World Academy of Science, Engineering and Technology
International Journal of Computer and Systems Engineering
Vol:3, No:3, 2009

Pattern Recognition of Biological Signals

Paulo S. Caparelli, Eduardo Costa, Alexsandro S. Soares and Hipólito Barbosa

Abstract—This paper presents an evolutionary method for designing electronic circuits and numerical methods associated with monitoring systems. The instruments described here have been used in studies of weather and climate changes due to global warming, and also in medical patient supervision. Genetic Programming systems have been used both for designing circuits and sensors, and also for determining sensor parameters. The authors advance the thesis that the software side of such a system should be written in computer languages with a strong mathematical and logic background in order to prevent software obsolescence, and achieve program correctness.

Keywords—Pattern recognition, evolutionary computation, biological signal, functional programming.

Open Science Index, Computer and Systems Engineering Vol:3, No:3, 2009 publications.waset.org/15962.pdf

I. INTRODUCTION

The modern industrial society has been made possible by the ingenuity of Honoré Blanc. Before him, complex mechanisms like muskets were made one by one. Each part was manufactured, adjusted and filed to fit the object that the craftsman was assembling at that moment. If a musket needed repair, it was sent to a gunsmith who would take measurements, and manufacture a replacement for the defective component. In 1790, Honoré Blank called together a group of high ranking officials, in front of whom he assembled muskets from parts drawn from bins. Thomas Jefferson by now had visited Blanc's workshop and learned about his method.

Eighteen years after Blanc's demonstration, when Jefferson was president of the United States, military authorities realizing the need to defend their country began to rearm. Around 1798, Treasury Secretary Wolcott sent a certain Eli Whitney one of Honoré Blanc's reports. To promote the new idea, Whitney faked a duplication of Blanc's demonstration. He hand-crafted each part so that the components would fit together. The swindle worked fine, and the American government promised to buy from ten to fifteen thousand muskets. However, the parts weren't interchangeable at all, so he took eight years to deliver the guns. Nevertheless, Whitney's enthusiasm in promoting Blanc's ideas made interchangeable parts a DE FACTO manufacturing standard in modern industry.

With the advent of computers, industry needed two subsystems — hardware and software. In principle, software could be designed to fit any hardware. The problem was that technological limitations, like speed and memory, imposed serious restrictions on the computer industry. When limitations are lifted by advances in hardware design, programmers face a dilemma:

P.S. Caparelli and E. Costa are with Department of Electrical Engineering, Federal University of Uberlândia (UFU), Uberlândia, Minas Gerais, Brazil e-mail: pscaparelli@ufu.br, costa@ufu.br

A.S.Soares is with Department of Computer Science, Federal University of Goiás (UFG), Catalão, Goiás, Brazil e-mail: alex@catalao.ufg.br

H.Barbosa is with Federal Institute of Education, Science and Technology of Goiás (IFG), Goiânia, Goiás, Brazil e-mail: hipolito@cefet.br

- They can keep their languages, methods and libraries, so old software can run in the new hardware. However, it will lack the functionality that improved technology made available.
- They can write more powerful software with added features and safety tools. This is what is usually done.

The conclusion of the above discussion is that software becomes obsolete, and needs constant replacement. The trouble is that there isn't enough manpower to continually rewrite applications.

This paper is about software obsolescence and how to remedy it so that Honoré Blank's great contribution to industry can be applied to software development. Of course, no one can expect to give a short term solution to this problem, since any solution involves extirpation of deeply rooted habits and cherished methods. In fact, Honoré Blanc himself was not very successful in changing the manufacturing processes of his time. According to Lienhard[1], since Blanc used unskilled labor, he *had made factories independent of government control over crafts*. Therefore, the authorities in charge closed down Mr. Blanc's business.

Computer languages become obsolete because they suffer restriction imposed by the underline hardware. For instance, technology may impose memory limitations that prevent implementation of dynamic memory. Therefore, a computer language may need to reuse variables. This is the main reason for destructive updates being so widespread in mainstream computer languages. Speed or security reasons may prevent implementation of global or tail call optimization. In few words, state of the art technology makes language designers deviate from mathematical systems, and invent formal methods that are easier to implement and deploy. When technological advances render the modifications unnecessary, computer languages are scrapped or updated, and one cannot compile vintage software anymore.

II. FUNCTIONAL PROGRAMMING

Technologically conditioned languages (TCL) are those whose syntax and semantics are dictated by the stage of computer technology, or by the knowledge people have of computers. The memory model, the lack of garbage collection, the poor representation of abstract data types, the execution strategy and numerical orientation were forced on language designers by the limited hardware they had. If John Backus tried to endow FORTRAN (the first high level language) with better data processing tools, it would have been too slow for the intended applications, and would not have fitted in the computers manufactured in 1950. It is interesting to note that Backus[6] made it clear that he did not want FORTRAN to be what it was. His design was entirely dictated by his

hardware and his knowledge of language implementation. What he really wanted was a functional language based on Combinatory Logic, a branch of mathematics created by the Ukrainian Mathematician Schonfinkel[7]. Here is what Backus wrote about FORTRAN and similar languages in a speech delivered when he received the prestigious Turing Award:

> *Conventional programming languages are growing ever more enormous, but not stronger. Inherent defects at the most basic level cause them to be both fat and weak: their primitive word-at-a-time style of programming..., their close coupling of semantics to state transitions, their division of programming into a world of expressions and a world of statements, their inability to effectively use powerful combining forms for building new programs from existing ones, and their lack of useful mathematical properties for reasoning about programs.*

As one can see from Backus's speech, he did not like technologically conditioned languages, not even the one he invented.

