|  |  |  |
| --- | --- | --- |
| Министерство науки и высшего образования  Российской Федерации | | |
| Федеральное государственное бюджетное  образовательное учреждение высшего образования | | |
| «Новосибирский государственный технический университет» | | |
|  | | |
| Кафедра теоретической и прикладной информатики | | |
|  | | |
| Лабораторная работа № 3 | | |
| по дисциплине «Методы оптимизации» | | |
|  | | |
| **Метод штрафных функций** | | |
|  | | |
|  | Факультет: | ПМИ |
| Группа: | ПМИ-72 |
| Вариант: | 1 |
| Студент: | Сычев Егор |
|  |  |
|  |  |
| Преподаватель: | Постовалов Сергей Николаевич |
|  |  |
|  | | |
| Новосибирск | | |
| 2020 | | |

1. **Цель работы**

Ознакомиться с методами штрафных функций при решении задач нелинейного программирования. Изучить типы штраф­ных и барьерных функций, их особенности, способы и области применения, влияние штрафных функций на сходимость алгоритмов, зависимость точности решения задачи нелинейного программирования от величины коэффициента штрафа.

1. **Задания**

|  |  |
| --- | --- |
| № | Вид работы |
|  | Применяя методы поиска минимума 0-го порядка, реализовать программу для решения задачи нелиней­ного программирования с использованием **метода штрафных функций**. |
|  | Исследовать сходимость **метода штрафных функций** в зависимости от   * выбора штрафных функций, * начальной величины коэффициента штрафа, * стратегии изменения коэффициента штрафа, * начальной точки, * задаваемой точности .   Сформулировать выводы. |
| 3\* | Применяя методы поиска минимума 0-го порядка, реализовать программу для решения задачи нелиней­ного программирования с ограничением типа неравенства (**только пункт а**) с использованием **метода барьерных функций**. |
| 4\* | Исследовать сходимость **метода барьерных функций** (**только пункт а**) в зависимости от   * выбора барьерных функций, * начальной величины коэффициента штрафа, * стратегии изменения коэффициента штрафа, * начального приближения, * задаваемой точности .   Сформулировать выводы. |



при ограничении:

а) 

б) 

1. **Ход работы**
   1. **Метод штрафных функций**
      1. **Ограничение неравенством**
         1. **Стандартные параметры**

G(g(x)) = 0.5 \* (g(x) – |g(x)|)

r0 = 1

r = R(r) = r \* 2

x0 = (0, 0)

ε = 0.001

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| function calls | iterations | x | y | f(x, y) |
| 269 | 2 | 0,567 | 0,434 | 2,143 |

* + - 1. **Выбор штрафной функции**

G(g(x)) = (0.5 \* (g(x) - |g(x)|)2

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| function calls | iterations | X | y | f(x, y) |
| 218 | 13 | 0,563 | 0,437 | 2,144 |

G(g(x)) = (0.5 \* (g(x) - |g(x)|)100

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| function calls | iterations | X | y | f(x, y) |
| 1682 | 996 | 0,46 | 0,541 | 2,406 |

* + - 1. **Выбор начальной величины коэффициента штрафа**

r0 = 2-32

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| function calls | iterations | x | y | f(x, y) |
| 269 | 2 | 0,567 | 0,434 | 2,143 |

r0 = 232

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| function calls | iterations | x | y | f(x, y) |
| 269 | 2 | 0,567 | 0,434 | 2,143 |

* + - 1. **Выбор стратегии изменения коэффициента штрафа**

R(r) = r + r / (1 + r)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| function calls | iterations | x | y | | f(x, y) |
| 346 | 2 | 0,573 | 0,427 | 2,143 | | |

R(r) = r2 \* 2

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| function calls | iterations | x | y | f(x, y) |
| 269 | 2 | 0,567 | 0,434 | 2,143 |

* + - 1. **Выбор начального приближения**

x0 = (1, 1)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| function calls | iterations | x | y | f(x, y) |
| 146 | 2 | 0,559 | 0,441 | 2,146 |

x0 = (100, -100)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| function calls | iterations | x | y | f(x, y) |
| 928 | 2 | 0,567 | 0,433 | 2,143 |

