www.logofoundation.org

Fuzzy Logo

by

Brian Silverman and
Michael Tempel

Logo provides idealized environments for exploring ideas. The Turtle Geometry Microworld is a streamlined setting in which people may become acquainted with geometry and mathematics more generally. Feedback is quick and accurate.

```
repeat 4 [forward 100 right 90]
```

reliably and precisely produces a square. The idea of squareness, four equal sides and 90 degree corners, emerges from the process of telling the turtle to draw a square.

But what if Turtle Geometry was not quite so clean and neat? What if **forward** and **right** were somewhat inaccurate? What if these commands didn't work at all? Would Turtle Geometry lose its educational value, or would new learning possibilities emerge instead?

We began with two separate ideas. One was to create a Logo in which **forward**, **back**, **right,** and **left** worked, but with a random error of up to ten percent. **Forward 100** moves the turtle forward anywhere from 90 to 110 steps. This is not unlike the real world in which reliable systems, including the computers on which Logo runs, are engineered using imprecise components. Our expectation was that people would develop strategies utilizing feedback to correct or manage the defects we introduced into Logo.

A second starting point was to make a Logo in which **forward**, **back**, **right,** and **left** didn't work at all. By trying to rebuild Turtle Geometry people would become familiar with trigonometry and explore the relationship between the relative system of Turtle Geometry and the absolute

Cartesian system which is also part of Logo.

As we perpetrated our funny Logos on groups of teachers during workshops, the two topics merged. We also got more than we expected. New ideas came up. The strategies that people used to cope with our broken Logos fell into four distinct categories:

1. **Feedback** - In trying to get to a particular point, make a move. Then see where you actually ended up as compared to where you wanted to be. Use this information to correct your position.
2. **Law of Large Numbers** - Make your move in many small steps rather than one big one. Lots of little errors tend to cancel each other out.
3. **Start Again** - Forget about the messed up Turtle Geometry commands and make your own accurate ones by using the Cartesian system and trigonometry. The Turtle Geometry primitives can be built out of **setpos**, **setheading**, **sin,** and **cos.**
4. **Undo the Damage** - Figure out how we broke Logo and then fix it. This gets into some of the more obscure areas of Logo, such as burying procedures and redefining primitive procedures.

It may be argued that these topics should not all be together in one paper or in one workshop since they are not well related to each other. They are presented together here because they existed together in practice, in workshops.

Let's look at each of the four strategies more closely. We'll refer to our broken versions of the primitive turtle commands as **fuzzy.forward**, **fuzzy.right**, etc. to distinguish them from the well behaved versions of these commands. In workshops we generally redefine the primitives themselves into fuzziness.

---

1. Feedback

Start with **fuzzy.forward 100**. This leaves you somewhere near your goal. You can find out where you are by using the Logo reporter **pos** (or **xcor** and **ycor**). If the turtle starts at **home** with a heading of 0, then

```
fuzzy.forward 100
print ycor
```

will move the turtle forward about 100 steps. In a normal Logo, **forward 100** from **home** would leave you at a **ycor** of 100. In our fuzzy Logo you're somewhere between 90 and 110.

Now, we haven't tinkered with the reporters. **Pos**, **xcor**, **ycor**, **heading**, etc. are all accurate. This is not unlike the real world. You can buy an instrument at your local electronics supermarket that measures resistance to a far greater precision than the tolerances of its own resistors and other components.

Now that you know where you ended up you can plan your next move. Let's say that **ycor** is 95. You're five steps from the goal, so just go **fuzzy.forward 5**. But, you might argue, **fuzzy.forward** is off by up to 10%, so most likely you'll miss again. This is true, but with a 10% error, **fuzzy. forward 5** will be off by half a step at most. Let's take the worst case. **Fuzzy.forward 5** moves the turtle 5.5 steps high by the full 10%. Now you're only .5 steps away.

Fuzzy.back .5 results in an actual step back of between .45 and .55 steps. This leaves you within .05 steps of the goal, an error of only 1/20th of 1%.

After one move the maximum error is 10%. The second move leaves you off by no more than 1%. After the third move you are within 0.1%. You can continue if you wish, but this accuracy is already greater than the resolution of the graphics screen.

Starting at **home** with a heading of 0, we can summarize this solution in a procedure:

```
to fixed.forward :amount
name :amount "goal
repeat 3 [fuzzy.forward :goal - ycor]
end
```

Since **ycor** starts at 0, the first **fuzzy.forward** tries to go the full amount. In the second repetition the input to **fuzzy.forward** is the difference between the goal and the y-coordinate that resulted from the first **fuzzy.forward**. The third **fuzzy.forward** uses the difference between the goal and the y-coordinate resulting from the second **fuzzy. forward**.

