Лекция 2

1. Решение скалярных уравнений и систем уравнений методом Ньютона

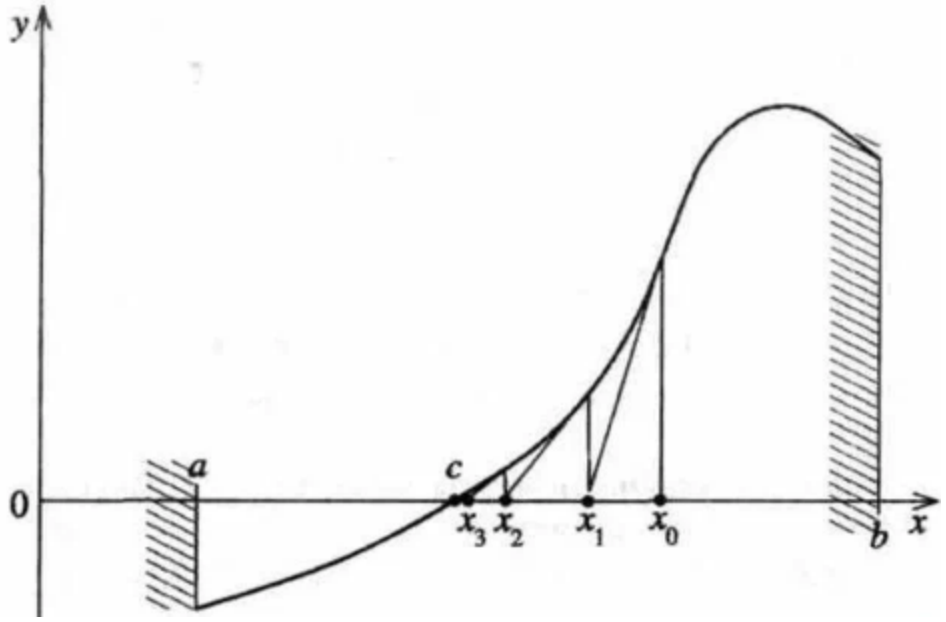
1.1. Вывод итерационной формулы

Пусть $f : \mathbb{R} \to \mathbb{R}$ - некоторая гладкая функция. Рассмотрим уравнение вида

$$f(x) = 0$$

Это уравнение мы будем решать итерационно, т.е. будем строить некоторую последовательность $x^{(0)}, x^{(1)}, x^{(2)}, \dots$, сходящуюся к искомому корню этого уравнения.

На следующем рисунке приведена графическая иллюстрация решения вещественного скалярного уравнения методом Ньютона, называемого так же методом касательных. Процедура состоит в том, сначала выбирается подходящее начальное приближение x_0, а затем, на каждой новой итерации в точке очередного приближения строится касательная к графику, и ищется пересечение этой касательной с остью абсцисс (осью x). Эта точка пересечения принимается за очередное приближение к искомому корню.



Из рисунка видно, что если начальное приближение x_0 выбрано удачно, то процесс сходится очень быстро: уже на 4-ой итерации очередная касательная практически сливается с графиком функции. Но если бы начальное приближение выбрать чуть ближе к точке b, то итерационная процедура уже не будет сходиться к искомому коню уравнения.

Возможно, она бы сходилась к другому корню, или не сходилась бы вовсе - это зависит от вида графика функции за пределами отрезка $[a, b]$.

Рассмотрим теперь вывод рекуррентной формулы для $x^{(k)}$, лежащей в основе метода Ньютона. Если ограничиться только случаем вещественного скалярного уравнения, то эту формулу легко было бы получить непосредственно из геометрических соображений (основываясь на приведенном выше рисунке, например). Однако мы получим эту формулу аналитически. Потом это позволит нам сделать значительные обобщения полученной аналитическим способом рекуррентной формулы.

Пусть $x^{(k)}$ - это некоторая точка в окрестности этого корня. Воспользуемся формулой Лагранжа

$$f(x^{(k)} + \Delta x^{(k)}) = f(x^{(k)}) + f'(x^{(k)})\Delta x^{(k)} + o(\Delta x^{(k)})$$

Полагая $x = x^{(k)} + \Delta x^{(k)}$, исходное нелинейное, вообще говоря, уравнение заменим линейным уравнением, отбросив в последнем выражении $o(\Delta x)$:

$$f(x^{(k)}) + f'(x^{(k)})\Delta x = 0$$

Это линейное уравнение можно записать также в виде

$$f'(x^{(k)})\Delta x^{(k)} = -f(x^{(k)})$$

Будем считать, что $\Delta x^{(k)} = x^{(k+1)} - x^{(k)}$, т.е. что $x^{(k+1)} = x^{(k)} + \Delta x^{(k)}$. Таким образом, задача построения последовательности $\{x^{(k)}\}$ сводится к итерационному вычислению значения $\Delta x^{(k)}$ путем решения последнего линейного уравнения.

