

Lista de Computação II (Python)

DCC / UFRJ

Pedro Maciel Xavier (monitor)
pedromxavier@poli.ufrj.br

30 de dezembro de 2019

Introdução

Essa lista de exercícios ainda se encontra em desenvolvimento. A intenção é que ela tenha um gabarito bem aberto, deixando muito das respostas para a criatividade do aluno. As questões são, em geral, grandes para se resolver e podem necessitar de alguma pesquisa adicional. Elas tem estrelinhas ★ indicando a dificuldade estimada. Alguns exercícios foram inspirados em outros propostos em materiais cujas fontes estão devidamente referenciadas no final. É importante você tire suas dúvidas e dê um retorno do que achou dos exercícios através do e-mail no cabeçalho.

Boa diversão!



Sumário

1	Orientação a Objeto (class)	4
1.1	O Bidicionário 1★	4
1.2	Ora Bolas! 1★	4
1.3	Frações 1★	5
1.4	Polinômios [3] 2★	6
1.5	Grupos 5★	8
1.5.1	Subparte I: Produto	9
1.5.2	Subparte II: Quociente	9
1.5.3	Subparte III: Subgrupos	9
1.6	Música III - Partituras 4★	10
1.7	Quaterniões! 3★	12
2	Interface Gráfica (tkinter)	14
2.1	O Triângulo de Sierpinsky 2★	14
2.2	Pôr-do-Sol [4] 3★	14
2.3	O Método de Monte Carlo 3★	15
2.4	A proporção áurea 2★	16
2.5	Música IV - O Piano 3★	17
3	Métodos numéricos (numpy)	18
3.1	Computação Quântica 5★	19
4	Plotagem de Gráficos (Matplotlib)	20
5	Conexão e Redes (socket)	21

Revisão de Computação I

Não sei se vai ter isso aqui não em.

1 Orientação a Objeto (**class**)

1.1 O Bidicionário ★

Todos conhecemos os dicionários do Python, que guardam diversos objetos em pares da forma "chave":valor. Vejamos um exemplo:

```
1 >>> casa = {
2     'quartos':4,
3     'banheiros':5,
4     'andares':3,
5     'm^2':210
6 }
7
8 >>> casa['quartos']
9 4
```

Listing 1: "Dicionário Normal"

O objetivo deste exercício é criar um dicionário de mão dupla! Tudo que você precisa fazer é sobrescrever o método `__setitem__` em um novo tipo que herda as propriedades de um dicionário comum do Python, o **dict**. Basta completar o exemplo abaixo!

```
1 class Bidict(dict):
2     def __init__(self, mapping={}):
3         dict.__init__(self, mapping)
4         ...
5 >>> bd = Bidict()
6 >>> bd["a"] = 4
7 >>> bd[3] = "d"
8 >>> print(bd)
9 {"a":4, 4:"a", 3:"d", "d":3}
```

Lembrando que se um objeto não puder ser chave de um dicionário, ele também não poderá ser um valor do bidicionário!

1.2 Ora Bolas! ★

Mais um exercício pra esquentar: Crie uma classe chamada `Bola` que deve implementar objetos com as seguintes características:

1. O raio nominal da bola.
2. A pressão de ar máxima (em *bar*).
3. A pressão de ar atual. (em *bar*).
4. A condição da bola (furada ou não).
5. Uma onomatopeia correspondente ao barulho que a bola faz quando quica, e outra para caso ela fure.
6. A probabilidade da bola furar quando quica.

Além disso, tendo uma bola em mãos você deve poder:

1. Quicar! Caso ela não esteja vazia.
2. Encher em alguma quantidade de *bar*, passada como argumento da função, caso ela não esteja furada. Se você encher de mais ela deve furar!
3. Calcular o seu volume.

Feita a bola, você deve criar uma classe `BolaQuadrada` que herde as propriedades de uma bola comum, mas tenha as adaptações necessárias para o seu formato. Ela deve, por exemplo, ter 50% de chance de quicar em uma tentativa.

Faça também a classe `BolaDeFesta`, que deve furar sempre que quicar. Dê a ela um som de estouro interessante.

