Contents

Title: Quasar - Algorithmic Differentiation manual

Introduction

Algorithmic differentiation (AD, also known as *automatic differentiation*) refers to the ability of automatically deriving an algorithm that computes a (partial) derivative of a given function that is also specified by an algorithm. Therefore, AD is a transformation in the space of algorithms. AD has several advantages, such as not having to deal with the numerical inaccuracies of numerical differentiation ($\frac{\partial f}{\partial x} \approx f(x+1) - f(x)$) and the expansive expressions of symbolic differentiations. AD consists of representing the input algorithm as a chain of input-output operations to which the chain rule can be applied. As such, AD can handle control structures such as loops and branches (at least, as long as the involved conditions do not depend on the derivative variables). Suppose we have three operations f, g and h, then the input algorithm is represented by the function compositions $y = f(g(h(x)))$. Applying the chain rule yields

$$\frac{dy}{dx} = \frac{dy}{dw_2}\frac{dw_2}{dw_1}\frac{dw_1}{dx}$$

The main advantage of AD is there is no longer a need to manually write functions that implement the derivative of a cost function (which is often a tedious and error-prone process, especially for higher-order functions). Additionally, the AD framework can leverage on the parallelization facilities of the compiler.

There are two modes of AD: *forward accumulation*, in which the chain rule is traversed from inside to outside (just like when calculating the derivative symbolically):

$$\frac{dw_i}{dx} = \frac{dw_i}{dw_{i-1}}\frac{dw_{i-1}}{dx}, i = 1, 2, 3 \quad \text{with} \quad w_3 = y$$

and *backward accumulation* (also called reverse accumulation) which traverses the chain rule from outside to inside:

$$\frac{dy}{dw_i} = \frac{dy}{dw_{i+1}}\frac{dw_{i+1}}{dw}, i = 0, 1, 2 \quad \text{with} \quad w_0 = x$$

Both modes give the same result but they have different characteristics when applied to multivariate functions versus vector functions. In particular, the application to multivariate functions is generally more efficient using backward differentiation while forward differentiation is more suited for univariate functions or vector-valued functions.

In backward accumulation, intermediate values of the forward computation tend to be out-of-scope at the moment that the backward accumulation is applied. Therefore it is necessary to store these intermediate values in memory. In sequential backward AD, intermediate values are stored in a so-called tape drive, which is "played back" in reverse during the differentiation. Quasar implements parallelized versions of forward AD and backward AD: * first, the algorithm is analyzed and the dependencies between the variables are being determined * next, forward or backward AD is applied to obtain a new algorithm * the resulting algorithm is parallelized and further optimized using target-specific code transformations

With this approach, the parallelism inherent in the initial algorithm is also present in the derived algorithm. The tape drive is implemented by defining extra variables that store the intermediate value(s), which can be vectors, matrices or higher dimensional arrays. This is in contrast to *symbolic* differentiation, for which the function to be derived is expressed in one expression, without using any intermediate values. Symbolic differentiation can therefore lead to expansive expressions.

Additionally, *automatic adjoint calculation* is similar to backward differentiation, with the main difference that an algorithm is derived to compute the adjoint of a linear operation. Linear operations can be expressed as matrix-vector product (e.g.,

$\mathbf{y} = \mathbf{A}\mathbf{x}$) and the adjoint operation is then simply $\mathbf{x} = \mathbf{A}^T\mathbf{y}$ (or $\mathbf{x} = \mathbf{A}^H\mathbf{y}$ in cause of complex-valued data). Here an algorithm is specified to calculate $\mathbf{A}$, so that the matrix itself no longer needs to be densely stored in the computer memory. In fact, this easily allows to build matrices of millions of rows by millions of columns. Once the adjoint of the algorithm is determined, sparse matrix solvers such as the method of conjugate gradients can be used to invert $\mathbf{A}$.

An illustrative example

To illustrate the different differentiation modes, consider symbolic differentiation for the product $a_1a_2a_3$:

$$\frac{\partial}{\partial x}(a_1a_2a_3) = \frac{\partial a_1}{\partial x}a_2a_3 + a_1\frac{\partial a_2}{\partial x}a_2a_3 + a_1a_2\frac{\partial a_3}{\partial x}$$

The resulting expression has three terms with each three factors (in total 6 multiplications and 2 additions). For multiplication of n factors, the resulting number of factors becomes quadratic after derivation (n^2).

Forward mode AD avoids this problem by introducing temporary variables to store the intermediate results of the calculation. In forward mode:

$$t_1 = a_1a_2, t_2 = t_1a_3$$
$$\frac{\partial t_1}{\partial x} = \frac{\partial a_1}{\partial x}a_2 + a_1\frac{\partial a_2}{\partial x}$$
$$\frac{\partial t_2}{\partial x} = \frac{\partial t_1}{\partial x}a_3 + t_1\frac{\partial a_3}{\partial x}$$

which results in 5 multiplications and 2 additions (saves one multiplication). However, the bigger the expression is, the more can be gained by reusing computations such as in the above example.

Backward mode AD splits up the calculations as follows (here $\dot{x}$ denotes a derivative with respect to the output variable, t_2):

$$t_1 = a_1a_2, t_2 = t_1a_3 \quad \text{(forward step)}$$

$$\dot{t_2} = 1 \quad \text{(seed value)}$$

$$\dot{t_1} = a_3\dot{t_2} \quad \text{(reverse step 1)}$$

$$\dot{x} = t_1\dot{t_2}\frac{\partial a_3}{\partial x}a_3 + \dot{t_1}a_2\frac{\partial a_1}{\partial x} + \dot{t_1}a_1\frac{\partial a_2}{\partial x} \quad \text{(reverse step 2)}$$

The backward mode AD first performs the forward calculation to obtain $a_1a_2a_3$ then calculates the derivatives of the temporary variables with respect to t_2 in a reversed manner. The variable $\dot{x}$ then holds the desired result. In this example, the backward mode AD brings little advantage in computing $\frac{\partial}{\partial x}(a_1a_2a_3)$. However, suppose that $\mathbf{x}$ is a vector, then through $\dot{\mathbf{x}}$ we directly obtain a vector containing all of the partial derivatives. The forward mode analyzes how the output variables change as function of the input variables, while the backward mode works the other way around, making it more suitable to be applied when the output is scalar (or a vector of a small dimension) and when the input is a large vector. As in the above example, the vector with the partial derivative is obtained at once.

For the backward mode to work, intermediate values t_1 and t_2 need to be stored for later use. During the forward calculation, it may occur that variables are overwritten (for example, due to inplace operations such as +=, *=, ...). When this happens, previous values need to be stored for later retrieval. In sequential AD algorithms, this is done using a linear array called a

tape drive, which can during the reverse step be played in reverse to obtain the right intermediate values at the right time. For parallel AD implementations, the parallelism is inherent in the output vector (i.e., the derivative can be calculated w.r.t. each output variable in parallel), although this requires special care because a fully sequential tape drive can no longer be used.

Algorithmic differentiation in Quasar

A Quasar function has K inputs, of which each input can be a scalar value, a vector/matrix/higher dimensional array, a cell array or even an object. The function has L outputs which can have a scalar type, a vector type etc. To simplify the AD framework, Quasar functions are restricted to have a single output parameter, without loss of generality (the output parameter can still be a cell vector, which is equivalent to returning each component individually).

For convenience, it is also possible to calculate the derivative with respect to groups of parameters (for example, the derivative with respect to a variable of type vec[mat](2), which represents a vector containing two matrices). Such derivatives needs to be interpreted as a derivative with respect to one vector that contains all the individual components of the two matrices. For example, the cost function of a neural network may contain several sets of parameters (e.g., one set for every layer). The parameter grouping (either using cell vectors/matrices or using classes) allows the derivative of the cost function to be calculated with respect to all parameters of interest. We can therefore say that the function $\mathbf{f}$ constitutes a mapping from $\mathbb{R}^M \rightarrow \mathbb{R}^N$ (or in case of complex-valued functions, $\mathbb{C}^M \rightarrow \mathbb{C}^N$). *Summarizing: Mathemetically, we are working with vectors, while in the programming language we are using the available data structures to represent the data in an organized manner that is convenient to work with.*

In the AD framework, there are different building blocks:

1. The derivative may be calculated with respect to a scalar value $r_i \in \mathbb{R}$. Therefore, the derivative $\frac{\partial \mathbf{f}}{\partial r_i}(\mathbf{x})$ is a mapping from $\mathbb{R}^M \rightarrow \mathbb{R}^N$ (i.e., the derivative of each of the scalar output variables with respect to *one* of the input variables, evaluated at a certain point in the input space). We will call this *standard method*.
2. The derivative may be calculated with respect to a vector $\mathbf{s} \in \mathbb{R}^J$ with $J \leq M$. In this case, the derivative $\left[\frac{\partial \mathbf{f}}{\partial s_1}(\mathbf{x}) \cdots \frac{\partial \mathbf{f}}{\partial s_J}(\mathbf{x})\right]$ (also called Jacobi matrix) is a mapping from $\mathbb{R}^M \rightarrow \mathbb{R}^{MN}$ (the derivative of each of the scalar output variables to *each* of the input variables). The dimensionality easily becomes intractible, especially of the input and output space already have high dimensionality (such as for images). Luckily, for many optimization algorithms, the Jacobi matrix does not need to be evaluated and stored in memory, instead many algorithms use the product of the transposed Jacobi matrix and a vector $\left[\frac{\partial \mathbf{f}}{\partial s_1}(\mathbf{x}) \cdots \frac{\partial \mathbf{f}}{\partial s_J}(\mathbf{x})\right]^T \cdot \mathbf{w}$ (also called *tangent method*).
3. Furthermore, in case $\mathbf{f}$ is a linear function, the adjoint $\mathbf{f}^\star$ is of interest (called *adjoint method*). When the framework can determine that a function is linear, the adjoint may be used instead of the derivative, leading to a slightly more efficient implementation (in particular, the AD framework uses the following relationship: $\frac{\partial}{\partial \mathbf{x}}(\mathbf{A}\mathbf{x}, \mathbf{x}) = \mathbf{A}^T\mathbf{x}$.

Using these building blocks, it is possible to define and calculate higher order derivatives (e.g., Hessian). The algorithmic differentiation library in Quasar (Quasar.CompMath.dll) defines the following meta functions:

Meta-function	Method
$diff(f(x), x)	Standard (automatic mode)
$diff(f(x), x, "forward")	Standard (forward mode)
$diff(f(x), x, "backward")	Standard (backward mode)
$diffmul(f(x), x, w)	Tangent
$adjoint(f(x), x)	Adjoint
$hessian(f(x), x)	Hessian of a scalar function
$islinear(f(x), x)	Returns true if f is a linear function in x, otherwise false
$isaffine(f(x),x)	Returns true if f is an affine function in x, otherwise false
$unapply(f(x), x)	Converts a function call to a lambda expression $x \rightarrow f(x)$

These functions are evaluated at compile-time and automatically generate code which can be inspected using the code workbench window in Redshift. The result of the evaluation can be used as if it was a regular function (i.e., evaluated or stored in a variable).

For derivatives of functions with respect to scalar values, *forward mode AD* is used, while for derivatives with respect to vectors or matrices, *backward mode AD* is used. It is known that for vector-valued functions, the optimal calculation of the derivative is best done using a combination of both forward and backward mode AD. Being an NP-hard problem, this is currently not (yet) supported and the backward mode is used instead.

Important is that the argument is specified each time (e.g., `$diff(f(x), x)` and not `$diff(f, x)`. This may cause a problem when `x` is a variable that is non-existing in the outer context. Here the solution is to define the derivative within a lambda function `(x : cube) -> $diff(f(x), x)`.

The function `$unapply(f(x), x)` performs the same as the explicit lambda expression `x -> f(x)`, however, there is a crucial difference: `$unapply()` takes over the type of `x` in the local context. This allows differentiation to proceed on the new lambda expression. It is therefore common to see expressions like:

```
f_deriv = $unapply($diff(f(x), x), x)
```

To enable efficient code generation, the AD compiler requires the parameter types of the functions being differentiated, to be known *at compile-time*. An error will be generated is a parameter type is not specified.

The functions `$islinear` and `$isaffine` symbolically check linearity and affinity in a rather naive way. The check may give a false negative result (for example, due to the use of mathematical identities unknown to the compiler). The functions are mostly useful for performance optimization purposes (in which the result of the meta-function only affects the computation time).

Example: Total Variation minimization

The total variation cost function can be calculated as follows:

```
function C = total_variation_costfunc(g : mat, b : mat, lambda : scalar)
    C = 0.0
    for m=0..size(g,0)-1
        for n=0..size(g,1)-1
            C += (g[m,n] - b[m,n]).^2 % data fidelity
            C += lambda*abs(g[m,n-1] - g[m,n]) % horizontal partial derivative
            C += lambda*abs(g[m-1,n] - g[m,n]) % vertical partial derivative
        endfor
    endfor
endfunction
```

Here, we use simple loops without vector or matrix expressions, because then the AD framework can easily handle the function. For parallelization, we rely on the automatic loop parallelizer (with parallel reduction recognition).