Если рассматривать отображение $f : \mathbb{R} \to \mathbb{R}$, с чего мы, собственно, и начали, то решение этого уравнения сводится к последовательным вычислениям по формуле:

$$\Delta x^{(k)} = -f(x^{(k)}) / f'(x^{(k)})$$

Однако, мы могли бы рассмотреть и отображение вида $f : \mathbb{C} \to \mathbb{C}$ (комплекснозначную функцию комплексного переменного). Например, необходимость этого возникнет, если понадобится вычислить некоторый комплексный корень какого-либо многочлена (с вещественными или с комплексными коэффициентами).

Оказывается, что все полученные формулы сохраняют свою силу и в этом случае. Это происходит потому, что формула Лагранжа, на которой основывался вывод рекуррентной формулы $x^{(k+1)} = x^{(k)} + \Delta x^{(k)}$ справедлива и для так называемых аналитических функций комплексного переменного. В частности, аналитическими являются все элементарные функции, которые распространяются на комплексные числа с помощью представляющих их рядов Тейлора (каждый рад Тейлора можно рассматривать в комплексной области).

Единственное уточнение, которое требуется сделать в этой связи, это определение понятия производной для аналитической функции комплексного переменного, которое по форме не отличается от определения производной вещественной функции:

$$f'(z) = \lim_{|\Delta z| \to 0} \frac{f(z + \Delta z) - f(z)}{\Delta z}$$

Из этого определения легко получить формулы дифференцирования элементарных функций, которые по виду в точности совпадают с известными формулами для функций вещественного переменного. В частности, легко проверить, что

$$(z^n)' = nz^{n-1}$$

Более того, мы могли бы рассмотреть, например, отображение вида

$$f : \mathbb{R}^N \to \mathbb{R}^N$$

($N > 1$). Такое отображение определяется набором функций $f_k : \mathbb{R}^N \to \mathbb{R}$.
Таким образом, система из N уравнений с N неизвестными

$$\begin{cases} f_1(x_1, \ldots, x_N) = 0 \\ f_2(x_1, \ldots, x_N) = 0 \\ \ldots\ldots\ldots\ldots\ldots \\ f_N(x_1, \ldots, x_N) = 0 \end{cases}$$

кратко может быть записана в том же самом виде, что и в скалярном случае

$$f(x) = 0$$

с тем лишь отличием, что теперь $x \in \mathbb{R}^N$ и $f : \mathbb{R}^N \to \mathbb{R}^N$.

Вторая важная деталь, состоит в том, что теперь в уравнении

$$f'(x^{(k)})\Delta x^{(k)} = -f(x^{(k)})$$

под $f'(x)$ надо понимать матрицу Якоби:

$$f'(x) = \begin{bmatrix} \frac{\partial f_1(x)}{\partial x_1} & \ldots & \frac{\partial f_1(x)}{\partial x_N} \\ \ldots & \ldots & \ldots \\ \frac{\partial f_N(x)}{\partial x_1} & \ldots & \frac{\partial f_N(x)}{\partial x_N} \end{bmatrix}$$

Таким образом, это уравнение представляет собой систему линейных уравнений, матрица коэффициентов которой есть матрица Якоби.

Все это является просто следствием того, что формула Лагранжа для гладких отображений

$$f : \mathbb{R}^N \to \mathbb{R}^N$$

имеет тот же самый вид, что и в случае $N = 1$ (за исключением того, что теперь $N > 1$).

1.2. Выбор начального приближения

Описанная рекуррентная (итерационная) процедура поиска корня уравнения (скалярного или многомерного) в общем случае не гарантирует сходимости к корню. Для обеспечения сходимости необходимо выбрать подходящее начальное приближение. Это начальное приближение должно быть достаточно "близко" к корню уравнения. Эта "близость" не всегда означает малое расстояние, иногда начальное приближение может находится и на весьма большем расстоянии от предельной точки. Важно лишь, чтобы это начальное приближение попадало в "область притяжения" к искомому корню.
Если уравнение имеет несколько корней, то начальное приближение может попасть в "область притяжения" какого-то одного из его корней. На практике иногда бывает трудно определить заранее в область притяжения какого именно корня попадает выбранное начальное приближение. Кроме того, начальное приближение может быть выбрано совсем неудачно, так, что получаемая последовательность вообще не будет сходиться ни к какому пределу. На практике часто выбор подходящего начального приближения связан с численными экспериментами.

