Lecture 05: Continued fractions continued, and Newton's method ++

Last time we looked at continued fractions for irrational numbers like $\sqrt{2}$, and the golden ratio $(1+\sqrt{5})/2$. We that the continued fraction sequences for these numbers are generated by recursion reltaions of the form

$$r_{n+1} = \frac{r_n + 1}{r_n}, \quad \text{where} \quad r_0 \ = \ 1.$$

This is the sequence that generates the golden ratio in the sense that

$$\lim_{n\to\infty} r_n \ = \ \phi \ = \ \frac{1+\sqrt{5}}{2}.$$

For finite n, we know that each r_n is a *rational* number, that means it's the ratio of two integers. Let's see if we can learn something about these integers. Say that

$$r_n \ = \ \frac{i_{n+1}}{i_n},$$

were i_n are just integers. Plug this into the recursion for r_{n+1} and see what happens.

$$\frac{i_{n+2}}{i_{n+1}} \ = \ \frac{\frac{i_{n+1}}{i_n} + 1}{\frac{i_{n+1}}{i_n}} \ = \ \frac{i_{n+1} + i_n}{i_{n+1}}$$

We can cancel the demenator as long as $i_{n+1} \neq 0$. Therefore,

$$i_{n+2} \ = \ i_{n+1} + i_n.$$

This is the famous Fiboncci sequence. We also want $i_0 = i_1 = 1$, since we want $r_0 = 1$.

Translating between math and code

Before I go any further, I noticed somthing in the tutorials. A lot of the newcommers to code were having a hard time putting a mathematical expression into code. I think it's becuse they thought it needed to be complicated.

When you get started, a lot of people tend to show some kind of fear associated with just starting to write code. I know this becuase I remember being in the same situation. I don't know what the reason is, but it's a pretty universal feeling.

But the *great* thing about an interpreted language like Python (using netebooks no less) is that you can make mistakes all day and not hurt anything.

So, if I ask to define a function that does XYZ, and you don't know what that means exactly, just start by defining a function. *Any function*. At least any function that you can get working right away. It can be as simple as $f(x) = x^2$, anything to get the ball rolling. Then once you have the `def` and `return` statements in place you can work on filling in the things you don't understand yet.

How can we translate between math and code?

Take

$$r_{n+1} = \frac{r_n + 1}{r_n}, \quad \text{where} \quad r_0 \ = \ 1.$$

This is somewhat out of order. In code you don't put the initial condition last. You put it at the beginning, that's why it's called "inital".

$$r_0 \ = \ 1.$$

$$r_{n+1} = \frac{r_n + 1}{r_n}$$

Better.

Now you need to be careful how you tell a computer to count. If you are not careful you might tell it to count to infinity and it will never get there. We want a rule for giving us r_n given the values that we already have, i.e., r_{n-1}. So let's make the small shift.

$$r_0 \ = \ 1.$$

$$r_n = \frac{r_{n-1} + 1}{r_{n-1}}$$

Even better.

But remember that r_n is both a number with a subscript n, and a discrete function of n. Always remember: a function is a black box. You give it something (n), and it gives you something back (r_n). It doesn't matter how we *denote* the function, it is still a function. But programming languages don't traffic in subscripts. They almot all like olde fashion function notation; discrete or not. So lets make the change $r_n \to r(n)$.

$$r(0) \ = \ 1.$$

$$r(n) = \frac{r(n-1)+1}{r(n-1)}$$

Looking more `computer` all the time.

But now we notice that there are a few things left unsaid about the second equation. It doens't work when $n = 0$. Becuase then we ask for $r(-1)$, which we don't know and that would ask for $r(-2)$, etc, and it would be off to the races. And we really don't want to have $r(0)$ as a thing. We want $r(n)$ for any legimitate n we might ask for. So

$if \quad n \; = \; 0:$

$$r(n) \; = \; 1.$$

$if \quad n \; > \; 0:$

$$r(n) = \frac{r(n-1)+1}{r(n-1)}$$

We can even make one last reformulation to make it so we don't write $r(n-1)$ more than once.

$if \quad n \; = \; 0:$

$$r(n) \; = \; 1.$$

$if \quad n \; > \; 0:$

$$r(n) \; = \; 1 \; + \; \frac{1}{r(n-1)}$$

This all makes the hidden assumption that n is a non-negative integer. You could make this more explicit if you were worried about someone missing the point.