Three repetitions is arbitrary. The more corrections, the greater the precision.

It is possible to overshoot the goal. In that case **ycor** will be greater than **:goal**. The input to **fuzzy.forward** will be negative so the turtle will move back.

This version of **fixed.forward** only works when the heading is 0 and the starting y-coordinate is 0. Here's a more general solution:

```
to fixed.forward :amount
name list
(:amount * sin heading)
(:amount * cos heading)
"goal
repeat 3 [fuzzy.forward distance :goal]
end
```

Actually, the first line of this procedure is by itself a fix for **fuzzy.forward**. Look at strategy number 3 below. Here's another approach using feedback:

```
to fixed.forward :dist
if :dist < 0
[rt 180 fixed.forward - :dist rt 180 stop]
name pos "start
if :dist < 0.01 [stop]
fuzzy.forward :dist
fixed.forward :dist - distance :start
end
```

Fixed.forward needs a **distance** procedure:

```
to distance :point
op sqrt sum
sq xcor - first :point
sq ycor - last :point
end

to sq :n
op :n * :n
end
```

This **fixed.forward** stops when we are within 0.01 turtle steps of the goal. We could have insisted on greater accuracy by using 0.001. What is of more interest is that we have chosen to use an absolute value rather than a percentage of the original distance. We limit our absolute error to .01 turtle steps. **Fixed.forward 100** would be precise to 0.01%. But **fixed.forward 10** would only be accurate to within 0.1% and **fixed.forward 1** would be within 1%. Here's a version that achieves an accuracy of 0.1% regardless of the distance being attempted:

```
to fixed.forward :dist
name 0.001 * :distance "tolerance
fx.fd :dist
end

to fx.fd :dis
if :dis < 0 [rt 180 fx.fd - :dis rt 180 stop]
name pos "start
if :dis < :tolerance [stop]
fuzzy.forward :dis
fx.fd :dis - distance :start
end
```

This is not necessarily a better solution. You just have to decide whether absolute or percentage error is important to you. This depends upon what you're doing. If you plan to attend a concert that begins at 8 o'clock and you want to avoid waiting more than a few minutes at the theatre, you will need to arrive within, let's say, five minutes either side of 7:55. If the concert begins in an hour this represents a much higher percentage error than if the concert is a year from now, but the

absolute error is what counts.

On the other hand, you may want to know how fast you can run over a given distance. If the distance is 10 kilometers, an absolute error in measurement of 10 meters would throw your calculation off by 0.1%. If the distance were 100 meters the same 10 meters would represent a 10% error. In this case, percentage error is more important.

Now there's still a minor bug in **fixed.forward**. You might not want to let the turtle overshoot its goal since this leaves extra little lines sticking out of the corners of your drawings. There is a solution. Don't try to go **fuzzy.forward** the full distance. Instead, try a distance that will bring you to your goal when a maximum error occurs on the high side. In other words, 110% of the amount you choose should bring you to your goal.

```
fuzzy.forward 1 / 1.1 * :amount
```

will do the trick. Instead of an error of plus or minus up to 10% you get results that range between about 82% of the goal and the goal itself. The trade-off for not overshooting is that you potentially have a larger error to correct for. This really isn't a problem since you can achieve any degree of accuracy you need by making more moves at your target.

This strategy also allows us to simplify **fixed.forward**. Since :**dist - distance :start** will only be negative if **fuzzy.forward :dist** overshot the goal, we no longer need the first line of the procedure.

Here's an appropriate modification of the version of **fixed.forward** that achieves an absolute level of accuracy.

```
to fixed.forward :dist
name pos "start
if dist < 0.01 [stop]
fuzzy.forward 1 / 1.1 * :dist
fixed.forward :dist - distance :start
end
```

How can we fix **fuzzy.right** using feedback? Here's a solution that is similar to the fix for **fuzzy.forward**:

```
to fixed.right :turn
if (abs :turn) < 0.01 [stop]
name heading "start
fuzzy.right :turn
fixed.right :start - :turn
end
```

```
to abs :num
op if :num < 0 [-:num] [:num]
end
```

There are many examples of feedback mechanisms in the real world. A thermostat controls room temperature in this way. The heating system just pumps out heat. It has no way of dispensing well-measured amounts. The thermostat can measure the actual temperature. When it gets above a certain point, the heating system is shut down. This causes the temperature to drop and when it reaches a predetermined low point the heat goes back on. You can decide how precisely to control the heat by adjusting the interval between the "turn on" and "turn off" temperatures.

The amount of water going into a toilet's tank is controlled by feedback. After each flush, the tank must fill with a fixed amount of water, enough for the next flush, but not so much as to overflow. The amount of water is never actually measured. Instead, the tank just starts to fill. As the water level rises, the float attached to the shutoff valve also rises, eventually shutting the incoming water flow when it reaches a certain height.