There are computer languages whose features are dictated not by the technology itself, but by the way people perceive it. Most of these human oriented languages are derived from the λ-Calculus (see [8]), while others come from developments in Symbolic Logic. From now on, we will refer to this family of language as Logic Languages.

III. Haskell

In order to protect themselves against obsolescence and in order to use the rich inheritance of Mathematics and Logic, the authors of this paper decided to use a functional language based on λ-Calculus. There are two options: Clean and Haskell. Although Clean is safer and more elegant than Haskell, and more faithful to λ-calculus, this paper will present its results in Haskell, since it is more widely known than Clean. In fact, Haskell is used as a teaching tool in many important colleges and universities. However, the reader should consider switching to Clean if he has a stake in safety and performance.

Since Haskell is a branch of Mathematics, it can be easily learned by those who have a working knowledge of functional analysis, set theory, and algebra. In the rest of this section, the interested reader will find a short introduction to functional programming. The method of programming numerical algorithms used here are largely borrowed from Hughes[2].

Let us write a small program to calculate the prime factors of a natural number. As a computable function, `factors` is defined thus:

```
1  import System
2
3  -- Least divisor starting from k
4  ldf k n | rem n k==0 = k
5          | k^2 > n = n
6          | otherwise= ldf (k+1) n
7
8  factors 1 = []
9  factors n = p : factors (n ÷ p) where
10   p= ldf 2 n
11
12 main = do
13   args ∈ getArgs
14   case args of
15     [n] → print $ factors (read n)
16     otherwise → putStrLn "e.g. usus: factors 57"
```

Function `getArgs` returns the command line arguments as a list of **String**s. The first argument of this list is converted to a numerical value by the `read` function. Here is how to use the above program:

```
C:\fp>ghc -O2 fibo.hs --make
[1 of 1] Compiling Main
Linking fibo.exe ...

C:\fp>factors.exe 5754
[2,3,7,137]
```

In order to get a better understanding of how the program works, let us remove the factors function.

```
1  -- File: args.hs
2  import System(getArgs)
3
4  main = do
5     a ∈ getArgs
6     print a
```

Here is how to compile and execute `args.hs`:

```
C:\fp>ghc args.hs --make
[1 of 1] Compiling Main
Linking args.exe ...

C:\fp>args 4 5 6 7
["4","5","6","7"]
```

As can be seen in the above example, `getArgs` returns the command line arguments as a list of strings. In the main function, the variable `args` matches the list `[n]`, and assigns the string `"5754"` to `n`. Finally, `(read n)` converts the string into an integer number.

The definition of `ldf` (least divisor) is somewhat obvious. Vertical bars introduce conditions; for instance `ldf k n |rem n k==0 =k` means: *When the remainder* `rem n k` *equals 0,* `k` *is a divisor of* `n`. Since `k` starts from the smallest prime, and it is incremented at each interaction, it is the lowest divisor.

Function `factors n` is defined as a set of rewriting equations. Each equation has a left hand side, and a right hand side. If an expression matches the left hand side, it is rewritten as the right hand side. For instance, the expression `factors 1` matches the first equation, and is rewritten as `[ ]` (the empty list, since 1 does not have factors greater than 1). The expression `factors 8` matches the second equation; the right hand side builds a list whose first element is `p` (the least divisor of 8), and the tail contains the factors of `n` divided by 2; since `n` is 8, the tail contains the factors of 4.

Lists are ordered sequences of elements. In general, a list is represented by its elements between square brackets. In Haskell, the list constructor is a colon; e.g. the expression `(3:[2,4])` builds a list `[3,2,4]`, whose first element is `3`, and whose tail is `[2,4]`. The two components of a list (head and tail) are taken apart by pattern matches; for instance, if the

Open Science Index, Computer and Systems Engineering Vol:3, No:3, 2009 publications.waset.org/15962.pdf

pattern `(x:xs)` matches `[1,2,3]`, the list head is assigned to `x`(`x=1`), and the tail is assigned to `xs` (`xs=[2,3]`).

Mathematicians have a notation to represent sets that both Clean and Haskell have borrowed; it is called Zermelo Frankel notation after the two logicians who made important contributions to set theory.

Both in Mathematics, and in Haskell, `[(i, i^2) |i ∈[1..n]]` represents a list of pairs `(i, i^2)`, where $i \in [1..n]$, and `i^2` is the square of `i`. N. B. Most Haskell programmers prefer ← to ∈.

The reader will notice that Haskell uses a '`do`' keyword in order to introduce a sequence of commands. However, this is nothing else than an alternative to the Zermelo-Frankel notation.

Haskell has both an interpreter, useful for testing ideas, and a compiler, when speed and efficiency are at stake. If one looks at the following interpreter session, one will see that the `do`-notation is equivalent to the Zermelo-Frankel notation.

```
1  Prelude> [(x, x*x) | x ∈ [2..6]]
2    [(2,4),(3,9),(4,16),(5,25),(6,36)]
3  Prelude> do x ∈ [2..6]; return (x, x*x)
4    [(2,4),(3,9),(4,16),(5,25),(6,36)]
```

Around 1932, the American logician Alonzo Church[3] introduced the notions of computable functions and untyped λ-calculus into the world of Mathematics. Untyped λ-calculus is inconsistent, but Church published two other papers in 1936 (see [4] and [5]), where a modification of his original theory proved to be consistent. Around 1990, Mathematicians like Barendregt, Hughes, Plasmeijer and Peyton-Jones relied on λ-calculus in an effort to create a computer language that could solve two of the greatest problems of Computer Science: software obsolescence and referencial opacity.

