# An Automatic Differentiation Extension for R, and its Implementation in pqR

Radford M. Neal, University of Toronto

Dept. of Statistical Sciences and Dept. of Computer Science

Vector Institute Affiliate

`http://www.cs.utoronto.ca/∼radford`

`http://radfordneal.wordpress.com`

`http://pqR-project.org`

RIOT2019, Toulouse

## The Need for Automatic Differentiation

Derivatives (gradients) are crucial for efficient implementation of many statistical methods:

- Maximum likelihood estimation, using gradient-based optimization. (Plus standard errors for the MLE are found from second derivatives.)
- Neural network training by gradient descent, optimizing the likelihood or some other criterion.
- Markov chain Monte Carlo using gradients (eg, Hamiltonian Monte Carlo).

Traditionally, researchers have spent much time manually figuring out how to compute derivatives. Finding derivatives automatically instead greatly facilitates exploration of new methods.

Automatic differentiation implementations exist are are being developed for TensorFlow, Stan, Python, Julia, and Swift.

## Approaches to Automatically Computing Derivatives

**Numerical differentiation:** Approximation using finite differences. For derivatives w.r.t. $N$ values, requires evaluating the expression at least $N+1$ times. Implemented by R's `numericDeriv` function.

**Symbolic differentiation:** Exact apart from roundoff error. Hard to implement for expressions more complex than a composition of simple functions. Implemented by R's `deriv` function.

**Run-time use of the chain rule:** Exact apart from roundoff error. Comes in two flavours:

- **Forward mode:** Compute derivatives of sub-expressions at the same time as the sub-expression's value. Fast if derivatives are computed w.r.t. only a few values.
- **Reverse mode:** Record how values are computed. Once the final value is known, compute derivatives by working backward. Fast if the final value is scalar (or is at least of low dimension).

# An R Language Extension for Automatic Differentiation

## Summary and Objectives

- New language constructs are introduced to allow convenient use of automatic differentiation.
- The aim is for gradients to be efficiently computable for R expressions and functions written using most common language features, in any reasonable style, without previous thought of derivatives.
- The tracking of gradients associated with values is invisible at the R level, ensuring that functions behave the same when gradients are requested as when not.
- The strategy used to integrate automatic differentiation into the interpreter can support either forward or reverse mode differentiation. A hybrid approach is currently used, though for some operations only forward mode is implemented at this time.

## The `with gradient` Construct

A new `with gradient` language construct delivers both the value of an expression, and its gradient with respect to a specified set of variables:

```
> with gradient (a=1.2) sin(3*a) # gradient is 3*cos(3*1.2)
[1] -0.4425204
attr(,"gradient")
[1] -2.690275
```

The gradient is attached as a `"gradient"` attribute – an existing convention used by the standard `numericDeriv` and `nlm` functions.

Gradients can be with respect to several variables, giving a list:

```
> r <- with gradient (a=3, b=8) a*b + a^2
> attr(r,"gradient") $ a
[1] 14
> attr(r,"gradient") $ b
[1] 3
```

## An Example Function

```
# Find the distance travelled by a projectile launched on level
# ground with initial velocity (vx,vy), with no air resistance.

distance_travelled <- function (vx, vy, dt=0.0001, g=9.8) {
    x <- y <- 0
    repeat {
        last_x <- x
        last_y <- y
        x <- x + vx*dt
        y <- y + vy*dt
        if (y < 0)  # return impact x location, interpolating
            return ((x*last_y - last_x*y) / (last_y-y))
        vy <- vy - g*dt
    }
}
```

## Using Automatic Differentiation with this Example

Let's use this function to find the angle of launch maximizing distance (for fixed initial speed), using `nlm`, with derivatives found by `with gradient`.

```
> nlm (function (a)
+         with gradient (a) -distance_travelled (cos(a), sin(a)),
+       0) $ estimate * 180/pi
[1] 44.99701
```

Note that derivatives are automatically tracked through the function call, assignments, loop, and if statement. Derivatives aren't tracked when not needed — eg, `y < 0` fetches `y` without its derivative.

The `distance_travelled` function might have been written with no thought of differentiation — though it is essential to this example that it interpolates the impact position; not doing so makes it piecewise constant in `vy`, with a zero derivative w.r.t. `vy` for any value of `dt`.