1.3. Программная реализация метода Ньютона

Все сказанное позволяет реализовать метод Ньютона в виде следующего кода на языке Julia

```
using LinearAlgebra: norm

function newthon(delta::Function, x; atol = 1e-8, nmax_iter = 20)
    dx = delta(x)
    n = 0
    while norm(dx) > atol
        x .+= dx
        dx .= delta(x)
        n += 1
        if n > nmax_iter
            @warn(
                "Превышено установленое число итераций (nmax_iter =
$(nmax_iter))"
            )
            return nothing
        end
    end
    return x
end
```

```
function newthon(delta::Function, x::Number; atol = 1e-8, nmax_iter = 20)
    dx = delta(x)
    n = 0
    while abs(dx) > atol
        x += dx
        dx = delta(x)
        n += 1
        if n > nmax_iter
            @warn(
               "Превышено установленное число итераций (nmax_iter =
$(nmax_iter))"
            )
            return nothing
        end
    end
    return x
end
```

Здесь для случая скалярного уравнения пришлось реализовать отдельную специализацию функции `newthon` поскольку для скалярных переменных поэлементные операции (broadcasting) не определены. Аргумент аннотирован абстрактным типом `Number`, это позволит с помощью функции `newthon` искать как вещественные, так и комплексные корни.

В обоих вариантах функции `newthon` введен параметр `nmax_iter` для задания максимального числа итераций, поскольку при неудачном выборе начального приближения сходимость процесса может отсутствовать.

В случае векторного аргумента вместо функции `abs` использована функция `norm` из стандартного пакета `LineraAlgebra`. Эта функция возвращает евклидову норму вектора (т.е. его длину).

Строго говоря, сравнение величины приращения `dx` с пороговым значением `atol` не гарантирует, что абсолютная погрешность не будет превосходить величины `atol`. Тем не менее, соответствующее условие означает, что в результате расстояние между двумя последовательными приближениями окажется достаточно малым. На практике этого часто бывает достаточно.

1.4. Способы определения функции `delta(x)`

Оба рассмотренных метода функции `newthon`, касающихся как скалярного, так и многомерного случаев, требуют передачи ей функции `delta` в качестве аргумента функционального типа. При этом функция `delta(x)` должна возвращать решение линейного уравнения

$$f'(x)\Delta x = -f(x)$$

относительно Δx. В случае скалярного уравнения $f(x) = 0$, $f'(x)$ - это скалярный коэффициент соответствующего линейного уравнения, а в случае, если уравнение $f(x) = 0$ является N-мерным, то $f'(x)$ - квадратная матрица размера $N \times N$ (матрица Якоби). В любом случае `f(x)` и `f'(x)` должны вычисляться внутри функции `delta(x)`.

Рассмотрим сначала скалярный случай, когда $N = 1$. В этом случае решение соответствующего линейного уравнения имеет вид:

$$\Delta x = -\frac{f(x)}{f'(x)}$$

(разумеется, должно выполняться условие $f'(x) \neq 0$). Если функция $f(x)$ задана аналитически, и это аналитическое выражение не является чрезмерно громоздким, то можно вручную вычислить производную, после чего функция `delta(x)` может быть легко определена и передана в функцию `newthon`.

Если же аналитическое дифференцирование не возможно, то можно, например, воспользоваться приближенной формулой

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}$$

где величина h должна быть выбрана исходя из требования получить максимальную точность аппроксимации. Если не учитывать, что в реальности вычисления осуществляются с мантиссой ограниченной длины, то тогда получается, что чем меньше величина $h > 0$, тем точнее будет результат. Однако конечность длины мантиссы означает, что значения функции вычисляются с некоторой погрешностью, и это ограничивает возможность бесконечного уменьшения величина h.

В самом деле, если, образно говоря, рассмотреть график функции "под очень сильным увеличительным стеклом", то в окрестности произвольной точки x этот график будет выглядеть, как пила, с очень мелкими зубчиками. Расстояние между соседними зубчиками - это расстояние между соседними машинными числами с плавающей точкой, а высоты зубчиков являются случайными и вызваны погрешностью вычислений с мантиссой конечной длины. Ясно, что если приращение h брать близким к расстоянию между соседними зубчиками этой воображаемой пилы, то точность оценки производной (тангенса угла наклона касательной) будет падать катастрофически (чем меньше h, тем в большей степени наклон касательной определяется случайными высотами зубчиков).

Можно показать, что для `Float64` для достижения максимальной точности оценки производной значение `h`, близкое к оптимальному, будет порядка 1e-8`.

