LINUX VOICE TUTORIAL

ADA LOVELACE AND THE ANALYTICAL ENGINE

JULIET KEMP

Use the Linux Voice time machine to take a trip to Victorian England, and visit one of the pioneers of the computer age.

Back in the 19th century, if you wanted to do complicated mathematical calculations you had to do them by hand. To speed things up, you could buy printed tables of specific calculations such as logarithms – but as these too were calculated by hand, they were full of errors.

Enter Charles Babbage, mathematician, philosopher, engineer and inventor, who in the early 1820s designed a Difference Engine to do these calculations automatically. The Difference Engine could only add up, so it wasn't a general-purpose 'computer'. It also never existed in Babbage's time, although part of a prototype was constructed. Babbage fell out with his engineer and ran out of funding, so construction stalled around 1833 and was finally abandoned in 1842.

Meanwhile, in 1834 Babbage began to design a more complex machine called the Analytical Engine. This would be able to add, subtract, multiply, and divide, and it is the Analytical Engine that can be considered as the first general-purpose computer. Or could, if it had ever existed: Babbage built a few pieces of prototype, and carried on refining the design until his death in 1871, but never found funding for the full thing. But despite its lack of concrete existence, other mathematicians were interested in it, including Louis Menebrae, and Ada Lovelace, who was already corresponding with Babbage.

Augusta Ada King, Countess of Lovelace

Lovelace had had extensive mathematical training as a child. She first met Babbage in 1833, aged 17, and corresponded with him on mathematics and logic. Around 1841 Luigi Menabrae wrote a 'Sketch' of the Analytical Engine, describing its operation and how one might use it for a calculation. Lovelace was asked to translate it into English; not only did she do that, but at Babbage's request she added her own extensive Notes, which went much further than Menabrae had.

Lovelace probably saw more in the Analytical Engine than Babbage himself had. She suggests, for example that it might act upon 'other things beside number', and that it might be possible to compose music by representing it in terms of the Engine's notation and operations. This jump from a mathematical engine to one that could act on symbols of any sort was visionary and well ahead of her time.

The Notes, importantly, contained the first computer algorithm – a series of steps of operations to solve a particular (in this case mathematical) problem. This is what any computer program does, and is what makes Ada the first computer programmer, even if she was never able to run her program on a real machine.

Installing the Analytical Engine

Although no physical Analytical Engine exists (the Science Museum in London has a working replica of the Difference engine), Fourmilab Switzerland have an emulator available. It runs on Java, so all you need to run it is a JDK. Download the emulator object code from **www.fourmilab.ch/babbage/contents.html**, unzip it, and type **java aes card.ae** from that directory to run the card file **card.ae**.

The emulator is the best guess, based on Babbage's drawings and papers over the years, of how the Engine would have worked. You can also use it as an applet, for which you'll have to download and compile the source code, but we couldn't easily get this to compile. The applet gives a more visual interface.

Basic operations and a first program

The Analytical Engine consisted of the Mill (where processing was done) and the Store (where numbers and intermediate results were held). The Store had 1,000 registers (a far bigger memory than the first 'real' computers had), and the Mill could take in two numbers, conduct an operation on them, and output a single number. The Engine would also run a printing device for output, to avoid errors in transcription. It would be operated by punch cards, as were used in Jacquard looms to weave complex patterns.

To use the emulator, then, we type in punch-card-type instructions to be run one at a time. For ease, you can put any number of cards into a single text file.

There are three types of punch cards:

- **Operation Cards** Tell the Mill to add/subtract/multiply/divide, and can also move the chain of cards forwards or backwards (like a jump or loop instruction).
- **Number Cards** Supply numbers to the Store as necessary.
- **Variable cards** Transfer values between the Mill and the Store.

For engineering reasons Babbage intended these to have three separate hoppers, but in the emulator they

Ada Lovelace was the daughter of Lady Annabella Byron, who was deeply interested in mathematics, and Lord Byron. What would she have thought of the person who's produced Engine code that draws a cat?

go in a single stream. (This is also how Menabrea and Lovelace expressed their example programs.) The emulator 'cards' also allow some flexibility in format. Numbers aren't right-justified and there's no need for leading zeros, as there would be in a real punch card.

A number card looks like this:

```
N001 3
```

This sets column 1 in the Store (which has 0–999 columns) to the value 3.