## Language Constructs that Track Gradients

Gradients can be tracked for expressions with real or list values that involve

- Arithmetic operators (`+`, `-`, `*`, `/`, `^`).
- Matrix operations (`%*%`, `crossprod`, `tcrossprod`, `t`).
- Subsetting operators (`$`, `[.]`, `[[.]]`).
- Most mathematical functions (eg, `sin`) and functions involving distributions (eg, `pnorm`).
- Many random generation functions (eg, `dnorm`) — based on how the value would change with the seed fixed.
- Many other builtin/primitive functions (eg, `rep`, `c`, `list`, `lapply`).
- Calls of functions, including S3 methods.
- Assignments to local variables.
- Subset assignment (`$<-`, `[<-`, `[[<-`).
- `if`, `while`, and `repeat` expressions.

# Language Constructs that Don't Track Gradients

Gradients are currently not tracked for

- Some functions for which they just haven't been implemented yet (eg, `aperm`, `cov`).
- Values stored as attributes.
- Calls of S4 methods.
- Complex-valued expressions.

Gradient tracking in the above situations might be implemented in future.

Gradient tracking will probably not be implemented for

- Assignments with `<<-`.
- Storing values in environments with `$<-`.

These have semantic issues with gradients w.r.t. no-longer-existing variables.

## Form of the Gradient

The gradient of $A$ with respect to $B$ is

- A scalar, if $A$ and $B$ are both scalar.
- A vector, is $A$ is scalar and $B$ is a vector.
- An $n$-by-$m$ matrix (the "Jacobian"), if $A$ is a vector of length $n$ and $B$ is a vector of length $m$.
- A list, if $A$ is a scalar or vector and $B$ is a list. Each element of the list gives the gradient with respect to the corresponding element of $B$.
- A list, if $A$ is a list and $B$ is a vector or scalar. Each element of the list gives the gradient of the corresponding element of $A$.
- A list of lists, if both $A$ and $B$ are lists — the upper levels of the list correspond to elements of $B$ (perhaps recursively), the lower levels to elements of $A$ (perhaps recursively).

Dimensions for $A$ and $B$ are ignored — the Jacobian is always a matrix.

## The `compute gradient` Construct

When desired, the method for computing the gradient of an expression can be specified explicitly. For example, rather than

```
sigmoid <- function (x) 1 / (1+exp(-x))
```

for which gradients will be computed automatically, one could instead write

```
sigmoid <- function (x)
    compute gradient (x) { v <- 1 / (1+exp(-x)); v }
    as v * (1-v)
```

Using the already-computed function value, `v`, may be more efficient.

The `compute gradient` construct could be also useful in cases where the interpreter does not know how to compute the gradient — for example, when the function is computed by an external C or Fortran routine.

More than one variable may be specified:

```
compute gradient (x, y) { s <- x^2+y^2; s^2 } as 4*s*x, 4*s*y
```

## Specifying the Gradient in `compute gradient`

The expression for computing the gradient may return the gradient in the same form as it is returned by `with gradient`.

**But**. . . a diagonal Jacobian matrix can be specified by a vector of just the diagonal elements. This is convenient (and faster) for vectorized functions — eg, the earlier `sigmoid` example works this way when `x` is a vector.

**Also**. . . the gradient expression can produce a function that is called to compute the gradient, rather than the gradient itself. If this function has `left` and/or `right` arguments, it may be called to compute the product of the Jacobian with a matrix on the left or right — which it may be able to do efficiently without computing the full Jacobian. (Though currently the function is always called without a `left` or `right` argument.)

Gradient computations in `compute gradient` are skipped when it is evaluated in a context where the corresponding gradient is not needed.

## Using `track gradient` and `gradient_of`

An alternative to `with gradient` when you don't want the gradient attached as an attribute is to use `track gradient` and `gradient_of`.

For example:

```
> track gradient (a=7) {
+   r <- a^2
+   list (value = r, grad = gradient_of(r))
+ }
$value
[1] 49


$grad
[1] 14
```

You can call `gradient_of` for any expression when inside (dynamicallly) a `track gradient` or `with gradient` construct.

