Salt Documentation

Release 0.1.0

Volker Siepmann

Dec 01, 2016

CHAPTER

ONE

BRIEF DESCRIPTION

SALT stands for **S**ymbolic **A**lgebra **L**ig**T**. It's main purpose is to provide symbolic derivatives for systems of several thousand variables efficiently in terms of performance and memory. With that, the objective is not to be fit to hold a candle to established packages like sympy or CaSAdi, but to have a lightweight package to fulfil particular needs. The main feature and strategy is that performance of core functionality is not compromised by any additional functionality or convenience.

In its current version, it is even pure python. Porting into a *C*-extension is possible for further performance gain, but the subsequent compiler/version/platform dependencies freak me out.

Date 13.11.2016

Author Volker Siepmann <volker.siepmann@gmail.com>

Projects https://bitbucket.org/repo/all?name=Volker+Siepmann

Contents:

1.1 Introduction

1.1.1 Background and objective

SALT stands for **S**ymbolic **A**lgebra **L**ig**T**, and is also a common additive in food preparation, which makes the name fit in the palette of software developed by me. Salt (as NaCl) is also a substance that can be met in high quantities in nature. This can be seen in parallel to *SALT* being developed to obtain large-scale derivatives.

Without compromise, this package is developed to be small and efficient solely for the purpose of obtaining symbolic derivatives of native-looking python code. I called the first version of this code *sympy*, but others made a much bigger, more general, and for my purposes too slow :-) package with that very same name: Sympy. To call a symbolic package in python *sympy* is of course not a great act of creativity, so this didn't come as a big surprise.

SALT does hardly anything of what Sympy does, even if much of the functionality could be included of course, but this is not the objective of *SALT*. The closest package in terms of objective currently known to me is CasADi, and the basic syntax and approach is similar.

CasADi is again a bigger and embracing package, and with no doubt as well more advanced than *SALT*. Its documentation states for the following interesting benchmark as CasADi code:

```
from casadi import *
x=SX.sym("x")
y=x
for i in range(100):
   y = sin(y)*y
```

"*In Casadi, this code executes in the blink of a second, whereas conventional computer algebra systems fail to construct the expression altogether.*"

A direct comparison has not been done here, but the following native python code with *SALT* executes in 0.6 ms (excluding the import statement):

```
from salt import Leaf, sin
y = x = Leaf(3.14159)
for _ in xrange(100):
    y = sin(y) * y
```

... and it takes 2 ms to derive that equation with respect to *x* and 0.5 ms to evaluate the derivative:

```
from salt import Derivative
z = Derivative([y], [x])[0,0] # derive: z = dy/dx

z.invalidate() # only necessary if value of x changed
print z.value # re-evaluate value after invalidate
```

As with CasADi, the runtime increases linearly with problem size, that is for instance the `100` in the `xrange` statement. The performance is less than an order of magnitude slower than CasADi, but this is not a big shame given that *SALT* is purely python.

A noticeable difference is that *SALT* does not use names (as strings) for the symbols, as they are not needed for the intended purpose. Furthermore, the derivative algorithm is symbolic and with that rather primitive compared to the advanced application of forward and reverse algorithmic differentiation as implemented in CasADi.

The main reason to maintain SALT aside is to hold a lightweight package (currently 50 kB source files, of which most of it is inline documentation) that is not dependent on anything but python itself. I developed SALT to create and evaluate Jacobian matrices of systems with up to several thousand variables - as efficient and pythonic as possible.

1.1.2 Key concept

Each algebraic operation is defined as a node in a directed graph, whereas the edges of the graph represent the dependencies between nodes. For the expression `c = a + b`, `c` is a node of type `+`, whereas the types of `a` and `b` are determined by their definition. Node `c` points to `a` and `b`, but not vice versa. *SALT* supports the following node types:

Leaf (source) nodes These nodes are not dependent on any *child* nodes (operands), but contain a value that can be changed by the client code.

Zero and One and constant nodes These nodes also have no *child* nodes, and their value is fixed to zero, one, or any value respectively. In particular automatically derived code can easily be simplified if these entities are explicitly identified. Other used constants are also cached to some degree, mainly to identify duplicates for simplification purposes.

Unary nodes These represent all unary mathematical functions, such as trigonometric functions, *log*, *exp*, *sqrt*, etc. They have one *child* node, representing their argument. Additional nodes are defined for *1/x* and *x**2* for more efficiency, called *inv* and *squ* respectively.