The Mill has two Ingress Axes and an Egress Axis (plus two auxiliary axes for division, which we'll look at shortly). Once an operation is selected, the Mill will keep doing that until another is selected. The Operations cards are **+**, **-**, **x** or *****, and **/** or the divison sign, which all do what you'd expect.

Finally, the Variable Cards transfer things in and out of the Mill:

L Transfer from Store to Mill Ingress Axis, leaving Store column intact.

Z Transfer from Store to Mill Ingress Axis, zeroing Store column.

S Transfer from Mill Egress Axis to Store column.

The letter is followed by a number specifying the Store column.

A program on the Analytical Engine consists of a chain of cards; each text line in an emulator file is a single card. You submit a card chain to the Attendant, who will check it for errors and 'requests for actions' (such as inserting manually generated loops and subroutines). The chain of cards is then mounted on the Engine and processed.

Let's give it a go! Since The Analytical Engine doesn't lend itself to Hello World, we'll add 2 and 2. Save this as **card1.ae**:

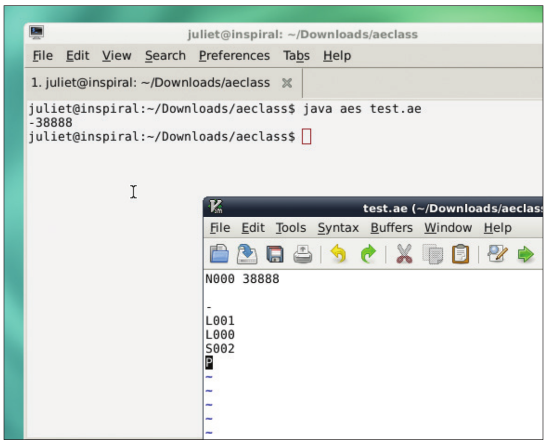
```
N000 2
N001 2
+
L000
L001
S002
P
```

This code puts 2 in column 0 of the Store, 2 in column 1 of the Store, sets the operation to add, transfers column 1 and then column 2 to the Ingress Axes (whereupon the operation will be applied), then the result back to the Store in column 2. **P** prints the result of the last operation to standard output. Run it with **java aes card1.ae** to see what happens.

In fact, you could miss out the second line, and transfer the value from Store column 0 twice, and it will automatically be transferred into both Ingress Axes. So this will work fine:

```
N000 2
+
. About to put values into Mill
L000
L000
S001
P
```

Replacing the first L000 with Z000 won't work, as this zeros the Store column after transfer. This card also includes a comment line. Comments begin with a space or a dot in column 1 of the card.


juliet@inspiral: ~/Downloads/aeclass
File Edit View Search Preferences Tabs Help
1. juliet@inspiral: ~/Downloads/aeclass
test.ae (~/Downloads/aeclas
File Edit Tools Syntax Buffers Window Help

```
juliet@inspiral:~/Downloads/aeclass$ java aes test.ae
-38888
juliet@inspiral:~/Downloads/aeclass$
```

```
N000 38888

-
L001
L000
S002
P
~
~
~
~
~
```

The Analytical Engine emulator running a test card (in the Vim window), which subtracts 38888 from 0.

To do more operations, you need to replace both values on the Ingress Axes – they are discarded after their use in a computation. Each time two arguments go in, the current calcuation is applied.

Menabrae and simultaneous equations

Menabrae in his Sketch described an algorithm to solve a pair of simultaneous equations. He divided the process of solving the equations into a series of individual operations, and tabulated them as Analytical Engine operations. This is handily arranged so that all the multiplications happen, then the subtractions, then the divisions, minimising the number of Operations cards.

Let's translate this into Analytical Engine code. See the LV website for the whole thing; I'll look at the structure and a couple of operations here. Here are our sample equations:

```
2x + y = 7
3x - y = 8
```

First, we put all the numbers (2, 1, 7; 3, -1, 8) into the Store. Then, following Menabrae's calculations, cards 1–6 do all the multiplying and store the results. Cards 7–9 are subtractions. Then cards 10 and 11 generate and print the results. (I've described each operation as a 'card', as Lovelace does, although in the terms of the emulator, each line is a card.)

```
 Card 10 - gives x value
/
L013
L012
S015'
P
 Card 11 - gives y value
L014
L012
S016'
P
```

If you're debugging, it's useful to print at every step.