Calculating the derivative w.r.t. `g` using the *forward mode* results in:


```
> $diff(total_variation_costfunc(g,b,lambda),g,"forward")

function C_deriv = total_variation_costfunc_deriv(g : mat, b : mat, lambda : scalar)
    C_deriv = 0.0
    for M=0..size(g,0)-1
        for N=0..size(g,1)-1
            for m=0..size(g,0)-1
                for n=0..size(g,1)-1
                    C_deriv += 2*(g[m,n] - b[m,n]) * delta(m-M) * delta(n-N)
                    C += lambda*sign(g[m,n-1] - g[m,n]) * delta(m-M) * (delta(n-1-N)-delta(n-N))
                    C += lambda*sign(g[m-1,n] - g[m,n]) * delta(n-N) * (delta(m-1-M)-delta(n-1-N))
                endfor
            endfor
        endfor
    endfor
endfunction
```

Note that the derivative calculation now contains 4 loops instead of 2, and the inner loops contain a lot of Dirac delta functions (which only give a non-zero response when their argument is zero). The number of loops can be reduced by using code analysis techniques, however this is not trivial and not guaranteed to work under all circumstances, resulting in an algorithm of complexity $\mathcal{O}(N^4)$. Instead, the *backward mode* derivative directly gives a calculation procedure employing only two loops (with complexity $\mathcal{O}(N^2)$, the same as the function that is differentiated):

```
> $diff(total_variation_costfunc(g,b,lambda),g,"backward")

function g_deriv:mat=total_variation_costfunc_deriv(g:mat,b:mat,lambda:scalar)
    g_deriv=zeros(size(g))
    for m=0..size(g,0)-1
        for n=0..size(g,1)-1
            g_deriv[(m-1),n]+= sign((g[(m-1),n]-g[m,n])) * lambda
            g_deriv[m,n] += -sign((g[(m-1),n]-g[m,n])) * lambda
            g_deriv[m,(n-1)]+= sign((g[m,(n-1)]-g[m,n])) * lambda
            g_deriv[m,n] += -sign((g[m,(n-1)]-g[m,n])) * lambda
            g_deriv[m,n] += g[m,n] * 2 - b[m,n] * 2
        endfor
    endfor
endfunction
```

In general, by *backward mode AD* the algorithmic complexity of the input function is preserved (up to constant factors). The resulting loops can also easily be parallelized. In Quasar, the addition += is implemented using an atomic update, the parallel loops are therefore free of data races. After AD, the generated function is further compiled using the Quasar compiler, not only performing automatic parallelization, but also allowing target-specific optimizations to be added. For example, the atomic operation may be performed in shared memory instead of global memory, significantly improving the performance of the generated kernel. Additionally, atomic operations on scalar variables or matrices with constant indices are automatically translated into parallel reduction algorithms.

AD framework overview

In the next sections we give an overview of the features supported by the AD framework.

Type restrictions

It is important that all parameter types (except return parameter types) of functions to be differentiated are statically typed. Without type information, the AD framework cannot know whether a variable is a vector, a matrix or a scalar value.

Currently, the derivatives to be calculated need to be *scalar* values or vectors of *scalar* values. Calculating the derivative with respect to an integer value is not allowed (in this case numerical differentiation is recommended).

Complex-valued scalar values (cscalar) can be used but have not been tested thoroughly (it is best to check if the generated derivative functions are correct).

It is also possible to take derivatives with respect to cell vectors/matrices (grouping individual parameters) or with respect to objects. For example

```
type Parameters : dynamic class
    w1 : cube{4}
    w2 : mat
    w3 : cube
endtype

function C = cost_function(p : Parameters)
    ...
endfunction
```

This way, the derivative of cost_function with respect to all parameters can be obtained:

```
cost_function_deriv = (p : Parameters) -> $diff(cost_function(p), p)
```

Then cost_function_deriv can be used as a regular Quasar function and will calculate the gradient of cost_function in point p.

Suppose we are using a block-coordinate descent method in which we want to calculate a partial derivative (e.g., with respect to w1). In this case, it is best to define a function to split the parameters:

```
function C = cost_function_split(w1 : cube{4}, w2 : mat, w3 : cube)
    cost_function_split = cost_function(Parameters(w1:=w1, w2:=w2, w3:=w3))
endfunction

cost_function_deriv_w1 = (p : Parameters) -> $diff(cost_function_split(p.w1,p.w2,p.w3), p.w1)
```

Then a gradient step on the w1 parameter can easily be calculated:

```
p.w1 += step_size * cost_function_deriv_w1
```

Modularity and device functions

The AD framework supports *modular programming* and it is encouraged to use this approach such that the functionality can be separated in small functions that can potentially be reused. For example, a *loss function* could be defined as follows:

```
function C = lossfunction(u : Data, y : Data, f : Parameters)
    x = degradation_function(y, f)
    C = mse_lossfunction(u, x)
endfunction
```

When determining the derivative of lossfunction, the AD framework will also differentiate the functions degradation_function and mse_lossfunction automatically. This is a result of applying the chain rule recursively. Essentially, every function in the call graph will be differentiated, up to arbitrary nesting level. When a function occurs multiple times in the call graph, this function will only be differentiated once: the result will be "remembered" by the framework. Correspondingly, the result of derivation or adjoint calculation is a function that has again a call graph with (generally) the same depth. This permits further high-level optimizations by the Quasar compiler (kernel merging, inlining, loop fusion etc.).

Important to remark is that the metafunctions diff, diffmul and adjoint preserve the __device__ function modifier of the input function. This means that when the original (non-differentiated) function is declared as a __device__ function, the derivated/adjoint functions are also a __device__ function. When used inside a loop, this permits parallelization of the loop.

For example, consider the pointwise RELU function:

```
function y = __device__ RELU(x : scalar)
    y = max(x, 0)
endfunction

function y = pointwise_RELU(x : cube{4})
    y = uninit(size(x))
    for m=0..size(x,0)-1
        for n=0..size(x,1)-1
            for p=0..size(x,2)-1
                for q=0..size(x,3)-1
                    y[m,n,p,q] = RELU(x[m,n,p,q])
                endfor
            endfor
        endfor
    endfor
endfunction
```

The first declaration defines the RELU function in a pointwise manner. The pointwise_RELU function then applies the function to each element of a 4D cube. When calculating $diff (pointwise_RELU(x),x), the AD framework will also derive the derivative of RELU. Because RELU is a device function, its derivative will also be a device function, so that the 4D for-loop can be parallelized (and eventually executed in parallel on a multi-core CPU or GPU).

Control structures (loop, while, repeat, if)

One of the advantages of AD is that control structures can easily be handled. AD therefore nicely integrates in imperative programming languages. One restriction is that the control structure conditions should not depend on the active variables

(i.e., the variables which depend on the derivative variable).

For example, calculating the derivative of:

```
function C = total_variation_costfunc(g : mat, b : mat, lambda : scalar)
    C = 0.0
    for m=0..size(g,0)-1
        for n=0..size(g,1)-1
            if abs(g[m,n] - b[m,n]) > 20
                C += (g[m,n] - b[m,n]).^2 % data fidelity
                C += lambda*abs(g[m,n-1] - g[m,n]) % horizontal partial derivative
                C += lambda*abs(g[m-1,n] - g[m,n]) % vertical partial derivative
            endif
        endfor
    endfor
endfunction
```

with respect to g will not work due to the if condition $\text{abs}(\text{g[m,n]} - \text{b[m,n]}) > 20$ being dependent on the derivative g (moreover, the condition is intrinsically non-differentiable). This issue is known as the if-problem in algorithmic differentiation. A solution might be to replace the if with the multiplication by a weight calculated using a smooth function:

```
compare = x -> 1 / (1 + exp(-x)) % logistic function

function C = total_variation_costfunc(g : mat, b : mat, lambda : scalar)
    C = 0.0
    for m=0..size(g,0)-1
        for n=0..size(g,1)-1
            weight = compare(abs(g[m,n] - b[m,n]) - 20)
            C += weight * (g[m,n] - b[m,n]).^2 % data fidelity
            C += weight * lambda*abs(g[m,n-1] - g[m,n]) % horizontal partial derivative
            C += weight * lambda*abs(g[m-1,n] - g[m,n]) % vertical partial derivative
        endfor
    endfor
endfunction
```

This approach guarantees that the obtained gradient is meaningful (i.e. does not suffer from the so-called vanishing gradient problem).

Concerning loops, one restriction is that break and continue statements are currently not supported, due to difficulty in reversing the resulting control flow.

Nested functions

Derivation of functions containing inner functions is currently not supported. A possible solution is to place the inner function within the same scope as the parent function.

However, taking derivatives of inner functions is permitted! This way, it is for example to implement an optimization algorithm with gradient calculation, iterative update steps entirely within one single function.

In Quasar, functions are first-class citizens allowing functions to be passed as parameters to other functions, or allowing functions to return function variables. Within the context of AD, this is only permitted when the compiler has full information on the function. Logically, the compiler should be able to access the source code of the function. This is not possible

in case the function is part of a conditional expression such as `fn = x -> x < 0 ? func1 : func2`. When the compiler cannot access a function the compiler will

1. generate a warning indicating that the derivative of this function could not be determined automatically
2. when possible, mark the parent function as generic, so that in a later compilation step, the function parameter can be determined at call-time (i.e., when the generic function is being called itself with concrete values).

This mechanism allows to write generic optimization methods which depend on function parameters. The algorithmic differentiation then takes place at the moment that the optimization function is called with a concrete cost function. The example section contains a few samples of this technique.

Built-in differentiation rules

For correct functioning, a number of differentiation rules are built-in into Quasar.CompMath.dll. This section gives an overview of these differentiation rules.

Reduction	Result	Description
$diff(x[m,n], x[k,l])	delta(m−k,n−l)	Dirac delta (forward mode only)
$diff(x[m,n,p], x[k,l,j])	delta(m−k,n−l,p−j)	Dirac delta (forward mode only)
$diff(f(a + b *x),x)	$diff(f(x), x)(a + b * x)*b	Chain rule
$diff(f(g(x)),x)	$diff(f(x), x)(g(x)) * $diff(g(x), x)	Chain rule

Additionally, standard differentiation rules for the operators +,−,.*,*,./,/,^ and .^ are integrated.

Several mathematical functions have their derivatives already defined. These functions are defined using reductions in Quasar:

Expression	Replacement pattern
x -> $diff(abs(x),x)	sign(x)
x -> $diff(abs(x),x)	sign(x)
x -> $diff(exp(x),x)	exp(x)
x -> $diff(exp2(x),x)	exp2(x) .* log(2)
x -> $diff(log(x),x)	1./x
x -> $diff(log2(x),x)	1./(x*log(2))
x -> $diff(log10(x),x)	1./(x*log(10))
x -> $diff(sin(x),x)	cos(x)
x -> $diff(cos(x),x)	−sin(x)
x -> $diff(tan(x),x)	1 + tan(x).^2
x -> $diff(sinh(x),x)	cosh(x)
x -> $diff(cosh(x),x)	sinh(x)
x -> $diff(tanh(x),x)	1−tanh(x).^2
x -> $diff(asin(x),x)	rsqrt(1−x.^2)
x -> $diff(acos(x),x)	rsqrt(1−x.^2)
x -> $diff(atan(x),x)	1./(1+x.^2)
x -> $diff(asinh(x),x)	1./sqrt(1+x.^2)
x -> $diff(acosh(x),x)	1./sqrt(x.^2−1)

Expression	Replacement pattern
x −> $diff(atanh(x),x)	1./(1−x.^2)
x −> $diff(sqrt(x),x)	0.5.* rsqrt(x)
x −> $diff(rsqrt(x),x)	−0.5*x^(−1.5)
x −> $diff(erf(x),x)	2*exp(−x^2).* rsqrt(x)
x −> $diff(erfc(x),x)	−2*exp(−x^2).*rsqrt(x)
(a, x) −> $diff(a.^x,x)	log(a) .* a.^x
x −> $diff(pow(a,x),x)	log(a) .* a.^x
(x, n:int) −> $diff(x^n,x)	n*x^(n−1)
(x, y) −> $diff(max(x,y),x)	x>y?1:0
(x, y) −> $diff(max(x,y),y)	x>y?0:1
(x, y) −> $diff(min(x,y),x)	x<y?1:0
(x, y) −> $diff(min(x,y),y)	x<y?0:1

Some of these definitions rely on relaxed differentiability conditions: the definitions may ignore that the function is non-differentiable in a certain subset of their domain. In this case, it is common to either take the left derivative in each non-differentiable point, the right derivative or an average of them. For example, the abs(x) function is non-differentiable in x=0, nevertheless, the value of the sign function 0 is assigned to the derivative.