1.3 Frações ★

Um número racional $r \in \mathbb{Q}$ é aquele que pode ser escrito como:

$$r = \frac{p}{q}; p, q \in \mathbb{Z}; q \neq 0$$

Desafio: Crie uma classe `Fracao` que implemente as seguintes operações e métodos:

1. Um método que simplifique a fração.
2. As operações:
 - `+` (`__add__`)
 - `-` (`__sub__`)
 - `*` (`__mul__`)
 - `**` (`__pow__`)
 - `/` (`__truediv__`)
 - `-` (`__neg__`)
 - `~` (`__invert__`).
3. `__repr__` que retorna "`p|q`".
4. `__float__`, que calcula a divisão em ponto-flutuante.

```

1 class Fracao(object):
2
3     def __init__(self, p, q):
4         assert type(p) is int
5         assert type(q) is int
6         assert q != 0
7         ...

```

1.4 Polinômios [3] ★ ★

Um polinômio de grau n é aquele da forma

$$p(x) = \sum_{i=0}^n a_i x^i = a_0 + a_1 x + \cdots + a_n x^n.$$

Desafio: Sabendo isso, faça uma classe `Polinomio` que implemente:

1. A definição de um coeficiente do polinômio através do método `__setitem__`, ou seja, `p[2] = 3` faria com que o coeficiente a_2 do polinômio p tivesse valor 3.
2. O cálculo do grau do polinômio p , que deve ser retornado quando chamamos `len(p)`.

3. A soma, a subtração e a multiplicação usual de polinômios usando os operadores da linguagem (+, -, *), que deve retornar um novo objeto da classe `Polinomio`.
4. E por fim, a avaliação da função num ponto x , usando o método especial `__call__`.

Desafio Bônus:

1. O método especial `__repr__` que deve retornar uma **string** que represente o polinômio $2 + x + 3x^2$ na forma `"2 + x + 3x^2"`, por exemplo.
2. Duas funções, `Polinomio.integral` e `Polinomio.derivada` que retornem os respectivos polinômios resultantes destas operações.
3. Divisão de polinômios (f/g e $f\%g$), que devem retornar respectivamente o quociente $q(x)$ e o resto $r(x)$ tais que $f(x) = g(x)q(x) + r(x)$.

Interlúdio: Conjuntos

No Python temos um tipo muito bacana, mas muito mesmo, chamado **set**. Esse carinha simula um conjunto do ponto de vista matemático, ou seja, é uma coleção de objetos distintos. Um **set** pode ser criado usando-se outro objeto ou informando os elementos entre colchetes:

```
1      >>> A = set([1, 2, 3])
2      >>> B = {x for x in range(2, 5)}
3      >>> A
4      set([1, 2, 3])
5      >>> B
6      set([2, 3, 4])
7      >>> A - B
8      set([1])
9      >>> A | B
10     set([1, 2, 3, 4])
```

O **set** permite realizar diversas operações entre conjuntos, sendo as mais importantes: a diferença ($-$), a união ($|$, *ou binário*) e a interseção ($\&$, *e binário*).

1.5 Grupos ★ ★ ★ ★ ★

Em Álgebra, um Grupo $G = (A, *)$ é formado por um conjunto A e uma operação qualquer $*$, quando valem as seguintes regras:

1. $(a * b) \in A \quad \forall a, b \in A$ [O grupo é fechado para a operação.]
2. $\exists e \in A : e * a = a * e = a \quad \forall a \in A$ [Existe o elemento neutro.]
3. $\exists a^{-1} \in A : a * a^{-1} = a^{-1} * a = e \quad \forall a \in A$ [Existe o elemento inverso.]

Construa um objeto Grupo que herda as características do **set**. Devem ser passados ao construtor: os elementos em uma coleção A, e uma função $f(a, b)$ que realiza a operação $*$ sobre dois elementos do Grupo.

```
1 def f(a,b):
2     return (a + b) % 5
3
4 class Grupo(set):
5     def __init__(self, A, f):
6         ...
7
8     def __hash__(self):
9         return hash(tuple(self))
10
11 >>> G = Grupo([0,1,2,3,4], f)
12 >>> G
13 {0, 1, 2, 3, 4}
14 >>> H = Grupo([0,1,2], f)
15 Traceback (most recent call last):
16   File "<pysHELL#1>", line 1, in <module>
17     raise Exception('Isso_não_é_grupo!')
18 Exception: Isso não é grupo!
```


Sabido isso:

1. O construtor deve criar um erro quando as regras de Grupo não forem atendidas.
2. A visualização do grupo mostre os elementos entre colchetes usando o método `__repr__`.