Division is a little more complicated than other operations. The format is roughly the same, but dividing uses the Primed Egress Output. Specifically,

Ada Lovelace's equation for deriving the Bernoulli numbers.

$$0 = -A_0 + A_2B_2 + A_4B_4 + A_6B_6 + ... + B_{2n}$$

$$0 = -\frac{1}{2}\frac{2n-1}{2n+1} + (\frac{2n}{2})B_2 + (\frac{2n(2n-1)(2n-2)}{2.3.4})B_4 + (\frac{2n(2n-1)(2n-3)(2n-3)(2n-4)}{2.3.4.5.6})B_6 + ... + B_{2n}$$

$$0 = -\frac{1}{2}\frac{2n-1}{2n+1} + (\frac{2n}{2})B_2 + (\frac{2n(2n-1)(2n-2)}{2.3.4})B_4 + (A_4\frac{(2n-3)(2n-4)}{5.6})B_6 + ... + B_{2n}$$

the remainder from the operation goes on the regular Egress Output, and the quotient (which is usually what you want) goes on the Primed Egress Output. You get at this by using an apostrophe. (Very large numbers can also use the Primed Ingress Axis.) Run this with **java aes simeqcard.ae** and you should get two numbers output: 3 (the x value) and 1 (the y value).

The dividing shown works fine if you have integer results or only need integer precision. But what if you want a greater precision? The Analytical Engine uses fixed point arithmetic: like a slide rule, it calculates only in whole numbers, and it is the programmer's responsibility to keep track of decimal places. So there is a "step up" and a "step down" operation, which shifts the decimal point either to the right (stepping up x times, or multiplying by 10x) or to the left (stepping down, or dividing by 10x). We just need to change the last two cards:

```
 Card 10 - gives x value
/
L013
<5
L012
S015'
P
 Card 11 - gives y value
L014
<5
L012
S016'
P
```

We must put the decimal point back in to the output ourselves, by manually dividing by 100,000 (105).

Ada and the Bernoulli numbers

The most interesting part of Ada Lovelace's notes on the Menabrae paper describes how to calculate the Bernoulli numbers (a set of numbers of deep interest to theoretical mathematicians) using the Engine. Her diagram of the process is too complicated to reproduce here, but can be seen (with the rest of the Notes) at **www.fourmilab.ch/babbage/sketch.html**. It can, however, be translated into code for the Analytical Engine emulator. Download the full code from the LV website; here we'll look at the structure and ideas.

The non-zero Bernoulli numbers are usually referred to by modern mathematicians as B2, B4, B6, etc. However, Ada Lovelace refers to them as B1, B3, etc. I will refer to them here by the modern numbers (so subtract one if you're comparing with the Notes directly). There are many ways to derive them, but the equation that Lovelace uses is shown, left. Note that the very last Bernoulli number has no accompanying A-equation. What we're trying to calculate.

The important point is that from A2 onwards, each following A-value takes the preceding one and multiplies by another two terms. This makes it possible to construct an iterative process to calculate each succeeding term.

Onwards then to the code! Following Lovelace's diagram, we will put in an already-calculated version of B2, B4, and B6, and will calculate B8, so **n** is 4. As Lovelace was keen to point out, in a 'real' calcuation the Engine itself would have already calculated these values on a previous round of the program, so they're stored in a later register. The first section of the code, then, sets up our numbers. Register 3 holds our **n**, and registers 21–23 the first 3 Bernoulli numbers, multiplied by 10,000 (to allow for later dividing, as discussed above).

```
Cards 1-6 calculate -1/2 x (2n - 1)/(2n + 1). The last three are
the most interesting:
 Card 4: (2n - 1) / (2n + 1)
/
L004
<5
L005
S011'
 Card 5: 1/2 * (2n - 1) / (2n + 1) Y
L011
L002
S011'
 Card 6: -1/2 * (2n - 1) / (2n + 1) Y
-
L013
L011
S013
```

In Card 4, we step the first value up 5 places before dividing, to avoid a rounding error. In Card 5, we take the value stored in the previous step and overwrite it, since it won't be needed again. In Card 6, we take advantage of the fact that any unused register reads 0, to get a minus number by subtracting register 11 from zero. Effectively this switches the sign of the value in step 5, but we store this result in register 13.