Operator nodes These are individually implemented nodes to represent all binary operators (+, -, *, /, **) as well as unary minus and a primitive selector (decision) node that evaluates to:

```
x.select(y) := 0 if y<=0 else x
```

The entire data structure of a node consists of

1. A type identifier

2. A cached scalar float value
3. A reference counter
4. A fixed size list of operands (child nodes)

This minimal structure allows to treat large quantities of symbols. In particular, without requiring bidirectional links, the resulting fixed-size record avoids dynamic memory allocation and can on demand easily be implemented in *C* as a python extension for further performance gain, if that is one day necessary. But to place the 853862th quotation of the following:

```
# Premature optimization is the root of all evil
# (or at least most of it) in programming.
#           -- Computer Programming as an Art (1974), Donald E. Knuth
```

Expressions are simplified at earliest possible stage. The code:

```
y = sqrt(x) * sqrt(x)
```

would be instantaneously simplified to:

```
y = squ(sqrt(x)) = x
```

1.1.3 Typical application and workflow

Graph generation

The client code instantiates a number of *Leaf* objects (independent variables). The subsequent procedural code defines the graph, while its procedural nature guaranties the graph to be acyclic.

The user-visible datatype is *Salt*, being the base-class of *Leaf*. It behaves very similar to the built-in `float` type with one major exception, that is the non-existance of comparison operators. We cannot compare the value of symbols at graph generation time, as their value is dynamic.

The procedural code can be part of any *python* language construct, including loops, functions, recursions and classes. It can also be part of container types, due to the mutable nature though not as keys in dictionaries or as items in sets. The *Salt* datatype is a smart-pointer to the node objects (with reference counting) and defines the convenience operators and functions to give the (almost) full `float` experience.

At the end of this phase, the client code has obtained the *dependent* variables, thus both *independent* and *dependent* variables are now available as *Salt* objects.

A small example without any practical justification is:

```
from salt import Leaf, sin, cos, log

x1, x2 = map(Leaf, 2.5, 0.1)
a = x1 * cos(x2)
b = x1 * sin(x2)
y1 = sqrt(a) + log(b)
y2 = y1 * b

x = [x1, x2] # independent variables
y = [y1, y2] # dependent variables
```

Repeated evaluation

Given above code, we can now re-evaluate the dependent variables for different values of the independent variables. To do so, the dependent variables are marked as invalid, and the new values are set to the independent ones. Afterwards, the new values of the dependent variables can be queried:

```
while nobody_is_bored:
    y1.invalidate()
    y2.invalidate()

    x1.value = 2.0 # in real application of course ...
    x2.value = 0.2 #   ... non-constant assignments

    print y1.value, y2.value # ... and processing of these
```

The step calling `invalidate` seems nasty, but is a small price for not requiring bidirectional links between the nodes - with all disadvantages that would yield.

Generating derivatives

For optimisation, equation solving, and other exercises of this kind, the derivatives *dy/dx* are more than welcome. The ability to derive equations is *my* entire motivation to use symbolic algebra:

```
z = Derivative(y, x)
simplify(z)
```

The derivative algorithm already performs the same simplifications as applied by the graph generation phase. In the explicit *in-place* `simplify` call, common terms are identified and simplified to be represented only once, for instance:

```
y = sin(a + b) * cos(b + a)
```

will be simplified to:

```
var_1 = a + b
y = sin(var_1) * cos(var_1)
```

This elimination of duplicates is essential to generate efficient derivatives and might in future versions well be included into the *`Derivative`* class.

Normally, the generated derivative symbols undergo the same repeated evaluation as the dependent variables. Consequently, higher order derivatives are naturally supported, as long as the exponential growths of symbols required to represent higher order derivatives can be handled in memory. You would probably not want to take the 5th derivative of an 800 times 800 system.

1.2 Advanced topics

There are not many advanced topics to *SALT* as a main objective is to keep things simple. Yet, there are some hidden peanuts:

1.2.1 Floats and Leafs

The python operator overloading in *SALT* makes it possible to smoothly mix `float` and *`Salt`* data types. Naturally, the symbolic graph is only built when using *`Salt`* entities. Consider

```
a = Leaf(3.0)
b = 4.0 * 2.0
c = a + b
```

The + operator still converts `b` to a symbolic node before creating the node representing `c`, but this is an anonymous node with no user reference to change its value later on - in contrast to `a`. In the symbolic context, `b` can therefore be called a *constant*. Obviously, the information that `b` is the product of four and two is not preserved either.