* + - 1. **Выбор точности**

ε = 1e-5

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| function calls | iterations | x | y | f(x, y) |
| 1011 | 2 | 0,56344 | 0,43656 | 2,1442 |

ε = 1e-7

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| function calls | iterations | x | y | f(x, y) |
| 2232 | 2 | 0,563426 | 0,436574 | 2,144202 |

* + 1. **Ограничение равенством**
       1. **Стандартные параметры**

H(h(x)) = |h(x)|

r0 = 1

r = R(r) = r \* 2

x0 = (0, 0)

ε = 0.001

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| function calls | iterations | x | y | f(x, y) |
| 451 | 2 | 0,098 | -0,098 | 3,81 |

* + - 1. **Выбор штрафной функции**

G(g(x)) = (0.5 \* (g(x) - |g(x)|)2

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| function calls | iterations | X | y | f(x, y) |
| 213 | 12 | 0,086 | -0,085 | 3,811 |

G(g(x)) = (0.5 \* (g(x) - |g(x)|)100

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| function calls | iterations | X | y | f(x, y) |
| 559 | 90 | 0,069 | -0,068 | 3,823 |

* + - 1. **Выбор начальной величины коэффициента штрафа**

r0 = 2-32

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| function calls | iterations | x | y | f(x, y) |
| 451 | 2 | 0,098 | -0,098 | 3,81 |

r0 = 232

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| function calls | iterations | x | y | f(x, y) |
| 451 | 2 | 0,098 | -0,098 | 3,81 |

* + - 1. **Выбор стратегии изменения коэффициента штрафа**

R(r) = r + r / (1 + r)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| function calls | iterations | x | y | | f(x, y) |
| 559 | 3 | 0,087 | -0,087 | 3,811 | | |

R(r) = r2 \* 2

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| function calls | iterations | x | y | f(x, y) |
| 451 | 2 | 0,098 | -0,098 | 3,81 |

* + - 1. **Выбор начального приближения**

x0 = (1, 1)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| function calls | iterations | x | y | f(x, y) |
| 280 | 2 | 0,099 | -0,098 | 3,809 |

x0 = (100, -100)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| function calls | iterations | x | y | f(x, y) |
| 1065 | 2 | 0,101 | -0,1 | 3,809 |

* + - 1. **Выбор точности**

ε = 1e-5

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| function calls | iterations | x | y | f(x, y) |
| 1422 | 2 | 0,09718 | -0,09718 | 3,8096 |

ε = 1e-7

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| function calls | iterations | x | y | f(x, y) |
| 2876 | 2 | 0,09719 | -0,09719 | 3,809604 |

* 1. **Метод барьерных функций**
     1. **Ограничение неравенством**
        1. **Стандартные параметры**

G(g(x)) = -1 / g(x)

r0 = 1

r = R(r) = r / 2

x0 = (0, 0)

ε = 0.001

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| function calls | iterations | x | y | f(x, y) |
| 22 | 1 | 0,215 | 0,036 | 3,347 |

* + - 1. **Выбор штрафной функции**

G(g(x)) = (-1 / g(x))3

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| function calls | iterations | X | y | f(x, y) |
| 18 | 1 | 0,041 | -0,154 | 4,028 |

G(g(x)) = -ln(-g(x))

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| function calls | iterations | X | y | f(x, y) |
| 45 | 1 | 0,293 | 0,122 | 3,062 |

* + - 1. **Выбор начальной величины коэффициента штрафа**

r0 = 2-32

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| function calls | iterations | x | y | f(x, y) |
| 22 | 1 | 0,215 | 0,036 | 3,347 |

r0 = 232

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| function calls | iterations | x | y | f(x, y) |
| 22 | 1 | 0,215 | 0,036 | 3,347 |

* + - 1. **Выбор стратегии изменения коэффициента штрафа**

R(r) = r – r / (1 + r)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| function calls | iterations | x | y | | f(x, y) |
| 22 | 1 | 0,215 | 0,036 | 3,347 | |

R(r) = |sin(r)|

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| function calls | iterations | x | y | f(x, y) |
| 22 | 1 | 0,215 | 0,036 | 3,347 |

