# The Monad.Reader Issue 21

by Amy de Buitléir ⟨amy.butler@ericsson.com⟩
and Michael Russell ⟨mrussell@ait.ie⟩
and Mark Daly ⟨mdaly@ait.ie⟩
and Felipe Zapata ⟨felipe.zapata@edu.uah.es⟩
and Angel J. Alvarez ⟨a.alvarez@uah.es⟩

March 16, 2013

Edward Z. Yang, editor.

# Contents

# Editorial

by Edward Z. Yang ⟨ezyang@cs.stanford.edu⟩

This issue, we bring to you two articles which tie Haskell together with other domains outside of the ordinary Haskell experience. One combines Haskell with machine learning; the other combines Haskell with computational quantum chemistry. These articles don't use the most sophisticated type-level programming or Kan extensions; however, I do think they offer a glimpse at the ways practitioners in other fields use Haskell. I think it's quite interesting to see what kinds of problems they care about and what features of Haskell they lean on to get things done. I hope you agree!

# A Functional Approach to Neural Networks

by Amy de Buitléir ⟨amy.butler@ericsson.com⟩
and Michael Russell ⟨mrussell@ait.ie⟩
and Mark Daly ⟨mdaly@ait.ie⟩

*Neural networks can be useful for pattern recognition and machine learning. We describe an approach to implementing a neural network in a functional programming language, using a basic back-propagation algorithm for illustration. We highlight the benefits of a purely functional approach for both the development and testing of neural networks. Although the examples are coded in Haskell, the techniques described should be applicable to any functional programming language.*

## Back-propagation

*Back-propagation* is a common method of training neural networks. After an input pattern is propagated forward through the network to produce an output pattern, the output pattern is compared to the target (desired) pattern, and the error is then propagated backward. During the back-propagation phase, each neuron's contribution to the error is calculated, and the network configuration can be modified with the goal of reducing future errors. Back-propagation is a supervised training method, so the correct answers for the training set must be known in advance or be calculable. In this paper, we use a simple "no-frills" back-propagation algorithm; this is sufficient for demonstrating a functional approach to neural networks.

# Neural networks

## An artificial neuron

The basic building block of an artificial neural network is the neuron, shown in Figure 1. It is characterized by the elements listed below [1].

- a set of inputs $x_i$, usually more than one;
- a set of weights $w_i$ associated with each input;
- the weighted sum of the inputs $a = \Sigma x_i w_i$;
- an activation function $f(a)$ which acts on the weighted sum of the inputs, and determines the output;
- a single output $y = f(a)$.

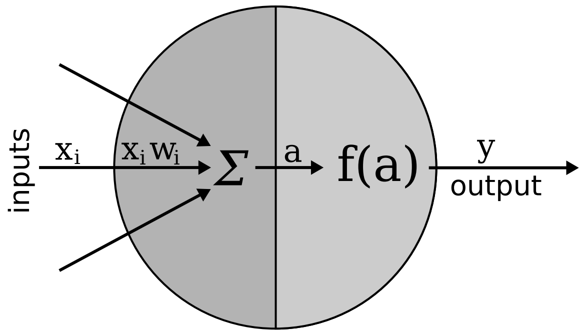

**Figure 1:** An artificial neuron.

## A simple network

The most common type of artificial neural network is a *feed-forward network.* In a feed-forward network, the neurons are grouped into layers, as shown in Figure 2. Each neuron feeds its output forward to every neuron in the following layer. There is no feedback from a later layer to an earlier one and no connections within a layer, e.g. there are no loops. The elements of the *input pattern* to be analyzed are presented to a *sensor layer*, which has one neuron for every component of the input. The sensor layer performs no processing; it merely distributes its input to the next layer. After the sensor layer comes one or more *hidden layers*; the number of neurons in these layers is arbitrary. The last layer is the *output layer*; the outputs from these neurons form the elements of the *output pattern.* Hence, the number of neurons in the output layer must match the desired length of the output pattern.

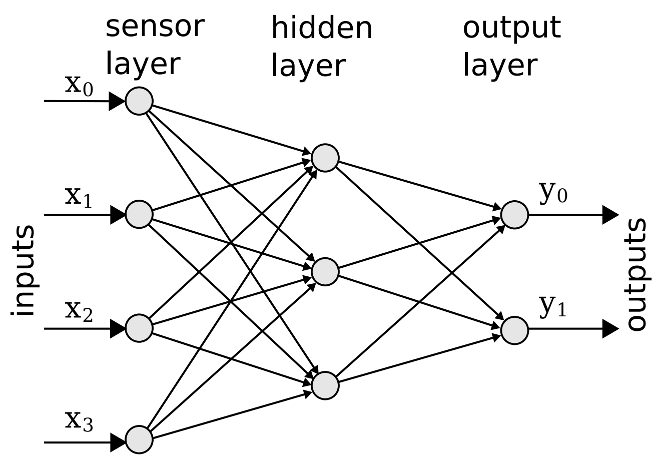

**Figure 2:** A simple neural network.

## Training the network

The *error* of a neural network is a function of the difference between the output pattern and the *target pattern* (desired output). The network can be trained by adjusting the network weights with the goal of reducing the error. Back-propagation is one technique for choosing the new weights. [2] This is a *supervised learning* process: the network is presented with both the input pattern as well as the target pattern. The error from the output layer is propagated backward through the hidden layers in order to determine each layer's contribution to the error, a process is illustrated in Figure 3. The weights in each layer are then adjusted to reduce the error for that input pattern.

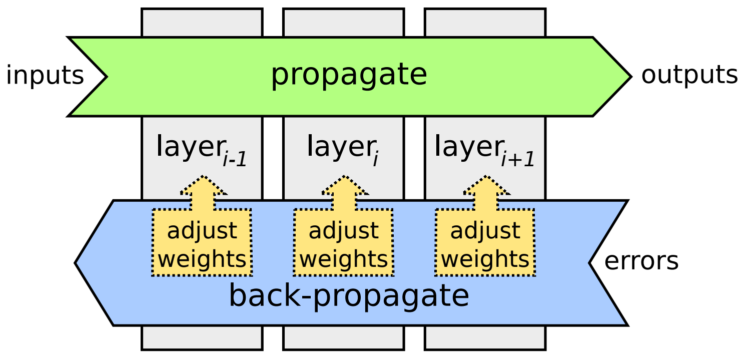

**Figure 3:** Back-propagation.

## Building a neural network

### Building a neuron

In this implementation, we use matrices to represent the weights for the neurons in each layer. The matrix calculations are performed using Alberto Ruiz's `hmatrix` [3, 4], a purely functional Haskell interface to basic matrix computations and other numerical algorithms in GSL [5], BLAS [6, 7] and LAPACK [8, 9]. With a matrix-based approach, there is no need for a structure to represent a single neuron. Instead, the implementation of the neuron is distributed among the following entities

- the inputs from the previous layer
- the output to the next layer
- a column in the weight matrix
- an activation function (in this implementation, the same function is used for all neurons in all layers except the sensor layer)

For the weight matrix, we use the `Matrix` type provided by `hmatrix`. The inputs, outputs and patterns are all column vectors. We use the `Matrix` type for these as well, but we introduce the type synonym `ColumnVector`. In Haskell, the `type` keyword defines an alternative name for an existing type; it does not define a new type. (A complete code listing, along with a sample character recognition application, is available online [10].)

```
type ColumnVector a = Matrix a
```

The activation function is the final element needed to represent the neuron. Here, we encounter one of the advantages of a functional approach. Like most most functional programming languages, Haskell supports **first-class functions**; a function can be used in the same way as any other type of value. It can be passed as an argument to another function, stored in a data structure, or returned as result of function evaluation. Hence, we don't need to do anything special to allow this neural network to use any activation function chosen by the user. The activation function can be supplied as an argument at the time the network is created.

It is convenient to create a structure to hold both the activation function and its first derivative. (The back-propagation algorithm requires that the activation function be differentiable, and we will need the derivative to apply the back-propagation method.) This helps to reduce the chance that the user will change the activation function and forget to change the derivative. We define this type using Haskell's record syntax, and include a string to describe the activation function being used.

```
data ActivationSpec = ActivationSpec
    {
      asF :: Double -> Double,
      asF' :: Double -> Double,
      desc :: String
    }
```

The first field, `asF`, is the activation function, which takes a `Double` (double precision, real floating-point value) as input and returns a `Double`. The second field, `asF'`, is the first derivative. It also takes a `Double` and returns a `Double`. The last field, `desc`, is a `String` value containing a description of the function.

Accessing the fields of a value of type `ActivationSpec` is straightforward. For example, if the name of the record is `s`, then its activation function is `asF s`, its first derivative is `asF' s`, and its description is `desc s`.

As an example of how to create a value of the type `ActivationSpec`, here is one for the identity function $f(x) = x$, whose first derivative is $f'(x) = 1$.

```
identityAS = ActivationSpec
    {
      asF = id,
      asF' = const 1,
      desc = "identity"
    }
```

The function `id` is Haskell's predefined identity function. The definition of `asF'` may seem puzzling. The first derivative of the identity function is 1, but we cannot simply write `asF' = 1`. Why not? Recall that the type signature of `asF'` is `Double -> Double`, so we need to assign an expression to it that takes a `Double` and returns a `Double`. However, `1` is just a single number. It could be of type `Double`, but not `Double -> Double`. To solve this issue, we make use of the predefined `const` function, which takes two parameters and returns the first, ignoring the second. By partially applying it (supplying `1` as the first parameter), we get a function that takes a single parameter and always returns the value `1`. So the expression `const 1` can satisfy the type signature `Double -> Double`.

The hyperbolic tangent is a commonly-used activation function; the appropriate `ActivationSpec` is defined below.

```
tanhAS :: ActivationSpec
tanhAS = ActivationSpec
    {
      asF = tanh,
      asF' = tanh',
      desc = "tanh"
    }

tanh' x = 1 - (tanh x)^2
```

At this point, we have taken advantage of Haskell's support for **first-class functions** to store functions in a record structure and to pass functions as parameters to another function (in this case, the ActivationSpec constructor).

## Building a neuron layer

To define a layer in the neural network, we use a record structure containing the weights and the activation specification. The weights are stored in an $n \times m$ matrix, where $n$ is the number of inputs and $m$ is the number of neurons. The number of outputs from the layer is equal to the number of neurons, $m$.

```
data Layer = Layer
    {
      lW :: Matrix Double,
      lAS :: ActivationSpec
    }
```

The weight matrix, lW, has type Matrix Double. This is a matrix whose element values are double-precision floats. This type and the associated operations are provided by the hmatrix package. The activation specification, lAS uses the type ActivationSpec, defined earlier. Again we use the support for first-class functions; to create a value of type Layer, we pass a record containing function values into another function, the Layer constructor.

## Assembling the network

The network consists of a list of layers and a parameter to control the rate at which the network learns new patterns.

```
data BackpropNet = BackpropNet
    {
      layers :: [Layer],
      learningRate :: Double
    }
```

The notation [Layer] indicates a list whose elements are of type Layer. Of course, the number of outputs from one layer must match the number of inputs to the next layer. We ensure this by requiring the user to call a special function (a “smart constructor”) to construct the network. First, we address the problem of how to verify that the dimensions of a consecutive pair of network layers is compatible. The following function will report an error if a mismatch is detected.

```
checkDimensions :: Matrix Double -> Matrix Double -> Matrix Double
checkDimensions w1 w2 =
  if rows w1 == cols w2
       then w2
       else error "Inconsistent dimensions in weight matrix"
```

Assuming that no errors are found, `checkDimensions` simply returns the second layer in a pair. The reason for returning the second layer will become clear when we see how `checkDimensions` is used.