Таким образом, выбрав правильно значение h мы также можем получить простое выражение, определяющее функцию `delta(x)`:

```
const h = 1e-8 # для случая вычислений с Float64
delta(x) = - f(x)/(f(x+h)-f(x-h)*2h
```

Кроме этого, значение производной функции может быть получено одновременно с вычислением значения самой функции (подобно тому, как это мы делали, вычисляя значение многочлена в точке по расширенной схеме Горнера) методом так называемого автоматического дифференцирования, о котором речь пойдет далее.

Если рассматривается случай $N > 1$, то, как уже был отмечено, $f'(x)$ - это матрица Якоби размера $N \times N$, т.е. матрица составленная из частных производных координатных функций $\frac{\partial f_k(x_1,..,x_N)}{\partial x_m}$.

Эти частные производные могут быть, например, вычислены аналитически, или заменены отношением конечных приращений:

$$\frac{\partial f_k(x_1,\ldots,x_N)}{\partial x_m} \approx \frac{f_k(x_1,\ldots,x_m+h,\ldots,x_N) - f_k(x_1,\ldots,x_m-h,\ldots,x_N)}{2h}$$

Кроме того, в многомерном случае также можно воспользоваться уже упомянутым автоматическим дифференцированием.

1.5. Решение систем линейных уравнений стандартными средствами языка Julia

В случае $N > 1$ для вычисления функции `delta(x)` необходимо решать соответствующую систему линейных алгебраических уравнений (СЛАУ) (матрица которой может быть получена одним из рассмотренных выше способов). В связи с этим рассмотрим стандартные средства языка Julia для решения СЛАУ.

Пусть имеется система линейных уравнений, которая в матричной форме имеет вид

$$Ax = b$$

где A - не вырожденная квадратная матрица размера N , b - произвольный вектор-столбец длины N, $x = [x_1,\ldots,x_N]$ - неизвестный вектор, который требуется найти из этого уравнения.

В языке Julia имеется простое встроенное средство решения таких уравнений. Рассмотрим это на следующем примере.

```
julia> A=[1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> b = [1, 1]
2-element Vector{Int64}:
 1
 1

julia> x = A\b
2-element Vector{Float64}:
 -1.0
  1.0
```

Запись вида `A\b` приводит к вызову специальной процедуры решения системы линейных уравнений. При этом надо обратить внимание, что здесь использована не операция обычного деления - деления слева-направо, а операция обратного деления - справа-налево. Этим как бы говорится, что b надо разделить на A слева (по аналогии с умножением на обратную матрицу слева), но на самом деле тут осуществляется процедура решения системы линейных уравнений наподобие метода Жордана-Гаусса (вычисление обратной матрицы тут было бы не оправдано затратным).

Замечание . Если бы вместо $Ax = b$ рассматривалась бы уравнение вида $xA = b$, где `x, b` - это вектор-строки (а не вектор-столбцы), то решение такого уравнения в Julia получается с помощью операции обычного деления (слева-направо):

```
x=b/A
```

Такая запись как бы показывает, что в математическом (но не в практическом) смысле это равнозначно умножению на обратную матрицу справа.

2. Автоматическое дифференцирование на основе дуальных чисел

Под автоматическим дифференцированием понимают возможность вычисления производной функции вместе с ее значением. Причем вычисление производной не основывается на приближенных формулах типа отношения конечных приращений, а получается с той же погрешностью, что и сама функция, и обусловливается исключительно погрешностью вычислений с плавающей точкой (а не тем, что сам метод дифференцирования является приближенным - в этом смысле автоматическое дифференцирование - это точный метод дифференцирования).

Автоматическое дифференцирование не сводится также и к символьному дифференцированию, реализуемому пакетами компьютерной алгебры (символьной математики). В отличие от символьного дифференцирования, результатом которого является символьное выражение, при автоматическом дифференцировании, и в результате, и на всех промежуточных шагах получается непосредственно численное значение производной (что является существенно более экономичным).

Автоматическое дифференцирование играет важную роль, например, при решении оптимизационных задач, где требуется эффективное вычисление градиента функции многих переменных. Автоматическое дифференцирование также важно для реализации метода Ньютона, где требуется вычисление якобианов (каждая строка матрицы Якоби есть градиент некоторой функции). Существует два подхода к автоматическому дифференцированию - это "дифференцирование вперед" (Forward Mode, ForwardDiff) и "дифференцирование назад" (Reverse Mode, Backpropagation).

Такие названия связаны с тем, что для каждой вычисляемой функции можно построить так называемый вычислительный граф. В вершинах такого графа находятся исходные данные и вычисляемые значения (значения операций), включая и промежуточные, а его ребрам соответствуют операции (преобразования), осуществляемые с этими значениями.