## No Higher-Order Derivatives (Yet)

It's permitted to nest `with gradient` or `track gradient` constructs.

But this won't let you compute second or higher-order derivatives:

- Attributes, including a `gradient` attribute, don't record gradient information.
- The value of `gradient_of` doesn't have gradient information.

So, for example, we get zero when we try

```
> track gradient (a=7)
+   gradient_of (track gradient (a) gradient_of(a^3))
[1] 0
```

It's unsurprising that there's no trick that does this — getting higher-order derivatives would require code to compute them for primitive functions.

## How Higher-Order Derivatives Might be Handled

Higher-order derivatives could be implemented by just relaxing these limitations.

The `gradient_of` function could come with gradient information for *outer* `with gradient` or `track gradient` constructs (but not the innermost one).

Then we'd see

```
> track gradient (a=7)
+   gradient_of (track gradient (a) gradient_of(a^3))
[1] 42
```

Some syntactic sugar might make this more convenient.

Getting higher-order derivatives from nesting `with gradient` or `track gradient` constructs would ensure that the primitives know what order of derivatives they need to compute (based on the depth of nesting).

## Explicit Reverse Mode with `back gradient`

The pqR implementation tries to automatically use reverse mode differentiation when it's beneficial, but it doesn't always do so yet.

One can do reverse mode explicitly with `back gradient`:

```
L <- as.list(seq(0,1,length=11))
with gradient (L) {   # tracks gradient w.r.t. 11 elements of L
  p <- 0
  for (i along L)     # 'along' is a pqR extension
    p <- p + i*L[[i]]
  p^2+p^3+p^4+p^5     # every operation computes derivatives
}                     #   w.r.t. all 11 elements of L
with gradient (L) {   # compute same result more efficiently...
  p <- 0
  for (i along L)
    p <- p + i*L[[i]]
  back gradient (p)   # operations in the expresson below
    p^2+p^3+p^4+p^5   #   compute derivative w.r.t. p only, then
}                     #   chain rule gives gradient w.r.t. L
```

Implementing Automatic Differentiation in pqR

## The pqR implementation of R

This project is part of my pqR implementation of R (`pqR-project.org`). This is a fork of the R Core implementation with many improvements. Some changes relevant to automatic differentiation are:

- A new parser makes introducing new language constructs easier.
- Interpretive speed has been greatly improved, eliminating any need to use the "byte-code compiler". Implementation by direct interpretation makes extending the language (eg, for automatic differentiation) easier.
- Multiple processor cores can be automatically used to parallelize numerical computations. This may be especially useful when gradients can be computed in parallel with the function value.
- An internal mechanism allows expressions to be evaluated asking for a "variant" result — crucial for how automatic differentiation is implemented in this project.

## Environments Storing Gradient Information

Environments in which variables should store gradient information are marked with a `STORE_GRAD` bit (in `sxpinfo`). These include

- The environment created for the body of a `with gradient` or `track gradient` construct.
- The environment created when a function is called from an environment with `STORE_GRAD` set.
- Environments set up for S3 methods called from an environment with `STORE_GRAD` set.

An environment created for `with gradient` or `track gradient` will also contain information on the gradient variables it declares.

# When Gradient Information is Requested

The internal “variant result” mechanism in pqR is used to control whether evaluation of an expression comes with a request for gradient information.

Contexts where gradient information is requested include

- The body of a `with gradient` construct — so the gradient can be attached to the value as a `gradient` attribute.
- The argument of `gradient_of`.
- The right-hand-side of a local assignment, when the local environment has `STORE_GRAD` set — so the gradient can be stored along with the variable’s value.
- Arguments of operators and functions that can compute gradients, such as `+` and `sin`, but not `floor`.
- Whichever branch of an `if` statement is taken (but not the condition), if the gradient of the whole expression has been requested.

## Where Gradient Information is Stored

Information on the gradients for a value is stored in the attribute field of the binding cell or promise that references the value. This includes

- Binding cells for the gradient variables created by `with gradient`, `track gradient`, or `back gradient` (for which the gradient is initially the identity).
- Binding cells for variables created in an environment with `STORE_GRAD` set, if their value has gradient information.
- Promises for arguments of functions called from environments with `STORE_GRAD` set. These promises are also marked with `STORE_GRAD`. When forced, if their value has gradient information, it is stored in the attribute of the promise.
- Cells in argument lists created for `BUILTIN` primitives that handle gradients, when the argument value has gradient information.

