第 1 回 演習課題

1026-30-8137
多田 拓生[*1]

2020 年 10 月 29 日

[*1] tada.takumi.34w@st.kyoto-u.ac.jp

電気電子計算工学及演習

1026-30-8137
多田 拓生

説明日
2020/10/8

課題 1.1

二分法およびニュートン法を用いて非線形方程式を解くプログラムをそれぞれソースコード 1、ソースコード 2 に示す。

まず二分法を用いたソースコード 1 について説明する。

bisection_method 関数は、引数として range、e、f、expected_value を受け取る。これらはそれぞれ範囲、許容誤差、関数、真値である。まず初期区間を range として与えると、bisection_method は bisection_method_inner 関数に range、e、f、expected_value を渡し、さらに回数として times に 1 を、また反復回数と近似解のデータを書き込むバッファ data を渡す。bisection_method_inner 関数は範囲を半分に区切り、解が存在すると思われる範囲を再帰的に渡して times を一つ進める。この時その範囲が許容誤差内に収まったなら、半分に区切った時の値を近似解として返す。

次にニュートン法を用いたソースコード 2 について説明する。

まず、ニュートン法で非線形方程式を解くには関数を微分する必要がある。関数の微分には、微分係数の定義である

$$\lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \tag{1}$$

を用いて微分した関数を返す differential_f 関数を作成した。

newton_raphson_method 関数は次のようなアルゴリズムで方程式を解く。まず、関数には $f(x)$ と初期近似解を与える。すると、その関数を微分し、

$$g(x) = x - \frac{f(x)}{f'(x)} \tag{2}$$

となる $g(x)$ を計算する newton_transform 関数に $f(x)$、$f'(x)$ を渡し、また閾値と回数として 1、反復回数上限として 1,000,000、バッファとしての data、真値 expected_value とともに newton_method 関数に渡す。

newton_method 関数では、$g(x)$ を用いて近似解の候補を求め、元の x との距離が閾値よりも小さい時、その計算した値を近似解として返す。閾値よりも大きかった場合は計算した値を再帰的に

newton_method に渡す。それを繰り返すことで非線形方程式を解く。最初に next の値をチェックしているのは、$g(x)$ の値が想定していない値になった時の処理をまとめてあるだけであり、アルゴリズムに直接は影響しない。これについては後で言及する。

なお、バッファに保存した data は gnuplot を用いて描画する。

ソースコード 1　bisection_method.rs

```
#![allow(dead_code)]

pub use std::ops::Range;
pub use std::rc::Rc;

pub fn bisection_method(
    range: Range<f64>,
    e: f64,
    f: Rc<dyn Fn(f64) -> f64>,
    expected_value: f64,
) -> (f64, Vec<(f64, f64)>) {
    let data: Vec<(f64, f64)> = Vec::new();
    bisection_method_inner(
        range, e, f, 1, expected_value, data
    )
}

fn bisection_method_inner(
    mut range: Range<f64>,
    e: f64,
    f: Rc<dyn Fn(f64) -> f64>,
    times: usize,
    expected_value: f64,
    mut data: Vec<(f64, f64)>,
) -> (f64, Vec<(f64, f64)>) {
    let x_new = (range.end + range.start) / 2.;
    if f(x_new) * f(range.start) >= 0. {
```

```
            range.start = x_new;
        } else {
            range.end = x_new;
        }
        data.push((times as f64, (x_new - expected_value).abs()));
        if range.end - range.start <= e {
            (x_new, data)
        } else {
            bisection_method_inner(
                range, e, f, times + 1, expected_value, data
            )
        }
    }

    #[cfg(test)]
    mod tests_bisection_method {
        use crate::bisection_method::*;

        #[test]
        fn tests_bisection_method() {
            let f = Rc::new(|x: f64| {
                x.powf(5.) - 3. * x.powf(4.) + x.powf(3.)
                    + 5. * x.powf(2.) - 6. * x + 2.
            });
            assert_eq!(
                (bisection_method(
                    -2f64..0f64, 1e-3, f.clone(), -1.414213566237)
                ).0,
                -1.4150390625
            );
            assert_eq!(
                (bisection_method(
                    -2f64..0f64, 1e-4, f.clone(), -1.414213566237)
                ).0,
```

```
            -1.41424560546875
        );
        assert_eq!(
            (bisection_method(
                -2f64..0f64, 1e-5, f.clone(), -1.414213566237)
            ).0,
            -1.4142074584960938
        );
    }
}
```