The constructor function should invoke `checkDimensions` on each pair of layers. In an imperative language, a for loop would typically be used. In functional languages, a recursive function could be used to achieve the same effect. However, there is a more straightforward solution using an operation called a *scan*. There are several variations on this operation, and it can proceed either from left to right, or from right to left. We’ve chosen the predefined operation `scanl1`, read “scan-ell-one” (not “scan-eleven”).

```
scanl1 f [x1, x2, x3, ...] == [x1, f x1 x2, f (f x1 x2) x3, ...]
```

The l indicates that the scan starts from the left, and the 1 indicates that we want the variant that takes no starting value. Applying `scanl1 checkDimensions` to a list of weight matrices gives the following result (again assuming no errors are found).

```
scanl1 checkDimensions [w1, w2, w3, ...]
  == [w1, checkDimensions w1 w2,
          checkDimensions (checkDimensions w1 w2) w3, ...]
```

If no errors are found, then `checkDimensions` returns the second layer of each pair, so:

```
scanl1 checkDimensions [w1, w2, w3, ...]
  == [w1, checkDimensions w1 w2, checkDimensions w2 w3, ...]
  == [w1, w2, w3, ...]
```

Therefore, if the dimensions of the weight matrices are consistent, this operation simply returns the list of matrices, e.g. it is the identity function.

The next task is to create a layer for each weight matrix supplied by the user. The expression `map buildLayer checkedWeights` will return a new list, where each element is the result of applying the function `buildLayer` to the corresponding element in the list of weight matrices. The definition of `buildLayer` is simple, it merely invokes the constructor for the type `Layer`, defined earlier.

```
buildLayer w = Layer { lW=w, lAS=s }
```

Using the operations discussed above, we can now define the constructor function, `buildBackpropNet`.

```
buildBackpropNet ::
  Double -> [Matrix Double] ->  ActivationSpec -> BackpropNet
buildBackpropNet lr ws s = BackpropNet { layers=ls, learningRate=lr }
  where checkedWeights = scanl1 checkDimensions ws
        ls = map buildLayer checkedWeights
        buildLayer w = Layer { lW=w, lAS=s }
```

The primary advantage of using functions such as `map` and `scanl1` is not that they save a few lines of code over an equivalent *for loop*, but that these functions more clearly indicate the programmer's intent. For example, a quick glance at the word `map` tells the reader that the *same* operation will be performed on *every* element in the list, and that the result will be a *list* of values. It would be necessary to examine the equivalent for loop more closely to determine the same information.

## Running the Network

### A closer look at the network structure

The neural network consists of multiple layers of neurons, numbered from 0 to $L$, as illustrated in Figure 4. Each layer is fully connected to the next layer. Layer 0 is the sensor layer. (It performs no processing; each neuron receives one component of the input vector $\mathbf{x}$ and distributes it, unchanged, to the neurons in the next layer.) Layer $L$ is the output layer. The layers $l = 1..(L-1)$ are hidden layers. $z_{lk}$ is the output from neuron $l$ in layer $l$.

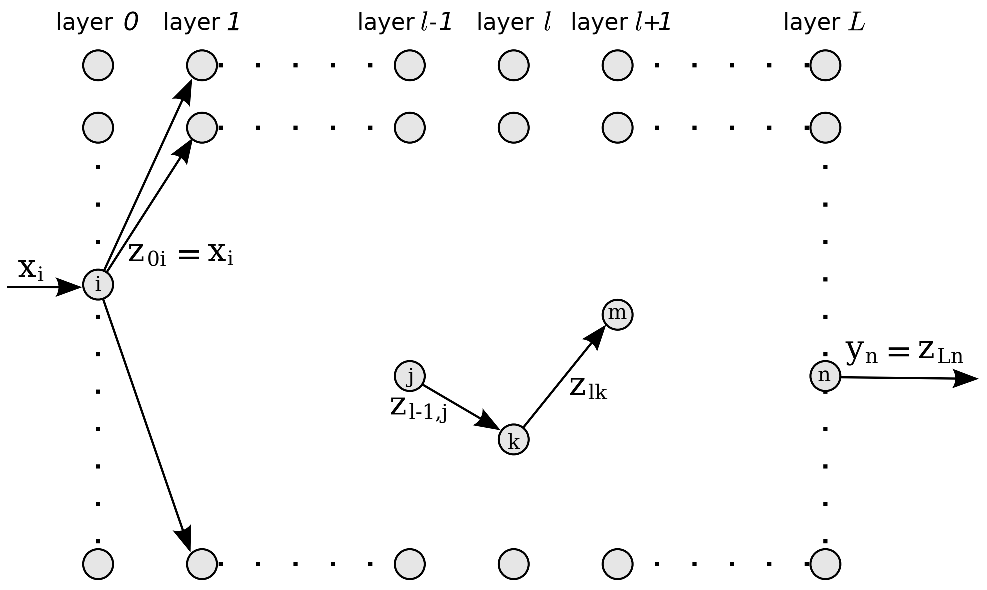

**Figure 4:** Propagation through the network.

We use the following notation:

- $x_i$ is the $i$th component of the input pattern;
- $z_{li}$ is the output of the $i$th neuron in layer $l$;
- $y_i$ is the $i$th component of the output pattern.

## Propagating through one layer

The activation function for neuron $k$ in layer $l$ is

$$a_{0k} = x_k$$

$$a_{lk} = \sum_{j=1}^{N_{l-1}} w_{lkj} z_{l-1,j} \qquad\qquad l > 0$$

where

- $N_{l-1}$ is the number of neurons in layer $l-1$.
- $w_{lkj}$ is the weight applied by the neuron $k$ in layer $l$ to the input received from neuron $j$ in layer $l-1$. (Recall that the sensor layer, layer 0, simply passes along its inputs without change.)

We can express the activation for layer $l$ using a matrix equation.

$$\mathbf{a_l} = \begin{cases} \mathbf{x} & l = 0 \\ \mathbf{W}_l\mathbf{x} & l > 0 \end{cases}$$

The output from the neuron is

$$z_{lk} = f(a_{lk})$$

where $f(a)$ is the activation function. For convenience, we define the function `mapMatrix` which applies a function to each element of a matrix (or column vector). This is analogous to Haskell's `map` function. (The definition of this function is in the appendix.) Then we can calculate the layer's output using the Haskell expression `mapMatrix f a`, where `f` is the activation function.

If we've only propagated the input through the network, all we need is the output from the final layer, $\mathbf{z}_L$. However, we will keep the intermediate calculations because they will be required during the back-propagation pass. We will keep all of the necessary information in the following record structure. Note that anything between the symbol `--` and the end of a line is a comment and is ignored by the compiler.

```
data PropagatedLayer
    = PropagatedLayer
        {
          -- The input to this layer
          pIn :: ColumnVector Double,
          -- The output from this layer
          pOut :: ColumnVector Double,
          -- The value of the first derivative of the activation function
          -- for this layer
          pF'a :: ColumnVector Double,
          -- The weights for this layer
          pW :: Matrix Double,
          -- The activation specification for this layer
          pAS :: ActivationSpec
        }
    | PropagatedSensorLayer
        {
          -- The output from this layer
          pOut :: ColumnVector Double
        }
```

This structure has two variants. For the sensor layer (`PropagatedSensorLayer`), the only information we need is the output, which is identical to the input. For all other layers (`PropagatedLayer`), we need the full set of values. Now we are ready to define a function to propagate through a single layer.

```
propagate :: PropagatedLayer -> Layer -> PropagatedLayer
propagate layerJ layerK = PropagatedLayer
        {
          pIn = x,
          pOut = y,
          pF'a = f'a,
          pW = w,
          pAS = lAS layerK
        }
  where x = pOut layerJ
        w = lW layerK
        a = w <> x
        f = asF ( lAS layerK )
        y = P.mapMatrix f a
        f' = asF' ( lAS layerK )
        f'a = P.mapMatrix f' a
```

The operator `<>` performs matrix multiplication; it is defined in the `hmatrix` package.

## Propagating through the network

To propagate weight adjustments through the entire network, we create a sensor layer to provide the inputs and use another *scan* operation, this time with `propagate`. The `scanl` function is similar to the `scanl1` function, except that it takes a starting value.

```
scanl f z [x1, x2, ...] == [z, f z x1, f (f z x1) x2), ...]
```

In this case, the starting value is the sensor layer.

```
propagateNet :: ColumnVector Double -> BackpropNet -> [PropagatedLayer]
propagateNet input net = tail calcs
  where calcs = scanl propagate layer0 (layers net)
        layer0 = PropagatedSensorLayer{ pOut=validatedInputs }
        validatedInputs = validateInput net input
```

The function `validateInput` verifies that the input vector has the correct length and that the elements are within the range [0,1]. Its definition is straightforward.

## Training the network

### The back-propagation algorithm

We use the matrix equations for basic back-propagation as formulated by Hristev [11, Chapter 2]. (We will not discuss the equations in detail, only summarize them and show one way to implement them in Haskell.) The back-propagation algorithm requires that we operate on each layer in turn (first forward, then backward) using the results of the operation on one layer as input to the operation on the next layer. The input vector $\mathbf{x}$ is propagated **forward** through the network, resulting in the output vector $\mathbf{z_L}$, which is then compared to the target vector $\mathbf{t}$ (the desired output). The resulting error, $\mathbf{z_L} - \mathbf{t}$ is then propagated **backward** to determine the corrections to the weight matrices:

$$W_{new} = W_{old} - \mu \nabla E \tag{1}$$

where $\mu$ is the learning rate, and $E$ is the error function. For $E$, we can use the sum-of-squares error function, defined below.

$$E(W) \equiv \frac{1}{2} \sum_{q=1}^{N_L} [z_{Lq}(x) - t_q(x)]^2$$

where $z_{Lq}$ is the output from neuron q in the output layer (layer L). The error gradient for the last layer is given by:

$$\nabla_{z_L} E = \mathbf{z}_L(x) - \mathbf{t} \tag{2}$$

The error gradient for a hidden layer can be calculated recursively according to the equations below. (See [11, Chapter 2] for the derivation.)

$$(\nabla E)_l = [\nabla_{z_l} E \odot f'(a_l)] \cdot \mathbf{z}_{l-1}^T \qquad \textit{for layers } l = \overline{1, L}$$

$$\nabla_{z_l} E = W_{l+1}^t \cdot [\nabla_{z_{l+1}} E \odot f'(a_{l+1})] \qquad \textit{calculated recursively from L-1 to 1} \tag{3}$$

The symbol $\odot$ is the *Hadamard*, or element-wise product.