## Format of Stored Gradient Information

The gradient information for a value records its gradient with respect to one or more variables in `with gradient`, `track gradient`, or `back gradient` constructs (which may be nested).

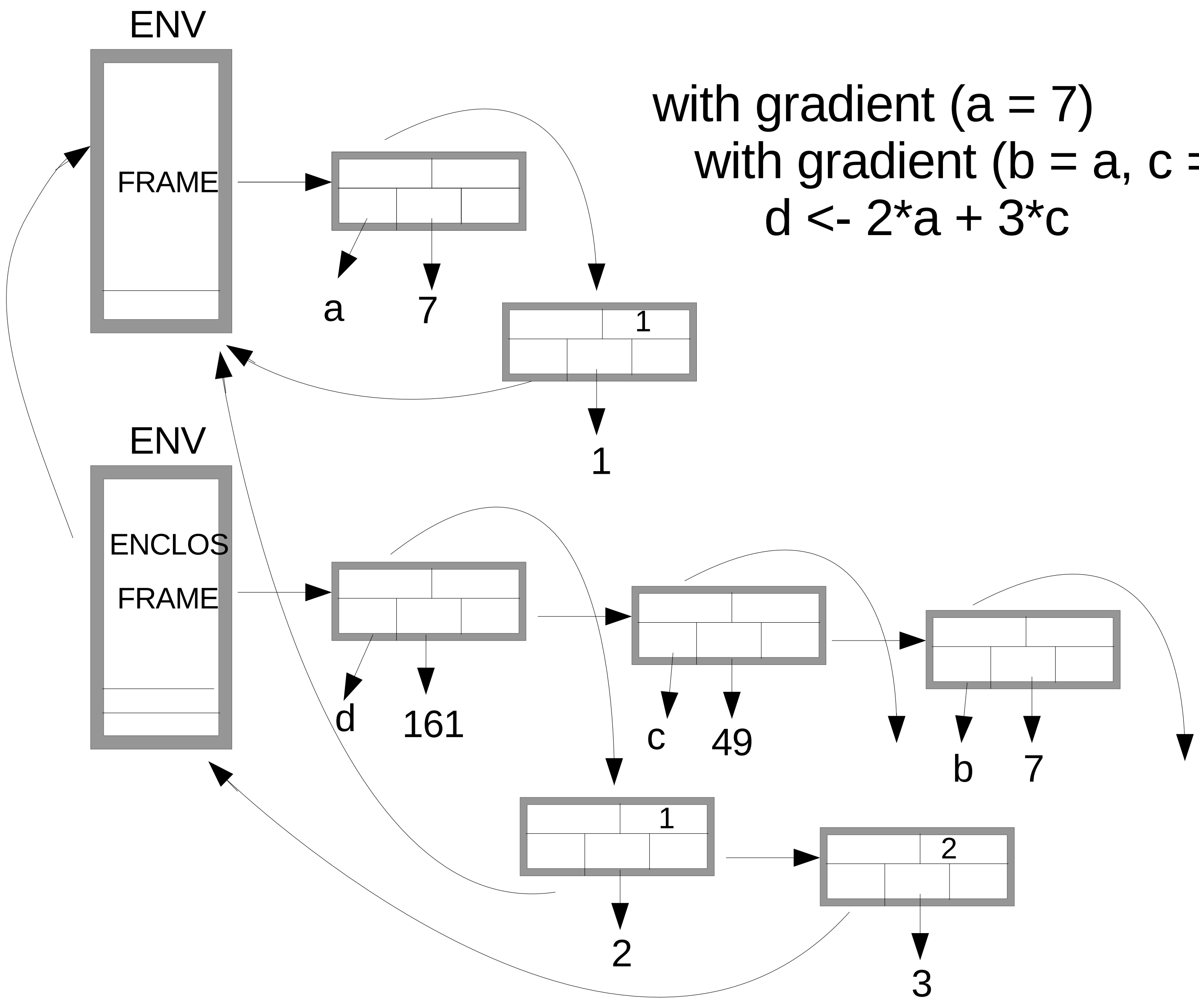
Such a gradient variable is identified by the environment for the body of the construct in which it is declared, and its index in the list of gradient variables that construct declares.