Defining custom reductions

In some cases, the AD framework cannot determine the derivative (or adjoint) of a function automatically. Then the following error will be reported:

```
Could not determine ‘$diff(f(x), x)‘. Consider defining a reduction to specify this function.
```

As suggested by the above error message, it is possible to manually specify derivatives of functions through reductions. An example for the sum function is given below:

```
reduction (x : cube{:}) -> $diff(sum(x), x) = 1
```

The derivative of sum(x) in x is one because each component of x occurs in sum(x) with coefficient 1. This is an example of a multivariate function for which the derivative is currently not built-in (note that instead another mechanism exist to handle multivariate functions, see Derivatives involving multivariate functions).

There may also be other reasons for defining custom derivatives: for example, to ensure that the derivative code is numerically stable. An example is the softmax function, which calculation depends on the exponential function and is prone to over/underflow. These numerical problems can be solved using various methods (e.g., a scaling trick, logarithmic domain calculations, ...), however the AD framework has currently no means of determining that derivative code may become numerically instable. In this situation, it is best to define the derivate *manually*.

When defining custom derivatives, it is important that the domain (in the above example x : cube {:}) is well-defined, so that the AD framework knowns under which conditions the reduction can be applied. For example, it is possible that a reduction is defined for a realvalued domain, while a separate reduction is defined for a complexvalued domain. The AD framework

then selects the best matching reduction, according to a reduction selection strategy (see Quasar's quick reference manual for more information).

Reductions can also have additional where clauses which express extra conditions to be checked when applying the reduction. If the where clause is evaluated at compile-time to be false, the differentiation rule is not applied. Reductions defining differentiation rules follow the Quasar reduction rules and therefore do not differ from 'ordinary' reductions. The reductions can also be prioritized, by adding the priority =high attribute:

```
reduction {priority=high}, (x,...f) -> $diff(sum([f(x)]),x) = sum_diff(f, x)
```

Prioritized reductions can override the builtin reductions from the previous section. Note that applying $diff, $diffmul and $adjoint will generate a reduction with *normal* priority. This reduction allows the AD framework to remember derivatives that have been generated earlier on.

When defining a custom differentiation rule for a multi-variate function, e.g., in a library of specialized functions, it is best to define the two main 'building blocks': $diffmul and $adjoint (of the derived function). This guarantees that the AD framework can apply the functions in general circumstances. The following table gives an overview of which definitions are required:

Univariate function	Multivariate function
$diff ()	$diffmul ()
$adjoint ()	$adjoint ()

In other words, for a multivariate vector function, the AD framework will never use the $diff () function, due to the high dimensionality (for a function with M inputs and N outputs, the result needs to be stored in a matrix of size $M \times N$).

Perhaps it is not so obvious how $diffmul and $adjoint need to be defined. For this purpose, we first consider the example of a leaky integrator:

```
function y = __device__ leaky_RELU(x : scalar, alpha : scalar)
    y = max(x, x * alpha)
endfunction

reduction (x, alpha) -> $diff(leaky_RELU(x, alpha), x) = x < 0 ? -1 : alpha
reduction (x, alpha) -> $adjoint(leaky_RELU(x, alpha), x) = leaky_RELU(x, alpha)
```

Here, __device__ ensures that the function (and its adjoint/derivatives) can be executed on accelerator devices (e.g., GPU). The leaky rely function is self-adjoint since it is a pointwise operation.

As an example of a multivariate function, we consider the logistic regression function:

```
function y : vec = logres(x : vec)
    y = exp(x - max(x))
    y = y / sum(x)
endfunction
```

In this function, a scaling is applied which relies on a multi-variate function sum. One possibility is to follow the approach from Derivatives involving multivariate functions, although here we indicate how the derivative can be specified manually.


```
function dy : scalar = logresderiv(x : vec, dx : vec)
    y = exp(x - max(x))
    h = sum(x)
    n = numel(x)
    dy = dotprod((dy * h - y .* x)/h.^2, dx)
endfunction

reduction (x, dx) -> $diffmul(logres(x), x, dx) = logresderiv(x, dx)
```

As mentioned before, the `$diffmul` function evaluates the derivative of `logres(x)` multiplied by the vector `dx`.

Index transformations and boundary extension methods

The AD framework can handle various index transformations, such as `y[k,l]=x[2*k+20,4*l-20]` and also the boundary extension methods are used in the derived code (for example, function parameters or variables declared with 'clamped, ' circular ' and 'mirror' access specifiers).

There is one restriction: the indices cannot contain active variables (i.e., variables which depend on the derivative variable). The following error will be generated:

> Auto-differentiation of function func_deriv$u: Could not determine the derivative of x[u,v] with respect to out.
> The result I get is $diff(x[u,v],out).

The reason is that the chain rule cannot be directly be applied to a discrete function such as `x[k,l]`. In these situations it is common to use a finite difference method. Although the AD framework does not offer automated support for such methods, it is still possible to calculate derivatives involving discrete functions.

A finite difference method can be implemented as follows:

```
f = (x : mat, k : scalar, l : scalar) -> x[int(k), int(l)]
f_dk = (x : mat, k : scalar, l : scalar) -> f(x, k+1, l) - f(x, k, l)
f_dl = (x : mat, k : scalar, l : scalar) -> f(x, k, l+1) - f(x, k, l)

reduction (x : mat, k : scalar, l : scalar) -> $diff(f(x, k, l), k) = f_dk
reduction (x : mat, k : scalar, l : scalar) -> $diff(f(x, k, l), l) = f_dl
```

This technique essentially tells the AD system to treat `f[k,l]` as continuous function, for which the gradient can be calculated. This gives the flexibility of defining the finite difference formula to be used (e.g., forward difference $f(x,k+1,l)-f(x,k,l)$, backward difference $f(x,k,l)-f(x,k-1,l)$ or central difference $f(x,k+1,l)-f(x,k-1,l)$.

An alternative method consists of replacing the discrete derivative by a first order Taylor series expansion: $f[k+\Delta k, l+\Delta l] = f[k,l] + \Delta k f_x(k,l) + \Delta l f_y(k,l)$ where f_x and f_y are again discrete derivatives (e.g., finite differences). With such expansion, the AD system can differentiate with respect to Δk and Δl. This approach is useful for image registration applications.

Higher order derivatives

Obtaining higher order derivatives of a univariate scalar function can be obtained by applying $diff () multiple times:

```
f_xx = $diff($diff(f(x),x),x)
```

However, this becomes more interesting when looking at higher-order derivatives of multivariate functions. The Hessian of a function $f : \mathbb{R}^N \to \mathbb{R}$ can be defined using the Jabobian of the adjoint gradient:

$$\mathbf{H}(f(\mathbf{x})) = \mathbf{J}(\nabla f(\mathbf{x}))^T$$

where the Jacobian matrix of a function $\mathbf{g} : \mathbb{R}^N \to \mathbb{R}^M$ is defined by:

$$\mathbf{J} = \begin{bmatrix} \frac{\partial \mathbf{g}}{\partial x_1} \cdots \frac{\partial \mathbf{g}}{\partial x_n} \end{bmatrix} = \begin{pmatrix} \frac{\partial g_1}{\partial x_1} & \cdots & \frac{\partial g_1}{\partial x_N} \\ \vdots & \ddots & \vdots \\ \frac{\partial g_M}{\partial x_1} & \cdots & \frac{\partial g_M}{\partial x_N} \end{pmatrix}$$

The gradient of a function f can be determined as follows:

```
gradient = $unapply($diff(cost_fn(x), x), x)
```

where the function $unapply() undoes the function parameter binding. It is similar to the lambda expression:

```
gradient = x -> $diff(cost_fn(x), x)
```

However, the difference is that the $unapply() function takes the parameter type information of x from the surrounding scope, whereas the type of x in the lambda expression is unspecified. To obtain higher-order derivatives, the type information needs to be preserved, therefore we use the $unapply() function here.

The product of the Hessian matrix and the gradient can then be calculated using the $diffmul () function:

```
hessian = $unapply($diffmul(gradient(x), x, dx), x, dx)
```

Suppose that we additionally require the adjoint of the Hessian operator. This is no problem:

```
hessian_t = $unapply($adjoint(hessian(x, dx), dx), x, dx)
```

Important to notice is that hessian and hessian_t need to be considered as operators: they evaluate the multiplication of the Hessian matrix in point $\mathbf{x}$ with an arbitrary vector $\mathbf{\Delta x}$. Here, the Hessian matrix never needs to be stored in memory; therefore approximation methods such as the diagonal of the matrix are no longer required. This approach opens up the way to implement generic Newton-Raphson optimization procedures.

If one is nonetheless interested in the entire Hessian matrix, then the generated function hessian needs to be evaluated for all N unit basis vectors $\mathbf{\Delta x}$.

Derivatives involving multi-variate functions

Implementation of backward AD mode involving multi-variate builtin functions (e.g., sum, prod(x,1)) pose several challenges: 1) the derivatives and adjoint operators need to be implemented explicitly (e.g., using reductions) and 2) the derivative of the composition of several of these functions can be calculated using the chain rule, although the resulting implementation is not necessarily computationally efficient.

Instead, the AD framework allows vector and matrix operations to be converted to loops. Once in loop form, the AD functions $diff, $diffmul, $adjoint can be applied. This can be achieved by placing the following code attribute inside the function to be derived:

```
{!ad_support convert_matrix_ops_to_loops=true}
```

The following operations are supported:

- unary/binary operations involving matrices
- vector slices a .. b and a .. b .. c
- univariate mathematical functions sin, cos, tan etc.
- aggregation operations sum(x), prod(x,1), maxdim(x,2) etc.
- reshaping functions shuffledims(x, [2,1,0]), reshape, transpose
- initialization functions eye, zeros, ones etc.
- cumulative functions cumsum(), cumprod()

The loop approach relies on the automatic loop parallelizer, parallel reduction and parallel prefix sum transforms to generate an efficient implementation for the differentiated function. The following example demonstrates how to calculate the derivative of the cumsum function.

```
import "Quasar.CompMath.dll"

function y = cumsum_p(x : mat)
    {!ad_support convert_matrix_ops_to_loops=true}
    y = cumsum(x)
endfunction

cumsum_deriv = (x,dx) -> $diffmul(cumsum_p(x),x,dx)
```

Here, the cumsum needs to be wrapped in a outer function, in order to be able to set the {!ad_support} code attribute.

Multi-GPU Algorithmic Differentiation

The AD framework also processes scheduling instructions. During the forward or backward accumulation, the scheduling instructions of the differentiated function are adjusted so that the derivative calculation is performed on the same device (CPU, GPU) as for the original function. For example, based on some auxiliary functions, a simple CNN can be defined as follows:


```
function y = forward_network(x : Data, w : NetworkParameters)
    {!sched gpu_index=0}
    part1 = pointwise_RELU(convolve(convolve_biased(x, w.w1a, w.bias), w.w2b))
    {!sched gpu_index=1}
    part2 = pointwise_RELU(convolve(convolve_biased(x, w.w1b, w.bias), w.w2b))
    {!sched mode=auto}
    y = convolve(merge_layers(part1, part2), w.w3)
endfunction
```

The differentiation for part1 will execute on GPU 0, while the differentiation for part2 will be performed on GPU 1. The code related to the differentiation of final step (with merge_layers) is subjected to automatic runtime scheduling.

Automatic vectorization

Differentiated functions follow the same vectorization rules as the underlying function. Generally, when a scalar function is vectorizable, the differentiated function is also vectorizable. By inserting

```
{!kernel_transform enable="simdprocessing"}
```

inside the for loops, the code will automatically vectorize and map onto the best suited SIMD processing implementation for the underlying device (e.g. AVX for x64, half2 for CUDA half precision calculations):

```
function C = data_fidelity(g : mat, b : mat)
    C = 0.0
    for m=0..size(g,0)-1
        for n=0..size(g,1)-1
            {!kernel_transform enable="simdprocessing"}
            C += (g[m,n] - b[m,n]).^2
        endfor
    endfor
endfunction
```


The AD compiler framework

The AD framework is in the first place designed for a loop-based modular programming style, such as in the following example:

```
function C = data_fidelity(g : mat, b : mat)
    C = 0.0
    for m=0..size(g,0)-1
        for n=0..size(g,1)-1
            C += (g[m,n] - b[m,n]).^2
        endfor
    endfor
endfunction

function C = total_variation(g : mat)
    C = 0.0
    for m=0..size(g,0)-1
        for n=0..size(g,1)-1
            C += abs(g[m,n-1] - g[m,n]) % horizontal partial derivative
            C += abs(g[m-1,n] - g[m,n]) % vertical partial derivative
        endfor
    endfor
function

function C = costfunc(g : mat, b : mat, lambda : scalar)
    C = data_fidelity(g, b) + lambda * total_variation(g)
endfunction
```

In this programming style, every operation is expressed explicitly as a calculation using scalar values and the AD framework can perform the various required operations (differentiation, adjoint calculations, reversal of operations). The multidimensional loops are further handled and optimized by the Quasar compiler infrastructure (e.g. automatic loop parallelization).