Referencial opacity has to do with functions that have side effects. In a program written in C, for instance, it is not enough to look at the text of the program in order to understand what it does, or to find a bug. Since the computational process can change the values of variables, one has to trace the history of computation to localize the malfunction. On the other hand, Mathematicians don't need to examine the process used by Archimedes to find a proof of his theorems in order to perceive that they are correct. Mathematics is referentially clear, while C is referentially opaque.

Opaque languages hide bugs and make programs obscure, and this brings difficulties in proving implementation correctness. In fact, algorithms are specified in a mathematical language that is clear, but implemented in opaque languages that are obscure. Mathematical tools can be used to prove that the specification is correct, but it is almost impossible to use them to prove that the implementation is also correct.

Before passing to the use of genetic programming in trouble shooting, the main technical contribution of the present paper, let us examine a simple numerical problem in order to get a firmer hold on the functional programming concepts.

IV. Parallel programming

Since functional languages are based on Mathematics, they have the ability to *effectively use powerful combining forms for building new programs from existing ones* (Backus[6]). This ability allows them to keep pace with both hardware and software technology. For instance, modern machines are multiprocessing; people who program in traditional languages are rarely able to use this new technology in building applications that explore hardware parallelism. Below, we will show an example of how easily a Haskell program can be parallelized in a multi-core PC.

Cryptography is one of the most important technologies of our computer based society. It is used not only to protect privacy and industrial or military secrets, but also assets, since commerce and financial transactions are increasingly dependent on the Internet.

One of the most useful methods of cryptography is public key ciphers; this method is based on large prime numbers. A prime number is divisible only by itself. Checking whether a number is prime is a difficult problem. Therefore, one often approaches it using parallel computation. A simple (and inefficient) test for primality consists in verifying whether the number equals to its least divisor. The program below applies this method in parallel to find a list of prime numbers.

```
1  import Data.Maybe
2  import Control.Parallel.Strategies
3  import Control.Parallel
4  import System
5
6  ld n= ldf 2 where
7    ldf k |  rem n k == 0 = k
8          | k^2 > n = n
9          | otherwise= ldf (k+1)
10
11 prime n | n==1 = False
12         | otherwise= ld n==n
13
14 prim n= if (prime n) then  n else 0
15
16 bigNums = [25431045773439..]
17 primlist n =  (parMap rwhnf) prim (take n bigNums)
18
19 main= do
20    args ∈ getArgs
21    case args of
22      [n] → prt (primlist (read n))
23      otherwise → putStrLn "e.g. usus: pprims 5"
24
25 prt [] = return ()
26 prt (x:xs) | x==0 = prt xs
27 prt (x:xs)= do print x
28                prt xs
```

Here is the command line used to compile and run the above program:

```
D:\par>ghc -O2 -threaded pprims.hs --make
[1 of 1] Compiling Main
Linking pprims.exe ...

D:\par>pprims.exe 400 +RTS -N2
25431045773453
25431045773461
25431045773477
25431045773503
25431045773813
```

Open Science Index, Computer and Systems Engineering Vol:3, No:3, 2009 publications.waset.org/15962.pdf

In line 16, the program assigns an infinite list of integer numbers to `bigNums`. The list starts with `25431045773439`. In line 17, `(take n bigNums)` takes `n` numbers from `bigNums`; this is always possible because `bigNums` has an infinite number of elements; as one can see, Haskell allows the use of mathematical concepts like infinite sequences to reason about programs. As for the form `(parMap rwhnf)`, it applies `prim` to each element of `(take n bigNums)`, producing a list whose elements are either a prime number or 0.

Function `parMap` is an example of what Backus calls powerful combining forms. It applies a function to every element of a list. The argument `rwhnf` is a parallel execution strategy; the interested reader should consult Peyton-Jones[9]. Figure 1 shows how `parMap` distributes processing between the two cores of a desktop computer. The machine used in the experiment is an old model, in order to show that one can explore parallelism even in dated hardware.

