ДУАЛЬНЫЕ ЧИСЛА В ЧИСЛЕННОМ ИНТЕГРИРОВАНИИ

В работе [1] представлен метод численного интегрирования на базе усеченных гипер-дуальных чисел и автоматического дифференцирования. Для его восприятия необходимо иметь определенный уровнь знаний из разных прикладных и фундаментальных областей. Более того практическое использование автоматического дифференцирования началось с начала двухтысячных годов [2], гипер-дуальные числа впервые были описаны [3] во втором десятилетии этого века, а усеченные гипер-дуальные числа и того позже. Поэтому цель этой работе дать упрощенный подход на основе [1], но с использованием классических дуальных чисел [4], которые применяются в разных приложениях достаточно давно.

1. Построение базового соотношения

Обычно численное интегрирование используется, когда подынтегральная функция не задана аналитически либо её первообразная не выражается через аналитические функции. Большинство методов численного интегрирования состоит в следующем. Заданный отрезок интегрирования $[a,\ b]$ делится опорными точками $x_0,\ x_1,\ x_2,\ \ldots,\ x_n,$ на n равных элементарных интервалов длиной $\Delta x = 2\mathrm{h} = (b - a)/n$. На каждом i-ом интервале $[x_i\ ,\ x_{i+1}]$ исходная подынтегральная функция $f(x)$ обычно аппроксимируется многочленами той либо иной степени. Рассмотрим i-й интервал как составной интервал, состоящий из двух подинтервалов $[x_i,\ \bar{x}_i]$ и $[\bar{x}_i,\ x_{i+1}]$ с $\bar{x}_i = (x_{i+1} - x_i)/2$. Применим к каждому из подинтервалов формулу трапеций [5] и выполним элементарные алгебраические преобразования:

$$\begin{aligned} S_i &= \frac{h}{2}(f_{ic} + f_{i1}) + \frac{h}{2}(f_{ic} + f_{i2}) = h\left(f_{ic} + \frac{f_{i1} + f_{i2}}{2}\right) = \frac{2h}{3}\left[\frac{3}{2}f_{ic} + \frac{3}{4}(f_{i1} + f_{i2})\right] \\ &= \frac{2h}{3}\left[f_{ic} + f_{i1} + f_{i2} + \frac{1}{4}(2f_{ic} - f_{i1} - f_{i2})\right] \\ &= \frac{2h}{3}\left\{f_{ic} + f_{i1} + f_{i2} + \frac{h}{4}\left[\frac{(f_c - f_1) - (f_2 - f_c)}{h}\right]\right\} \\ &= \frac{2h}{3}\left[f_{ic} + f_{i1} + f_{i2} + \frac{h}{4}\left(\tilde{f}'_{i1} - \tilde{f}'_{i2}\right)\right], \end{aligned} \tag{1.1}$$

где: $f_{i1} = f(x_i),\ f_{ic} = f(\bar{x}_i),\ f_{i2} = f(x_{i+1})$ и $\tilde{f}'_{1i} = \frac{f_{ic} - f_{i1}}{h},\ \tilde{f}'_{i2} = \frac{f_{i2} - f_{ic}}{h}$ – значения конечно-разносных производных «вперед» и «назад» соответственно. Заменяя их на точные значения производных $f'_{1i} = f'(x_i)$ и $f'_{2i} = f'(x_{i+1})$ получим

$$S_i = \frac{2h}{3}\left[f_{ic} + f_{i1} + f_{i2} + \frac{h}{4}(f'_{i1} - f'_{i2})\right] \tag{1.2}$$

Тогда велечина интеграла от $f(x)$ на интервале $[a, b]$ равна

$$J = \int_a^b f(x)dx \approx \frac{2h}{3}\sum_{i=1}^{n} S_i \tag{1.3}$$

Формулу для S_i можно получить более строгим путем, а именно разложить функцию $f(x)$ в ряд Тейлора в окрестности $\bar{x}_i$ с удержанием первых трех членов, затем ввести два корректирующих многочлена второй степени для удовлетворения условий на границах интервала (значений функции и ее первых производных). В этом случае наглядно будет видно, что аппроксимирующая функция состоит из двух парабол, сопряженных в точке $\bar{x}_i$, проходящих через f_{i1} и f_{i2}, а также удовлетворяющая значениям производных в точках x_i и x_{i+1}. При этом $\bar{x}_i$ является точкой разрыва первого рода для производной аппроксимирующей функции т. е. значения пределов производной слева и справа не равны.

