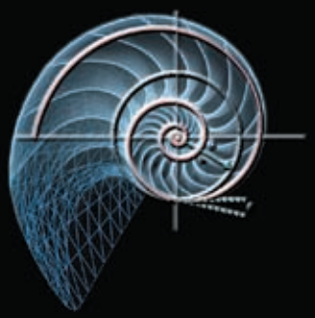

Пишем свой diff()

БОНУС В этом приложении-практикуме Тихон Тарнавский покажет, как использовать Maxima для решения «настоящих» задач.

Сначала я хотел рассмотреть несколько отдельных практических примеров: и маленьких, и чуть побольше. Но потом мне подумалось, что один, но более серьезный пример будет значительно лучше: с одной стороны, его можно строить понемногу, отрабатывая отдельные приемы точно так же, как это было бы сделано и с меньшими примерами, а с другой – в результате все эти приемы переплетутся между собой во что-то объемное, и на этих переплетениях возникнет более цельное ощущение возможностей программы, чем на несвязанных маленьких кусочках. К тому же по ходу дела мы соорудим несколько небольших вспомогательных функций, а заодно, для дополнительной практики, и более расширенную версию одной из них, которая, вполне возможно, пригодится вам и в дальнейшем.

А писать мы будем настоящую функцию дифференцирования. Практически такую же, как встроенная diff(), только без вычисления полного дифференциала – чтобы не слишком сложно было «охватить» пониманием сразу весь пример. Ну а если будет интерес, то дописать вычисление полного дифференциала к этой же функции вы можете попробовать самостоятельно – после освоения возможностей, которые сейчас будут продемонстрированы, это будет уже несложно. Примеров применения по ходу создания функции я давать не буду. Если вы хотите смотреть на практические результаты, по мере добавления кода можно сохранять его в файле, скажем, **~/.maxima/deriv.mac** и выполнять в *Maxima* строку load(deriv)$ deriv(какое-нибудь-выражение);.

Я буду писать код постепенно и по ходу написания давать комментарии к последнему написанному участку. Комментировать буду, просто вставляя куски кода в текст. К слову: *Maxima* поддерживает комментарии в коде «в стиле Си», то есть комментарий начинается символами /*, а заканчивается */. Причем, в отличие от Си, допускаются вложенные комментарии: /* вот /* такие */ */.

Чтобы не повторять каждый раз весь код от самого начала, я буду сокращать его с помощью многоточия. Если вы будете проверять код по мере чтения, не забывайте о разделяющих запятых после последних строк предыдущих участков.

Начнем с «подготовительных работ»: проверки определенных условий и сохранения нужных значений в локальных переменных.

```
deriv([l]):=block([f,len,x],
        len:length(l),
        if len=0 then
        error("deriv can't be used without arguments"),
        f:l[1],
        x:listofvars(f)
)$
```

Итак, по порядку. Символ в квадратных скобках означает, что ему будет присвоен список из всех аргументов, с которыми вызвана функция. Эта конструкция предназначена для создания функций с переменным числом аргументов.

Функция block() – это расширенный аналог составного оператора. Отличается она двумя вещами. Во-первых, поддерживается возврат значений через return(), точно так же как из цикла, то есть по return(выражение) будет осуществлен выход из блока и результатом вычисления блока станет «выражение». А во-вторых, в блоке можно использовать локальные переменные – то есть такие, которые не повлияют на значения символов вне блока, даже если будут иметь совпадающие с ними имена. Такие локальные символы перечисляются в виде списка в самом начале блока.

Далее мы сохраняем в одной из таких локальных переменных длину списка аргументов (функция length) и в случае, если она равна нулю (то есть аргументов нет), генерируем ошибку функцией error, которая может принимать произвольное число аргументов, которые она вычисляет и выводит прежде чем создать ошибку.

Функция listofvars возвращает список переменных переданного ей выражения. Этот список понадобится нам для небольшого расширения возможностей: так как мы не будем вычислять полный дифференциал, то вызов с одним аргументом у нас освобождается, и мы будем его использовать аналогично функции solve: если переданное выражение включает в себя только одну неизвестную, будем дифференцировать его по ней. Продолжаем:

```
deriv([l]):=block([f,len,x],
...
    x:listofvars(f),
    if len=1 then (
        if length(x)=0 then
            return(0),
        if length(x)>1 then
            error("Expression has more than one unknowns and none was
specified.","Unknowns given:", x),
        x:x[1] )
    else
        x:l[2]
)$
```

Если параметр дифференцирования в списке аргументов не задан, то проверяем длину списка неизвестных. Если она равна нулю – то это