Typical applications of such mixing for the sake of readability is:

```
m = Leaf(75.0) # kg
v = Leaf(4.0) # m/s
E = 0.5 * m * squ(v) # Energy of a person running
```

The alternative code with pure data types would look like (**don't do this for the reason described below**):

```
m = Leaf(75.0) # kg
v = Leaf(4.0) # m/s
a_half = Leaf(0.5)
E = a_half * m * squ(v) # Energy of a person running
```

Not only is this less readable or natural, but also can *Salt* in the latter code not know whether the user intends later to change the value of `a_half`. For the upper code, *SALT* can recognise this and reuse nodes of the same value in other expressions by caching. If you are to simulate the *Paris Marathon* with 50000 participants, the upper code would still only hold one reference to constant node of value `0.5`. Simplification could (*does not yet though*) multiply out that factor when adding the energies, and reuse it when deriving the terms.

There is more:

```
f = Leaf(20) # frequency [f] = 1/sec
t = 1 / f # period [t] = sec
```

Above code will recognise `1` as the famous *one* and simplify above equation to

```
t = inv(f)
```

with a simpler derivative and more simplification chances when used further on. This works, because the floats *zero*, *one* and *two* are pre-cached as the special nodes dedicated to them.

As an exception, the *`select`* method does not accept `float` type arguments, just because it would never make sense.

See also:

Attributes *`ALLOW_MIX_FLOAT`* and *`FLOAT_CACHE_MAX`*

1.2.2 Empanadas and Empanadiñas

Empanada is a delicious wrap dish originating from Galicia in Spain, coincidently also the place where my wife grew up. Now, in this context, it is a metaphor for the functionality to wrap your own *meat* into the network (bread) of *SALT* symbolic algebra nodes. Empanadiñas are just small Empanadas.

Say you largely rely on *SALT* to generate the derivatives of the dependent variables y with respect to the independent ones x, but for a block of intermediate equations $u(v)$ with

$$v = v(x) \quad \text{and} \quad y = y(x, u)$$

a manual implementation of the derivatives $\mathrm{d}u/\mathrm{d}v$ is desired. This can have several reasons:

- You need to implement an existing subroutine that can only be evaluated with *float*, but on the other hand is capable of delivering its derivative.
- A considerable part of the equations is more efficiently derived manually.

The concept of a `plain` operator enables this feature in an elegant, even if probably not most efficient way, such that the outer derivative $\mathrm{d}u/\mathrm{d}v$ still can be generated, and new values for y and $\mathrm{d}y/\mathrm{d}x$ can be evaluated without having to consider the inclusion.

The `plain` operator $\mathrm{plain}(x)$ evaluates always to x, but we *forget* the dependencies when deriving, i.e. $\mathrm{d}p/\mathrm{d}x \equiv 0$. Now, this sounds like giving a monkey a screw to open a banana, doesn't it!?

To explain this, we denote symbolic variables with an accent $\hat{\psi}$, and pure numerical variables without (ψ)

Given $u(v)$ and $J = \mathrm{d}u/\mathrm{d}v$ as numerical values from the *unSALTed* subroutine, define the symbols $\hat{u}(\hat{v})$ as a *Taylor* expansion:

$$\hat{u} = u(v) + J \cdot (\hat{v} - \mathrm{plain}(\hat{v}))$$

With multiple variables (that is, u and v are vectors), J is a matrix and the multiplication an inner product. This way, the value and the first derivative of u are correctly evaluated. The series can be expanded in order to reproduce higher order derivatives - though this is not supported by *Empanada* and *Empanadiña* I'm afraid.

For first order (derivative consistent) embedding however, the functionality is implemented as the *empanada* function in general and as the *empanadina* function for scalar functions.

Empanadiña example

Consider the desire to embed the following (`float` type) function into the *SALT* symbolic graph:

```
def func(x):
    y = x ** 6
    J = 6 * x ** 5
    return y, J
```

This is a scalar function that turns its argument x into a function value y, also providing the manually implemented derivative $J = \frac{\mathrm{d}y}{\mathrm{d}x}$.