Из полученного соотношения для S_i следует, что помимо значений исходной функции в трех точках интервала необходимо иметь значения производных в концевых точках этого интервала.

Если же аппроксимирующая функция состоит из двух сопряженных гипербол в точке $\bar{x}_i$, проходящих через f_{i1} и f_{i2}, а также удовлетворяющая значениям производных в точках x_i, $\bar{x}_i$ и x_{i+1}, то получим

$$S_i = h[f_{ic} + (f_{i1} + f_{i2})/2 + h(f'_{i1} - f'_{i2})/12] \tag{1.4}$$

Нетрудно видеть, что первые два члена формулы (1.4) соответствуют формуле трапеций, а последний является корректирующим членом, учитывающим значения производных в концевых точках интервала.

Для точного вычисления производных, входящих с (1.2) и (1.4), можно воспользоваться классическими дуальными числами.

2. Дуальные числа

Классические дуальные числа [4] это – гиперкомплексные параболические числа вида $X = x + x_1\boldsymbol{\varepsilon}$, где x и x_1 – вещественные числа, а $\boldsymbol{\varepsilon}$ – абстрактный элемент, квадрат которого равен нулю. Число x называется главной (*Re* - действительной) частью дуального числа, а x_1 – мнимой (*Im* - инфинитезимальной) его частью. Абстрактный элемент $\boldsymbol{\varepsilon}$ является *основным базисом* мнимой части дуального числа. Причем $\boldsymbol{\varepsilon} \neq 0$ и $\boldsymbol{\varepsilon}^k = 0$ при $k > 1$.

Для дуальных чисел определены операции сложения, умножения и деления:

$$\begin{gathered} X + Y = (x + x_1\boldsymbol{\varepsilon}) + (y + y_1\boldsymbol{\varepsilon}) = (x + y) + (x_1 + y_1)\boldsymbol{\varepsilon}, \\ X{\cdot}Y = (x + x_1\boldsymbol{\varepsilon}){\cdot}(y + y_1\boldsymbol{\varepsilon}) = (x{\cdot}y) + (x{\cdot}y_1 + y{\cdot}x_1)\boldsymbol{\varepsilon}, \\ \frac{X}{Y} = \frac{(x + x_1\boldsymbol{\varepsilon})}{(y + y_1\boldsymbol{\varepsilon})} = \left(\frac{x}{y}\right) + \frac{(y{\cdot}x_1 - x{\cdot}y_1)}{y^2}\boldsymbol{\varepsilon} \quad \text{при} \quad y \neq 0 \end{gathered} \tag{2.1}$$

Разложение аналитической функции дуального аргумента (с учетом свойств $\boldsymbol{\varepsilon}$) в ряд Тейлора дает

$$F(x + x_1\boldsymbol{\varepsilon}) = f(x) + x_1 f'(x), \tag{2.2}$$

где: $f(x) = F(x + 0\boldsymbol{\varepsilon})$, $f'(x) = \partial f(x)/\partial x$.

Как видно, значение функции дуального аргумента $F(x + x_1\boldsymbol{\varepsilon})$ определяется через значения этой функции $f(x) = F(x + 0\boldsymbol{\varepsilon})$ и ее производной $f'(x)$ от главной части дуального числа x и величины мнимой части x_1. При $x_1 = 1$ получим $F(x + \boldsymbol{\varepsilon}) = f(x) + f'(x)$ – представление функции дуального аргумента с мнимой частью, содержащей только значение производной.