* + - 1. **Выбор начального приближения**

x0 = (1, 1)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| function calls | iterations | x | y | f(x, y) |
| 10216 | 10000 | 1,978 | 1,974 | 0,001 |

x0 = (100, -100)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| function calls | iterations | x | y | f(x, y) |
| 696 | 1 | 0,215 | 0,036 | 3,345 |

* + - 1. **Выбор точности**

ε = 1e-5

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| function calls | iterations | x | y | f(x, y) |
| 87 | 1 | 0,21501 | 0,03651 | 3,34552 |

ε = 1e-7

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| function calls | iterations | x | y | f(x, y) |
| 184 | 1 | 0,215008 | 0,036509 | 3,345506 |

1. **Выводы**

Для всех методов и всех ограничений, существенно на сходимость метода штрафных/барьерных функций влияет только сама штрафная/барьерная функция: чем сильнее она увеличивала штраф, тем больше было нарушений и тем менее точное получалось решение.

Начальные значения коэффициентов ни на что не влияет.

Стратегия изменения штрафа и точность повлияли только на сходимость метода спуска.

Выбор начального приближения для метода штрафных функций влияет только на сходимость используемого метода спуска (метод Гаусса). В случае барьерной функции, выбор начального приближения, не удовлетворяющего условию, делает невозможным решение задачи, так как за пределами допустимой области барьерные функции не определены.

Также с помощью барьерных функций удалось получить ответ всего за одну итерацию, вместо двух для штрафных функций, но из-за этого результат оказался значительно хуже.

1. **Текст программы**

**minimize.rs**

pub struct IterationResult<X> {

    x: X,

    dx: X,

    func\_calls: usize,

    is\_extra: bool,

}

impl<X> IterationResult<X>

where

    X: Clone,

{

    pub fn new(x: X, dx: X, func\_calls: usize, is\_extra: bool) -> Self {

        Self {

            x,

            dx,

            func\_calls,

            is\_extra,

        }

    }

    pub fn x(&self) -> X {

        self.x.clone()

    }

    pub fn dx(&self) -> X {

        self.dx.clone()

    }

    pub fn func\_calls(&self) -> usize {

        self.func\_calls

    }

    pub fn is\_extra(&self) -> bool {

        self.is\_extra

    }

}

pub struct FinalResult<X> {

    x: X,

    iters: usize,

    func\_calls: usize,

}

impl<X> FinalResult<X>

where

    X: Clone,

{

    pub fn new(x: X, iters: usize, func\_calls: usize) -> Self {

        Self {

            x,

            iters,

            func\_calls,

        }

    }

    pub fn x(&self) -> X {

        self.x.clone()

    }

    pub fn func\_calls(&self) -> usize {

        self.func\_calls

    }

    pub fn iters(&self) -> usize {

        self.iters

    }

}

pub trait Minimize<X: Clone>: Iterator<Item = IterationResult<X>> {

    fn x(&self) -> X;

    fn dx(&self) -> X;

    fn func\_calls(&self) -> usize;

    fn iters(&self) -> usize;

    fn result(&mut self) -> FinalResult<X> {

        let x = self.x();

        self.take\_while(|i| !i.is\_extra())

            .fold(FinalResult::new(x, 0, 0), |result, i| {

                FinalResult::new(i.x(), result.iters + 1, result.func\_calls + i.func\_calls())

            })

    }

}

**one\_dimension\_searchers.rs**

use super::minimize;

use super::minimize::{FinalResult, IterationResult};

use nalgebra::RealField;

use std::sync::Arc;

fn get\_interval<Scalar>(

    f: Arc<dyn Fn(Scalar) -> Scalar>,

    x0: Scalar,

    delta: Scalar,

) -> (Scalar, Scalar, usize)

where

    Scalar: RealField,

{

    let mut h;

    let mut x = x0;

    let mut func\_calls = 2;

    let mut f1 = f(x);

    let mut f2 = f(x + delta);

    if f1 > f2 {

        h = delta;

    } else {

        func\_calls += 1;

        f2 = f(x - delta);

        if f1 > f2 {

            h = -delta;

        } else {

            return (x - delta, x + delta, 3);