The following code wraps this function into the symbolic algebra graph:

```
a = Leaf(2.0)
b = sqrt(a)
y = empanadina(func, b) # has the effect of "y = func(b)" in symbolic context
```

A subsequent `dyda = Derivative([y],[a])[0,0]` will give the correct total derivative $\frac{\mathrm{d}y}{\mathrm{d}a} = J \cdot \frac{\mathrm{d}b}{\mathrm{d}a}$.

Empanada example

In most practical cases, the function to embed has either a vectorial input argument, a vectorial return value, or both. The bigger sister of *empanadina*, namely *empanada* is used in this case. Let the function now be:

```
import math
def func(x):
    a, b = x
    c = math.exp(a + b)
    y = [a, a * math.sin(b), c]
    J = [[1.0, 0.0],
         [math.sin(b), a * math.cos(b)],
         [c, c]]
    return y, J
```

The embedding is very similar to above example. We just need to tell the dimensionality of the function result as `dim_out`, because *empanada* needs to prepare the symbols and would not like to call the function just to find it out:

```
x = Leaf(2.0)
z = [x * x, sqrt(x)]
y = empanada(func, z, dim_out=3)

dydx = Derivative(y, [x])
```

The current implementation of *empanadina* is actually only a wrapper of *empanada* to relieve the user from cluttering indexing, like so:

```
def empanadina(func, inp):
    def _func(inp):
        out, jac = func(inp[0])
        return [out], [[jac]]
    return empanada(_func, [inp])[0]
```

This might change in the future according to the plan to let `empanadina` embed functions that deliver n^{th} order derivatives.

1.2.3 Iterative algorithms

The following thinking applies to all iterative algorithms, but is here exemplified with the task of solving an implicit equation or equation system.

Warning: Do not do the following - ever!

You might have the glorious idea to use *SALT* or any other symbolic algebra system as follows in for instance a fixpoint iteration:

```
# 1. solve for some fixpoint
x = Leaf(start_value)
while not converged and still_memory_left:
    dx = f(x, p)
    x = x + dx

# 2. Obtain the derivative of x(p) with respect to p
dxdp = Derivative([x], [p])
```

Warning: Do not do the above - ever!

If you now think: "*Why not?*", please read on.

Here is what you might do instead:

```
x = Leaf(start_value)
y = f(x, p) # generate the function symbolically once!
partial = Derivative([y], [x, p])[0] # take the derivative already
dxdp = -partial[1] / partial[0] # magic equation, see below

while not_converged:
    x.value += y.value # iterate on the graph, don't extend it
    y.invalidate() # don't forget to invalidate before re-evaluate
```

The magic assignment of *dxdp* represents the following mathematics: We know the algorithm to terminate (if successful) with $f(x, p) = 0$. The total differential gives the equation:

$$\left.\frac{\partial f}{\partial x}\right|_p \mathrm{d}x + \left.\frac{\partial f}{\partial p}\right|_x \mathrm{d}p = 0 \quad \Rightarrow \quad \frac{\mathrm{d}x}{\mathrm{d}p} = -\left(\left.\frac{\partial f}{\partial x}\right|_p\right)^{-1} \cdot \left.\frac{\partial f}{\partial p}\right|_x$$

And once you have the derivative $\partial f/\partial x|_p$ at hand, you might just as well use Newton's method to solve $f(x, p) = 0$ instead of the primitive fix-point iteration:

```
x = Leaf(start_value)
y = f(x, p)
partial = Derivative([y], [x, p])[0]
dx = -y / partial[0]

while not_converged:
    x.value += dx.value
    dx.invalidate()
```

This works also perfectly for multi-variant systems.

1.3 Limitations

Limitations can be a bad thing, but also prevent the user from doing stupid things. In that sense, please see the following limitations as features.

1.3.1 Necessity of *invalidate*

I should be sorry for this one, but it is part of the key for the performance.

In a previous version of this package, nodes automatically send their invalidity status upwards the graph whenever their value was set, until an already invalid node was reached. This was convenient from a programmers' point of view. Now, that I don't have it anymore, I myself find me frequently swearing when I discover that I forgot to call `invalidate` again.

