло (Monte-Carlo). Простейший метод оптимизации на бинарных строках. В простонародье его называют «методом научного тыка». Алгоритм оптимизации. Ищет максимум функции пригодности FitnessFunction.

Код 414. Синтаксис

```

int MHL_BinaryMonteCarloAlgorithm(int *Parameters, double (*FitnessFunction)(int*, int
), int *VMHL_ResultVector, double *VMHL_Result);

```

#### Входные параметры:

Parameters:

- 0 — длина бинарной строки (определяется задачей оптимизации, что мы решаем);
- 1 — число вычислений функции пригодности (CountOfFitness);

FitnessFunction — указатель на целевую функцию (если решается задача условной оптимизации, то учет ограничений должен быть включен в эту функцию);

VMHL\_ResultVector — найденное решение (бинарный вектор);

VMHL\_Result — значение функции в точке, определенной вектором VMHL\_ResultVector.

#### Возвращаемое значение:

1 — завершил работу без ошибок. Всё хорошо.

0 — возникли при работе ошибки. Скорее всего в этом случае в VMHL\_ResultVector и в VMHL\_Result не содержится решение задачи.

#### Пример значений рабочего вектора Parameters:

Parameters[0]=20;

Parameters[1]=100\*100;

#### Принцип работы:

Принцип прост. Берутся случайно CountOfFitness решений независимо друг от друга. Выбирается лучшее. Всё.

#### О функции:

В простонародье алгоритм называют «методом научного тыка».

Алгоритм оптимизации. Ищет максимум функции пригодности FitnessFunction.

Решением является бинарная строка, то есть вектор, состоящий из 0 и 1.

## Код 415. Оптимизируемая функция

```
double Func(int *x, int VMHL_N)
{
    //Сумма всех элементов массива
    return TMHL_SumVector(x, VMHL_N);
}
//-----
```

## Код 416. Пример использования

```

int LengthBinarString=50;//Длина хромосомы
int CountOfFitness=50*50;//Число вычислений функции пригодности

int *ParametersOfBinaryMonteCarloAlgorithm;
ParametersOfBinaryMonteCarloAlgorithm=new int[2];
ParametersOfBinaryMonteCarloAlgorithm[0]=LengthBinarString;//Длина хромосомы
ParametersOfBinaryMonteCarloAlgorithm[1]=CountOfFitness;//Число вычислений целевой функции

int *Decision;//бинарное решение
Decision=new int[LengthBinarString];
double ValueOfFitnessFunction;//значение функции пригодности в точке Decision
int VMHL_Success=0;//Успешен ли будет запуск СГА

//Запуск алгоритма
VMHL_Success=MHL_BinaryMonteCarloAlgorithm (ParametersOfBinaryMonteCarloAlgorithm,
    Func, Decision, &ValueOfFitnessFunction);

//Используем полученный результат
MHL_ShowNumber(VMHL_Success, "Как прошел запуск", "VMHL_Success");
//Как прошел запуск:
//VMHL_Success=1

if (VMHL_Success==1)
{
    MHL_ShowVectorT(Decision,LengthBinarString, "Найденное решение", "Decision");
    //Найденное решение:
    //Decision =
//1 0 1 1 1 1 0 1 1 1 1 1 1 1 1 0 0 1 0 0 1 1 1 1 0 1 1 1 1 1 1 1 1 0 1 1
0 1 1 0 1 1 1 0 0 1 1 1 1 1 0 0 1 1 1 0 1 1 1 1 0 1 1 1 1 1 1 1 1 0 1 1

    MHL_ShowNumber(ValueOfFitnessFunction, "Значение функции пригодности",
        "ValueOfFitnessFunction");
    // Значение функции пригодности:
    //ValueOfFitnessFunction=37
}
delete [] ParametersOfBinaryMonteCarloAlgorithm;
delete [] Decision;

```

## 7.15.2 MHL\_DichotomyOptimization

Метод дихотомии. Метод одномерной оптимизации унимодальной функции на интервале. Ищет минимум.

Код 417. Синтаксис

```
void MHL_DichotomyOptimization (double Left, double Right, double (*Function)(double)
, double Interval, double Epsilon, double *VMHL_Result_X,double *VMHL_Result_Y);
```

**Входные параметры:**

Left — начало интервала поиска;

Right — конец интервала поиска;

Function — унимодальная функция, минимум которой ищется;

Interval — длина конечного интервала неопределенности (точность поиска);

Epsilon — малое число;

VMHL\_Result\_X — вычисленная точка минимума (сюда записывается результат);

VMHL\_Result\_Y — значение функции в точке минимума (сюда записывается результат).

**Возвращаемое значение:**

Отсутствует.

**Примечание:** Epsilon должен быть меньше, чем Interval/2, иначе возвращается левая граница просто.

Код 418. Оптимизируемая функция

```
double Func4(double x)
{
    return (x-1)*(x-1);
}
//-----
```

Код 419. Пример использования

```
double Left=-5;
double Right=5;
double Interval=0.01;
double Epsilon=0.001;
double x,f;

//Вызов функции
MHL_DichotomyOptimization (Left, Right, Func4, Interval, Epsilon, &x, &f);

//Используем полученный результат
MHL_ShowNumber(x,"Найденное решение", "x");
//Найденное решение:
//x=0.998335

MHL_ShowNumber(f,"Значение целевой функции в найденном решении", "f");
//Значение целевой функции в найденном решении:
//f=2.77073e-06
```

### 7.15.3 MHL\_FibonacciOptimization

Метод Фибоначчи. Метод одномерной оптимизации унимодальной функции на интервале. Ищет минимум.

Код 420. Синтаксис

```
void MHL_FibonacciOptimization (double Left, double Right, double (*Function)(double), int Count, double *VMHL_Result_X, double *VMHL_Result_Y);
```

**Входные параметры:** Left — начало интервала поиска;

Right — конец интервала поиска;

Function — унимодальная функция, минимум которой ищется;

Count — число вычислений целевой функции (не делайти сильно большим, а то будет переполнение);

VMHL\_Result\_X — вычисленная точка минимума (сюда записывается результат);

VMHL\_Result\_Y — значение функции в точке минимума (сюда записывается результат).

**Возвращаемое значение:**

Отсутствует.

Код 421. Оптимизируемая функция

```
double Func4(double x)
{
    return (x-1)*(x-1);
}
//-----
```

Код 422. Пример использования

```
double Left=-5;
double Right=5;
int Count=30;
double x,f;

//Вызов функции
MHL_FibonacciOptimization (Left, Right, Func4, Count, &x, &f);

//Используем полученный результат
MHL_ShowNumber(x,"Найденное решение", "x");
//Найденное решение:
//x=1

MHL_ShowNumber(f,"Значение целевой функции в найденном решении", "f");
//Значение целевой функции в найденном решении:
//f=3.7817e-12
```

### 7.15.4 MHL\_GoldenSectionOptimization

Метод золотого сечения. Метод одномерной оптимизации унимодальной функции на интервале. Ищет минимум.

Код 423. Синтаксис

```
void MHL_GoldenSectionOptimization (double Left, double Right, double (*Function)(  
    double), double Interval, double *VMHL_Result_X,double *VMHL_Result_Y);
```

**Входные параметры:**

Left — начало интервала поиска;

Right — конец интервала поиска;

Function — унимодальная функция, минимум которой ищется;

Interval — длина конечного интервала неопределенности (точность поиска);

VMHL\_Result\_X — вычисленная точка минимума (сюда записывается результат);

VMHL\_Result\_Y — значение функции в точке минимума (сюда записывается результат).

**Возвращаемое значение:**

Отсутствует.

Код 424. Оптимизируемая функция

```
double Func4(double x)  
{  
    return (x-1)*(x-1);  
}  
//-----
```

Код 425. Пример использования

```
double Left=-5;  
double Right=5;  
double Interval=0.001;  
double x,f;  
  
//Вызов функции  
MHL_GoldenSectionOptimization (Left, Right, Func4, Interval, &x, &f);  
  
//Используем полученный результат  
MHL_ShowNumber(x,"Найденное решение","x");  
//Найденное решение:  
//x=0.999934  
  
MHL_ShowNumber(f,"Значение целевой функции в найденном решении","f");  
//Значение целевой функции в найденном решении:  
//f=4.37013e-09
```

### 7.15.5 MHL\_QuadraticFitOptimization

Метод квадратичной интерполяции. Метод одномерной оптимизации унимодальной функции на интервале. Ищет минимум.

Код 426. Синтаксис

```
void MHL_QuadraticFitOptimization (double Left, double Right, double (*Function)(  
    double), double Epsilon, double Epsilon2, double *VMHL_Result_X,double *  
    VMHL_Result_Y);
```

### **Входные параметры:**

Left — начало интервала поиска;

Right — конец интервала поиска;

Function — унимодальная функция, минимум которой ищется;

Epsilon — точность;

Epsilon2 — шаг, фактически еще одно малое число;

VMHL\_Result\_X — вычисленная точка минимума (сюда записывается результат);

VMHL\_Result\_Y — значение функции в точке минимума (сюда записывается результат).

### **Возвращаемое значение:**

Отсутствует.

#### Код 427. Оптимизируемая функция

```
double Func4(double x)
{
    return (x-1)*(x-1);
}
//-----
```

#### Код 428. Пример использования

```
double Left=-5;
double Right=5;
double Epsilon=0.001;
double Epsilon2=0.001;
double x,f;

//Вызов функции
MHL_QuadraticFitOptimization (Left, Right, Func4, Epsilon,Epsilon2, &x, &f);

//Используем полученный результат
MHL_ShowNumber(x,"Найденное решение","x");
//Найденное решение:
//x=1

MHL_ShowNumber(f,"Значение целевой функции в найденном решении","f");
//Значение целевой функции в найденном решении:
//f=0
```

## **7.15.6 MHL\_RealMonteCarloAlgorithm**

Метод Монте-Карло (Monte-Carlo). Простейший метод оптимизации на вещественных строках. В простонародье его называют "методом научного тыка". Алгоритм оптимизации. Ищет максимум функции пригодности FitnessFunction.

#### Код 429. Синтаксис

```
int MHL_RealMonteCarloAlgorithm(int *Parameters, double *Left, double *Right, double
(*FitnessFunction)(double*,int), double *VMHL_ResultVector, double *VMHL_Result);
```

## **Входные параметры:**

Parameters:

0 — длина вещественной строки (определяется задачей оптимизации, что мы решаем);

1 — число вычислений функции пригодности (CountOfFitness);

Left — массив левых границ изменения каждой вещественной координаты (размерность Parameters[0]);

Right — массив правых границ изменения каждой вещественной координаты (размерность Parameters[0]);

FitnessFunction — указатель на целевую функцию (если решается задача условной оптимизации, то учет ограничений должен быть включен в эту функцию);

VMHL\_ResultVector — найденное решение (вещественный вектор);

VMHL\_Result — значение функции в точке, определенной вектором VMHL\_ResultVector.

## **Возвращаемое значение:**

1 — завершил работу без ошибок. Всё хорошо.

0 — возникли при работе ошибки. Скорее всего в этом случае в VMHL\_ResultVector и в VMHL\_Result не содержится решение задачи.

## **Пример значений рабочего вектора Parameters:**

Parameters[0]=5;

Parameters[1]=100\*100;

## **Принцип работы:**

Принцип прост. Берутся случайно CountOfFitness решений независимо друг от друга. Выбирается лучшее. Всё.

## **О функции:**

В простонародье алгоритм называют «методом научного тыка».

Алгоритм оптимизации. Ищет максимум функции пригодности FitnessFunction.

Решением является вещественная строка, то есть вектор, состоящий из 0 и 1.

### Код 430. Оптимизируемая функция

```
double Func(double *x, int VMHL_N)
{
    return -((x[0]-2)*(x[0]-2)+(x[1]-2)*(x[1]-2));
}
//-----
```

### Код 431. Пример использования

```
int ChromosomeLength=2; //Длина хромосомы
int CountOfFitness=50*50; //Число вычислений целевой функции

int *ParametersOfAlgorithm;
```

```

ParametersOfAlgorithm=new int[2];
ParametersOfAlgorithm[0]=ChromosomeLength; //Длина хромосомы
ParametersOfAlgorithm[1]=CountOffitness;//Число вычислений целевой функции

double *Left; //массив левых границ изменения каждой вещественной координаты
Left=new double[ChromosomeLength];
double *Right; //массив правых границ изменения каждой вещественной координаты
Right=new double[ChromosomeLength];

//Заполним массивы
//Причем по каждой координате одинаковые значения выставим
TMHL_FillVector(Left,ChromosomeLength,-5.); //Пусть будет интервал [-3;3]
TMHL_FillVector(Right,ChromosomeLength,5.);

double *Decision; //вещественное решение
Decision=new double[ChromosomeLength];
double ValueOfFitnessFunction; //значение целевой функции в точке Decision
int VMHL_Success=0; //Успешен ли будет запуск СГА

//Запуск алгоритма
VMHL_Success=MHL_RealMonteCarloAlgorithm (ParametersOfAlgorithm,Left,Right,Func2,
    Decision, &ValueOfFitnessFunction);

//Используем полученный результат
MHL_ShowNumber(VMHL_Success,"Как прошел запуск","VMHL_Success");
if (VMHL_Success==1)
{
    MHL_ShowVectorT(Decision,ChromosomeLength,"Найденное решение","Decision");
    //Найденное решение:
    //Decision =
    //1.91864 1.93604
    MHL_ShowNumber(ValueOfFitnessFunction,"Значение целевой функции",
        "ValueOfFitnessFunction");
    //Значение целевой функции:
    //ValueOfFitnessFunction=-0.0107109
}

delete [] ParametersOfAlgorithm;
delete [] Decision;
delete [] Left;
delete [] Right;

```

### 7.15.7 MHL\_RealMonteCarloOptimization

Метод Монте-Карло (Monte-Carlo). Простейший метод оптимизации на вещественных строках. Ищет минимум. От функции MHL\_RealMonteCarloAlgorithm отличается тем, что ищет минимум, а не максимум, и не у многомерной функции, а одномерной. Вводится, чтобы было продолжением однотипных методов оптимизации одномерных унимодальных функций.

Код 432. Синтаксис

```

void MHL_RealMonteCarloOptimization (double Left, double Right, double (*Function)(
    double), int Count, double *VMHL_Result_X,double *VMHL_Result_Y);

```

**Входные параметры:**

Left — начало интервала поиска

Right — конец интервала поиска

Function — унимодальная функция, минимум которой ищется

Count — число вычислений целевой функции

VMHL\_Result\_X — вычисленная точка минимума (сюда записывается результат);

VMHL\_Result\_Y — значение функции в точке минимума (сюда записывается результат).

#### **Возвращаемое значение:**

Отсутствует.

Код 433. Оптимизируемая функция

```
double Func4(double x)
{
    return (x-1)*(x-1);
} //-----
```

Код 434. Пример использования

```
double Left=-5;
double Right=5;
int Count=30;
double x,f;

//Вызов функции
MHL_RealMonteCarloOptimization (Left, Right, Func4, Count, &x, &f);

//Используем полученный результат
MHL_ShowNumber(x,"Найденное решение","x");
//Найденное решение:
//x=1.11359

MHL_ShowNumber(f,"Значение целевой функции в найденном решении","f");
//Значение целевой функции в найденном решении:
//f=0.0129019
```

### **7.15.8 MHL\_UniformSearchOptimization**

Метод равномерного поиска. Метод одномерной оптимизации функции на интервале. Ищет минимум.

Код 435. Синтаксис

```
void MHL_UniformSearchOptimization (double Left, double Right, double (*Function)(
    double), double Interval, double *VMHL_Result_X,double *VMHL_Result_Y);
```

#### **Входные параметры:**

Left — начало интервала поиска;

Right — конец интервала поиска;

Function — унимодальная функция, минимум которой ищется;

Interval — длина шага, с которым будет проводится поиск;

VMHL\_Result\_X — вычисленная точка минимума (сюда записывается результат);

VMHL\_Result\_Y — значение функции в точке минимума (сюда записывается результат).

#### **Возвращаемое значение:**

Отсутствует.

#### Код 436. Оптимизируемая функция

```
double Func4(double x)
{
    return (x-1)*(x-1);
}
//-----
```

#### Код 437. Пример использования

```
double Left=-5;
double Right=5;
double Interval=0.001;
double x,f;

//Вызов функции
MHL_UniformSearchOptimization (Left, Right, Func4, Interval, &x, &f);

//Используем полученный результат
MHL_ShowNumber(x,"Найденное решение","x");
//Найденное решение:
//x=1

MHL_ShowNumber(f,"Значение целевой функции в найденном решении","f");
//Значение целевой функции в найденном решении:
//f=2.3863e-29
```

### **7.15.9 MHL\_UniformSearchOptimizationN**

Метод равномерного поиска. Метод одномерной оптимизации функции на интервале. Ищет минимум. От MHL\_UniformSearchOptimization отличается тем, что вместо параметра шага равномерного прохода используется число вычислений целевой функции, но они взаимозаменяемы.

#### Код 438. Синтаксис

```
void MHL_UniformSearchOptimizationN (double Left, double Right, double (*Function)(
    double), int Count, double *VMHL_Result_X,double *VMHL_Result_Y);
```

#### **Входные параметры:**

Left — начало интервала поиска;

Right — конец интервала поиска;

Function — унимодальная функция, минимум которой ищется;

Count — число вычислений целевой функции;

VMHL\_Result\_X — вычисленная точка минимума (сюда записывается результат);  
VMHL\_Result\_Y — значение функции в точке минимума (сюда записывается результат).

#### **Возвращаемое значение:**

Отсутствует.

#### Код 439. Оптимизируемая функция

```
double Func4(double x)
{
    return (x-1)*(x-1);
}
//-----
```

#### Код 440. Пример использования

```
double Left=-5;
double Right=5;
int Count=30;
double x,f;

//Вызов функции
MHL_UniformSearchOptimizationN (Left, Right, Func4, Count, &x, &f);

//Используем полученный результат
MHL_ShowNumber(x,"Найденное решение","x");
//Найденное решение:
//x=1

MHL_ShowNumber(f,"Значение целевой функции в найденном решении","f");
//Значение целевой функции в найденном решении:
//f=0
```

## 7.16 Оптимизация - свалка алгоритмов

### 7.16.1 MHL\_BinaryGeneticAlgorithmWCC

Генетический алгоритм для решения задач на бинарных строках, в котором есть только два вида скрещивания: одноточечное и двухточечное скрещивание с возможностью полного копирования одного из родителей. Равномерное скрещивание отсутствует.

#### Код 441. Синтаксис

```
int MHL_BinaryGeneticAlgorithmWCC(int *Parameters, double (*FitnessFunction)(int*, int ),
    int *VMHL_ResultVector, double *VMHL_Result);
```

#### **Входные параметры:**

Parameters — Вектор параметров генетического алгоритма. Каждый элемент обозначает свой параметр:

- 0 — длина бинарной хромосомы (определяется задачей оптимизации, что мы решаем);
- 1 — число вычислений целевой функции (CountOfFitness);

2 — тип селекции (TypeOfSel):

- 0 — ProportionalSelection (Пропорциональная селекция);
- 1 — RankSelection (Ранговая селекция);
- 2 — TournamentSelection (Турнирная селекция).

3 — тип скрещивания (TypeOfCros):

- 0 — SinglepointCrossoverWithCopying (Одноточечное скрещивание с возможностью полного копирования одного из родителей);
- 1 — TwopointCrossoverWithCopying (Двухточечное скрещивание с возможностью полного копирования одного из родителей).

4 — тип мутации (TypeOfMutation):

- 0 — Weak (Слабая мутация);
- 1 — Average (Средняя мутация);
- 2 — Strong (Сильная мутация).

5 — тип формирования нового поколения (TypeOfForm):

- 0 — OnlyOffspringGenerationForming (Только потомки);
- 1 — OnlyOffspringWithBestGenerationForming (Только потомки и копия лучшего индивида).

FitnessFunction — указатель на целевую функцию (если решается задача условной оптимизации, то учет ограничений должен быть включен в эту функцию);

VMHL\_ResultVector — найденное решение (бинарный вектор);

VMHL\_Result — значение целевой функции в точке, определенной вектором VMHL\_ResultVector.

#### **Возвращаемое значение:**

1 — завершил работу без ошибок. Всё хорошо.

0 — возникли при работе ошибки. Скорее всего в этом случае в VMHL\_ResultVector и в VMHL\_Result не содержится решение задачи.

#### **О функции:**

Отличается от стандартного генетического алгоритма тем, что есть только два вида скрещивания: одноточечное и двухточечное скрещивание с возможностью полного копирования одного из родителей. Равномерное скрещивание отсутствует. То есть данным алгоритмом проверяем: есть ли разница в эффективности алгоритма, если точки разрыва при скрещивании делать и по краям родителей, а не только внутри хромосомы.

Алгоритм бинарной оптимизации. Ищет максимум целевой функции FitnessFunction.

Решением является бинарная строка, то есть вектор, состоящий из 0 и 1.

Тип алгоритма: исследовательский алгоритм оптимизации.

Подробное описание алгоритма можно найти тут:

<https://github.com/Harrix/HarrixOptimizationAlgorithms>

**Примерный настройки** (для примера Вы можете поставить такие рабочие настройки):

```
Parameters[0]=50;  
Parameters[1]=100*100;  
Parameters[2]=2;  
Parameters[3]=1;  
Parameters[4]=1;  
Parameters[5]=1;
```

**Примечание:**

В сГА на бинарных строках не нужно задавать в параметрах число поколений и размер популяции, а только число вычислений целевой функции. Почему? Алгоритм сам определит число поколений и размер популяции, исходя из принципа, что число поколений и размер популяции должны быть примерно равны. Поэтому выбирайте значение Parameters[1] в виде:

```
int K=100;  
Parameters[1]=K*K;
```

То есть в виде квадрата целого числа. В противном случае реальное число вычислений целевой функции и значение Parameters[1] будут не совпадать.