Таким образом, производя вычисления не над вещественными, а над дуальными числами, можно автоматически получать точные значения самой функции и ее производной в заданной точке. Особенно удобно рассматривать с этих позиций сложные композиции функций.

Исходя из изложенного, соотношение (1.2) для S_i в терминах дуальных функций примет вид

$$S_i = \frac{2h}{3}\left[F_{ic}.Re + F_{i1}.Re + F_{i2}.Re + \frac{h}{4}(F_{i1}.Im - F_{i2}.Im)\right], \tag{2.3}$$

а для (1.4)

$$S_i = h[F_{ic}.Re + (F_{i1}.Re + F_{i2}.Re)/2 + h(F_{i1}.Im - F_{i2}.Im)/12], \tag{2.4}$$

где: $F_{ic} = F(\overline{\boldsymbol{x}_i} + \boldsymbol{\varepsilon})$, $F_{i1} = F(x_i + \boldsymbol{\varepsilon})$, $F_{i2} = F(x_{i+1} + \boldsymbol{\varepsilon})$.

3. Автоматическое дифференцирование

С развитием вычислительной техники возникло новое направление в вопросах численного дифференцирования – автоматическое дифференцирование [2, 3], связанное с точным (точностью представления чисел в компьютерной системе) вычислением производных сложных математических функций. Автоматическое дифференцирование (АД) позволяет избежать дублирование функциональности программного кода (изменение кода функции не требует изменения кода ее производной). Для компьютерной реализации АД необходимо создать новый тип данных (дуальное число), перезагрузить базовые математические функции и операции над ними.

4. Компьютерная реализация

На языке SWIFT операционной системы macOS был описан новый тип данных DualNumber (см. Приложение 1), осуществляющий перезагрузку базовых математических функций и операций над ними. В Приложении 2 представлены две процедуры Integral_12(…) и Integral_14(…) для вычисления значений интеграла от заданной функции по формулам (1.2) и (1.4) соответственно. Обе процедуры осуществляют вычисление интеграла с заданной точностью путем удвоения количества интервалов интегрирования. В Приложении 3 даны некоторые примеры обращения к этим процедурам.

5. Численный эксперименты

Для проведения численного анализа предлагаемого подхода были рассмотрены самые разнообразные исходные функции с разными пределами интегрирования и количеством расчетных точек. Анализ полученных данных дает возможность заключить, что в общем случае (при прочих равных условиях) наибольшую точность обеспечивает метод предложенный в [1] т. к. учитывает первые и вторые производные. Для градких функций метод по формуле (1.4) является достойным конкурентом предыдущему метода. Для разрывных функций методы по (1.2) и [1] являются лидирующими. В качестве наглядного примера в табл.1 приведены результаты расчета для двух функций $f_1(x) = sin(x)$ и $f_2(x) = sin(x) + \theta(x - \pi/2) \cdot sin\,(2x - \pi)$ где $\theta(\dots)$ – единичная функция Хэвисайда [6]. Вычисления проводились при $[a, b]$ =$[0, \pi]$ и $h = \pi/2$.

$f(x)$	Симпсон [5]		По (1.2)		По (1.4)		По [1]		Точное значение
	J	δ%	J	δ%	J	δ%	J	δ%	
$f_1(x)$	2.094…	4.7%	1.869…	6.5%	1.982…	0.9%	2.019…	0.95%	2.0
$f_2(x)$	2.094…	30.3%	2.692…	10.27%	2.393…	20.2%	3.252…	8.4%	3.0

Таблица 1. Результаты численного интегрирования функций
$f_1(x) = sin(x)$ и $f_2(x) = sin(x) + \theta(x - \pi/2) \cdot sin\,(2x - \pi)$

Из приведенных результатов видно, что метод Симпсона для гладких функций может конкурировать только с методом (1.2). В общем случае целесообразно использование методов описанных здесь или в [1].