Так вот, "дифференцирование вперед" означает, что значение производной получается при продвижении по этому графу из начала в конец (вместе с направлением вычисления самой функции). Напротив, "дифференцирование назад" означает, что значение производной получается при обратном прохождении этого графа, в направлении, противоположном направлению вычисления самой функции.

Когда требуется вычисление градиента функции многих переменных "дифференцирование назад" может оказаться существенно более эффективным, особенно при очень большом числе переменных. При не слишком большом числе переменных дифференцирование вперед может оказаться предпочтительнее. Если же дифференцируется отображение $f : \mathbb{R}^N \to \mathbb{R}^M$ и M - достаточно большое (например, $M = N$), то "дифференцирование вперед" тоже будет предпочтительнее.

Мы ограничимся рассмотрением только реализацией "дифференцирования вперед" на основе так называемых дуальных чисел. Высокоэффективная реализация процедуры "дифференцирования назад" является существенно более сложной задачей. Она реализована в специальных пакетах (в Julia - это `Zygote.jl`, `Enzyme.jl`). Процедура "дифференцирования вперед" тоже реализована в специальных пакетах (в Julia - это пакет `ForwardDiff.jl`, где реализация основывается на дуальных числах, и уже упомянутый пакет `Enzyme.jl`, где это реализовано иначе).

2.1 Дуальные числа

Дуальные числа есть разновидность комплексных чисел, но в отличие от привычных комплексных чисел, где квадрат мнимой единицы равен -1, квадрат мнимой единицы дуальных чисел (обозначаемой ε) равен 0. Множество дуальных чисел, в отличие от стандартных комплексных чисел, не является полем, оно является только кольцом.

Итак, под дуальными числами понимаются алгебраические объекты вида

$$a + b\varepsilon$$

где $a,\ b \in \mathbb{R}$ и где $\varepsilon^2 = 0$ (ε - это просто символ, который может участвовать в арифметических операциях, и для которого определено такое свойство).

Правила арифметических действий с дуальными числами.

1. По отношению к операциям сложения и вычитания дуальные числа можно рассматривать просто как двумерные вектора (также, как и обычные комплексные числа).
2. Операция умножения дуальных чисел согласуется с обычными правилами арифметических действий и определяется свойством $\varepsilon^2 = 0$. А именно, если $z_1 = a_1 + b_1\varepsilon$ и $z_2 = a_2 + b_2\varepsilon$, то

$$z_1 z_2 = a_1 a_2 + (b_1 a_2 + a_1 b_2)\varepsilon$$

3. Аналогично и операция деления дуальных чисел согласуется с обычными правилами арифметических действий и определяется свойством $\varepsilon^2 = 0$. А именно, если $z_1 = a_1 + b_1\varepsilon$ и $z_2 = a_2 + b_2\varepsilon$, то

$$\frac{z_1}{z_2} = \frac{a_1 + b_1\varepsilon}{a_2 + b_2\varepsilon}$$

```
Избавляясь в последнем выражении от мнимой части в знаменателе путем
домножения на соответствующее сопряженное выражение, получаем
```

$$\frac{z_1}{z_2} = \frac{a_1 a_2 + (b_1 a_2 - a_1 b_2)\varepsilon}{a_2^2} = \frac{a_1}{a_2} + \frac{(b_1 a_2 - a_1 b_2)}{a_2^2}\varepsilon$$

Сравнивая эти свойства с правилами дифференцирования суммы, разности, произведения и частного двух функций, можно заметить следующее. Если действительные части двух дуальных чисел будут представлять собой значения некоторых двух функции в некоторой точке, а мнимые части этих дуальных чисел, соответственно, будут представлять собой значения производных этих функции в той же точке, то сумма, разность, произведение и частное таких дуальных чисел в вещественной части даст, соответственно, сумму, разность, произведение или частное значений этих функций, а в мнимой части - соответствующее значение

производной разности, суммы, произведения или частного рассматриваемых функций.