константа и следовательно возвращаем ноль. Если больше единицы, то неизвестно, по чему дифференцировать, следовательно, снова генерируем ошибку. Ну а в случае единицы, просто превращаем список из одного элемента в сам этот элемент. Если же список аргументов длиннее, то берем параметр оттуда.

```
deriv([l]):=block([f,len,x],
...
    else
        x:l[2]
    if len>=3 then
    error("More than 2 arguments not implemented yet.")
)$
```

Пока ограничимся производной первого порядка по одной переменной. Когда этот этап будет пройден, остальное будет уже нетрудно написать на основе имеющегося кода. Теперь, когда проверки закончены, приступаем непосредственно к реализации. Строить эту функцию мы будем поэтапно. Для начала научим ее дифференцировать просто переменную и константу:

```
deriv([l]):=block([f,len,x],
...
    error("More than 2 arguments not implemented yet.")
    if atom(f) or subvarp(f) then
        if f=x then
            return(1)
        else
            return(0),
    else
         return 'diff(f,x)
)$
```

Предикат atom() проверяет, является ли его аргумент атомарным выражением, то есть константой (целой либо с плавающей точкой) или одиночным символом. Второй предикат – subvarp() – расшифровывается как subscripted variable (predicate), где первые два слова означают «индексированная переменная», то есть что-то вида **a[1]**. Добавлен этот предикат в эту же проверку в связи с тем, что *Maxima* такие выражения атомарными не считает, а с точки зрения дифференцирования они как раз являются атомами. Дальше в этом варианте все просто: если атомарное выражение является параметром дифференцирования, то результат будет равен единице, иначе – нулю: в полном соответствии с правилами дифференцирования.

В самом конце функции добавляем строку, которая в нештатном случае (таком, который мы еще не посчитали) просто вернет несовершенную форму производной от оставшегося выражения. Эта строка у нас вплоть до самой полной реализации будет оставаться последней, а все остальное мы будем вписывать до нее, сокращая тем самым этому некрасивому умолчательному случаю шансы на выживание. А двигаться дальше мы будем достаточно интересным способом, с помощью уже упомянутой в статье рекурсии. Мы будем постепенно обучать нашу функцию все новым и новым трюкам (точнее, правилам дифференцирования), разбивая неизвестные выражения некоторыми способами на более простые, уже обработанные варианты; то есть действуя снова по известному «принципу чайника». И вы увидите, что математики не зря так любят этот принцип: с его помощью такая, на первый взгляд, сложная задача будет разбита на множество простых подзадачек и таким образом упростится сама. Например, первым пойдет вычитание. Точнее, унарный минус или попросту отрицательные величины: бинарного минуса в *Maxima* по сути не существует, а любое выражение вида **a-b** имеет внутреннюю форму a+(–b), то есть сводится по все тому же принципу к плюсу. Итак, приступим:

```
setup_autoload(stringproc,sequal)$
deriv([l]):=block([f,len,x,o],
...
        else
            return(0),
    o:op(f),
    return (
        if sequal(o,"-") then
            -deriv(-f,x)
        else
            'diff(f,x)
    )
)$
```

Тут мы уже начинаем использовать те самые функции по «глубокой» обработке выражений. Функция op() возвращает основной оператор заданного выражения. Основным считается самый внешний; например op(a+b/c) будет равен "+", op((a+b)*2) – "*", а op(sin(x^2+y^2)) – sin. Дальше включается «принцип чайника»: для отрицательного выражения мы просто выносим минус за скобки, а для остального, теперь уже положительного, вызываем саму же функцию deriv.

Здесь для сверки значения оператора с минусом используется не equal(), а ее строковый аналог – sequal(), проверяющий на равенство две строки. Связано это с тем, что разные операторы *Maxima* хранит в разном виде, и при сверке, скажем, того же минуса, который хранится как текстовый знак с синусом, хранящимся как символ (идентификатор) *Maxima*, обычный equal() просто выдаст ошибку.

Функция sequal() – внешняя, она хранится в файле **stringproc** (от фразы «string processing» – обработка строк), который и нужно подгрузить до использования этой функции. А для того чтобы, файл не приходилось загружать вручную, но при этом он и не загружался бы при каждом вызове функции (как было бы в случае вызова load() внутри функции deriv()), есть, с одной стороны, традиционный способ: определить внутри файла некую константу или свойство, а перед его загрузкой проверять их наличие: если нету – тогда и подгружать. Мы же используем не общепринятый, но в чем-то более простой метод: рассмотренную в статье функцию setup_autoload. Благодаря ей, нам с одной стороны, не надо лезть в исходники библиотек (которые, кстати говоря, часто бывают не на языке *Maxima*, а на Lisp) и искать там флаги; а с другой – мы все же уверены, что файл будет загружаться не больше одного раза: именно это и гарантируется функцией setup_autoload.