ソースコード 2 newton_raphson_method.rs

```
#![allow(dead_code)]

// pub mod newton_raphson_method {
pub use std::rc::Rc;
pub use std::result::Result;

pub fn newton_raphson_method(
    f: Rc<dyn Fn(f64) -> f64>,
    init: f64,
    expected_value: f64,
) -> Result<(f64, Vec<(f64, f64)>), String> {
    let threshold = 0.1e-10;
    let f_dir = differential_f(f.clone());
    let data: Vec<(f64, f64)> = Vec::new();
    newton_method(
        newton_transform(f, f_dir),
        init,
        threshold,
        1,
        1_000_000,
        expected_value,
        data,
    )
```

```
}

fn differential_f(
    f: Rc<dyn Fn(f64) -> f64>
) -> Rc<dyn Fn(f64) -> f64> {
    let dx = 0.1e-10;
    let f_dir = move |x: f64| -> f64 { (f(x + dx) - f(x)) / dx };
    Rc::new(f_dir)
}

unsafe fn partial_derivative(
    f: Rc<dyn Fn(Vec<f64>) -> f64>,
    i: usize,
) -> Rc<dyn Fn(Vec<f64>) -> f64> {
    let dx = 0.1e-10;
    let f_der = move |v: Vec<f64>| -> f64 {
        let mut v_dx = v.clone();
        v_dx[i] += dx;
        (f(v_dx) - f(v)) / dx
    };
    Rc::new(f_der)
}

fn newton_transform(
    f: Rc<dyn Fn(f64) -> f64>,
    f_dir: Rc<dyn Fn(f64) -> f64>,
) -> Rc<dyn Fn(f64) -> f64> {
    Rc::new(move |x: f64| -> f64 { x - f(x) / f_dir(x) })
}

fn newton_method(
    f: Rc<dyn Fn(f64) -> f64>,
    guess: f64,
    threshold: f64,
```

```
    times: usize,
    limit: usize,
    expected_value: f64,
    mut data: Vec<(f64, f64)>,
) -> Result<(f64, Vec<(f64, f64)>), String> {
    let next = f(guess);
    if next == f64::NEG_INFINITY
    || next == f64::INFINITY
    || next.is_nan() {
        return Err(format!(
            "x^(k+1) is not a number: last value is {}.", guess)
        );
    }
    if limit == times + 1 {
        return Err(format!(
            "solution doesn't converge: last value is {}.",
            next
        ));
    }
    data.push((times as f64, (next - expected_value).abs()));
    if (next - guess).abs() <= threshold {
        Ok((next, data))
    } else {
        newton_method(
            f, next, threshold, times + 1,
            limit, expected_value, data
        )
    }
}

#[cfg(test)]
mod tests_newton_raphson_method {
    use crate::newton_raphson_method::newton_method;
    use crate::newton_raphson_method::*;
```

```
#[test]
fn test_newton_raphson_method_newton_raphson_method() {
    let f: Rc<dyn Fn(f64) -> f64> = Rc::new(|x: f64| -> f64 {
        x.powf(5.) - 3. * x.powf(4.) + x.powf(3.)
            + 5. * x.powf(2.) - 6. * x + 2.
    });
    assert_eq!(
        newton_raphson_method(f, -1., -1.414213566237).unwrap().0,
        -1.4142135623730951
    );
}

#[test]
fn test_newton_raphson_method_newton_method_neg_inf() {
    let f: Rc<dyn Fn(f64) -> f64> = Rc::new(|x: f64| -> f64 { x });
    assert_eq!(
        newton_method(
            f,
            f64::NEG_INFINITY,
            0.1e-10,
            1,
            10000,
            -1.41,
            vec![(0f64, 0f64)]
        ),
        Err("x^(k+1) is not a number: last value is -inf.".to_string())
    );
}

#[test]
fn test_newton_raphson_method_newton_method_inf() {
    let f: Rc<dyn Fn(f64) -> f64> = Rc::new(|x: f64| -> f64 { x });
    assert_eq!(
```

```
            newton_method(
                f,
                f64::INFINITY,
                0.1e-10,
                1,
                10000,
                -1.41,
                vec![(0f64, 0f64)]
            ),
            Err("x^(k+1) is not a number: last value is inf.".to_string())
        );
    }

    #[test]
    fn test_newton_raphson_method_newton_method_nan() {
        let f: Rc<dyn Fn(f64) -> f64> = Rc::new(|x: f64| -> f64 { x });
        assert_eq!(
            newton_method(f, f64::NAN, 0.1e-10, 1, 10000, -1.41, vec![(0f64, 0f6
            Err("x^(k+1) is not a number: last value is NaN.".to_string())
        );
    }
}
```

1 課題 1.1.1

$$f(x) = x^5 - 3x^4 + x^3 + 5x^2 - 6x + 2 \tag{3}$$

とする。5 次方程式 $f(x) = 0$ の解を最初に説明した二分法およびニュートン法を用いたプログラムを実行して解く。

二分法の初期期間を [-2, 0] とし、ニュートン法の初期近似解を-1 とする。そして反復回数を横軸に、それぞれの手法で得られた近似解と真値 $(-\sqrt{2})$ との誤差の絶対値を縦軸にとった片対数グラフをそれぞれ図 1 に作成し、示す。

図 1 より、二分法 (青色) は収束にこそ時間がかかるが比較的誤差の大きさは小さいので初めからある程度は真値近くの値を示すのに対し、ニュートン法では収束の速さが速いが収束する前は真値とよりかけ離れた値を解の候補として提示することがわかる。