### Back-propagating through a single layer

The result of back-propagation through a single layer is stored in the structure below. The expression $\nabla_{z_l} E$ is not easily represented in ASCII text, so the name "dazzle" is used in the code.

```
data BackpropagatedLayer = BackpropagatedLayer
    {
      -- Del-sub-z-sub-l of E
      bpDazzle :: ColumnVector Double,
      -- The error due to this layer
      bpErrGrad :: ColumnVector Double,
      -- The value of the first derivative of the activation
      --   function for this layer
      bpF'a :: ColumnVector Double,
      -- The input to this layer
      bpIn :: ColumnVector Double,
      -- The output from this layer
      bpOut :: ColumnVector Double,
      -- The weights for this layer
      bpW :: Matrix Double,
      -- The activation specification for this layer
      bpAS :: ActivationSpec
    }
```

The next step is to define the `backpropagate` function. For hidden layers, we use Equation (3), repeated below.

$$\nabla_{z_l} E = W_{l+1}^t \cdot [\nabla_{z_{l+1}} E \odot f'(a_{l+1})] \quad \textit{calculated recursively from L-1 to 1} \tag{3}$$

Since subscripts are not easily represented in ASCII text, we use `J` in variable names in place of $_l$, and `K` in place of $_{l+1}$. So `dazzleJ` is $\nabla_{z_l} E$, `wKT` is $W_{l+1}^t$, `dazzleJ` is $\nabla_{z_{l+1}} E$, and `f'aK` is $f'(a_{l+1})$. Thus, Equation (3) is coded as

```
dazzleJ = wKT <> (dazzleK * f'aK)
```

The operator `*` appears between two column vectors, `dazzleK` and `f'aK`, so it calculates the Hadamard (element-wise) product rather than a scalar product. The `backpropagate` function uses this expression, and also copies some fields from the original layer (prior to back-propagation).

```
backpropagate ::
  PropagatedLayer -> BackpropagatedLayer -> BackpropagatedLayer
backpropagate layerJ layerK = BackpropagatedLayer
    {
      bpDazzle = dazzleJ,
      bpErrGrad = errorGrad dazzleJ f'aJ (pIn layerJ),
      bpF'a = pF'a layerJ,
      bpIn = pIn layerJ,
      bpOut = pOut layerJ,
      bpW = pW layerJ,
      bpAS = pAS layerJ
    }
    where dazzleJ = wKT <> (dazzleK * f'aK)
          dazzleK = bpDazzle layerK
          wKT = trans ( bpW layerK )
          f'aK = bpF'a layerK
          f'aJ = pF'a layerJ

errorGrad :: ColumnVector Double -> ColumnVector Double ->
  ColumnVector Double -> Matrix Double
errorGrad dazzle f'a input = (dazzle * f'a) <> trans input
```

The function `trans`, used in the definition of `wKT`, calculates the transpose of a matrix. The final layer uses Equation (2), repeated below.

$$\nabla_{z_L} E = \mathbf{z}_L(x) - \mathbf{t} \tag{2}$$

In the function `backpropagateFinalLayer`, `dazzle` is $\nabla_{z_L} E$.

```
backpropagateFinalLayer ::
    PropagatedLayer -> ColumnVector Double -> BackpropagatedLayer
backpropagateFinalLayer l t = BackpropagatedLayer
    {
      bpDazzle = dazzle,
      bpErrGrad = errorGrad dazzle f'a (pIn l),
      bpF'a = pF'a l,
      bpIn = pIn l,
      bpOut = pOut l,
      bpW = pW l,
      bpAS = pAS l
    }
    where dazzle =  pOut l - t
          f'a = pF'a l
```

## Back-propagating through the network

We have already introduced the `scanl` function, which operates on an array from left to right. For the back-propagation pass, we will use `scanr`, which operates from right to left. Figure 5 illustrates how `scanl` and `scanr` will act on the neural network. The boxes labeled `pc` and `bpc` represent the result of each propagation operation and back-propagation operation, respectively. Viewed in this way, it is clear that `scanl` and `scanr` provide a layer of abstraction that is ideally suited to back-propagation.

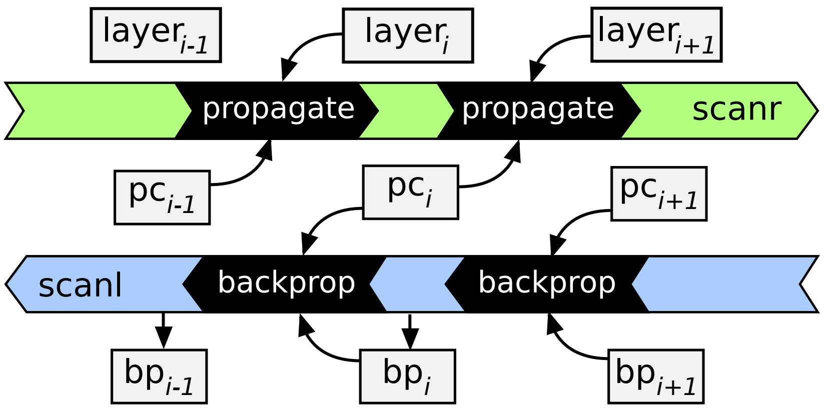

**Figure 5:** A schematic diagram of the implementation.

The definition of the `backpropagateNet` function is very similar to that of `propagateNet`.

```
backpropagateNet ::
  ColumnVector Double -> [PropagatedLayer] -> [BackpropagatedLayer]
backpropagateNet target layers = scanr backpropagate layerL hiddenLayers
  where hiddenLayers = init layers
        layerL = backpropagateFinalLayer (last layers) target
```

## Updating the weights

After the back-propagation calculations have been performed, the weights can be updated using Equation (1), which is repeated below.

$$W_{new} = W_{old} - \mu \nabla E \tag{1}$$

The code is shown below.

```
update :: Double -> BackpropagatedLayer -> Layer
update rate layer = Layer { lW = wNew, lAS = bpAS layer }
    where wOld = bpW layer
          delW = rate `scale` bpErrGrad layer
          wNew = wOld - delW
```

The parameter name `rate` is used for the learning rate $\mu$, and the local variable `rate` represents the second term in Equation (1). The operator `scale` performs element-wise multiplication of a matrix by a scalar.

## A functional approach to testing

In traditional unit testing, the code is written to test individual cases. For some applications, determining the desired result for each test case can be time-consuming, which limits the number of cases that will be tested.

Property-based testing tools such as QuickCheck [12] take a different approach. The tester defines properties that should hold for all cases, or, at least, for all cases satisfying certain criteria. In most cases, QuickCheck can automatically generate suitable pseudo-random test data and verify that the properties are satisfied, saving the tester's time.

QuickCheck can also be invaluable in isolating faults, and finding the simplest possible test case that fails. This is partially due to the way QuickCheck works: it begins with "simple" cases (for example, setting numeric values to zero or using zero-length strings and arrays), and progresses to more complex cases. When a fault is found, it is typically a minimal failing case. Another feature that helps to find a minimal failing case is "shrinking". When QuickCheck finds a fault, it simplifies (shrinks) the inputs (for example, setting numeric values to zero, or shortening strings and arrays) that lead to the failure, and repeats the test. The shrinking process is repeated until the test passes (or until no further shrinking is possible), and the simplest failing test is reported. If the default functions provided by QuickCheck for generating pseudo-random test data or for shrinking data are not suitable, the tester can write custom functions.

An in-depth look at QuickCheck is beyond the scope of this article. Instead, we will show one example to illustrate the value of property-based testing. What properties should a neural network satisfy, no matter what input data is provided? One property is that if the network is trained once with a given input pattern and target pattern and immediately run on the same input pattern, the error should be reduced. Put another way, training should reduce the error in the output layer, unless the error is negligible to begin with. Since the final layer has a different implementation than the hidden layers, we test it separately.

In order to test this property, we require an input vector, layer, and training vector, all with consistent dimensions. We tell QuickCheck how to generate suitable test data as follows:

```
-- A layer with suitable input and target vectors, suitable for testing.
data LayerTestData =
  LTD (ColumnVector Double) Layer (ColumnVector Double)
    deriving Show

-- Generate a layer with suitable input and target vectors, of the
-- specified "size", with arbitrary values.
sizedLayerTestData :: Int -> Gen LayerTestData
sizedLayerTestData n = do
    l <- sizedArbLayer n
    x <- sizedArbColumnVector (inputWidth l)
    t <- sizedArbColumnVector (outputWidth l)
    return (LTD x l t)

instance Arbitrary LayerTestData where
  arbitrary = sized sizedLayerTestData
```

The test for the hidden layer is shown below.

```
-- Training reduces error in the final (output) layer
prop_trainingReducesFinalLayerError :: LayerTestData -> Property
prop_trainingReducesFinalLayerError (LTD x l t) =
    -- (collect l) . -- uncomment to view test data
    (classifyRange "len x " n 0 25) .
    (classifyRange "len x " n 26 50) .
    (classifyRange "len x " n 51 75) .
    (classifyRange "len x " n 76 100) $
    errorAfter < errorBefore || errorAfter < 0.01
        where n = inputWidth l
              pl0 = PropagatedSensorLayer{ pOut=x }
              pl = propagate pl0 l
              bpl = backpropagateFinalLayer pl t
              errorBefore = P.magnitude (t - pOut pl)
              lNew = update 0.0000000001 bpl
                  -- make sure we don't overshoot the mark
              plNew = propagate pl0 lNew
              errorAfter =  P.magnitude (t - pOut plNew)
```

The `$` operator enhances readability of the code by allowing us to omit some parenthesis: `f . g . h $ x == (f . g . h) x`. This particular property only checks that training works for an output layer; our complete implementation tests other properties, including the effect of training on hidden layers. The `classifyRange` statements are useful when running the tests interactively; they display a brief report indicating the distribution of the test inputs. The function `trainingReducesFinalLayerError` specifies that a custom generator for pseudo-random test data, `arbLayerTestData`, is to be used. The generator `arbLayerTestData` ensures that the "simple" test cases that QuickCheck starts with consist of short patterns and a network with a small total number of neurons.

We can run the test in `GHCi`, an interactive Haskell REPL.

```
ghci> quickCheck prop_trainingReducesFinalLayerError
+++ OK, passed 100 tests:
62% len x 0..25
24% len x 26..50
12% len x 51..75
 2% len x 76..100
```

By default, QuickCheck runs 100 test cases. Of these, 62% of the patterns tested were of length 25 or less. We can request more test cases: the test of 10,000 cases below ran in 20 seconds on a 3.00GHz quad core processor running Linux. It would not have been practical to write unit tests for this many cases, so the benefit of property-based testing as a supplement to unit testing is clear.

```
ghci> quickCheckWith Args{replay=Nothing, maxSuccess=10000,
maxDiscard=100, maxSize=100} prop_trainingReducesFinalLayerError
+++ OK, passed 10000 tests:
58% len x 0..25
25% len x 26..50
12% len x 51..75
 3% len x 76..100
```

## Conclusions

We have seen that Haskell provides operations such as `map`, `scanl`, `scanr`, and their variants, that are particularly well-suited for implementing neural networks and back-propagation. These operations are not unique to Haskell; they are part of a category of functions commonly provided by functional programming languages to factor out common patterns of recursion and perform the types of operations that would typically be performed by loops in imperative languages. Other operations in this category include *folds*, which operate on lists of values using a combining function to produce a single value, and *unfolds*, which take a starting value and a generating function, and produce a list.

Functional programming has some clear advantages for implementing mathematical solutions. There is a straightforward relationship between the mathematical equations and the corresponding function definitions. Note that in the backpropagation example, we merely created data structures and wrote definitions for the values we needed. At no point did we provide instructions on how to sequence the operations. The final results were defined in terms of intermediate results, which were defined in terms of other intermediate results, eventually leading to definitions in terms of the inputs. The compiler is responsible for either finding an appropriate sequence in which to apply the definitions or reporting an error if the definitions are incomplete.

