**СОДЕРЖАНИЕ**

[ВВЕДЕНИЕ](#_gjdgxs) 3

[1 Метод Нелдера-Мида](#_30j0zll) 4

[1.1](#_1fob9te) Постановка задачи 4

[1.2 Ручной расчет](#_2et92p0) 4

[1.3](#_tyjcwt) Результат работы [программного кода](#_tyjcwt) 10

2 [Метод Флетчера-Ривса](#_30j0zll) 11

2[.1](#_1fob9te) Постановка задачи 11

2[.2 Ручной расчет](#_2et92p0) 11

2[.3](#_tyjcwt) Результат работы [программного кода](#_tyjcwt) 18

3 [Метод Ньютона-Рафсона](#_30j0zll) 19

3[.1](#_1fob9te) Постановка задачи 19

3[.2 Ручной расчет](#_2et92p0) 19

3[.3](#_tyjcwt) Результат работы [программного кода](#_tyjcwt) 22

[ЗАКЛЮЧЕНИЕ](https://docs.google.com/document/d/12v_-yH_6rTCSU0AzqWBR-aL3fZoSotv8/r/edit/edit#heading=h.1hmsyys) 23

[СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ](https://docs.google.com/document/d/12v_-yH_6rTCSU0AzqWBR-aL3fZoSotv8/r/edit/edit#heading=h.41mghml) 24

[ПРИЛОЖЕНИЯ](https://docs.google.com/document/d/12v_-yH_6rTCSU0AzqWBR-aL3fZoSotv8/r/edit/edit#heading=h.2grqrue) 25

**ВВЕДЕНИЕ**

В современной математике и прикладных науках задача поиска минимума функции играет ключевую роль. Она возникает в различных областях, таких как оптимизация, машинное обучение, экономика, физика и инженерия. Для её решения разработано множество численных методов, каждый из которых обладает своими преимуществами и ограничениями. В данной практической работе мы рассмотрим и реализуем три популярных метода оптимизации: метод Хука-Дживса, метод наискорейшего градиентного спуска и метод Ньютона-Рафсона.

Метод Хука-Дживса относится к классу прямых (безградиентных) методов и основывается на чередовании поискового и исследующего этапов. Он не требует вычисления производных функции и хорошо подходит для задач, где такие производные трудно получить или они отсутствуют. Метод наискорейшего градиентного спуска использует информацию о градиенте целевой функции и на каждом шаге перемещается в направлении, противоположном градиенту, с адаптивным выбором длины шага. Метод Ньютона-Рафсона является методом второго порядка и использует вторые производные (гессиан), что обеспечивает высокую точность и быструю сходимость при выполнении определённых условий.

Целью данной работы является реализация указанных методов на языке программирования Python и их сравнение по таким критериям, как точность нахождения минимума, количество итераций и вычислительная сложность. Также будет проведён анализ эффективности каждого метода в зависимости от типа функции и условий задачи.

Результаты работы позволят глубже понять особенности рассматриваемых методов и определить области их наилучшего применения. Полученные знания могут быть полезны при выборе оптимального численного метода для решения практических задач с учётом доступных вычислительных ресурсов и требований к точности.

**1 Метод Хука–Дживса**

## **Постановка задачи**

Найти минимум функции

помощью метода Хука–Дживса (нулевого порядка)

## **Ручной расчет**

Условия:

* Начальная точка: (3, -2)
* Начальный шаг:
* Точность:
* Коэффициент уменьшения шага:

**Ход решения**

Итерация 1. Исследующее движение из (3, -2):

1. – лучше, берём (2,-2)
2. – равное значение, оставляем (2, -2)

Опорное движение:

* Направление: (2, -2) – (3, -2) = (-1, 0)
* Новая точка: (1, -2),

– лучше, берём (1, -2)

Итерация 2. Исследующее движение из (1, -2):

1. – лучше

Опорное движение:

* Направление: (0, -2) – (1, -2) = (-1, 0)
* Новая точка: (-1, -2),

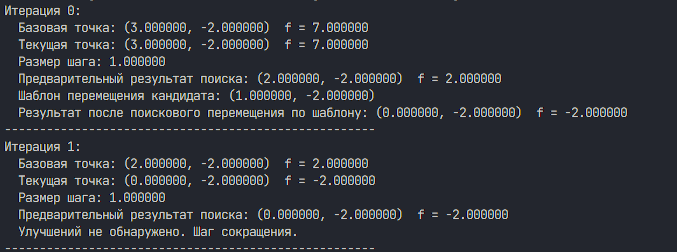
– хуже, оставляем (0, -2)

Алгоритм продолжается, шаг постепенно уменьшается, и при достижении получаем приближение к точке минимума.

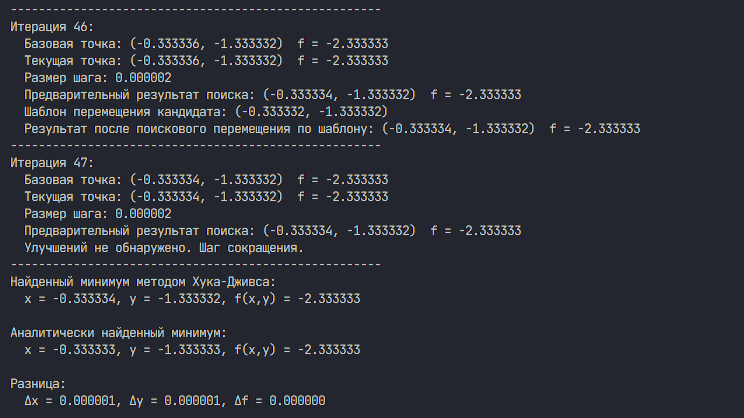
**Результат:**

Метод Хука–Дживса сходится к точке минимума (47 итераций):

## **Результат работы программного кода**



**Рисунок 1.3.1 – Начало работы алгоритма**

****

**Рисунок 1.3.2 – Результат работы алгоритма**

## **2 Метод наискорейшего градиентного спуска**

## **Цель и задачи работы**

Найти минимум функции

методом градиентного спуска с оптимальным выбором шага (наискорейший спуск) с точностью .

## **Ручной расчет**

Зададим начальные значения параметров:

* Размерность задачи оптимизации: n = 2
* Начальная точка многогранника: x0 = [3; 2]
* Точность поиска:

Градиент функции:

**Итерация: 0:**

* Текущая точка (3.0, -2.0)
* Значение функции:
* Градиент: (6.0, 2.0)
* Норма градиента: 6.324555
* Оптимальный шаг: 𝑡 = 0.384615

**Итерация: 1:**

* Текущая точка (0.692308, -2.769231)
* Значение функции:
* Градиент: (0.615385, -1.846154)
* Норма градиента: 1.946017
* Оптимальный шаг: 𝑡 = 0.714286

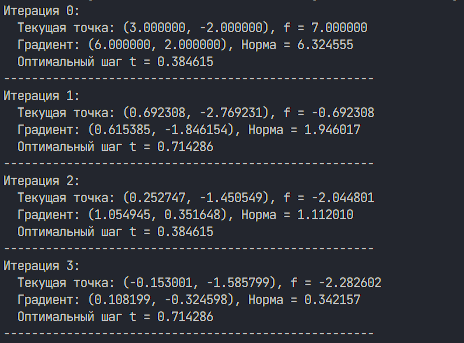
**Итерация: 2:**

* Текущая точка (0.252747,−1.450549)
* Значение функции:
* Градиент: (1.054945,0.351648)
* Норма градиента: 1.112010
* Оптимальный шаг: 𝑡 = 0.384615

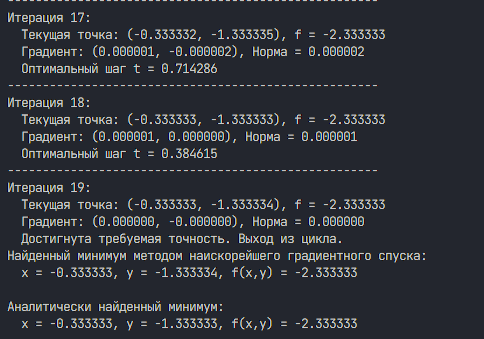
Алгоритм продолжается до выполнения условия .