These gradients are kept in linked CONS cells, with the CAR of the cell holding the gradient, the TAG of the cell pointing to the environment, and the top byte of the "gp" field holding the index.

For gradients with respect to list values, the gradient will be a corresponding list, flagged with a `GRAD_WRT_LIST` bit (in `sxpinfo`), perhaps to more than one level.

**At this point....** the implementation as described could support various representations of the gradients with respect to numeric values, without affecting the interface to the rest of the interpreter.

# A Picture of All This...



```
with gradient (a = 7)
   with gradient (b = a, c = a^2)
      d <- 2*a + 3*c
```

## The Need for Compact Gradient Representions

We now need to represent the gradient of a value with respect to some numeric value (maybe not scalar) — which is either the value of a gradient variable, or an element of a list that is the value of a gradient variable.

A naive implementation might represent these gradients as they would be if attached as a `gradient` attribute, or returned by `gradient_of`.

But this would be very inefficient. Consider

```
  M <- matrix(something,1000,1000)
  val <- with gradient (M) sum(M^2)
```

The gradient attribute attached to `val` will have 1000000 elements. But in a naive implementation, `M^2` will have a 1000000-by-1000000 Jacobian matrix (with $10^{12}$ elements) associated with it. Indeed, at the start of the `with gradient` construct, `M` itself will have a 1000000-by-1000000 Jacobian — set to the identity matrix!

We need to often represent gradients more compactly.

## Some Compact Gradient Representations

- Diagonal Jacobians, represented by only the diagonal elements, or by a single number if a multiple of the identity matrix.
- "One-in-row" Jacobians with at most one non-zero element in each row. Useful for instance in `with gradient (a) { a[2] <- a[5]; ... }`
- Scaled Jacobians — diagonal matrix times another matrix. Need only update the diagonal factor when doing `x <- 3*sin(x)`.
- Matrix product Jacobians — they have lots of zeros, so better to not represent them explicitly.
- Reverse-mode Jacobians — simplest form is a Jacobian times another, which might itself be a Jacobian times another, etc. When a full Jacobian is finally needed, the factors can be multiplied in whatever order is most efficient.

This aspect of the implementation is a work in progress.

## Testing Performance

Here are some silly functions for performance testing:

```
f <- function (x) {
  s <- 1
  for (i in 1:100) s <- s + (x^2+1)/s
  4 * cumsum(s)^2
}

g <- function (x,n) {
  for (j in 1:n) r <- f(x)
  r
}
```

I've so far been concentrating on completing full support for automatic differentiation, and on eliminating drastic performance issues, rather than on detailed optimizations. So performance in the following tests could probably be improved fairly easily.

## Performance Results with No Gradients

Comparison when no gradients are requested:

With pqR-2019-02-19 (no autodiff support):

```
> x <- c(7,4,9,5,3)
> print (system.time (print(g(x,10000))))
[1]  50162.08 119580.26 422851.01 646335.83 802707.50
   user  system elapsed
  0.262   0.000   0.262
```

With pqR-2019-07-05 (preliminary version with autodiff):

```
> x <- c(7,4,9,5,3)
> print (system.time (print(g(x,10000))))
[1]  50162.08 119580.26 422851.01 646335.83 802707.50
   user  system elapsed
  0.286   0.004   0.290
```

The slowdown is about 10%, but changes of 5% or more can happen due to random factors (eg, memory layout affecting cache performance).

## Performance Results with Gradients (Scalar)

```
> x <- 7
> print (system.time (print(g(x,10000))))
[1] 50162.08
   user  system elapsed
  0.099   0.000   0.099
> print (system.time (print(with gradient (x) g(x,10000))))
[1] 50162.08
attr(,"gradient")
[1] 16822.64
   user  system elapsed
  0.318   0.004   0.322
> print (system.time (print(numericDeriv (quote(g(x,10000)), "x"))))
[1] 50162.08
attr(,"gradient")
         [,1]
[1,] 16822.64
   user  system elapsed
  0.195   0.000   0.195
```