        }

    }

    x = x + h;

    while f1 > f2 {

        h = h \* Scalar::from\_i8(2).unwrap();

        x = x + h;

        f1 = f2;

        f2 = f(x);

        func\_calls += 1;

    }

    let left = (x - Scalar::from\_f64(3. / 2.).unwrap() \* h).min(x);

    let right = (x - Scalar::from\_f64(3. / 2.).unwrap() \* h).max(x);

    (left, right, func\_calls)

}

trait OneDimensionalMinimize<Scalar>: minimize::Minimize<Scalar>

where

    Scalar: Clone,

{

}

#[derive(Clone)]

struct Fibonacci<Scalar> {

    f: Arc<dyn Fn(Scalar) -> Scalar>,

    left: Scalar,

    right: Scalar,

    fn1: u128,

    fn2: u128,

    x1: Scalar,

    x2: Scalar,

    f1: Scalar,

    f2: Scalar,

    dx: Scalar,

    iters: usize,

    func\_calls: usize,

    eps: Scalar,

    max\_iters: usize,

}

impl<Scalar> Fibonacci<Scalar>

where

    Scalar: RealField,

{

    fn new(

        left: Scalar,

        right: Scalar,

        f: Arc<dyn Fn(Scalar) -> Scalar>,

        eps: Scalar,

        max\_iters: usize,

    ) -> Self {

        let mut fn1 = 1u128;

        let mut fn2 = 1u128;

        while Scalar::from\_u128(fn2).unwrap() <= (right - left) / eps {

            let ft = fn2;

            fn2 += fn1;

            fn1 = ft;

        }

        let x1 = left

            + Scalar::from\_u128(fn2 - fn1).unwrap() / Scalar::from\_u128(fn2).unwrap()

                \* (right - left);

        let x2 = left

            + Scalar::from\_u128(fn1).unwrap() / Scalar::from\_u128(fn2).unwrap() \* (right - left);

        let f1 = f(x1);

        let f2 = f(x2);

        Self {

            left,

            right,

            f,

            fn1,

            fn2,

            f1,

            f2,

            x1,

            x2,

            dx: Scalar::max\_value(),

            iters: 0,

            func\_calls: 2,

            eps,

            max\_iters,

        }

    }

}

impl<Scalar> OneDimensionalMinimize<Scalar> for Fibonacci<Scalar> where Scalar: RealField {}

impl<Scalar> minimize::Minimize<Scalar> for Fibonacci<Scalar>

where

    Scalar: RealField,

{

    fn iters(&self) -> usize {

        self.iters

    }

    fn func\_calls(&self) -> usize {

        self.func\_calls

    }

    fn x(&self) -> Scalar {

        (self.right + self.left) / Scalar::from\_i8(2).unwrap()

    }

    fn dx(&self) -> Scalar {

        self.dx

    }

}

impl<Scalar> Iterator for Fibonacci<Scalar>

where

    Scalar: RealField,

{

    type Item = IterationResult<Scalar>;

    fn next(&mut self) -> Option<Self::Item> {

        if self.fn1 == 0 {

            return None;

        }

        let is\_extra = if self.dx.abs() < self.eps || self.iters >= self.max\_iters {

            true

        } else {

            false

        };

        let \_x = (self.right + self.left) / Scalar::from\_i8(2).unwrap();

        if self.f1 < self.f2 {

            self.right = self.x2;

            self.x2 = self.x1;

            self.x1 = self.left

                + Scalar::from\_u128(self.fn2 - self.fn1).unwrap()

                    / Scalar::from\_u128(self.fn2).unwrap()

                    \* (self.right - self.left);

            self.f2 = self.f1;

            self.f1 = (self.f)(self.x1);

        } else {

            self.left = self.x1;

            self.x1 = self.x2;

            self.x2 = self.left

                + Scalar::from\_u128(self.fn1).unwrap() / Scalar::from\_u128(self.fn2).unwrap()

                    \* (self.right - self.left);

            self.f1 = self.f2;

            self.f2 = (self.f)(self.x2);

        }

        let ft = self.fn1;

        self.fn1 = self.fn2 - self.fn1;

        self.fn2 = ft;

        let x = (self.right + self.left) / Scalar::from\_i8(2).unwrap();

        let \_dx = self.dx;

        self.dx = x - \_x;

        self.iters += 1;

        self.func\_calls += 1;