**Результат**:  
Результат работы наискорейшего градиентного спуска (19 итерация):

## **Результат работы программного кода**



**Рисунок 2.3.1 – Начало работы алгоритма**

****

**Рисунок 2.3.2 – Результат работы алгоритма**

## **3 Метод Ньютона-Рафсона**

## **3.1 Цель и задачи работы**

Найти минимум функции

методом сопряжённых градиентов Флетчера–Ривса с точностью .

## **3.2 Ручной расчет**

Зададим начальные значения параметров:

* Размерность задачи оптимизации: n = 2
* Начальная точка многогранника: x0 = [3; -2]
* Точность поиска:

**Итерация 0**

* Текущая точка: (3.0, -2.0)
* Значение функции:
* Градиент: (6.000000, 2.000000), Норма = 6.324555
* Шаг Ньютона: (dx, dy) = (3.333333, -0.666667)
* Обновленная точка: (-0.333333, -1.333333),

**Итерация 1**

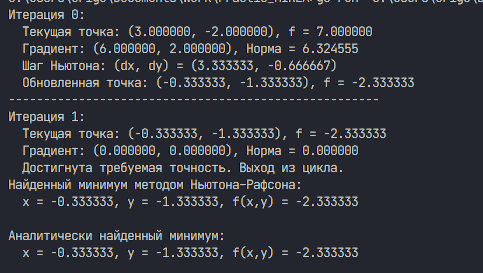
* Текущая точка: (-0.333333, -1.333333)
* Значение функции:
* Градиент: (0.000000, 0.000000), Норма = 0.000000

Достигнута требуемая точность

Результат:  
 Найденный минимум методом Ньютона–Рафсона:

x = -0.333333, y = -1.333333, f(x,y) = -2.333333

## **3.3 Результат работы программного кода**



**Рисунок 3.3.1 – Результат работы алгоритма**

**ЗАКЛЮЧЕНИЕ**

На основе проведённых экспериментов можно сравнить эффективность различных методов оптимизации при поиске минимума функции.

1. Точность результата:

- Метод Хука–Дживса: достиг точного результата (-2.333333).

- Метод наискорейшего градиентного спуска: достиг точного результата (-2.333333).

- Метод Ньютона-Рафсона: достиг точного результата (-2.333333).

2. Количество итераций:

- Метод Хука–Дживса: 47 итераций.

- Метод наискорейшего градиентного спуска: 19 итераций.

- Метод Ньютона-Рафсона: 2 итераций.

3. Точность точки минимума:

- Метод Хука–Дживса: точка [3, -2], точно совпадает с аналитическим решением.

- Метод наискорейшего градиентного спуска: точка [3, -2], точно совпадает с аналитическим решением.

- Метод Ньютона-Рафсона: точка [3, -2], точно совпадает с аналитическим решением.

**СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ**

1. Сорокин А.Б., Платонова О.В., ЖелезнякЛ.М., Безусловная оптимизация [Электронный ресурс] методическое пособие пособие / А. Б. Сорокин, О. В. Платонова, Л. М. Железняк . - М.: РТУ МИРЭА, 2020.
2. Хоботов, Е. Н. Лекции по дисциплине Математическое обеспечение систем поддержки принятия решений [Электронный ресурс] учебно-метод. пособие / Е. Н. Хоботов. - М.: РТУ МИРЭА, 2020.