To take advantage of multivariate functions like sum, min, max, the corresponding high-level expressions can be lifted to scalar calculations (see Derivatives involving multi-variate functions). This way, these operations are all handled by the AD framework.

In this section, we give an overview of the relevant techniques that are used behind the scenes. This information is mostly useful when inspecting the code that is generated by the AD framework.

Simplified assignment form

The Quasar language allows various ways for assignments to be performed. Assignments can be nested within expressions `fn(a=1,b=2)`, multiple variable assignments can be used `[a,b]=[1,2]`, staged assignments `a=b=c` and self-updates `a=f(a)`. To be handled correctly by the AD framework, these assignments are splitted into separate statements and when necessary temporary variables are introduced:

1. Multiple assignments within one statement (such as `a=1; b=2; c=3`) are splitted into three seperate statements.
2. Multiple variable assignment (such as `[a,b]=[1,2]`) are splitted into two separate statements, with exception when performing swapping operations as `[a,b]=[b,a]` or `[a,b,c]=[c,b,a]`. In the latter case, temporary variables are introduced.

3. Staged assignments (such as a=b=c) are also splitted into separate statements.
4. Self updates (such as a=f(a)) becomes tmp=a; a=f(tmp)

Ternary if conversion

The AD framework internally converts ternary if conditionals to "regular" if-control structures:

```
result = a?b:c
```

becomes

```
if a
    result = b
else
    result = c
endif
```

Correspondingly, the restrictions of "regular" control structures also apply to ternary if conditionals. In particular the condition a should not depend on active variables (variables that depend on the derivative variable).

Tape drive

The tape drive is used for backward AD, and denotes the sequential array that is used to store intermediate values from the forward calculations. To enable loop parallelization without sequential dependencies, the AD framework implements uses compiler technique called "data privatization". The approach is functionally identical to the tape drive, with the difference that the dependencies between subsequent loop iterations (and hence other threads) are removed.

Data privatization essentially ensures that every thread in a parallel execution has its own copy of the data. The compiler generates memory allocation statements to store the intermediate values. Two disadvantages of this technique are 1) extra (temporary) memory is required and 2) for a GPU, global memory is used for storing the privatized variables. In future versions, shared memory may be used, potentially yielding a significant speedup.

To alleviate the disadvantages, it is desirable to limit the number of intermediate values in the code. However, when most intermediate values are removed, the expansion problem of symbolic differentiation comes into play. It is therefore necessary to trade the benefits of privatized variables with the increased memory usage and memory access times. This is still an open problem.

Generic gradient solvers

As mentioned before, the meta functions $diff, $diffmul and $adjoint can be applied without the function definitions being specified. This requires the compiler to specialize the function at call-time, giving a technique to specify gradient-based optimization techniques in a genral way.

In this section, we give a number of examples of such gradient solvers.

Conjugate gradient solver

The iterative conjugate gradient (CG) solver is applicable to large sparse systems of linear equations. It can also be used to solve L^2 minimization problems. Compared to gradient descent, the conjugate gradient method optimizes the step size based on the previous gradient that was calculated. The method produces the exact solution after a finite number of iterations, independent of the matrix dimensions (apart from rounding errors).

```
function x = conjugate_gradients(A, y, max_iter=20, tol=1e-15)
    x = 0
    A_H = $unapply($adjoint(A(y), y), y)
    r = A_H(y)
    p = r
    new_err = dotprod(r, r)

    for it=1..max_iter
        err=new_err
        print "CG iteration ",it," err=",err
        A_p = A_H(A(p))
        alpha = err / dotprod(p, A_p)
        x = x + alpha * p
        r = r - alpha * A_p
        new_err = dotprod(r, r)
        if (new_err/numel(x) < tol)
            break
        endif
        beta = new_err / err
        p = r + beta * p
    endfor
endfor
```

Non-linear conjugate gradient solver

Non-linear CG is a generalization of linear CG to cost functions with nonlinear derivatives (i.e. nonlinear optimization). The method calculates a local conjugate gradient direction and then employs a line search to determine the optimal step size.

```
function y = nonlinear_cg(cost_fn, x, max_iter=120, epsilon=1e-4)
    gradient = $unapply($diff(cost_fn(x), x), x)

    function alpha = line_search(cost_fn, y, s)
        alpha = 4.0
        orig_val = cost_fn(y)
        while cost_fn(y + alpha * s) > orig_val
```

```
            alpha *= 0.25 %* can be adjusted
        endwhile
    endfunction

    % Initial solution
    y = x

    % Steepest decent solution
    Delta = -gradient(y)
    s = Delta
    alpha = line_search(cost_fn, y, s)
    y += alpha * s

    for k=0..max_iter
        print "iter ",k," cost=",cost_fn(y)

        % Calculate the steepest direction
        Delta_last = Delta
        Delta = -gradient(y)

        % Fletcher-Reeves update (other update formulas may be used as well)
        beta = dotprod(Delta,Delta)/dotprod(Delta_last,Delta_last)

        % Calculate the conjugate direction
        s_last = s
        s = Delta + beta * s_last

        % Perform a line search
        alpha = line_search(cost_fn, y, s)

        % Update the solution
        y_old = y
        y = y_old + alpha * s

        % Stop condition
        if sum(abs(y-y_old)) < epsilon
            break
        endif
    endfor
endfunction
```

A hybrid conjugate gradient minimizer

Nonlinear CG can as well be applied to quadratic cost functions, although in this case, the line search is unnecessary. Taking into account that the AD framework allows to determine whether a function is linear (or affine), we can write a generic arguments of minima (argmin) function:

```
function z = argmin(f : [(??,??) -> scalar], y0)
    symbolic x : cube, y : cube

    % Calculate the derivative of f with respect to x
    g = $unapply($diff(f(x, y), x), x, y)

    if $isaffine(g(x, y), x)
        print "f is linear in x: using conjugate gradients"
```



```
        z = linear_cg(f, g, y0)
    else
        print "f is non-linear in x: using non-linear conjugate gradients"
        z = nonlinear_cg(f, g, y0)
    endif
endfunction
```

Newton-Raphson (second order AD) minimizer

The Newton-Raphson algorithm generally converges in fewer iterations than the nonlinear CG method, although each iteration requires more computation. The Newton-Raphson algorithm requires the Hessian matrix (which is a second order derivate). Luckily, the Hessian matrix does not need to be fully calculated: only the product of the Hessian matrix with a vector is required. The (sparse) Hessian matrix is inverted using the conjugate gradient method.

```
function y = newton_raphson(cost_fn : [?? -> scalar], x, max_iter=10, epsilon=1e-6)
    dx=zeros(size(x))
    gradient = $unapply($diff(cost_fn(x), x), x)
    % The hessian is actually the Jacobian of the gradient
    hessian = $unapply($diffmul(gradient(x), x, dx), x, dx)
    % Calculate the adjoint of the Hessian
    hessian_t = $unapply($adjoint(hessian(x, dx), dx), x, dx)

    max_inner_iter = 1
    step_size = 0.1
    tol = 1e-6
    lambda = 1e-4 % regularization

    % Initial solution
    y = x
    for k=0..max_iter
        % Solve H f(y) = x using normal equations
        % => H^T H f(y) = H^T grad(y)
        u = 0
        g = gradient(y)
        r = hessian_t(y,g)
        p = r
        new_err = dotprod(r, r)
        print "iteration ",k," err=",new_err

        for it=1..max_inner_iter
            err=new_err
            A_p = lambda*p + hessian_t(y, hessian(y, p))
            alpha = err / dotprod(p, A_p)
            u = u + alpha * p
            r = r - alpha * A_p
            new_err = dotprod(r, r)
            if new_err/numel(u) < tol
                break
            endif
            beta = new_err / err
            p = r + beta * p
        endfor

        y_old = y
        y = y_old - step_size * u
```



```
        % Stop condition
        if sum(abs(y-y_old)) < epsilon
            break
        endif
    endfor
endfunction
```


Examples

In this section, we give a number of examples on how to use the AD framework in practice.

Example 1: Total variation denoising

Total variation regularization is one of the basic solutions for several inverse problems, including image denoising and deblurring.

```
import "Quasar.CompMath.dll"

% C = sum((g - b).^2) + lambda*|dg/dx| + lambda*|dg/dy|
function C = costfunc(g : mat, b : mat, lambda : scalar)
    C = 0.0
    for m=0..size(g,0)-1
        for n=0..size(g,1)-1
            C += (g[m,n] - b[m,n]).^2 % data fidelity
            C += lambda*abs(g[m,n-1] - g[m,n]) % horizontal partial derivative
            C += lambda*abs(g[m-1,n] - g[m,n]) % vertical partial derivative
        endfor
    endfor
endfor

function y = gradient_descent(cost_fn, x, step_size=0.01, max_iter=100, epsilon=1e-4)
    % Maple-like syntax for obtaining the gradient
    gradient = $unapply($diff(cost_fn(x), x), x)

    % Initial solution
    y = x
    for k=0..max_iter
        y_old = y
        y = y_old - step_size * gradient(y)
        print "iteration ",k+1, ": ", cost_fn(y)

        % Stop condition
        if sum(abs(y-y_old)) < epsilon
            break
        endif
    endfor
endfunction

im = imread("lena_big.tif")[:,:,1]
im_noisy = im + 25 * randn(size(im))
lambda = 40

% Call the gradient descent algorithm to derive the cost function
im_den = gradient_descent((x -> costfunc(x, im_noisy, lambda)), im_noisy)
imshow(im_den,[0,255])

psnr = (x, y) -> 10*log10(255^2/mean((x - y).^2))
print "psnr_in=", psnr(im_noisy, im), " dB"
print "psnr_out=", psnr(im_den, im), " dB"
```

Note in particular that for calling the gradient_descent function, a lambda expression x -> costfunc(x, im_noisy, lambda) is passed. This allows for closure variable binding, and gives a mechanism to essentially store pass extra data to a function

which expects a function $f(x)$ with a single input argument. In other words, the variables im_noisy and lambda which are locally defined, can be passed to the cost function.

Example 2: Non-blind image deconvolution

This example illustrates modularity using the chain rule, for a deblurring application. The degradation is modeled using a forward blur filter. Then a data fidelity function measures the closeness of the current solution to the measurement data. Both the blur filter and data fidelity function are joined together in the costfunc. The remaining of the example is essentially the same as in the TV example.

```
function [y : mat] = blur_filter(x : mat, f : vec)
    N = int(numel(f)/2)
    y = zeros(size(x))
    x1 = zeros(size(y))

    % Horizontal filter
    for m=0..size(x,0)-1
        for n=0..size(x,1)-1
            sum = 0.0
            for k=0..numel(f)-1
                sum += x[m,k+(n-N)] * f[k]
            end
            x1[m,n] = sum
        endfor
    endfor

    % Vertical filter
    for m=0..size(x,0)-1
        for n=0..size(x,1)-1
            sum = 0.0
            for k=0..numel(f)-1
                sum += x1[k+(m-N),n] * f[k]
            endfor
            y[m,n] = sum
        endfor
    endfor
endfor

function C = data_fidelity(u : mat, y : mat)
    C = 0.0
    for m=0..size(u,0)-1
        for n=0..size(u,1)-1
            C += (u[m,n] - y[m,n])^2
        endfor
    endfor
endfor

function C = costfunc(x : mat, y : mat, f : vec)
    u = blur_filter(x, f)
    C = data_fidelity(u, y)
endfunction

function y = gradient_descent(cost_fn, x, step_size=0.1, max_iter=500, epsilon=1e-4)
    % Maple-like syntax for obtaining the gradient
    gradient = $unapply($diff(cost_fn(x), x), x)
```



```
    % Initial solution
    y = x

    for k=0..max_iter
        y_old = y
        y = y_old - step_size * gradient(y)

        % Stop condition
        if sum(abs(y-y_old)) < epsilon
            break
        endif
    endfor
endfunction

f = [1,2,3,2,1]
f = f / sum(f)

im = imread("lena_big.tif")[:,:,1]

% Blur and add noise
im_blurred = blur_filter(im, f) + 2*randn(size(im))

im_restored = gradient_descent(((x : mat) -> costfunc(x, im_blurred, f)), im_blurred)

imshow(im_restored,[0,255])

psnr = (x, y) -> 10*log10(255^2/mean((x - y).^2))
print "psnr_in=", psnr(im_blurred, im), " dB"
print "psnr_out=", psnr(im_restored, im), " dB"
```