6. Заключение

Как правило, большинство балансовых уравнений представлены системой интегральных и дифференциальных уравнений, в результате решения которых могут быть получены зависимости, характеризующие протекание процесса. Получение численного решения таких уравнений связано с использованием процедур численного дифференциарования и интегрирования. В этом случае весьма целесообразно примение методов, дающих точные значения самих функций и их производных. Методы численного интегрирования, рассмотренные в этой статье и в [1], являются достойными кандидатами в решение таких задач.

Приложение 1.

Код реализующий тип данных `DualNumber` на языке Swift 5 (macOS 11.2).

```
public struct DualNumber{
    var re,im: Double;
```

```
    //  Initializers:
    init() {self.re = 0; self.im = 0}
    init(re:Double) {self.re = re; self.im = 0}
    init(re:Double,im:Double){self.re = re; self.im = im}
}

extension DualNumber {
    // OPERATORS (overloading):
    static prefix func - (A:DualNumber)->DualNumber{return DualNumber(re: -A.re,
                          im: -A.im)}
    static prefix func + (A:DualNumber)->DualNumber{return A}
    static func + (A:DualNumber, B:DualNumber)->DualNumber{
        return DualNumber(re: A.re + B.re, im: A.im + B.im);
    }
    static func - (A:DualNumber, B:DualNumber)->DualNumber{
        return DualNumber(re: A.re - B.re, im: A.im - B.im);
    }
    static func -= (lhs: inout DualNumber, rhs: DualNumber) {lhs = lhs - rhs}
    static func *(A: DualNumber, B: DualNumber) -> DualNumber {
        return  DualNumber( re: A.re * B.re, im: A.re * B.im + A.im * B.re);
    }
    static func *(A: DualNumber, B: Double) -> DualNumber {
        return  DualNumber(re:  A.re * B, im:  A.im * B);
    }
    static func *(A: Double, B: DualNumber) -> DualNumber {return B * A}
    static func / (A: DualNumber, B: DualNumber) -> DualNumber {
        let b = B.re, b2 = b * b;
        return DualNumber(re:  A.re / b, im: (A.im / b - A.re * B.im/b2));
    }
    static func / (A: DualNumber, B: Double) -> DualNumber {
        return DualNumber(re: A.re / B, im: (A.im)/B);
    }
    // FUNCTIONS:
    static func inverse(A:DualNumber)->DualNumber{
        let a2 = A.re * A.re;
        return  DualNumber(re: 1.0/A.re, im: -A.im/a2);
    }
    static func pow(x:Double, n:Double)->DualNumber{
        return DualNumber(re:  Darwin.pow(x, n), im:  n * Darwin.pow(x, n - 1));
    }
    static func pow(X:DualNumber, n:Double)->DualNumber{return  self.HD(X: X,
                   f: self.pow, n: n)}
    static func log(x:Double)->DualNumber{
        return DualNumber(re:  Darwin.log(x), im: 1.0/x);
    }
    static func log(X:DualNumber)->DualNumber{return  self.HD(X: X, f: self.log)}
    static func sqrt(x:Double)->DualNumber{return self.pow(x: x, n: 0.5)}
    static func sqrt(X:DualNumber)->DualNumber{return  self.HD(X: X, f: self.pow,
                    n: 0.5)}
    static func exp(x:Double)->DualNumber{
        let ex = Darwin.exp(x);
        return DualNumber(re: ex, im: ex);
    }
    static func exp(X:DualNumber)->DualNumber{return  self.HD(X: X, f: self.exp)}
    static func sin(x:Double)->DualNumber{
        let s = Darwin.sin(x), c = Darwin.cos(x);
```

```
        return DualNumber(re:  s, im:  c);
    }
    static func sin(X:DualNumber)->DualNumber{return  self.HD(X: X, f: self.sin)}
    static func cos(x:Double)->DualNumber{
        let s = Darwin.sin(x), c = Darwin.cos(x)
        return DualNumber(re:  c, im: -s);
    }
    static func cos(X:DualNumber)->DualNumber{return  self.HD(X: X, f: self.cos)}
    static func tan(x:Double)->DualNumber{
        let t = Darwin.tan(x), t_1 = t*t + 1.0;
        return DualNumber(re:  t, im: t_1);
    }
    static func tan(X:DualNumber)->DualNumber{return  self.HD(X: X, f: self.tan)}
    static func asin(x:Double)->DualNumber{
        let t = Darwin.asin(x), t_1 = 1.0/Darwin.sqrt(1.0 - x*x);
        return DualNumber(re:  t, im: t_1);
    }
    static func asin(X:DualNumber)->DualNumber{return  self.HD(X: X, f: self.asin)}
    static func acos(x:Double)->DualNumber{
        let t = Darwin.acos(x), t_1 = -1.0/Darwin.sqrt( 1.0 - x*x);
        return DualNumber(re:  t, im: t_1);
    }
    static func acos(X:DualNumber)->DualNumber{return  self.HD(X: X, f: self.acos)}
    static func atan(x:Double)->DualNumber{
        let t = Darwin.atan(x), t1 = 1.0/(1.0 + x*x);
        return DualNumber( re:  t, im: t1);
    }
    static func atan(X:DualNumber)->DualNumber{return  self.HD(X: X, f: self.atan)}
    // PRIVATE MEMBERS:
    private static func HD(X:DualNumber, f:(Double)->DualNumber)->DualNumber{
      let F = f(X.re);
      return DualNumber.ZX(F: F, X: X);
    }
    private static func HD(X:DualNumber, f:(Double, Double)->DualNumber,
                           n:Double)->DualNumber{
      let F = f(X.re, n);
      return DualNumber.ZX(F: F, X: X);
    }
    private static func ZX(F:DualNumber,X:DualNumber)->DualNumber{
      return DualNumber(re:   F.re, im:  X.im * F.im);
    }
}
```