**ПРИЛОЖЕНИЯ**

Приложение А - Код программы. Метод Хука–Дживса.

Приложение Б - Код программы. Метода наискорейшего градиентного спуска.

Приложение В - Код программы. Метода Ньютона-Рафсона.

**Приложение А**

Метод Хука–Дживса.

*Листинг А.1 - Метод Хука–Дживса*

|  |
| --- |
| package main  import (  "fmt"  "math"  )  func f(x, y float64) float64 {  return x\*x + x\*y + y\*y + 2\*x + 3\*y  }  func exploratorySearch(x, y, step float64) (float64, float64) {  bestX, bestY := x, y  bestVal := f(x, y)  // Поиск по оси X: сначала в положительном направлении  trialX := bestX + step  trialVal := f(trialX, bestY)  if trialVal < bestVal {  bestX = trialX  bestVal = trialVal  } else {  // Если не улучшилось, пробуем в отрицательном направлении  trialX = bestX - step  trialVal = f(trialX, bestY)  if trialVal < bestVal {  bestX = trialX  bestVal = trialVal  }  }  // Поиск по оси Y: сначала в положительном направлении  trialY := bestY + step  trialVal = f(bestX, trialY)  if trialVal < bestVal {  bestY = trialY  bestVal = trialVal  } else {  // Если не улучшилось, пробуем в отрицательном направлении  trialY = bestY - step  trialVal = f(bestX, trialY)  if trialVal < bestVal {  bestY = trialY  bestVal = trialVal  }  }  return bestX, bestY  }  func hookeJeeves(initialX, initialY, step, tol float64, maxIter int) (float64, float64, float64) {  baseX, baseY := initialX, initialY  newX, newY := initialX, initialY  iter := 0 |

*Продолжение листинга А.1*

|  |
| --- |
| for step > tol && iter < maxIter {  fmt.Printf("Итерация %d:\n", iter)  fmt.Printf(" Базовая точка: (%.6f, %.6f) f = %.6f\n", baseX, baseY, f(baseX, baseY))  fmt.Printf(" Текущая точка: (%.6f, %.6f) f = %.6f\n", newX, newY, f(newX, newY))  fmt.Printf(" Размер шага: %.6f\n", step)  xExpl, yExpl := exploratorySearch(newX, newY, step)  fmt.Printf(" Предварительный результат поиска: (%.6f, %.6f) f = %.6f\n", xExpl, yExpl, f(xExpl, yExpl))  if f(xExpl, yExpl) >= f(newX, newY) {  fmt.Println(" Улучшений не обнаружено. Шаг сокращения.")  step = step / 2.0  } else {  patternX := xExpl + (xExpl - baseX)  patternY := yExpl + (yExpl - baseY)  fmt.Printf(" Шаблон перемещения кандидата: (%.6f, %.6f)\n", patternX, patternY)  newPatternX, newPatternY := exploratorySearch(patternX, patternY, step)  fmt.Printf(" Результат после поискового перемещения по шаблону: (%.6f, %.6f) f = %.6f\n", newPatternX, newPatternY, f(newPatternX, newPatternY))  baseX, baseY = xExpl, yExpl  newX, newY = newPatternX, newPatternY  }  fmt.Println("-----------------------------------------------------")  iter++  }  return newX, newY, f(newX, newY)  }  func main() {  initialX := 3.0  initialY := -2.0  step := 1.0  tol := 1e-6  maxIter := 10000  bestX, bestY, bestVal := hookeJeeves(initialX, initialY, step, tol, maxIter)  fmt.Printf("Найденный минимум методом Хука-Дживса:\n")  fmt.Printf(" x = %.6f, y = %.6f, f(x,y) = %.6f\n", bestX, bestY, bestVal)  analyticX := -1.0 / 3.0  analyticY := -4.0 / 3.0  analyticVal := f(analyticX, analyticY)  fmt.Printf("\nАналитически найденный минимум:\n")  fmt.Printf(" x = %.6f, y = %.6f, f(x,y) = %.6f\n", analyticX, analyticY, analyticVal)  diffX := math.Abs(bestX - analyticX)  diffY := math.Abs(bestY - analyticY)  diffVal := math.Abs(bestVal - analyticVal)  fmt.Printf("\nРазница:\n Δx = %.6f, Δy = %.6f, Δf = %.6f\n", diffX, diffY, diffVal)  } |

**Приложение Б**

Метод наискорейшего градиентного спуска.