Property-based testing has obvious benefits. With minimal effort, we were able to test the application very thoroughly. But the greatest advantage of property-based testing may be its ability to isolate bugs and produce a minimal failing test case. It is much easier to investigate a problem when the matrices involved in calculations are small.

Functional programming requires a different mind-set than imperative programming. Textbooks on neural network programming usually provide derivations and definitions, but with the ultimate goal of providing an algorithm for each technique discussed. The functional programmer needs only the definitions, but it would be wise to read the algorithm carefully in case it contains additional information not mentioned earlier.

Functional programming may not be suited to everyone, or to every problem. However, some of the concepts we have demonstrated can be applied in imperative languages. Some imperative languages have borrowed features such as first-class functions, maps, scans and folds from functional languages. And some primarily functional languages, such as OCaml, provide mechanisms for doing object-oriented programming.

A complete code listing, along with a sample character recognition application, is available online [10].

## References

[1] Kevin Gurney. **An Introduction to Neural Networks**. CRC (1997).

[2] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. pages 318–362 (1986). `http://portal.acm.org/citation.cfm?id=104293`.

[3] Alberto Ruiz. hmatrix. http://code.haskell.org/hmatrix/. `http://code.haskell.org/hmatrix/`.

[4] Alberto Ruiz. A simple scientific library for haskell. `http://code.haskell.org/hmatrix/hmatrix.pdf`.

[5] Mark Galassi. **GNU Scientific Library : reference manual for GSL version 1.12**. Network Theory, Bristol, 3rd ed., for GSL version 1.12. edition (2009).

[6] National Science Foundation and Department of Energy. BLAS. http://www.netlib.org/blas/. `http://www.netlib.org/blas/`.

[7] J. Dongarra. Preface: Basic linear algebra subprograms technical (Blast) forum standard. **International Journal of High Performance Computing Applications**, 16(1):pages 1–1 (2002). `http://hpc.sagepub.com/cgi/doi/10.1177/10943420020160010101`.

[8] National Science Foundation and Department of Energy. LAPACK – linear algebra PACKage. http://www.netlib.org/lapack/. `http://www.netlib.org/lapack/`.

[9] E Anderson. **LAPACK users' guide**. Society for Industrial and Applied Mathematics, Philadelphia, 3rd ed. edition (1999).

[10] Amy de Buitléir. Github: backprop-example. https://github.com/mhwombat/backprop-example. `https://github.com/mhwombat/backprop-example`.

[11] R. M. Hristev. **The ANN Book**. 1 edition (1998). `ftp://ftp.informatik.uni-freiburg.de/papers/neuro/ANN.ps.gz`.

[12] Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of haskell programs. In **Proceedings of the fifth ACM SIGPLAN international conference on Functional programming - ICFP '00**, pages 268–279 (2000). `http://portal.acm.org/citation.cfm?doid=351240.351266`.

# Haskell ab initio: the Hartree-Fock Method in Haskell

by Felipe Zapata ⟨felipe.zapata@edu.uah.es⟩
and Angel J. Alvarez ⟨a.alvarez@uah.es⟩

*Scientific computing is a transversal subject where professionals of many fields join forces to answer questions about the behaviour of Nature using a variety of models. In this area, Fortran has been king for many years. It is now time to end Fortran's tyrannical reign! It is time to use a language which offers a high level of abstraction; a language which allows a straightforward translation of equations to code. It is time to use a language which has appropriate tools for parallelism and concurrency. Haskell is our language of choice: its levels of abstraction lead to a brief, elegant and efficient code. In this article, we will describe a minimal but complete Haskell implementation of the Hartree-Fock method, which is widely used in quantum chemistry and physics for recursively calculating the eigenvalues of the quantized levels of energy of a molecule and the eigenvectors of the wave function. Do not be afraid about the formidable name; we will skip most of the technical details and focus on the Haskell programming.*

## Joining two worlds

Haskell and its underlying theory have made us ask ourself some irresistible questions: have those equations written in the piece of paper the same mathematical meaning of those that we have implemented in Fortran? If programming is as much mathematical as it is artistic creation, then why are we still working with such twisted and ugly ideas? You ask the same questions to your workmates and professors, and after while working locked in your office, you will find out that an angry mob of Fortran programmers is waiting outside. After all, you dared to say that a pure and lazy functional language is the future of programming in science!

While waiting for the mob to get into our office, we will describe the Jacobi algorithm for calculating the eigenvalues and eigenvectors of a symmetric square matrix using the repa library. Then, equipped with this useful recursive function, we will see some basic details of the Hartree-Fock methodology and the self-consistent field (SCF) procedure for iteratively computing the eigenvalues and eigenvectors of a molecular system. In doing so, we will try to connect the simulation ideas with the powerful abstraction system of Haskell. We note that there is an excellent collection of modules written by Jan Skibinski for quantum mechanics and mathematics, but the approach used in those modules is different from ours [1].

## The Jacobi Algorithm

The Jacobi Algorithm is a recursive procedure for calculating all of the eigenvalues and eigenvectors of a symmetric matrix. The standard matrix eigenvalue problem seeks to find matrices $x$ and $\lambda$ such that:

$$\mathbf{Ax} = \lambda \mathbf{x}$$

(The $\lambda$ is a diagonal matrix of the eigenvalues; not a function abstraction!) The Jacobi algorithm is based on applying a transformation of the form

$$\mathbf{A}^{*}\mathbf{x}^{*} = \lambda \mathbf{x}^{*}$$

where

$$\mathbf{x}^{*} = \mathbf{Rx}$$

$$\mathbf{A}^{*} = \mathbf{R^{T}AR}$$

The transformation is applied to the original problem in such a way that the new expression obtained has the same eigenvalues and eigenvectors, but contains a matrix $\mathbf{A}^{*}$ which is diagonal. The matrix $\mathbf{R}$ is called the Jacobi rotation matrix, which is an orthogonal matrix ($\mathbf{R}^{-1} = \mathbf{R}^{T}$, i.e. the inverse is equal to the transpose) with all the entries of the matrix equal to zero except for the diagonal and two off-diagonal elements in the positions $kl$ and $lk$ of the matrix, as shown below.

$$\mathbf{R} = \begin{pmatrix} 1 & 0 & 0 & \dots & 0 & 0 \\ 0 & 1 & 0 & \dots & 0 & 0 \\ 0 & \dots & R_{k,k} & \dots & R_{k,l} & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & \dots & R_{l,k} & \dots & R_{l,l} & 0 \\ 0 & 0 & \dots & 0 & 0 & 1 \end{pmatrix}$$

When a similar transformation is applied over the matrix **A**, the off-diagonal elements of the new matrix $\mathbf{A}^*$ are equal to zero, meaning that $\mathrm{A^*}_{\mathrm{kl}} = \mathrm{A^*}_{\mathrm{lk}} = 0$.

The idea of the algorithm is to find the largest off-diagonal element of the matrix **A**, apply a rotation involving the row and column of the largest element and save the rotation matrix **R**. The rotations are applied until all the off-diagonal elements are lower than a delta. The application of the rotation matrix **R** over the matrix **A** produces the new matrix $\mathbf{A}^*$, whose elements are given by

$$A^*_{kk} = A_{kk} - tA_{kl} \tag{1}$$

$$A^*_{ll} = A_{ll} + tA_{kl} \tag{2}$$

$$A^*_{kl} = A^*_{lk} = 0 \tag{3}$$

$$A^*_{kj} = A^*_{jk} = A^*_{kj} - s(A_{lj} + \tau A_{kj}), j \neq k \wedge j \neq l \tag{4}$$

$$A^*_{lj} = A^*_{jl} = A^*_{lj} + s(A_{kj} - \tau A_{lj}), j \neq k \wedge j \neq l \tag{5}$$

where $s$, $t$ and $\tau$ are functions of $\mathrm{A}_{\mathrm{kl}}$.

Once all the rotations are applied, the eigenvalues are the diagonal elements of the final $\mathbf{A}^*$ and the eigenvectors **EV** are columns of the matrix product over all the Jacobi rotation matrices.

$$\mathbf{EV} = \prod_{i=1} \mathbf{R}_i$$

Because the rotation matrices are sparse, a partial product can be calculated in each rotation step through the following transformation,

$$R^*_{jk} = R_{ik} - s(R_{jl} + \tau R_{jk}) \tag{6}$$

$$R^*_{jl} = R_{il} + s(R_{jk} - \tau R_{jl}) \tag{7}$$

where $\mathbf{R}^*$ denotes the partial product matrix.

## Haskell Implementation

The repa library [2] offers efficient operations over arrays; the data structures and the functions of this library will be the basis for our implementation.

Since the matrix is symmetric, we can work with either the upper or lower triangular matrix. Then both repa unidimensional unboxed arrays and bidimensional

```
import Data.Array.Repa  as R

type EigenValues = VU.Vector Double
type EigenVectors = Array U DIM2 Double

data EigenData = EigenData {
             eigenvals :: !EigenValues
           , eigenvec  :: !EigenVectors } deriving (Show)

jacobiP ::  (Monad m,VU.Unbox Double) =>
            Array U DIM2 Double  ->
           m LA.EigenData
jacobiP !arr = let (Z:. dim :. _dim) = extent arr
                   tolerance = 1.0e-9
               in  jacobi arr (LA.identity dim) 0 tolerance

jacobi :: (Monad m, VU.Unbox Double)
          => Array U DIM2 Double
          -> Array U DIM2 Double
          -> Step
          -> Tolerance
          -> m EigenData
jacobi !arrA !arrP step tol

  | step > 5*dim*dim = error "Jacobi method did not converge "

  | otherwise = case abs maxElem > tol of
       True -> do
               arr1 <- rotateA arrA (matrixA arrA args)
               arr2 <- rotateR arrP (matrixR arrP args)
               jacobi arr1 arr2 (step+1) tol

       False -> return $
                EigenData (diagonalElems arrA) arrP

  where (Z:. dim :. _dim) = extent arrA
        sh@(Z:. k :. l) = maxElemIndex arrA
        maxElem = arrA ! sh
        args = parameters maxElem aDiff k l
        aDiff = toval (l,l) - toval (k,k)
        toval (i,j) = arrA ! (Z:. i :. j)
```

Listing 3.1: Jacobi Method

arrays duplicating the data are suitable choices to represent our matrix. We have chosen the bidimensional representation.

The main function has the signature depicted in Listing 1, where the Jacobi function takes as input a bidimensional array representing the symmetric matrix **A**, a bidimensional array for the rotational matrix **R**, the current iteration (an integer) and the numerical tolerance (which is just a synonym for a double). The function returns an algebraic data type containing the eigenvalues and eigenvectors, represented as a unboxed vector and a repa bidimensional matrix, respectively. The *jacobiP* function is the driver to initialize the rotation procedure, using the identity matrix as the initial value of the matrix **R**.

The first guard in the Jacobi function takes care of the maximum number of rotations allowed, where *dim* is the number of rows (or columns) of the symmetric matrix. The second guard checks that the greatest off-diagonal element of the symmetric matrix is larger than the tolerance. If it is not, then the matrix is considered diagonalized and we return an *EigenData* value containing the eigenvalues in the diagonal of the symmetric matrix *arrA* and the final rotation matrix contained in *arrP*.