Here it is in Python code:

In [1]:
```
def r(n):
    if n == 0: return 1
    return 1 + 1 / r(n-1)
```

It takes less Python code to express the function that the math it self!

PS: This is the style of question you might expect on a final exam.

In [2]:
```
r(300)
```

Out[2]:
```
1.618033988749895
```

Kind of cool, notice that it works if the `if` statements are the other way around.

In [3]:
```
def r(n):
    if n > 0: return 1 + 1 / r(n-1)
    return 1

print(r(300))
```

```
1.618033988749895
```

This might be even better because most cases will be `> 0`, and will therefore end sooner in the logic. It doesn't matter at all in this case for speed. But this can make a big difference in bigger programs.

Remember, we got the math into a form that was very easy to translate into code with the following steps

```
1) initial conditions come first.

3) make sure an index counts in a direction that will stop eventually.

2) replace subscripts in favour of function notation, even if everythins is discret
e.

3) replace spcial cases in functions with conditional statements.

4) redo algebra to aviod extra function evaluations. Or use "memory" to store data.
```

Knowing what you've seen in the last couple lectures, it would be straighforward to write a recursion formula for the Fibonacci numbers. *You would have to be careful to not to call the recursion too many times at each stage (look at Lecture 04 if you don't recall why). Otherwise it would be slowed down exponentially!*

But I want to show a slightly different way to do it. We'll explanin more about why this is a good approach as the semester progresses. For now, I want it to be an easy introduction into:

Matrices and Abstraction

Consider the matrix,

$$F = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}.$$

Let's apply F recursively to a 2-dimensional vector.

$$\begin{bmatrix} a_{n+1} \\ b_{n+1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} . \begin{bmatrix} a_n \\ b_n \end{bmatrix}$$

This is the same as

$$a_{n+1} = a_n + b_n, \quad b_{n+1} = a_n$$

We can shift the index of the second equation backwards: $b_n = a_{n-1}$. Alltogether this is the same as

$$a_{n+1} = a_n + a_{n-1}.$$

Another version of the Fibonacci sequence.

This illustrates **a very important principle in abstraction**:

```
A two-term recursion formula for one variable is the same as a one-term recursion f
or two variables.
The same principle applies to more variables. You probably did this in your ODEs cl
ass; it also works here too.

If this recursion formula is linear, then the one-term version can be encoded into
 a matrix.

If we want, we don't even need to worry about the initial conditions until we are d
one.
We can do the recursion driectly on the matrix itself!
```

Let's program this in Python.

I'm going to use the same structure as we saw in Lecture 02 for `fast_exp`. Go back and look at that example if you need to.

Also: the version I'm using only works for non-negative values of n. You should try to figure out how to make this work with negative values of n.

In [4]:
```
import numpy as np

def matrix_power(F,n):

    if n == 0:
        return np.array([[1,0],[0,1]])
    elif n%2==0:
        M = matrix_power(F,n//2)
        return M.dot(M)
    else:
        return F.dot(matrix_power(F,n-1))
```

In [5]:
```
F = np.array([[1,1],[1,0]])
print(F)
```

```
[[1 1]
 [1 0]]
```

In [6]:
```
print(matrix_power(F,10))
```

```
[[89 55]
 [55 34]]
```

What do you notice about the entries of the matrix?

In [7]:
```
phi = (1+np.sqrt(5))/2

def phi_continued_fraction(n):
    M = matrix_power(F,n)
    return M[0][0]/M[0][1]
```