Card 7 subtracts one from n. This isn't used in the code as it stands, but it is a notional counter to keep track of whether we need to do another round of calcuation. If we were calculating B2 (so n = 1), then card 7 would give the result 0, and we would be done. Otherwise, it should add 1 to n and go round again. Lovelace presupposed that the Analytical Engine would have a way of detecting a specific result and acting accordingly. (The emulator provides an alternation card to do exactly this.)

Steps 8–10 produce (2n / 2) * B2 (the latter being stored already). Card 11 adds the value from the first stage (A0), and card 12 again checks whether we're finished yet.

The intriguing part is the next stage, cards 13–23. This is the section that could be repeated almost exactly for any stage of the process, however many numbers you wanted to calculate. What you need to calculate each time is:

```
2n . (2n - 1) . (2n - 2) ... / 2 . 3 . 4 ...
```

This is equivalent to

```
2n / 2 . (2n - 1)/3 . (2n - 2)/4 ...
```

The first time we go through the loop, when calculating A3, we can forget about 2n / 2 as we already calculated that on card 9, and saved it in location 011. So we work out 2n - 1 (card 13) and 2 + 1 (card 14), divide them and save the result (card 15; note again that we step up 5 decimal places), and then multiply it with A0 and save this new value in location 11. We then repeat the exercise, with cards 17-20, with (2n - 2) / 4, multiply it with the previous result, and overwrite location 011 again. So, once again, our A-value is stored in location 11.

In card 21, we multiply with our pre-saved value for B4, then add the whole sequence up and save it in location 13. Card 23 once again checks for 0.

At this point, all we need to do is to run cards 13–23 all over again. Because we saved 2n - 2 as our 'new' 2n, in location 6, applying cards 13–16 produces the result (2n - 4)/ 5, just as we want. And the same again for cards 17-20, with (2n - 5) / 6 multiplied in this time. The only change is that in card 21, we have to grab B6 from its location rather than B4. Then we add it all together again. In the code, these second-time-around cards are labelled 13B-23B.

```
 Card 13: 2n - 1 Y
L006
L001
S006
 Card 14: 2 + 1 Y
+
L002
L001
S007
 Card 15: (2n - 1) / (2 + 1)
/
L006
<5
L007
S008'
 Card 16: (2n / 2) * ((2n - 1) / 3) Y
*
L011
L008
S011
 Card 17: 2n - 2 Y
-
L006
L001
S006
 Card 18: 3 + 1 Y
+
L001
L007
S007
 Card 19: (2n - 2) / 4  Y
/
L006
<5
L007
S009'
 Card 20: (2n / 2) * (2n - 1)/3 * (2n - 2)/4  Y
*
L009
L011
>5
S011
 Card 21: B(4) * [Card 20]
L022
L011
>5
S012
 Card 22: A0 + B2A2 + B4A4 Y
+
L012
L013
S013
```

There's only one new thing to notice, which is that in cards 20 and 21 we have to step our result from the multiplication back down by five decimal places, as we're multiplying two stepped-up values together.

The final step is 24, in which we add our saved value from step 23 to a zero register, to give our calculated Bernoulli number. In actual fact, we should be subtracting this from zero to get the sign of the number correct, but Lovelace explicitly chose to ignore this. Once the result is output, remember that you'll also need to manually put in the decimal point, five places to the left. So our result is -0.03341.

This is not far off the 'official' -0.033333333. Try altering the accuracy of our calculations (remember also to alter the accuracy of the stored Bernoulli numbers) to improve the accuracy of the result.

The Analytical Engine emulator also supports looping code, using conditional and unconditional cycle (backing) cards, and straightforward backing/advancing cards; and an if/then clause with the alternation card. See the website for more details, and have a go at rewriting the provided code to loop over one Bernoulli number at a time, up to a given n, generating the result and storing it for the next loop around. Remember that you'll need to calculate A0, A2, and B2 separately, as here (cards 1–12), before you can get into the real 'loop' part. As the emulator is Turing-complete you can also, as Lovelace suggested, produce anything you can translate into Engine-operations; or, as we now think of it, assembly language. In theory you could even write a compiler in Engine code...

Juliet Kemp is a scary polymath, and is the author of O'Reilly's *Linux System Administration Recipes*.