Parallel computation on arrays in repa is abstracted using a generic monad $m$, as stated in the signature of the Jacobi function; therefore, *rotateA* and *rotateR* are monadic functions. Taking advantage of syntactic sugar, we extract the two new rotated matrices *arr1* and *arr2* and bind them to a new call of the Jacobi function. For calculating the $k$ and $l$ indexes, the *maxElemIndex* function finds the largest index of the bidimensional array. Finally, the *parameters* functions compute an algebraic data type containing the numerical parameters required for the rotation functions.

Listing 2 contains the implementation of *rotateA*. The key piece of the rotation implementation is the *fromFunction* function, which is included in the repa library and has the following signature *fromFunction :: sh -> (sh -> a) -> Array D sh a*. This function creates an array of a given shape from a function that takes as an argument an index of an entry in the new array, and calculates the numerical value for that entry. The result is a "delayed" array which can be evaluated in parallel using the *computeUnboxedP* function. Taking advantage of the symmetric properties of the matrix, we can rotate only the upper triangular matrix and leave the rest of the elements untouched. Therefore, we pass to *rotateA* a partially applied *matrixA*, which takes the indexes $m$ and $n$ for an upper triangular matrix and generates the numerical values using equations (1) to (5), leaving the values below the diagonal untouched.

The implementation of *rotateR* only differs from the previous one, in that equations (6) and (7) are used to calculate the numerical values and that the whole matrix is rotated not only the triangular part, as depicted in Listing 3.

```haskell
rotateA :: (Monad m ,VU.Unbox Double) =>
           Array U DIM2 Double ->
           (Int -> Int -> Double) ->
          m(Array U DIM2 Double)
rotateA !arr !fun =
  computeUnboxedP $ fromFunction (extent arr)
                  $ ( \sh@(Z:. n:. m) ->
                    case n <= m of
                         True -> fun n m
                         False -> arr ! sh)

matrixA :: VU.Unbox Double =>
           Array U DIM2 Double ->
           Parameters ->
           Int -> Int -> Double

matrixA !arr (Parameters !maxElem !t !s !tau !k !l) n m
  | (n,m) == (k,l) = 0.0
  | (n,m) == (k,k) = val - t*maxElem
  | (n,m) == (l,l) = val + t*maxElem
  | n < k && m == k = val - s*(toval (n,l) + tau*val)
  | n < k && m == l = val + s*(toval (n,k) - tau*val)
  | k < m && m < l && n == k = val - s*(toval (m,l) + tau*val)
  | k < n && n < l && m == l = val + s*(toval (k,n) - tau*val)
  | m > l && n == k = val - s*(toval (l,m) + tau*val)
  | m > l && n == l = val + s*(toval (k,m) - tau*val)
  | otherwise = val

  where val = toval (n,m)
        toval (i,j) = arr ! (Z :. i:. j)
```

Listing 3.2: rotateA function

```
rotateR :: (Monad m ,VU.Unbox Double) =>
           Array U DIM2 Double ->
           (Int -> Int -> Double) ->
           m(Array U DIM2 Double)
rotateR !arr !fun =
  computeUnboxedP $ fromFunction (extent arr)
                  $ ( \sh@(Z:. n:. m) -> fun n m)

matrixR :: VU.Unbox Double =>
           Array U DIM2 Double ->
           Parameters ->
           Int -> Int -> Double
matrixR !arr (Parameters !maxElem !t !s !tau !k !l) n m
  | m == k = val - s*((toval (n,l)) + tau*val)
  | m == l = val + s*((toval (n,k)) - tau*val)
  | otherwise = val

  where val = toval (n,m)
        toval (x,y) = arr ! (Z:. x :. y)
```

Listing 3.3: rotateR function

## Performance: When to be lazy

As we already know, Haskell is a non-strict language, where major implementations (for example, GHC) use a strategy called call-by-need or laziness to evaluate the code.

There is a slight difference between laziness and non-strictness. Non-strict semantics refers to a given property of Haskell programs that you can rely on: nothing will be evaluated until it is needed. The way we apply this strategy to our code is by using a mechanism called lazy evaluation. Lazy evaluation is the mechanism used by Haskell to implement non-strictness, using a device called the thunk.

Laziness can be a useful tool for improving performance on large arrays as one would deploy schemes that do not need to evaluate all array members to compute certain matrix operations. However, in the case where most matrix values will eventually be evaluated, it will reduce performance by adding a constant overhead to everything that needs to be evaluated.

Furthermore, due to laziness, function arguments will not always be evaluated, so they are instead recorded on the heap as a thunk in case they are evaluated later by the function.

Storing and then evaluating most thunks is costly, and unnecessary in this case, when we know most of the time the complete array of values needs to be fully evaluated. So, instead, it is necessary to enforce strictness when we know it is better. Optimising compilers like GHC yet try to reduce the cost of laziness

```
type EigenValues = VU.Vector Double
type EigenVectors = Array U DIM2 Double
data EigenData = EigenData {
              eigenvals :: !EigenValues
            , eigenvec :: !EigenVectors } deriving (Show)
```

Listing 3.4: Strict data types for eigenvalue operations

using strictness analysis [3], which attempts to determine if a function is strict in one or more of its arguments, (which function arguments are always needed to be evaluated before entering the function). Sometimes this leads to better performance, but sometimes the programmer has better knowledge about what is worth evaluating beforehand.

With bang patterns, we can hint the compiler about strictness on any binding form, making the function strict in that variable. In the same way that explicit type annotations can guide type inference, bang patterns can help guide strictness inference. Bang patterns are a language extension, and are enabled with the *BangPatterns* language pragma.

Data constructors can be made strict, thus making your values strict (weak head normal form) whenever you use them. You can see that we also used unboxed types of the vector library, as those ones are carefully coded to guarantee fast vector operations. You can see some examples of our data types in Listing 4, following the suggestion given by the repa authors [4].

As we have seen before, Jacobi's method its a recursive algorithm that attempts to converge values below a certain threshold in order to compute the desired $\mathbf{A}^*$ matrix. As we are using recursion we keep passing arguments every iteration and we need to ensure those arguments will be evaluated just before we pass them, avoiding to carry thunks along the way. These arguments must be forced, as shown in Listing 5. The hope is that the conjunction of strict values and tight loops will guide the compiler on the way of generating unboxed values as much as it is desired.

## Benchmark

In order to establish a baseline for the performance of our code, we wrote a Python implementation, as it allows us to compare "near C speed code" (via Numpy's built-in array type) with our repa implementation. We developed a test framework for comparing test runs between Python and Haskell.

Every test run loads a $100 \times 100$ matrix to be diagonalized. Our test bed was an Intel Core i5 @2.5 GHz laptop with 8GB RAM installed with OpenSuSE 11.4

```
jacobi !arrA !arrP step tol
...
...
              arr1 <- rotateA arrA (matrixA arrA args)
              arr2 <- rotateR arrP (matrixR arrP args)
              jacobi arr1 arr2 (step+1) tol
...
...
```

Listing 3.5: Jacobi strict argument passing

| Prototype | Threads | Total Memory (MB) | Productivity (%) |
|---|---|---|---|
| Python | 1 | - | 99 |
| Haskell -N1 | 1 | 8 | 95.6 |
| Haskell -N2 | 2 | 11 | 95.9 |
| Haskell -N4 | 3 | 14 | 96.7 |

**Table 1:** Space Comparison

x64.

As we have little expertise with Haskell performance tuning, we did not initially try to outperform the Python code. Despite this, we got near Python times with very little effort. However, we were concerned about the garbage collector, as our code consistently spent several seconds on garbage collection.

Provided that Haskell delivers parallel garbage collection (from GHC 7.0 onwards) we tried to perform as much memory management in parallel as mutation activity, in order to free mutator threads (running repa threads mainly) from garbage related work.

From the GHC manual [5], we found some experimental switches to allow the RTS to perform such activities in parallel with user code and also the possibility of performing parallel garbage collection only on younger generations. We tried to see if this set-up would allow repa threads to run concurrently with garbage collection without disrupting each other.

As it is shown in Table 3, We record the estimated memory side in the Haskell prototypes just to see the effects of different switches in the RTS. While in the Python prototype, we did not measure any memory usage at all. Also, we tried to see the overall effect of increasing the available cores (mainly the effect in the garbage collector). As you can see in Table 3 the maximal performance is achieved with two cores, adding more core does not speed up the calculation at this step of development. Further test will be carry out in the future.

| Prototype | Mutator Time | Mutator (elapsed) | GC Time | GC (elapsed) |
|---|---|---|---|---|
| Python | 60.2s | - | - | - |
| Haskell -N1 | 47.0s | 46.8s | 2.2s | 2.2s |
| Haskell -N2 | 49.2s | 34.8s | 2.1s | 1.8s |
| Haskell -N4 | 63.8s | 35.0s | 2.2s | 1.9s |

**Table 2:** Time Comparison

Being almost newcomers on this arena, we still are not certain about what is going on, but in the end we manage to low the running times (mostly by lowering garbage collection times) but this is a matter we will work in the future (we are afraid of that). Therefore we will provide criterion based benchmarking facilities in our cabal package to allow readers to test and validate our measurements.

## The Hartree-Fock Method

We are now in a position to talk about Hartree-Fock. In the beginning of the previous century, it was discovered that the energy of physical systems like atoms and molecules is quantized, contradicting our intuition that it must be a continuous. The scientific community had no choice but to accept the mathematical beauty of quantum theory. With this theory, we can study any molecular system we like…so long as we can solve the Schrödinger equation! Thus began the race to develop approximate methods for solving the Schrödinger equation. The Hartree-Fock method was established as the basic methodology upon which more accurate methods were developed. These methods, which only used fundamental constants of the mathematics and quantum physics without introducing any parameters (apart from the mass, charge, etc...), are called “ab initio” calculations. These methods are referred to as “from the beginning” or “first principles” methods. By the middle of the previous century, the first programs were written to solve the iterative equations that are the core of the Hartree-Fock method. These programs have persisted until today; there is still an irrational and cruel practice in many universities of punishing Ph.D. students in physics and chemistry with the debugging of thousand of lines of code written in Fortran 77; code that is written poorly and documented even worse.

The idea of the Hartree-Fock method is to solve the time-independent Schrödinger equation that can be formulated as

$$\mathbf{H}\Psi = E\Psi$$

Where $\Psi$ is the famous wave function that represents the physical system and $\mathbf{H}$

is the Hamiltonian operator. This equation can be transformed to our old friend the eigenvalue problem and solved using the Jacobi Method.

In quantum mechanics, the wave function contains all of the information about a system that we may need, while the operators represent properties that we can measure (called observables). In particular, the operator extracts information from the wave function: in the case of the Schrödinger equation, the Hamiltonian operator extracts the energy from the wave function that describes the electrons and nuclei of the molecules.

The only problem with the Schrödinger equation is that we do not know how to solve it! (Actually, there are solutions but they are only for the most trivial cases). Some approximations must be introduced to bring the equation into a formulation that it is solvable, though the nature of such approximations is out of the scope of this article. Henceforth, we will only be interested in solving the part of the system involving electrons. Do not run away, we are almost ready to have fun.

Since we our only interested in the electrons, the Schrödinger equation could be rewritten as

$$\mathbf{H_{elec}}\,\Phi_{elec} = E_{elec}\,\Phi_{elec}$$

where the subindex *elec* refers to the "electronic" part of the system.