In [8]:
```
# Number of iterations versus digits of accuracy:
for n in range(1,50+1):
    print("(n,digits) =", (n,int(-np.log10(np.abs(phi_continued_fraction(n) - phi)))))
```

```
(n,digits) = (1, 0)
(n,digits) = (2, 0)
(n,digits) = (3, 0)
(n,digits) = (4, 1)
(n,digits) = (5, 1)
(n,digits) = (6, 2)
(n,digits) = (7, 2)
(n,digits) = (8, 2)
(n,digits) = (9, 3)
(n,digits) = (10, 3)
(n,digits) = (11, 4)
(n,digits) = (12, 4)
(n,digits) = (13, 5)
(n,digits) = (14, 5)
(n,digits) = (15, 5)
(n,digits) = (16, 6)
(n,digits) = (17, 6)
(n,digits) = (18, 7)
(n,digits) = (19, 7)
(n,digits) = (20, 8)
(n,digits) = (21, 8)
(n,digits) = (22, 8)
(n,digits) = (23, 9)
(n,digits) = (24, 9)
(n,digits) = (25, 10)
(n,digits) = (26, 10)
(n,digits) = (27, 10)
(n,digits) = (28, 11)
(n,digits) = (29, 11)
(n,digits) = (30, 12)
(n,digits) = (31, 12)
(n,digits) = (32, 13)
(n,digits) = (33, 13)
(n,digits) = (34, 13)
(n,digits) = (35, 14)
(n,digits) = (36, 14)
(n,digits) = (37, 15)
(n,digits) = (38, 15)
(n,digits) = (39, 15)

/usr/lib/python3.6/site-packages/ipykernel_launcher.py:3: RuntimeWarning: div
ide by zero encountered in log10
  This is separate from the ipykernel package so we can avoid doing imports u
ntil

---------------------------------------------------------------------------
OverflowError                             Traceback (most recent call last)
<ipython-input-8-3dd5ddc698ad> in <module>()
      1 # Number of iterations versus digits of accuracy:
      2 for n in range(1,50+1):
----> 3     print("(n,digits) =", (n,int(-np.log10(np.abs(phi_continued_fract
ion(n) - phi)))))

OverflowError: cannot convert float infinity to integer
```

Newton's method

In their simplest form, the numbers $\sqrt{2}$ and ϕ are solutions to simple aglebraic equations such as

$$x^2 - 2 = 0$$

and

$$x^2 - x - 1 = 0$$

What do we learn if we try to just solve these equations, and pretend for the moment that we know nothing of continued fractions.

Newton's method is perhaps the best place when it comes to root finding. You have probably seen it in 1st year, but it's best to review a little.

https://en.wikipedia.org/wiki/Newtons_method (https://en.wikipedia.org/wiki/Newtons_method)

Suppose you have a real function of a real variable, $f(x)$. And you want to find a value $x = a$ such that

$$f(a) \; = \; 0$$

Note: I'm using a as the *specific* value of the solution, and x for any real number that you could plug into the function.

Also suppose you have a reasonable guess $x_0 \approx a$ and

$$f(x_0) \approx 0$$

.

We want to find a new guess that does even better.

$$f(x_1) = f(x_0 + (x_1 - x_0)) \approx f(x_0) + f'(x_0)(x_1 - x_0) \approx 0$$

. Therefore

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

More generally,

$$F(x) \; = \; x - \frac{f(x)}{f'(x)} \; = \; \frac{x\,f'(x) - f(x)}{f'(x)}$$

We therefore make the iteration

$$x_{n+1} \; = \; F(x_n)$$

Let's use Newton's method to find $\sqrt{2}$; a notoriously irrational number:
https://en.wikipedia.org/wiki/Square_root_of_2 (https://en.wikipedia.org/wiki/Square_root_of_2)

Then

$$f(x) = x^2 - 2, \quad F(x) \; = \; \frac{1}{x} + \frac{x}{2} \; = \; \frac{x^2 + 2}{2x}$$

The result should be the stationary solution. Let's find out if $x = F(x)$ has a solution. This is the same as $2x^2 = x^2 + 2$. It looks like everything should work in this case.

Recall from before

$$x_{n+1} \; = \; \frac{x_n + 2}{x_n + 1}$$

Alternatively for ϕ,

$$f(x) = x^2 - x - 1, \quad F(x) \; = \; \frac{x^2 + 1}{2x - 1}.$$

In either case we are (once again) going to iterate,

$$x_{n+1} = F(x_n).$$

Let's do the Newton iteraton for ϕ.

In [ ]:
```
def phi_Newton(n):
    if n==0: return 1
    x = phi_Newton(n-1)
    return (x*x+1)/(2*x-1)
```

Remember how is said we could pass functions to functions. Here is a little helper function that check the number of decimal digits of accuracy for different methods. Remember, 64-bit numbers are only going to be able to cope wiht 16 digits. The helper checks this and retunrs 16 if for all errors less than this 10^{-16}.