Приложении 2.

Код процедур `integral_12(…)` и `integral_14(…)` на языке Swift 5 (macOS 11.2).

```
FUNCTION integral_12(...) - numerical integration in finite limits of integration
//     INPUT:  xo - start point, xn - end point, n - starting number of intervals,
//             F - function pointer, δ - accuracy, maxN - max number of interactions
//     OUTPUT:  (J, N) - tuple, where J - integral value; N - number of interactions
func integral_12(xo:Double, xn: Double,  n:Int = 10, F:(DualNumber) -> DualNumber,
                 δ:Double = 1E-6, maxN:Int = 10)->(J:Double, N: Int) {
    var n = n, i = 0, Δ = (xn - xo)/Double(n), h, r:Double;
```

```
    var Jo = 0.0, J = 0.0, S = 0.0, N = 0, E = Double.greatestFiniteMagnitude;
    var Ji:[DualNumber] = [], F1:DualNumber, Fc:DualNumber, F2:DualNumber;
    //--calculating F() for input n:
    for x in stride(from: xo, through: xn, by: Δ){Ji.append(F(DualNumber(re: x,
                                                    im: 1.0)))}
    while(E > δ){
        N += 1;
        if N > maxN {N -= 1; break}
        Δ = (xn - xo)/Double(n);
        h = Δ/2.0;  r = h*2.0/3.0;
        Jo = J; J = 0.0;
        for k in 0..<n {
           i  = 2*k + 1;
           F1 = Ji[i - 1];    F2 = Ji[i];
           Fc = F(DualNumber(re: xo + Double(i)*h, im: 1.0));
           S  = r*(Fc.re + F1.re + F2.re + h*(F1.im - F2.im)/4.0);
           J += S;   Ji.insert(Fc, at: i);
        }
        E = abs(J - Jo); n *= 2; //--doubling the number of intervals
    }
    return (J:J, N:N);
}

FUNCTION integral_14(...) - numerical integration in finite limits of integration
//    INPUT:  xo - start point, xn - end point, n - starting number of intervals,
//            F - function pointer, δ - accuracy, maxN - max number of interactions
//    OUTPUT:  (J, N) - tuple, where J - integral value; N - number of interactions
func integral_14(xo:Double, xn: Double,  n:Int = 10, F:(DualNumber) -> DualNumber,
                 δ:Double = 1E-6, maxN:Int = 10)->(J:Double, N: Int) {
    var n = n, i = 0, Δ = (xn - xo)/Double(n), h:Double;
    var Jo = 0.0, J = 0.0, S = 0.0, N = 0, E = Double.greatestFiniteMagnitude;
    var Ji:[DualNumber] = [], F1:DualNumber, Fc:DualNumber, F2:DualNumber;
    //--calculating F() for input n:
    for x in stride(from: xo, through: xn, by: Δ){Ji.append(F(DualNumber(re: x,
                                                    im: 1.0)))}
    while(E > δ){
        N += 1;
        if N > maxN {N -= 1; break}
        Δ = (xn - xo)/Double(n);
        h = Δ/2.0;
        Jo = J; J = 0.0;
        for k in 0..<n {
            i  = 2*k + 1;
            F1 = Ji[i - 1];    F2 = Ji[i];
            Fc = F(DualNumber(re: xo + Double(i)*h, im: 1.0));
            S  = h*(Fc.re + (F1.re + F2.re)/2.0) + h*h*(F1.im - F2.im)/12.0;
            J += S;   Ji.insert(Fc, at: i);
        }
        E = abs(J - Jo); n *= 2; //--doubling the number of intervals
    }
    return (J:J, N:N);
}
```