In other words, we are trying to build an equivalent system which only describes the electrons. To approximate the electronic wave function indicated by $\Phi_{\mathrm{elec}}$, we will use a product of monoelectronic functions. A monoelectronic function is just an abstraction of how electrons behave around a nuclei. Each monoelectronic function (actually, the square of it) gives us the probability of finding an electron at some position around the nucleus. Each of these functions depends on the coordinates of the electron as well as the coordinates of the particular nucleus around which it is most probable to find the electron. Electrons "live", in some way, around the atomic nuclei.

In this manner, the electronic wave function is expanded as follows,

$$\Phi_{elec}(\mathbf{r_1}, \mathbf{r_2}, ..., \mathbf{r_n}) = \chi_1(\mathbf{r_1})\chi_2(\mathbf{r_2})...\chi_n(\mathbf{r_n}) \tag{8}$$

where $\mathrm{r_i}$ is the coordinate of the $n$th electron. Note that the coordinates of the nuclei do not appear in these equation, because we have assumed that the nuclei are fixed: this is the Born-Oppenheimer approximation.

Now, we can redefine the electronic Schrödinger equation as a set of $n$-coupled equations of the form

$$f_i\chi_i(\mathbf{r_i}) = \epsilon_i\chi_i(\mathbf{r_i}) \tag{9}$$

where $f_i$ is the Fock operator which is made up of three operators,

$$\hat{f}_i = \hat{T}_i + \hat{V}_i + \hat{V}_i^{HF} \tag{10}$$

The first term in the Fock operator represents the kinetic energy, the second term represents the electronic interactions between nuclei and the *ith* electron, and the last term represents the interaction between the $i$th electron and all of the other electrons.

## The Basis Set

How do we represent the monoelectronic functions of equation (8)? For reasons that will become clear later, a set of Gaussian functions is usually used; the list of Gaussian functions which represents the monoelectronic function is known as the basis set. Gaussian functions have the form,

$$\phi(\mathbf{R}, \alpha, l, m, n) = x^l y^m z^n e^{-\alpha \mathbf{R^2}} \tag{11}$$

Every basis set depends on the nuclear coordinates around which the expansion is made, denoted by $\mathbf{R}$. Each monoelectronic function is expressed as linear combination of $m$ Gaussian functions, each of which is multiplied by a coefficient,

$$\chi_i = \sum_{\mu=1}^{M} C_{\mu i} \phi_\mu \tag{12}$$

This expansion should contain infinite terms, in order to fully describe the original function. But if we want to compute something at all, we should choose a finite basis.

## The Roothaan-Hall Equations

The basis set is useful because we do not know the analytical form of the monoelectronic functions. The goal of the Gaussian basis set is to transform equation (9), which we still do not know how to solve, into some easy equation on matrices. When we do so, we arrive to the following matrix equation:

$$\mathbf{FC} = \mathbf{SC}\epsilon \tag{13}$$

In this equation, the Fock $\mathbf{F}$ operator now has a matrix representation and is multiplied by the $\mathbf{C}$ matrix which contains the coefficients of (12). $\boldsymbol{\varepsilon}$ is a diagonal matrix containing the energies for every equation like (9) and the $\mathbf{S}$ matrix called the overlap matrix, whose meaning will be discussed later. Notice that (13) would be an eigenvalue problem if there was no $\mathbf{S}$ matrix.

Matrices representing operators are Hermitian matrices, which are the generalization of symmetric matrices to the complex numbers. We will not worry about this, however, as our representation contains only real entries and therefore our operators are symmetric matrices.

```haskell
type NucCoord = [Double]

data Operator =  T | V NucCoord
                 deriving Show

(<<|) :: (NucCoord,Basis) -> Operator ->  ((NucCoord,Basis),Operator)
b1 <<| op  = (b1,op)

(|>>) ::  ((NucCoord,Basis),Operator) -> (NucCoord,Basis) -> Double
(b1,op) |>> b2 = case op of
                 T -> tijTotal b1 b2
                 V rc -> vijTotal b1 rc b2

kinetic_12   = (r1,b1) <<| T |>> (r2,b2)
potential_12 = (r1,b1) <<| V r3 |>> (r2,b2)
```

Listing 3.6: Operators definition

Introducing a basis set implies that the Fock operator should be expressed in the basis introduced. The question is this: how do we express the operator in the Gaussian basis set? The answer is that every element of the Fock matrix is just some mathematical operation involving the Gaussian functions and Fock operator. The Fock matrix entries are given by the following set of integrals,

$$\mathbf{F}_{\alpha\beta} = \int \phi_\alpha(\mathbf{r_i})\, \hat{\mathbf{f}}_i\, \phi_\beta(\mathbf{r_i})\mathbf{dr_i}$$

In other words, the element $(\alpha\,\beta)$ in the Fock matrix representation $\mathbf{F}$ is the integral of the the $\alpha$Gaussian function multiplied by the Fock operator of (9) applied to the $\beta$Gaussian function.

Paul Dirac introduced a shorter and more elegant notation for these kinds of integrals. Using the Dirac notation, these integrals are rewritten as

$$\langle \phi_\alpha \mid \hat{\mathbf{F}} \mid \phi_\beta \rangle = \mathbf{F}_{\alpha\beta} \tag{14}$$

Since Haskell is a great language to build domain specific languages, we have seen a great opportunity to implement our own DSL, introducing the Dirac notation directly in the code. This notation will be introduced in the next section.

## The Fock Matrix and the core Hamiltonian

In Listing 6, we define the infix notation for Dirac notation: every monoelectronic function over which the operator is applied is represented by a tuple containing the basis in which the function is expanded and the nuclear coordinates. Then, an

```
hcore :: [NucCoord] -> [Basis] -> [ZNumber] -> Nelec -> Array U DIM1
    Double
hcore coords basis atomicZ nelec =
 LA.list2ArrDIM1 dim (cartProd `using` parList rdeepseq)

 where dim = (nelec^2 + nelec) `div`2
       list = zip coords basis
       cartProd = do
          (i,atomi) <- zip [1..] list
          (j,atomj) <- zip [1..] list
          guard (i<=j)
          let sumVij = foldl1' (+) . getZipList $
                       (\z rc -> ((-z) * atomi <<|Vij rc|>> atomj))
                         <$> ZipList atomicZ <*> ZipList coords
          return $ (atomi <<|Tij|>> atomj) + sumVij
```

Listing 3.7: Core Hamiltonian

algebraic data type is used for representing the operators that make up the Fock operator. Using the two infix operators of Listing 6, we can squeeze the operators of (10) into the middle of two monoelectronic functions, giving us a representation in Dirac notation, as exemplified by the kinetic and potential expressions in Listing 6. We use the Dirac notation as a synonym for other functions behind the scenes, helping with the readability of the code.

The integrals resulting from the kinetic and electron-nucleus operators applied on the Gaussian functions have an analytical solution, but for the interaction among the electrons we do not have an analytical solution for more than 3 electrons interacting among themselves; this is the many-body problem. To deal with this, we applied a very human principle: if you do not know how to solve some problem, ignore it! Hence, once we ignore interactions between electrons, we have our first representation of the Fock matrix. This matrix is called the core matrix Hamiltonian.

Before going into details about the core Hamiltonian, let's take a look at its form. Below is the equation describing the entries of the core Hamiltonian:

$$HCore_{ij} = \langle \chi_i \mid \hat{\mathbf{T}} \mid \chi_j \rangle + \sum_{k=1}^{N} \langle \chi_i \mid \frac{\mathbf{1}}{\mathbf{R_k}} \mid \chi_j \rangle \tag{15}$$

Each element of the core Hamiltonian matrix is the sum of integrals represented using the Dirac notation of (14). This equation tells us that each element is composed of the kinetic energy plus the summation of interactions between one electron and all the $n$ nuclei that made up the molecule.

In agreement with the Dirac notation of Listing 6, in our implementation we represent the monoelectronic function $\chi_i$ with a tuple ($\mathbf{r_i}$,basis), containing the nuclear coordinates and the basis for doing the expansion of (12).

In Listing 7, we show a Haskell implementation of our first representation of the core Hamiltonian. Since the matrix is symmetric, we have decided to implement it as a unidimensional array containing the upper triangular matrix. The function for calculating the matrix requires the nuclear coordinates of all atoms, the basis used for expanding the monoelectronic functions, the charge of each atom (the *Znumber*, necessary to calculate the attraction between the nuclei and electrons), and the number of electrons. First, we calculate the entries of the matrix as a parallel list with a parallel strategy (see more about strategies at [6]). In order to take maximal advantage of sparks, a right level of granularity must be chosen; each monoelectronic function should contain a minimal set (minimal number of Gaussian functions) in order to balance the workload of each processor. This is a good thing, because in real calculations we have very large basis sets.

After we have evaluated the list using the auxiliary functions *list2ArrDIM1* and the dimension of the array, the list is transformed into an unboxed unidimensional repa array. The function *cartProd* which builds the entries of the core Hamiltonian takes advantage of the list monad. We first form a list of tuples representing the monoelectronic functions by zipping all the coordinates with their respective basis. Then, we generate the indexes *i,j* and the associated monoelectronic functions for those indexes in the core Hamiltonian matrix. Using a guard, we ensure that only the indexes of upper triangular matrix are taken into account. Then, according to (17), we return the result of applying the kinetic operator to two monoelectronic functions plus a summation which use the applicative style and the alternative applicative functor instance of the list functor, the *ZipList* instance. There is a lambda function that accepts two parameters, the atomic number Z and the nuclear coordinates, and returns the desired interaction. We partially apply this function to every element of the *ZipList* which contains all the atomic numbers; then, we apply the functor *ZipList* of partially applied functions to the *ZipList* containing all the coordinates. Finally, we fold over the final list after extracting the result with *getZipList*.

## The overlap matrix and the Jacobi Method

The overlap matrix is a result of expanding the monoelectronic functions using a basis of functions which are not completely orthogonal. The nature of the overlap matrix can be visualized if you think about a 2-dimensional vector: you can write any real 2-dimensional vector using a linear combination of the two vectors (1,0) and (0,1); this is because the vectors are orthogonal to each other. But in the case of using a basis that is not orthogonal, non-linear terms will appear and it is not possible to represent the vector as a linear combination. However, if you manage to normalize the basis in some way, a linear expansion can be used with the new normalized basis. In the same fashion, if you make a linear expansion of a function in some basis, the functions of the basis must be orthogonal with each other. Each element of the overlap matrix has the form shown below. An orthogonalization procedure makes one the elements for which $i = j$ in (14), and the rest of elements become zero.

Now, we will put all the pieces together in the implementation.

$$S_{ij} = \int_{-\infty}^{+\infty} dz \int_{-\infty}^{+\infty} dy \int_{-\infty}^{+\infty} \phi_i^* \phi_j dx \tag{16}$$

In the previous section, we have learnt how to build an approximation of the Fock matrix, but for solving our target equation (13), we needed to get rid of the overlap matrix. A transformation for the overlap matrix is required in such a way that the overlap matrix is reduced to the identity matrix as follows,

$$\mathbf{X}^{\dagger}\mathbf{S}\mathbf{X} = \mathbf{I} \tag{17}$$

Where **I** is the identity matrix.