2.2. Расширение области определения дифференцируемых функций вещественной переменной на соответствующую область дуальной плоскости

Пусть функция $f(x)$ дифференцируемая на $\mathbb{R}$ (или хотя бы не некотором интервале числовой оси), тогда если $z = x + y\varepsilon$ ($x,\ y \in \mathbb{R}$), то положим

$$f(z) = f(x) + yf'(x)\varepsilon$$

Такое определение мотивировано следующим обстоятельством. Пусть рассматриваемая функция не просто дифференцируема, а является аналитической в некоторой окрестности точки $x \in \mathbb{R}$, т.е. представима на этом интервале своим рядом Тейлора:

$$f(x + \Delta x) = \sum_{k=0}^{+\infty} \frac{1}{k!} f(x)^{(k)} (\Delta x)^k$$

Тогда, если формально положить $\Delta x = y\varepsilon$, то обозначив $z = x + y\varepsilon$, получим, что в сумме ряда останется всего два члена (все члены при $k > 1$ обратятся в 0, в силу того, что $\varepsilon^2 = 0$)

$$f(z) = f(x) + yf'(x)\varepsilon$$

Таким образом, если область определения дифференцируемой функции вещественной переменной расширена до соответствующей области дуальных чисел, то для получения значения этой функции в вещественной точке x одновременно с значением ее производной в той же точке, ее надо вычислить для $z = x + 1\varepsilon$. Тогда вещественная часть результата будет равна $f(x)$, а мнимая - $f'(x)$.

Рассмотрим теперь, чему равно значение композицию двух дифференцируемых функций при значении аргумента $z = x + y\varepsilon$

$$f(g(z)) = f(g(x) + yg'(x)\varepsilon) = f(g(x)) + yg'(x)f'(g(x))\varepsilon$$

Таким образом, если $y = 1$, то мнимая часть результата равна

$$g'(x)f'(g(x)),$$

что в соответствии с цепным правилом дифференцирования композиции функций есть

$$(f(g(x)))'$$

Замечание 1. В действительности, вещественную и мнимую части дуального числа $z = x + y\varepsilon$, т.е. x, y не обязательно считать вещественными числами. Например, если

положить $x, y \in \mathbb{C}$, то это позволит вычислять значение вместе с производной комплексной функции (при условии, что комплексная функция дифференцируема).

Кроме того, для получения значений производных более высокого порядка величины x, y следует тоже считать дуальными числами. Дуальные числа, у которых вещественная и мнимая части тоже являются дуальными числами называются **гипердуальными** числами. С помощью гипердуальных можно реализовать автоматическое дифференцирование высших порядков.

Замечание 2. Интересно, что если бы мы рассмотрели случай, когда $z = x + yi \in \mathbb{C}$ и положили бы $\Delta x = yi$, то получили бы близкий по содержанию результат:

$$f(z) = f(x) + yf'(x)i + o(y)$$

где член $o(y) \in \mathbb{C}$ можно отбросить, если взять величину y достаточно малой.

Это могло бы быть использовано для реализации идеи автоматического дифференцирования в некоторых старых языках программирования, которые поддерживают комплексные типы данных, но не поддерживают возможность определять новые пользовательские типы данных, потому что в них отсутствовали структуры (например, в очень старых версиях языка FORTRAN; в современных версиях этого языка структуры и элементы ООП уже имеются).

2.3. Автоматическое дифференцирование на основе пользовательского типа Dual{T}

Для реализации идеи автоматического дифференцирования спроектируем параметрический тип, представляющий дуальные числа.

```
struct Dual{T} <: Number
    re::T
    im::T
end

Base.zero(::Type{Dual{T}}) where {T} = Dual(zero(T), zero(T))
# zero(Dual{Float64}) -> 0.0 + 0.0*ε
Base.zero(x::Dual{T}) where {T} = zero(Dual{T})
# x = Dual(1.0,2.0); zero(x) -> 0.0 + 0.0*ε

Base.one(::Type{Dual{T}}) where {T} = Dual(one(T), zero(T))
# zero(Dual{Float64}) -> 1.0 + 0.0*ε
Base.one(x::Dual{T}) where {T} = one(Dual{T})
# x = Dual(1.0,2.0); zero(x) -> 1.0 + 0.0*ε
```

```
Base. +(x::Dual, y::Dual) = Dual(x.re + y.re, x.im + y.im)
Base. +(x::Dual, y) = Dual(x.re + y, y.im)
Base. +(x, y::Dual) = y + x
# Аналогичным образом требуется определить остальные арифметические операции^
# -, *, /

Base. <(x::Dual, y::Dual) = x.re < y.re
Base. >(x::Dual, y::Dual) = x.re > y.re
Base. <=(x::Dual, y::Dual) = x.re <= y.re
Base. >=(x::Dual, y::Dual) = x.re >= y.re

Base. isapprox(x::Dual, y::Dual) = isapprox(x.re, y.re)

Base.sin(x::Dual) = Dual(sin(x.re), x.im*cos(x.re)
Base.cos(x::Dual) = Dual(cos(x.re), -x.im*sin(x.re)
Base.exp(x::Dual) = Dual(exp(x.re), x.im*exp(x.re)
Base.log(x::Dual) = Dual(log(x.re), x.im/x.re)
# Аналогичным образом требуется определить все остальные
# стандартные математические функции

Base. ^(x::Dual, y::Dual) = exp(y*log(x))
Base. ^(x::Dual, y) = exp(y*log(x))
Base. ^(x, y::Dual) = exp(y*log(x))
```

Замечание 1. В приведенном коде у типа `Dual` в некоторых случаях не указывается параметр. Это возможно потому, что мы используем конструктор по умолчанию, а в конструкторе по умолчанию всегда реализован вывод типа из полученных аргументов.