But the price for the automatic propagation of validity status upwards is a bidirectional linking of nodes. Profiling my old package revealed that 99% of the time was spent in memory-allocations to handle the dynamic lengths list of node parent pointers - even and in particular after I desperately ported the package to C. Note that frequently used nodes can have thousands of parents within the symbolic graph.

Having written this, I play with the thought to follow another concept, namely to freeze a graph once all knitting, derivatives and simplifications are done. Freezing would install the upwards links (once and for all) and allow again automatic, slightly more efficient, and consistent invalidation. The drawback of this would be memory usage and the

necessity to be strict in keeping frozen graphs imutable. Currently, I would not know how to enforce this at least half way elegantly.

1.3.2 Acyclicity

Would it not be nice to allow cycles in the graph and that way encode iterative algorithms? Or what about replacing existing nodes within the graph with new ones? – **Well you wish!**

The guaranteed non-circular nature of the symbolic graph is a required property for efficient evaluation and construction of derivatives. If you need iterations, please do that outside *SALT* (which is exactly the targeted application) or use another package that provides such functionality.

1.3.3 Numpy and Scipy incompatibility

Well, this one is not easy to sell as a feature, but as a fact, the full numpy functionality is only accessible with a set of standard data types, of which the *SALT* symbols are not one of them.

However, of course the result of what comes out of *SALT* in terms of values is mostly meant to be processed by numpy, scipy and similar packages.

If you however find a native python numeric library, there is a chance that *SALT* objects fit right in – at least as long as nobody tries to use comparison operators on the entities, as for instance to pivot a matrix for decomposition.

Pulling the inside out, it could be useful to define entire linear algebra objects as single symbols. The reason this is not done in *SALT* is the *LT* in the name, and the horrible number (and variants) of binary operators to consider.

1.3.4 Conditionals

The `select()` method is a primitive conditional, but for the sake of differentiability, such support is on purpose kept to a minimum. In the end, conditionals are not differentiable, and the approach in *SALT* is just pragmatic: *Nobody is going to hit that corner.*

1.3.5 Stack-size

The initially presented example:

```
from salt import Leaf, sin
y = x = Leaf(3.14159)
for _ in xrange(100):
    y = sin(y) * y
```

is nice, but what happens if you increase the *xrange* argument to 1000? Most likely, there will be some error messages about maximum recursion depth. For most actual applications, this should not pose any problem. Hence if it happens, consider first whether the way your implementation works is as intended.

If really necessary, do this:

```
from sys import setrecursionlimit
setrecursionlimit(2000) # or whatever you need
```

1.4 API documentation

1.4.1 SALT datatypes

Salt

class salt.datatype.**Salt**(*node*)

Most of the user objects to deal with in *SALT* are instance of this class, representing a Salt entity. The operators and most unary functions from the *math* module are emulated. With that, as intended, most *scalar* mathamatics implementations are doable.

Technically, the :py:obj:*Salt* class is only a smart pointer to the underlying node object. Consequently, coding:

```
a = Leaf(2.1)
b = a + 0
```

will simplify the sum and let *b* and *a* point to the same leaf node with value 2.1.

Objects of type *Salt* are normally only created via mathematical operations on other instances of the same type, whereas the start is made by the sub-class *Leaf*

ALLOW_MIX_FLOAT = True

Normally, it is convenient to be allowed writing:

```
E = 0.5 * m * sq(v)
```

Still, it is easy to imagine to code bugs by forgetting to instantiate important input variables as *Leaf* objects. By setting this attribute to `False`, operators with mixed `float`- *Salt* datatypes are disallowed **unless the constant is already cached**.

This implies that 0, 1 and 2 are always allowed as constant contributions.

While `ALLOW_MIX_FLOAT` is `True`, newly encountered constants will be cached unless `FLOAT_CACHE_MAX` is exceeded.

Type bool

FLOAT_CACHE_MAX = 100

Newly encountered constants are cached if `ALLOW_MIX_FLOAT` is set to `True`, but to prevent uncontrolled growths in memory, caching is stopped after the given number of entries.

Type int

invalidate()

Between two queries for values, when also independent variables have changed their value, invalidate has to be called to trigger a new evaluation. The method can be called before, during or after the independent variables are actually changed - with the same effect.

Returns None

plain()

Creates a new symbol of the same value, but not propagating dependencies and derivatives:

```
a = Leaf(1.0)
b = exp(a)

b_plain = b.plain()

c = Derivative([b, b_plain], [a])
```

The symbols in `c` will now keep the values 2.718... and zero.