        Some(IterationResult::new(x, x - \_x, 1, is\_extra))

    }

}

pub enum Method {

    Fibonacci,

}

pub struct Minimize<Scalar>

where

    Scalar: RealField,

{

    x: Scalar,

    dx: Scalar,

    func\_calls: usize,

    iters: usize,

    method: Box<dyn OneDimensionalMinimize<Scalar>>,

}

impl<Scalar> Minimize<Scalar>

where

    Scalar: RealField,

{

    pub fn new(

        x0: Scalar,

        f: Arc<dyn Fn(Scalar) -> Scalar>,

        method: Method,

        eps: Scalar,

        max\_iters: usize,

    ) -> Self {

        let (left, right, func\_calls) = get\_interval(f.clone(), x0, eps);

        let m = match method {

            Method::Fibonacci => Fibonacci::new(left, right, f, eps, max\_iters),

        };

        Self {

            x: (right + left) / Scalar::from\_i8(2).unwrap(),

            dx: Scalar::max\_value(),

            func\_calls: func\_calls + m.func\_calls,

            iters: 0,

            method: Box::new(m),

        }

    }

    pub fn result(

        x0: Scalar,

        f: Arc<dyn Fn(Scalar) -> Scalar>,

        method: Method,

        eps: Scalar,

        max\_iters: usize,

    ) -> FinalResult<Scalar> {

        <Self as minimize::Minimize<Scalar>>::result(&mut Self::new(x0, f, method, eps, max\_iters))

    }

}

impl<Scalar> minimize::Minimize<Scalar> for Minimize<Scalar>

where

    Scalar: RealField,

{

    fn iters(&self) -> usize {

        self.iters

    }

    fn func\_calls(&self) -> usize {

        self.func\_calls

    }

    fn x(&self) -> Scalar {

        self.x

    }

    fn dx(&self) -> Scalar {

        self.dx

    }

}

impl<Scalar> Iterator for Minimize<Scalar>

where

    Scalar: RealField,

{

    type Item = IterationResult<Scalar>;

    fn next(&mut self) -> Option<Self::Item> {

        match self.method.next() {

            Some(r) => {

                self.iters += 1;

                self.x = r.x();

                self.dx = r.dx();

                self.func\_calls += r.func\_calls();

                Some(IterationResult::new(

                    self.x,

                    self.dx,

                    r.func\_calls(),

                    r.is\_extra(),

                ))

            }

            None => None,

        }

    }

}

**descent\_methods.rs**

use super::minimize;

use super::minimize::{FinalResult, IterationResult};

use super::one\_dimension\_searchers;

use nalgebra::{allocator::Allocator, DefaultAllocator, DimName, RealField, VectorN};

use std::sync::Arc;

trait DescentMethod<Scalar, Dimension>:

    Iterator<Item = IterationResult<VectorN<Scalar, Dimension>>>

    + minimize::Minimize<VectorN<Scalar, Dimension>>

where

    Scalar: RealField,

    Dimension: DimName,

    DefaultAllocator: Allocator<Scalar, Dimension>,

{

    #[allow(non\_snake\_case)]

    fn S(&self) -> VectorN<Scalar, Dimension>;

}

struct Gauss<Scalar, Dimension>

where

    Scalar: RealField,

    Dimension: DimName,

    DefaultAllocator: Allocator<Scalar, Dimension>,

{

    x: VectorN<Scalar, Dimension>,

    dx: VectorN<Scalar, Dimension>,

    func\_calls: usize,

    iters: usize,

    f: Arc<dyn Fn(VectorN<Scalar, Dimension>) -> Scalar>,

    eps: Scalar,

    max\_iters: usize,

}

impl<Scalar, Dimension> Gauss<Scalar, Dimension>

where

    Scalar: RealField,

    Dimension: DimName,

    DefaultAllocator: Allocator<Scalar, Dimension>,

{

    fn new(

        x0: VectorN<Scalar, Dimension>,

        f: Arc<dyn Fn(VectorN<Scalar, Dimension>) -> Scalar>,

        eps: Scalar,

        max\_iters: usize,

    ) -> Self {

        Self {

            x: x0.clone(),

            dx: VectorN::<Scalar, Dimension>::from\_element(Scalar::max\_value()),

            f,

            eps,

            iters: 0,

            func\_calls: 0,

            max\_iters,

        }

    }

}

impl<Scalar, Dimension> DescentMethod<Scalar, Dimension> for Gauss<Scalar, Dimension>