これは、二分法がもともと限られた範囲を二分していくためそこまで誤差が大きくなく安定して解に収束していくのに対し、ニュートン法では関数の形に依存する。今回のグラフでは収束の様子が図 2 のように観測できた。このグラフでは候補の点の接線が x 軸と交わった点が次の候補点になる様子がわかった。

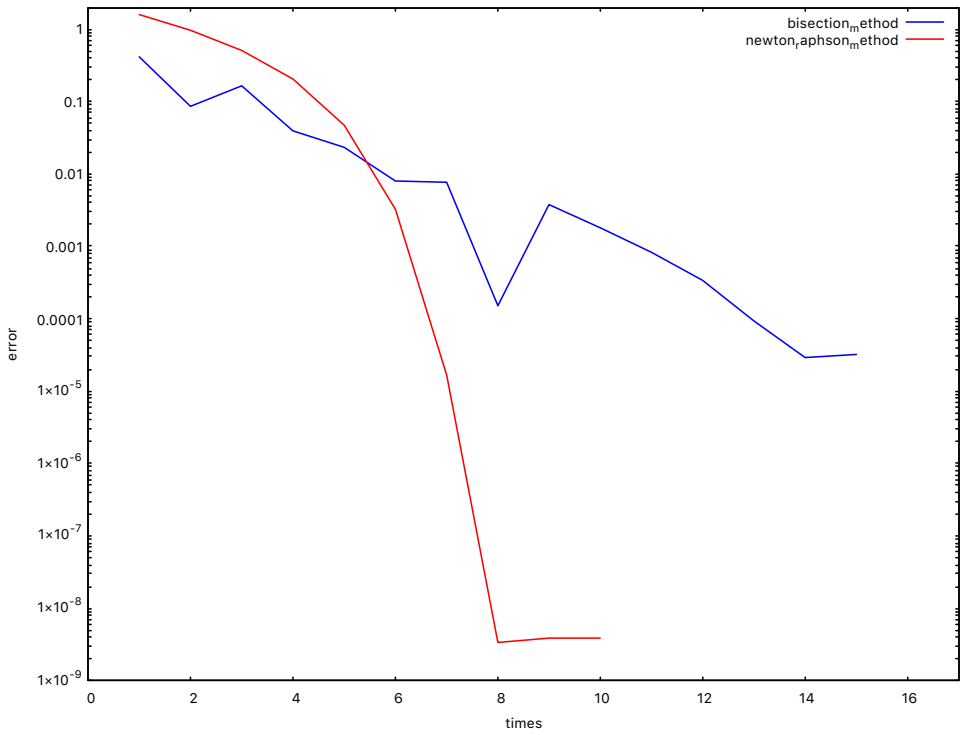

図 1　二分法・ニュートン法の収束の速さ

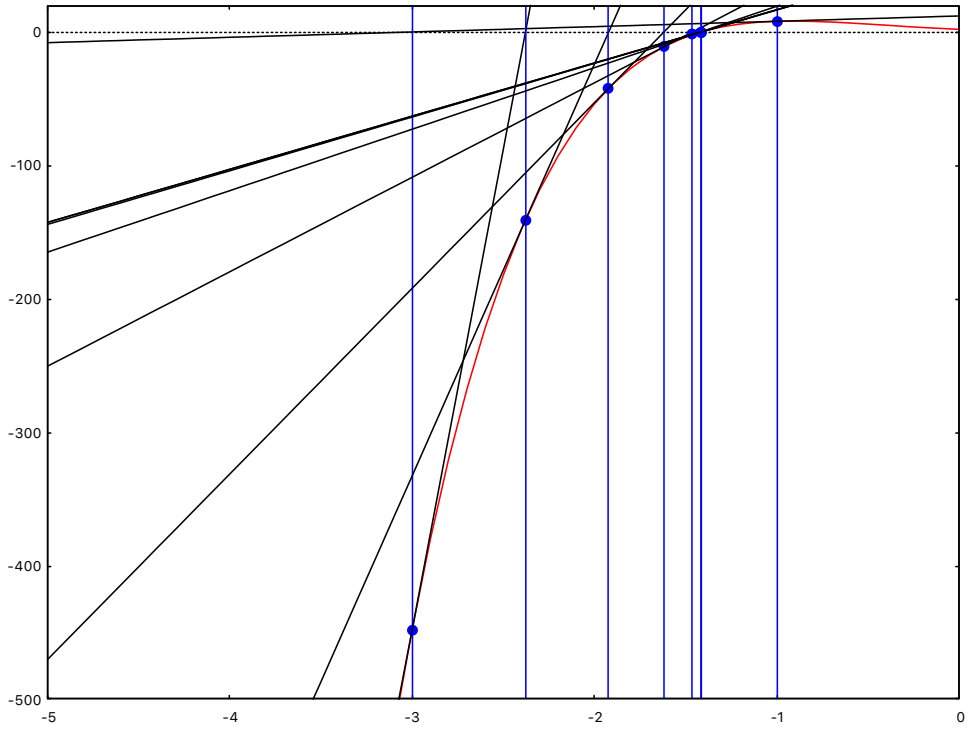

図 2　ニュートン法による収束の様子

2　課題 1.1.2

二分法の初期区間を [0, 1.2]、ニュートン法の初期近似解を-0.6 として課題 1.1.1 と同様のグラフを作成しようとしたところ、newton_raphson_method が、”x$\hat{}$(k+1) is not a number: last value is 0.9964477602428009.” というエラーを吐いて以上終了した。これは 0 除算を行う等の演算により、値が正常な値にならない時に Rust では値が NaN になるため、その時に吐き出すように設定したエラーメッセージである。ここで、エラー時にエラーメッセージを吐くのではなく、バッファの data を返すようにして値をプロットしたグラフが図 3 になる。