(В принципе, вывод типа можно реализовать и при переопределении конструктора по умолчанию, для этого в языке предусмотрена специальная синтаксическая конструкция).

Замечание 2. Как можно заметить, редактор кода выделил имя `im` красным цветом. Произошло это потому, что слово `im` в языке Julia является зарезервированным ключевым словом, оно обозначает мнимую единицу поля $\mathbb{C}$, обозначаемую обычно буквой `i` или `j` (в технических приложениях). Так, например, комплексное число $1 + 2i$ на языке Julia в виде значения типа `Complex{Float64}` записывается, как `1.0 + 2.0im` (знак умножения тут ставить не обязательно). Однако у нас имя `im` - это имя поля структуры, и в этом качестве оно допустимо (хотя редактор кода это проигнорировал).

Для автоматического дифференцирования функций будет полезна также следующая функция

```
# Возвращает кортеж из значения заданной функции и ее производной в заданной
точке
```

```
function valdiff(f::Function, x::Number)
     x = Dual(x, one(x))
     y = f(x)
     return y.re, y.im
end
```

Разумеется, наша реализация дуальных чисел получится сильно упрощенной, и она предназначается исключительно для учебных целей. Для практического же использования дуальных чисел в задачах, где требуется автоматическое дифференцирование (вперед) нужно использовать существующий для этого пакет `ForwardDiff.jl`.

3. Суммирование степенных рядов

Рассмотрим пример. Пусть требуется написать функцию, вычисляющую функцию $\sin(x)$ по ее разложению в ряд Тейлора:

$$\sin(x) = \sum_{k=1}^{+\infty} (-1)^{k-1} \frac{x^{2k-1}}{(2k-1)!},$$

Т.е. требуется при достаточно большом n вычислить n-ю частичную сумму этого ряда. При этом необходимо обеспечить оценку вычислительной сложности полученного алгоритма - $O(n)$.

Хотя, разумеется, речь идет о приближенных вычислениях, точность вычислений должна быть по возможности высокой. Ясно, что при каждом фиксированном значении x члены этого ряда монотонно убывают по абсолютной величине, по крайней мере, начиная с некоторого k. Поэтому мы могли бы задаться, например, некоторым достаточно малым значением $\varepsilon > 0$, и суммировать члены ряда до тех пор, пока по абсолютной величине они больше этого ε. Есть теорема о знакочередующихся рядах (теорема Лейбница), а наш ряд как раз именно таковым и является, которая утверждает, что при соблюдении определенных условий (которые в нашем случае соблюдаются) погрешность вычисления суммы ряда не превосходит величины первого отброшенного члена ряда (вследствие того, что его величина стала меньше ε).

Таким образом, с оценкой погрешности вычислений, по крайней мере теоретической, будем считать, что разобрались. Хотя, на самом деле, мы здесь пока не учли особенности арифметики с плавающей точкой, а это весьма важный момент, к которому мы еще вернемся. Рассмотрим теперь вопрос о оценке асимптотической сложности вычислений.

Обозначим

$$a_k = (-1)^{k-1}\frac{x^{2k-1}}{(2k-1)!}$$

(x мы считаем произвольной, но фиксированной величиной). Тогда задача состоит в вычислении суммы

$$\sum_{k:\ |a_k|>\varepsilon} a_k$$

Пусть в сумму с таким условием вошло ровно n первых членов ряда, тогда, если члены ряда будут вычисляться непосредственно по приведенной выше общей формуле для a_k, то оценка сложности алгоритма составит $O(n^2)$. В самом деле, по этой формуле мы должны будем последовательно вычислять $1!, 3!, 5!, 7!, \ldots$, т.е. 1, $1*2*3$, $1*2*3*4*5$, $1*2*3*4*5*6*7$, и т.д. Если просуммировать все операции умножения, то получим

$$0+2+4+6+\ldots+2n = n(n+1) = O(n^2)$$

(не говоря уже о том, что надо еще вычислять все степени x).