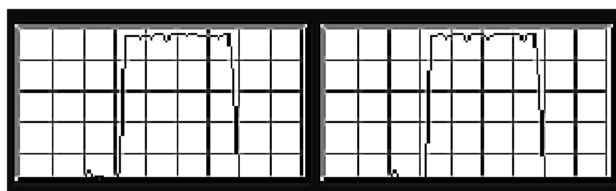

Fig. 1. CPU use in parallel processing

For the sake of completeness, one can find a listing of public cryptography below. One interesting feature of Haskell and Clean is the possibility of automatically deriving methods for many classes of programs. For instance, public cryptography has two keys, one for encoding, and another for decoding. The encoding key is made public, and anyone who wants to send coded information to the receiver can use it. The receiver is the only entity that has the decoding private key. Therefore, only the receiver can access the message. Line 2 allows Haskell to read and show both the public and private keys that are represented by a data structure. A similar job in conventional language requires compiler writing technology and complex parsers. Here is an interpreted session of the cryptography program:

```
Prelude> :l "crypto.hs"
*Main> let (pr,pub)= genRSAKey prim1 prim2
*Main> pr
PRIV 646738089133343592386779931
248745418897420280882781905
*Main> pub
PUB 646738089133343592386779931
129347617826658546059046593
*Main> let code= ersa pub 12345678987654
*Main> code
957410300300969344635 20585
*Main> drsa pr code
12345678987654
```

```
1  data RSAKey = PUB Integer Integer |
   PRIV Integer Integer
2                  deriving (Show, Read)
3
4  mInverse d f= loop (1, 0, f) (0, 1, d) where
5      loop (x1, x2, x3) (y1, y2, 0)= (0, y1)
6      loop (x1, x2, x3) (y1, y2, 1) |
   y2<0= (1, f+y2)
7      loop (x1, x2, x3) (y1, y2, 1)= (1, y2)
8      loop (x1, x2, x3) (y1, y2, y3)=
9        loop (y1, y2, y3) (x1-q*y1, x2-q*y2, x3-q*y3)
10         where q= x3 ÷ y3
11
12 expm m b k = ex b k 1 where
13   ex a k s
14     | k == 0 = s
15     | mod k  2 == 0 = ((ex (mod (a*a)  m)) (k ÷
   2)) s
16     | otherwise = ((ex (mod (a*a)  m)) (k ÷
   2)) ((s*a) `mod` m)
17
18 invm :: Integer → Integer → Integer
19 invm m a
20    | g /= 1 = error "No inverse exists"
21    | otherwise = x `mod` m
22  where (g,x) = mInverse a m
23
24 genRSAKey p q = (PRIV n d,PUB n e) where
25    phi = (p-1)*(q-1)
26    n = p*q
27    e = find (phi ÷  5)
28    d = invm phi e
29    find x
30       | g == 1 = x
31       | otherwise = find ((x+1) `mod` phi)
32            where (g,_) = mInverse x phi
33
34 ersa (PUB n e) x = expm n x e
35 drsa (PRIV n d) x = expm n x d
36
37 prim1= 25431045773477
38 prim2= 25431045773503
```

This section is about Haskell, not cryptography. However, the authors have chosen cryptography to show Haskell in action, as it is an interesting problem with a strong mathematical biasing that is apt at revealing the best features of functional programming.

As already mentioned, there are two keys, a public and a private one. Let e be the public key, d the private key, and n the product of p and q (two large prime numbers). The public and the private keys must obey the following relationship: $d \times e = 1 \bmod \phi(pq)$, i.e., $d \times e = 1 + k \times \phi(n)$. In this case, the cipher text c is given by $c = m^e \bmod n$.

Decryption of a message is possible because $c^d = m^{ed} \bmod n$. Since $d \times e = 1 + k \times \phi(n)$, one has:

$$m^{ed} = m^{1+k\phi(n)} = m(m^{\phi(n)})^k = m \bmod n = m, \text{ if } m < n$$

V. Proof of correction

Since Haskell has a strong mathematical foundation, it is possible to use mathematical principles to prove that a Haskell program is correct. This is a very desirable feature in critical systems, where a mistake can be fatal or very costly. In the following paragraphs, a detailed explanation on how this can be done for the `mInverse` function is presented.

Open Science Index, Computer and Systems Engineering Vol:3, No:3, 2009 publications.waset.org/15962.pdf

Open Science Index, Computer and Systems Engineering Vol:3, No:3, 2009 publications.waset.org/15962.pdf

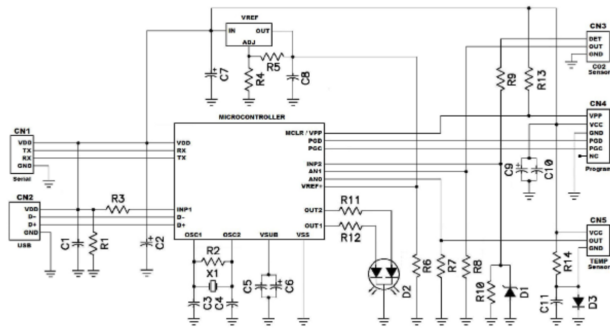

Fig. 2. Micro-controller for thermography and capnography

In real life, programs and electronic circuits are not obvious. Therefore, engineers and computer scientists need Mathematics to prove that their programs are correct. Let us prove that `mInverse` finds the multiplicative inverse of a number, i.e., if `mInverse d f ==1` and `d<f`, then there exists d^{-1} such that $d \times d^{-1} = 1 \mod f$. As one can see, this property is the basis of public cryptography, as described in the previous section.

Function `mInverse` relies on a loop to find the multiplicative inverse of its first argument. To prove that a loop is correct, one needs to search for relationships that remain invariable at each interaction. These relationships are given the name loop invariants. In the case of the local loop in the `mInverse` function, the invariants are:

1) $f \times x1 + d \times x2 = x3$
2) $f \times y1 + d \times y2 = y3$

Both invariants are true at the beginning of the loop, since $(x1, x2, x3) = (1, 0, f)$ and $(y1, y2, y3) = (0, 1, d)$. Let us assume that the invariants are true within iteration n. In this case, they are true within iteration $n + 1$ as well. After all, $f \times x1_{n+1} + d \times x2_{n+1} = f \times (x1_n - q \times y1_n) + d \times (x2_n - q \times y2_n) = (f \times x1_n + d \times x2_n) - q \times (f \times y1_n + d \times y2_n) = x3 - q \times y3 = x3_{n+1}$. The same argument can be used to prove that, if the relationship is valid for $(y1_n, y2_n, y3_n)$, then it is also valid for $(y1_{n+1}, y2_{n+1}, y3_{n+1})$.