И последний момент в этом кусочке: обратите внимание на оператор if, сместившийся внутрь функции return(). Напомню, что if в *Maxima* является полноценным оператором, то есть всегда возвращает последнее вычисленное значение. А раз так, нет никакого смысла вызывать return() много раз. По большому счету, здесь и один вызов return() не нужен: результатом block(), как и примитивного составного оператора, будет последнее вычисленное выражение. Так что для еще большей краткости напишем даже так:

```
...
    o:op(f),
    if sequal(o,"-") then
        -deriv(-f,x)
    else
         'diff(f,x)
)$
```

После минуса логично было бы заняться плюсом; но поскольку сумма при дифференцировании переходит в сумму, то проще будет реализовать ее сразу для произвольного числа слагаемых, а это уже немного сложнее. Потому начнем с более простых в реализации арифметических действий: умножения и деления.

```
...
    if sequal(o,"-") then
        -deriv(-f,x)
    else if sequal(o,"*") then
        deriv(first(f),x)*rest(f)+first(f)*deriv(rest(f),x)
    else if sequal(o,"//") then
    (deriv(first(f),x)*last(f)-first(f)*deriv(last(f),x))/last(f)^2
    else
        'diff(f,x)
)$
```

Здесь мы сталкиваемся с одним очень интересным и весьма полезным свойством: многие из функций работы со списками, которых в *Maxima* немало, воспринимают как списки также и любые выражения. Так, «списковая» функция first(), возвращающая первый элемент заданного списка, вызванная как first(a*b*c), вернет **a**; а у функции rest() («остаток»), отдающей (в варианте вызова с одним аргументом), наоборот, весь список кроме первого элемента, на том же выражении результатом будет **b*c**. Этим мы и воспользовались, вызывая при этом снова для каждого слагаемого саму функцию deriv(). Если сомножителей будет больше чем два, то вызов deriv(rest(f),x) пройдет по этой же ветке и отсечет еще один.

Так же мы поступаем и с делением. Здесь, так как аргумента всегда два, вместо rest() используется функция last() – последний элемент списка (rest() в этом же случае вернула бы список из одного элемента, а потому last() более удобна). Только одно «но»: деление почему-то обозначается во внутреннем представлении *Maxima* не одиночной, а двойной косой чертой.

Точно таким же образом можно обработать последний бинарный оператор (кроме оставленного на закуску сложения) – возведение в степень. Здесь тоже нет никаких сложностей, и даже нечего дополнительно объяснять по сравнению с делением:

```
...
    else if sequal(o,"^") then
        first(f)^last(f)*log(first(f))*deriv(last(f),x)+
        first(f)^(last(f)-1)*last(f)*deriv(first(f),x)
    else
        'diff(f,x)
)$
```

Теперь вернемся к сложению. Тут нам уже пригодятся упомянутые в статье функции по работе с функциями, а конкретно – функция map(). Она принимает в качестве первого аргумента имя функции и как бы вкладывает эту функцию внутрь выражений – последующих аргументов. Проще всего будет пояснить на примере: map(f,[a,b,c]) даст результат [f(a),f(b),f(c)]. И, что самое замечательное, она, точно так же, как и «списковые» функции, работает не только со списками, но и с любыми выражениями; например, map(f,a+b+c) –> f(a)+f(b)+f(c). Как хорошо подходит для нашей задачи, не правда ли? Именно так и должна действовать на сумму функция дифференцирования. Все было бы совсем хорошо, если бы deriv() принимала, кроме выражения, только один аргумент. С двумя выражениями map тоже умеет работать, но только если у них одинаковый основной оператор; то есть сумму можно «отобразить» только на сумму: map(f,a+b+c,x+y+z) –> f(c,z)+f(b,y)+f(a,x). Проблема здесь в том, что у нас второй аргумент во всех вызовах deriv(), которые должны попасть внутрь суммы, одинаков, а выражение вида x+x+x передать невозможно: оно автоматически упростится в 3*x. Но, как известно, из любой безвыходной ситуации всегда есть как минимум два выхода. И в данном случае один из этих выходов достаточно прост: написать небольшую функцию-«обертку» вокруг map:

```
map1st(f,expr,x):=block([o],
    o:op(expr),
    subst(o,"[",map(f,subst("[",o,expr),makelist(x,i,1,length(expr))))
)$
```