*Листинг Б.1 - Метод наискорейшего градиентного спуска*

|  |
| --- |
| package main  import (  "fmt"  "math"  )  // f вычисляет значение функции:  // f(x, y) = x^2 + x\*y + y^2 + 2\*x + 3\*y  func f(x, y float64) float64 {  return x\*x + x\*y + y\*y + 2\*x + 3\*y  }  // steepestDescent реализует метод наискорейшего градиентного спуска.  // initialX, initialY — начальная точка; tol — допустимая точность;  // maxIter — максимальное число итераций.  func steepestDescent(initialX, initialY, tol float64, maxIter int) (float64, float64, float64) {  x, y := initialX, initialY  iter := 0  for iter < maxIter {  // Вычисляем градиент: grad = [∂f/∂x, ∂f/∂y]  gradX := 2\*x + y + 2 // ∂f/∂x = 2x + y + 2  gradY := x + 2\*y + 3 // ∂f/∂y = x + 2y + 3  normGrad := math.Sqrt(gradX\*gradX + gradY\*gradY)  fmt.Printf("Итерация %d:\n", iter)  fmt.Printf(" Текущая точка: (%.6f, %.6f), f = %.6f\n", x, y, f(x, y))  fmt.Printf(" Градиент: (%.6f, %.6f), Норма = %.6f\n", gradX, gradY, normGrad)  // Если норма градиента меньше tol, считаем, что достигнута сходимость.  if normGrad < tol {  fmt.Println(" Достигнута требуемая точность. Выход из цикла.")  break  }  // Для квадратной функции оптимальный шаг можно вычислить аналитически.  // Оптимальный шаг по направлению антиградиента:  // t = (||grad||^2) / (grad^T \* H \* grad)  // Для H = [[2, 1], [1, 2]]:  // grad^T \* H \* grad = 2\*gradX^2 + 2\*gradX\*gradY + 2\*gradY^2.  gradNormSq := gradX\*gradX + gradY\*gradY  denom := 2 \* (gradX\*gradX + gradX\*gradY + gradY\*gradY)  t := gradNormSq / denom  fmt.Printf(" Оптимальный шаг t = %.6f\n", t)  // Обновляем точку по правилу: x\_new = x - t \* grad  x = x - t\*gradX |

*Продолжение листинга Б.1*

|  |
| --- |
| y = y - t\*gradY  fmt.Println("-----------------------------------------------------")  iter++  }  return x, y, f(x, y)  }  func main() {  // Задаём начальную точку (одинаковую для всех работ)  initialX := 3.0  initialY := -2.0  // Параметры алгоритма: допустимая точность и максимальное число итераций  tol := 1e-6  maxIter := 10000  // Запускаем метод наискорейшего градиентного спуска  bestX, bestY, bestVal := steepestDescent(initialX, initialY, tol, maxIter)  fmt.Printf("Найденный минимум методом наискорейшего градиентного спуска:\n")  fmt.Printf(" x = %.6f, y = %.6f, f(x,y) = %.6f\n", bestX, bestY, bestVal)  // Ручной (аналитический) расчёт для проверки:  analyticX := -1.0 / 3.0  analyticY := -4.0 / 3.0  analyticVal := f(analyticX, analyticY)  fmt.Printf("\nАналитически найденный минимум:\n")  fmt.Printf(" x = %.6f, y = %.6f, f(x,y) = %.6f\n", analyticX, analyticY, analyticVal)  } |

**Приложение В**

Метод Ньютона-Рафсона.