The result that we have proven shows that, since the invariant is true for iteration 0, it is true for iteration 1; since it is true for iteration 1, it is true for iteration 2; and so on. In particular, it is true for the last iteration. However, in the last iteration, $y_3 = 1$, and the second invariant takes the following form: $f \times y1 + d \times y2 = 1$. Then, $d \times y2 = 1 + (-y1) \times f$, and $d \times y2 = 1 \; mod \; f$, which leads to the conclusion that $y2$ is the multiplicative inverse of d.

VI. Sensors

The main contribution of this paper is to propose of an architecture for a network of sensors that can be used in monitoring medical patient and climate changes.

Since different physiological signals require different amplifications and noise filterings, both the amplifier and the filter must be programmable.

Figure 3 presents a schematic diagram of a programmable, 5th order, low pass, switched capacitor filter. The clock controls the corner frequency: $f_C = f_{CLK}/100$. The clock signal is generated by the pulse width modulation module of a micro-controller.

In order to produce an example which illustrates the algorithm, a simplified version of the hardware was designed and built. Figure 2 shows the schematics of the simple processor, that can be used for data acquisition of temperature time series and capnograms. The complete system can deal with the following kinds of signal:

- Electrocardiography (ECG) — acquisition of the electrical activity of the heart over time captured and recorded by skin electrodes.
- Electroencephalography (EEG) — recording of electrical activity of firing brain neurons.
- Electromyography (EMG) — recording muscle electrical activation.
- Thermometry — time series of body or environmental temperature on a single point of a surface.
- Capnography — monitoring of the partial pressure of carbon dioxide (CO_2) in the respiratory gases. It is used for patient monitoring during anaesthesia and intensive care. Figure 4 shows a normal capnography.

The authors of the present paper became interested in temperature and CO_2 measurements during their study of the contribution of greenhouse gases to the urban heat island, which is a metropolitan area significantly warmer than its surroundings. They believe that urban heat islands may have a negative impact on global warming and climate change. After designing and building sensors for climatology studies, they were contacted by medical doctors who were studying the correlation between global warming and respiratory diseases. From these interactions, there came the idea of using the same sensors both for measuring air quality and the respiratory patterns of medical patients with ailments related to climate change.

Communication and control of each sensor is performed by a small computer, as shown in figure 2. A sensor has hundreds of parameters that one must adjust in order to get the correct readings. These parameters are stored in an EPROM memory, inside the sensor microcontroler. Since there is a relatively small number of times that one can rewrite an

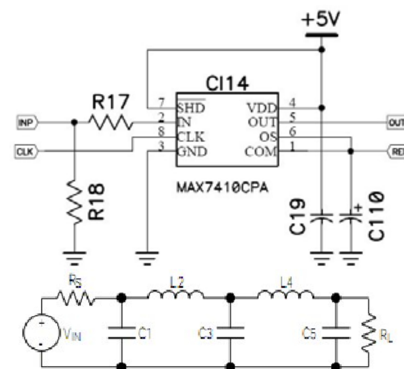

Fig. 3. Programmable Filter

Fig. 4. Normal capnography

EPROM memory, the microcontroler sends the parameters and raw data to a training program, hosted in the main computer. Let us examine the communication program that makes this process possible.

```
1  getTData outh n | n<1 = return ()
2  getTData outh n = do
3     tx ∈ sendMessage 1 "t"
4     let tp = filter (▷ ' ') tx
5     rawVal ∈ sendMessage 1 "x"
6     let raw = filter (▷ ' ') rawVal
7     let p= show (hex2dec raw)
8     threadDelay 500000
9     hPutStrLn outh (tp ++ " " ++ p)
10    getTData outh (n−1)
```

On line 3, the main computer sends a message to the sensors asking for the local temperature; On line 5, it asks for the temperature raw value. Of course, the sensor that one wants to calibrate must have means of measuring a reference value of the quantity that one needs to adjust. This value of reference can be obtained from a previously calibrated sensor. Of course, the first sensor must be adjusted by hand. The program writes a file containing raw measurements, and values calibrated to within an acceptable error.

The sensors must learn how to produce correct readings from collected data. This can be done through evolutionary computation.

VII. Evolutionary computation

A Genetic Algorithm (GA) is an evolutionary process of estimating parameters. Genetic Programming (GP) is a methodology of automatic program generation. Both Genetic Programming and Genetic Algorithms were inspired by biological evolution. They evolve through stochastic processes of change; a few of these processes are explained below:

1) Create generation 0 by random application of program constructors.
2) Step over successive generations until fit individuals appear:
   (a) Execute each individual program in the population and calculate its fitness to solve the problem at hand.
   (b) Selection — uses a fitness measure to rank the population.
   (c) Survival — high ranked snippets survive into the next generation.
   (d) Crossover — creates offspring by recombining snippets.
   (e) Mutation — creates mutant programs through heuristic search.