where

    Scalar: RealField,

    Dimension: DimName,

    DefaultAllocator: Allocator<Scalar, Dimension>,

{

    fn S(&self) -> VectorN<Scalar, Dimension> {

        let mut s = nalgebra::zero::<VectorN<Scalar, Dimension>>();

        s[self.iters % Dimension::dim()] = Scalar::one();

        s

    }

}

impl<Scalar, Dimension> minimize::Minimize<VectorN<Scalar, Dimension>> for Gauss<Scalar, Dimension>

where

    Scalar: RealField,

    Dimension: DimName,

    DefaultAllocator: Allocator<Scalar, Dimension>,

{

    fn iters(&self) -> usize {

        self.iters

    }

    fn func\_calls(&self) -> usize {

        self.func\_calls

    }

    fn x(&self) -> VectorN<Scalar, Dimension> {

        self.x.clone()

    }

    fn dx(&self) -> VectorN<Scalar, Dimension> {

        self.dx.clone()

    }

}

impl<Scalar, Dimension> Iterator for Gauss<Scalar, Dimension>

where

    Scalar: RealField,

    Dimension: DimName,

    DefaultAllocator: Allocator<Scalar, Dimension>,

{

    type Item = IterationResult<VectorN<Scalar, Dimension>>;

    fn next(&mut self) -> Option<Self::Item> {

        let is\_extra =

            if self.dx.iter().all(|xi| xi.abs() < self.eps) || self.iters >= self.max\_iters {

                true

            } else {

                false

            };

        let x = self.x.clone();

        let f = self.f.clone();

        let s = self.S();

        let lambda\_result = one\_dimension\_searchers::Minimize::result(

            Scalar::zero(),

            Arc::new(move |lambda| f(x.clone() + s.clone() \* lambda)),

            one\_dimension\_searchers::Method::Fibonacci,

            self.eps,

            self.max\_iters,

        );

        self.dx = self.S() \* lambda\_result.x();

        self.x += self.dx.clone();

        self.iters += 1;

        self.func\_calls += lambda\_result.func\_calls();

        Some(IterationResult::new(

            self.x.clone(),

            self.dx.clone(),

            lambda\_result.func\_calls(),

            is\_extra,

        ))

    }

}

#[derive(Clone)]

pub enum Method {

    Gauss,

}

pub struct Minimize<Scalar, Dimension>

where

    Scalar: RealField,

    Dimension: DimName,

    DefaultAllocator: Allocator<Scalar, Dimension>,

{

    x: VectorN<Scalar, Dimension>,

    dx: VectorN<Scalar, Dimension>,

    func\_calls: usize,

    iters: usize,

    method: Box<

        dyn DescentMethod<Scalar, Dimension, Item = IterationResult<VectorN<Scalar, Dimension>>>,

    >,

}

impl<Scalar, Dimension> Minimize<Scalar, Dimension>

where

    Scalar: RealField,

    Dimension: DimName,

    DefaultAllocator: Allocator<Scalar, Dimension>,

{

    pub fn new(

        x0: VectorN<Scalar, Dimension>,

        f: Arc<dyn Fn(VectorN<Scalar, Dimension>) -> Scalar>,

        method: Method,

        eps: Scalar,

        max\_iters: usize,

    ) -> Self {

        let m = match method {

            Method::Gauss => Gauss::new(x0.clone(), f, eps, max\_iters),

        };

        Self {

            x: x0,

            dx: VectorN::<Scalar, Dimension>::from\_element(Scalar::max\_value()),

            func\_calls: m.func\_calls,

            iters: 0,

            method: Box::new(m),

        }

    }

    pub fn result(

        x0: VectorN<Scalar, Dimension>,

        f: Arc<dyn Fn(VectorN<Scalar, Dimension>) -> Scalar>,

        method: Method,

        eps: Scalar,

        max\_iters: usize,

    ) -> FinalResult<VectorN<Scalar, Dimension>> {

        <Self as minimize::Minimize<VectorN<Scalar, Dimension>>>::result(&mut Self::new(

            x0, f, method, eps, max\_iters,

        ))