1.5.1 Subparte I: Produto

O produto entre dois grupos é dado da seguinte forma:

$$GH := \{g * h : g \in G, h \in H\}$$

Defina o método `__mul__` para atender a esse propósito.

1.5.2 Subparte II: Quociente

Existe uma operação entre grupos chamada *quociente* definida por:

$$G/H := \{gH : g \in G\} = \{\{g * h : h \in H\} : g \in G\}$$

Essa operação deve retornar um *conjunto de grupos* e isso só é possível se o método `__hash__` estiver definido como no modelo. Implemente essa operação usando o método `__truediv__`.

1.5.3 Subparte III: Subgrupos

Um Subgrupo H de um Grupo G , que escrevemos $H < G$, é aquele cujo conjunto é subconjunto de G e H também um Grupo. O tipo **set** já implementa os operadores `>`, `>=`, `<`, `<=` e `==` comparando dois objetos do ponto de vista dos conjuntos. Reescreva os métodos `__gt__`, `__ge__`, `__lt__`, `__le__` e `__eq__`, respectivamente, para dizer verificar, por exemplo, se H é subgrupo de G através da expressão $H < G$. Feito isso, construa um método da classe Grupo que retorne um conjunto contendo todos os subgrupos de um grupo G .

Dica Teorema:

O *Teorema de Lagrange* diz que sendo G um grupo finito e H um subgrupo de G , a ordem de H divide a ordem de G .

$$H < G \implies |G| = |H| \cdot k, k \in \mathbb{N}$$

A ordem de um grupo G , escrita como $|G|$, é simplesmente o número de elementos em G , que pode ser obtida diretamente pela expressão **len**(G).

1.6 Música III - Partituras ★ ★ ★ ★

A nota *lá* (A4) na musica ocidental corresponde a frequência de 440Hz. A partir desta nota podemos conhecer todas as outras frequências usando a simples fórmula:

$$f(n) = 440 \times \sqrt[12]{2^n}$$

A escala musical *cromática* é dividida em 12 semitons, cada um correspondente a uma nota:

..., fá#[-3], sol[-2], sol#[-1], lá[0], lá#[1], si[2], dó[3], dó#[4], ré[5], ré#[6], mi[7], fá[8], fá#[9]...

Essa sequência se repete nos dois sentidos e símbolo # lê-se *sustenido*. Assim, se queremos a frequência de um *dó* logo após o *lá* central (A4), basta calcular $f(3) \approx 523.25\text{Hz}$.

Note que se temos alguma nota cuja frequência é $f(\bar{n})$, ao avançarmos 12 semitons na escala cromática voltamos para a mesma nota, mas com frequência $f(\bar{n} + 12)$. Ao tirar a razão entre as duas frequências:

$$\frac{f(\bar{n} + 12)}{f(\bar{n})} = \frac{440 \times \sqrt[12]{2^{\bar{n}+12}}}{440 \times \sqrt[12]{2^{\bar{n}}}} = 2$$

Isso nos diz que cada vez que avançamos até a próxima repetição de uma nota, dobramos a frequência! Na escala de *dó maior*, que todos sabemos de cor, isso equivale a avançar uma *oitava acima*.

Interlúdio: Beep

Um jeito fácil de fazer barulho no computador é usando a função Beep. Se você usa o Python no Windows boas notícias: você só vai precisar importar a função Beep do módulo winsound.

```
>>> Beep(440, 1000)
```

```
1 # windows
2 from winsound import Beep
```

Pra turminha do Linux, que precisa construir a função:

```
$ sudo apt-get install beep
```

```
1 # linux
2 import os
3 def Beep(f, ms):
4     os.system("beep_-f_%i_-l_%i" % (f, ms))
```

Vamos construir agora um sensacional tocador de músicas! Você deve fazer duas classes: Som e Musica conforme os protótipos a seguir:

```
1 class Som(object):
2
3     def __init__(self, nota, duracao):
4         ...
5
6 class Musica(list):
7
8     def __init__(self, partitura, tempo):
9         ...
10
11     def play(self):
12         """ Toca a musica.
13         """
14         ...
```

Um Som deve ter:

1. O número da nota ($l\acute{a}=0$, $si=2$, ...), que será usado para calcular a frequência dela.
2. A duração da nota, que deve ser uma fração da *batida* que seja potência de $\frac{1}{2}$. Você pode até informar somente o expoente m de $(\frac{1}{2})^m$, onde $m \in [-2, 6]$.