*Листинг В.1 - Метод Ньютона-Рафсона*

|  |
| --- |
| package main  import (  "fmt"  "math"  )  // f вычисляет значение функции:  // f(x, y) = x^2 + x\*y + y^2 + 2\*x + 3\*y  func f(x, y float64) float64 {  return x\*x + x\*y + y\*y + 2\*x + 3\*y  }  // newtonRaphson реализует метод Ньютона–Рафсона для минимизации функции f.  // initialX, initialY — начальная точка; tol — допустимая точность;  // maxIter — максимальное число итераций.  func newtonRaphson(initialX, initialY, tol float64, maxIter int) (float64, float64, float64) {  x, y := initialX, initialY  iter := 0  for iter < maxIter {  // Вычисляем градиент:  // ∂f/∂x = 2\*x + y + 2, ∂f/∂y = x + 2\*y + 3  gradX := 2\*x + y + 2  gradY := x + 2\*y + 3  normGrad := math.Sqrt(gradX\*gradX + gradY\*gradY)  fmt.Printf("Итерация %d:\n", iter)  fmt.Printf(" Текущая точка: (%.6f, %.6f), f = %.6f\n", x, y, f(x, y))  fmt.Printf(" Градиент: (%.6f, %.6f), Норма = %.6f\n", gradX, gradY, normGrad)  // Если норма градиента меньше tol, считаем, что достигнута сходимость  if normGrad < tol {  fmt.Println(" Достигнута требуемая точность. Выход из цикла.")  break  }  // Для функции f матрица Гессе равна:  // H = [ [2, 1],  // [1, 2] ]  // Ее обратная матрица:  // H⁻¹ = 1/3 \* [ [2, -1],  // [-1, 2] ]  //  // Обновление по методу Ньютона–Рафсона:  // [x\_new, y\_new]^T = [x, y]^T - H⁻¹ \* grad  //  // Вычисляем компоненты шага: |

*Продолжение листинга В.1*

|  |
| --- |
| // dx = (1/3) \* (2\*gradX - gradY)  // dy = (1/3) \* (-gradX + 2\*gradY)  dx := (1.0 / 3.0) \* (2\*gradX - gradY)  dy := (1.0 / 3.0) \* (-gradX + 2\*gradY)  fmt.Printf(" Шаг Ньютона: (dx, dy) = (%.6f, %.6f)\n", dx, dy)  // Обновляем точку  x = x - dx  y = y - dy  fmt.Printf(" Обновленная точка: (%.6f, %.6f), f = %.6f\n", x, y, f(x, y))  fmt.Println("-----------------------------------------------------")  iter++  }  return x, y, f(x, y)  }  func main() {  // Задаём начальную точку (одинаковую для всех работ)  initialX := 3.0  initialY := -2.0  // Параметры: допустимая точность и максимальное число итераций  tol := 1e-6  maxIter := 10000  // Запускаем метод Ньютона–Рафсона  bestX, bestY, bestVal := newtonRaphson(initialX, initialY, tol, maxIter)  fmt.Printf("Найденный минимум методом Ньютона–Рафсона:\n")  fmt.Printf(" x = %.6f, y = %.6f, f(x,y) = %.6f\n", bestX, bestY, bestVal)  // Ручной (аналитический) расчёт для проверки:  // Решая систему ∇f(x)=0:  // 2x + y + 2 = 0  // x + 2y + 3 = 0  // Получаем x = -1/3, y = -4/3, и f(-1/3,-4/3) = -7/3 ≈ -2.333333  analyticX := -1.0 / 3.0  analyticY := -4.0 / 3.0  analyticVal := f(analyticX, analyticY)  fmt.Printf("\nАналитически найденный минимум:\n")  fmt.Printf(" x = %.6f, y = %.6f, f(x,y) = %.6f\n", analyticX, analyticY, analyticVal)  } |