Example 3: 1D Newton Raphson

The following example demonstrates how to implement Newton Raphson in a 1D (scalar) setting. The implementation is significantly simpler compared to the multivariate version, because the inverse of the Hessian matrix amounts to the reciprocal of a scalar number.

```
function y = f(x : scalar)
    y = -x.^4 + 6*x.^2 + 4*x
endfunction

function y = f(x : vec)
    y = -x.^4 + 6*x.^2 + 4*x
endfunction

function x = gauss_newton(costfunc, x0, epsilon = 1e-8)
    x = x0
    f1 = $unapply($diff(costfunc(x), x), x)
    f2 = $unapply($diff($diff(costfunc(x), x), x), x)

    repeat
        x_old = x
        x = x - f1(x)/f2(x)
        print x
    until abs(x - x_old) < epsilon
endfunction
```



```
function [] = main()
    % Finding the minimum of a scalar function
    x = gauss_newton((x : scalar -> f(x)), 0.0)
    print x

    r = linspace(-1,1,256)
    plot(r, f(r))
endfunction
```


Differentiable Programming: Convolutional Neural Networks step by step

In this tutorial, we explain how to implement deep neural (DL) networks with forward and backward propagation using *differentiable programming*, a new programming paradigm that is recently mentioned on the Facebook page of Yann Lecunn (Jan, 2018):

> "*OK, Deep Learning has outlived its usefulness as a buzz-phrase. Deep Learning est mort. Vive Differentiable Programming! Yeah, Differentiable Programming is little more than a rebranding of the modern collection Deep Learning techniques, the same way Deep Learning was a rebranding of the modern incarnations of neural nets with more than two layers. But the important point is that people are now building a new kind of software by assembling networks of parameterized functional blocks and by training them from examples using some form of gradient-based optimization. An increasingly large number of people are defining the networks procedurally in a data-dependent way (with loops and conditionals), allowing them to change dynamically as a function of the input data fed to them. It's really very much like a regular progam, except it's parameterized, automatically differentiated, and trainable/optimizable. Dynamic networks have become increasingly popular (particularly for NLP), thanks to deep learning frameworks that can handle them such as PyTorch and Chainer (note: our old deep learning framework Lush could handle a particular kind of dynamic nets called Graph Transformer Networks, back in 1994. It was needed for text recognition). People are now actively working on compilers for imperative differentiable programming languages. This is a very exciting avenue for the development of learning-based AI.*"

In differentiable programming, networks are specified using parametric loss functions, in a modular fashion, and the language provides a means to calculate the derivative of the loss function. This is accomplished using algorithmic differentiation (AD, also called automatic differentiation), which - rather than calculating the derivatives numerically or symbolically - applies the chain rule to every part of the algorithm, taking control structures like loops and branches into account. When applied to a modular design such as a neural network, AD calculates the derivatives of all of the network components as well as the derivative of the loss function which combines the component derivatives. For neural networks, this automatically yields the backpropagation.

Different to many existing DL approaches is that AD is applied on a much finer granularity, namely, instead of treating the network components as building blocks and relying on pre-implemented derivatives of the building blocks, the AD framework can analyze and calculate derivatives for any algorithm that implements a forward layer. This brings a lot of flexibility in designing the individual layers (think about non-standard layers such as layers that estimate depth maps from 2D images), but also leads to a programming model that is conceptually easy to understand.

Additionally, AD can also be applied (and is intended to be applied) in applications other than neural networks (for example, to optimize parameters of differentiable cost functions using gradient descent-based methods), but for this tutorial we stick to the deep learning application.

Step-by-step tutorial

Algorithmic differentiation

Quasar has a special function $diff () which allows you to calculate the (partial or non-partial) derivative of a function specified in Quasar, using algorithmic differentiation (AD). How does this work? A Quasar function is interpreted as a sequence of function compositions $y = f(g(h(x)))$ to which the chain rule can be applied:

$$\frac{dy}{dx} = \frac{dy}{dw_2}\frac{dw_2}{dw_1}\frac{dw_1}{dx}$$

There are two modes of AD: *forward accumulation*, in which the chain rule is traversed from inside to outside (just like when calculating the derivative symbolically):

$$\frac{dw_i}{dx} = \frac{dw_i}{dw_{i-1}}\frac{dw_{i-1}}{dx} \quad \text{with} \quad w_3 = y$$

and *backward accumulation* which traverses the chain rule from outside to inside:

$$\frac{dy}{dw_i} = \frac{dy}{dw_{i+1}}\frac{dw_{i+1}}{dw} \quad \text{with} \quad w_0 = x$$

Both modes give the same result but they have different characteristics when applied to multivariate functions versus vector functions. Quasar implements mainly the backward accumulation in the form of source-to-source translation with some support for forward accumulation (actually, the most efficient code generation, optimization and parallelization is obtained using a mix of forward and backward propagation). The advantage of AD is that is can be applied to complex functions with control structures (if, for, while etc.). Moreover, when applied to the cost function of a neural network, the backward accumulation gives the backpropagation functions in an *automatic* way. A side effect is that AD is not limited to cost functions of neural networks, it is therefore a generalizing framework and also called *differentiable programming*.

Library imports

At the beginning of the program, we import the Quasar CompMath library.

```
import "Quasar.CompMath.dll"
```

This library is actually a compiler plugin, and adds several *metafunctions* to Quasar (metafunctions are functions that are evaluated at compile-time and are always indicated by the prefix $), such as $diff (for differentiation) and $adjoint (for adjoint calculation of a linear function).

Next, we also import the DNN library, which is thin layer on top of the NVIDIA CuDNN library and hence implements basic blocks of neural networks.

```
import "Quasar.DNN.dll"
```


For visualization purposes, we also import the Quasar graphical user interface library:

```
import "Quasar.UI.dll"
```

Defining the basic layers

Rectified linear unit The most straightforward way to implement a rectified linear unit is the following:

```
function y = __device__ RELU(x : scalar)
    if x<0
        y=0
    else
        y=x
    endif
endfunction
```

Here, __device__ indicates that the function will be compiled to run on an accelerator (e.g., GPU). We can calculate the derivative of the function with respect to the variable x simply as follows:

```
RELU_deriv_x = x -> $diff(RELU(x), x)
```

which generates the following function:

```
function y_deriv = __device__ RELU_deriv_x(x:scalar)
    if x<0
        y_deriv=0
    else
        y_deriv=1
    endif
endfunction
```

In this case, the resulting function is quite trivial, but the AD framework is much more general: various algorithmic constructs, such as loops, nested branches, matrix/vector operations, aggregation functions can be used within the function. Note that the derivative of RELU(x) with respect to x does not exist in x=0, however, it is common for AD frameworks (as well as in neural networks) to ignore this and to provide either the left or the right derivative in this point.

Alternatively, the RELU function can be implemented in branch-free manner, as follows:

```
function y = __device__ RELU(x : scalar)
    y=max(x,0)
endfunction
```

Note: the Quasar compiler actually has an internal branch divergence reduction transform which converts the if branch from the above example to a branch-free form.

The $diff () function applies the chain rule to the function that calculates the RELU. To obtain the result, the derivative of max(x,0) with respect to x is known. In Quasar, derivatives of arbitrary functions can be specified using reductions. For this

bivariate function, this gives:

```
reduction (x, y) -> $diff(max(x, y), x) = (x > y ? 1 : 0)
reduction (x, y) -> $diff(max(x, y), y) = (x > y ? 0 : 1)
```

A reduction is a pattern that works on the abstract syntax tree of the computer program (sometimes also called "rewriting rule"): whenever the compiler encounters an "pattern" expression `$diff(max(x, y), x)` with free variables `x` and `y`, the compiler will replace the left-handed side of the equality by the right-handed side. Reductions provide a simple mechanism to add domain knowledge to the compiler. Some additional diff-reductions that are useful:

```
reduction x -> $diff(exp(x), x) = exp(x)
reduction x -> $diff(tanh(x), x) = 1 - tanh(x).^2
```

Note that all of these "basic" differentiation reductions are integrated in Quasar.CompMath.dll, so it is no longer necessary to define them yourselves for the most cases. The above code applies a RELU function to an individual scalar value. To implement a neural network layer, it is desirable to apply the RELU function to each element of the data cube. This can be achieved by using for-loops:

```
function y = pointwise_RELU(x : Data)
    y = zeros(size(x))
    for [m,m,p,q]=0..size(x,0..3)-1
        y[m,n,p,q] = RELU(x[m,n,p,q])
    endfor
endfunction
```

Here we use a 4-dimensional for-loop (a syntax that was introduced in Quasar in Feb. 2019). The Quasar compiler will parallelize the for-loop automatically, to run on GPU (or on multi-core CPU). Also, when calculating the derivative of pointwise_RELU, for loops will be generated that are automatically parallelized and optimized for the target device. The advantage of AD is then that we no longer need to think about the implementation of the derivatives (or, in alternative words, the back propagation).

Data and parameter type definitions

Next, we can think about the data structures and data layout. In principle, Quasar does not impose any restrictions here, but to avoid unnecessary conversions, it is a good idea to align with the CuDNN library. For example, a common choice is to represent the data in NCHW (number, channels, height and width) format, which means that the first dimension is the batch, the second dimension contains the color channels, the third dimension the image height and the fourth the image width. Idem for the parameters:

```
type Data : cube{4} % in NCHW format
```

where `cube{4}` is a 4D hypercube (also called tensor). For convolutional layers, we can represent the weights in a similar format


```
type Parameters : cube{4} % in PCHW format
```

where PCHW stands for (output, channels, height and with). A convolutional layer has then C input channels and P output channels, the convolution is applied to every image of the batch.

The parameters of an entire network with three convolutional layers and one bias can then be represented as follows:

```
type NetworkParameters : vec[{Parameters, Parameters, Parameters, vec}]
```

which denotes a "cell" vector of length 4, where each element is of the specified type. Each element of a variable of type NetworkParameters is then either a 4D cube or a 1D vector, depending on the position. Note that arbitrary (nested) cell structures can be defined that work within the AD framework. The advantage of cell structures in Quasar is that they allow arithmetic calculations. For example, a recursive filter on the network parameters can be implemented simply as follows:

```
    params : NetworkParameters = init_network_params()
    new_params : NetworkParameters = ...
    params = params + alpha * (new_params - params)
```

Later, we will see that the network parameters can also be represented by a user-defined class (dynamic class) which also supports high-level arithmetic. This allows naming the individual parameters/parameter groups, which generally leads to more readible code and less bugs. To keep things simple, we will proceed with the cell vector definition from above.

Convolutional layers

A convolution is fairly straightforward to implement in Quasar. To enable AD, we specify the implementation using a loop-nest rather than using a kernel function.

```
function [y : Data] = convolve(x : Data, f : Parameters, radius : int)
  radius = int(size(f,2)/2)
  for [b,p,m,n]=0..[size(input,0),size(weights,2),size(input,2),size(input,3)]-1
    sum = 0.0
    for [dy,dx]=-radius..radius
      for c = 0..size(input,1)-1
        sum += weights[c,p,dy+radius,dx+radius,c] * input[b,c,m+dy,n+dx]
      endfor
    endfor
    result[b,p,m,n] = sum
  endfor
endfunction
```

Again, we may obtain the derivatives as follows:

```
deriv_conv_f = (x, f, bias, radius) -> $diff(convolve(x, f, bias, radius), f)
deriv_conv_x = (x, f, bias, radius) -> $diff(convolve(x, f, bias, radius), x)
```


In practice, convolutional layers are quite computationally intensive, therefore the above implementation needs to be optimized more (in particular, by exploiting the exact value of radius and size(input,1); and by taking of advantages of shared memory of the GPU). Many (semi)automatic optimizations are available in the Quasar compiler; the 4D loop will be parallelized and the resulting kernel function will run efficiently on a GPU.

As an alternative, it is useful to use hand-tuned functions from the CuDNN library. For this purpose, we implement convolve using the cudnn_convolution_forward.

```
function [y : Data] = convolve(x : Data, f : Parameters)
    [K,L] = int(size(f,2..3)/2) % half filter support size
    conv_desc = new(qdnn_conv_descriptor)
    conv_desc.pad_w = L
    conv_desc.pad_h = K
    input_c = x
    output_c = uninit(size(x,0),size(f,0),size(x,2),size(x,3))
    cudnn_convolution_forward(1.0, input_c, 0.0, output_c, f, conv_desc)
    y = output_c
endfunction
```

However, the side effect is that the $diff function now cannot see the derivatives of cudnn_convolution_forward with respect to x and f. We therefore need to specify these derivatives manually. The derivatives are actually nothing more than the backward steps of the convolutional layer:

```
function [df : Parameters] = convolve_diff_f(x : Data, f : Parameters, y : Data)
    [K,L] = int(size(f,2..3)/2)
    conv_desc = new(qdnn_conv_descriptor)
    conv_desc.pad_w = L
    conv_desc.pad_h = K
    input_c = x
    dzdoutput_c = y
    dzdw = uninit(size(f))
    cudnn_convolution_backward_filter(1.0,input_c,dzdoutput_c,0.0,dzdw,conv_desc)
    df = dzdw
endfunction

function [dx : Data] = convolve_diff_x(x : Data, f : Parameters, y : Data)
    [K,L] = int(size(f,2..3)/2)
    conv_desc = new(qdnn_conv_descriptor)
    conv_desc.pad_w = L
    conv_desc.pad_h = K
    input_c = copy(x)
    dzdoutput_c = copy(y)
    dzdinput = uninit(size(input_c))
    cudnn_convolution_backward_data(1.0,f,dzdoutput_c,0.0,dzdinput,conv_desc)
    dx = dzdinput
endfunction
```

As above, we use reductions to allow the compiler to use the convolve_diff_f and convolve_diff_x as derivatives of convolve, respectively with respect to f and x:

```
reduction (x, f, y) -> $diffmul(convolve(x, f), f, y) = convolve_diff_f(x, f, y)
reduction (x, f, y) -> $diffmul(convolve(x, f), x, y) = convolve_diff_x(x, f, y)
reduction (x, f, y) -> $adjoint(convolve_diff_f(x, f, y), y) = convolve_diff_f(x, f, y)
```



```
reduction (x, f, y) -> $adjoint(convolve_diff_x(x, f, y), y) = convolve_diff_x(x, f, y)
```

Now something special has happened: we are specifying $diffmul() and $adjoint(). For $diffmul() the reason is that convolve is a function that maps a 4D cube onto a 4D cube. If we determine the derivative, the result would have 8 dimensions, which cannot be stored in the memory of the computer (unless the cube size would be small). Instead, $diffmul(f(x),x,u) calculates the inner product of the derivative of f(x) w.r.t. x and a vector (or cube) u. By taking the inner product, the result of the operator is always a function that returns a 4D cube. When applying the chain rule, this appears to be sufficient.