    }

}

impl<Scalar, Dimension> minimize::Minimize<VectorN<Scalar, Dimension>>

    for Minimize<Scalar, Dimension>

where

    Scalar: RealField,

    Dimension: DimName,

    DefaultAllocator: Allocator<Scalar, Dimension>,

{

    fn iters(&self) -> usize {

        self.iters

    }

    fn func\_calls(&self) -> usize {

        self.func\_calls

    }

    fn x(&self) -> VectorN<Scalar, Dimension> {

        self.x.clone()

    }

    fn dx(&self) -> VectorN<Scalar, Dimension> {

        self.dx.clone()

    }

}

impl<Scalar, Dimension> Iterator for Minimize<Scalar, Dimension>

where

    Scalar: RealField,

    Dimension: DimName,

    DefaultAllocator: Allocator<Scalar, Dimension>,

{

    type Item = IterationResult<VectorN<Scalar, Dimension>>;

    fn next(&mut self) -> Option<Self::Item> {

        match self.method.next() {

            Some(r) => {

                self.iters += 1;

                self.func\_calls += r.func\_calls();

                self.x = r.x();

                self.dx = r.dx();

                Some(IterationResult::new(

                    self.x.clone(),

                    self.dx.clone(),

                    r.func\_calls(),

                    r.is\_extra(),

                ))

            }

            None => None,

        }

    }

}

**penalty\_methods.rs**

use super::descent\_methods;

use super::descent\_methods::Method;

use super::minimize;

use super::minimize::{FinalResult, IterationResult};

use nalgebra::{allocator::Allocator, DefaultAllocator, DimName, RealField, VectorN};

use std::sync::Arc;

#[derive(Clone)]

pub enum BoundType {

    Equal,

    Unequal,

}

#[derive(Clone)]

