The Babylonian method for computing square roots

Eric Martin, CSE, UNSW

COMP9021 Principles of Programming

Let a and x be strictly positive real numbers. Let $(x_n)_{n\in\mathbb{N}}$ be the sequence defined as:

- $x_0 = x$;
- for all $n \in \mathbb{N}$, $x_{n+1} = \frac{1}{2}(x_n + \frac{a}{x_n})$.

If $x_n = \sqrt{a}$ for some $n \in \mathbb{N}$, then clearly $x_m = \sqrt{a}$ for all $m \geq n$. Note that given $n \in \mathbb{N}$, if $x_n < \sqrt{a}$ then $\frac{a}{x_n} > \sqrt{a}$, and if $x_n > \sqrt{a}$ then $\frac{a}{x_n} < \sqrt{a}$, so x_{n+1} is the average of a number that is smaller with $\sqrt{a}$ with a number that is greater than $\sqrt{a}$. Actually, $(x_n)_{n\in\mathbb{N}}$ quadratically converges to $\sqrt{a}$, as we now show. For all $n \in \mathbb{N}$, set $\varepsilon_n = \frac{x_n}{\sqrt{a}} - 1$. It suffices to show that:

1. $(\varepsilon_n)_{n\in\mathbb{N}}$ converges to 0, and
2. when n is large enough, $\varepsilon_{n+1} < \varepsilon_n^2$.

It is trivially verified by induction that $x_n > 0$ for all $n \in \mathbb{N}$, hence $\varepsilon_n > -1$ for all $n \in \mathbb{N}$. Let $n \in \mathbb{N}$ be given. Then $\varepsilon_{n+1} = \frac{x_n + \frac{a}{x_n}}{2\sqrt{a}} - 1 = \frac{x_n^2 + a - 2\sqrt{a}x_n}{2\sqrt{a}x_n}$. Also, $\varepsilon_n^2 = (\frac{x_n - \sqrt{a}}{\sqrt{a}})^2 = \frac{x_n^2 - 2x_n\sqrt{a} + a}{a}$ and $\sqrt{a} = \frac{x_n}{1+\varepsilon_n}$. Hence $\varepsilon_{n+1} = \frac{\varepsilon_n^2\sqrt{a}}{2x_n} = \frac{\epsilon_n^2}{2(1+\varepsilon_n)}$; in particular, $\varepsilon_{n+1} \geq 0$. It follows that for all $n > 0$:

- $\varepsilon_{n+1} \leq \frac{\epsilon_n^2}{2(1+0)} = \frac{\epsilon_n^2}{2}$
- $\varepsilon_{n+1} \leq \frac{\epsilon_n^2}{2(\epsilon_n)} = \frac{\epsilon_n}{2}$

that is, $\varepsilon_{n+1} \leq \min(\frac{\epsilon_n^2}{2}, \frac{\epsilon_n}{2})$, from which 1 and 2 follow immediately.

The following generator function allows one to generate on demand an initial segment of a sequence of the form $(f(x), f^2(x), f^3(x), f^4(x), \dots)$:

[1]:
```python
def iterate(f, x):
    while True:
        next_x = f(x)
        yield next_x
        x = next_x
```

Applied to $f : x \mapsto x + 3$ and $x = 5$, `iterate()` is a generator for the sequence $(5 + 3, (5 + 3) + 3, ((5 + 3) + 3) + 3, (((5 + 3) + 3) + 3) + 3, \dots)$:

[2]:
```python
S = iterate(lambda x: x + 3, 5)
next(S)
next(S)
next(S)
next(S)
```

[2]: 8

[2]: 11

[2]: 14

[2]: 17

Let x_0 be a strictly positive integer. For all $n \in \mathbb{N}$, let x_{n+1} be $\frac{n}{2}$ if n is even, and $3x + 1$ if n is odd. The Collatz conjecture states that 1 eventually occurs in $(x_n)_{n\in\mathbb{N}}$; equivalently, $(x_n)_{n\in\mathbb{N}}$ ends in $(1, 4, 2, 1, 4, 2\dots)$. If we want to use `iterate()` and get a generator for such a sequence, we can pass as a first argument to `iterate()` a lambda expression, even though lambda expressions cannot contain `if` statements; indeed, we can take advantage of the way Boolean expressions are evaluated, as illustrated below:

[3]:
```
# None, 0, '', {}, [], () all evaluate to False,  2, [3], 4.5 all
# evaluate to True. Only when () is processed can we conclude that the
# expression is false.
None or 0 or '' or {} or [] or ()
# When 2 is processed, and not before, we can conclude that the
# expression is true.
None or 0 or '' or {} or [] or () or 2 or [3] or 4.5
# When 4.5 is processed, and not before, we can we conclude that the
# expression is true.
2 and [3] and 4.5
# When {} is processed, and not before, we can conclude that the
# expression is false.
2 and [3] and 4.5 and {} and [] and () and '' and 0 and None
```

[3]: ()

[3]: 2

[3]: 4.5

[3]: {}

So we can define the sequence of interest with the lambda expression `lambda x: x % 2 == 0 and x // 2 or 3 * x + 1`; we pass it as first argument to `iterate()` to generate the first few members of that sequence for $x_0 = 2$, $x_0 = 3$, $x_0 = 6$, and $x_0 = 7$:

[4]:
```
S = lambda a: iterate(lambda x: (x % 2 == 0 and x // 2) or 3 * x + 1, a)

S_2 = S(2)
[next(S_2) for _ in range(10)]

S_3 = S(3)
[next(S_3) for _ in range(10)]
```

```python
S_6 = S(6)
[next(S_6) for _ in range(10)]

S_7 = S(7)
[next(S_7) for _ in range(20)]
```

[4]: [1, 4, 2, 1, 4, 2, 1, 4, 2, 1]

[4]: [10, 5, 16, 8, 4, 2, 1, 4, 2, 1]

[4]: [3, 10, 5, 16, 8, 4, 2, 1, 4, 2]

[4]: [22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1, 4, 2, 1, 4]

Let us use `iterate()` to compute approximations of the square roots of 2 and 3, starting with initial guesses of 100 and 1,000, respectively:

[5]:
```python
S = lambda x: lambda a: iterate(lambda x: (x + a / x) / 2 , x)

S_100_2 = S(100)(2)
list(next(S_100_2) for _ in range(12))

S_1000_3 = S(1_000)(3)
list(next(S_1000_3) for _ in range(15))
```

[5]:
```
[50.01,
 25.024996000799838,
 12.552458046745903,
 6.35589469493114,
 3.335281609280434,
 1.967465562231149,
 1.4920008896897232,
 1.4162413320389438,
 1.4142150140500531,
 1.41421356237384,
 1.414213562373095,
 1.414213562373095]
```

[5]:
```
[500.0015,
 250.00374999100003,
 125.00787490550158,
 62.515936696807486,
 31.281962230272214,
 15.688932071312008,
 7.940074837656162,
 4.158952514802515,
 2.440143996371878,
```

```
 1.8347898190318692,
 1.7349272417977204,
 1.7320531920705653,
 1.7320508075705185,
 1.7320508075688772,
 1.7320508075688772]
```

We make `iterate()` an **inner function** of a function `square_root()` meant to compute the square root of its first argument, up to a precision given by its second argument:

[6]:
```python
def square_root(a,  ):
    def iterate(f, x):
        while True:
            next_x = f(x)
            yield next_x
            x = next_x

    x = 1
    approximating_sequence = iterate(lambda x: (x + a / x) / 2 , x)
    next_x = next(approximating_sequence)
    while abs(next_x - x) >  :
        next_x, x = next(approximating_sequence), next_x
    return next_x
```

[7]:
```python
square_root(2, 0.000001)
square_root(3, 0.000001)
```

[7]: 1.414213562373095

[7]: 1.7320508075688772