Код целевой функции:

Код 442. Оптимизируемая функция

```
double Func(int *x,int VMHL_N)  
{  
//Сумма всех элементов массива  
return TMHL_SumVector(x,VMHL_N);  
}  
//-----
```

Код 443. Пример использования

```
int ChromosomeLength=50;//Длина хромосомы  
int CountOffitness=50*50;//Число вычислений целевой функции  
int TypeOfSel=1;//Тип селекции  
int TypeOfCros=0;//Тип скрещивания  
int TypeOfMutation=1;//Тип мутации  
int TypeOfForm=0;//Тип формирования нового поколения  
  
int *Parameters;  
Parameters=new int[6];  
Parameters[0]=ChromosomeLength;//Длина хромосомы  
Parameters[1]=CountOffitness;//Число вычислений целевой функции  
Parameters[2]=TypeOfSel;//Тип селекции  
Parameters[3]=TypeOfCros;//Тип скрещивания  
Parameters[4]=TypeOfMutation;//Тип мутации  
Parameters[5]=TypeOfForm;//Тип формирования нового поколения  
  
int *Decision;//бинарное решение  
Decision=new int[ChromosomeLength];  
double ValueOfFitnessFunction;//значение функции пригодности в точке Decision  
int VMHL_Success=0;//Успешен ли будет запуск сГА
```

### 7.16.2 MHL\_BinaryGeneticAlgorithm WDPOfNOfGPS

Генетический алгоритм для решения задач на бинарных строках с изменяющимся соотношением числа поколений и размера популяции. Отличается от стандартного генетического алгоритма, тем, что размер популяции и число поколений рассчитывается из числа вычислений целевой функции не как одинаковые величины (извлечение квадратного корня), а через некоторое соотношение.

## Код 444. Синтаксис

```
int MHL_BinaryGeneticAlgorithmWDPOfNOfGPS(double *Parameters, double (*FitnessFunction)(int*, int), int *VMHL_ResultVector, double *VMHL_Result);
```

## Входные параметры:

Parameters — Вектор параметров генетического алгоритма. Каждый элемент обозначает свой параметр:

0 — длина бинарной хромосомы (определяется задачей оптимизации, что мы решаем);

1 — число вычислений целевой функции (CountOfFitness);

2 – тип селекции (TypeOfSel):

- 0 — ProportionalSelection (Пропорциональная селекция);
  - 1 — RankSelection (Ранговая селекция);
  - 2 — TournamentSelection (Турнирная селекция).

3 — тип скрещивания (TypeOfCros):

- 0 – SinglepointCrossover (Одноточечное скрещивание);
- 1 – TwopointCrossover (Двухточечное скрещивание);
- 2 – UniformCrossover (Равномерное скрещивание).

4 — тип мутации (TypeOfMutation):

- 0 – Weak (Слабая мутация);
- 1 – Average (Средняя мутация);
- 2 – Strong (Сильная мутация).

5 — тип формирования нового поколения (TypeOfForm):

- 0 – OnlyOffspringGenerationForming (Только потомки);
- 1 – OnlyOffspringWithBestGenerationForming (Только потомки и копия лучшего индивида).

6 — «доля» (Proportion) числа поколений от общего числа вычислений целевой функции. Определяется как возвведение числа вычислений целевой функции в степень Proportion. Может принимать значения в интервале [0; 1]. Число поколений =  $\text{int}(\text{CountOfFitness}^{\text{Proportion}})$ . Размер популяции =  $\text{int}(\text{CountOfFitness}/\text{Число поколений})$ . При Proportion=0.5 получим обычный стандартный генетический алгоритм. Чем меньше Proportion, тем меньше число поколений будет. Желательно, чтобы принимались следующие значения:

- 0;
- 0.1;
- 0.2;
- 0.3;
- 0.4;
- 0.5;
- 0.6;
- 0.7;
- 0.8;
- 0.9;
- 1.

`FitnessFunction` — указатель на целевую функцию (если решается задача условной оптимизации, то учет ограничений должен быть включен в эту функцию);

`VMHL_ResultVector` — найденное решение (бинарный вектор);

`VMHL_Result` — значение целевой функции в точке, определенной вектором `VMHL_ResultVector`.

#### **Возвращаемое значение:**

1 — завершил работу без ошибок. Всё хорошо.

0 — возникли при работе ошибки. Скорее всего в этом случае в `VMHL_ResultVector` и в `VMHL_Result` не содержится решение задачи.

## О функции:

Отличается от стандартного генетического алгоритма, что число поколений может изменяться по описанному выше принципу.

Алгоритм бинарной оптимизации. Ищет максимум целевой функции FitnessFunction.

Решением является бинарная строка, то есть вектор, состоящий из 0 и 1.

Тип алгоритма: исследовательский алгоритм оптимизации.

Подробное описание алгоритма можно найти тут:

<https://github.com/Harrix/HarrixOptimizationAlgorithms>

**Примерный настройки** (для примера Вы можете поставить такие рабочие настройки):

Parameters[0]=50;

Parameters[1]=100\*100;

Parameters[2]=2;

Parameters[3]=2;

Parameters[4]=1;

Parameters[5]=1;

Parameters[6]=0.5;

Код целевой функции:

Код 445. Оптимизируемая функция

```
double Func(int *x,int VMHL_N)
{
//Сумма всех элементов массива
return TMHL_SumVector(x,VMHL_N);
}
//-----
```

Код 446. Пример использования

```
int ChromosomeLength=50; //Длина хромосомы
int CountOfFitness=50*50; //Число вычислений целевой функции
int TypeOfSel=1; //Тип селекции
int TypeOfCros=0; //Тип скрещивания
int TypeOfMutation=1; //Тип мутации
int TypeOfForm=0; //Тип формирования нового поколения
double Proportion=0.4; //Доля числа поколений от числа вычислений целевой функции

double *ParametersOfAlgorithm;
ParametersOfAlgorithm=new double[7];
ParametersOfAlgorithm[0]=ChromosomeLength; //Длина хромосомы
ParametersOfAlgorithm[1]=CountOfFitness; //Число вычислений целевой функции
ParametersOfAlgorithm[2]=TypeOfSel; //Тип селекции
ParametersOfAlgorithm[3]=TypeOfCros; //Тип скрещивания
ParametersOfAlgorithm[4]=TypeOfMutation; //Тип мутации
ParametersOfAlgorithm[5]=TypeOfForm; //Тип формирования нового поколения
ParametersOfAlgorithm[6]=Proportion; //Доля числа поколений от числа вычислений целевой функции
```

### 7.16.3 MHL\_BinaryGeneticAlgorithmWDTs

Генетический алгоритм для решения задач на бинарных строках с турнирной селекцией, где размер турнира изменяется от 2 до размера популяции. Отличается от стандартного генетического алгоритма, тем, что присутствует только турнирная селекция, но размер турнира может изменяться.

Код 447. Синтаксис

```
int MHL_BinaryGeneticAlgorithmWDTS(double *Parameters, double (*FitnessFunction)(int *, int), int *VMHL_ResultVector, double *VMHL_Result);
```

## Входные параметры:

Parameters — Вектор параметров генетического алгоритма. Каждый элемент обозначает свой параметр:

- 0 — длина бинарной хромосомы (определяется задачей оптимизации, что мы решаем);
  - 1 — число вычислений целевой функции (CountOfFitness);
  - 2 — размер турнирной селекции (SizeOfTournament): от 2 до  $\sqrt{\text{CountOfFitness}}$ ;
  - 3 — тип скрещивания (TypeOfCross):

- 0 – SinglepointCrossover (Одноточечное скрещивание);
- 1 – TwopointCrossover (Двухточечное скрещивание);
- 2 – UniformCrossover (Равномерное скрещивание).

4 — тип мутации (TypeOfMutation):

- 0 – Weak (Слабая мутация);
- 1 – Average (Средняя мутация);
- 2 – Strong (Сильная мутация).

5 — тип формирования нового поколения (TypeOfForm):

- 0 – OnlyOffspringGenerationForming (Только потомки);
- 1 – OnlyOffspringWithBestGenerationForming (Только потомки и копия лучшего индивида).

FitnessFunction — указатель на целевую функцию (если решается задача условной оптимизации, то учет ограничений должен быть включен в эту функцию);

VMHL\_ResultVector — найденное решение (бинарный вектор);

VMHL\_Result — значение целевой функции в точке, определенной вектором VMHL\_ResultVector.

#### **Возвращаемое значение:**

1 — завершил работу без ошибок. Всё хорошо.

0 — возникли при работе ошибки. Скорее всего в этом случае в VMHL\_ResultVector и в VMHL\_Result не содержится решение задачи.

#### **О функции:**

Отличается от стандартного генетического алгоритма, тем, что присутствует только турнирная селекция, но размер турнира может изменяться.

Алгоритм бинарной оптимизации. Ищет максимум целевой функции FitnessFunction.

Решением является бинарная строка, то есть вектор, состоящий из 0 и 1.

Тип алгоритма: добавочный алгоритм оптимизации.

Подробное описание алгоритма можно найти тут:

<https://github.com/Harrix/HarrixOptimizationAlgorithms>

**Примерный настройки** (для примера Вы можете поставить такие рабочие настройки):

Parameters[0]=50;

Parameters[1]=100\*100;

Parameters[2]=2;

Parameters[3]=2;

Parameters[4]=1;

Parameters[5]=1;

Код целевой функции:

## Код 448. Оптимизируемая функция

```
double Func(int *x, int VMHL_N)
{
    //Сумма всех элементов массива
    return TMHL_SumVector(x, VMHL_N);
}
```

## Код 449. Пример использования

#### 7.16.4 MHL\_RealGeneticAlgorithmWCC

Генетический алгоритм для решения задач на вещественных строках, в котором есть только два вида скрещивания: одноточечное и двухточечное скрещивание с возможностью полного копирования одного из родителей. Равномерное скрещивание отсутствует.

Код 450. Синтаксис

```
int MHL_RealGeneticAlgorithmWCC(int *Parameters, int *NumberOfParts, double *Left,
                                double *Right, double (*FitnessFunction)(double*, int), double *VMHL_ResultVector,
                                double *VMHL_Result);
```

##### Входные параметры:

Parameters — Вектор параметров генетического алгоритма. Каждый элемент обозначает свой параметр:

0 — длина вещественной хромосомы (определяется задачей оптимизации, что мы решаем);

1 — число вычислений целевой функции (CountOfFitness);

2 — тип селекции (TypeOfSel):

- 0 — ProportionalSelection (Пропорциональная селекция);
- 1 — RankSelection (Ранговая селекция);
- 2 — TournamentSelection (Турнирная селекция).

3 — тип скрещивания (TypeOfCros):

- 0 — SinglepointCrossoverWithCopying (Одноточечное скрещивание с возможностью полного копирования одного из родителей);
- 1 — TwopointCrossoverWithCopying (Двухточечное скрещивание с возможностью полного копирования одного из родителей).

4 — тип мутации (TypeOfMutation):

- 0 — Weak (Слабая мутация);
- 1 — Average (Средняя мутация);
- 2 — Strong (Сильная мутация).

5 — тип формирования нового поколения (TypeOfForm):

- 0 — OnlyOffspringGenerationForming (Только потомки);
- 1 — OnlyOffspringWithBestGenerationForming (Только потомки и копия лучшего индивида).

6 — тип преобразования задачи вещественной оптимизации в задачу бинарной оптимизации (TypeOfConverting);

- 0 — IntConverting (Стандартное представление целого числа — номер узла в сетке дискретизации);
- 1 — GrayCodeConverting (Стандартный рефлексивный Грей-код).

NumberOfParts — указатель на массив: на сколько частей делить каждую вещественную координату при дискретизации (размерность Parameters[0]);

Желательно брать по формуле  $NumberOfParts[i] = 2^k - 1$ , где  $k$  — натуральное число, например, 12.

Left — массив левых границ изменения каждой вещественной координаты (размерность Parameters[0]);

Right — массив правых границ изменения каждой вещественной координаты (размерность Parameters[0]);

FitnessFunction — указатель на целевую функцию (если решается задача условной оптимизации, то учет ограничений должен быть включен в эту функцию);

VMHL\_ResultVector — найденное решение (вещественный вектор);

VMHL\_Result — значение целевой функции в точке, определенной вектором VMHL\_ResultVector.

### **Возвращаемое значение:**

1 — завершил работу без ошибок. Всё хорошо.

0 — возникли при работе ошибки. Скорее всего в этом случае в VMHL\_ResultVector и в VMHL\_Result не содержится решение задачи.

### **О функции:**

Отличается от стандартного генетического алгоритма тем, что есть только два вида скрещивания: одноточечное и двухточечное скрещивание с возможностью полного копирования одного из родителей. Равномерное скрещивание отсутствует. То есть данным алгоритмом проверяем: есть ли разница в эффективности алгоритма, если точки разрыва при скрещивании делать и по краям родителей, а не только внутри хромосомы.

Алгоритм вещественной оптимизации. Ищет максимум целевой функции FitnessFunction.

Решением является вещественная строка.

Тип алгоритма: исследовательский алгоритм оптимизации.

Подробное описание алгоритма можно найти тут:

<https://github.com/Harrix/HarrixOptimizationAlgorithms>

**Примерный настройки** (для примера Вы можете поставить такие рабочие настройки):

Parameters[0]=50;

Parameters[1]=100\*100;

Parameters[2]=2;

Parameters[3]=1;

Parameters[4]=1;

Parameters[5]=1;

Parameters[6]=0;

### Примечание:

В сГА на вещественных строках не нужно задавать в параметрах число поколений и размер популяции, а только число вычислений целевой функции. Почему? Алгоритм сам определит число поколений и размер популяции, исходя из принципа, что число поколений и размер популяции должны быть примерно равны. Поэтому выбирайте значение Parameters[1] в виде:

```
int K=100;
```

```
Parameters[1]=K*K;
```

То есть в виде квадрата целого числа. В противном случае реальное число вычислений целевой функции и значение Parameters[1] будут не совпадать.

Код целевой функции:

Код 451. Оптимизируемая функция

```
double Func2(double *x,int VMHL_N)
{
return -((x[0]-2)*(x[0]-2)+(x[1]-2)*(x[1]-2));
}
```

Код 452. Пример использования

```
int ChromosomeLength=2; //Длина хромосомы
int CountOfFitness=50*50; //Число вычислений целевой функции
int TypeOfSel=1; //Тип селекции
int TypeOfCros=0; //Тип скрещивания
int TypeOfMutation=1; //Тип мутации
int TypeOfForm=0; //Тип формирования нового поколения

int *Parameters;
Parameters=new int[7];
Parameters[0]=ChromosomeLength; //Длина хромосомы
Parameters[1]=CountOfFitness; //Число вычислений целевой функции
Parameters[2]=TypeOfSel; //Тип селекции
Parameters[3]=TypeOfCros; //Тип скрещивания
Parameters[4]=TypeOfMutation; //Тип мутации
Parameters[5]=TypeOfForm; //Тип формирования нового поколения
Parameters[6]=0; //Тип преобразование задачи вещественной оптимизации в задачу бинарной оптимизации

double *Left; //массив левых границ изменения каждой вещественной координаты
Left=new double[ChromosomeLength];
double *Right; //массив правых границ изменения каждой вещественной координаты
Right=new double[ChromosomeLength];
int *NumberOfParts; //на сколько делить каждую координату
NumberOfParts=new int[ChromosomeLength];

//Заполним массивы
//Причем по каждой координате одинаковые значения выставим
TMHL_FillVector(Left,ChromosomeLength,-5.); //Пусть будет интервал [-3;3]
TMHL_FillVector(Right,ChromosomeLength,5.);
TMHL_FillVector(NumberOfParts,ChromosomeLength,TMHL_PowerOf(2,15)-1); //Делим на 32768-1 частей каждую вещественную координату

double *Decision; //вещественное решение
Decision=new double[ChromosomeLength];
double ValueOfFitnessFunction; //значение целевой функции в точке Decision
int VMHL_Success=0; //Успешен ли будет запуск сГА
```

```

//Запуск алгоритма
VMHL_Success=MHL_RealGeneticAlgorithmWCC (Parameters,NumberOfParts,Left,Right,Func2,
    Decision, &ValueOfFitnessFunction);

//Используем полученный результат
MHL_ShowNumber(VMHL_Success,"Как прошел запуск","VMHL_Success");
if (VMHL_Success==1)
{
    MHL_ShowVectorT(Decision,ChromosomeLength,"Найденное решение","Decision");
    //Найденное решение:
    //Decision =
    //2.0105 2.00195
    MHL_ShowNumber(ValueOfFitnessFunction,"Значение целевой функции",
        "ValueOfFitnessFunction");
    //Значение целевой функции:
    //ValueOfFitnessFunction=-0.000114024

```

## 7.16.5 MHL\_RealGeneticAlgorithmWDPOfNOfGPS

Генетический алгоритм для решения задач на вещественных строках с изменяющимся соотношением числа поколений и размера популяции. Отличается от стандартного генетического алгоритма, тем, что размер популяции и число поколений рассчитывается из числа вычислений целевой функции не как одинаковые величины (извлечение квадратного корня), а через некоторое соотношение.

Код 453. Синтаксис

```

int MHL_RealGeneticAlgorithmWDPOfNOfGPS(double *Parameters, int *NumberOfParts,
    double *Left, double *Right, double (*FitnessFunction)(double*,int), double *
    VMHL_ResultVector, double *VMHL_Result);

```

### Входные параметры:

Parameters — Вектор параметров генетического алгоритма. Каждый элемент обозначает свой параметр:

- 0 — длина вещественной хромосомы (определяется задачей оптимизации, что мы решаем);
- 1 — число вычислений целевой функции (CountOfFitness);
- 2 — тип селекции (TypeOfSel):
  - 0 — ProportionalSelection (Пропорциональная селекция);
  - 1 — RankSelection (Ранговая селекция);
  - 2 — TournamentSelection (Турнирная селекция).
- 3 — тип скрещивания (TypeOfCros):
  - 0 — SinglepointCrossover (Одноточечное скрещивание);
  - 1 — TwopointCrossover (Двуточечное скрещивание);
  - 2 — UniformCrossover (Равномерное скрещивание).
- 4 — тип мутации (TypeOfMutation):

- 0 — Weak (Слабая мутация);
- 1 — Average (Средняя мутация);
- 2 — Strong (Сильная мутация).

5 — тип формирования нового поколения (TypeOfForm):

- 0 — OnlyOffspringGenerationForming (Только потомки);
- 1 — OnlyOffspringWithBestGenerationForming (Только потомки и копия лучшего индивида).

6 — тип преобразования задачи вещественной оптимизации в задачу бинарной оптимизации (TypeOfConverting):

- 0 — IntConverting (Стандартное представление целого числа — номер узла в сетке дискретизации);
- 1 — GrayCodeConverting (Стандартный рефлексивный Грей-код).

7 — «доля» (Proportion) числа поколений от общего числа вычислений целевой функции. Определяется как возведение числа вычислений целевой функции в степень Proportion. Может принимать значения в интервале [0; 1]. Число поколений =  $\text{int}(\text{CountOfFitness}^{\text{Proportion}})$ . Размер популяции =  $\text{int}(\text{CountOfFitness}/\text{Число поколений})$ . При Proportion=0.5 получим обычный стандартный генетический алгоритм. Чем меньше Proportion, тем меньше число поколений будет. Желательно, чтобы принимались следующие значения:

- 0;
- 0.1;
- 0.2;
- 0.3;
- 0.4;
- 0.5;
- 0.6;
- 0.7;
- 0.8;
- 0.9;
- 1.

NumberOfParts — указатель на массив: на сколько частей делить каждую вещественную координату при дискретизации (размерность Parameters[0]);

Желательно брать по формуле  $\text{NumberOfParts}[i] = 2^k - 1$ , где  $k$  — натуральное число, например, 12.

Left — массив левых границ изменения каждой вещественной координаты (размерность Parameters[0]);

Right — массив правых границ изменения каждой вещественной координаты (размерность Parameters[0]);

FitnessFunction — указатель на целевую функцию (если решается задача условной оптимизации, то учет ограничений должен быть включен в эту функцию);

VMHL\_ResultVector — найденное решение (вещественный вектор);  
VMHL\_Result — значение целевой функции в точке, определенной вектором VMHL\_ResultVector.

### **Возвращаемое значение:**

1 — завершил работу без ошибок. Всё хорошо.  
0 — возникли при работе ошибки. Скорее всего в этом случае в VMHL\_ResultVector и в VMHL\_Result не содержится решение задачи.

### **О функции:**

Отличается от стандартного генетического алгоритма, что число поколений может изменяться по описанному выше принципу.

Алгоритм вещественной оптимизации. Ищет максимум целевой функции FitnessFunction.

Решением является вещественная строка.

Тип алгоритма: исследовательский алгоритм оптимизации.

Подробное описание алгоритма можно найти тут:

<https://github.com/Harrix/HarrixOptimizationAlgorithms>

**Примерный настройки** (для примера Вы можете поставить такие рабочие настройки):

Parameters[0]=50;

Parameters[1]=100\*100;

Parameters[2]=2;

Parameters[3]=2;

Parameters[4]=1;

Parameters[5]=1;

Parameters[6]=0;

Parameters[7]=0.5;

Код целевой функции:

Код 454. Оптимизируемая функция

```
double Func2(double *x, int VMHL_N)
{
    return -((x[0]-2)*(x[0]-2)+(x[1]-2)*(x[1]-2));
}
```

Код 455. Пример использования

```
int ChromosomeLength=50; //Длина хромосомы
int CountOfFitness=50*50; //Число вычислений целевой функции
int SizeOfTournament=2; //Размер турнира в турнирной селекции
int TypeOfCross=0; //Тип скрещивания
int TypeOfMutation=1; //Тип мутации
int TypeOfFormation=0; //Тип формирования нового поколения
```

## 7.16.6 MHL RealGeneticAlgorithm WDTs

Генетический алгоритм для решения задач на вещественных строках с турнирной селекцией, где размер турнира изменяется от 2 до размера популяции. Отличается от стандартного генетического алгоритма, тем, что присутствует только турнирная селекция, но размер турнира может изменяться.