## Performance Results with Gradients (Vectors)

```
> x <- x0 <- c(7,4,9,5,3)
> print (system.time ((g(x,10000))))
   user  system elapsed
   0.28    0.00    0.28
> print (system.time ((with gradient (x) g(x,10000))))
   user  system elapsed
   0.63    0.00    0.63
> print (system.time ((numericDeriv (quote(g(x,10000)), "x"))))
   user  system elapsed
  1.649   0.000   1.650
> x <- rep(x0,20)
> print (system.time ((g(x,10000))))
   user  system elapsed
  0.437   0.000   0.437
> print (system.time ((with gradient (x) g(x,10000))))
   user  system elapsed
  2.636   0.000   2.637
> print (system.time ((numericDeriv (quote(g(x,10000)), "x"))))
   user  system elapsed
 44.533   0.000  44.533
```

## Language Extensions to Make Using Gradients Easier

## Easier Access to Attributes

Since `with gradient` attaches the gradient as an attribute of the value, more convenient access to this attribute would make code easier to read. Of course, this would also benefit other code that accesses attributes.

**Proposal:** Allow the `@` operator to access (or change) attributes of any object, not just slots of S4 objects. For example:

```
> r <- with gradient (x) f(x)
> r @ gradient        # same as attr(r,"gradient")
>
> v <- c(3,1,5,2)
> v @ dim <- c(2,2)  # same as dim(r) <- c(2,2)
```

**Benefit:** Postfix subsetting operators are easier to read, since the variable is in a highly visible position on the left, not hiding inside an argument list.

Since S4 slots are implemented as attributes, there is no conflict with the existing use of `@`. Only downside is that debugging S4 code might be harder, since accessing a non-exisent slot no longer gives an immediate error.

## The Need for Arithmetic on Lists

It is currently an error to apply arithmetic operators or math functions to lists — only vectors or derived types such as matrices are allowed.

But parameters of complex statistical models are most naturally represented as lists of parameters of various kinds. Optimizing such parameters by gradient methods, or sampling them by MCMC methods, naturally involves arithmetic on these lists.

At present, this can be done using the `unlist` and `relist` functions, but this is both slow and inconvenient.

**Proposal:** Implement extensions so that one can do a gradient ascent step on parameters represented as lists like so:

```
loglik <- with gradient (params) log_likelihood(params,data)
params <- params + stepsize * loglik@gradient
```

## How Arithmetic on Lists Would Work

```
> v <- list (x=3, y=c(-4,5))
> w <- list (x=2, y=c(1,2))
>
> abs(v)  # Math functions are applied recursively to list elements
$x
[1] 3


$y
[1] 4 5


> v + w   # Lists with same structure can be added, multiplied, etc.
$x
[1] 5


$y
[1] -3 7
```

```
> v <- list (x=3, y=c(-4,5))
>
> v * 10  # A scalar can operate on a list - applied to each element
$x
[1] 30


$y
[1] -40 50


> v * list (x=2, y=10)  # OK for a scalar to appear at a lower level
$x
[1] 6


$y
[1] -40 50
```

This extension is probably best implemented at the C level, within the arithmetic primitives, but it could also be done at the R level, after a small change to make arithmetic primitives dispatch to methods for `list`.

## Generalizing Dimension to Shape

Once arithmetic can be done on lists, it makes sense to look at the “shape” of a vector, matrix, array, or list.

```
> shape (c(3,4))
[1] 2
> shape (matrix(0,3,4)
[1] 3 4
> shape (list (x=c(3,4), y=list(1,2), z=matrix(0,4,5))
$x
[1] 2
$y
$y[[1]]
[1] 1
$y[[2]]
[1] 1
$z
[1] 4 5
```

# Using Shape for Random Generation

One use of the shape concept would be to generate random structures with a given shape. Such a facility would be very useful for MCMC proposals.

```
> rnorm (list(x=c(2,2),y=4))
$x
         [,1]       [,2]
[1,] 1.859454 -0.3312653
[2,] 1.380187 -1.2248318
$y
[1]  0.9131368 -0.3557320  0.7749956 -0.5363131

> rnorm (list(x=2,y=2), list(x=100,y=c(10,20)))
$x
[1]  99.55952 100.10348
$y
[1]  9.777663 19.232496
```