Below, there is a program that performs approximation of data through Bernstein polynomials, in order to calibrate the sensors.

```
1  {- ghc gaCap.hs −O2 --make -}
2  {- Execute:  gaCap.hs out.txt -}
3  import Data.Array.IO
4  import System.Random
5  import Data.Array
6  import Control.Parallel
7  import Data.Bits
8  import System
9  import Control.Monad.ST
10 import Data.Array.ST
11 import System.IO
12
13 type Sta s = STArray s Int Double
14 type SLD = [(Int, Double)]
15 type AD = Array Int Double
16 type POP= IO (IOArray Int (AD, Double))
17
18  (psz, thr, inf, sup, npar)= (30, 0.01, 0.0, 500.0, 2)
19  (alpha, order, ger) = (0.5, 3, 100)
20
21 main = do
22    let ind= listArray (0, order) [0.0 ..]
23    arr ∈ newArray (1,psz) (ind, 100.0) :: POP
24    args ∈ getArgs
25    case args of
26      [fn] → do
27         contents ∈ readFile fn
28         let dataset= rd (words contents) []
29         xs ∈ rnList (0.0, 1.0)
30         xs0 ∈ gen0 (ind,100.0) dataset arr psz xs
31         ind ∈ readArray arr 1
32          (bb, xs1) ∈
   evolve dataset ind ger arr xs0
33         print bb
34       otherwise → putStrLn "e.g. usus: gbVec training.set"
35
36 rd [] acc= acc
37 rd [x] acc= acc
38 rd (x1:x2:xs) acc= rd
   xs ((read x1, read x2):acc)
39
40 gen0 (b,fb)  s a i xs | i ≤ 1 = do
41       writeArray a i (b,fb)
42       return xs
43 gen0 bfb  s a i xs = do
44   let ind = listArray (0, order)
45              [ inf + x*(sup−inf) | x ∈
   take (order + 1) xs]
46   writeArray a i (ind, fitness s ind)
47   gen0 bfb  s a (i−1) (drop (order + 1) xs)
```

On line 29, an infinite list of random numbers is built to provide the elements necessary to fill the population vector. Each pair in the population is a tuple containing a set of parameters and its fitness.

The `gen0` program goes through the population vector, inserting random generated individuals into the population. The fitness function (line 65) calculates the error that an individual makes while estimating a quantity.

```
48 -- Pascal's Triangle
49 pascal :: [[Double]]
50 pascal  = iterate (λrow →
   zipWith (+) ([0.0] ++ row)
51                                        (row ++
   [0.0])) [1.0]
52
53 -- Binomial numbers
```

Open Science Index, Computer and Systems Engineering Vol:3, No:3, 2009 publications.waset.org/15962.pdf

```
54  bi    :: Int → Int → Double
55  bi n m = pascal !! n !! m
56
57  -- Bernstein's polynomials
58  bernstein :: (Int,Int) → Double → Double
59  bernstein (i,n) t = (bi n i) * (t **
    (fromIntegral i)) *
60        ((1.0 - t) ** (fromIntegral (n - i)))
61
62  rnList :: (Double, Double) → IO [Double]
63  rnList r=getStdGen>>=(λx→return(randomRs r x))
64
65  fitness s c = errSum (poly 0 0.0) s 0.0 where
66    poly i p x | i > order = p
67    poly i p x = poly (i+1) (p+(c!i) *
    bernstein (i,order) x) x
68    errSum p [] acc = acc
69    errSum p ((x,y):xs) acc =
    errSum p xs (acc+(y - p x)^2)
```

The `cross` function takes two individuals from the population, and performs a crossover of their genetic contents, producing an offspring.

```
70  map2 f p q= listArray (0, order) (loop 0) where
71    loop i | i > order = []
72    loop i =  f (p!i) (q!i) : loop (i+1)
73
74  cross (p1,p2) rnl | bounds p1 /= bounds p2 =
    (p1, rnl)
75  cross (p1, p2) (r:rnl)  =
    (offspring, rnl) where
76    offspring = map2 (λ x y → x +
    beta*(y-x)) p1 p2
77    beta = -alpha + r * (1.0 + 2.0 * alpha)
```

The `evolve s (b, fb) i pop xs` program is a loop with five arguments: the training set, the best individual found so far, the generation counter, the population, and an infinite list of random numbers. It searches out two potential parents from the population, generates children, and inserts them into the population if they are fit enough to solve the problem at hand. The program stops if it finds an individual able to solve the problem within a given error. If an apt individual is not found after 100 generations, `evolve` destroys the whole population, and starts everything again; the new start, also known as *The Deluge*, is shown on lines 113, 114, and 115.

```
78  best i b pop xs | i > psz= return (b, xs)
79  best i b pop (r:xs) = do
80     (ix, ifit) ∈ readArray pop i
81     (bx, bfit) ∈ readArray pop b
82     if ifit < bfit && r < 0.2
83       then best (i+1) i pop xs
84       else best (i+1) b pop xs
85
86  findworse i i1 pop | i > psz= return i1
87  findworse i i1 pop = do
88     (_, fi) ∈ readArray pop i
89     (_, f1) ∈ readArray pop i1
90     if fi > f1 then findworse (i+1) i  pop else
91          findworse (i+1) i1 pop
92
93  mutt s (arr, fm) = runST $ do
94    starr ∈ thaw arr
95    grad ∈ loop (bounds arr) starr []
96    newarr ∈ freeze starr
97    let t= 0.02
98    let m2 = accum (λ c g → c - t*g) newarr grad
99    return (m2, fitness s m2)
100  where
101    loop::(Int, Int)→Sta s→ SLD →ST s SLD
102    loop (i, n) arr acc | i >
    n= return $ reverse acc
103    loop (i, n) arr acc = do
104       old ∈ readArray arr i
105       let dx= 0.01
106       writeArray arr i (old+dx)
107       xx ∈ freeze arr
108       let f1 = fitness s xx; writeArray arr i old
109       let g  = (f1 - fm) / dx
110       loop (i+1, n) arr ((i, g):acc)
111
112 evolve s (b, fb) i pop xs | fb < thr =
    return ((b, fb), xs)
113 evolve s bfb i pop xs | i < 1 = do
114   xs0 ∈ gen0 bfb s pop psz xs
115   evolve  s bfb ger pop xs0
116 evolve  s b i pop (r:r1:r2:xs) = do
117   xs2 ∈ parents 10 s pop xs
118   (ib, xs5) ∈ best 1 2 pop xs2
119   bb ∈ readArray pop ib
120   evolve  s bb (i-1) pop xs5
```