Cruise controls on cars sense changes in speed and adjust the amount of gasoline being fed to the engine.

The average temperature of the world is a fairly pleasant 15 degrees Celsius (59 degree Fahrenheit). However, because of uneven heating of the Earth's surface, there are wide variations from this mean over time and in different places. The system's attempts to even things out create weather.

Lieutenant Island in Welfleet harbor on Cape Cod hosted populations of rabbits and foxes. The foxes controlled the size of the rabbit population by eating them. A well-fed fox population would increase in size and eat more rabbits. The reduction in the rabbit population would cause some foxes to go hungry and die before reproducing. The drop in the fox population would allow more rabbits to survive.

This natural feedback mechanism kept the populations stable within fairly narrow limits until two years ago when an island resident shot all the foxes. The island is now overrun by rabbits and a new feedback mechanism may be coming into play, balancing the rabbit population with the availability of edible vegetation.

This raises an interesting question. When do stable feedback mechanisms break down? Often some additional factor is introduced into the system. If you open all your windows on a cold day the temperature may drop far below the minimum setting of your thermostat.

2. Law of Large Numbers

When we first created our fuzzy Logo we tried some familiar turtle graphics activities.

```
repeat 4 [fuzzy.forward 100 fuzzy.right 90]
```

produced something that looked like it was trying to be a square.

```
repeat 3 [fuzzy.forward 100 fuzzy.right 120]
```

almost gave us a triangle.

Then we tried

```
repeat 360 [fuzzy.forward 1 fuzzy.right 1]
```

which to our initial surprise, produced an almost perfect circle! Why?

Well the 360 errors in **fuzzy.forward 1** and in **fuzzy.right 1** tended to cancel each other. **Fuzzy.forward 100** has an even chance of moving the turtle anywhere between 90 and 110 steps. However, **repeat 100 [fuzzy.forward 1]** is very unlikely to end up far off.

To help see why, let's flip some coins. One coin gives you two possibilities, Head or Tail. With two coins there are four combinations (**HH HT TH TT**). Extending the pattern we have:

Number of combinations of N coins that contain H heads

number of coins (N) \ number of heads (H)	0	1	2	3	4	5	6	7	8	9	10
1	1	1									
2	1	2	1								
3	1	3	3	1							
4	1	4	6	4	1						
5	1	5	10	10	5	1					
6	1	6	15	20	15	6	1				
7	1	7	21	35	35	21	7	1			
8	1	8	28	56	70	56	28	8	1		
9	1	9	36	84	126	126	84	36	9	1	
10	1	10	45	120	210	252	210	120	45	10	1

Let's examine how this works with ten flips. First of all, how many possible outcomes are there for these ten flips? The first flip has two possible outcomes. For each of these two, there are two possible outcomes of the second flip, making a total four possibilities (**HH HT TH TT**). With three flips there are eight combinations (**HHH HHT HTH THH HTT THT TTH TTT**). With

ten flips there are 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 = 1024 possibilities.

Now what's the probability of getting "about half" heads as compared to getting "mostly heads"? Let's say that "about half" of ten coins is 4, 5 or 6, and that "mostly heads" is 8, 9 or 10. The probability of getting about half heads is 210 + 252 + 210 out of 1024, or 672 / 1024 = 0.656, a two out of three chance. The probability of getting "mostly heads" is 1 + 10 + 45 out of 1024, or 56 / 1024 = 0.055, only a little better than one in 20.

---

3. Start Again

A third approach to working with our fuzzy Logo is to forget about the broken turtle commands and start again from scratch. Try to build new turtle commands out of other parts of Logo.

Turtle geometry is a relative system. **Forward** moves the turtle some amount from where it is. There is no reference to an absolute frame of reference. Similarly, **right** rotates the turtle some amount from the direction in which it is headed. It doesn't matter which way it was pointing.

Another geometry that is also built into Logo is the Cartesian system. There is a frame of reference in which position is referred to by x and y coordinates and orientation is indicated by headings of 0 to 360 degrees. **setpos** (or **setx** and **sety**) sends the turtle to a given place on the screen regardless of where it was. **Setheading** orients the turtle independent of which way it was initially pointing.

There are also reporters that tell you where the the turtle is (**pos**, **xcor,** and **ycor**) and which way it's pointing (**heading**). The relative system of turtle geometry can be built by using these reporters to tell you where you are and which way you're heading. Based on that information, the absolute commands may be used to change position and heading.

Let's start with the simple case in which the turtle is at "home" (a position of [0 0]) and has a heading of 0.