The famous physicist Per-Olov Löwdin proposed the following transformation, which is called symmetric orthogonalization:

$$\mathbf{X} = \mathbf{S}^{-\frac{1}{2}} \tag{18}$$

Because **S** is an Hermitian matrix, $\mathbf{S^{-1/2}}$ is Hermitian too.

$$\mathbf{S}^{-\frac{1}{2}^{\dagger}} = \mathbf{S}^{-\frac{1}{2}}$$

then

$$\mathbf{S}^{-\frac{1}{2}}\ \mathbf{S}\ \mathbf{S}^{-\frac{1}{2}} = \mathbf{S}^{-\frac{1}{2}}\mathbf{S}^{\frac{1}{2}} = \mathbf{S^0} = \mathbf{1}$$

When it is applied the transformation in (14), we get a new set of equations of the form

$$\mathbf{F}'\mathbf{C}' = \mathbf{C}'\epsilon \tag{19}$$

where

$$\mathbf{F}' = \mathbf{X}^{\dagger}\mathbf{F}\mathbf{X} \text{ and } \mathbf{C}' = \mathbf{X^{-1}}\mathbf{C} \tag{20}$$

Finally, we have arrived at a standard eigenvalue problem! However, we need to generate the symmetric orthogonalization of (17). The matrix $\mathbf{S^{-1/2}}$ can be visualized as the application of the square root over the matrix **S**. For calculating a function over a diagonal matrix, we simply apply the function over the diagonal elements. For non-diagonal matrices, they should be first diagonalized, and then

```haskell
import qualified LinearAlgebra as LA
import qualified Data.Vector.Unboxed as VU

symmOrtho :: (Monad m, VU.Unbox Double)
             => Array U DIM2 Double
             -> m (Array U DIM2 Double)
symmOrtho !arr = do
symmOrtho arr = do
  eigData <- jacobiP $ arr
  let eigVal = LA.eigenvals eigData
      eigVecs = LA.eigenvec  eigData
      invSqrt = VU.map (recip . sqrt) eigVal
      diag = LA.vec2Diagonal invSqrt
  eigVecTrans <- LA.transpose2P eigVecs
  mtx1 <- LA.mmultP eigVecs diag
  LA.mmultP mtx1 eigVecTrans
```

Listing 3.8: Symmetric Orthogonalization

the function applied over the diagonal elements. Therefore, the $\mathbf{S}^{\text{-}1/2}$ matrix can be computed as:

$$\mathbf{S}^{-\frac{1}{2}} = \mathbf{U}\mathbf{s}^{-\frac{1}{2}}\mathbf{U}^{\dagger} \tag{21}$$

where the lower case $\mathbf{s}^{\text{-}1/2}$ is a diagonal matrix.

The Jacobi algorithm can be used to diagonalizing a matrix **M**, where the eigenvalues calculated are the entries of the diagonal matrix and the eigenvectors make up the matrix that diagonalized **M**, which are denoted as **U** in (21).

In Listing 8, we have the symmetric orthogonalization procedure to calculate the $\mathbf{S}^{\text{-}1/2}$ matrix. The *LinearAlgebra* module contains some subroutines tailored for performing matrix algebra using repa. Some of these functions are taken from the repa examples [7], the rest are based on the repa library functions. The *symmOrtho* function only requires the overlap matrix, which is first diagonalized using the Jacobi algorithm, resulting in an algebraic data type containing the eigenvalues as an unboxed vector and the eigenvectors as a bidimensional matrix. The *eigenvals* and *eigenvec* are accessor functions for retrieving the eigenvalues and eigenvectors, respectively. Then, the inverse square root of the eigenvalues is taken and the resulting vector a new diagonal matrix is created using *vec2Diagonal*. Using the functions *transpose2P* and *mmultP*, which are the transpose and the matrix multiplication functions respectively, the diagonal function is multiplied by the matrix containing the eigenvalues and by its transpose, resulting in the desired **X** matrix of (18).

Using the symmetric orthogonalization procedure and the Jacobi method, equations (19) and (20) can be solved, giving us a first approximation of the energies of the system.

## The Variational Method

In the previous section, we derived a first approximation for the calculating the coefficients which defined the electronic wave function by ignoring the interactions between electrons. Unfortunately, we cannot ignore the interactions between electrons. An analytical formulation for the interaction of many electrons is not known; instead, we calculate only interactions between pairs of electrons, approximating the overall force acting on electron as the average of the interacting pairs. The average is built using the coefficient for expanding the monoelectronic functions of (12). The average force rises a fundamental question: how do we know that the chosen coefficients of (12) are the best ones for approximating the interactions among the electrons? The variational principle is the answer.

**Theorem 1** (Variational Principle). *Given a normalized function $\Phi$which vanishes at infinity, the expected value of the Hamiltonian is an upper bound to the exact energy, meaning that*

$$\langle \Phi \mid \mathbf{H} \mid \Phi \rangle \geqslant \epsilon$$

This theorem states that if we have a function for representing $\Phi_{\text{elec}}$, the resulting energy after applying the Hamiltonian operator over the function is always greater that the real energy. Because $\Phi_{\text{elec}}$ depends on the expansion coefficients of (12), if we vary those coefficients in a systematic way we can generate a better electronic wave function $\Phi_{\text{elec}}$ and a more accurate value for the energy.

## The Contraction: Squeezing Dimensions

The recursive procedure described previously required the inclusion of the operator for describing the pair interactions between electrons. Then, the Fock Matrix can be reformulated as,

$$\mathbf{F} = \mathbf{HCore} + \mathbf{G} \tag{22}$$

where the $\mathbf{G}$ term stands for the interactions between electrons. This term depends on the coefficients matrix in (13), and on two types of integrals associated with the interacting electrons ($J$ and $K$, called the Coulomb and interchange integrals). To give an analytical expression to the previous term, let us define a matrix that is function of the coefficients used for expanding the monoelectronic function, called the density matrix, whose elements are given by

$$P_{\alpha\beta} = 2 \sum_{i=1}^{n} C_{\alpha i} C_{\beta i} \tag{23}$$

```haskell
import Data.Array.Repa  as R

calcGmatrix !density !integrals =
  computeUnboxedP $ fromFunction (Z:. dim)
                  (\(Z :. i ) -> sumAllS $
                  fromFunction (Z :. nelec)
                    (\( Z:. l) ->
                    let vec1 = unsafeSlice density (getRow l)
                        vec2 = map2Array integrals sortKeys (i,l)
                               nelec
                    in sumAllS . R.zipWith (*) vec1 $ vec2 ))

  where getRow x = (Any :. (x :: Int) :. All)
        (Z:. nelec :. _) = extent density
        dim = (nelec^2 + nelec) `div` 2
```

Listing 3.9: Computation of the G matrix

where the summation is carried out over the number of electrons.

The elements of the **G** matrix are given by,

$$G_{\alpha\beta} = \sum_{k=1}^{n}\sum_{l=1}^{n} P_{lk} * (\langle \alpha\beta \mid kl \rangle - \frac{1}{2} \langle \alpha l \mid k\beta \rangle) \tag{24}$$

In an imperative language, the usual way of implementing the **G** matrix is to nest four loops, using a four dimensional array for saving the $J$ and $K$ integrals which depend on four indexes as shown in (23). In our prototype, we have chosen a Map for storing the numerical values of the integrals, since is very easy to work with in our implementation. (Unboxed arrays could be a better data structure to query the values of the integrals.)

Before we dive into this multidimensional sea, a rearrangement of (24) can help us bring this equation to more familiar lands,

$$G_{\alpha\beta} = \sum_{l=1} \begin{bmatrix} P_{l1}, & P_{l2}, & \dots & P_{ln} \end{bmatrix} \bullet \begin{bmatrix} \langle \alpha\beta \mid\mid 1l \rangle, & \langle \alpha\beta \mid\mid 2l \rangle, & \dots & \langle \alpha\beta \mid\mid nl \rangle \end{bmatrix} \tag{25}$$

where

$$\langle \alpha\beta \mid\mid kl \rangle = \langle \alpha\beta \mid kl \rangle - \frac{1}{2} \langle \alpha l \mid k\beta \rangle = J - K \tag{26}$$

Equations (25) and (26) tell us that an entry of the **G** matrix can be considered as a summation over an array of dot products between vectors.

In Listing 9, the implementation for calculating the **G** matrix is shown, which fortunately is a symmetric matrix too. We use the recommended strategy suggested

```
map2Array :: M.Map [Int] Double
             -> ([Int] -> [Int])
             -> (Int,Int)
             -> Nelec
             -> Array D DIM1 Double
map2Array mapIntegrals sortKeys (i,l) nelec =
  R.fromFunction (Z:.nelec)
     (\(Z:.indx) ->
        let coulomb  = LA.map2val mapIntegrals $ sortKeys [a,b,indx,l]
            exchange = LA.map2val mapIntegrals $ sortKeys [a,l,indx,b]
        in coulomb - 0.5* exchange)

  where ne = nelec-1
        pairs = [(x,y) | x <- [0..ne], y <- [0..ne], x<=y ]
        (a,b) = pairs !! i
```

Listing 3.10: The Map to Array Function

by the repa authors, evaluating in parallel the whole array, but using sequential evaluation for the inner loops. With the previous notes in mind, we begin our journey from the first *fromFunction* which is in charge of building the whole array: we pass to this function the dimension of the final array (which is an upper triangular matrix) and the function for building the elements. Notice that as the implementation is done using unidimensional arrays for representing triangular matrices, the first index $i$ encodes the $\alpha$ and $\beta$ indexes of (25), meaning that $i$ should be decoded as the index of a bidimensional array. According to equations (25) and (26), the first *sumAllS* function adds up all the dot products, the innermost *sumAllS* collects the elements of each dot product, while the repa *zipWith* function carries out the desire dot operation between the vectors. The first vector is simply a row of the density matrix; the second vector, however, deserves a detailed analysis.

The four indexes integrals have the following symmetry:

$$\begin{aligned} \langle \alpha\beta \mid kl \rangle &= \langle \beta\alpha \mid kl \rangle = \langle \beta\alpha \mid lk \rangle = \langle \alpha\beta \mid lk \rangle \\ &= \langle kl \mid \alpha\beta \rangle = \langle lk \mid \alpha\beta \rangle = \langle lk \mid \beta\alpha \rangle = \langle kl \mid \beta\alpha \rangle \end{aligned} \tag{27}$$

Therefore, we only need to calculate one of the eight integrals. Nevertheless, a systematic way should be selected for choosing the indexes of the integral to be evaluated. The increasing order is a good criteria; from the eight possible integrals, only the integral with the lowest indexes is calculated and stored in a map.

In Listing 10, there is an implementation of the *map2Array* function for calculating the vector of integrals used in the computation of the **G** matrix. The arguments of this functions are the map containing the integrals, a function for sorting the keys, two indexes provided for the *calcGmatrix* function and the total number of electrons. The two indexes are used for generating the key of the desired integral. The first of these indexes encodes the $\alpha$ and $\beta$ indexes of (24) and (25); to decode these indexes, a list of tuples representing the indexes of a bidimensional matrix is calculated; then, the $i$th index of the unidimensional array corresponds to the indexes ($\alpha$,$\beta$). The second index corresponds to the row of the density matrix according to (25). Finally, the *map2val* function, which is a lookup function with some error reporting properties, retrieves the required key for the map of integrals and builds the numerical values of the vector. You may have been wondering why we have use a list of tuples for decoding the indexes instead of using the functions *toIndex* and *fromIndex* provided by the class *shape* of repa. The problem is that we are working with a unidimensional representation of diagonal matrices and we cannot use this pair of functions. If you are unconvinced, try using the *fromIndex* function to flatten an array representing a diagonal matrix.