It was shown that crossover alone can produce degenerated populations, i.e., populations that don't evolve towards the solution of the problem at hand. Therefore, part of the children must suffer a mutation before insertion into the population. On lines 78 to 110, one can see the mutation function, together with two other functions: `best` and `findworse`. The `best` function finds the fittest individual, i.e., the individual that finds the solution with the smallest error. In fact, to prevent the algorithm from becoming trapped in local optima, the `best` function may drop the fittest individual in favor of another not as well suited; the decision to choose a suboptimal individual is carried out on line 82, and is based on the value of a random number. A child is inserted into the population if the `findworse` function can find a place occupied by an individual not suited for the job.

```
121 parents i s pop xs | i < 1 = return xs
122 parents i s pop (r1:r2:xs) = do
123     let i1 = 1+truncate (r1*(fromIntegral psz))
124     let i2 = 1+truncate (r2*(fromIntegral psz))
125     (t1, ft1) ∈ readArray pop i1
126     (t2, ft2) ∈ readArray pop i2
127     let (m1, xs1) = cross (t1, t2) xs
128     let (m2, xs2) = cross (t1, t2) xs1
129     let (im1,f1) = mutt s (m1, fitness s m1)
130     let (im2,f2) = mutt s (m2, fitness s m2)
131     iw1 ∈ findworse 1 2 pop
132     writeArray pop iw1 (im1,f1)
133     iw2 ∈ findworse 1 2 pop
134     writeArray pop iw2 (im2,f2)
135     parents (i-1) s pop xs2
```

In general, sensors do not output correct results. Therefore, in order to calibrate a sensor, one needs a data cloud containing the sensor measurements, and the corresponding correct value. The calibration process builds a function `f::Measurements →Values` that minimizes

$$\sum_i (v_i - m_i)^2$$

Open Science Index, Computer and Systems Engineering Vol:3, No:3, 2009 publications.waset.org/15962.pdf

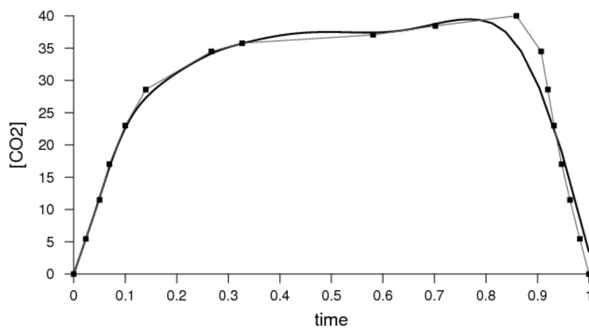

Fig. 5. Normal capnography

where each point (m_i, v_i) belongs to the data cloud, m_i is a measurement, and v_i is the corresponding value.

VIII. Memetic programming

Optimization refers to minimizing or maximizing a scalar, real-valued objective function, by choosing parameters from within an allowed set. In the present case, the objective function measures the error of a polynomial expansion that calibrates a sensor in order to obtain correct measurements from raw data. In figure 5, reference values are shown as small filled squares. The calibration model curve, found by a genetic programming system, is the continuous bold line.

The calibration curve is a series expansion that uses Bernstein polynomials as base functions. The authors have used a similar model in their study of the climatology of F region zonal plasma drifts over Jicamarca (see [14]). In that study, they have described the basic properties of Bernstein polynomials briefly. For ready reference, the reader will find the same description below.

Bernstein[15] polynomials are named after the Ukrainian mathematician Sergei Bernstein, who used them to prove the Weierstrass approximation theorem. They can represent monotone, piecewise smooth functions having left and right derivatives at every point, without the occurrence of the Gibbs phenomenon. The Gibbs phenomenon is the peculiar behavior of the Fourier series of a piecewise continuously differentiable periodic function at a jump discontinuity; near the jump, the nth partial sum of the Fourier series shows an overshoot that does not die out as the frequency increases, but approaches a finite limit.

As Fejer[14] says, Bernstein polynomials provide global approximations of data point clouds in contrast to local approximations given by splines and other popular methods. They are particularly useful for modeling incomplete and noisy data sets, which are not well suited for local approximation.

The learning algorithm is based on the calculation and correction of the error made by the predicting function. The error of a given set of examples is calculated by the `errSum`-function, which implements the expression below, where e is an example, v_c is the predicted value, and v is the given value.

$$\sum_e (v_c(e) - v(e))^2$$

Function `mutt` updates the weights through the gradient descent method in such a way as to reduce the error to a minimum. Gradient descent is an optimization algorithm that finds a local minimum of a function by taking steps proportional to the negative of the gradient of the function at the current point. At each step of the gradient descent method, the weight is updated according to the following formula:

$$\omega_{n+1} = \omega_n - \gamma \nabla \textbf{error}(\omega)$$

In this formula, if $\gamma > 0$ is a small enough number, $\textbf{error}(\omega_{n+1}) < \textbf{error}(\omega_n)$. If one starts with ω_0, the sequence $\omega_0, \omega_1, \omega_2 \ldots$ will converge to a minimum. The minimum will be reached when the gradient becomes zero (or close to zero, in practical situations).