Однако мы можем получить алгоритм вычислений с оценкой сложности $O(n)$, если избавимся от повторных вычислений. Для этого величины a_k надо вычислять не по общей формуле, а рекуррентно. Чтобы получить требуемую рекуррентную формулу, проделаем следующее:

$$\frac{a_{k+1}}{a_k} = \frac{(-1)^k\frac{x^{2k+1}}{(2k+1)!}}{(-1)^{k-1}\frac{x^{2k-1}}{(2k-1)!}} = -x^2\frac{(2k-1)!}{(2k+1)!} = -x^2\frac{(2k-1)!}{(2k-1)!2k(2k+1)} = -\frac{x^2}{2k(2k+1)}$$

Таким образом, искомая рекуррентная формула имеет вид

$$a_{k+1} = -a_k\frac{x^2}{2k(2k+1)}$$

при этом $a_1 = x$.

Вернемся теперь к обсуждению точности вычислений. Вспомним, что мы хотели просуммировать все те первые члены ряда, абсолютная величина которых не превосходит заданной малой величины $\varepsilon > 0$ (надеясь при этом, что получим погрешность не превосходящую этого ε, как утверждает соответствующая теорема Лейбница). В этом случае, с учетом полученной рекуррентной формулы, соответствующий программный код мог бы (при наивном взгляде на вещи) выглядеть следующим образом.

```
function sinus(x, ε)
    s = 0.0
    xx = x*x
```

```
    a = x # - начальное значение для последующего использования рекуррентной
формулы
    k = 1 # - номер очередного члена ряда
    while abs(a) > ε
        s += a
        a = -a*xx/2k/(2k+1) # - по рекуррентной формуле
        k += 1
    end
    return s
end
```

Однако в этом коде не учитывается, что вычисления производятся не с абсолютной точностью, а в арифметике с плавающей точкой. Но если мы учтем это, то обнаружится, что мы не всегда сможем увеличивать величину `s` на величину `a`. Например, если мы вычисляем в формате `Float64`, где длина мантиссы составляет пример 16 десятичных знаков, то как только величина `a` станет на 16 десятичных порядков меньше величины `s`, то при `a != 0` будет иметь место равенство `s + a == s`.

Поэтому эти вычисления правильно программировать так

```
function sinus(x)
    s = zero(typeof(x)) # - значение нуля того же типа, что и x
    # (это - для обеспечения стабильности типа переменной s)
    xx = x*x
    a = x # - начальное значение для последующего использованя рекуррентной
формулы
    k = 1 # - номер очередного члена ряда
    while s + a != s
        s += a
        a = -a*xx/2k/(2k+1) # - по рекуррентной формуле
        k += 1
    end
    return s
end
```

Здесь явно учтено, что вычисления выполняются с точностью до машинного эпсилон (т.е. фактически на переделе возможной точности, которую дает арифметика с плавающей точкой), и это позволило избавиться от лишнего параметра функции. Более того, в первом (наивном) варианте нашей функции параметр `ε` может вводить в заблуждение, поскольку значение этого параметра формально ничем не ограничено, но если ему присвоить значение меньше машинного эпсилон, то погрешность результата не сможет быть получена с такой точностью (реальная точность результата всё-рано будет ограничена именно машинным эпсилон).

Приведенный код можно еще немного оптимизировать, а именно, можно сделать так, чтобы сумма `s+a` не вычислялась дважды на каждой итерации цикла (для этого надо ввести буферную переменную). Кроме того, вместо вычисления `2k` (причем дважды) можно ввести переменную `m`, в которой рекуррентно вычислять последовательность всех четных чисел (до начала цикла: `m=0`, в цикле: `m += 2`).

Замечание. Чем больше значение аргумента `x`, тем больше членов ряда потребуется включить в вычисляемую сумму для обеспечения предельной точности. Более того, из за этого при слишком больших значениях аргумента вычисления могут стать численно не устойчивыми, т.е. погрешность вычислений, обусловленная арифметикой с плавающей точкой, может стать катастрофически большой. В случае вычисления периодической функции эта проблема может быть решена использованием формул приведения, с тем, чтобы фактически исключить значения аргумента за пределами главного периода. В случае же непериодической функции такой возможности нет, и для предотвращения потери точности остается использовать тип данных `BigFloat`, который может иметь мантиссу практически любой длины (в языке Julia тип `BigFloat` является встроенным).

Чтобы узнать установленную длину мантиссы типа `BigFloat` надо сделать следующее:

```
julia> precision(BigFloat,base=10)
77

julia> precision(BigFloat,base=2)
256
```

Как видим, по умолчанию длина мантиссы установлена раной 256 двоичных разрядов, что примерно равно 77 десятичным разрядам. Чтобы изменить длину мантиссы типа `BigFloat` надо воспользоваться функцией `setprecision`.