## Код 456. Синтаксис

```
int MHL_RealGeneticAlgorithmWDTs(double *Parameters, int *NumberOfParts, double *Left  
, double *Right, double (*FitnessFunction)(double*,int), double *VMHL_ResultVector  
, double *VMHL_Result);
```

## Входные параметры:

**Parameters** — Вектор параметров генетического алгоритма. Каждый элемент обозначает свой параметр:

- 0 — длина вещественной хромосомы (определяется задачей оптимизации, что мы решаем);
- 1 — число вычислений целевой функции (CountOfFitness);
- 2 — размер турнирной селекции (SizeOfTournament): от 2 до  $\sqrt{\text{CountOfFitness}}$ ;
- 3 — тип скрещивания (TypeOfCros):
  - 0 — SinglepointCrossover (Одноточечное скрещивание);
  - 1 — TwopointCrossover (Двухточечное скрещивание);
  - 2 — UniformCrossover (Равномерное скрещивание).
- 4 — тип мутации (TypeOfMutation):
  - 0 — Weak (Слабая мутация);
  - 1 — Average (Средняя мутация);
  - 2 — Strong (Сильная мутация).
- 5 — тип формирования нового поколения (TypeOfForm):
  - 0 — OnlyOffspringGenerationForming (Только потомки);
  - 1 — OnlyOffspringWithBestGenerationForming (Только потомки и копия лучшего индивида).
- 6 — тип преобразования задачи вещественной оптимизации в задачу бинарной оптимизации (TypeOfConverting):
  - 0 — IntConverting (Стандартное представление целого числа — номер узла в сетке дискретизации);
  - 1 — GrayCodeConverting (Стандартный рефлексивный Грей-код).

**NumberOfParts** — указатель на массив: на сколько частей делить каждую вещественную координату при дискретизации (размерность Parameters[0]);

Желательно брать по формуле  $\text{NumberOfParts}[i] = 2^k - 1$ , где  $k$  — натуральное число, например, 12.

**Left** — массив левых границ изменения каждой вещественной координаты (размерность Parameters[0]);

**Right** — массив правых границ изменения каждой вещественной координаты (размерность Parameters[0]);

**FitnessFunction** — указатель на целевую функцию (если решается задача условной оптимизации, то учет ограничений должен быть включен в эту функцию);

**VMHL\_ResultVector** — найденное решение (вещественный вектор);

**VMHL\_Result** — значение целевой функции в точке, определенной вектором VMHL\_ResultVector.

**Возвращаемое значение:**

1 — завершил работу без ошибок. Всё хорошо.

0 — возникли при работе ошибки. Скорее всего в этом случае в VMHL\_ResultVector и в VMHL\_Result не содержится решение задачи.

## О функции:

Отличается от стандартного генетического алгоритма, тем, что присутствует только турнирная селекция, но размер турнира может изменяться.

Алгоритм вещественной оптимизации. Ищет максимум целевой функции FitnessFunction.

Решением является вещественная строка.

Тип алгоритма: добавочный алгоритм оптимизации.

Подробное описание алгоритма можно найти тут:

<https://github.com/Harrix/HarrixOptimizationAlgorithms>

**Примерный настройки** (для примера Вы можете поставить такие рабочие настройки):

Parameters[0]=50;

Parameters[1]=100\*100;

Parameters[2]=2;

Parameters[3]=2;

Parameters[4]=1;

Parameters[5]=1;

Parameters[6]=0;

Код целевой функции:

Код 457. Оптимизируемая функция

```
double Func2(double *x, int VMHL_N)
{
    return -((x[0]-2)*(x[0]-2)+(x[1]-2)*(x[1]-2));
}
```

Код 458. Пример использования

```
int ChromosomeLength=2; //Длина хромосомы
int CountOfFitness=50*50; //Число вычислений целевой функции
int SizeOfTournament=2; //Размер турнира в турнирной селекции
int TypeOfCros=0; //Тип скрещивания
int TypeOfMutation=1; //Тип мутации
int TypeOfForm=0; //Тип формирования нового поколения

double *ParametersOfAlgorithm;
ParametersOfAlgorithm=new double[7];
ParametersOfAlgorithm[0]=ChromosomeLength; //Длина хромосомы
ParametersOfAlgorithm[1]=CountOfFitness; //Число вычислений целевой функции
ParametersOfAlgorithm[2]=SizeOfTournament; //Размер турнира в турнирной селекции
ParametersOfAlgorithm[3]=TypeOfCros; //Тип скрещивания
ParametersOfAlgorithm[4]=TypeOfMutation; //Тип мутации
ParametersOfAlgorithm[5]=TypeOfForm; //Тип формирования нового поколения
```

```

ParametersOfAlgorithm[6]=0; //Тun преобразование задачи вещественной оптимизации в задачу бинарной оптимизации

double *Left; //массив левых границ изменения каждой вещественной координаты
Left=new double[ChromosomeLength];
double *Right; //массив правых границ изменения каждой вещественной координаты
Right=new double[ChromosomeLength];
int *NumberOfParts; //на сколько делить каждую координату
NumberOfParts=new int[ChromosomeLength];

//Заполним массивы
//Причем по каждой координате одинаковые значения выставим
TMHL_FillVector(Left,ChromosomeLength,-5.); //Пусть будет интервал [-3;3]
TMHL_FillVector(Right,ChromosomeLength,5.);
TMHL_FillVector(NumberOfParts,ChromosomeLength,TMHL_PowerOf(2,15)-1); //Делим на
32768-1 частей каждую вещественную координату

double *Decision; //вещественное решение
Decision=new double[ChromosomeLength];
double ValueOfFitnessFunction; //значение целевой функции в точке Decision
int VMHL_Success=0; //Успешен ли будет запуск СГА

//Запуск алгоритма
VMHL_Success=MHL_RealGeneticAlgorithmWDTS (ParametersOfAlgorithm,NumberOfParts,Left,
Right,Func2, Decision, &ValueOfFitnessFunction);

//Используем полученный результат
MHL_ShowNumber(VMHL_Success,"Как прошел запуск","VMHL_Success");
if (VMHL_Success==1)
{
    MHL_ShowVectorT(Decision,ChromosomeLength,"Найденное решение","Decision");
    //Найденное решение:
    //Decision =
    //2.00226 1.98883

    MHL_ShowNumber(ValueOfFitnessFunction,"Значение целевой функции",
ValueOfFitnessFunction);
    //Значение целевой функции:
    //ValueOfFitnessFunction=-0.000129856
}

delete [] ParametersOfAlgorithm;
delete [] Decision;
delete [] Left;
delete [] Right;
delete [] NumberOfParts;

```

## 7.17 Перевод единиц измерений

### 7.17.1 MHL\_DegToRad

Функция переводит угол из градусной меры в радианную.

Код 459. Синтаксис

```
double MHL_DegToRad(double VMHL_X);
```

### **Входные параметры:**

VMHL\_X — градусная мера угла.

**Возвращаемое значение:** Радианная мера угла.

Код 460. Пример использования

```
double Rad;
double Deg=90; //угол в градусах

//Вызов функции
Rad=MHL_DegToRad(Deg);

//Используем полученный результат
MHL_ShowNumber(Rad, "Угол "+MHL_NumberToText(Deg)+" градусов", "равен в радианах");
//Угол 90 градусов:
//равен в радианах=1.5708
```

### **7.17.2 MHL\_RadToDeg**

Функция переводит угол из радианной меры в градусную.

Код 461. Синтаксис

```
double MHL_RadToDeg(double VMHL_X);
```

### **Входные параметры:**

VMHL\_X — радианная мера угла.

**Возвращаемое значение:** Градусная мера угла.

Код 462. Пример использования

```
double Deg;
double Rad=MHL_PI; //Угол в радианах

//Вызов функции
Deg=MHL_RadToDeg(Rad);

//Используем полученный результат
MHL_ShowNumber(Deg, "Угол "+MHL_NumberToText(Rad)+" радиан", "равен в градусах");
//Угол 3.14159 радиан:
//равен в градусах=180
```

## **7.18 Случайные объекты**

### **7.18.1 MHL\_BitNumber**

Функция с вероятностью P (или 0.5 в переопределенной функции) возвращает 1. В противном случае возвращает 0.

Код 463. Синтаксис

```
int MHL_BitNumber(double P);
int MHL_BitNumber();
```

Есть две функции с разным набором аргументов.

Для первой функции:

**Входные параметры:**

P — вероятность появления 1.

**Возвращаемое значение:** 1 или 0.

Для второй функции:

**Входные параметры:**

Отсутствуют.

**Возвращаемое значение:** 1 или 0.

Код 464. Пример использования

```
int x;
double P=0.8; //Угол в радианах

//Вызов функции
x=MHL_BitNumber(P);

//Используем полученный результат
MHL_ShowNumber(x, "Из 0 и 1 с вероятностью "+MHL_NumberToText(P), "выбрано");

//Вызов функции
x=MHL_BitNumber();

//Используем полученный результат
MHL_ShowNumber(x, "Из 0 и 1 с вероятностью 0.5", "выбрано");
```

### 7.18.2 MHL\_RandomRealMatrix

Функция заполняет матрицу случайными вещественными числами из определенного интервала [Left;Right].

Код 465. Синтаксис

```
void MHL_RandomRealMatrix(double **VMHL_ResultMatrix, double Left, double Right, int
    VMHL_N, int VMHL_M);
```

**Входные параметры:**

VMHL\_ResultMatrix — указатель на матрицу;

Left — левая граница интервала;

Right — правая граница интервала;

VMHL\_N — размер массива (число строк);

VMHL\_M — размер массива (число столбцов).

**Возвращаемое значение:** Отсутствует.

Код 466. Пример использования

```

int i;
int VMHL_N=3; //Размер массива (число строк)
int VMHL_M=3; //Размер массива (число столбцов)
double **a;
a=new double*[VMHL_N];
for (i=0;i<VMHL_N;i++) a[i]=new double[VMHL_M];

double Left=-3; //левая граница интервала;
double Right=3; //правая граница интервала;

//Вызов функции
MHL_RandomRealMatrix(a,Left,Right,VMHL_N,VMHL_M);

//Используем полученный результат
MHL_ShowMatrix (a,VMHL_N,VMHL_M,"Случайная матрица", "a");
//Случайная матрица:
//a =
//1.97571  0.862793 -0.357422
//-2.62701 -0.202515 -2.79932
//1.38794  1.35535 -2.29449

for (i=0;i<VMHL_N;i++)
    delete [] a[i];
delete [] a;

```

### 7.18.3 MHL\_RandomRealMatrixInCols

Функция заполняет матрицу случайными вещественными числами из определенного интервала. При этом элементы каждого столбца изменяются в своих пределах.

Код 467. Синтаксис

```
void MHL_RandomRealMatrixInCols(double **VMHL_ResultMatrix, double *Left, double *
    Right, int VMHL_N, int VMHL_M);
```

#### Входные параметры:

VMHL\_ResultMatrix — указатель на матрицу;

Left — левые границы интервала изменения элементов столбца (размер VMHL\_M);

Right — правые границы интервала изменения элементов столбца (размер VMHL\_M);

VMHL\_N — размер массива (число строк);

VMHL\_M — размер массива (число столбцов).

**Возвращаемое значение:** Отсутствует.

Код 468. Пример использования

```

int i;
int VMHL_N=3; //Размер массива (число строк)
int VMHL_M=3; //Размер массива (число столбцов)
double **a;
a=new double*[VMHL_N];
for (i=0;i<VMHL_N;i++) a[i]=new double[VMHL_M];
double *Left;
Left=new double[VMHL_M];
double *Right;

```

```

Right=new double[VMHL_M];

Left[0]=-5; //левая границы интервала изменения 1 столбца
Right[0]=-4; //правая граница интервала изменения 1 столбца

Left[1]=0; //левая границы интервала изменения 2 столбца
Right[1]=3; //правая граница интервала изменения 2 столбца

Left[2]=100; //левая границы интервала изменения 3 столбца
Right[2]=200; //правая граница интервала изменения 3 столбца

//Вызов функции
MHL_RandomRealMatrixInCols(a,Left,Right,VMHL_N,VMHL_M);

//Используем полученный результат
MHL_ShowMatrix (a,VMHL_N,VMHL_M,"Случайная матрица", "a");
//Случайная матрица:
//a =
// -4.20267 2.20367 148.468
// -4.42432 2.09418 138.654
// -4.07089 1.95831 140.198

for (i=0;i<VMHL_N;i++) delete [] a[i];
delete [] a;
delete [] Left;
delete [] Right;

```

#### 7.18.4 MHL\_RandomRealMatrixInElements

Функция заполняет матрицу случайными вещественными числами из определенного интервала. При этом каждый элемент изменяется в своих пределах.

Код 469. Синтаксис

```
void MHL_RandomRealMatrixInElements(double **VMHL_ResultMatrix, double **Left, double
**Right, int VMHL_N, int VMHL_M);
```

**Входные параметры:**

VMHL\_ResultMatrix — указатель на матрицу;

Left — левые границы интервала изменения каждого элемента (размер VMHL\_N x VMHL\_M);

Right — правые границы интервала изменения каждого элемента (размер VMHL\_N x VMHL\_M);

VMHL\_N — размер массива (число строк);

VMHL\_M — размер массива (число столбцов).

**Возвращаемое значение:** Отсутствует.

Код 470. Пример использования

```
int i,j;
int VMHL_N=3; //Размер массива (число строк)
int VMHL_M=3; //Размер массива (число столбцов)
```

```

double **a;
a=new double*[VMHL_N];
for (i=0;i<VMHL_N;i++) a[i]=new double[VMHL_M];
double **Left;
Left=new double*[VMHL_N];
for (i=0;i<VMHL_N;i++) Left[i]=new double[VMHL_M];
double **Right;
Right=new double*[VMHL_N];
for (i=0;i<VMHL_N;i++) Right[i]=new double[VMHL_M];

//Возьмем для примера границы интервала равными около номера ячейки в матрице
for (i=0;i<VMHL_N;i++)
for (j=0;j<VMHL_M;j++)
{
    Left[i][j]=i*VMHL_N+j-0.1;
    Right[i][j]=Left[i][j]+0.2;
}

//Вызов функции
MHL_RandomRealMatrixInElements(a,Left,Right,VMHL_N,VMHL_M);

//Используем полученный результат

MHL_ShowMatrix (Left,VMHL_N,VMHL_M,"Матрица левых границ", "Left");
// Матрица левых границ:
//Left =
// -0.1 0.9 1.9
// 2.9 3.9 4.9
// 5.9 6.9 7.9

MHL_ShowMatrix (Right,VMHL_N,VMHL_M,"Матрица правых границ", "Right");
// Матрица правых границ:
//Right =
// 0.1 1.1 2.1
// 3.1 4.1 5.1
// 6.1 7.1 8.1

MHL_ShowMatrix (a,VMHL_N,VMHL_M,"Случайная матрица", "a");
// Случайная матрица:
//a =
// 0.0829529 1.04504 1.9892
// 2.90126 3.92388 4.90221
// 5.96102 6.90623 8.09661

for (i=0;i<VMHL_N;i++) delete [] a[i];
delete [] a;
for (i=0;i<VMHL_N;i++) delete [] Left[i];
delete [] Left;
for (i=0;i<VMHL_N;i++) delete [] Right[i];
delete [] Right;

```

### 7.18.5 MHL\_RandomRealMatrixInRows

Функция заполняет матрицу случайными вещественными числами из определенного интервала. При этом элементы каждой строки изменяются в своих пределах.

Код 471. Синтаксис

```
void MHL_RandomRealMatrixInRows(double **VMHL_ResultMatrix, double *Left, double *Right, int VMHL_N, int VMHL_M);
```

### Входные параметры:

VMHL\_ResultMatrix — указатель на матрицу;

Left — левые границы интервала изменения элементов строки (размер VMHL\_N);

Right — правые границы интервала изменения элементов строки (размер VMHL\_N);

VMHL\_N — размер массива (число строк);

VMHL\_M — размер массива (число столбцов).

**Возвращаемое значение:** Отсутствует.

### Код 472. Пример использования

```
int i;
int VMHL_N=3; //Размер массива (число строк)
int VMHL_M=3; //Размер массива (число столбцов)
double **a;
a=new double*[VMHL_N];
for (i=0;i<VMHL_N;i++) a[i]=new double[VMHL_M];
double *Left;
Left=new double[VMHL_N];
double *Right;
Right=new double[VMHL_N];

Left[0]=-5; //левая границы интервала изменения 1 строки
Right[0]=-4; //правая граница интервала изменения 1 строки

Left[1]=0; //левая границы интервала изменения 2 строки
Right[1]=3; //правая граница интервала изменения 2 строки

Left[2]=100; //левая границы интервала изменения 3 строки
Right[2]=200; //правая граница интервала изменения 3 строки

//Вызов функции
MHL_RandomRealMatrixInRows(a,Left,Right,VMHL_N,VMHL_M);

//Используем полученный результат

MHL_ShowMatrix (a,VMHL_N,VMHL_M,"Случайная матрица", "a");
// Случайная матрица:
//a =
//-4.98376 -4.64868 -4.38959
//1.14386 2.70071 2.76151
//141.309 192.12 100.122

for (i=0;i<VMHL_N;i++) delete [] a[i];
delete [] a;
delete [] Left;
delete [] Right;
```

### 7.18.6 MHL\_RandomRealVector

Функция заполняет массив случайными вещественными числами из определенного интервала [Left;Right].

Код 473. Синтаксис

```
void MHL_RandomRealVector(double *VMHL_ResultVector, double Left, double Right, int VMHL_N);
```

#### Входные параметры:

VMHL\_ResultVector — указатель на массив;

Left — левая граница интервала;

Right — правая граница интервала;

VMHL\_N — размер массива.

**Возвращаемое значение:** Отсутствует.

Код 474. Пример использования

```
int VMHL_N=10; //Размер массива
double *a;
a=new double[VMHL_N];

double Left=-3;
double Right=3;

//Вызов функции
MHL_RandomRealVector(a,Left,Right,VMHL_N);

//Используем полученный результат
MHL_ShowVector (a,VMHL_N,"Массив", "a");
// Массив:
//a =
//1.73822
//-0.406311
//-2.7572
//-0.351013
//0.367493
//1.40991
//0.662476
// -1.15576
// -1.75781
// -2.06927

delete [] a;
```

### 7.18.7 MHL\_RandomRealVectorInElements

Функция заполняет массив случайными вещественными числами из определенного интервала, где на каждую координату свои границы изменения.

Код 475. Синтаксис

```
void MHL_RandomRealVectorInElements(double *VMHL_ResultVector, double *Left, double *
Right, int VMHL_N);
```

### Входные параметры:

VMHL\_ResultVector — указатель на массив;

Left — левые границы интервалов (размер VMHL\_N);

Right — правые границы интервалов (размер VMHL\_N)

VMHL\_N — размер массива.

**Возвращаемое значение:** Отсутствует.

### Код 476. Пример использования

```
int VMHL_N=2; //Размер массива
double *a;
a=new double[VMHL_N];

double *Left;
Left=new double[VMHL_N];
Left[0]=-3;//Левая граница изменения первого элемента массива
Left[1]=5;//Левая граница изменения второго элемента массива

double *Right;
Right=new double[VMHL_N];
Right[0]=3;//Правая граница изменения первого элемента массива
Right[1]=10;//Правая граница изменения второго элемента массива

//Вызов функции
MHL_RandomRealVectorInElements(a,Left,Right,VMHL_N);

//Используем полученный результат

MHL_ShowVector (Left,VMHL_N,"Массив левых границ", "Left");
// Массив левых границ:
//Left =
// -3
// 5

MHL_ShowVector (Right,VMHL_N,"Массив правых границ", "Right");
// Массив правых границ:
//Right =
// 3
// 10

MHL_ShowVector (a,VMHL_N,"Случайных массив", "a");
// Случайных массив:
//a =
// 1.32111
// 6.5625

delete [] a;
delete [] Left;
delete [] Right;
```

### 7.18.8 MHL\_RandomVectorOfProbability

Функция заполняет вектор случайными значениями вероятностей. Сумма всех элементов вектора равна 1.

Код 477. Синтаксис

```
void MHL_RandomVectorOfProbability(double *VMHL_ResultVector, int VMHL_N);
```

**Входные параметры:**

VMHL\_ResultVector — указатель на вектор вероятностей (одномерный массив);

VMHL\_N — размер массива.

**Возвращаемое значение:** Отсутствует.

Код 478. Пример использования

```
int VMHL_N=10; //Размер массива (число строк)
double *a;
a=new double[VMHL_N];

//Заполним вектор случайными значениями вероятностей
//Вызов функции
MHL_RandomVectorOfProbability(a, VMHL_N);

//Используем полученный результат
MHL_ShowVector (a,VMHL_N, "Вектор вероятностей выбора", "a");
// Вектор вероятностей выбора:
//a =
//0.0662721
//0.0681826
//0.083972
//0.0554142
//0.18878
//0.160006
//0.0698625
//0.0652843
//0.127822
//0.114404

MHL_ShowNumber (TMHL_SumVector(a,VMHL_N), "Его сумма", "Sum");
// Его сумма:
//Sum=1
```

### 7.18.9 TMHL\_BernulliVector

Функция формирует случайный вектор Бернулли.

Код 479. Синтаксис

```
template <class T> void TMHL_BernulliVector(T *VMHL_ResultVector, int VMHL_N);
```

**Входные параметры:**

VMHL\_ResultVector — указатель на вектор (одномерный массив);

VMHL\_N — размер массива.

**Возвращаемое значение:**

Отсутствует.

Код 480. Пример использования

```
int VMHL_N=10; //Размер массива (число строк)
double *a;
a=new double[VMHL_N];

//Вызов функции
TMHL_BernulliVector(a,VMHL_N);

//Используем полученный результат
MHL_ShowVector (a,VMHL_N,"Случайный вектор Бернулли", "a");
    //Случайный вектор Бернулли:
//a =
//1
//1
//1
//1
//1
//1
//1
//1
//1
//1
```

### 7.18.10 TMHL\_RandomArrangingObjectsIntoBaskets

Функция предлагает случайный способ расставить N объектов в VMHL\_N корзин при условии, что в каждой корзине может располагаться только один предмет.