A duração absoluta da nota dependerá da música em que o Som será tocado.

Já uma instância de Musica deve receber na construção:

1. A partitura, que é uma lista de objetos do tipo `Som`.
2. O tempo, dado em *batidas por minuto* (bpm).

`Musica` herda as propriedades de lista, e assim pode ser percorrida com um `for`, tocando nota por nota.

Tendo essa parafernália toda em mãos, construiremos esses objetos de forma que a música seja tocada através da função `Beep(f, ms)`, que recebe a frequência `f` da nota e a sua duração `ms` em *milissegundos*.

1.7 Quaterniões! ★ ★ ★

Não contente com os números complexos (\mathbb{C}) da forma $z = a + bi$, a turma da matemática trouxe pra gente um ser ainda mais esquisito: o conjunto dos *Quaternions* (\mathbb{H}).

$$\begin{aligned}
 q \in \mathbb{H} &\iff q = a + bi + cj + dk : a, b, c, d \in \mathbb{R} \\
 \mathbf{i} \times \mathbf{j} &= -(\mathbf{j} \times \mathbf{i}) = \mathbf{k} \\
 \mathbf{j} \times \mathbf{k} &= -(\mathbf{k} \times \mathbf{j}) = \mathbf{i} \\
 \mathbf{k} \times \mathbf{i} &= -(\mathbf{i} \times \mathbf{k}) = \mathbf{j} \\
 \mathbf{i} \times \mathbf{j} \times \mathbf{k} &= -1
 \end{aligned}$$

Com regras de soma convencionais, similares aos números reais e complexos, mas com uma multiplicação que mais parece o produto externo entre vetores, implemente uma classe `Quaternion`, que é criada a partir dos seus 4 coeficientes reais e implementa:

1. O operador de soma `+` (`--add--`)
2. A subtração `-` (`--sub--`)
3. A multiplicação `*` (`--mul--`)
4. O método `--repr--` que imprima na tela, pra $q = 3 + 4\mathbf{j} - 2\mathbf{k}$ a **string**
`(3.00) + (0.00)i + (4.00)j + (-2.00)k`

Extra:

1. O módulo do quaternião, através da função **abs**(q), implementada pelo método especial `--abs--`
2. A divisão / (`--truediv--`)

Dica: Implemente os métodos `--invert--` e `--neg--` para auxiliar na definição de `--truediv--` e `--sub--`, respectivamente.

2 Interface Gráfica (`tkinter`)

2.1 O Triângulo de Sierpinsky ★ ★

A figura a seguir chama-se *Triângulo de Sierpinsky*:

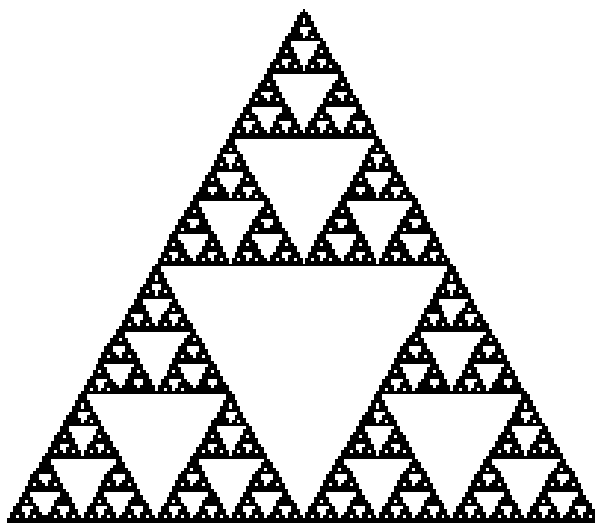


Figura 1: Triângulo de Sierpinsky

Ela é composta por triângulos equiláteros dispostos de forma que cada triângulo contém três outros em seu interior, cada um com $\frac{1}{2}$ do lado do triângulo original.

Desafio: Desenhe essa forma usando o `tkinter` ou o `turtle`. Crie uma função para compôr a figura de forma recursiva.