In [ ]:
```
def digits(method,n):
    err = np.abs(method(n) - phi)
    if err < 1e-16: return 16
    return int(-np.log10(err))
```

In [ ]:
```
# Number of iterations, digits of accuracy continued_fraction, digits of accur
acy Newton's method:

print('  n   d_c  d_n')

for n in range(1,40+1):
    print(" %2i   %2i   %2i " %(n,digits(phi_continued_fraction,n), digits(phi
_Newton,n)) )
```

The Newton method is converging *a lot* faster than the continued fraction! I thought I said continued fractions are the "fastest converging" way to represent irrational numbers. Was I wrong?

In [ ]:
```
for i in range(1,7):
    print(phi_Newton(i)-phi_continued_fraction(2**i))
```

Both are still true. Continued fractions are still the best. But it's possible to find sequences that *skip* steps in the original sequence.

This is a big ***acceleration***. In this case, it looks like Newton's method is finding the powers of 2 in the continued fraction sequence. Can we prove this? We are *conjecturing* that

$$x_n \ = \ r_{2^n}.$$

I'll leave it as an execise to prove this. The same is true for the $\sqrt{2}$ case.

If we were able to find a seuence that skiped big steps in the continued fraction sequence, is it possible to skip even bigger steps?

Halley's method

The goal of Newton's method is to solve equations of the form

$$f(x) \;=\; 0.$$

Not long after Newton, Sir Edmond Halley (of the Comet fame) realised a simple fact. If $h(x) \neq 0$. Or at least not at the same place as $f(x)$, then solving the modified equation

$$\tilde{f}\,(x) \;\equiv\; \frac{f(x)}{h(x)} \;=\; 0$$

gives the same result as solving $f(x) = 0$. But maybe we can find an $h(x)$ that make Newton's for $\tilde{f}\,(x)$ go faster? Sir Edmond discovered that

$$h(x) \;=\; \sqrt{|f'(x)|}$$

works for this purpose. It works as long as $f(x)$ is twice differentiable and doesn't share any roots with $f'(x)$. After going the calculus for Newton's method, but only for $\tilde{f}\,(x)$, we find that we need to iterate

$$F(x) \;=\; x + \frac{2f(x)f'(x)}{f(x)f''(x) - 2f'(x)^2}$$

In the case of the golden ratio

$$F(x) \;=\; \frac{x^3 + 3x - 1}{3x^2 - 3x + 2}$$

Let's try it in Python. This will use more examples of `functional` programming. See if you can follow the flow of logic and compare it to the mathematical formulation.

In [ ]:
```
# Here we can pass a function
def recur_fun(f,x,n):
    if n==0: return x
    return f(recur_fun(f,x,n-1))

def phi_fun(x):
    return (x**3 + 3*x - 1)/(3*x*x - 3*x + 2)

def phi_Halley(n):
    return recur_fun(phi_fun,1,n)
```

In [ ]:
```
# Number of iterations, digits of accuracy:
for n in range(1,7+1):
    print("(n,d)=", (n,digits(phi_continued_fraction,n), digits(phi_Newton,n),
 digits(phi_Halley,n)) )
print("Boom!")
```

In [ ]:
```
for i in range(1,5):
    print(phi_Halley(i)-phi_continued_fraction(3**i))
```

We can do the same for π if we want. In this case $f(x) = \sin(x)$.

In [ ]:
```
def pi_fun(x):
    return x - (2*np.sin(2*x))/(3 + np.cos(2*x))

recur_fun(pi_fun,3,3)-np.pi
```

Or for e; $f(x) = \log(x) - 1$.

In [ ]:
```
def e_fun(x):
    return x*(4/(1 + np.log(x)) - 1)

recur_fun(e_fun,2,3) - np.e
```

Halley's method goes by *powers of* 3 in the continued fraction sequence. This is exponentially faster than the powers of 2 in Newton's method.

Summary

Newton's method is the most practically useful aspect of this lecture.

```
The golden ratio and continued fraction sequences are to give you more ideas about
 the fundemantal structure of irrational numbers.

Halley's method is not widely used in serious applications. Newton and even simpler
 methods are still the best for most cases.

The point of showing Halley's method is to highlight the idea of acceleration. It m
ight not surprise you that there are even faster methods.
```

You can look up Householder's methods if you want more info:

https://en.wikipedia.org/wiki/Householder's_method (https://en.wikipedia.org/wiki/Householder's_method)

Newton and Helley are the first two in the sequence of Householder methods. The kth method in Householder's formula skips powers of k in the continued fraction sequence.