Код 481. Синтаксис

```
template <class T> void TMHL_RandomArrangingObjectsIntoBaskets(T *VMHL_ResultVector,
    int N, int VMHL_N);
```

**Входные параметры:**

VMHL\_ResultVector — массив, в который записывается результат;

N — число предметов;

VMHL\_N — размер массива (и число корзин).

**Возвращаемое значение:**

Отсутствует.

Код 482. Пример использования

```
int VMHL_N=10; //Размер массива
int *a;
a=new int[VMHL_N];

int N=MHL_RandomUniformInt(0,10); // Размер турнира
```

```

//Вызов функции
TMHL_RandomArrangingObjectsIntoBaskets(a,N,VMHL_N);

//Используем полученный результат
MHL_ShowNumber (N,"Число предметов", "N");
// Число предметов:
// N=5
MHL_ShowVectorT (a,VMHL_N,"Случаное расположение по 10 корзинам", "a");
// Случаное расположение по 10 корзинам:
//a =
//0   1   0   0   1   1   0   1   1

delete [] a;

```

### 7.18.11 TMHL\_RandomBinaryMatrix

Функция заполняет матрицу случайно нулями и единицами.

Код 483. Синтаксис

```
template <class T> void TMHL_RandomBinaryMatrix(T **VMHL_ResultMatrix,int VMHL_N,int VMHL_M);
```

**Входные параметры:**

VMHL\_ResultMatrix — указатель на преобразуемый массив;

VMHL\_N — размер массива VMHL\_ResultMatrix (число строк);

VMHL\_M — размер массива VMHL\_ResultMatrix (число столбцов).

**Возвращаемое значение:**

Отсутствует.

Код 484. Пример использования

```

int i;
int VMHL_N=10;//Размер массива (число строк)
int VMHL_M=3;//Размер массива (число столбцов)
int **a;
a=new int*[VMHL_N];
for (i=0;i<VMHL_N;i++) a[i]=new int[VMHL_M];

//Вызов функции
TMHL_RandomBinaryMatrix(a,VMHL_N,VMHL_M);

//Используем полученный результат
MHL_ShowMatrix (a,VMHL_N,VMHL_M,"Случайная бинарная матрица", "a");
//Случайная бинарная матрица:
//a =
//1   0   1
//0   0   0
//1   1   1
//1   0   0
//1   1   0
//1   1   0
//0   1   1

```

```

//0  0  1
//1  0  0
//1  1  0

for (i=0;i<VMHL_N;i++) delete [] a[i];
delete [] a;

```

### 7.18.12 TMHL\_RandomBinaryVector

Функция заполняет вектор (одномерный массив) случайно нулями и единицами.

Код 485. Синтаксис

```
template <class T> void TMHL_RandomBinaryVector(T *VMHL_ResultVector, int VMHL_N);
```

**Входные параметры:**

VMHL\_ResultVector — указатель на преобразуемый массив;

VMHL\_N — размер массива VMHL\_ResultMatrix (число строк).

**Возвращаемое значение:**

Отсутствует.

Код 486. Пример использования

```

int VMHL_N=10; //Размер массива (число строк)
int *a;
a=new int[VMHL_N];

//Вызов функции
TMHL_RandomBinaryVector(a,VMHL_N);

//Используем полученный результат
MHL_ShowVector (a,VMHL_N, "Случайный бинарный вектор", "a");
//Случайный бинарный вектор:
//a =
//1
//1
//0
//0
//0
//0
//0
//1
//1
//0
//0
//0

delete [] a;

```

### 7.18.13 TMHL\_RandomIntMatrix

Функция заполняет матрицу случайными целыми числами из определенного интервала [n;m).

Код 487. Синтаксис

```
template <class T> void TMHL_RandomIntMatrix(T **VMHL_ResultMatrix, T n, T m, int VMHL_N, int VMHL_M);
```

**Входные параметры:**

VMHL\_ResultMatrix — указатель на матрицу;

n — левая граница интервала;

m — правая граница интервала;

VMHL\_N — размер массива (число строк);

VMHL\_M — размер массива (число столбцов).

**Возвращаемое значение:**

Отсутствует.

Код 488. Пример использования

```
int i;
int VMHL_N=3; //Размер массива (число строк)
int VMHL_M=3; //Размер массива (число столбцов)
int **a;
a=new int*[VMHL_N];
for (i=0;i<VMHL_N;i++) a[i]=new int[VMHL_M];

int n=-3; //левая граница интервала;
int m=3; //правая граница интервала;

//Вызов функции
TMHL_RandomIntMatrix(a,n,m,VMHL_N,VMHL_M);

//Используем полученный результат

MHL_ShowMatrix (a,VMHL_N,VMHL_M,"Случайная матрица", "a");
// Случайная матрица:
//a =
// -1  -1  2
// 2   0   1
// -3  2   -1ss

for (i=0;i<VMHL_N;i++) delete [] a[i];
delete [] a;
```

### 7.18.14 TMHL\_RandomIntMatrixInCols

Функция заполняет матрицу случайными целыми числами из определенного интервала [n;m). При этом элементы каждого столбца изменяются в своих пределах.

Код 489. Синтаксис

```
template <class T> void TMHL_RandomIntMatrixInCols(T **VMHL_ResultMatrix, T *n, T *m,
int VMHL_N, int VMHL_M);
```

**Входные параметры:**

VMHL\_ResultMatrix — указатель на матрицу;

n — левые границы интервала изменения элементов столбцов (размер VMHL\_M);

m — правые границы интервала изменения элементов столбцов (размер VMHL\_M);

VMHL\_N — размер массива (число строк);

VMHL\_M — размер массива (число столбцов).

#### **Возвращаемое значение:**

Отсутствует.

Код 490. Пример использования

```
int i;
int VMHL_N=3; //Размер массива (число строк)
int VMHL_M=3; //Размер массива (число столбцов)
int **a;
a=new int*[VMHL_N];
for (i=0;i<VMHL_N;i++) a[i]=new int[VMHL_M];
int *n;
n=new int[VMHL_M];
int *m;
m=new int[VMHL_M];

n[0]=-50; //левая границы интервала изменения 1 столбца
m[0]=-40; //правая граница интервала изменения 1 столбца

n[1]=0; //левая границы интервала изменения 2 столбца
m[1]=3; //правая граница интервала изменения 2 столбца

n[2]=100; //левая границы интервала изменения 3 столбца
m[2]=200; //правая граница интервала изменения 3 столбца

//Вызов функции
TMHL_RandomIntMatrixInCols(a,n,m,VMHL_N,VMHL_M);

//Используем полученный результат

MHL_ShowMatrix (a,VMHL_N,VMHL_M,"Случайная матрица", "a");
//Случайная матрица:
//a =
//-47 2 142
//-47 1 139
//-44 0 199

for (i=0;i<VMHL_N;i++) delete [] a[i];
delete [] a;
delete [] n;
delete [] m;
```

#### **7.18.15 TMHL\_RandomIntMatrixInElements**

Функция заполняет матрицу случайными целыми числами из определенного интервала [n;m). При этом каждый элемент изменяется в своих пределах.

Код 491. Синтаксис

```
template <class T> void TMHL_RandomIntMatrixInElements(T **VMHL_ResultMatrix, T **n,
    T **m, int VMHL_N, int VMHL_M);
```

### Входные параметры:

VMHL\_ResultMatrix — указатель на матрицу;

н — левые границы интервала изменения каждого элемента (размер VMHL\_N x VMHL\_M);

м — правые границы интервала изменения каждого элемента (размер VMHL\_N x VMHL\_M);

VMHL\_N — размер массива (число строк);

VMHL\_M — размер массива (число столбцов).

### Возвращаемое значение:

Отсутствует.

Код 492. Пример использования

```
int i,j;
int VMHL_N=3; //Размер массива (число строк)
int VMHL_M=3; //Размер массива (число столбцов)
int **a;
a=new int*[VMHL_N];
for (i=0;i<VMHL_N;i++) a[i]=new int[VMHL_M];
int **n;
n=new int*[VMHL_N];
for (i=0;i<VMHL_N;i++) n[i]=new int[VMHL_M];
int **m;
m=new int*[VMHL_N];
for (i=0;i<VMHL_N;i++) m[i]=new int[VMHL_M];

//Заполним границы изменения каждого элемента
for (i=0;i<VMHL_N;i++)
    for (j=0;j<VMHL_M;j++)
    {
        n[i][j]=i*VMHL_N+j-10;
        m[i][j]=n[i][j]+20;
    }

//Вызов функции
TMHL_RandomIntMatrixInElements(a,n,m,VMHL_N,VMHL_M);

//Используем полученный результат

MHL_ShowMatrix (n,VMHL_N,VMHL_M,"Матрица левых границ", "n");
//Матрица левых границ:
//n =
// -10 -9 -8
// -7 -6 -5
// -4 -3 -2

MHL_ShowMatrix (m,VMHL_N,VMHL_M,"Матрица правых границ", "m");
// Матрица правых границ:
//m =
// 10 11 12
// 13 14 15
// 16 17 18
```

```

MHL_ShowMatrix (a, VMHL_N, VMHL_M, "Случайная матрица", "a");
// Случайная матрица:
//a =
// -4   6   -8
// -1   1   1
// -3   16  4

for (i=0;i<VMHL_N;i++) delete [] a[i];
delete [] a;
for (i=0;i<VMHL_N;i++) delete [] n[i];
delete [] n;
for (i=0;i<VMHL_N;i++) delete [] m[i];
delete [] m;

```

### 7.18.16 TMHL\_RandomIntMatrixInRows

Функция заполняет матрицу случайными целыми числами из определенного интервала [n;m). При этом элементы каждой строки изменяются в своих пределах.

Код 493. Синтаксис

```

template <class T> void TMHL_RandomIntMatrixInRows(T **VMHL_ResultMatrix, T *n, T *m,
    int VMHL_N, int VMHL_M);

```

#### Входные параметры:

VMHL\_ResultMatrix — указатель на матрицу;

n — левые границы интервала изменения элементов строки (размер VMHL\_N);

m — правые границы интервала изменения элементов строки (размер VMHL\_N);

VMHL\_N — размер массива (число строк);

VMHL\_M — размер массива (число столбцов).

#### Возвращаемое значение:

Отсутствует.

Код 494. Пример использования

```

int i;
int VMHL_N=3; //Размер массива (число строк)
int VMHL_M=3; //Размер массива (число столбцов)
int **a;
a=new int*[VMHL_N];
for (i=0;i<VMHL_N;i++) a[i]=new int[VMHL_M];
int *n;
n=new int[VMHL_N];
int *m;
m=new int[VMHL_N];

n[0]=-50; //левая границы интервала изменения 1 строки
m[0]=-40; //правая граница интервала изменения 1 строки

n[1]=0; //левая границы интервала изменения 2 строки
m[1]=3; //правая граница интервала изменения 2 строки

```

```

n[2]=100; //левая границы интервала изменения 3 строки
m[2]=200; //правая граница интервала изменения 3 строки

//Вызов функции
TMHL_RandomIntMatrixInRows(a,n,m,VMHL_N,VMHL_M);

//Используем полученный результат

MHL_ShowMatrix (a,VMHL_N,VMHL_M,"Случайная матрица", "a");
// Случайная матрица:
//a =
// -42    -42    -45
//2      2      0
//113   102   109

for (i=0;i<VMHL_N;i++) delete [] a[i];
delete [] a;
delete [] n;
delete [] m;

```

### 7.18.17 TMHL\_RandomIntVector

Функция заполняет массив случайными целыми числами из определенного интервала [n,m).

Код 495. Синтаксис

```
template <class T> void TMHL_RandomIntVector(T *VMHL_ResultVector, T n, T m, int VMHL_N);
```

**Входные параметры:**

VMHL\_ResultVector — указатель на массив;

n — левая граница интервала;

m — правая граница интервала;

VMHL\_N — размер массива.

**Возвращаемое значение:**

Отсутствует.

Код 496. Пример использования

```

int VMHL_N=10;//Размер массива
int *a;
a=new int[VMHL_N];

int n=3;
int m=50;

//Вызов функции
TMHL_RandomIntVector(a,n,m,VMHL_N);

//Используем полученный результат

MHL_ShowVector (a,VMHL_N,"Массив", "a");

```

```

//Массив:
//a =
//6
//23
//40
//19
//39
//37
//48
//46
//31
//42

delete [] a;

```

### 7.18.18 TMHL\_RandomIntVectorInElements

Функция заполняет массив случайными целыми числами из определенного интервала  $[n_i, m_i)$ . При этом для каждого элемента массива свой интервал изменения.

Код 497. Синтаксис

```
template <class T> void TMHL_RandomIntVectorInElements(T *VMHL_ResultVector, T *n, T
*m, int VMHL_N);
```

#### Входные параметры:

VMHL\_ResultVector — указатель на массив;

$n$  — указатель на массив левых границ интервала;

$m$  — указатель на массив правых границ интервала;

VMHL\_N — размер массива.

#### Возвращаемое значение:

Отсутствует.

Код 498. Пример использования

```

int VMHL_N=2; //Размер массива
int *a;
a=new int[VMHL_N];

int *n;
n=new int[VMHL_N];
n[0]=3; //Левая граница изменения первого элемента массива
n[1]=-90; //Левая граница изменения второго элемента массива

int *m;
m=new int[VMHL_N];
m[0]=40; //Правая граница изменения первого элемента массива
m[1]=-10; //Правая граница изменения второго элемента массива

//Вызов функции
TMHL_RandomIntVectorInElements(a, n, m, VMHL_N);

//Используем полученный результат

```

```

MHL_ShowVector (n,VMHL_N,"Массив левых границ", "n");
//Массив левых границ:
//n =
//3
// -90

MHL_ShowVector (m,VMHL_N,"Массив правых границ", "m");
// Массив правых границ:
//m =
//40
// -10

MHL_ShowVector (a,VMHL_N,"Случайных массив", "a");
// Случайных массив:
//a =
//31
// -52

delete [] a;
delete [] n;
delete [] m;

```

### 7.18.19 TMHL\_RandomMatrixOfPermutation

Функция создает случайный массив строк-перестановок чисел от 1 до VMHL\_M.

Код 499. Синтаксис

```
template <class T> void TMHL_RandomMatrixOfPermutation(T **VMHL_ResultMatrix, int
VMHL_N, int VMHL_M);
```

**Входные параметры:**

VMHL\_ResultMatrix — указатель на матрицу;

VMHL\_N — размер массива (число строк);

VMHL\_M — размер массива (число столбцов).

**Возвращаемое значение:**

Отсутствует.

**О функции:**

Строка-перестановка — это массив натуральных чисел от 1 до VMHL\_M расположенных в произвольном порядке. Например, если VMHL\_M=10, то строкой-перестановкой будет массив 7 5 2 4 6 8 1 10 3 9. Эти строки используются в комбинаторной оптимизации.

Код 500. Пример использования

```

int i;
int VMHL_N=10; //Размер массива (число строк)
int VMHL_M=5; //Размер массива (число строк)
int **a;
a=new int*[VMHL_N];
for (i=0;i<VMHL_N;i++) a[i]=new int[VMHL_M];

```

```

//Вызов функции
TMHL_RandomMatrixOfPermutation(a,VMHL_N,VMHL_M);

//Используем полученный результат
MHL_ShowMatrix (a,VMHL_N,VMHL_M,"Матрица строк-перестановок", "а");
//Матрица строк-перестановок:
//a =
//3   2   1   4   5
//4   1   3   2   5
//5   2   3   4   1
//5   3   4   2   1
//5   4   2   1   3
//1   4   3   5   2
//5   4   1   3   2
//1   4   2   5   3
//3   1   2   4   5
//5   3   4   2   1

for (i=0;i<VMHL_N;i++) delete [] a[i];
delete [] a;

```

### 7.18.20 TMHL\_RandomVectorOfPermutation

Функция создает случайную строку-перестановку чисел от 1 до VMHL\_N (включительно).

Код 501. Синтаксис

```

template <class T> void TMHL_RandomVectorOfPermutation(T *VMHL_ResultVector, int
VMHL_N);

```

**Входные параметры:**

VMHL\_ResultVector — указатель на массив;

VMHL\_N — размер массива.

**Возвращаемое значение:**

Отсутствует.

**О функции:**

Строка-перестановка — это массив натуральных чисел от 1 до VMHL\_N расположенных в произвольном порядке. Например, если VMHL\_N=10, то строкой-перестановкой будет массив 7 5 2 4 6 8 1 10 3 9. Эти строки используются в комбинаторной оптимизации.

Код 502. Пример использования

```

int VMHL_N=10;//Размер массива (число строк)
double *a;
a=new double[VMHL_N];

//Вызов функции
TMHL_RandomVectorOfPermutation(a,VMHL_N);

//Используем полученный результат
MHL_ShowVector (a,VMHL_N,"Строка-перестановка", "а");
//Строка-перестановка:

```

```
//a =  
//1  
//4  
//8  
//7  
//9  
//10  
//5  
//3  
//2  
//6  
  
delete [] a;
```

## 7.19 Случайные числа

### 7.19.1 MHL\_RandomNormal

Случайное число по нормальному закону распределения.

Код 503. Синтаксис

```
double MHL_RandomNormal(double Mean, double StdDev);
```

**Входные параметры:**

Mean — математическое ожидание;

StdDev — среднеквадратичное отклонение.

**Возвращаемое значение:** Случайное число по нормальному закону.

Код 504. Пример использования

```
double x;  
double Mean=10; //математическое ожидание  
double StdDev=3; //среднеквадратичное отклонение  
  
//Вызов функции  
x=MHL_RandomNormal(Mean, StdDev);  
  
//Используем полученный результат  
MHL_ShowNumber(x, "Случайное число по нормальному закону (Mean="+MHL_NumberToText(Mean)  
    +", StdDev="+MHL_NumberToText(StdDev)+")", "x");  
//Случайное число по нормальному закону (Mean=10, StdDev=3):  
//x=10.9968
```

### 7.19.2 MHL\_RandomUniform

Случайное вещественное число в интервале [a;b] по равномерному закону распределения.

Код 505. Синтаксис

```
double MHL_RandomUniform(double a, double b);
```

### **Входные параметры:**

a — левая граница;  
b — правая граница.

**Возвращаемое значение:** Случайное вещественное число в интервале [a;b].

Код 506. Пример использования

```
double x;

//Вызов функции
x=MHL_RandomUniform(10,100);

//Используем полученный результат
MHL_ShowNumber(x,"Случайное число из интервала [10;100]", "x");
//Случайное числ
```

### **7.19.3 MHL\_RandomUniformInt**

Случайное целое число в интервале [n,m) по равномерному закону распределения.

Код 507. Синтаксис

```
int MHL_RandomUniformInt(int n, int m);
```

### **Входные параметры:**

n — левая граница;  
m — правая граница.

**Возвращаемое значение:**

Случайное целое число от  $n$  до  $m - 1$  включительно.

Код 508. Пример использования

```
double x;
int s0=0,s1=0,s2=0,s3=0;

//Вызов функции
for (int i=0;i<1000;i++)
{
x=MHL_RandomUniformInt(0,3);
if (x==0) s0++;
if (x==1) s1++;
if (x==2) s2++;
if (x==3) s3++;
}

//Используем полученный результат
MHL_ShowNumber(x,"Случайное целое число из интервала [0;3]", "x");
MHL_ShowNumber(s0,"Число выпадений 0", "s0");
MHL_ShowNumber(s1,"Число выпадений 1", "s0");
MHL_ShowNumber(s2,"Число выпадений 2", "s0");
MHL_ShowNumber(s3,"Число выпадений 3", "s0");
//Случайное целое число из интервала [0;3]:
//x=1
```

```
//Число выпадений 0:  
//s0=324  
//Число выпадений 1:  
//s0=374  
//Число выпадений 2:  
//s0=302  
//Число выпадений 3:  
//s0=0
```

#### 7.19.4 MHL\_RandomUniformIntIncluding

Случайное целое число в интервале  $[n,m]$  по равномерному закону распределения.

Код 509. Синтаксис

```
int MHL_RandomUniformIntIncluding(int n, int m);
```

**Входные параметры:**

$n$  — левая граница;

$m$  — правая граница.

**Возвращаемое значение:**

Случайное целое число от  $n$  до  $m$  включительно.

**Примечание:**

В отличии от функции MHL\_RandomUniformInt правая граница тоже включается, то есть может сгенерироваться  $m$ , а не  $m - 1$ .

Код 510. Пример использования

```
double x;  
int s0=0,s1=0,s2=0,s3=0;  
  
//Вызов функции  
for (int i=0;i<1000;i++)  
{  
x=MHL_RandomUniformIntIncluding(0,3);  
if (x==0) s0++;  
if (x==1) s1++;  
if (x==2) s2++;  
if (x==3) s3++;  
}  
  
//Используем полученный результат  
MHL_ShowNumber(x,"Случайное целое число из интервала [0;3]","x");  
MHL_ShowNumber(s0,"Число выпадений 0","s0");  
MHL_ShowNumber(s1,"Число выпадений 1","s0");  
MHL_ShowNumber(s2,"Число выпадений 2","s0");  
MHL_ShowNumber(s3,"Число выпадений 3","s0");  
//Случайное целое число из интервала [0;3]:  
//x=1  
//Число выпадений 0:  
//s0=324  
//Число выпадений 1:  
//s0=374
```

```
//Число выпадений 2:  
//s0=302  
//Число выпадений 3:  
//s0=0
```

## 7.20 Сортировка

### 7.20.1 TMHL\_BubbleDescendingSort

Функция сортирует массив в порядке убывания методом «Сортировка пузырьком».

Код 511. Синтаксис

```
template <class T> void TMHL_BubbleDescendingSort(T *VMHL_ResultVector, int VMHL_N);
```

**Входные параметры:**

VMHL\_ResultVector — указатель на исходный массив;

VMHL\_N — количество элементов в массиве.

**Возвращаемое значение:**

Отсутствует.