2.2 Pôr-do-Sol [4] ★ ★ ★

Todos os dias o Sol se põe no horizonte da mesma maneira.

Desafio: Faça uma visualização do pôr-do-Sol no `tkinter`, da forma que quiser. Se você não se sente muito inspirado, segue a receita do bolo:

1. Crie um Canvas. **Dica:** uma vez criado o Canvas, posicione ele com o método `pack`, passando as opções `expand=True` e `fill="both"` para que o Canvas ocupe toda a tela.
2. Faça primeiro o céu. Você pode criar um retângulo ou simplesmente alterar a propriedade `bg` do Canvas (cor do *background*).

```
1 import tkinter as tk
2
3 class Janela:
4     def __init__(self, root):
5         self.root = root
6
7         self.canvas = tk.Canvas(self.root)
8         self.canvas.pack(expand=True, fill="both")
9
10
11 root = tk.Tk()
12 self = Janela(root)
13 root.mainloop()
```

2.3 O Método de Monte Carlo ★ ★ ★

O Método de *Monte Carlo* funciona da seguinte forma: Se temos uma determinada região e queremos calcular sua área, basta fazer com que ela esteja contida em uma outra região cuja área é conhecida. Em seguida, sorteamos pontos aleatórios $P_i = (x_i, y_i)$ e contamos quantos pontos caem dentro da região que estamos avaliando. Assim:

$$\frac{A_{figura}}{A_{total}} \approx \frac{P_{dentro}}{P_{total}} \rightarrow A_{figura} \approx \frac{P_{dentro}}{P_{total}} \times A_{total}$$

Queremos então calcular a área de um círculo cujo raio é 1. Sabemos de antemão que valor da área é π , e vamos então usar o método acima para estimar o seu valor numérico.

Faça duas janelas usando o `tkinter`. A primeira mostra onde os pontos estão sendo posicionados, colorindo os que caem dentro de azul e os que caem

fora de **vermelho**. Na segunda tela, mostre em tempo real a aproximação para a área em azul conforme os pontos vão sendo posicionados.

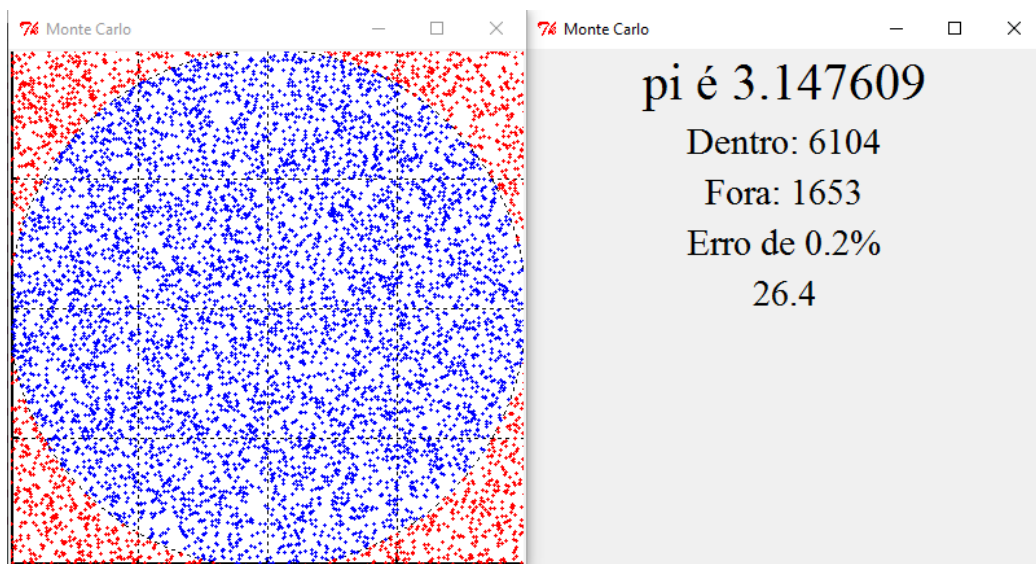


Figura 2: Cálculo de π pelo método de Monte Carlo

2.4 A proporção áurea ★ ★

0 1 1 2 3 5 8 13 21 34 55 89 ...