これは、ニュートン法の暗黙の仮定を破ったことによるエラーだと考えた。求めるべき解の真値は-1 だが、これは $f(x)$ の重解になっている。したがって解の近傍では $f(x)$、$f'(x)$ の双方が共にゼロに近づき、今回は $f'(x)$ の方が 0 に収束するのが速く、値が不正なものになってしまったのだと考えた。

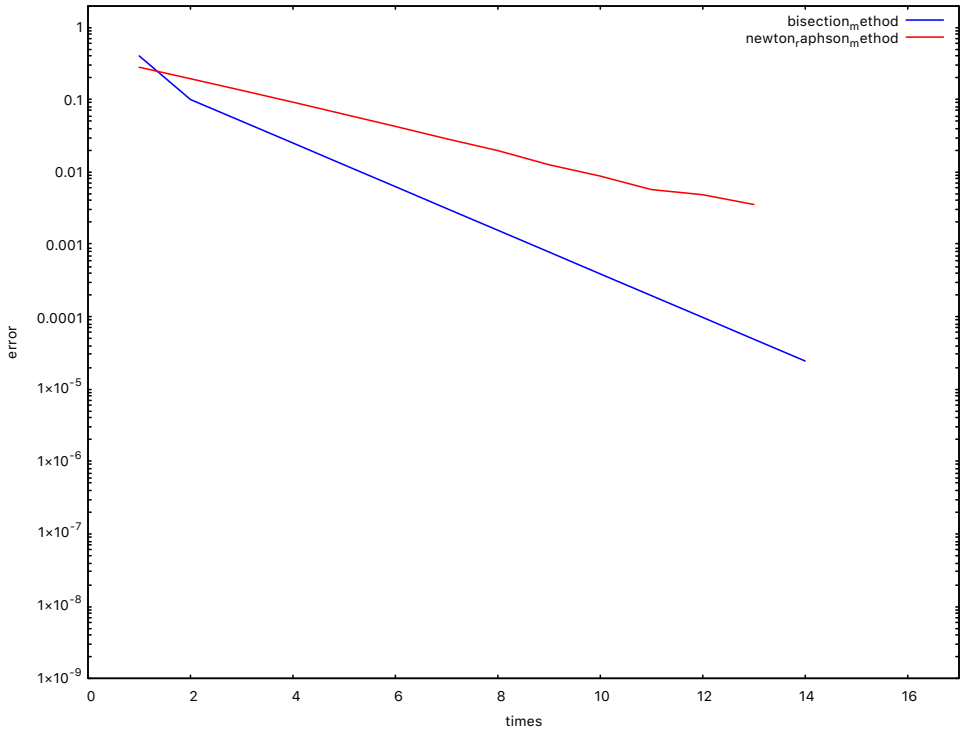

図 3　エラー終了するまでの値

電気電子計算工学及演習

1026-30-8137
多田 拓生

説明日
2020/10/*

課題 1.2

3　課題 1.2.1

与えたれたアルゴリズムを実行するプログラムを作成した。コードをソースコード 3 に示す。ソースコード 3 中では、自作行列演算ライブラリである matrix を使用している。このコードは 3,000 行を越えるためここに示すことはできないが、ソースコード 3 中で使用している Matrix の定義、append_line、内積計算、スカラー倍、ベクタとみなして L2 ノルムを求める norm2 をソースコード 4 に示し説明する。

ソースコード 3 では、まず行列 A を生成し、ループさせる関数を定義し、初期値からベクトル x を行列として生成し、30 回ループさせている。関数内では、まずベクトル y を行列演算 A*x をして行列として計算し、L2 ノルムを求め、繰り返し回数と求めたノルムをバッファに保存し、次の x の値を生成している。

ソースコード 4 では行列に関するコードを示した。

まず Matrix が行列の定義である。次に、append_line 関数が入れ子になっているベクタを元に行列を生成する関数である。MulSelf に示してあるのが行列の内積を計算する関数である。内積の定義通りに各行各列の要素を計算している。演算子のオーバーロードにより行列*行列をするとこの関数が呼ばれる。MulOutput=T に示すのがスカラー倍をする関数で、各要素にスカラー倍をしている。これも演算子オーバーロードにより行列*スカラで呼ばれる。Div は各要素に除算を行う。行列/スカラにより呼ばれる。norm2 関数は各要素を 2 乗し、全て足し合わせた後ルートを取っている。これにより L2 ノルムが計算できる。