Еще одна новая функция «глубинной» работы с выражениями: subst(). Она способна заменять в выражении... да почти что угодно и почти на что угодно. Вызывается так: subst(стало, было, выражение), заменяя в «выражении» все, что «было», на «стало». Опять же, в качестве подвыражений могут использоваться операторы, то есть subst("*","+",x+y+z) –> x*y*z. Мы используем ее для временной подмены основного оператора выражения оператором списка (который обозначается как "[", то есть [a,b,c] – это, по сути, "["(a,b,c)). Затем генерируем список такой же, как выражение, длины, заполненный заданной переменной, – и применяем к двум полученным спискам функцию map(), а затем возвращаем назад вместо списка первоначальный базовый оператор. То есть теперь, к примеру, map1st(f,a+b+c,x) будет равно как раз f(c,x)+f(b,x)+f(a,x). Et voila, как говорят французы! И теперь внутри deriv() можно применить к сложению именно эту новую функцию. Заодно применим ее и к списку – (deriv([f,g],x) будет равно [deriv(f,x),deriv(g,x)]) и, чего уж там мелочиться, и к множеству:

```
...
    o:op(f),
    if sequal(o,"+") or sequal(o,"[") or sequal(o,set) then
        map1st(deriv,f,x)
    else if sequal(o,"-") then
        -deriv(-f,x)
...
```

Множества, к слову, в *Maxima* реализованы в самом что ни на есть математическом смысле: множество может включать в себя каждый элемент только один раз; и это учитывается и встроенными операциями по работе с множествами: пересечением, объединением и т.д. Есть еще некоторые ошибки, но они документированы и потому не неожиданны.

Движемся дальше. У нас уже реализована производная от всех бинарных операторов, а дальше мы нарисуем «таблицу производных» и будем работать с нею:

```
deriv([l]):=block([f,len,o,x,func,fdrv],
...
    o:op(f),
    func:[sqrt, sin, cos, abs, exp, log, tan, cot, sec, csc, asin, acos, atan,
acot, asec, acsc, sinh, cosh, tanh, coth, asinh, acosh, atanh, acoth,
asech, acsch],
    fdrv:[1/2/arg, cos(arg), -sin(arg), arg/abs(arg), exp, 1/arg, sec(arg)^2,
-csc(arg)^2, tan(arg)*sec(arg), -cot(arg)*csc(arg), 1/sqrt(1-arg^2), -1/
sqrt(1-arg^2), 1/(1+arg^2), -1/(1+arg^2), 1/arg^2/sqrt(1-1/arg^2), -1/
arg^2/sqrt(1-1/arg^2), cosh(arg), sinh(arg), sech(arg), -csch(arg), 1/
sqrt(arg^2+1), 1/sqrt(arg^2-1), 1/(1-arg^2), 1/(1-arg^2), -1/arg^2/sqrt(1/
arg^2-1), -1/arg^2/sqrt(1/arg^2+1)],
    if sequal(o,"+") or sequal(o,"[") or sequal(o,set) then
...
```

Для упрощения работы с «таблицей» напишем еще две небольших вспомогательных функции: одна будет проверять, входит ли заданный элемент в заданный список, а вторая – возвращать номер, соответствующий заданному элементу в заданном списке, при условии что он там есть.

```
smember(expr,list):=
    if sequal(true,
        for i in list do
            if sequal(expr,i) then
                return(true) )
    then true$
sindex(expr,list):=block([num],
```

```
    num:for i:1 thru length(list) do
            if sequal(expr,list[i]) then
                return(i),
    if integerp(num) then num
)$
```

Здесь есть только одна тонкость, связанная с небольшой проблемой. Заключается эта проблема в том, что для возвращения значения из блока и из цикла в *Maxima* используется одна и та же функция return(). Это приводит к тому, что выйти из блока, находясь внутри цикла в нем, невозможно – приходится выдумывать некоторые несложные ухищрения. Теперь с использованием двух новых функций заменяем элементы «таблицы» их производными; с помощью уже знакомой нам subst, которая подставит нужное выражение внутрь табличной функции вместо ключевого слова arg.

```
...
    else if smember(o,func) then
    deriv(first(f),x)*subst(first(f),arg,fdrv[sindex(o,func)])
    else
        'diff(f,x)
)$
```