Muitos conhecem esse conjunto de números, a famosa *Sequência de Fibonacci*, que tem seu n -ésimo número definido por:

$$\begin{cases} F_n = F_{n-1} + F_{n-2} \\ F_0 = 0 \text{ e } F_1 = 1 \end{cases}$$

Se tomamos a razão entre dois números de *Fibonacci* consecutivos obtemos, no limite, um número que já era conhecido pelos gregos como símbolo da

beleza e da perfeição da natureza.

$$\varphi = \lim_{n \rightarrow \infty} \frac{F_{n+1}}{F_n} = \frac{1 + \sqrt{5}}{2} \approx 1.618$$

Sabido isso, sua tarefa é gerar a figura abaixo, usando o módulo que preferir, mas de forma recursiva. Para que os gregos realmente fiquem contentes com a majestade da sua figura é preciso que ela esteja conforme a proporção dada por φ .

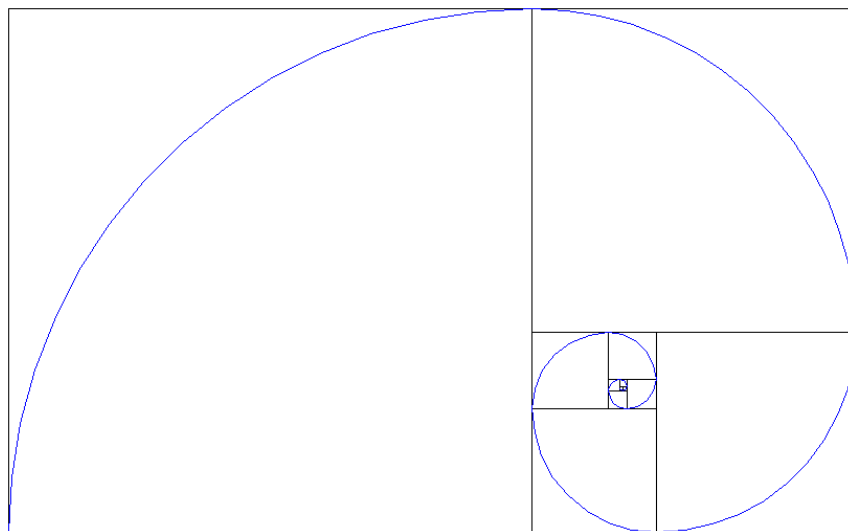


Figura 3: A proporção áurea

2.5 Música IV - O Piano ★ ★ ★

Desafio: Use os resultados das etapas anteriores(**Musical** e construa um Piano Virtual com o `tkinter`. Desenhe as teclas em um `Canvas` e use o método `Canvas.bind` para associar letras como `a, s, d, f, g, h, j` e `k` às teclas brancas e `w, e, t, y` e `u` às pretas, por exemplo.

- A mantissa é composta pela soma de potências de $\frac{1}{2}$, portanto números que não são resultado de somas finitas dessas potências vão apresentar erros de representação.
-

3.2 Computação Quântica ★ ★ ★ ★ ★

4 Plotagem de Gráficos (Matplotlib)

Interlúdio: `nan` e `inf`

Um outro tópico interessante de se falar, ainda no escopo da **aritmética de ponto-flutuante** (`float`), é a presença de três números especiais no padrão **IEEE 754**: `nan`, `inf` e `-inf`.

Como o nome já indica, `inf` e `-inf` são usados pra representar o infinito positivo ∞ e o negativo $-\infty$, respectivamente. É comum que surjam durante operações de divisão por zero, ou alguma outra operação que exceda a precisão do expoente.

O `nan` (***n**ot **a** **n**umber*), por sua vez, é fruto de operações indeterminadas, como $\infty - \infty$ e $\frac{0}{0}$. É particularmente útil para representar valores ausentes. Quando presente em um `array` que será usado para plotar um gráfico, o `Matplotlib` simplesmente ignora as entradas, criando lacunas na curva desenhada.

É possível obter esses números usando a função `float`, passando como parâmetro o nome do número desejado:

```
1 >>> x = float("inf")
2 >>> y = float("nan")
3 >>> z = float("-inf")
```

Listing 2: 'nan e inf'

5 Conexão e Redes (**socket**)

Referências

- [1] Prof. Pedro Asad (2016)
- [2] Prof. Brunno Goldstein (2016)
- [3] Prof. Claudio Esperança (2018)
- [4] Prof. Jonas Knopman (2018)