We can write a **fixed.forward**:

```
to fixed.forward :distance
sety :distance
end
```

This works only for the first move.

```
to fixed.forward :distance
sety ycor + :distance
end
```

works as long as the heading is 0. Another special case is when the turtle has a heading of 90.

```
to fixed.forward :distance
setx xcor + :distance
end
```

In general we need to distribute **:distance** proportionally into x and y components depending upon heading. If heading is 0, then the full **:distance** is in the y direction. At a heading of 5 degrees, there's some movement in the x direction, but it's mostly in the y direction.

Trigonometry takes care of the apportioning of **:distance** into the proper x and y components. In fact that's exactly what **sin** and **cos** do for a living. A general **fixed.forward** is

```
to fixed.forward :distance
setpos list
(:distance * sin heading)
(:distance * cos heading)
end
```

We can build a **fixed.right** out of **heading** and **setheading**:

```
to fixed.right :turn
setheading heading + :turn
end
```

---

4. Undo the Damage

In presenting fuzzy Logo we generally give people a Logo disk with a "**startup**" file that automatically creates the fuzzy turtle commands. When Logo is booted, it looks for a file named **startup**. If such a file is found it is automatically loaded. When any file is loaded, Logo looks for the special name **startup**. If it is found, and if it is the name of a list, that list is run as a list of instructions.

This means that you can automatically cause a file to load and procedures to run upon booting Logo. We created a **startup** file that contained the procedures needed to redefine **forward**, **back**, **right,** and **left** into fuzziness and then erase most of the evidence of our crime. However, there were enough clues left for some people to figure out what we did.

Here's what was in the Apple Logo *II* version of our startup file:

```
to setup
makeolds
makenews
er [setup makeolds makenews]
er [cfd crt cbk clt]
er "startup
end
to makeolds
copydef "fd "oldfd
copydef "bk "oldbk
copydef "rt "oldrt
copydef "lt "oldlt
end

to makenews
copydef "cfd "fd
copydef "cfd "forward
copydef "cbk "bk
copydef "cbk "back
copydef "clt "lt
copydef "clt "left
copydef "crt "rt
copydef "crt "right
end

to cfd :n
oldfd nearly :n
end

to cbk :n
oldbk nearly :n
end

to clt :n
oldlt nearly :n
end

to crt :n
oldrt nearly :n
```

```
end

to cleft :n
oldlt nearly :n
end

to cright :n
oldrt nearly :n
end

to cback :n
oldbk nearly :n
end

to cforward :n
oldfd nearly :n
end

to nearly :n
output .9 * :n + :n * 2.n3 * random 100
end

make "startup [setup buryall]
```

The last line of the file shows that **startup** is the name of the list **[setup buryall],** so these two procedures are run when the file is loaded.

Setup first calls **makeolds** which uses **copydef** to make copies of the four turtle commands. These copies have the prefix **old**. Then **makenews** copies our new procedures onto the original names. Each of these new procedures (**cforward**, **cright**, etc.) uses an original turtle command, now renamed with the **old** prefix, along with **nearly** to create a fuzzy version.

The next step is to erase the procedures **makeolds**, **makenews** and **setup**. Then **cfd**, **cbk**, **crt** and **clt** are erased. They're no longer needed since their definitions have been copied onto the original primitive names.

Then the name **startup** is erased in the last line of **setup**. Even though it is erased, the list it named continues to run until completed. The primitive command **buryall** buries the procedures that have not been erased, **oldfd, oldbk**, **oldrt**, **oldlt,** and **nearly**. This means that these procedures will not respond to the commands **edit**, **po**, **pots, pops**, **poall**, **save,** or **erase**. Although this hides our dirty work to some degree, knowledgeable Logo people have strategies for finding out what we did.

After **poall** turns up a blank, some people try **unburyall** and then look again. Even though part of the contents of the file has been erased, enough is left, especially **nearly** to figure out what's

going on.

Pofile "startup will display the full contents of the file as it was created, just as listed above. (We could have prevented this by including the command **erasefile "startup** in the **startup** list.)

People who are not familiar with **bury**ing or using **pofile** may still stumble upon our procedures. This has often happened when someone **stop**s a procedure during execution. Logo will tell you what procedure you are in when you press the **stop** character. This may be **nearly** or **oldfd**. With these names revealed, the procedures may be printed out and edited.

One may restore Logo to normalcy by using **copydef** to restore the old definitions to their original names. For example:

```
copydef "oldfd "forward
```

fixes **forward**.

---

5. Summary

We started with a broken turtle which did not seem to have much to do with anything outside of itself. As we played with it and gave it to others to play with, we found that we had created a microworld for exploring geometry, trigonometry, probability, statistics, feedback and things like that.

According to Bob Lawler, microworlds are built out of "neat phenomena" and "powerful ideas." We took a neat phenomenon as our starting point. It spun off a number of powerful ideas, some intended, others surprising.