さて、以上のコードを用いて収束を確かめたグラフを図 4 に示す。概ね 3.7 に収束していることがわかる。最終的な収束値は 3.682506 であった。

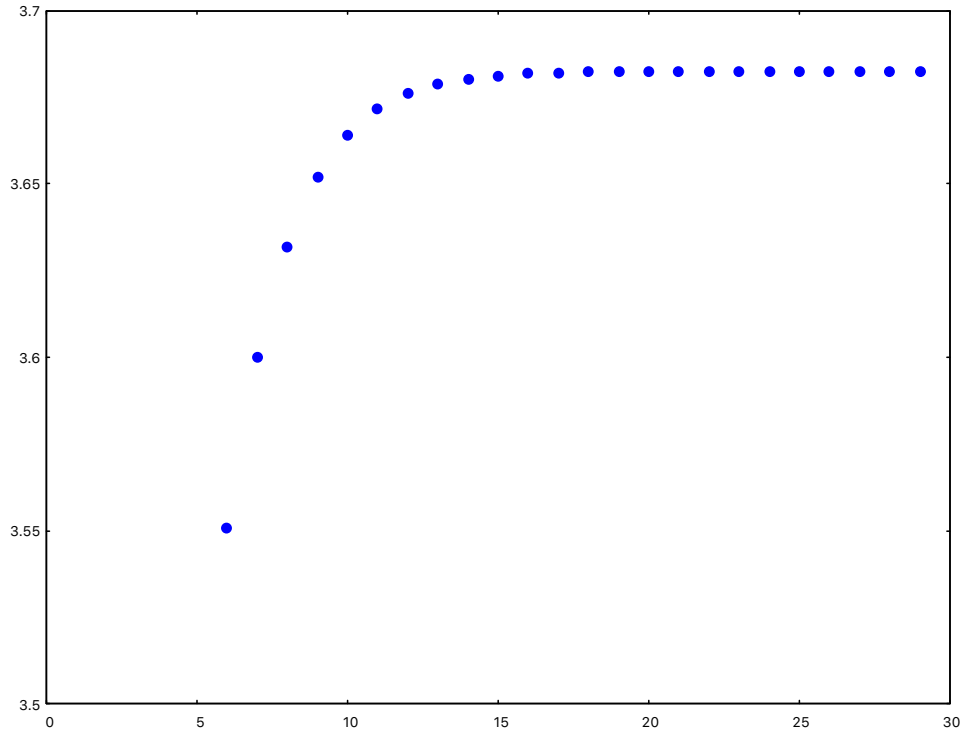

図 4　試行 30 回での収束

ソースコード 3　main.rs

```
fn kadai123(init: f32, times: usize) -> Vec<(f64, f64)> {
    let a = Matrix::append_line(vec![
        vec![2.0, -1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
        vec![-1.0, 2.0, -1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
        vec![0.0, -1.0, 2.0, -1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
        vec![0.0, 0.0, -1.0, 2.0, -1.0, 0.0, 0.0, 0.0, 0.0, 0.0],
        vec![0.0, 0.0, 0.0, -1.0, 2.0, -1.0, 0.0, 0.0, 0.0, 0.0],
        vec![0.0, 0.0, 0.0, 0.0, -1.0, 2.0, -1.0, 0.0, 0.0, 0.0],
        vec![0.0, 0.0, 0.0, 0.0, 0.0, -1.0, 2.0, -1.0, 0.0, 0.0],
        vec![0.0, 0.0, 0.0, 0.0, 0.0, 0.0, -1.0, 2.0, -1.0, 0.0],
        vec![0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, -1.0, 2.0, -1.0],
        vec![0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, -1.0, 2.0],
    ]);

    let mut data = Vec::new();
```

```
    let mut f = |x: Matrix<f32>, i: usize| -> Matrix<f32> {
        let y = &a * &x;
        let y_norm = y.norm2();
        println!("M: {}, y_norm: {}", i, y_norm);
        data.push((i as f64, y_norm as f64));
        &y / y_norm
    };

    let mut x = Matrix::new(10, 1);
    x += init;

    for i in 0..times {
        x = f(x, i);
    }

    data
}

fn main() {
    let s0 =
        Plot::new(kadai123(1.0, 30))
            .point_style(
                PointStyle::new().marker(PointMarker::Circle)
            );

    let v0 = ContinuousView::new()
        .add(s0)
        .x_range(0., 30.)
        .y_range(3.5, 4.)
        .x_label("times")
        .y_label("value");

    println!(
```

```
        "{}",
        Page::single(&v0).dimensions(80, 10).to_text().unwrap()
    );
}
```