Код 512. Пример использования

```
int i;  
int VMHL_N=10; //Размер массива (число строк)  
double *a;  
a=new double[VMHL_N];  
for (i=0;i<VMHL_N;i++)  
    a[i]=MHL_RandomNumber();  
  
MHL_ShowVector (a,VMHL_N, "Случайный вектор", "a");  
// Например  
// Случайный вектор:  
//Случайный вектор:  
//a =  
//0.233978  
//0.29541  
//0.142914  
//0.719482  
//0.489319  
//0.610382  
//0.667908  
//0.596069  
//0.92099  
//0.88327  
  
//Вызов функции  
TMHL_BubbleDescendingSort(a,VMHL_N);  
  
//Используем полученный результат  
MHL_ShowVector (a,VMHL_N, "Отсортированный вектор", "a");  
//Отсортированный вектор:  
//a =  
//0.92099
```

```

//0.88327
//0.719482
//0.667908
//0.610382
//0.596069
//0.489319
//0.29541
//0.233978
//0.142914

delete [] a;

```

## 7.20.2 TMHL\_BubbleSort

Функция сортирует массив в порядке возрастания методом «Сортировка пузырьком».

Код 513. Синтаксис

```
template <class T> void TMHL_BubbleSort(T *VMHL_ResultVector, int VMHL_N);
```

**Входные параметры:**

VMHL\_ResultVector — указатель на исходный массив;

VMHL\_N — количество элементов в массиве.

**Возвращаемое значение:**

Отсутствует.

Код 514. Пример использования

```

int i;
int VMHL_N=10; //Размер массива (число строк)
double *a;
a=new double[VMHL_N];
for (i=0;i<VMHL_N;i++)
a[i]=MHL_RandomNumber();

MHL_ShowVector (a,VMHL_N, "Случайный вектор", "a");
// Например
//Случайный вектор:
//a =
//0.889862
//0.575836
//0.741882
//0.0479736
//0.788879
//0.873413
//0.343933
//0.32196
//0.0332031
//0.0214844

//Вызов функции
TMHL_BubbleSort(a,VMHL_N);

//Используем полученный результат
MHL_ShowVector (a,VMHL_N, "Отсортированный вектор", "a");

```

```

// Отсортированный вектор:
//a =
//0.0214844
//0.0332031
//0.0479736
//0.32196
//0.343933
//0.575836
//0.741882
//0.788879
//0.873413
//0.889862

delete [] a;

```

### 7.20.3 TMHL\_BubbleSortColWithOtherConjugateColsInMatrix

Функция сортирует матрицу по какому-то столбцу под номером в порядке возрастания методом «Сортировка пузырьком». При этом все остальные столбцы являются как бы сопряженным с данным столбцом. То есть элементы в этом столбце сортируются, а все строки остаются прежними.

Код 515. Синтаксис

```

template <class T> void TMHL_BubbleSortColWithOtherConjugateColsInMatrix(T **  
VMHL_ResultMatrix, int Col, int VMHL_N, int VMHL_M);

```

#### Входные параметры:

VMHL\_ResultMatrix — указатель на матрицу, которую будем сортировать;

Col — номер сортируемого столбца в матрице;

VMHL\_N — количество строк в матрице;

VMHL\_M — количество столбцов в матрице.

#### Возвращаемое значение:

Отсутствует.

Код 516. Пример использования

```

int i;
int VMHL_N=5; //Размер массива (число строк)
int VMHL_M=3; //Размер массива (число столбцов)
int **a;
a=new int*[VMHL_N];
for (i=0;i<VMHL_N;i++) a[i]=new int[VMHL_M];

TMHL_RandomIntMatrix(a,0,5,VMHL_N,VMHL_M);

MHL_ShowMatrix (a,VMHL_N,VMHL_M,"Случайная матрица", "a");
//Случайная матрица:
//a =
//4 0 1
//4 0 4
//2 2 0

```

```

//2   3   1
//1   3   1

int Col=0; //Будем сортировать столбец под номером 2

//Вызов функции

TMHL_BubbleSortColWithOtherConjugateColsInMatrix(a, Col, VMHL_N, VMHL_M);

//Используем полученный результат
MHL_ShowMatrix (a,VMHL_N,VMHL_M, "Случайная матрица отсортированная по столбцу с номером " + MHL_NumberToText(Col), "a");
//Случайная матрица отсортированная по столбцу с номером 0:
//a =
//1   3   1
//2   2   0
//2   3   1
//4   0   1
//4   0   4

for (i=0;i<VMHL_N;i++) delete [] a[i];
delete [] a;

```

#### 7.20.4 TMHL\_BubbleSortEveryColInMatrix

Функция сортирует каждый столбец матрицы в отдельности.

Код 517. Синтаксис

```
template <class T> void TMHL_BubbleSortEveryColInMatrix(T **VMHL_ResultMatrix, int VMHL_N, int VMHL_M);
```

**Входные параметры:**

VMHL\_ResultMatrix — указатель на матрицу, которую будем сортировать;

VMHL\_N — количество строк в матрице;

VMHL\_M — количество столбцов в матрице.

**Возвращаемое значение:**

Отсутствует.

Код 518. Пример использования

```

int i;
int VMHL_N=5; //Размер массива (число строк)
int VMHL_M=6; //Размер массива (число столбцов)
int **a;
a=new int*[VMHL_N];
for (i=0;i<VMHL_N;i++) a[i]=new int[VMHL_M];

TMHL_RandomIntMatrix(a,0,5,VMHL_N,VMHL_M);

MHL_ShowMatrix (a,VMHL_N,VMHL_M, "Случайная матрица", "a");
//Случайная матрица:
//a =
//4   1   4   3   0   4

```

```

//2   1   1   0   3
//0   4   2   2   0   3
//1   2   2   2   4   0
//3   0   2   4   1   4

//Вызов функции
TMHL_BubbleSortEveryColInMatrix(a, VMHL_N, VMHL_M);

//Используем полученный результат
MHL_ShowMatrix (a,VMHL_N,VMHL_M,"Случайная матрица, где каждый столбец отсортирован независимо", "a");
//Случайная матрица, где каждый столбец отсортирован независимо:
//a =
//0   0   1   1   0   0
//1   1   2   2   0   3
//2   1   2   2   0   3
//3   2   2   3   1   4
//4   4   4   4   4   4

for (i=0;i<VMHL_N;i++) delete [] a[i];
delete [] a;

```

## 7.20.5 TMHL\_BubbleSortEveryRowInMatrix

Функция сортирует каждую строку матрицы в отдельности.

Код 519. Синтаксис

```
template <class T> void TMHL_BubbleSortEveryRowInMatrix(T **VMHL_ResultMatrix, int
VMHL_N, int VMHL_M);
```

**Входные параметры:**

VMHL\_ResultMatrix — указатель на матрицу, которую будем сортировать;

VMHL\_N — количество строк в матрице;

VMHL\_M — количество столбцов в матрице.

**Возвращаемое значение:**

Отсутствует.

Код 520. Пример использования

```

int i;
int VMHL_N=5;//Размер массива (число строк)
int VMHL_M=6;//Размер массива (число столбцов)
int **a;
a=new int*[VMHL_N];
for (i=0;i<VMHL_N;i++) a[i]=new int[VMHL_M];

TMHL_RandomIntMatrix(a,0,5,VMHL_N,VMHL_M);

MHL_ShowMatrix (a,VMHL_N,VMHL_M,"Случайная матрица", "a");
//Случайная матрица:
//a =
//3   1   2   1   1   2
//0   1   4   0   2   1

```

```

//4   4   3   2   1
//1   3   0   3   4   0
//2   3   1   1   2   3

//Вызов функции
TMHL_BubbleSortEveryRowInMatrix(a, VMHL_N, VMHL_M);

//Используем полученный результат
MHL_ShowMatrix (a,VMHL_N,VMHL_M, "Случайная матрица, где каждая строка отсортиро-
вана независимо", "a");
//Случайная матрица, где каждая отсортирована независимо:
//a =
//1   1   2   2   3
//0   0   1   1   2   4
//1   2   3   4   4   4
//0   0   1   3   3   4
//1   1   2   2   3   3

for (i=0;i<VMHL_N;i++) delete [] a[i];
delete [] a;

```

## 7.20.6 TMHL\_BubbleSortInGroups

Функция сортирует массив в порядке возрастания методом «Сортировка пузырьком» в группах данного массива. Имеется массив. Он делится на группы элементов по  $m$  элементов. Первые  $m$  элементов принадлежат первой группе, следующие  $m$  элементов — следующей и т.д. (Разумеется, в последней группе может и не оказаться  $m$  элементов). Потом в каждой группе элементы сортируются по возрастанию.

Код 521. Синтаксис

```
template <class T> void TMHL_BubbleSortInGroups(T *VMHL_ResultVector, int VMHL_N, int
m);
```

### Входные параметры:

VMHL\_ResultVector — указатель на исходный массив;

VMHL\_N — количество элементов в массиве;

$m$  — количество элементов в группе.

### Возвращаемое значение:

Отсутствует.

Код 522. Пример использования

```

int i;
int VMHL_N=9; //Размер массива (число строк)
double *a;
a=new double[VMHL_N];
for (i=0;i<VMHL_N;i++)
a[i]=MHL_RandomUniformInt(10,50);

// Например
MHL_ShowVectorT (a,VMHL_N, "Случайный вектор", "a");

```

```

//Случайный вектор:
//a =
//20 42 39 19 27 33 35 44 32

int m=3;

//Вызов функции
TMHL_BubbleSortInGroups(a,VMHL_N,m);

//Используем полученный результат
MHL_ShowVectorT (a,VMHL_N,"Отсортированный вектор по три элемента", "a");
//Отсортированный вектор по три элемента:
//a =
//20 39 42 19 27 33 32 35 44

delete [] a;

```

### 7.20.7 TMHL\_BubbleSortRowWithOtherConjugateRowsInMatrix

Функция сортирует матрицу по какой-то строке под номером в порядке возрастания методом «Сортировка пузырьком». При этом все остальные строки являются как бы сопряженными с данной строкой. То есть элементы в этой строке сортируются, а все столбцы остаются прежними.

Код 523. Синтаксис

```

template <class T> void TMHL_BubbleSortRowWithOtherConjugateRowsInMatrix(T **
VMHL_ResultMatrix,int Row, int VMHL_N, int VMHL_M);

```

**Входные параметры:**

VMHL\_ResultMatrix — указатель на матрицу, которую будем сортировать;

Row — номер сортируемой строки в матрице;

VMHL\_N — количество строк в матрице;

VMHL\_M — количество столбцов в матрице.

**Возвращаемое значение:**

Отсутствует.

Код 524. Пример использования

```

int i;
int VMHL_N=5;//Размер массива (число строк)
int VMHL_M=5;//Размер массива (число столбцов)
int **a;
a=new int*[VMHL_N];
for (i=0;i<VMHL_N;i++) a[i]=new int[VMHL_M];

TMHL_RandomIntMatrix(a,0,5,VMHL_N,VMHL_M);

MHL_ShowMatrix (a,VMHL_N,VMHL_M,"Случайная матрица", "a");
//Случайная матрица:
//a =
//0 1 2 3

```

```

//1  2  1  4  1
//3  1  2  0  1
//3  4  1  0  0
//4  4  1  0  2

int Row=2; //Будем сортировать строку под номером 2

//Вызов функции

TMHL_BubbleSortRowWithOtherConjugateRowsInMatrix(a, Row, VMHL_N, VMHL_M);

//Используем полученный результат
MHL_ShowMatrix (a,VMHL_N,VMHL_M, "Случайная матрица отсортированная по строке с номером "+MHL_NumberToText(Row), "a");
//Случайная матрица отсортированная по строке с номером 2:
//a =
//2  0  3  1  0
//4  2  1  1  1
//0  1  1  2  3
//0  4  0  1  3
//0  4  2  1  4

for (i=0;i<VMHL_N;i++) delete [] a[i];
delete [] a;

```

## 7.20.8 TMHL\_BubbleSortWithConjugateVector

Функция сортирует массив вместе с сопряженным массивом в порядке возрастания методом «Сортировка пузырьком». Пары элементов первого массива и сопряженного остаются без изменения.

Код 525. Синтаксис

```
template <class T, class T2> void TMHL_BubbleSortWithConjugateVector(T * VMHL_ResultVector, T2 *VMHL_ResultVector2, int VMHL_N);
```

### Входные параметры:

VMHL\_ResultVector — указатель на исходный массив;

VMHL\_ResultVector2 — указатель на сопряженный массив;

VMHL\_N — количество элементов в массиве.

### Возвращаемое значение:

Отсутствует.

Код 526. Пример использования

```
int i;
int VMHL_N=10; //Размер массива (число строк)
double *a;
a=new double[VMHL_N];
int *b;
b=new int[VMHL_N];
for (i=0;i<VMHL_N;i++)
{
```

```

a[i]=MHL_RandomUniformInt(10,50);
b[i]=MHL_RandomUniformInt(10,50);
}

// Например
MHL_ShowVectorT (a,VMHL_N,"Случайный вектор", "a");
// Случайный вектор:
//a =
//31 32 13 26 40 40 47 26 10 18

MHL_ShowVectorT (b,VMHL_N,"Сопряженный вектор", "b");
//Сопряженный вектор:
//b =
//31 20 44 32 21 36 46 30 31 15

//Вызов функции
TMHL_BubbleSortWithConjugateVector(a,b,VMHL_N);

//Используем полученный результат
MHL_ShowVectorT (a,VMHL_N,"Отсортированный вектор", "a");
// Отсортированный вектор:
//a =
//10 13 18 26 26 31 32 40 40 47

MHL_ShowVectorT (b,VMHL_N,"Сопряженный вектор", "b");
// Сопряженный вектор:
//b =
//31 44 15 32 30 31 20 21 36 46

delete [] a;
delete [] b;

```

### 7.20.9 TMHL\_BubbleSortWithTwoConjugateVectors

Функция сортирует массив вместе с двумя сопряженными массивами в порядке возрастания методом «Сортировка пузырьком». Пары элементов первого массива и сопряженного остаются без изменения.

Код 527. Синтаксис

```
template <class T, class T2, class T3> void TMHL_BubbleSortWithTwoConjugateVectors(T
    *VMHL_ResultVector, T2 *VMHL_ResultVector2, T3 *VMHL_ResultVector3, int VMHL_N);
```

#### Входные параметры:

VMHL\_ResultVector — указатель на исходный массив;

VMHL\_ResultVector2 — указатель на сопряженный массив;

VMHL\_ResultVector3 — указатель на второй сопряженный массив;

VMHL\_N — количество элементов в массивах.

#### Возвращаемое значение:

Отсутствует.

Код 528. Пример использования

```

int i;
int VMHL_N=10; //Размер массива (число строк)
double *a;
a=new double[VMHL_N];
int *b;
b=new int[VMHL_N];
int *c;
c=new int[VMHL_N];
for (i=0;i<VMHL_N;i++)
{
    a[i]=MHL_RandomUniformInt(10,50);
    b[i]=MHL_RandomUniformInt(10,50);
    c[i]=MHL_RandomUniformInt(10,50);
}

// Например
MHL_ShowVectorT (a,VMHL_N,"Случайный вектор", "a");
//Случайный вектор:
//a =
//45 27 11 18 24 25 16 19 34 43

MHL_ShowVectorT (b,VMHL_N,"Сопряженный вектор", "b");
//Сопряженный вектор:
//b =
//33 32 24 33 32 49 33 43 25 47

MHL_ShowVectorT (c,VMHL_N,"Сопряженный вектор", "c");
//Сопряженный вектор:
//c =
//15 24 27 43 17 47 25 11 13 26

//Вызов функции
TMHL_BubbleSortWithTwoConjugateVectors(a,b,c,VMHL_N);

//Используем полученный результат
MHL_ShowVectorT (a,VMHL_N,"Отсортированный вектор", "a");
//Отсортированный вектор:
//a =
//11 16 18 19 24 25 27 34 43 45

MHL_ShowVectorT (b,VMHL_N,"Сопряженный вектор", "b");
// Сопряженный вектор:
//b =
//24 33 33 43 32 49 32 25 47 33

MHL_ShowVectorT (c,VMHL_N,"Второй сопряженный вектор", "c");
//Второй сопряженный вектор:
//c =
//27 25 43 11 17 47 24 13 26 15

delete [] a;
delete [] b;
delete [] c;

```

## 7.21 Статистика и теория вероятности

### 7.21.1 MHL\_DensityOfDistributionOfNormalizedCenteredNormalDistribution

Плотность распределения вероятности нормированного и центрированного нормального распределения.

Код 529. Синтаксис

```
double MHL_DensityOfDistributionOfNormalizedCenteredNormalDistribution(double x);
```

**Входные параметры:**

$x$  — входная переменная.

**Возвращаемое значение:**

Значение функции в точке.

**Формула:**

$$F(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}.$$

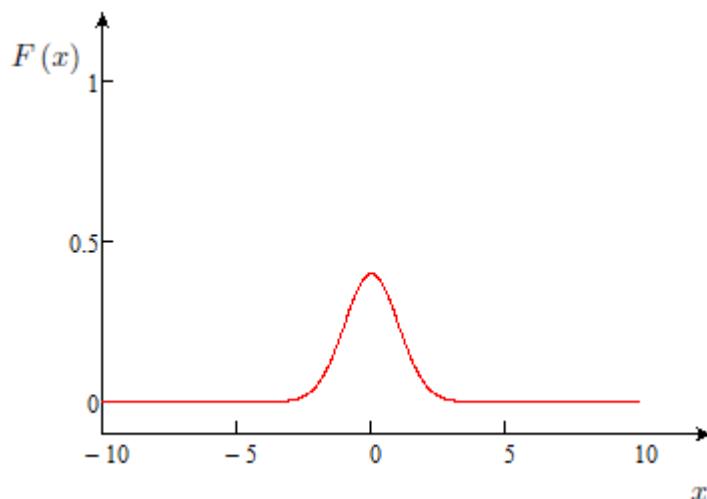


Рисунок 28. График функции

Код 530. Пример использования

```
double t;
double f;
t=MHL_RandomUniform(0,3);

//Вызов функции
f=MHL_DensityOfDistributionOfNormalizedCenteredNormalDistribution(t);

//Используем полученный результат

MHL>ShowNumber (t,"Параметр", "t");
// Параметр:
//t=1.42401
MHL>ShowNumber (f,"Значение функции", "f");
// Значение функции:
```

```
//f=0.144736
```

### 7.21.2 MHL\_DistributionFunctionOfNormalDistribution

Функция распределения нормального распределения.

Код 531. Синтаксис

```
double MHL_DistributionFunctionOfNormalDistribution(double x, double mu, double sigma  
, double Epsilon);
```

**Входные параметры:**

x — входная переменная (правая граница интегрирования);

mu — математическое ожидание;

sigma — стандартное отклонение ( $\sigma > 0$ );

Epsilon — погрешность (например,  $Epsilon = 0.00001$  — больше не берите, а то будет большая погрешность).

**Возвращаемое значение:**

Значение функции в точке.

Код 532. Пример использования

```
double x;  
double f;  
x=MHL_RandomUniform(0,3);  
double mu=3;  
double sigma=1;  
  
//Вызов функции  
f=MHL_DistributionFunctionOfNormalizedCenteredNormalDistribution(x,0.001);  
  
//Используем полученный результат  
MHL_ShowNumber (x,"Входная переменная", "x");  
//Входная переменная:  
//x=0.527979  
MHL_ShowNumber (mu,"Параметр mu", "mu");  
//Параметр mu:  
//mu=3  
MHL_ShowNumber (sigma,"Параметр sigma", "sigma");  
//Параметр sigma:  
//sigma=1  
MHL_ShowNumber (f,"Значение функции распределения нормального распределения", "f");  
//Значение функции распределения нормального распределения:  
//f=0.701244
```

### 7.21.3 MHL\_DistributionFunctionOfNormalizedCenteredNormalDistribution

Функция распределения нормированного и центрированного нормального распределения.

Код 533. Синтаксис

```
double MHL_DistributionFunctionOfNormalizedCenteredNormalDistribution(double x,
    double Epsilon);
```

**Входные параметры:**

x — входная переменная (правая граница интегрирования);

Epsilon — погрешность (например, Epsilon=0.00001).

**Возвращаемое значение:**

Значение функции в точке.

**Формула:**

$$F(x) = \frac{1}{\sqrt{2\pi}} \int_0^x e^{-\frac{x^2}{2}} dx + 0.5, \text{ если } x \geq 0.$$

$$F(x) = \frac{1}{\sqrt{2\pi}} \int_0^{-x} e^{-\frac{-x^2}{2}} dx + 0.5, \text{ если } x < 0.$$

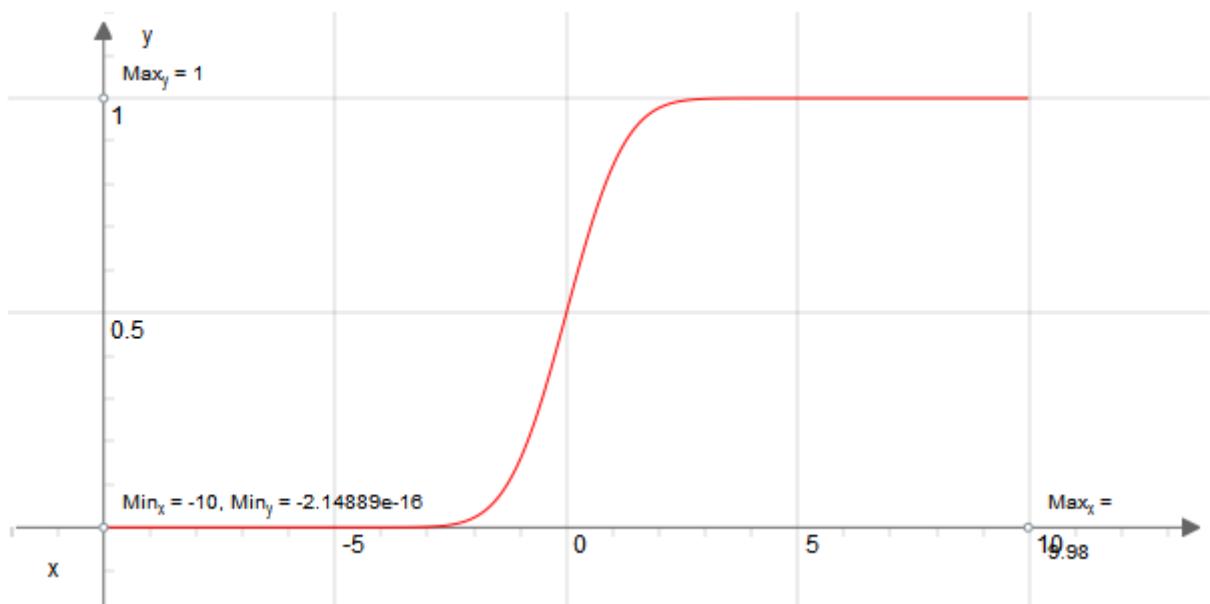


Рисунок 29. График функции

Код 534. Пример использования

```
double t;
double f;
t=MHL_RandomUniform(0,3);

//Вызов функции
f=MHL_DistributionFunctionOfNormalizedCenteredNormalDistribution(t,0.001);

//Используем полученный результат

MHL>ShowNumber (t,"Параметр", "t");
//Параметр:
//t=1.36576
MHL>ShowNumber (f,"Значение функции", "f");
```

```
//Значение функции:  
//f=0.914011
```

#### 7.21.4 MHL\_LeftBorderOfWilcoxonWFromTable

Функция возвращает левую границу интервала критический значений статистики W для критерия Вилкоксена по табличным данным.