Additionally, the adjoint of convolve_diff_f(x, f, y) with respect to y is required for AD. Again, the compiler can derive adjoints of functions automatically, except when the function based on "black box" building blocks (such as the cudnn_convolution_forward in the above example). Therefore, the adjoint needs to be specified which is in this case simply an identity operation (because convolve_diff_x does not depend on y).

Batch normalization

Batch normalization can fairly easily be implemented using the Quasar repmat and mean functions.

```
function [y : Data] = batch_normalization(x : Data)
    {!ad_support convert_matrix_ops_to_loops=true}
    mu = repmat(mean(x, 0), [size(x,0),1,1,1])
    sigma = repmat(sqrt(mean((x - mu).^2, 0)), [size(x,0),1,1,1])
    y = (x - mu) ./ sigma
endfunction
```

Because these functions have arrays as input and arrays as output, we call them "higher level" functions. By default, using {!ad_support} with no additional parameters, higher level functions are differentiated using reductions. Practically, this means that the derivatives and adjoint of repmat and mean need to be specified. As an easier alternative, it is also possible to specify that Quasar converts the operations to loops involving only scalar values. This is achieved using the code attribute {!ad_support}, which allows to specify some algorithmic differentiation options (here, convert_matrix_ops_to_loops=true).

This setting also activates a nested loop optimization pipeline within the compiler.

Miscellaneous layers

Similarly, it is possible to incorporate various other layers. For example, a 2D dual-tree complex wavelet layer can be defined as follows:

```
my_idtcwt : [(mat[Data], mat, mat, int) -> mat] = (w, w1, w2, J) -> pidtcwt(w, w1, w2, J)
my_dtcwt : [(mat, mat, mat, int) -> mat[Data]] = (w, w1, w2, J) -> pdtcwt(w, w1, w2, J)
```

Here, the types are specified statically, although it is optional. To be able to use the functions my_dtcwt and my_idtcwt as layers in a neural network, it is necessary to define the $adjoint() and $diffmul() (in theory, the AD framework can do it by itself, but in this case the functions are linear and because the 2D dual-tree complex wavelets form a tight frame, the adjoint is simply the inverse. Therefore, we can help the AD framework by defining the adjoints and derivatives explicitly:


```
reduction (w, w1, w2, J) -> $adjoint(my_dtcwt(w, w1, w2, J), w) = my_idtcwt(w, w1, w2, J)
reduction (w, w1, w2, J) -> $adjoint(my_idtcwt(w, w1, w2, J), w) = my_dtcwt(w, w1, w2, J)

reduction (w, w1, w2, J, dw) -> $diffmul(my_dtcwt(w, w1, w2, J), w, dw) = my_dtcwt(dw, w1, w2, J)
reduction (w, w1, w2, J, dw) -> $diffmul(my_idtcwt(w, w1, w2, J), w, dw) = my_idtcwt(dw, w1, w2, J)
```

With only a few lines of code, new layers can be defined and integrated in the neural network. Again, the AD framework will automatically "glue" the different layers together by the use of the derivative chain rule.

Network parameter Initialization

For the initialization of the neural network parameters, we write a function that returns a NetworkParameters instance.

```
function w : NetworkParameters = init_network()
    bias = rand(3)
    w1 = get_gabor_convolve_layer(radius:=5, channels_in:=1, channels_out_factor:=13)
    w2 = rand(4, 13, 1, 1) % radius = 1
    w3 = rand(1, 4, 5, 5) % radius = 2

    % Group the weights and return a NetworkParameters value
    w = {w1, w2, w3, bias}
endfunction
```

Here, we initialize the weights of the first convolutional layer using a set of Gabor filter bank (the implementation of this function is omitted here). The other weights are initialized as uniformly distributed random values between 0 and 1. The dimensions of the parameters need to match either the input data cube (for example, the number of color channels) or are (hyper)parameters themselves (for example, the support size of the filter coefficients).

Forward network implementation and loss function

Once the individual components of the neural network have been defined, the forward network implementation is as easy as chaining these components together.

```
function y = forward_network(x : Data, w : NetworkParameters)
    x1 = convolve(x, w[0])
    x2 = convolve(x1, w[1])
    x3 = pointwise_RELU(x2)
    y = convolve(x3, w[2])
endfunction
```

When desired, the temporary variables x1, x2 and x3 can be avoided:

```
y = convolve(pointwise_RELU(convolve(convolve(x, w[0]), w[1])), w[2])
```

although here the code readability slightly diminishes.

Similarly, we can define the MSE loss function:


```
function C = mse_lossfunction(u : Data, y : Data)
    C = 0.0
    for [m,n,p,q]=0..size(u,0..3)-1
        C += (u[m,n,p,q] - y[m,n,p,q])^2
    endfor
endfunction
```

Then the loss function of the neural network is obtained by chaining the forward network and the MSE loss function.

```
function C = loss_function(u : Data, y : Data, f : NetworkParameters)
    x = forward_network(y, f)
    C = mse_lossfunction(u, x)
endfunction
```

The gradient of the loss function can be obtained simply as follows:

```
gradient = (x, y, f) -> $diff(lossfunction(x, y, f), f)
```

This operation will do all of the heavy work and generate the backpropagation for the neural network.

Adam optimizer

The next step is writing an optimizer that uses the gradient from the previous section. For this purpose, we will use Adam optimizer, which is an extension of the stochastic gradient descent algorithm. Adam combines the advantages of two extensions of stochastic gradient descent: 1) the adaptive gradient algorithm (AdaGrad) that maintains a learning rate for each parameter and 2) Root Mean Square Propagation (RMSProp) that adapts per-parameter learning rates to the average of recent magnitudes of the gradients of the weights. Additionally, momentum is incorporated in the training algorithm using the first and second order moments of the parameters.

```
function [] = train_adam( _
    gradient : [(Data, Data, NetworkParameters) -> NetworkParameters], _
    inputs : vec[Data], _
    groundtruths : vec[Data], _
    params : NetworkParameters, _
    nr_iterations = 100, _
    rate = 1e-3, _
    momentum_first_order = 0.9, _
    momentum_second_order = 0.999)

    nr_images = numel(inputs)
    assert(numel(groundtruths) == nr_images)

    % initialization
    first_order_moments = 0*params
    second_order_moments = 0*params

    for iteration = 0..nr_iterations-1
        img_idx = mod(iteration,nr_images)
        input = inputs[img_idx]
        groundtruth = groundtruths[img_idx]
```



```
        % for each layer, backward propagation, updating the momentum and taking the step
        unbiased_rate = rate * sqrt(1-momentum_second_order.^(iteration+1)) / (1-momentum_first_order
            .^(iteration+1)) %see Adam paper
        optimization_epsilon = 1e-8

        % calculate the gradient
        grad = gradient(input, groundtruth, params)

        % updates the moments and the parameters (uses cell arithmetic)
        first_order_moments = first_order_moments * momentum_first_order + grad .* (1 -
            momentum_first_order)
        second_order_moments = second_order_moments * momentum_second_order + grad.^2 .* (1 -
            momentum_second_order)
        params = params - unbiased_rate * first_order_moments ./ (sqrt(second_order_moments) +
            optimization_epsilon)
    endfor
endfunction
```

Multi-GPU processing

```
type NetworkParameters : dynamic class
    w1a : Parameters
    w2a : Parameters
    w3 : Parameters
    w1b : Parameters
    w2b : Parameters
    bias : vec
endtype

function [y : Data] = merge_layers(x1 : Data, x2 : Data)
    y = zeros(size(x1))
    for [p,c,m,n]=0..size(y, 0..3)-1
        bestval = max(x1[p,c,m,n],x2[p,c,m,n])
        y[p,c,m,n]=bestval
    endfor
endfunction


function y = forward_network(x : Data, w : NetworkParameters)

    {!sched gpu_index=0}
    part1 = pointwise_RELU(convolve(convolve_biased(x, w.w1a, w.bias), w.w2a))
    {!sched gpu_index=1}
    part2 = pointwise_RELU(convolve(convolve_biased(x, w.w1b, w.bias), w.w2b))
    {!sched mode=auto}
    y = convolve(merge_layers(part1, part2), w.w3)

endfunction
```


Discussion and Conclusion

The implementation of neural networks in Quasar is quite straightforward thanks to the integrated support for algorithmic differentiation. The above tutorial is self-containing, it only relies on this special $diff metafunction to calculate the gradient.

The strong point of this approach is that it is fairly easy to extend layers or define new layers. Also, the neural network definition does not necessary need to follow the traditional graph structure as the AD allows a general modular structure (i.e. multiple levels of functions calling each other). The network layers can have branches and other control structures, which allows for example data-adaptive network designs. It is interesting to explore the new possibilities that these new designs can bring for various applications.

Reductions play a central role in the AD system: once a derivative has been calculated, it is stored as a reduction for later use. Before calculation of the derivative, the AD tool looks if a derivative has already been specified, which allows the user to override derivative implementations (for example, to be able to access CuDNN functions).

AD is also naturally integrated with the Quasar data types. In the CNN example, the parameters were passed as a cell vector. On this cell vector, arithmetic is then applied (for example, to calculate the first and second order moments). Powerful high-level expressions then manipulate an entire irregular data structure.

Instead of cell vectors, alternatively, it is also possible to use structured data such as dynamic classes:

```
type NetworkParameters : dynamic class
    w1 : Parameters
    w2 : Parameters
    w3 : Parameters
    bias : vec
endtype
```

This has the advantage that the network parameters can be named explicitly, rather than relying on fixed indices (w[0], w[1] etc.). It is even possible to mix classes and cell vectors in this definition. This allows defining an algorithm with several (layers of) parameters and consequently the derivative of the cost function with respect to all of the parameters can be calculated at once.