If the gradient descent is the only optimization method used to find the calibration curve, all one can hope for is a local minimum for the error. The use of genetic programming prevents the gradient descent to become trapped in local minima. On the other hand, the gradient descent accelerates the convergence, that would become very slow in a purely evolutionary system. This combination of an evolutionary approach with local improvement procedures for problem search is called Memetic Algorithm. The idea comes from a theory by Richard Dawkins, who believes that evolution is not exclusive to biological systems, but can be applied to any complex system that exhibits the principles of inheritance, variation and selection.

IX. Conclusions

This paper presents a method of bringing interchangeable parts to computer software. A computer has two subsystems, hardware and software, where software is by far the most costly in price and human labor. When hardware becomes obsolete, one must update software as well. By using a computer language founded in Mathematics, developers can be sure that their work will not become obsolete, and can be adapted to any hardware that technological advances may produce. The best candidate for a mathematical formal system that can be used as a programming language is λ-Calculus.

Public cryptography is a good example of software that is currently used and may well show itself be used in future hardware, since it is based on a modification of Euclid's algorithm for finding the greatest common divisor. The algorithm was designed more than two thousand years ago by the Egyptian born Greek mathematician Euclid, and was published in a book edited by Theon of Alexandria and his daughter Hipatia, who died in the year 415 BC. Nevertheless, this old algorithm is the basis of modern cryptography. The authors of the present paper have shown another benefit of a mathematically founded language: one can prove that the algorithm is correct. In the example, there is a proof that the modification of the Euclid algorithm correctly finds the modular inverse of a number.

Designing software that resists obsolescence is especially desirable in areas where homologation takes a long time, and is a costly process, such as in medicine. This paper gives examples of medical instruments with software written in Haskell, a language based on λ-Calculus. Without measures against

Open Science Index, Computer and Systems Engineering Vol:3, No:3, 2009 publications.waset.org/15962.pdf

obsolescence, one is condemned to use vintage hardware in order to preserve homologated software.

Air traffic control and avionics provide further examples of areas where obsolescence is a problem. One of the components of air traffic control is the terminal radar approach control facilities (TRACOM). As late as 1995, and possibly even today (2009), some of the busiest TRACONs used a microprocessor designed around 1980 (see [11]). Costa[12] et al. suggest the use of a sophisticated documentation system to ease the task of updating air traffic control systems. However, the best solution is software that does not require updating; mathematically founded languages can provide such software.

A fringe benefit of mathematically founded languages is a set of tools that can prove the correctness of a program, which stops in due time and delivers the specified result. In fact, code that can be proved correct is vitally important in critical applications and becomes the main justification for programming in λ-Calculus. One can read more about the use of Haskell in critical applications in [13]; in a sense, medical instruments like those described in the present paper are also examples of critical applications.

As for the method used to solve the ergodic system, the genetic programming algorithm prevents the solution from becoming trapped in local minima, while the gradient descent method provides fast convergence.

REFERENCES

[1] J. H. Lienhard. Interchangeable Parts. April 18, 2009. http://www.uh.edu/engines/epi1252.htm.
[2] J. Hughes. The Computer Journal, ISSN:0010-4620, Volume 32, Issue 2 (April 1989), Pages: 98-107.
[3] A. Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, Series 2, 33:346-366, 1932.
[4] A. Church. A note on the Entscheidungsproblem. *Journal of Symbolic Logic*, 1:40-41, 1936.
[5] A. Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58:345-363, 1936.
[6] John Backus. A Functional Style and Its Algebra of Programs. *Communications of the ACM*. August 1978. Volume 2 i. Number 8.
[7] Moses Schonfinkel. Uber die Bausteine der mathematischen Logik. *Mathematische Annalen* 92, pp. 305-316.
[8] Henk Barendregt, The Impact of the Lambda Calculus in Logic and Computer Science. *The Bulletin of Symbolic Logic*, Volume 3, Number 2, June 1997.
[9] P. W. Trinder, K. Hammond, H. W. Loidl and S. Peyton Jones. Algorithm+Strategy= Parallelism. *Journal of Functional Programming* 1 (1), January, 1993. Cambridge University Press.
[10] John Koza. Genetic Programming: On the Programming of Computers by Means of Natural Selection. ISBN-10: 0262111705; ISBN-13: 978-0262111706. The MIT Press; First Printing edition (1992).
[11] T. S. Perry. In search of the future of air traffic control. *IEEE spectrum*, 34(8) 18-35.
[12] E. Costa, A. Grings, and M. V. Santos. Documentation Methods for Visual Languages, in Visual Languages for Interactive Computing. Fernando Ferry, editor. ISBN 978-1-59904-534-4. IGI Global. 2008.
[13] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch and S. Winwood. seL4: Formal verification of an OS kernel, *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, Big Sky, MT, USA, October, 2009.
[14] B. G. Fejer, J. R. Souza, A. S. Santos, and A. E. Costa Pereira. Climatology of F region zonal plasma drifts over Jicamarca. *Journal of Geophysical Research*, Vol. 110, A12310, doi: 10.1029/2005JA011324, 2005.
[15] S. N. Bernstein. Démonstration du théorème de Weierstrass fondée sur le calcul des probabilités. *Comm. Soc. Math. Kharkov*, 13, 1-2.

Open Science Index, Computer and Systems Engineering Vol:3, No:3, 2009 publications.waset.org/15962.pdf