Код 535. Синтаксис

```
double MHL_LeftBorderOfWilcoxonWFromTable(int m, int n, double Q);
```

**Входные параметры:**

m — объем первой выборки (не больше 25);

n — объем второй выборки (не больше 25);

Q — уровень значимости. Может принимать значения:

- 0.001;
- 0.005;
- 0.01;
- 0.025;
- 0.05;
- 0.1.

**Возвращаемое значение:**

Левая граница интервала критический значений статистики W для критерия Вилкоксена.

**Примечание:**

Если размеры выборок не из таблицы, если не правильный выбран уровень значимости, то возвратится -1.

Табличные значения взяты из справочника «Таблицы математической статистики» [1, с. 357]. Описание по данной книге можно прочитать на <https://github.com/Harrix/Wilcoxon-W-Test>.

Код 536. Пример использования

```
int m=20;  
MHL_ShowNumber(m, "Объем меньшей выборки", "m");  
  
int n=21;  
MHL_ShowNumber(n, "Объем большей выборки", "n");  
  
double Q=0.05;  
MHL_ShowNumber(Q, "Уровень значимости", "Q");  
  
//Вызов функции  
double Left=MHL_LeftBorderOfWilcoxonWFromTable(m, n, Q);
```

```

//Использование результата
MHL_ShowNumber(Left, "Левая граница интервала критический значений статистики W для критерия Вилкоксена", "Left");
//Левая граница интервала критический значений статистики W для критерия Вилкоксена:
//Left=356

```

### 7.21.5 MHL\_RightBorderOfWilcoxonWFromTable

Функция возвращает правую границу интервала критический значений статистики W для критерия Вилкоксена по табличным данным.

Код 537. Синтаксис

```
double MHL_RightBorderOfWilcoxonWFromTable(int m, int n, double Q);
```

**Входные параметры:**

m — объем первой выборки (не больше 25);

n — объем второй выборки (не больше 25);

Q — уровень значимости. Может принимать значения:

- 0.001;
- 0.005;
- 0.01;
- 0.025;
- 0.05;
- 0.1.

**Возвращаемое значение:**

Правая граница интервала критический значений статистики W для критерия Вилкоксена.

**Примечание:**

Если размеры выборок не из таблицы, если не правильный выбран уровень значимости, то возвратится -1.

Табличные значения взяты из справочника «Таблицы математической статистики» [1, с. 357]. Описание по данной книге можно прочитать на <https://github.com/Harrix/Wilcoxon-W-Test>.

Код 538. Пример использования

```

int m=20;
MHL_ShowNumber(m, "Объем меньшей выборки", "m");

int n=21;
MHL_ShowNumber(n, "Объем большей выборки", "n");

double Q=0.05;

```

```

MHL_ShowNumber(Q, "Уровень значимости", "Q");

//Вызов функции
double Right=MHL_RightBorderOfWilcoxonWFromTable(m,n,Q);

//Использование результата
MHL_ShowNumber(Right, "Правая граница интервала критический значений статистики W для
    критерия Вилкоксена", "Right");
//Правая граница интервала критический значений статистики W для критерия Вилкоксена:
//Right=484

```

### 7.21.6 MHL\_StdDevToVariance

Функция переводит среднеквадратичное уклонение в значение дисперсии случайной величины.

Код 539. Синтаксис

```
double MHL_StdDevToVariance(double StdDev);
```

**Входные параметры:**

StdDev — среднеквадратичное уклонение.

**Возвращаемое значение:**

Значение дисперсии случайной величины.

Код 540. Пример использования

```

double Variance;
double StdDev=6;

//Вызов функции
Variance=MHL_StdDevToVariance(StdDev);

//Используем результат
MHL_ShowNumber(Variance, "Дисперсия при среднеквадратичном уклонении, равным "+
    MHL_NumberToText(StdDev), "равна");
//Дисперсия при среднеквадратичном уклонении, равным 6:
//равна=2.44949

```

### 7.21.7 MHL\_VarianceToStdDev

Функция переводит значение дисперсии случайной величины в среднеквадратичное уклонение.

Код 541. Синтаксис

```
double MHL_VarianceToStdDev(double Variance);
```

**Входные параметры:**

Variance — значение дисперсии случайной величины.

**Возвращаемое значение:**

Значение среднеквадратичного уклонения.

Код 542. Пример использования

```
double StdDev;
double Variance=6;

//Вызов функции
StdDev=MHL_VarianceToStdDev(Variance);

//Используем полученный результат
MHL_ShowNumber(StdDev, "Среднеквадратичное уклонение при дисперсии, равной "+  
    MHL_NumberToText(Variance), "равно");
//Среднеквадратичное уклонение при дисперсии, равной 6:  
//равно=36
```

### 7.21.8 MHL\_WilcoxonW

Функция проверяет однородность выборок по критерию Вилкосена W.

Код 543. Синтаксис

```
int MHL_WilcoxonW(double *a, double *b, int VMHL_N1, int VMHL_N2, double Q);
```

#### Входные параметры:

a — первая выборка;

b — вторая выборка;

VMHL\_N1 — размер первой выборки;

VMHL\_N2 — размер второй выборки;

Q — уровень значимости. Может принимать значения:

- 0.002;
- 0.01;
- 0.02;
- 0.05;
- 0.1;
- 0.2.

#### Возвращаемое значение:

-2 — уровень значимости выбран неправильно (не из допустимого множества);

-1 — объемы выборок не позволяют провести проверку при данном уровне значимости (или они не положительные);

0 — выборки не однородны при данном уровне значимости;

1 — выборки однородны при данном уровне значимости;

### Примечание:

Если размеры выборок не из таблицы, если не правильный выбран уровень значимости, то возвратится -1.

Обратите внимание, что допустимые значения значимости Q в функциях MHL\_LeftBorderOfWilcoxonWFromTable и MHL\_RightBorderOfWilcoxonWFromTable в два раза меньше. Это связано с тем, что критерий Вилкосена использует двухсторонний критерий. Поэтому уровень значимости должен быть повышен, по сравнению с табличными значениями.

Информация о критерии из справочника «Таблицы математической статистики» [1, с. 93]. Описание по данной книге можно прочитать на <https://github.com/Harrix/Wilcoxon-W-Test>.

Код 544. Пример использования

```
int VMHL_Result;

int VMHL_N1=10;
int VMHL_N2=10;

double *a = new double[VMHL_N1];
double *b = new double[VMHL_N2];
TMHL_RandomIntVector(a,0.,10.,VMHL_N1);
TMHL_RandomIntVector(b,0.,10.,VMHL_N2);
MHL_ShowVectorT(a,VMHL_N1,"Первая выборка", "a");
//Первая выборка:
//a =
//6  0  6  1  4  9  6  2  4  8

MHL_ShowVectorT(b,VMHL_N2,"Вторая выборка", "b");
//Вторая выборка:
//b =
//8  1  1  6  0  3  1  1  2  3

double Q=0.002;
//Q=0.2;
MHL>ShowNumber(Q,"Уровень значимости","Q");
//Уровень значимости:
//Q=0.002
//Вызов функции
VMHL_Result = MHL_WilcoxonW(a, b, VMHL_N1, VMHL_N2, Q);

//Используем результат
MHL>ShowNumber(VMHL_Result,"Итог проверка двух выборок критерием Вилкосена W","VMHL_Result");
//Итог проверка двух выборок критерием Вилкосена W:
//VMHL_Result=1

delete [] a;
delete [] b;
```

### 7.21.9 TMHL\_Mean

Функция вычисляет среднее арифметическое массива.

Код 545. Синтаксис

```
template <class T> T TMHL_Mean(T *x, int VMHL_N);
```

### **Входные параметры:**

x — массив;

VMHL\_N — размер массива.

### **Возвращаемое значение:**

Среднее арифметическое массива.

### **Примечание:**

Если будете считать для массива \*int, то ответ будет некорректным, так как ответ возвратится тоже в виде int.

#### Код 546. Пример использования

```

int i;
int VMHL_N=10; //Размер массива
double *a;
a=new double[VMHL_N];
//Заполним случайными числами
for (i=0;i<VMHL_N;i++)
    a[i]=MHL_RandomUniform(0,10);

//Вызов функции
double Mean=TMHL_Mean(a,VMHL_N);

//Используем полученный результат
MHL_ShowVector (a,VMHL_N, "Массив", "a");
// Массив:
//a =
//4.65149
//4.00574
//1.41113
//1.55457
//2.75055
//3.16559
//8.26508
//3.86902
//9.5401
//4.50836

MHL_ShowNumber (Mean, "Среднее арифметическое массива", "Mean");
//Среднее арифметическое массива:
//Mean=4.37216

delete [] a;

```

### **7.21.10 TMHL\_Median**

Функция вычисляет медиану выборки.

#### Код 547. Синтаксис

```
template <class T> T TMHL_Median(T *x, int VMHL_N);
```

### **Входные параметры:**

`x` — массив;

`VMHL_N` — размер массива.

**Возвращаемое значение:**

Медиана массива.

**О функции:**

Медиана (50-й процентиль, квантиль 0,5) — возможное значение признака, которое делит ранжированную совокупность (вариационный ряд выборки) на две равные части: 50

В случае, когда число элементов в выборке нечетно, то медиана равна элементу выборки посередине отсортированного массива.

В случае, когда число элементов в выборке четно, то медиана равна среднеарифметическому двух элементов выборки посередине отсортированного массива.

Код 548. Пример использования

```
int i;
int VMHL_N=MHL_RandomUniformInt(3,10); //Размер массива
double *a;
a=new double[VMHL_N];
//Заполним случайными числами
for (i=0;i<VMHL_N;i++)
    a[i]=MHL_RandomUniform(0,10);

//Вызов функции
double Median=TMHL_Median(a,VMHL_N);

//Используем полученный результат
MHL_ShowVector (a,VMHL_N, "Массив", "a");
//Массив:
//a =
//8.77167
//5.89142
//6.45966
//3.94775

MHL_ShowNumber (Median, "Медиана", "Median");
// Медиана:
//Median=6.17554

delete [] a;
```

### 7.21.11 TMHL\_SampleCovariance

Функция вычисляет выборочную ковариацию выборки (несмешенная, исправленная).

Код 549. Синтаксис

```
template <class T> T TMHL_SampleCovariance(T *x, T *y, int VMHL_N);
```

**Входные параметры:**

`x` — указатель на первую сравниваемую выборки;

$y$  — указатель на вторую сравниваемую выборки;

$VMHL\_N$  — размер массивов.

#### **Возвращаемое значение:**

Значение выборочной ковариации (несмешенная, исправленная).

#### **Формула:**

$$Cov(\bar{x}, \bar{y}) = \frac{1}{n-1} \sum_{i=1}^n \left( x_i - \frac{\sum_{j=1}^n x_j}{n} \right) \left( y_i - \frac{\sum_{j=1}^n y_j}{n} \right).$$

Код 550. Пример использования

```
int VMHL_N=10; //Размер массива
double *x;
x=new double[VMHL_N];
double *y;
y=new double[VMHL_N];
//Заполним случайными числами
MHL_RandomRealVector (x,0,10,VMHL_N);
MHL_RandomRealVector (y,0,10,VMHL_N);

//Вызов функции
double SampleCovariance=TMHL_SampleCovariance(x,y,VMHL_N);

//Используем полученный результат
MHL_ShowVector (x,VMHL_N, "Первый массив", "x");
// Первый массив:
//x =
//3.06915
//9.92218
//2.5592
//9.19586
//8.23486
//1.49231
//3.93158
//4.97345
//6.78223
//1.50909
```

### 7.21.12 TMHL\_Variance

Функция вычисляет выборочную дисперсию выборки (несмешенная, исправленная).

Код 551. Синтаксис

```
template <class T> T TMHL_Variance(T *x, int VMHL_N);
```

#### **Входные параметры:**

$x$  — указатель на исходную выборку;

$VMHL\_N$  — размер массива.

#### **Возвращаемое значение:**

Выборочная дисперсия выборки (несмешенная, исправленная).

Код 552. Пример использования

```
int VMHL_N=10; //Размер массива
double *x;
x=new double[VMHL_N];
//Заполним случайными числами
MHL_RandomRealVector (x,0,10,VMHL_N);

//Вызов функции
double Variance=MHL_Variance(x,VMHL_N);

//Используем полученный результат
MHL_ShowVector (x,VMHL_N,"Массив", "x");
//Массив:
//x =
//4.61365
//6.74438
//0.18219
//9.68933
//8.77136
//2.5177
//1.89178
//6.16455
//8.45978
//4.33228

MHL_ShowNumber (Variance,"Значение выборочной дисперсии", "Variance");
//Значение выборочной дисперсии:
//Variance=10.1197

delete [] x;
```

## 7.22 Тестовые функции для оптимизации

### 7.22.1 MHL\_TestFunction\_Ackley

Функция многих переменных: Ackley. Тестовая функция вещественной оптимизации.

Код 553. Синтаксис

```
double MHL_TestFunction_Ackley(double *x, int VMHL_N);
```

#### Входные параметры:

x — указатель на исходный массив;

VMHL\_N — размер массива x.

#### Возвращаемое значение:

Значение тестовой функции в точке x.

#### Описание функции

**Идентификатор:**

MHL\_TestFunction\_Ackley.

**Наименование:**

Функция Ackley.

**Тип:**

Задача вещественной оптимизации.

**Формула** (целевая функция):

$$f(\bar{x}) = 20 + e - 20e^{-0.2\sqrt{\frac{1}{n} \sum_{i=1}^n \bar{x}_i^2}} - e^{\frac{1}{n} \sum_{i=1}^n \cos(2\pi \cdot \bar{x}_i)}, \text{ где}$$

$\bar{x} \in X$ ,  $\bar{x}_j \in [Left_j; Right_j]$ ,  $Left_j = -5$ ,  $Right_j = 5$ ,  $j = \overline{1, n}$ .

**Обозначение:**

$\bar{x}$  — вещественный вектор;

$n$  — размерность вещественного вектора.

**Решаемая задача оптимизации:**

$$\bar{x}_{min} = \arg \min_{\bar{x} \in X} f(\bar{x}).$$

**Точка минимума:**

$$\bar{x}_{min} = (0, 0, \dots, 0)^T, \text{ то есть } (\bar{x}_{min})_j = 0 \ (j = \overline{1, n}).$$

**Минимум функции:**

$$f(\bar{x}_{min}) = 0.$$

**График:**

Рисунок 30 на с. 321

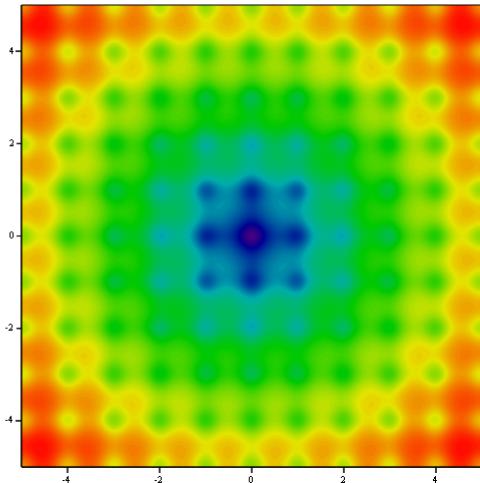
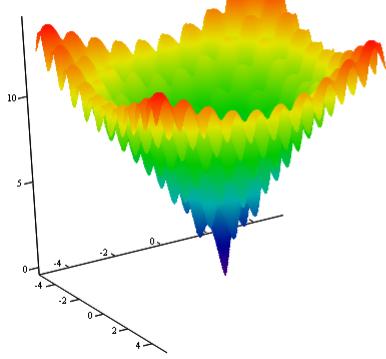


Рисунок 30. Функция Ackley

### Параметры для алгоритмов оптимизации

**Точность вычислений:**

$$\varepsilon = 0.025.$$

**Число интервалов, на которые предполагается разбивать каждую компоненту вектора  $\bar{x}$  в пределах своего изменения** (для алгоритмов дискретной оптимизации) :

$$NumberOfParts_j = 4095 \ (j = \overline{1, n}).$$

**Для этого длина бинарной строки для  $x_j$  координаты равна** (для алгоритмов бинарной оптимизации) :

$$(k_2)_j = 12 \ (j = \overline{1, n}).$$

**Замечание:**  $NumberOfParts_j$  выбирается как минимальное число, удовлетворяющее соотношению:

$$NumberOfParts_j = 2^{(k_2)_j} - 1 \geq \frac{10(Right_j - Left_j)}{\varepsilon}, \text{ где } (k_2)_j \in \mathbb{N}, (j = \overline{1, n}).$$

### Основная задача и подзадачи

<b>Изменяемый параметр:</b>	$n$ — размерность вещественного вектора.
<b>Значение в основной задаче:</b>	$n = 2$ .
<b>Подзадача №2:</b>	$n = 3$ .
<b>Подзадача №3:</b>	$n = 4$ .
<b>Подзадача №4:</b>	$n = 5$ .
<b>Подзадача №5:</b>	$n = 10$ .
<b>Подзадача №6:</b>	$n = 20$ .
<b>Подзадача №7:</b>	$n = 30$ .

### Нахождение ошибки оптимизации

Пусть в результате работы алгоритма оптимизации за  $N$  запусков мы нашли решения  $\bar{x}_{submin}^k$  со значениями целевой функции  $f(\bar{x}_{submin}^k)$  соответственно ( $k = \overline{1, N}$ ). Используем три вида ошибок:

#### Надёжность:

$$R = \frac{\sum_{k=1}^N S(\bar{x}_{submin}^k)}{N}, \text{ где}$$

$$S(\bar{x}_{submin}^k) = \begin{cases} 1, & \text{если } \left| (\bar{x}_{submin}^k)_j - (\bar{x}_{min})_j \right| \leq \varepsilon, j = \overline{1, n}; \\ 0, & \text{иначе.} \end{cases}$$

#### Ошибка по входным параметрам:

$$E_x = \frac{\sum_{k=1}^N \left( \sqrt{\frac{\sum_{j=1}^n ((\bar{x}_{submin}^k)_j - (\bar{x}_{min})_j)^2}{n}} \right)}{N}.$$

#### Ошибка по значениям целевой функции:

$$E_f = \frac{\sum_{k=1}^N |f(\bar{x}_{submin}^k) - f(\bar{x}_{min})|}{N}.$$

### Свойства задачи

**Условной или безусловной оптимизации:** Задача безусловной оптимизации.

**Одномерной или многомерной оптимизации:** Многомерной:  $n$ .

**Функция унимодальная или многоэкстремальная:** Функция многоэкстремальная.

**Функция стохастическая или нет:** Функция не стохастическая.

**Особенности:** Нет.

Код 554. Пример использования

```
double *x;
double f;
int VMHL_N=2;
x=new double[VMHL_N];
for (int i=0;i<VMHL_N;i++) x[i]=MHL_RandomUniform(-5,5);
f=MHL_TestFunction_Ackley(x,VMHL_N);

MHL_ShowVector (x,VMHL_N,"Входной вектор", "x");
//Входной вектор:
//x =
//4.51813
// -4.19861

MHL_ShowNumber (f,"Значение функции", "f");
//Значение функции:
//f=13.645

delete[] x;
```

### 7.22.2 MHL\_TestFunction\_AdditivePotential

Функция двух переменных: аддитивная потенциальная функция. Тестовая функция вещественной оптимизации.

Код 555. Синтаксис

```
double MHL_TestFunction_AdditivePotential(double x, double y);
```

**Входные параметры:**

$x$  — первая вещественная переменная;

$y$  — вторая вещественная переменная.

**Возвращаемое значение:**

Значение тестовой функции в точке  $(x, y)$ .

**Описание функции**

**Идентификатор:**

MHL\_TestFunction\_AdditivePotential.

**Наименование:**

Аддитивная потенциальная функция.

**Тип:**

Задача вещественной оптимизации.

**Формула** (целевая функция):

$$f(\bar{x}) = z(\bar{x}_1) + z(\bar{x}_2), \text{ где } (3)$$

$$z(v) = -\frac{1}{(v-1)^2 + 0.2} - \frac{1}{2(v-2)^2 + 0.15} - \frac{1}{3(v-3)^2 + 0.3},$$

$\bar{x} \in X, \bar{x}_j \in [Left_j; Right_j], Left_j = 0, Right_j = 4, j = \overline{1, n}, n = 2.$

**Обозначение:**

$\bar{x}$  — вещественный вектор;

$n = 2$  — размерность вещественного вектора.

**Решаемая задача оптимизации:**

$\bar{x}_{min} = \arg \min_{\bar{x} \in X} f(\bar{x}).$

**Точка минимума:**

$\bar{x}_{min} = (2, 2)^T$ , то есть  $(\bar{x}_{min})_j = 2 (j = \overline{1, n}).$

**Минимум функции:**

$f(\bar{x}_{min}) = -15.606060606060606.$

**График:**

Рисунок 31 на с. 324 стр.