Appendix: the complete code (single GPU network)

```
import "Quasar.CompMath.dll"
import "Quasar.DNN.dll"
import "Quasar.UI.dll"

type Data : cube{4} % in NCHW format
type Parameters : cube{4} % in PCHW format

function y = __device__ RELU(x : scalar)
    if x<0
        y=0
    else
        y=x
    endif
endfunction
```



```
function y = pointwise_RELU(x : Data)
    y = zeros(size(x))
    for [m,m,p,q]=0..size(x,0..3)-1
        y[m,n,p,q] = RELU(x[m,n,p,q])
    endfor
endfunction

function [y : Data] = convolve(x : Data, f : Parameters)
    [K,L] = int(size(f,2..3)/2) % half filter support size
    conv_desc = new(qdnn_conv_descriptor)
    conv_desc.pad_w = L
    conv_desc.pad_h = K
    input_c = x
    output_c = uninit(size(x,0),size(f,0),size(x,2),size(x,3))
    cudnn_convolution_forward(1.0, input_c, 0.0, output_c, f, conv_desc)
    y = output_c
endfunction

function [df : Parameters] = convolve_diff_f(x : Data, f : Parameters, y : Data)
    [K,L] = int(size(f,2..3)/2)
    conv_desc = new(qdnn_conv_descriptor)
    conv_desc.pad_w = L
    conv_desc.pad_h = K
    input_c = x
    dzdoutput_c = y
    dzdw = uninit(size(f))
    cudnn_convolution_backward_filter(1.0,input_c,dzdoutput_c,0.0,dzdw,conv_desc)
    df = dzdw
endfunction

function [dx : Data] = convolve_diff_x(x : Data, f : Parameters, y : Data)
    [K,L] = int(size(f,2..3)/2)
    conv_desc = new(qdnn_conv_descriptor)
    conv_desc.pad_w = L
    conv_desc.pad_h = K
    input_c = copy(x)
    dzdoutput_c = copy(y)
    dzdinput = uninit(size(input_c))
    cudnn_convolution_backward_data(1.0,f,dzdoutput_c,0.0,dzdinput,conv_desc)
    dx = dzdinput
endfunction

reduction (x, f, y) -> $diffmul(convolve(x, f), f, y) = convolve_diff_f(x, f, y)
reduction (x, f, y) -> $diffmul(convolve(x, f), x, y) = convolve_diff_x(x, f, y)
reduction (x, f, y) -> $adjoint(convolve_diff_f(x, f, y), y) = convolve_diff_f(x, f, y)
reduction (x, f, y) -> $adjoint(convolve_diff_x(x, f, y), y) = convolve_diff_x(x, f, y)

function [w : cube{4}] = get_gabor_convolve_layer(radius:int=5,channels_in:int=1,channels_out_factor:
    int=5)
    if channels_out_factor != 5 && channels_out_factor != 13
        error("Only gabor filter bank with 5 or 13 output factor is currently available")
    endif
    %now initialize those filters!
    channels_out = channels_out_factor*channels_in
    w = zeros(channels_out, channels_in,2*radius+1,2*radius+1)
    for i = 0..channels_in-1
        %bandwidth 0
        w[i*channels_out_factor+0 ,i,:,:] = gabor_kernel(radius,0,0)
        %bandwidth 1
```



```
        w[i*channels_out_factor+1 ,i,:,:] = gabor_kernel(radius, 0,1)
        w[i*channels_out_factor+2 ,i,:,:] = gabor_kernel(radius, pi/4,1)
        w[i*channels_out_factor+3 ,i,:,:] = gabor_kernel(radius, pi/2,1)
        w[i*channels_out_factor+4 ,i,:,:] = gabor_kernel(radius,3*pi/4,1)

        if channels_out > 5
        %bandwidth 2
        w[i*channels_out_factor+5 ,i,:,:] = gabor_kernel(radius, 0,2)
        w[i*channels_out_factor+6 ,i,:,:] = gabor_kernel(radius, pi/8,2)
        w[i*channels_out_factor+7 ,i,:,:] = gabor_kernel(radius, pi/4,2)
        w[i*channels_out_factor+8 ,i,:,:] = gabor_kernel(radius,3*pi/8,2)
        w[i*channels_out_factor+9 ,i,:,:] = gabor_kernel(radius, pi/2,2)
        w[i*channels_out_factor+10,i,:,:] = gabor_kernel(radius,5*pi/8,2)
        w[i*channels_out_factor+11,i,:,:] = gabor_kernel(radius,3*pi/4,2)
        w[i*channels_out_factor+12,i,:,:] = gabor_kernel(radius,7*pi/8,2)
        endif
    endfor
endfunction

function w : NetworkParameters = init_network()
    bias = rand(3)
    w1 = get_gabor_convolve_layer(radius:=5, channels_in:=1, channels_out_factor:=13)
    w2 = rand(4, 13, 1, 1) % radius = 1
    w3 = rand(1, 4, 5, 5) % radius = 2

    % Group the weights and return a NetworkParameters value
    w = {w1, w2, w3, bias}
endfunction

function C = mse_lossfunction(u : Data, y : Data)
    C = 0.0
    for [m,n,p,q]=0..size(u,0..3)-1
        C += (u[m,n,p,q] - y[m,n,p,q])^2
    endfor
endfunction

function C = loss_function(u : Data, y : Data, f : NetworkParameters)
    x = forward_network(y, f)
    C = mse_lossfunction(u, x)
endfunction

function [] = train_adam( _
    gradient : [(Data, Data, NetworkParameters) -> NetworkParameters], _
    inputs : vec[Data], _
    groundtruths : vec[Data], _
    params : NetworkParameters, _
    nr_iterations = 100, _
    rate = 1e-3, _
    momentum_first_order = 0.9, _
    momentum_second_order = 0.999)

    nr_images = numel(inputs)
    assert(numel(groundtruths) == nr_images)

    % initialization
    first_order_moments = 0*params
    second_order_moments = 0*params

    for iteration = 0..nr_iterations-1
        img_idx = mod(iteration,nr_images)
```



```
        input = inputs[img_idx]
        groundtruth = groundtruths[img_idx]

        % for each layer, backward propagation, updating the momentum and taking the step
        unbiased_rate = rate * sqrt(1-momentum_second_order.^(iteration+1)) / (1-momentum_first_order
            .^(iteration+1)) %see Adam paper
        optimization_epsilon = 1e-8

        % calculate the gradient
        grad = gradient(input, groundtruth, params)

        % updates the moments and the parameters (uses cell arithmetic)
        first_order_moments = first_order_moments * momentum_first_order + grad .* (1 -
            momentum_first_order)
        second_order_moments = second_order_moments * momentum_second_order + grad.^2 .* (1 -
            momentum_second_order)
        params = params - unbiased_rate * first_order_moments ./ (sqrt(second_order_moments) +
            optimization_epsilon)
    endfor
endfunction

function [] = main()
    % simplified data loading
    im_gt = imread("data/testimage_gt.png")[:,:,0]/255.0
    im_mask = imread("data/testimage_mask.png")[:,:,0]/255.0
    im_input = imread("data/testimage_input.png")[:,:,0]/255.0

    params = init_network()

    gradient = (x, y, f) -> $diff(lossfunction(x, y, f), f)

    train_adam(cost_fn := lossfunction,
               forward_network := forward_network,
               gradient := gradient,
               inputs := {shuffledims(im_input, [3,2,0,1])},
               groundtruths := {shuffledims(im_gt, [3,2,0,1])},
               params := params,
               nr_iterations := 1000,
               rate := 1e-3,
               visualization := true)

    save("trained_network.qd", params)
endfunction
```


Convex optimization framework

In this section we demonstrate how the algorithmic differentiation (AD) framework can be used to design a large-scale convex optimization framework, named GASPACHO. AD provides a means to perform matrix-free computation: all computations are specified by means of a computer program, but there is no matrix representation involved storing the coefficients. In combination with iterative solvers AD allows solving large systems of equations with many variables.

To extend the possibilities of the AD framework, differentiation is combined with custom solvers that are defined by means of reductions. This permits defining a generic "argmin" function to solve convex optimization problems. A reduction can then be used to define how to solve a certain subproblem of the convex optimization problem. Depending on the structure of the objective function, the cost function can be broken up in different "sub" functions that can be solved individually by simple methods (using a technique called "variable splitting"). The advantage of the technique outlined here is that the variable splitting can be done automatically and transparently for the user.

In the following, we will focus on convex optimization problems of the form:

$$\hat{\mathbf{x}} = \arg\min_{\mathbf{x}} ||\mathbf{y} - \mathbf{W}(\mathbf{x})||_2^2 + \sum_{i=1}^{I} \lambda_i |\mathbf{S}_i(\mathbf{x})|_1$$

These optimization problems frequently occur in image processing (e.g., in image reconstruction and restoration). The first term is a data fitting function, the second term contains I regularization terms. The functions $\mathbf{W}(\mathbf{x})$ and $\mathbf{S}_i(\mathbf{x})$, $i = 1, ...I$ are specified by means of an algorithm. The goal is then to algorithmically derive a solution method for the above problem.

First, a trivial solution would consist of the gradient descent method or non-linear conjugate gradient method. The reduce the computation time of these iterative methods, the approach outlined here will exploit the structure of the optimization problem in order to arrive at a more direct solution. The approach is one step in the direction of a generic $\arg\min$ solver that exploits the structure of the underlying problem. For example,

$$\arg\min_{\mathbf{x}} ||\mathbf{y} - \mathbf{W}(\mathbf{x})||^2 \rightarrow \text{conjugate gradient solver}$$

$$\arg\min_{\mathbf{x}} ||\mathbf{y} - \mathbf{x}||^2 + \lambda||\mathbf{x}||^2 \rightarrow \text{pointwise solver}$$

$$\arg\min_{\mathbf{x}} ||\mathbf{y} - \mathbf{x}||^2 + \lambda|\mathbf{x}|_1 \rightarrow \text{soft thresholding solver}$$

To take advantage of the structure in the objective function, pattern matching and term rewriting is employed. In Quasar, this is easily achieved by defining reductions. One reduction matches one specific $\arg\min$ expression and expands it into the corresponding solver. During pattern matching, "sub"-functions are identified (for example $||x||^2$) which are in a next step passed to the selected solver. The solver can then be specialized for the functions being used, yielding a solution method that is optimized toward the specific input optimization problem.

Results of this research have been presented at Wavelets and Sparsity XVII, SPIE Optics & Photonics 2017 in San Diego, USA:

B. Goossens, H. Luong and W. Philips, "GASPACHO: a Generic Automatic Solver using Proximal Algorithms for Convex Huge Optimization problems," Wavelets and Sparsity XVII, SPIE Optics & Photonics 2017, Aug. 6-10, 2017, San Diego, CA, USA.

Variadic lambda-capturing reductions

To be able to match expressions with arbitrary number of terms, *variadic lambda capturing reductions* have been added to Quasar. A lambda capturing reduction allows a function to be captured from the context. For example:

```
reduction (x, f : [??->??], g : [??->??]) -> f(x)^3+g(x)^3 = result(f,g,x)
```

will match cos(x)^3+sin(x)^3 with f=cos and g=sin. By lambda capturing, the reduction pattern matching can be applied under more general circumstances. The captured functions are then passed to the solution method (result ()).

Often, the number of terms is not known in advance (or, it is desirable to implement a convex solver that is general enough to handle a variable number of terms). Variadic lambda-capturing reductions generalize lambda capturing to an arbitrary number of terms and/or functions. For this purpose, it is necessary to be able to express notation such as

$$f_1(x) + f_2(x) + ... + f_N(x)$$

in Quasar. The classical mathematical formulation is not the best suited for a compiler, because of possible ambiguities. Therefore, an alternative notation has been defined. Declaring f as a vector of functions with one input parameter and one output parameter:

```
f : vec[[??->??]]
```

The expression $f_1(x) + f_2(x) + ... + f_N(x)$ can be written in Quasar as:

```
sum([f(x)])
```

which is essentially a vector-based translation of $\sum_{n=1}^{N} f_n(x)$. Because f is a cell array, the evaluation of sum([f(x)]) is unknown to the Quasar compiler. Therefore it is necessary to implement sum([f(x)]), using a function and a reduction:

```
function y = vecfunc_sum(f : vec[[??->??]], x, n)
    y = 0.0
    for k=0..numel(f)-1
        y += f[k](x)^n
    endfor
endfunction

reduction {priority=high}, (x,...f) -> sum([f(x)]) = vecfunc_sum(f, x, 1)
```

Here, { priority =high} indicates that the reduction has a high priority: in case of reduction conflicts, this reduction has precedence. Using the above expression, the reduction pattern matching is significantly simplified, without losing generality. For example, the expression

$$f_1^2(x) + f_2^2(x) + ... + f_N^2(x)$$

can be written as follows:


```
sum([f(x).^2])
```

and the implementation is entirely analogous:

```
reduction {priority=high}, (x,f : vec[[?? -> ??]]) -> sum([f(x).^2]) = vecfunc_sum(f, x, 2)
```

Using the above techniques, expressions involving a variadic number of terms can be matched:

```
argmin(sum(S1(x).^2)+sum(S2(x).^2)+sum(S3(x).^2), x)
```

A simple reduction then allows the problem solver to handle a variable number of terms.

```
reduction (x,...S:vec[[??->??]]) -> argmin(sum([sum(S(x).^2)]),x) = solver(S)
```

Often, terms of the cost function have different weights. By defining a variadic weight parameter vector, weights can be added:

```
argmin(0.5*sum(S1(x).^2)+0.2*sum(S2(x).^2)+0.1*sum(S3(x).^2), x)
reduction (x,...lambda,...S:vec[[??->??]]) -> argmin(sum(lambda.*[sum(S(x).^2)]),x) = solver(S,lambda)
```

Notice that lambda.*[sum(S(x).^2)] is an elementwise product of two vectors. The reduction pattern matching is able to extract the coefficients corresponding to each term and collect them in the vector lambda.