ソースコード 4 matrix_dash.rs

```
#[derive(Clone, Debug, PartialEq, PartialOrd)]
pub struct Matrix<T> {
    n: usize,       // line             [* * * * *]
    m: usize,       // column           [* * * * *] -> n = 3, m = 5
    array: Vec<T>, //                  [* * * * *]
}

impl<T> Matrix<T>
where
    T: Clone,
{
    pub fn append_line(vec: Vec<Vec<T>>) -> Self {
        let n = vec.len();
        let m = vec[0].len();
        if !vec.iter().all(|e| e.len() == m) {
            panic!(
                "`Matrix::append_line` needs appropriatly sized Vec<Vec<T>>."
            );
        }
        Matrix {
            n,
            m,
            array: vec.concat(),
        }
    }
}

impl<T> Mul<Self> for &Matrix<T>
where
```

```
    T: Mul<Output = T> + Add<Output = T> + Clone + Zero,
{
    type Output = Matrix<T>;
    fn mul(self, rhs: Self) -> Self::Output {
        // TODO: use Strassen algorithm
        if !(self.m == rhs.n) {
            panic!(
                "`Matrix::mul` needs n * m Matrix<T> and m * k Matrix<T>."
            )
        }
        Matrix {
            n: self.n,
            m: rhs.m,
            array: {
                let mut v = Vec::<T>::new();
                for i in 0..self.n {
                    for j in 0..rhs.m {
                        let mut sum = T::zero();
                        for k in 0..self.m {
                            sum = sum
                                + self.array[i * self.m + k].clone()
                                    * rhs.array[j + k * rhs.m].clone()
                        }
                        v.push(sum)
                    }
                }
                v
            },
        }
    }
}

impl<T> Mul<T> for &Matrix<T>
where
```

```
    T: Mul<Output = T> + Clone,
{
    type Output = Matrix<T>;
    fn mul(self, rhs: T) -> Self::Output {
        Matrix {
            n: self.n,
            m: self.m,
            array: {
                let mut v = Vec::new();
                for i in 0..self.n * self.m {
                    v.push(self.array[i].clone() * rhs.clone())
                }
                v
            },
        }
    }
}

impl<T> Div<T> for &Matrix<T>
where
    T: Div<Output = T> + Clone,
{
    type Output = Matrix<T>;
    fn div(self, rhs: T) -> Self::Output {
        Matrix {
            n: self.n,
            m: self.m,
            array: {
                let mut v = Vec::new();
                for i in 0..self.n * self.m {
                    v.push(self.array[i].clone() / rhs.clone())
                }
                v
            },
```

```
        }
    }
}

impl<T> Matrix<T>
where
    T: Zero
        + Clone
        + ToPrimitive
        + One
        + Sub<Output = T>
        + Mul<Output = T>
        + Add<Output = T>
        + Div<Output = T>,
{
    pub fn norm2<F>(&self) -> F
    where
        F: Float + Zero + FromPrimitive + Add<Output = F>,
    {
        let mut size = F::zero();
        for i in 0..self.n * self.m {
            size = size.clone()
                + F::from(self.array[i].clone())
                    .unwrap()
                    .powf(F::from_f32(2.0).unwrap())
        }
        size.sqrt()
    }
}
```

4　課題 1.2.2

べき乗法はある行列 A の固有値が互いにすべて異なるときを考える。固有ベクトルが一次独立なので各固有ベクトルを基底として x を表現できるので、繰り返し Ax を求めることで最大の固有値を持つ固有ベクトルの影響が大きくなりその係数のみが残り、固有値が求まるという方法である。

この時、各ベクトルの係数は最大係数の固有値を λ_0、その他の固有値を λ_i としたときに、(λ_i/λ_0) の大きさに依存しながら収束する。そのため最大の大きさを持つ固有値と二番目の大きさを持つ固有値が近い値を持つときは、収束に時間がかかると考えられる。したがって、求まった約 3.7 という値が絶対値最大固有値に近いと考えて良いかは、さらに試行回数を増やさなければ断言できないと考えた。

5　課題 1.2.3

反復回数を 1000 回、初期値をそれぞれ 0.1、1.0、3.0、4.0、300.0 にして同様にプロットしたグラフを図 5 に示す。

図より、初期値に依らず、試行回数 300 回程度で近似解が 3.9 の近くまで跳ね上がることがわかる。したがって課題 1.2.1 で求めた値は絶対値最大の固有値ではなかったことがわかった。なお、(https://keisan.casio.jp/exec/system/1505174268) を用いて固有値を求めると絶対値最大の固有値は 3.9189 と求まるので、試行回数 1000 回で得られた値 (3.918986) が求めたい固有値の値に近いと考えて良いと考えた。