Приложении 3.

Код некоторых дуальных функций и их использование в обращении к `integral_12(…)` и `integral_14(…)` на языке Swift 5 (macOS 11.2).

```
func Sin(X:DualNumber)->DualNumber{return DualNumber.sin(X: X)}

func SinX(X:DualNumber)->DualNumber{
    var S = DualNumber.sin(X: X);
    if X.re > Double.pi/2.0 {
        S = S + DualNumber.sin(X: 2.0*X - DualNumber(re: Double.pi))}
    return S;
}

var J12 = integral_12(xo: 0.0, xn: Double.pi, n:1, F: Sin);
var J14 = integral_14(xo: 0.0, xn: Double.pi, n:1, F: SinX);
```

Литература

1. *Олифер В.И.* Численное интегрирование с использованием гипер-дуальных чисел. – URL: http://viosolutions.amerihomesrealty.com/pdf/численное_интегрирование_с_использованием_гипер-дуальных_чисел.pdf
2. *Naumann U*. The art of differentiating computer programs. Society for industrial and applied mathematics, Philadelphia, USA, 2012.
3. *Fike J.A., Alonso J.J.* The development of hyper-dual numbers for exact second derivative calculations. AIAA paper 2011-886, 49th AIAA Aerospace Sciences Meeting, January 4-7, 2011.
4. *Гордеев В. Н.* Кватернионы и бикватернионы с приложениями в геометрии и механике. Киев: Сталь, 2016. - 315 с.
5. *Двайт Г.Б.* Таблицы интегралов и другие математические формулы. - М.: "Наука", 1973. - 228 с.
6. *Кеч В., Теодореску П.* Введение в теорию обобщённых функций с приложениями в технике. - М.: Мир, 1978, - 520 с.

Абстракт

В данной публикации рассматривается метод численного интегрирования на основе разложения в ряд Тейлора и автоматического дифференцирования с использованием классических дуальных чисел. Представлена компьютерная реализация этого метода для языка SWIFT операционной системы macOS.

Ключевые слова: *дуальные числа, автоматическое дифференцирование, численное интегрирование, dual numbers, automatic differentiation, numerical integration*

March 20, 2021