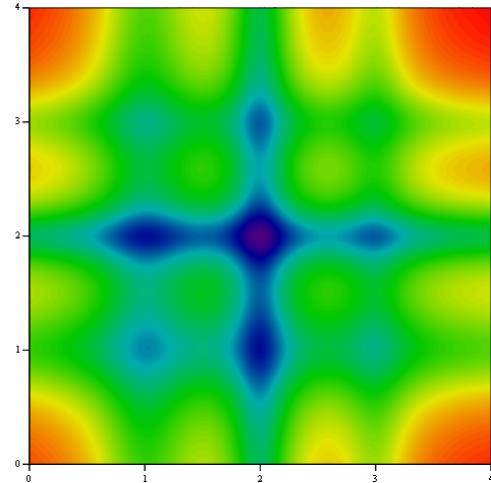
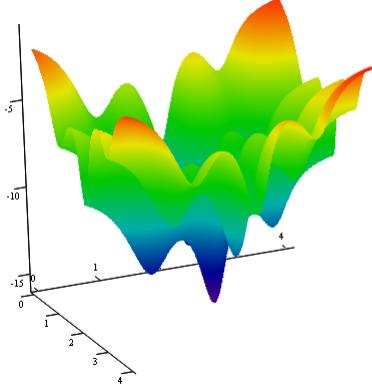


Рисунок 31. Аддитивная потенциальная функция

**Параметры для алгоритмов оптимизации**

**Точность вычислений:**

$\varepsilon = 0.01.$

**Число интервалов, на которые предполагается разбивать каждую компоненту вектора  $\bar{x}$  в пределах своего изменения** (для алгоритмов дискретной оптимизации) :

$NumberOfParts_j = 4095 (j = \overline{1, n}).$

**Для этого длина бинарной строки для  $x_j$  координаты равна** (для алгоритмов бинарной оптимизации) :

$(k_2)_j = 12 (j = \overline{1, n}).$

**Замечание:**  $NumberOfParts_j$  выбирается как минимальное число, удовлетворяющее соотношению:

$$NumberOfParts_j = 2^{(k_2)_j} - 1 \geq \frac{10(Right_j - Left_j)}{\varepsilon}, \text{ где } (k_2)_j \in \mathbb{N}, (j = \overline{1, n}).$$

## 7.23 Основная задача и подзадачи

**Изменяемый параметр:**

$n$  — размерность вещественного вектора.

**Значение в основной задаче:**

$n = 2$ .

## 7.24 Нахождение ошибки оптимизации

Пусть в результате работы алгоритма оптимизации за  $N$  запусков мы нашли решения  $\bar{x}_{submin}^k$  со значениями целевой функции  $f(\bar{x}_{submin}^k)$  соответственно ( $k = \overline{1, N}$ ). Используем три вида ошибок:

**Надёжность:**

$$R = \frac{\sum_{k=1}^N S(\bar{x}_{submin}^k)}{N}, \text{ где}$$

$$S(\bar{x}_{submin}^k) = \begin{cases} 1, & \text{если } \left| (\bar{x}_{submin}^k)_j - (\bar{x}_{min})_j \right| \leq \varepsilon, j = \overline{1, n}; \\ 0, & \text{иначе.} \end{cases}$$

**Ошибка по входным параметрам:**

$$E_x = \frac{\sum_{k=1}^N \left( \frac{\sqrt{\sum_{j=1}^n ((\bar{x}_{submin}^k)_j - (\bar{x}_{min})_j)^2}}{n} \right)}{N}.$$

**Ошибка по значениям целевой функции:**

$$E_f = \frac{\sum_{k=1}^N |f(\bar{x}_{submin}^k) - f(\bar{x}_{min})|}{N}.$$

## 7.25 Свойства задачи

**Условной или безусловной оптимизации:** Задача безусловной оптимизации.

**Одномерной или многомерной оптимизации:** Многомерной:  $n$ .

**Функция унимодальная или многоэкстремальная:** Функция многоэкстремальная.

**Функция стохастическая или нет:** Функция не стохастическая.

**Особенности:** Нет.

Код 556. Пример использования

```
double x;
double y;
double f;
x=MHL_RandomUniform(-5,5);
y=MHL_RandomUniform(-5,5);

//Вызываем функцию
f=MHL_TestFunction_AdditivePotential(x,y);

MHL>ShowNumber (x,"Первая вещественная переменная", "x");
//Первая вещественная переменная:
//x=-2.87701

MHL>ShowNumber (y,"Вторая вещественная переменная", "y");
//Вторая вещественная переменная:
//y=-0.443955

MHL>ShowNumber (f,"Значение функции", "f");
//Значение функции:
//f=-0.64441
```

### 7.25.1 MHL\_TestFunction\_MultiplicativePotential

Функция двух переменных: мультипликативная потенциальная функция. Тестовая функция вещественной оптимизации.

Код 557. Синтаксис

```
double MHL_TestFunction_MultiplicativePotential(double x, double y);
```

**Входные параметры:**

$x$  — первая вещественная переменная;

$y$  — вторая вещественная переменная.

**Возвращаемое значение:**

Значение тестовой функции в точке  $(x, y)$ .

<b>Идентификатор:</b>	MHL_TestFunction_Multiplicative
<b>Описание функции</b>	<b>Наименование:</b>
	Мультипликативная потенциальная
<b>Тип:</b>	
Задача вещественной оптимизации	

**Формула** (целевая функция):

$$f(\bar{x}) = -z(\bar{x}_1) \cdot z(\bar{x}_2), \text{ где} \quad (4)$$

$$z(v) = -\frac{1}{(v-1)^2 + 0.2} - \frac{1}{2(v-2)^2 + 0.15} - \frac{1}{3(v-3)^2 + 0.3},$$

$\bar{x} \in X, \bar{x}_j \in [Left_j; Right_j], Left_j = 0, Right_j = 4, j = \overline{1, n}, n = 2.$

**Обозначение:**

$\bar{x}$  — вещественный вектор;

$n = 2$  — размерность вещественного вектора.

**Решаемая задача оптимизации:**

$$\bar{x}_{min} = \arg \min_{\bar{x} \in X} f(\bar{x}).$$

**Точка минимума:**

$$\bar{x}_{min} = (2, 2)^T, \text{ то есть } (\bar{x}_{min})_j = 2 \ (j = \overline{1, n}).$$

**Минимум функции:**

$$f(\bar{x}_{min}) = -60.8872819100091.$$

**График:**

Рисунок 32 на с. 327 стр.

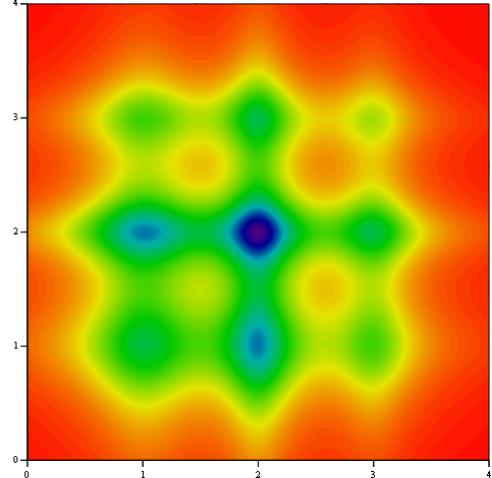
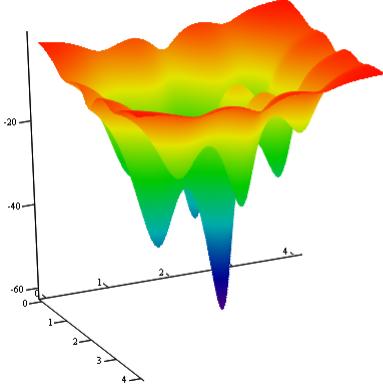


Рисунок 32. Мультипликативная потенциальная функция

## 7.26 Параметры для алгоритмов оптимизации

**Точность вычислений:**

$$\varepsilon = 0.01.$$

**Число интервалов, на которые предполагается разбивать каждую компоненту вектора  $\bar{x}$  в пределах своего изменения** (для алгоритмов дискретной оптимизации) :

**Для этого длина бинарной строки для  $x_j$  координаты равна** (для алгоритмов бинарной оптимизации) :

**Замечание:**  $NumberOfParts_j$  выбирается как минимальное число, удовлетворяющее соотношению:

$$NumberOfParts_j = 2^{(k_2)_j} - 1 \geq \frac{10 (Right_j - Left_j)}{\varepsilon}, \text{ где } (k_2)_j \in \mathbb{N}, (j = \overline{1, n}).$$

## 7.27 Основная задача и подзадачи

**Изменяемый параметр:**

$n$  — размерность вещественного вектора.

**Значение в основной задаче:**

$$n = 2.$$

## 7.28 Нахождение ошибки оптимизации

Пусть в результате работы алгоритма оптимизации за  $N$  запусков мы нашли решения  $\bar{x}_{submin}^k$  со значениями целевой функции  $f(\bar{x}_{submin}^k)$  соответственно ( $k = \overline{1, N}$ ). Используем три вида ошибок:

**Надёжность:**

$$R = \frac{\sum_{k=1}^N S(\bar{x}_{submin}^k)}{N}, \text{ где}$$

$$S(\bar{x}_{submin}^k) = \begin{cases} 1, & \text{если } \left| (\bar{x}_{submin}^k)_j - (\bar{x}_{min})_j \right| \leq \varepsilon, j = \overline{1, n}; \\ 0, & \text{иначе.} \end{cases}$$

**Ошибка по входным параметрам:**

$$E_x = \frac{\sum_{k=1}^N \left( \sqrt{\frac{\sum_{j=1}^n ((\bar{x}_{submin}^k)_j - (\bar{x}_{min})_j)^2}{n}} \right)}{N}.$$

**Ошибка по значениям целевой функции:**

$$E_f = \frac{\sum_{k=1}^N |f(\bar{x}_{submin}^k) - f(\bar{x}_{min})|}{N}.$$

## 7.29 Свойства задачи

**Условной или безусловной оптимизации:** Задача безусловной оптимизации.

**Одномерной или многомерной оптимизации:** Многомерной:  $n$ .

**Функция унимодальная или многоэкстремальная:** Функция многоэкстремальная.

**Функция стохастическая или нет:** Функция не стохастическая.

**Особенности:** Нет.

Код 558. Пример использования

```
double x;
double y;
double f;
x=MHL_RandomUniform(-5,5);
y=MHL_RandomUniform(-5,5);

//Вызываем функцию
f=MHL_TestFunction_MultiplicativePotential(x,y);

MHL>ShowNumber (x,"Первая вещественная переменная", "x");
//Первая вещественная переменная:
//x=0.520743

MHL>ShowNumber (y,"Вторая вещественная переменная", "y");
//Вторая вещественная переменная:
//y=-4.96242

MHL>ShowNumber (f,"Значение функции", "f");
//Значение функции:
//f=-0.113219
```

### 7.29.1 MHL\_TestFunction\_ParaboloidOfRevolution

Функция многих переменных: Эллиптический параболоид. Тестовая функция вещественной оптимизации.

Код 559. Синтаксис

```
double MHL_TestFunction_ParaboloidOfRevolution(double *x, int VMHL_N);
```

**Входные параметры:**

$x$  — указатель на исходный массив;

$VMHL\_N$  — размер массива  $x$ .

**Возвращаемое значение:**

Значение тестовой функции в точке  $x$ .

## Описание функции

**Идентификатор:** MHL\_TestFunction\_ParaboloidOfRevolution.

**Наименование:** Эллиптический параболоид.

**Тип:** Задача вещественной оптимизации.

**Формула** (целевая функция):

$$f(\bar{x}) = \sum_{i=1}^n \bar{x}_i^2, \text{ где}$$

$\bar{x} \in X, \bar{x}_j \in [Left_j; Right_j], Left_j = -2, Right_j = 2, j = \overline{1, n}.$

**Обозначение:**  $\bar{x}$  — вещественный вектор;

$n$  — размерность вещественного вектора.

**Решаемая задача оптимизации:**  $\bar{x}_{min} = \arg \min_{\bar{x} \in X} f(\bar{x}).$

**Точка минимума:**  $\bar{x}_{min} = (0, 0, \dots, 0)^T$ , то есть  $(\bar{x}_{min})_j = 0 (j = \overline{1, n}).$

**Минимум функции:**  $f(\bar{x}_{min}) = 0.$

**График:** Рисунок 33 на с 330 стр.

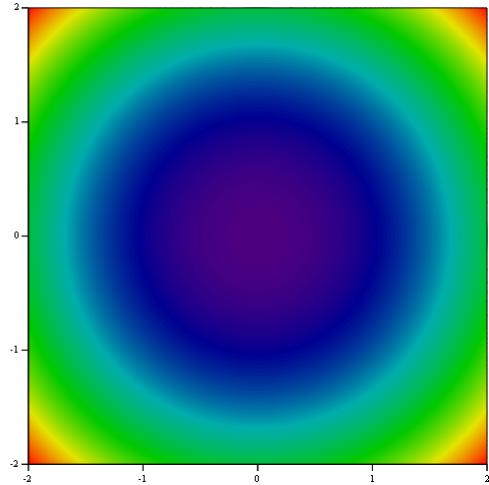
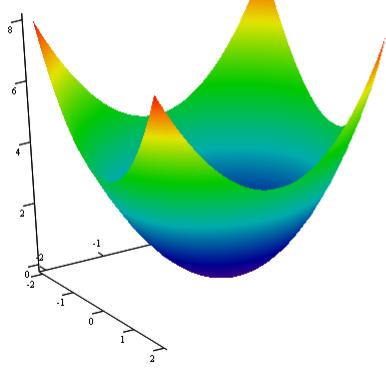


Рисунок 33. Эллиптический параболоид

## Параметры для алгоритмов оптимизации

**Точность вычислений:**  $\varepsilon = 0.01.$

**Число интервалов, на которые предполагается разбивать каждую компоненту вектора  $\bar{x}$  в пределах своего изменения** (для алгоритмов дискретной оптимизации) :

**Для этого длина бинарной строки для  $x_j$  координаты равна** (для алгоритмов бинарной оптимизации) :  $(k_2)_j = 12 (j = \overline{1, n}).$

**Замечание:**  $NumberOfParts_j$  выбирается как минимальное число, удовлетворяющее соотношению:

$$NumberOfParts_j = 2^{(k_2)_j} - 1 \geq \frac{10(Right_j - Left_j)}{\varepsilon}, \text{ где } (k_2)_j \in \mathbb{N}, (j = \overline{1, n}).$$

### Основная задача и подзадачи

<b>Изменяемый параметр:</b>	$n$ — размерность вещественного вектора.
<b>Значение в основной задаче:</b>	$n = 2$ .
<b>Подзадача №2:</b>	$n = 3$ .
<b>Подзадача №3:</b>	$n = 4$ .
<b>Подзадача №4:</b>	$n = 5$ .
<b>Подзадача №5:</b>	$n = 10$ .
<b>Подзадача №6:</b>	$n = 20$ .
<b>Подзадача №7:</b>	$n = 30$ .

### Нахождение ошибки оптимизации

Пусть в результате работы алгоритма оптимизации за  $N$  запусков мы нашли решения  $\bar{x}_{submin}^k$  со значениями целевой функции  $f(\bar{x}_{submin}^k)$  соответственно ( $k = \overline{1, N}$ ). Используем три вида ошибок:

#### Надёжность:

$$R = \frac{\sum_{k=1}^N S(\bar{x}_{submin}^k)}{N}, \text{ где}$$

$$S(\bar{x}_{submin}^k) = \begin{cases} 1, & \text{если } \left| (\bar{x}_{submin}^k)_j - (\bar{x}_{min})_j \right| \leq \varepsilon, j = \overline{1, n}; \\ 0, & \text{иначе.} \end{cases}$$

#### Ошибка по входным параметрам:

$$E_x = \frac{\sum_{k=1}^N \left( \sqrt{\frac{\sum_{j=1}^n ((\bar{x}_{submin}^k)_j - (\bar{x}_{min})_j)^2}{n}} \right)}{N}.$$

#### Ошибка по значениям целевой функции:

$$E_f = \frac{\sum_{k=1}^N |f(\bar{x}_{submin}^k) - f(\bar{x}_{min})|}{N}.$$

### Свойства задачи

**Условной или безусловной оптимизации:** Задача безусловной оптимизации.

**Одномерной или многомерной оптимизации:** Многомерной:  $n$ .

**Функция унимодальная или многоэкстремальная:** Функция унимодальная.

**Функция стохастическая или нет:** Функция не стохастическая.

**Особенности:** Нет.

Код 560. Пример использования

```
double *x;
double f;
int VMHL_N=2;
x=new double[VMHL_N];
for (int i=0;i<VMHL_N;i++) x[i]=MHL_RandomUniform(-2,2);
f=MHL_TestFunction_ParaboloidOfRevolution(x,VMHL_N);

MHL_ShowVector (x,VMHL_N,"Входной вектор", "x");
// Входной вектор:
//x =
//0.0842285
// -1.04395

MHL>ShowNumber (f,"Значение функции", "f");
//Значение функции:
//f=1.09692

delete[] x;
```

## 7.29.2 MHL\_TestFunction\_Rastrigin

Функция многих переменных: функция Раstrигина. Тестовая функция вещественной оптимизации.

Код 561. Синтаксис

```
double MHL_TestFunction_Rastrigin(double *x, int VMHL_N);
```

**Входные параметры:**

$x$  — указатель на исходный массив;

$VMHL\_N$  — размер массива  $x$ .

**Возвращаемое значение:**

Значение тестовой функции в точке  $x$ .

**Описание функции**

**Идентификатор:**

MHL\_TestFunction\_Rastrigin.

**Наименование:**

Функция Раstrигина.

**Тип:**

Задача вещественной оптимизации.

**Формула** (целевая функция):

$$f(\bar{x}) = 10n + \sum_{i=1}^n (\bar{x}_i^2 - 10 \cdot \cos(2\pi \cdot \bar{x}_i)), \text{ где}$$

$\bar{x} \in X, \bar{x}_j \in [Left_j; Right_j], Left_j = -5, Right_j = 5, j = \overline{1, n}.$

**Обозначение:**

$\bar{x}$  — вещественный вектор;

$n$  — размерность вещественного вектора.

**Решаемая задача оптимизации:**

$\bar{x}_{min} = \arg \min_{\bar{x} \in X} f(\bar{x}).$

**Точка минимума:**

$\bar{x}_{min} = (0, 0, \dots, 0)^T$ , то есть  $(\bar{x}_{min})_j = 0 (j = \overline{1, n}).$

**Минимум функции:**

$f(\bar{x}_{min}) = 0.$

**График:**

Рисунок 34 на 333 стр.

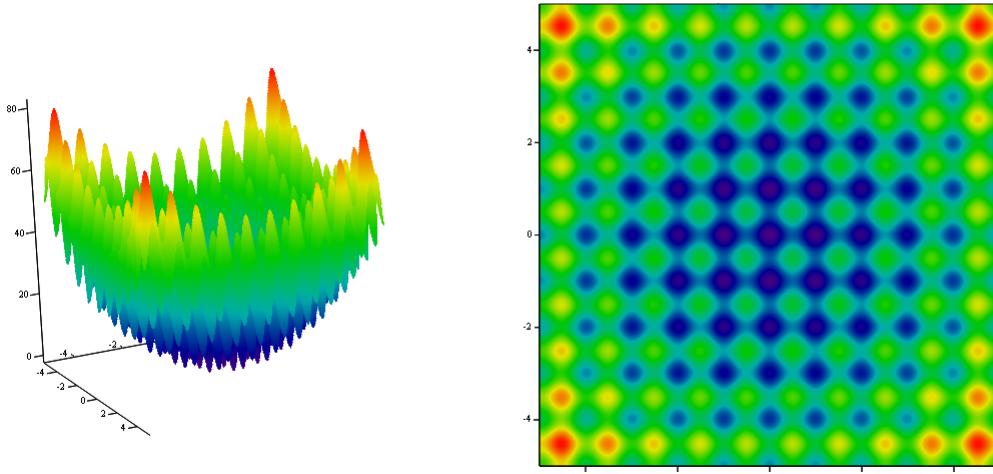


Рисунок 34. Функция Раstrигина

### Параметры для алгоритмов оптимизации

**Точность вычислений:**

$\varepsilon = 0.025.$

**Число интервалов, на которые предполагается разбивать каждую компоненту вектора  $\bar{x}$  в пределах своего изменения** (для алгоритмов дискретной оптимизации) :

$NumberOfParts_j = 4095 (j = \overline{1, n}).$

**Для этого длина бинарной строки для  $x_j$  координаты равна** (для алгоритмов бинарной оптимизации) :

$(k_2)_j = 12 (j = \overline{1, n}).$

**Замечание:**  $NumberOfParts_j$  выбирается как минимальное число, удовлетворяющее соотношению:

$$NumberOfParts_j = 2^{(k_2)_j} - 1 \geq \frac{10(Right_j - Left_j)}{\varepsilon}, \text{ где } (k_2)_j \in \mathbb{N}, (j = \overline{1, n}).$$

### Основная задача и подзадачи

<b>Изменяемый параметр:</b>	$n$ — размерность вещественного вектора.
<b>Значение в основной задаче:</b>	$n = 2$ .
<b>Подзадача №2:</b>	$n = 3$ .
<b>Подзадача №3:</b>	$n = 4$ .
<b>Подзадача №4:</b>	$n = 5$ .
<b>Подзадача №5:</b>	$n = 10$ .
<b>Подзадача №6:</b>	$n = 20$ .
<b>Подзадача №7:</b>	$n = 30$ .