pub struct Bound<Scalar, Dimension>

where

    Scalar: RealField,

    Dimension: DimName,

    DefaultAllocator: Allocator<Scalar, Dimension>,

{

    function: Arc<dyn Fn(VectorN<Scalar, Dimension>) -> Scalar>,

    bound\_type: BoundType,

    penalty: Arc<dyn Fn(Scalar) -> Scalar>,

    coefficient: Scalar,

    coefficient\_function: Arc<dyn Fn(Scalar) -> Scalar>,

}

impl<Scalar, Dimension> Bound<Scalar, Dimension>

where

    Scalar: RealField,

    Dimension: DimName,

    DefaultAllocator: Allocator<Scalar, Dimension>,

{

    pub fn new(

        function: Arc<dyn Fn(VectorN<Scalar, Dimension>) -> Scalar>,

        bound\_type: BoundType,

        penalty: Arc<dyn Fn(Scalar) -> Scalar>,

        coefficient: Scalar,

        coefficient\_function: Arc<dyn Fn(Scalar) -> Scalar>,

    ) -> Self {

        Self {

            function,

            bound\_type,

            penalty,

            coefficient,

            coefficient\_function,

        }

    }

}

pub struct Minimize<Scalar, Dimension>

where

    Scalar: RealField,

    Dimension: DimName,

    DefaultAllocator: Allocator<Scalar, Dimension>,

{

    x: VectorN<Scalar, Dimension>,

    dx: VectorN<Scalar, Dimension>,

    f: Arc<dyn Fn(VectorN<Scalar, Dimension>) -> Scalar>,

    func\_calls: usize,

    iters: usize,

    method: Method,

    max\_iters: usize,

    eps: Scalar,

    g: Vec<Bound<Scalar, Dimension>>,

    got\_result: bool,

}

impl<Scalar, Dimension> Minimize<Scalar, Dimension>

where

    Scalar: RealField,

    Dimension: DimName,

    DefaultAllocator: Allocator<Scalar, Dimension>,

{

    pub fn new(

        x0: VectorN<Scalar, Dimension>,

        f: Arc<dyn Fn(VectorN<Scalar, Dimension>) -> Scalar>,

        method: Method,

        g: Vec<Bound<Scalar, Dimension>>,

        eps: Scalar,

        max\_iters: usize,

    ) -> Self {

        Self {

            x: x0,

            f,

            dx: VectorN::<Scalar, Dimension>::from\_element(Scalar::max\_value()),

            iters: 0,

            func\_calls: 0,

            method,

            max\_iters,

            g,

            eps,

            got\_result: false,

        }

    }

    pub fn result(

        x0: VectorN<Scalar, Dimension>,

        f: Arc<dyn Fn(VectorN<Scalar, Dimension>) -> Scalar>,

        method: Method,

        g: Vec<Bound<Scalar, Dimension>>,

        eps: Scalar,

        max\_iters: usize,

    ) -> FinalResult<VectorN<Scalar, Dimension>> {

        <Self as minimize::Minimize<VectorN<Scalar, Dimension>>>::result(&mut Self::new(

            x0, f, method, g, eps, max\_iters,

        ))

    }

}

impl<Scalar, Dimension> minimize::Minimize<VectorN<Scalar, Dimension>>

    for Minimize<Scalar, Dimension>

where

    Scalar: RealField,

    Dimension: DimName,

    DefaultAllocator: Allocator<Scalar, Dimension>,

{

    fn iters(&self) -> usize {

        self.iters

    }

    fn func\_calls(&self) -> usize {

        self.func\_calls

    }

    fn x(&self) -> VectorN<Scalar, Dimension> {

        self.x.clone()

    }

    fn dx(&self) -> VectorN<Scalar, Dimension> {

        self.dx.clone()

    }

}

impl<Scalar, Dimension> Iterator for Minimize<Scalar, Dimension>

where

    Scalar: RealField,

    Dimension: DimName,

    DefaultAllocator: Allocator<Scalar, Dimension>,

{

    type Item = IterationResult<VectorN<Scalar, Dimension>>;

    fn next(&mut self) -> Option<Self::Item> {

        if self.g.iter().any(|g| {

            let \_g = (g.function)(self.x.clone());

            match g.bound\_type {

                BoundType::Equal => \_g.abs() >= self.eps,

                BoundType::Unequal => \_g >= self.eps,

            }

        }) && self

            .g

            .iter()

            .filter(|g| {

                let \_g = (g.function)(self.x.clone());

                match g.bound\_type {

                    BoundType::Equal => \_g.abs() >= self.eps,

                    BoundType::Unequal => \_g >= self.eps,

                }

            })

            .all(|r| !r.coefficient.is\_finite())

        {

            return None;

        }

        let g = self.g.clone();

        let f = self.f.clone();

        let result = descent\_methods::Minimize::result(

            self.x.clone(),

            Arc::new(move |x: VectorN<Scalar, Dimension>| -> Scalar {

                g.iter().fold(f(x.clone()), |result, i| {

                    result + i.coefficient \* (i.penalty)((i.function)(x.clone()))

                })

            }),

            self.method.clone(),

            self.eps,

            self.max\_iters,

        );

        self.func\_calls += result.func\_calls();

        let x = self.x.clone();

        self.x = result.x();

        self.dx = self.x.clone() - x;

        let x = self.x.clone();

        let eps = self.eps;

        let is\_extra = (self

            .g

            .iter\_mut()

            .filter(|g| {

                let \_g = (g.function)(x.clone());

                g.coefficient.is\_finite()

                    && match g.bound\_type {

                        BoundType::Equal => \_g.abs() >= eps,

                        BoundType::Unequal => \_g >= eps,

                    }

            })

            .fold(self.got\_result, |\_, r| {

                r.coefficient = (r.coefficient\_function)(r.coefficient);

                false

            })

            && self.iters > 0)

            || self.iters >= self.max\_iters;

        self.got\_result = self.g.iter().all(|g| {

            let \_g = (g.function)(self.x.clone());

            match g.bound\_type {

                BoundType::Equal => \_g.abs() < self.eps,

                BoundType::Unequal => \_g < self.eps,

            }

        });

        self.iters += 1;

        return Some(Self::Item::new(

            self.x.clone(),

            self.dx.clone(),

            result.func\_calls(),

            is\_extra,

        ));

    }

}