Solving L2 problems using conjugate gradient

To solve L2 problems such as $\arg\min_{\mathbf{x}} ||\mathbf{y} - \mathbf{W}(\mathbf{x})||^2$ and $\arg\min_{\mathbf{x}} ||\mathbf{y} - \mathbf{x}||^2 + \lambda||\mathbf{W}\mathbf{x}||^2$, the iterative conjugate gradient can be used, requiring no explicit matrix-forms for $\mathbf{W}$ (or any other involved matrix).

```
function x = conjugate_gradients(y, FHF, num_iterations=10)
    x = 0
    r = y
    p = r
    new_err = dotprod(r, r)
    tol = 1e-15

    for it=1..num_iterations
        err=new_err
        A_p = FHF(p)
        alpha = err / dotprod(p, A_p)
        x = x + alpha * p
        r = r - alpha * A_p
        new_err = dotprod(r, r)
        if (new_err/numel(x) < tol)
            break
        endif
        beta = new_err / err
        p = r + beta * p
    endfor
```



```
endfunction
```

The above implementation relies on the function FHF which implements the concatenation $\mathbf{F}^H\mathbf{F}$. In the convex solver, only the function F is specified. To obtain $\mathbf{F}^H$, algorithmic adjoint calculation (AAC) is used. Once $\mathbf{F}$ and $\mathbf{F}^H$ are known, their product readily follows by function composition. Below, conjugate gradient wrapper implementations are given for different forms of the L2 optimization problem.

```
function x = conjugate_gradient_auto_adjoint(y : cube, F : [?? -> ??], num_iterations = 10)
    {!auto_specialize}
    FH = z -> $adjoint(F(z), z)
    x = conjugate_gradients(FH(y), (x -> FH(F(x))), num_iterations)
endfunction

function x = conjugate_gradient_regularization_auto_adjoint(y : cube, W : [cube -> ??], A : [cube ->
     ??], lambda, num_iterations = 10)
    {!auto_specialize}
    WH = y -> $adjoint(W(y), y)
    AH = y -> $adjoint(A(y), y)
    x = conjugate_gradients(WH(y) + lambda * AH(y), (x -> WH(W(x)) + lambda * AH(A(x))),
        num_iterations)
endfunction

function x = conjugate_gradient_multireg_auto_adjoint(y : cube, W : [cube -> ??], A : vec[[cube ->
     ??]], lambda, num_iterations = 10)
    {!auto_specialize}
    WH = y -> $adjoint(W(y), y)
    AH = vec[function](numel(A))
    AHA = vec[function](numel(A))
    for i=0..numel(A)-1
        {!unroll times=numel(A)}
        AH[i] = y -> $adjoint(A[i](y), y)
        AHA[i] = x -> AH[i](A[i](x))
    endfor
    x = conjugate_gradients(WH(y) + lambda * sum([AHA(y)]), _
                          (x -> WH(W(x)) + lambda * sum([AHA(x)]), num_iterations)
endfunction

reduction (x, y, W) -> argmin(sum((y - W(x)).^2), x) = conjugate_gradient_auto_adjoint(y,W)
reduction (x, y, W, A, lambda) -> argmin(sum((y - W(x)).^2) + lambda .* sum(A(x).^2), x) =
     conjugate_gradient_regularization_auto_adjoint(y,W,A,lambda)
reduction (x, y, W, ...A : vec[[??->??]], lambda) -> _
    argmin(sum((y - W(x)).^2) + lambda.*sum([sum(A(x).^2)]),x) =
        conjugate_gradient_multireg_auto_adjoint(y,W,A,lambda)
```

The code attribute {! auto_specialize } specifies that the function needs to be specialized for every function F that is passed to it. This is required for AAC, so that the function is known at compile time. {! unroll times=numel(N)} entirely expands the loop based on the loop iteration count that is known at compile-time. By expanding the loop, we can guarantee that the adjoint is calculated for every function A[i].

Proximal Operator Framework

To deal with optimization problems involving convex cost functions that are not L^2, the proximal operator framework can be used. Many convex optimization problems are not differentiable, therefore, in essence gradient methods (and hence algorithmic differentiation) can not directly be used. Splitting methods proceed by splitting the objective functions into several sub-problems for which an efficient optimization method exists and that are therefore easier to solve. In GASPACHO, the splitting can be elegantly mapped onto a modular approach involving reductions. For example, splitting may result in an L^2-problem and an L^1-problem. The L^2 problem can then be solved using the conjugate gradient method from the previous section. For the L^1 problem, separate solvers need to be defined.

The proximal operator is the solution to the miminization problem:

$$\arg\min_{\mathbf{x}} f(\mathbf{x}) + \frac{1}{2}||\mathbf{y} - \mathbf{x}||^2$$

where $f(\mathbf{x})$ is a convex function. For several functions, a simple and efficient proximal operator exists. Some definitions are given below.

```
rectify = x -> x .* (x >= 0) + maxvalue(scalar) * (x < 0)

pointwise_l2_inverse = (y,lambda) -> 1.0/(lambda + 1) * y
pointwise_l1_inverse = (y,lambda) -> sign(y).*max(0.0, abs(y) - lambda)
pointwise_proxmax_inverse = (y,lambda) -> y.*(abs(y)<lambda) + _
                                          sign(y).*lambda.*(abs(y)>=lambda).*(abs(y)<2*lambda) + _
                                          sign(y).*(abs(y)-lambda).*(abs(y)>=2*lambda)
pointwise_proxmax_inverse = (y,lambda) -> max(0, y - lambda)

reduction (x, y, W) -> argmin(sum((y - W(x)).^2), x) = conjugate_gradient_bridge(y,W)
reduction (x, y, W, A, lambda) -> argmin(sum((y - W(x)).^2) + lambda .* sum(A(x).^2), x) =
    conjugate_gradient_reg_bridge(y,W,A,lambda)
reduction (x, y, W, ...A : vec[[??->??]], lambda) -> _
                                  argmin(sum((y - W(x)).^2) + lambda.*sum([sum(A(x).^2)]),x) =
                                      conjugate_gradient_reg2_bridge(y,W,A,lambda)
reduction (x, y, lambda) -> argmin(sum((y - x).^2) + lambda .* sum(x.^2), x) = pointwise_l2_inverse(y,
    lambda)
reduction (x, y, lambda) -> argmin(sum((y - x).^2) + lambda .* sum(abs(x)), x) = pointwise_l1_inverse(
    y,lambda)
reduction (x, y, lambda) -> argmin(sum((y - x).^2) + lambda .* max(abs(x)-lambda,0),x) =
    pointwise_proxmax_inverse(y, lambda)
reduction (x, y, lambda) -> argmin(sum((y - x).^2) + lambda .* rectify(x),x) =
    pointwise_proxrectify_inverse(y, lambda)
```

Splitting method

Several splitting methods to decompose the convex optimization problem into L1 and L2 subproblems exist. An implementation of two of these algorithms is given below.

Split-Bregman algorithm

The Split-Bregman method (also sometimes known as split-augmented Lagrangian method, or alternating direction method of multipliers, ADMM) performs alternating update steps, solving both L1 and L2 problems. The implementation relies on

the argmin() function, for which the necessary definitions are given in the previous sections.

```
function x = split_bregman_solver(x0, y0,
    W : [?? -> ??], Phi : [?? -> ??], psnr : [?? -> scalar], max_iter = 10, lambda = 1e-6)

    x = x0
    d = Phi(zeros(size(x)))
    b = Phi(zeros(size(x)))
    mu = 0.01 % penalizer param

    for iter=1..max_iter
        print "iter ",iter, " psnr=", psnr(x), " dB"

        % Solve the L2-problem
        x = argmin(sum((y0 - W(x)).^2) + mu .* sum(Phi(x).^2), x)

        % Solve the L1-problem
        d = argmin(sum((b - Phi(x)).^2) + lambda/mu .* sum(abs(b)), x)
        b += (Phi(x) - d)
    endfor
endfunction
```

Simultaneous-direction method of multipliers (SDMM)

The SDMM method generalizes the split-Bregman/ADMM method to a variable number of L1 terms. Some of the subproblems can be solved in parallel, e.g., via multi-GPU processing ({! sched gpu_index=X} code attributes).

```
function x = SDMM_solver_multi(x0, y0,
    C : [?? -> ??], W : [?? -> ??], Phi : vec[[?? -> ??]], psnr : [?? -> scalar], lambda : vec,
        max_iter = 50)
    {!auto_specialize}

    x = x0
    W_H = X -> $adjoint(W(X),X)
    C_H = X -> $adjoint(C(X),X)

    % Initialize temporary variables
    K = numel(Phi)
    y1 = copy(y0)
    y2 = cell(K)
    z1 = zeros(size(y0))
    z2 = cell(K)
    Phi_H = vec[function](K)
    Phi_H_Phi = vec[function](K)
    {!interpreted for}
    for k1=0..numel(Phi)-1
        {!unroll times=numel(Phi)}
        y2[k1] = Phi[k1](x0)
        z2[k1] = Phi[k1](zeros(size(x0)))
        Phi_H[k1] = X -> $adjoint(Phi[k1](X),X)
    endfor
    for k=0..K-1
        Phi_H_Phi[k] = x -> Phi_H[k](Phi[k](x))
    endfor
    gamma = 10.0
```



```
    for iter=1..max_iter
        print "iter ",iter, " psnr=", psnr(x), " dB"

        {!sched gpu_index=0}
        u = W_H(C_H(C(y1 - z1)))
        {!sched gpu_index=1}
        for k=0..K-1
            u += Phi_H[k](y2[k] - z2[k])
        endfor
        {!sched gpu_index=0}
        x = conjugate_gradients(u, (r -> W_H(C_H(C(W(r)))) + vecfunc_sum(Phi_H_Phi, r)))

        % Proximal operator 1
        s1 = W(x)
        y1 = (gamma * y0 + s1 + z1) / (gamma + 1)
        z1 += (s1 - y1)

        % Proximal operators 2, 3, ...
        {!sched gpu_index=1}
        for k=0..K-1
            s2 = Phi[k](x)
            y2[k] = argmin(sum(((s2 + z2[k]) - x).^2) + (gamma*lambda[k]).*sum(abs(x)), x)
            z2[k] += (s2 - y2[k])
        endfor
    endfor
endfunction

reduction (x, y, W, ...lambda, ...S) -> argmin(sum((y - W(x)).^2) + sum([lambda.*sum(abs(S(x)))]), x)
    = _
    SDMM_solver_multi(x0, y, (x -> x), W, S, plot_result, lambda, max_iter:=10)

reduction (x, y, C, W, ...lambda, ...S) -> argmin(sum(C(y - W(x)).^2) + sum([lambda.*sum(abs(S(x)))]),
     x) = _
    SDMM_solver_multi(x0, y, C, W, S, plot_result, lambda, max_iter:=50)
```

The last two reductions match expressions for convex optimization problems that the SDMM algorithm can solve. The SDMM solver then relies on its turn on the argmin function. This approach exploits the structure of the optimization problem and generates a specialized implementation solely based on the objective function.

Below this is illustrated using an image restoration (joint denoising and deblurring) example:

```
[S : [cube -> vec[??]], S_H : [vec[??]->cube]] = build_dst2d(size(y,0..2), 8, 4)
[S2 : [cube -> vec[??]], S2_H : [vec[??]->cube]] = build_tvtransform()

reduction x -> $adjoint(S(x),x) = S_H(x)
reduction x -> $adjoint(S2(x),x) = S2_H(x)

x : cube
x = argmin(sum(C(y - W(x)).^2) + (100.0.*sum(abs(S(x))) + 1000.0.*sum(abs(S2(x)))), x)
<!--*-->
```

S and S_H denote respectively the forward and adjoint shearlet transform. These functions are specified by a direct algorithm. Similarly, S2 and S2_H denote the forward and adjoint total variation transform (sometimes called gradient transform). Therefore, it is necessary to inform the AD framework that the adjoints of S and S2 are respectively S_H and S2_H, resulting in the two reduction definitions.

The image restoration problem is then simply solved in one line of code, using an argmin expression.

Note that type of x here needs to be predeclared due to self-reference (see x : cube).

Variadic parameter expansion

Often it is useful to pass parameters to the algorithms. This can be achieved by extending the argmin function to take an extra parameter variable, containing an object with the corresponding parameter values. Variadic parameter expansion allows the object to be expanded into a set of function parameters, as indicated below:

```
options={num_iterations:=10,alpha:=1e-3}
my_func(...options)
```

By applying this technique, parameters can be set specific to the problem that is being solved. The parameters are then propagated to the individual argmin implementations.

Limitations

The convex optimization framework currently has a few limitations:

- Currently there is no direct syntax support for boundary conditions (e.g. positivity, negativity conditions). Instead, boundary conditions could be passed as parameters to the optimization algorithms. This is part of ongoing work.
- The reduction pattern matching currently only supports a limited number of algebraic manipulations. For example, for sums involving a variable number of terms, the matching supports term reordering. Scaling of the cost function is not handled by the matching: for example the solution to argmin(2*x^2, x) is identical to the solution of argmin(x^2, x), although the matching is not automatic. This can be solved by defining an explicit reduction for this scaling parameter.
- For multiplications A*B, the reduction system cannot assume commutivity, unless the involved variables are all declared as scalar values (A: scalar , B: scalar). When the parameters of the variables in the reduction are not set, the reduction (A,B)->A*cos(B) will not match (A,B)->cos(B)*A even though for a specific instantiation (for example A:mat, B: scalar), the result is the same. The solution is to indicate variable types so that the reductions can be applied in wide enough circumstances. In some cases, it is therefore necessary to define the reduction multiple times to accomodate this scenario.