### Нахождение ошибки оптимизации

Пусть в результате работы алгоритма оптимизации за  $N$  запусков мы нашли решения  $\bar{x}_{submin}^k$  со значениями целевой функции  $f(\bar{x}_{submin}^k)$  соответственно ( $k = \overline{1, N}$ ). Используем три вида ошибок:

#### Надёжность:

$$R = \frac{\sum_{k=1}^N S(\bar{x}_{submin}^k)}{N}, \text{ где}$$

$$S(\bar{x}_{submin}^k) = \begin{cases} 1, & \text{если } \left| (\bar{x}_{submin}^k)_j - (\bar{x}_{min})_j \right| \leq \varepsilon, j = \overline{1, n}; \\ 0, & \text{иначе.} \end{cases}$$

#### Ошибка по входным параметрам:

$$E_x = \frac{\sum_{k=1}^N \left( \sqrt{\frac{\sum_{j=1}^n ((\bar{x}_{submin}^k)_j - (\bar{x}_{min})_j)^2}{n}} \right)}{N}.$$

#### Ошибка по значениям целевой функции:

$$E_f = \frac{\sum_{k=1}^N |f(\bar{x}_{submin}^k) - f(\bar{x}_{min})|}{N}.$$

### Свойства задачи

**Условной или безусловной оптимизации:** Задача безусловной оптимизации.

**Одномерной или многомерной оптимизации:** Многомерной:  $n$ .

**Функция унимодальная или многоэкстремальная:** Функция многоэкстремальная.

**Функция стохастическая или нет:** Функция не стохастическая.

**Особенности:** Нет.

Код 562. Пример использования

```
double *x;
double f;
int VMHL_N=2;
x=new double[VMHL_N];
for (int i=0;i<VMHL_N;i++) x[i]=MHL_RandomUniform(-5,5);
f=MHL_TestFunction_Rastrigin(x,VMHL_N);

MHL_ShowVector (x,VMHL_N,"Входной вектор", "x");
// Входной вектор:
//x =
//2.62268
//3.52692

MHL>ShowNumber (f,"Значение функции", "f");
//Значение функции:
//f=56.3483

delete[] x;
```

### 7.29.3 MHL\_TestFunction\_Rosenbrock

Функция многих переменных: функция Розенброка. Тестовая функция вещественной оптимизации.

Код 563. Синтаксис

```
double MHL_TestFunction_Rosenbrock(double *x, int VMHL_N);
```

**Входные параметры:**

$x$  — указатель на исходный массив;

$VMHL\_N$  — размер массива  $x$ .

**Возвращаемое значение:**

Значение тестовой функции в точке  $x$ .

**Описание функции**

**Идентификатор:**

MHL\_TestFunction\_Rosenbrock.

**Наименование:**

Функция Розенброка.

**Тип:**

Задача вещественной оптимизации.

**Формула** (целевая функция):

$$f(\bar{x}) = \sum_{i=1}^{n-1} \left( 100(\bar{x}_{i+1} - \bar{x}_i^2)^2 + (1 - \bar{x}_i)^2 \right), \text{ где}$$

$\bar{x} \in X, \bar{x}_j \in [Left_j; Right_j], Left_j = -2, Right_j = 2, j = \overline{1, n}$ .

**Обозначение:**

$\bar{x}$  — вещественный вектор;

$n$  — размерность вещественного вектора.

**Решаемая задача оптимизации:**

$\bar{x}_{min} = \arg \min_{\bar{x} \in X} f(\bar{x})$ .

**Точка минимума:**

$\bar{x}_{min} = (1, 1, \dots, 1)^T$ , то есть  $(\bar{x}_{min})_j = 1 (j = \overline{1, n})$ .

**Минимум функции:**

$f(\bar{x}_{min}) = 0$ .

**График:**

Рисунок 35 на с 336 стр.

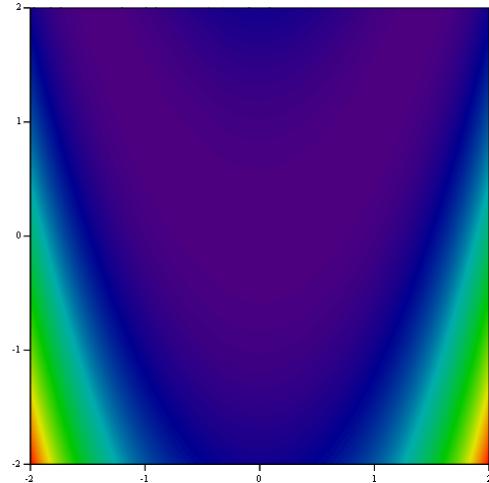
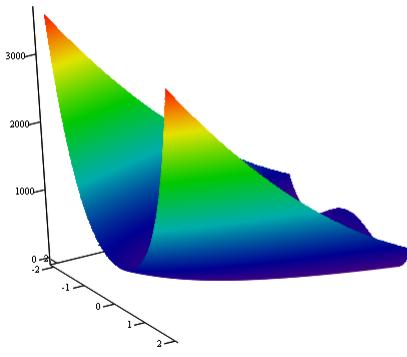


Рисунок 35. Функция Розенброка

### Параметры для алгоритмов оптимизации

**Точность вычислений:**

$\varepsilon = 0.01$ .

**Число интервалов, на которые предполагается разбивать каждую компоненту вектора  $\bar{x}$  в пределах своего изменения** (для алгоритмов дискретной оптимизации) :

$NumberOfParts_j = 4095 (j = \overline{1, n})$ .

**Для этого длина бинарной строки для  $x_j$  координаты равна** (для алгоритмов бинарной оптимизации) :

$(k_2)_j = 12 (j = \overline{1, n})$ .

**Замечание:**  $NumberOfParts_j$  выбирается как минимальное число, удовлетворяющее соотношению:

$$NumberOfParts_j = 2^{(k_2)_j} - 1 \geq \frac{10(Right_j - Left_j)}{\varepsilon}, \text{ где } (k_2)_j \in \mathbb{N}, (j = \overline{1, n}).$$

### Основная задача и подзадачи

**Изменяемый параметр:**  $n$  — размерность вещественного вектора.

**Значение в основной задаче:**  $n = 2$ .

**Подзадача №2:**  $n = 3$ .

**Подзадача №3:**  $n = 4$ .

**Подзадача №4:**  $n = 5$ .

**Подзадача №5:**  $n = 10$ .

**Подзадача №6:**  $n = 20$ .

**Подзадача №7:**  $n = 30$ .

### Нахождение ошибки оптимизации

Пусть в результате работы алгоритма оптимизации за  $N$  запусков мы нашли решения  $\bar{x}_{submin}^k$  со значениями целевой функции  $f(\bar{x}_{submin}^k)$  соответственно ( $k = \overline{1, N}$ ). Используем три вида ошибок:

#### Надёжность:

$$R = \frac{\sum_{k=1}^N S(\bar{x}_{submin}^k)}{N}, \text{ где}$$

$$S(\bar{x}_{submin}^k) = \begin{cases} 1, & \text{если } \left| (\bar{x}_{submin}^k)_j - (\bar{x}_{min})_j \right| \leq \varepsilon, j = \overline{1, n}; \\ 0, & \text{иначе.} \end{cases}$$

#### Ошибка по входным параметрам:

$$E_x = \frac{\sum_{k=1}^N \left( \sqrt{\frac{\sum_{j=1}^n ((\bar{x}_{submin}^k)_j - (\bar{x}_{min})_j)^2}{n}} \right)}{N}.$$

#### Ошибка по значениям целевой функции:

$$E_f = \frac{\sum_{k=1}^N |f(\bar{x}_{submin}^k) - f(\bar{x}_{min})|}{N}.$$

### Свойства задачи

**Условной или безусловной оптимизации:** Задача безусловной оптимизации.

**Одномерной или многомерной оптимизации:** Многомерной:  $n$ .

**Функция унимодальная или многоэкстремальная:** Функция многоэкстремальная.

**Функция стохастическая или нет:** Функция не стохастическая.

**Особенности:** Нет.

Код 564. Пример использования

```
double *x;
double f;
int VMHL_N=2;
x=new double[VMHL_N];
for (int i=0;i<VMHL_N;i++) x[i]=MHL_RandomUniform(-2,2);
f=MHL_TestFunction_Rosenbrock(x,VMHL_N);

MHL_ShowVector (x,VMHL_N,"Входной вектор", "x");
// Входной вектор:
//x =
// -1.28491
// 0.342896

MHL_ShowNumber (f,"Значение функции", "f");
// Значение функции:
// f=176.334

delete[] x;
```

#### 7.29.4 MHL\_TestFunction\_SumVector

Сумма всех элементов бинарного вектора. Тестовая функция бинарной оптимизации.

Код 565. Синтаксис

```
double MHL_TestFunction_SumVector(int *x, int VMHL_N);
```

**Входные параметры:**

$x$  — указатель на исходный массив;

$VMHL\_N$  — размер массива  $x$ .

**Возвращаемое значение:**

Значение тестовой функции в точке  $x$ .

**Описание функции**

<b>Идентификатор:</b>	MHL_TestFunction_SumVector.
<b>Наименование:</b>	Сумма всех элементов бинарного вектора.
<b>Тип:</b>	Задача бинарной оптимизации.
<b>Формула</b> (целевая функция):	$f(\bar{x}) = \sum_{i=1}^n \bar{x}_i, \text{ где}$
$\bar{x} \in X, \bar{x}_j \in \{0; 1\}, j = \overline{1, n}.$	
<b>Обозначение:</b>	$\bar{x}$ — бинарный вектор;
	$n$ — размерность бинарного вектора.
<b>Объем поискового пространства:</b>	$\mu(X) = 2^n.$
<b>Решаемая задача оптимизации:</b>	$\bar{x}_{max} = \arg \max_{\bar{x} \in X} f(\bar{x}).$
<b>Точка максимума:</b>	$\bar{x}_{max} = (1, 1, \dots, 1)^T$ , то есть $(\bar{x}_{max})_j = 1 (j = \overline{1, n}).$
<b>Максимум функции:</b>	$f(\bar{x}_{max}) = n.$
<b>Точка минимума:</b>	$\bar{x}_{min} = (0, 0, \dots, 0)^T$ , то есть $(\bar{x}_{min})_j = 0 (j = \overline{1, n}).$
<b>Минимум функции:</b>	$f(\bar{x}_{min}) = 0.$
<b>Основная задача и подзадачи</b>	
<b>Изменяемый параметр:</b>	$n$ — размерность бинарного вектора.
<b>Значение в основной задаче:</b>	$n = 20.$
<b>Подзадача №2:</b>	$n = 30.$
<b>Подзадача №3:</b>	$n = 40.$
<b>Подзадача №4:</b>	$n = 50.$
<b>Подзадача №5:</b>	$n = 60.$
<b>Подзадача №6:</b>	$n = 70.$
<b>Подзадача №7:</b>	$n = 80.$
<b>Подзадача №8:</b>	$n = 90.$
<b>Подзадача №9:</b>	$n = 100.$
<b>Подзадача №10:</b>	$n = 200.$

### Нахождение ошибки оптимизации

Пусть в результате работы алгоритма оптимизации за  $N$  запусков мы нашли решения  $\bar{x}_{submax}^k$  со значениями целевой функции  $f(\bar{x}_{submax}^k)$  соответственно ( $k = \overline{1, N}$ ). Используем три вида ошибок:

**Надёжность:**

$$R = \frac{\sum_{k=1}^N S(\bar{x}_{submax}^k)}{N}, \text{ где}$$
$$S(\bar{x}_{submax}^k) = \begin{cases} 1, & \text{если } \bar{x}_{submax}^k = \bar{x}_{max}; \\ 0, & \text{иначе.} \end{cases}$$

**Ошибка по входным параметрам:**

$$E_x = \frac{\sum_{k=1}^N \left( \frac{\sum_{j=1}^n |(\bar{x}_{submax}^k)_j - (\bar{x}_{max})_j|}{n} \right)}{N}.$$

**Ошибка по значениям целевой функции:**

$$E_f = \frac{\sum_{k=1}^N |f(\bar{x}_{submax}^k) - f(\bar{x}_{max})|}{N}.$$

**Свойства задачи**

**Условной или безусловной оптимизации:** Задача безусловной оптимизации.

**Одномерной или многомерной оптимизации:** Многомерной:  $n$ .

**Функция унимодальная или многоэкстремальная:** Функция унимодальная.

**Функция стохастическая или нет:** Функция не стохастическая.

**Особенности:** Нет.

Код 566. Пример использования

```
int VMHL_N=10; //Размер массива (число строк)
int *x;
x=new int[VMHL_N];
//Получим случайный бинарный вектор
TMHL_RandomBinaryVector(x,VMHL_N);

//Вызов функции
double f=MHL_TestFunction_SumVector(x,VMHL_N);

//Используем полученный результат
MHL_ShowVector (x,VMHL_N, "Вектор", "x");
//Вектор:
//x =
//0
//0
//1
//0
//0
//1
//1
//1
//1
```

```
//0
//1

MHL_ShowNumber (f,"Значение функции в точке", "f");
//Значение функции в точке:
//f=5

delete [] x;
```

## 7.30 Тригонометрические функции

### 7.30.1 MHL\_Cos

Функция возвращает косинус угла в радианах.

Код 567. Синтаксис

```
double MHL_Cos(double x);
```

**Входные параметры:**

x — угол в радианах.

**Возвращаемое значение:**

Косинус угла.

**Примечание:**

Вводится только для того, чтобы множество тригонометрических функций было полным.

Код 568. Пример использования

```
double y;
double Angle=MHL_PI; //Угол в радианах

//Вызов функции
y=MHL_Cos(Angle);

//Используем полученный результат
MHL_ShowNumber(y,"Косинус угла "+MHL_NumberToText(Angle)+" радианов", "равен");
//Косинус угла 3.14159 радианов:
//равен=-1
```

### 7.30.2 MHL\_CosDeg

Функция возвращает косинус угла в градусах.

Код 569. Синтаксис

```
double MHL_CosDeg(double x);
```

**Входные параметры:**

x — угол в градусах.

### **Возвращаемое значение:**

Косинус угла.

Код 570. Пример использования

```

double y;
double Angle=180;//Угол в градусах

//Вызов функции
y=MHL_CosDeg(Angle);

//Используем полученный результат
MHL_ShowNumber(y, "Косинус угла "+MHL_NumberToText(Angle)+" градусов", "равен");
//Косинус угла 180 градусов:
//равен=-1

```

### **7.30.3 MHL\_Cosec**

Функция возвращает косеканс угла в радианах.

Код 571. Синтаксис

```
double MHL_Cosec(double x);
```

### **Входные параметры:**

x — угол в радианах.

### **Возвращаемое значение:**

Косеканс угла.

Код 572. Пример использования

```

double y;
double Angle=MHL_PI/4.;//Угол в радианах

//Вызов функции
y=MHL_Cosec(Angle);

//Используем полученный результат
MHL_ShowNumber(y, "Косеканс угла "+MHL_NumberToText(Angle)+" радианов", "равен");
//Косеканс угла 0.785398 радианов:
//равен=1.41421

```

### **7.30.4 MHL\_CosecDeg**

Функция возвращает косеканс угла в градусах.

Код 573. Синтаксис

```
double MHL_CosecDeg(double x);
```

### **Входные параметры:**

x — угол в градусах.

### **Возвращаемое значение:**

Косеканс угла.

Код 574. Пример использования

```

double y;
double Angle=45;//Угол в градусах

//Вызов функции
y=MHL_CosecDeg(Angle);

//Используем полученный результат
MHL_ShowNumber(y, "Косеканс угла "+MHL_NumberToText(Angle)+" градусов", "равен");
//Косеканс угла 45 градусов:
//равен=1.41421

```

### **7.30.5 MHL\_Cotan**

Функция возвращает котангенс угла в радианах.

Код 575. Синтаксис

```
double MHL_Cotan(double x);
```

### **Входные параметры:**

x — угол в радианах.

### **Возвращаемое значение:**

Котангенс угла.

Код 576. Пример использования

```

double y;
double Angle=MHL_PI/4.;//Угол в радианах

//Вызов функции
y=MHL_Cotan(Angle);

//Используем полученный результат
MHL_ShowNumber(y, "Котангенс угла "+MHL_NumberToText(Angle)+" радианов", "равен");
//Котангенс угла 0.785398 радианов:
//равен=1

```

### **7.30.6 MHL\_CotanDeg**

Функция возвращает котангенс угла в градусах.

Код 577. Синтаксис

```
double MHL_CotanDeg(double x);
```

### **Входные параметры:**

x — угол в градусах.

### **Возвращаемое значение:**

Котангенс угла.

Код 578. Пример использования

```

double y;
double Angle=45; //Угол в градусах

//Вызов функции
y=MHL_CotanDeg(Angle);

//Используем полученный результат
MHL_ShowNumber(y, "Котангенс угла "+MHL_NumberToText(Angle)+" градусов", "равен");
//Котангенс угла 45 градусов:
//равен=1

```

### **7.30.7 MHL\_Sec**

Функция возвращает секанс угла в радианах.

Код 579. Синтаксис

```
double MHL_Sec(double x);
```

### **Входные параметры:**

x — угол в радианах.

### **Возвращаемое значение:**

Секанс угла.

Код 580. Пример использования

```

double y;
double Angle=MHL_PI/4.; //Угол в радианах

//Вызов функции
y=MHL_Sec(Angle);

//Используем полученный результат
MHL_ShowNumber(y, "Секанс угла "+MHL_NumberToText(Angle)+" радианов", "равен");
//Секанс угла 0.785398 радианов:
//равен=1.41421

```

### **7.30.8 MHL\_SecDeg**

Функция возвращает секанс угла в градусах.

Код 581. Синтаксис

```
double MHL_SecDeg(double x);
```

### **Входные параметры:**

x — угол в градусах.

### **Возвращаемое значение:**

Секанс угла.

Код 582. Пример использования

```
double y;
double Angle=45; //угол в градусах

//Вызов функции
y=MHL_SecDeg(Angle);

//Используем полученный результат
MHL>ShowNumber(y, "Секанс угла "+MHL_NumberToText(Angle)+" градусов", "равен");
//Секанс угла 45 градусов:
//равен=1.41421
```

### **7.30.9 MHL\_Sin**

Функция возвращает синус угла в радианах.

Код 583. Синтаксис

```
double MHL_Sin(double x);
```

### **Входные параметры:**

x — угол в радианах.

### **Возвращаемое значение:**

Синус угла.

### **Примечание:**

Вводится только для того, чтобы множество тригонометрических функций было полным.

Код 584. Пример использования

```
double y;
double Angle=MHL_PI/2.; //угол в радианах

//Вызов функции
y=MHL_Sin(Angle);

//Используем полученный результат
MHL>ShowNumber(y, "Синус угла "+MHL_NumberToText(Angle)+" радианов", "равен");
//Синус угла 1.5708 радианов:
//равен=1
```

### **7.30.10 MHL\_SinDeg**

Функция возвращает синус угла в градусах.

Код 585. Синтаксис

```
double MHL_SinDeg(double x);
```

### **Входные параметры:**

х — угол в градусах.

### **Возвращаемое значение:**

Синус угла.

Код 586. Пример использования

```
double y;
double Angle=90; //Угол в градусах

//Вызов функции
y=MHL_SinDeg(Angle);

//Используем полученный результат
MHL>ShowNumber(y, "Синус угла "+MHL_NumberToText(Angle)+" градусов", "равен");
//Синус угла 90 градусов:
//равен=1
```

### **7.30.11 MHL\_Tan**

Функция возвращает тангенс угла в радианах.

Код 587. Синтаксис

```
double MHL_Tan(double x);
```

### **Входные параметры:**

х — угол в радианах.

### **Возвращаемое значение:**

Тангенс угла.

### **Примечание:**

Вводится только для того, чтобы множество тригонометрических функций было полным.

Код 588. Пример использования

```
double y;
double Angle=MHL_PI/4.; //Угол в радианах

//Вызов функции
y=MHL_Tan(Angle);

//Используем полученный результат
MHL>ShowNumber(y, "Тангенс угла "+MHL_NumberToText(Angle)+" радианов", "равен");
//Тангенс угла 0.785398 радианов:
//равен=1
```

### **7.30.12 MHL\_TanDeg**

Функция возвращает тангенс угла в градусах.

Код 589. Синтаксис

```
double MHL_TanDeg(double x);
```

**Входные параметры:**

x — угол в градусах.

**Возвращаемое значение:**

Тангенс угла.

Код 590. Пример использования

```
double y;
double Angle=45; //угол в градусах

//Вызов функции
y=MHL_TanDeg(Angle);

//Используем полученный результат
MHL>ShowNumber(y, "Тангенс угла "+MHL_NumberToText(Angle)+" градусов", "равен");
//Тангенс угла 45 градусов:
//равен=1
```

## 7.31 Уравнения

### 7.31.1 MHL\_QuadraticEquation

Функция решает квадратное уравнение вида:  $a \cdot x^2 + b \cdot x + c = 0$ . Ответ представляет собой два действительных числа.

Код 591. Синтаксис

```
int MHL_QuadraticEquation(double a, double b, double c, double *x1, double *x2);
```

**Входные параметры:**

a — параметр уравнения;

b — параметр уравнения;

c — параметр уравнения;

x1 — первый корень;

x2 — второй корень.

**Возвращаемое значение:**

1 — все хорошо;

0 — решения нет.

Код 592. Пример использования

```
double a=MHL_RandomUniformInt(1,10);
double b=MHL_RandomUniformInt(1,10);
double c=MHL_RandomUniformInt(1,10);
```

```

double x1;
double x2;

int Result=MHL_QuadraticEquation(a,b,c,&x1,&x2);;

//Используем полученный результат
MHL_ShowText("Квадратное уравнение: "+MHL_NumberToText(a)+"x^2+"+MHL_NumberToText(b)+
    "x+"+MHL_NumberToText(c)+"=0");
//Квадратное уравнение: 1x^2+8x+5:
MHL_ShowNumber(Result,"Найдено ли решение", "Result");
//Найдено ли решение:
//Result=1
if (Result==1)
{
MHL_ShowNumber(x1,"Первый корень квадратного уравнения", "x1");
//Первый корень квадратного уравнения:
//x1=-0.683375
MHL_ShowNumber(x2,"Первый корень квадратного уравнения", "x2");
//Первый корень квадратного уравнения:
//x2=-7.31662
}

```

## Список литературы

1. Большев Л. Н., Смирнов Н. В. Таблицы математической статистики. М.: Наука. Главная редакция физико-математической литературы, 1983. 416 с.