Вот так, начиная с самых простых элементов, а затем, подобно Мюнхгаузену, вытаскивая самих себя сантиметр за сантиметром, мы и получили полноценную функцию дифференцирования. Правда, пока только первого порядка и только по одному аргументу. Но имея то, что имеем, двигаться дальше, следуя известному принципу, уже совсем не сложно: просто заменим строку «if len>=3 then error...» следующим куском:

```
...
    if len=3 then (
        integerp(l[3]) or
            return('diff(x,l[3])),
        if l[3]=0 then
            return(f),
        if l[3]<0 then
            error("Improper count to deriv:",l[3]),
        if l[3]>1 then
            return(deriv(deriv(f,x,l[3]-1),x))
    ),
    if len>3 then (
        if(evenp(len)) then
            l:endcons(1,l),
        return(deriv(apply(deriv,rest(l,-2)),l[len],l[len+1]))
    ),
...
```

Пройдемся по нескольким неосвещенным моментам. В силу способов вычисления в *Maxima* (которые сродны таковым во многих языках программирования) конструкция вида «условие or выражение» равносильно «if not условие then выражение» – и использована здесь исключительно для разнообразия, в учебных целях. Здесь мы в случае нецелого порядка дифференцирования просто возвращаем несовершенную форму – точно так же, как это делает и штатная функция diff().

Производная нулевого порядка от любой функции – это сама функция. А производные отрицательных порядков некорректны, о чем мы и генерируем сообщение. Для порядков, больших единицы, понижаем порядок как и раньше – за счет самовызова.

Далее я немного усовершенствовал поведение функции по сравнению со встроенной: если та не умеет принимать четное количество аргументов больше двух (то есть с неуказанным порядком дифференцирования по последней неизвестной когда неизвестных больше одной), то у нас в данном случае, так же как и для одной неизвестной, будет подразумеваться единица. Здесь предикат evenp() проверяет число на четность (even – четный), а функция endcons() добавляет заданный элемент в конец заданного списка. Ее имя носит исторический характер: парная к ней функция cons(), добавляющая элемент в начало списка, свое имя позаимствовала из Lisp, а слово end здесь добавлено «по смыслу».

Далее мы, снова самовызовом, укорачиваем список параметров дифференцирования. При этом используется еще одна функция, работающая с функциями, – apply() (применять). Она принимает два аргумента, первый из которых – имя функции, а второй – список, и применяет заданную функцию к списку как к списку аргументов. Также здесь использован более широкий вариант вызова rest(): он может принимать второй аргумент – целое число, не равное нулю. Если число положительно, то такое количество элементов выбрасывается из начала списка, а если отрицательно – то с конца; в данном случае мы теряем последние два элемента.

Вот и все. Мы уже имеем полную функцию дифференцирования, берущую производные с произвольным количеством параметров и любых порядков. Полный текст всех созданных функций вы можете найти в файле deriv.mac на прилагаемом к журналу диске.

Дополнительно хочется остановиться на одной незамысловатой функции, которая, тем не менее, может неплохо помочь в отладке собственных модулей. Это функция display(), которая принимает имена и отображает их значения в виде «имя=значение». В качестве эксперимента можете добавить ее где-нибудь внутри функции deriv() и отследить процесс самовызова (в файле на диске вызов display() достаточно раскомментировать).

И в качестве финального аккорда сделаем еще и более универсальную версию вспомогательной функции map1st() – возможно, тогда она вам пригодится и еще где-нибудь.

```
mapany(f,[lst]):=block([o,l],l:lst,
    if length(setify(map(length,l)))>1 then
    error("Arguments to mapany are not of the same length"),
    o:op(l[1]),
    for i:1 thru length(l) do
        l[i]:subst("[",op(l[i]),l[i]),
    subst(o,"[",apply(map,cons(f,l)))
)$
```

Здесь я уже воздержусь от столь подробных комментариев, так как практически все, что используется в этой функции, уже было в той или иной степени разъяснено в процессе описания deriv(). Остановлюсь только на одной строчке:

```
            if length(setify(map(length,l)))>1 then
```

Здесь используется не совсем простой прием для проверки длин списков на одинаковость. Так как l – это список из списков, то сначала получаем список длин «вкручиванием» внутрь внешнего списка функции length(). Дальше – интереснее. Функция setify (дословно – что-то вроде «множествицировать») превращает список в множество. Так как множество не может содержать несколько равных между собой элементов, то такие элементы при этом «склеиваются»: из них остается один. Таким образом если «длина» (количество элементов) множества больше единицы, то как минимум два элемента в первоначальном списке были неравны между собой.

И вернувшись к рассмотренной функции дифференцирования, хочется еще раз обратить ваше внимание на использованный прием: конструировать большие и сложные функции из более маленьких и простых кусочков с помощью рекурсии. Этот метод очень часто и продуктивно используется в функциональном программировании, к которому *Maxima*, в силу своих Lisp-овских корней, очень близка. LXF