Returns The plain symbol without derivative tracing

Return type *Salt*

recalc()
Force the evaluation of this symbol.

The normal purpose of *SALT* is to treat large quantities of dependent and independent symbols by following the cyclus of invalidating, setting values, and getting values. However, if you suddenly really need to know the value of a variable out of this scheme, use recalc.

If you don't know what you are doing, the only way to obtain the correct value from a node is to query the value (dump it), invalidate, and reevaluate, this time keeping the result. This is what `recalc` does.

Returns The symbols value

Return type float

select(*switch*)
This method implements a primitive conditional. The call:

```
y = x.select(a)
```

is equivalent to the float operation:

```
y = x if a > 0 else 0
```

Parameters **switch** (*Salt*) – The decission variable

Returns The Salt equivalent to *y* in above *if* construct

Return type *Salt*

value
Value is a property that is read-only except for instances of the *Leaf* subclass. Requesting this property returns its numerical (float) value, if necessary after re-evaluating the underlying Salt graph - or parts of it.

Type float

Leaf

class salt.datatype.**Leaf**(*value=0.0*)
Bases: *salt.datatype.Salt*

This class is the starting point to build up a Salt algebra graph.

"*In the beginning, there was a leaf!*" – Caterpillar's bible

Leafs are the only instances of *Salt* that support a read-write *value* attribute. It is per definition independent of any other symbols.

__init__(*value=0.0*)
Constructor to instantiate a *Leaf* object from a *float* value.

Parameters **value** (*float*) – The initial numerical value of the node

value
Same property as defined in base class *Salt*, but writable. Setting this property has no immediate side effects. In particular, dependent nodes do **not** get notified to re-evaluate automatically. For performance

reasons, *Salt.invaludate* needs to be called on the dependent variables in order to trigger reevaluation.

Type float

1.4.2 Tools

SaltArray

class salt.tools.**SaltArray**(*source=None*)

This class represents an array specialised for symbols in it.

You may instantiate this list with anything in it, but for the specific methods to work, the containing datatypes better are other containers or objects of type *Salt*. Derivatives are represented as SaltArray objects. Some slicing functionality is included, and indexing supports multi-dimensional lists. Furthermore the objects are iteratable.

__init__(*source=None*)

Constructor of an empty object or one based on the given source.

Parameters **source** (Iterable (nested) container of *Salt*) – The symbols to be treated as a collection. Containers can be nested and inhomogenious, as long as they are either iterable or of type *Salt*. If source is not provided (or None), the object is initialised as an empty container.

append(*data*)

Same method as the corresponding one for list objects.

extend(*data*)

Same method as the corresponding one for list objects.

invalidate()

Same as *Salt.invalidate*, just applied to the entire container, therefore slightly more efficient :return: None

static invalidate_container(*container*)

The static version of *invalidate*, can be applied to any (nested) container

recalc()

Same as *Salt.recalc*, just applied to the entire container, therefore slightly more efficient

Returns The values of the symbols within the container

Return type <list <list ...<float>...>

static recalc_container(*container*)

The static version of *recalc*, can be applied to any (nested) container

value

Same as *Salt.value*, just applied to the entire container and returning a nested container of same shape as original, containing the float values

Returns The values of the symbols within the container

Return type <list <list ...<float>...>

static value_container(*container*)

The static version of *value*, can be applied to any (nested) container

Derivative

class salt.tools.**Derivative**(*dependent*, *independent*)
Bases: *salt.tools.SaltArray*

This class could have been implemented as a function, as it consciously behaves like one. However, the cleanest way to encapsulate it's (private) content is probably to define it as a class, prepresenting its own result.

When deriving (right under construction), the dependent variables and their underlying graph are first analysed in order to avoid chewing on derivatives that are anyway zero. Then, the graph is again traversed recursively in order to obtain and pre-simplify the derivatives with respect to the independent variables.

In fear of performance issues, the final more rigorous simplification is not included here, but might be in the future.

__init__(*dependent*, *independent*)
Construct the result object as the symbolic derivative $\mathrm{d}y/\mathrm{d}x$.

Parameters

- **dependent** (Iterable container of *Salt*) – The variables *y* to derive
- **independent** (Iterable container of *Salt*) – The variables *x* to derive with to

dump

salt.tools.**dump**(*symbols*, *scope=None*)
This class dumps valid python code that defines the given symbols. The code is always generated down to the leaf nodes.