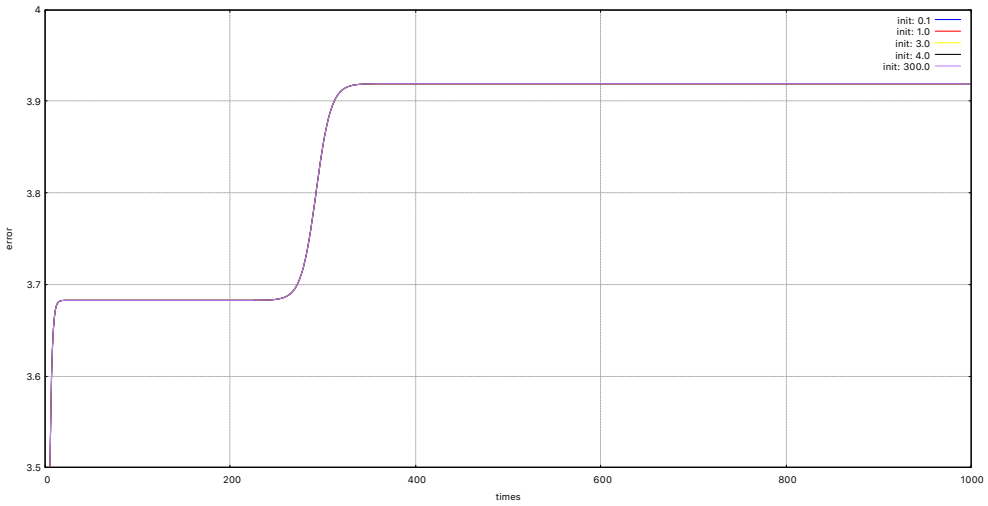

図 5　初期値、繰り返し回数を変えたべき乗法

電気電子計算工学及演習

1026-30-8137
多田 拓生

説明日
2019/*/*

課題 1.3

n 元連立非線形方程式の求解をニュートン法によって行うプログラムを作成した。そのコードをソースコード 5 に示す。

このコードでも matrix ライブラリを使用している。説明が必要な関数についてはソースコード 6 にコードを示し説明する。

まず、jacobian_newton_raphson_method は、関数のベクトル vec_f と初期近似解ベクトル vec_init を受け取る。真値との誤差を知りたければ vec_expected に真値を、値が欲しければ 0 ベクトルを与えれば良い。

関数の中では、まず dif_jacobi に関数のベクトルを与え、ヤコビアンを計算する。dif_jacobi は、その中で partial_dirivative 関数を用いて偏微分する。この関数は、割線法を応用して偏微分導関数を求めている。これにより、n 次元の関数のベクトルさえ与えればプログラムでヤコビアンを求められるように工夫した。そのあと、各ベクトルを演算しやすいように Matrix に変換し、またデータを保存するように data という変数をバッファとして宣言している。そしてそれらを再帰的な関数 jacobian_newton_method に与えている。

jacobian_newton_method では、関数のベクトル、ヤコビアン (値適用前)、解の候補、閾値、反復回数、最大反復回数、バッファを受け取る。この関数では、まず解の候補をヤコビアンに適用するために n^2 にしている。そして、それを matrix ライブラリの applicate 関数を用いてヤコビアンに適用し、その行列と関数のベクトルに解候補を適用したベクトルを使い、matrix の solve_eqn 関数を用いて修正量 Δx を求めている。applicate 関数と solve_eqn 関数はあとで説明するが、解を求めるのに LU 分解を行なっている。

そして、その修正量の絶対値が閾値よりも少なければ試行回数と解候補を記録し続けている data を返し、条件を満たさなければ再帰的に以上の処理を繰り返すようにしている。