The *map2Array* function returns a delayed array for performance reasons: it is more efficient to carry the indices of the elements, perform some operations with them, and finally evaluate the whole array, rather than compute the array in each step [2].

## The Self Consistent Field Procedure

The variational method establishes a theoretical tool for computing the best wave function. Starting from a core Hamiltonian, we derived an initial guess for the wave function. But we needed to account for the fact that electrons interact among themselves; therefore, we added some contribution for the description of this behaviour the **G** matrix term in (22). We still do not know how close is this new guess to the real system; therefore, we apply an iterative method to improve the wave function.

The Hartree-Fock self consistent field method is an iterative procedure which makes use of the variational principle to systematically improve our first guess from the core Hamiltonian.

It is now time to assemble the machinery. The SCF procedure is as follows:

1. Declare the nuclear coordinates, the basis set and the nuclear charges of all atoms.
2. Calculate all the integrals.
3. Diagonalize the overlap matrix using equations (17) and (18).
4. Compute a first guess for the density matrix (using the core Hamiltonian).
5. Calculate the **G** matrix.
6. Form the Fock matrix adding the core Hamiltonian and the **G** matrix.
7. Compute the new Fock matrix **F'** using (20).
8. Diagonalize **F'** obtaining **C'** and $\varepsilon$'.

```haskell
data HFData = HFData {
           getFock      :: !(Array U DIM1 Double)
         , getCoeff     :: !LA.EigenVectors
         , getDensity   :: !(Array U DIM2 Double)
         , getOrbE      :: !LA.EigenValues
         , getEnergy    :: !Double} deriving (Show)

scfHF :: (Monad m, VU.Unbox Double)
        => [NucCoord]
        -> [Basis]
        -> [ZNumber]
        -> Nelec
        -> m (HFData)
scfHF coords basis zlist nelec= do
        let core = hcore coords basis zlist nelec
            density = LA.zero nelec
            integrals = calcIntegrals coords basis nelec
        xmatrix <- symmOrtho <=< LA.triang2DIM2 $ mtxOverlap coords
           basis nelec
        scf core density integrals xmatrix 0 500
```

Listing 3.11: The Interface function

9. Calculate the new matrix of coefficients **C** using **C** = **XC'**.
10. Compute a new density matrix using the above **C** matrix and (23).
11. Check if the new and old density matrix are the same within a tolerance, if not, return to item 5 and compute again the **G** matrix.
12. Return the energies along with the Fock and the density matrices.

Now, using the syntactic sugar of the monads, we can cook our Hartree-Fock cake. First, a function can be set for collecting all the required data before forming the **G** matrix. In Listing 11 is the implementation of the *scfHF* function acting as collector of the required data and as interface with client codes asking for Hartree-Fock calculations. The algebraic data type containing the results is also shown.

The strict algebraic data type *HFData* stores: the Fock matrix as a triangular matrix; the matrix of coefficients *EigenVectors*; the density matrix; the eigenvalues of the equation *EigenValues* (19), which are called the orbital energies; and the total energy, which is given by the following expression,

$$E = \frac{1}{2}\sum_i \sum_j P_{ji}(HCore_{ij} + F_{ij}) \tag{28}$$

where **P** is the density matrix.

The *scfHF* is in charge of building the core Hamiltonian and calculating the map containing the integrals for computing the **G** matrix (for the first guess of the

```
scf :: (Monad m, VU.Unbox Double)
     => Array U DIM1 Double
     -> Array U DIM2 Double
     -> M.Map [Int] Double
     -> Array U DIM2 Double
     -> Step
     -> Int
     -> m(HFData)
scf !core !oldDensity !integrals !xmatrix step maxStep

   | step < maxStep = do
       fockDIM1 <- fock core oldDensity integrals
       hfData <- diagonalHF fockDIM1 xmatrix
       etotal <- variationalE core fockDIM1 oldDensity
       let newHFData = hfData {getFock=fockDIM1, getEnergy=etotal}
           bool =  converge oldDensity . getDensity $ newHFData
       case bool of
            True -> return newHFData
            False -> scf core (getDensity newHFData) integrals
                         xmatrix (step+1) maxStep

   | otherwise =  error "SCF maxium steps exceeded"
```

Listing 3.12: Self Consistent Field Function

density matrix, the zero matrix is usually used). The evaluation of the integrals deserves its own discussion, but we are not going to enter in any detail about the calculation of those integral. This function calculates the **X** matrix using the overlap matrix according to equations (17) and (18), but to apply the symmetric orthogonalization the upper triangular matrix should be reshape to a bidimensional symmetric matrix using the monadic function called *triang2DIM2*. Finally, the function which carries out the recursive part of the SCF procedure is called.

The *SCF* function is depicted in Listing 12: the function takes as arguments the core Hamiltonian, the current density matrix, the **X** matrix, the integer label of the current step and the maximum number of allowed steps. In the case where we exceed the maximum number of steps, we want to finish immediately regardless of the error. If the maximum number of steps is not exceeded, the Fock matrix is calculated by adding the core Hamiltonian and the **G** matrix together. This last matrix is calculated using the old density and the map of integrals.

Now, according to the algorithm, we need to generate a new matrix **F'** using the **X** matrix and then resolve this to a standard eigenvalue problem obtaining the energies as eigenvalues and a new matrix of coefficients as eigenvectors. In order to do so, we have defined a *diagonalHF* function defined in Listing 13. The *newFock* term on this function simply chains together two monadic functions which first

```
diagonalHF :: (Monad m, VU.Unbox Double)
           => Array U DIM1 Double
           -> Array U DIM2 Double
           -> m(HFData)
diagonalHF fock1 xmatrix = do
   fDIM2 <-newFock
   f' <- LA.toTriang fDIM2
   eigData <- jacobiP fDIM2
   let (coeff,orbEs) = LA.eigenvec &&& LA.eigenvals $ eigData
   newCoeff <- LA.mmultP xmatrix coeff
   newDensity <- LA.calcDensity newCoeff
   return $ HFData f' newCoeff newDensity orbEs 0.0

  where newFock = (LA.unitaryTransf xmatrix) <=< LA.triang2DIM2 $
     fock1
```

Listing 3.13: The DiagonalHF Function

take the unidimensional Fock matrix, translates it to its bidimensional form, and then applies equation (20) to the Fock matrix. This generates a bidimensional Fock matrix called *fDIM2*, which is diagonalized using the Jacobi method. The new **F'** is reshaped to a unidimensional array to be stored in the record. For retrieving the eigenvalues and eigenvectors of the resulting algebraic data type *EigenData*, we can use the arrow operator (&&&) in conjunction with the two accessor functions. Finally, we obtain the new matrix of coefficients and the density. Because the total energy is not calculated in this point, a zero is added to the value constructor.

Once the record containing the Hartree-Fock data has been calculated and coming back to the *SCF* function, we are in position to calculate the total energy using (28) and its implementation called the *variationalE* function, shown in Listing 14.

Finally, using the record syntax, we introduce the total energy and the Fock matrix before the diagonalization procedure, because it is useful for further calculations. Finally, we check for the convergence criteria. Based on the boolean returned by the convergence function, we decide if more variations of the coefficients are necessary of if we are done.

## Final Remarks

We are by far not Haskell experts, only new kids in the school. Therefore, all you feedback is much appreciated; please let us know your opinion about this project, and we will try to answer your questions as best as we can.

The code began as a challenge and playground for developing a big project in Haskell. After some months and to our own astonishment, we found that apart

```
variationalE ::(Monad m, VU.Unbox Double) =>
                 Array U DIM1 Double ->
                 Array U DIM1 Double ->
                 Array U DIM2 Double  ->
                 m Double
variationalE core fockMtx oldDensity =
  (0.5*)`liftM` do
  sumHF <- (R.computeUnboxedP $
           R.zipWith (+) core fockMtx) >>= \arr ->
           LA.triang2DIM2 arr
  result <- LA.mmultP sumHF oldDensity
  LA.tr result
```

Listing 3.14: The DiagonalHF Function

from performance tuning, we could easily design fairly complex structures with little effort. Many lessons are still to be learnt, but Haskell's powerful type system and the community support with hundreds of libraries are, from our point of view, what will make scientific software written in Haskell outstanding.

The SCF procedure described in this article is not the most popular method in the quantum chemistry packages due to convergence problems; instead, a method called direct inversion in the iterative subspace (DIIS) is used; this method is based on the SCF described above, and we are working on its implementation.

The set of modules making up the Hartree-Fock method, which will become a package in a near future, are not true competition to the electronic structure packages found either in the market or in the academic community [8]; but as far as we know, it is one of the first ones implemented in a functional language. Unlike one of the most famous pieces of software in computational quantum chemistry, we will not ban you from using our code if you compare the performance or the results of our code with some other package. [9]

It only remains to thank you, dear Haskeller, for following us through these lands, full of opportunities for applying the high abstraction level of Haskell to the challenge of simulating the natural phenomena. And remember: *Just Fun ... or Nothing.*

**BEWARE FortranIANS!!!**

$$\hat{\mathbf{H}}\mathbf{askell}\ \boldsymbol{\Psi} >>= \backslash\mathbf{E} - > \boldsymbol{\Psi}\ \mathbf{E}$$

## Acknowledgement

We want to thank Marco Marazzi for his help in the redaction of the paper, and our advisor Professor Luis Manuel Frutos for his patient and thorough support; without him, we would have been lynched by now!

## References

[1] Jan Skibinski. Numeric Quest.
`http://www.haskell.org/haskellwiki/Numeric_Quest`.

[2] Ben Lippmeier, Manuel Chakravarty, Gabriele Keller, and Simon Peyton Jones. Guiding parallel array fusion with indexed types. In **Proceedings of the 2012 symposium on Haskell symposium**, pages 25–36. Haskell '12, ACM, New York, NY, USA (2012). `http://doi.acm.org/10.1145/2364506.2364511`.

[3] Alan Mycroft. The theory and practice of transforming call-by-need into call-by-value. In **Proceedings of the Fourth 'Colloque International sur la Programmation' on International Symposium on Programming**, pages 269–281. Springer-Verlag, London, UK, UK (1980).
`http://dl.acm.org/citation.cfm?id=647324.721526`.

[4] Ben Lippmeier. Data.Array.Repa Haddock documentation.
`http://hackage.haskell.org/packages/archive/repa/latest/doc/html/Data-Array-Repa.html`.

[5] The GHC Team. The Glorious Glasgow Haskell Compilation System user's guide.
`http://www.haskell.org/ghc/docs/latest/html/users_guide/`.

[6] Simon Marlow. Control.Parallel.Strategies Haddock documentation.
`http://hackage.haskell.org/packages/archive/parallel/latest/doc/html/Control-Parallel-Strategies.html`.

[7] Ben Lippmeier. The repa-examples package.
`http://hackage.haskell.org/package/repa-examples`.

[8] Wikipedia. List of quantum chemistry and solid-state physics software.
`http://en.wikipedia.org/wiki/List_of_quantum_chemistry_and_solid-state_physics_software`.

[9] Anonymous. Banned By Gaussian. `http://www.bannedbygaussian.org/`.