A list of string representation of the symbolic graph. Here, the entire graph below *symbols* is processed down to the *Leaf* nodes. A scope can be provided as an argument, providing the algorithm with names of user-known variables. The following example:

```
a, b, c, d, e = map(Leaf, range(5))
f = (a + b) * c
g = b * b
h = d * e
i = h + f
scope = {"a": a, "b": b, "c": c, "d": d, "e": e,
         "f": f, "g": g, "h": h, "result": i}
print "\n".join(dump([f,g,h,i], scope))
```

will produce the following output:

```
var_1 = a + b
f = var_1 * c
g = b ** 2
h = d * e
result = h + f
```

Note that `var_1` is no variable known at user scope, but an internal node. It will therefore be given a generic name, as all variables that are not member of `scope`. Let us emphasise that *SALT* itself does not hold any symbol names in the nodes. When dumping the graph, the user is free to call them then and there by his/her favourite pet.

Actually, if the dumped code is to be used to be executed (somewhere else) later, you might want to utilise some variable groups as lists. If above code is to be a function with `[a,b,c,d,e]` as argument `x`, define scope as such:

```
scope = dict("(x[%d]" % i, var) for i, var in enumerate((a, b, c, d, e))}
scope.update(f=f, g=g, h=h, result=i)
```

Parameters

- **symbols** (Iterable 1D container of *Salt*) – The symbols for which the graph shall be dumped
- **scope** (dict(string, *Salt*)) – A dictionary to map variable names to known symbols

Returns A list of strings, each of them representing an assignment with one operator or function (representing one symbolic node)

Return type list<string>

Note that multiple variables can point to the same node, hence *SALT* cannot even distinguish them. If *scope* provides multiple variables representing the same symbol, an arbitrary name will be selected for generating the string representation.

simplify

salt.tools.**simplify**(*symbols*)

This function simplifies the given symnbol or container of symbols **in-place**.

In the current implementation, it applies the same simplifications as when creating the graph, but simultaneously removes duplicates. That is:

```
a, b = Leaf(3.14159), Leaf(2.71828)
c = (a + b) + sin(a + b)
simplify(c)
```

will simplify to:

```
x = a + b
c = x + sin(x)
```

This kind of simplification has a great impact on automatically generated derivatives, as the chain rule leaves a lot of common terms for the derivatives of different independent variables. Not all of them can be avoided while generating the derivatives.

Naturally, duplicate nodes (that is: same type and same child nodes) can only be detected if they are under the symbolic graph reachable by the given symbols:

```
c = sin(a+b)
d = cos(a+b)
simplify(c)
```

This code would not be able to detect existance of $a + b$ as duplicate somewhere else in the graph - another consequence of avoiding bidirectional linking, sorry - not.

Parameters **symbols** (Iterable container of *Salt*) – A single symbol or a container of symbols to be simplified. Containers can be nested and inhomogenious, as long as they are either iterable or of type *Salt*

Returns Number of duplicates found

Return type int

Empanada and Empanadiña

salt.tools.**empanada**(*func*, *inp*, *dim_out=1*)
This function is described *here*.

Parameters

- **func** (*f: list<float> -> (list<float>, list<list<float>>)*) – The function to be embedded, taking a list of input variables as argument - to be consistent with `inp`, and returning a list of values with its dimensionality given by `dim_out`, as well as the Jacobian J as the derivative matrix of output variables with respect to input variables.
- **inp** (list<*Salt*>) – The list of input symbols that will be linked to the function input arguments

Returns A list of symbols linked to the return values of `func`, with the first derivative being represented by J

Return type list<*Salt*>

salt.tools.**empanadina**(*func*, *inp*)
This function is described *here* and is very similar to `empanada`, just reduced for scalar usage.

Parameters

- **func** (*f: float -> (float, float)*) – The function to be embedded, taking the input variable as argument, and returning the function value and the derivative of it with respect to the input variable.
- **inp** (*Salt*) – The input symbol that will be linked to the function input argument

Returns The symbol linked to the return value of `func`, with the first derivative being represented by J

Return type *Salt*

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

Symbols

A

D

E

F

I

L

P

R

S

V