ソースコード 5　newton_raphson_method.rs

```
#![allow(dead_code)]
```

```
pub use crate::matrix::*;
pub use std::rc::Rc;
pub use std::result::Result;

pub fn jacobian_newton_raphson_method(
    vec_f: Vec<Rc<dyn Fn(Vec<f64>) -> f64>>,
    vec_init: Vec<f64>,
    vec_expected: Vec<f64>
) -> Result<Vec<(f64, Vec<f64>)>, String> {
    let n = vec_f.len();
    if n != vec_init.len() {
        panic!(
            "`jacobian_newton_raphson_method` needs
                vec_f and vec_init the same size."
        );
    }
    let threshold = 0.1e-10;
    let jacobian = dif_jacobi(vec_f.clone());
    let init = Matrix::append(n, 1, vec_init);
    let f_n: Matrix<Rc<dyn Fn(Vec<f64>) -> f64>> =
        Matrix::append(n, 1, vec_f);
    let data: Vec<(f64, Vec<f64>)> = Vec::new();
    let mtrx_expected =
        Matrix::append(vec_expected.len(), 1, vec_expected);
    jacobian_newton_method(f_n, jacobian, init,
        threshold, 1, 1_000_000, vec_expected, data)
}

fn dif_jacobi(
    vec_f: Vec<Rc<dyn Fn(Vec<f64>) -> f64>>
) -> Matrix<Rc<dyn Fn(Vec<f64>) -> f64>> {
    Matrix::append_line({
        let mut v = Vec::new();
        for i in 0..vec_f.len() {
```

```
            let mut u = Vec::new();
            for j in 0..vec_f.len() {
                unsafe {
                    u.push(partial_derivative(vec_f.index(i).clone(), j))
                }
            }
            v.push(u);
        }
        v
    })
}

unsafe fn partial_derivative(
    f: Rc<dyn Fn(Vec<f64>) -> f64>,
    i: usize,
) -> Rc<dyn Fn(Vec<f64>) -> f64> {
    let dx = 0.1e-10;
    let f_der = move |v: Vec<f64>| -> f64 {
        let mut v_dx = v.clone();
        v_dx[i] += dx;
        (f(v_dx) - f(v)) / dx
    };
    Rc::new(f_der)
}

fn jacobian_newton_method(
    vec_f: Matrix<Rc<dyn Fn(Vec<f64>) -> f64>>,
    jacobian: Matrix<Rc<dyn Fn(Vec<f64>) -> f64>>,
    v_guess: Matrix<f64>,
    threshold: f64,
    times: usize,
    limit: usize,
    mtrx_expected: Matrix<f64>,
    mut data: Vec<(f64, Vec<f64>)>
```

```
) -> Result<Vec<(f64, Vec<f64>)>, String> {
    let x_k_ = vec![
        vec![v_guess.to_vec().clone(); v_guess.to_vec().len()];
        v_guess.to_vec().len()
    ];
    let x_k = x_k_.concat();
    let jacobian_applicated = jacobian.applicate(&x_k);
    let f_x_k = vec_f.applicate(
        &vec![v_guess.to_vec().clone(); v_guess.to_vec().len()]
    );
    let v_next = Matrix::solve_eqn(&jacobian_applicated, &f_x_k);
    if limit == times + 1 {
        return Err(format!(
            "solution doesn't converge: last value is {:?}.",
            v_next.to_vec()
        ));
    }
    let dx: f64 = v_next.norm2();
    println!("dx : {}", dx);
    data.push((times as f64,
        (&(&v_guess - &v_next) - &mtrx_expected).to_vec()));
    if dx <= threshold {
        Ok(data)
    } else {
        println!("{}, {:?}", times, (&v_guess - &v_next).to_vec());
        jacobian_newton_method(
            vec_f,
            jacobian,
            &v_guess - &v_next,
            threshold,
            times + 1,
            limit,
            vec_expected,
            data
```

```
        )
    }
}

#[cfg(test)]
mod tests_newton_raphson_method {
    use crate::newton_raphson_method::newton_method;
    use crate::newton_raphson_method::*;

    #[test]
    fn test_newton_raohson_jacobi_newton_raphson() {
        let f1: Rc<dyn Fn(Vec<f64>) -> f64> =
            Rc::new(|x1: Vec<f64>| -> f64 {
                x1[0] * x1[0] + x1[1] * x1[1] - 2.0
            });
        let f2: Rc<dyn Fn(Vec<f64>) -> f64> =
            Rc::new(|x2: Vec<f64>| -> f64 {
                x2[0] - x2[1] * x2[1]
            });
        let mut vec_f: Vec<Rc<dyn Fn(Vec<f64>) -> f64>> =
            Vec::new();
        vec_f.push(f1.clone());
        vec_f.push(f2.clone());

        assert_eq!(
            jacobian_newton_raphson_method(vec_f,
                vec![2.0f64.sqrt(); 2],
                vec![0f64, 0f64]).unwrap().pop().unwrap().1,
            vec![1.0f64, 1.0f64]
        );
        let f3: Rc<dyn Fn(Vec<f64>) -> f64> =
            Rc::new(|x1: Vec<f64>| -> f64 {
                x1[0] * x1[0] + x1[1] - 5.0
            });
```

```
    let f4: Rc<dyn Fn(Vec<f64>) -> f64> =
        Rc::new(|x2: Vec<f64>| -> f64 {
            x2[0] - x2[1] * x2[1] - 1.0
        });

    let mut vec_f: Vec<Rc<dyn Fn(Vec<f64>) -> f64>> = Vec::new();
    vec_f.push(f3.clone());
    vec_f.push(f4.clone());

    assert_eq!(
        jacobian_newton_raphson_method(vec_f,
            vec![2.0f64.sqrt(); 2],
            vec![0f64, 0f64]).unwrap().pop().unwrap().1,
        vec![2.0f64, 1.0f64]
    );
}

#[test]
#[should_panic(
    expected = "`jacobian_newton_raphson_method` needs
        vec_f and vec_init the same size."
)]
fn test_newton_raohson_jacobi_newton_raphson_panic() {
    let f1: Rc<dyn Fn(Vec<f64>) -> f64> =
        Rc::new(|x1: Vec<f64>| -> f64 {
            x1[0] * x1[0] + x1[1] * x1[1] - 2.0
        });
    let f2: Rc<dyn Fn(Vec<f64>) -> f64> =
        Rc::new(|x2: Vec<f64>| -> f64 {
            x2[0] - x2[1] * x2[1]
        });
    let mut vec_f: Vec<Rc<dyn Fn(Vec<f64>) -> f64>> = Vec::new();
    vec_f.push(f1.clone());
    vec_f.push(f2.clone());
```

```
        let _ = jacobian_newton_raphson_method(vec_f, vec![2.0f64.sqrt()]);
    }
}
```

6 課題 1.3.1

7 課題 1.3.2

参考文献

[1] 森正武. 『数値解析 (第 2 版)』. 共立出版, 2018.
[2] 藤野和建伊理正夫. 『数値計算の常識』. 共立出版, 2011.