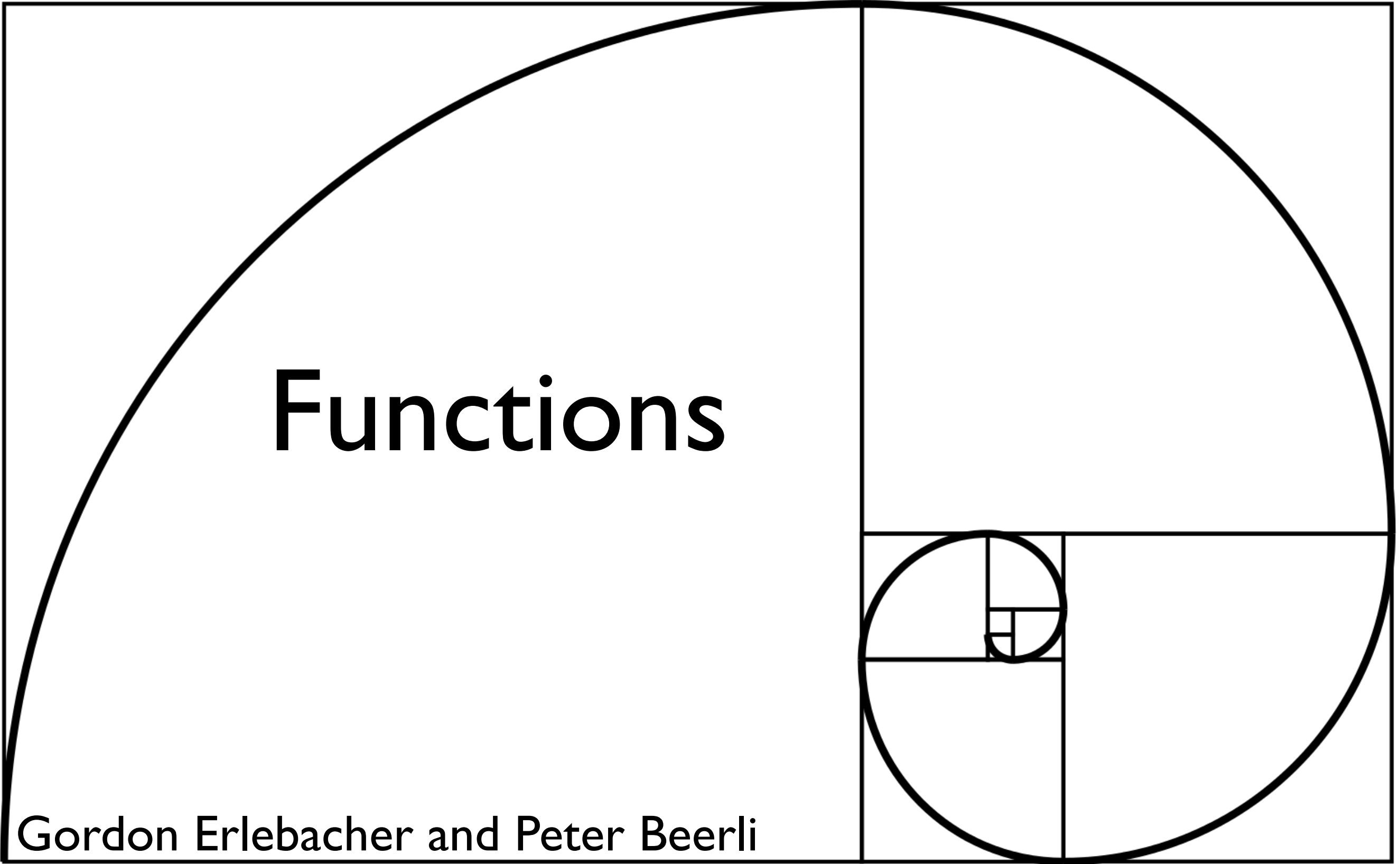


Functions

Gordon Erlebacher and Peter Beerli



Function Tutorials

- Intro to functions
 - <http://anh.cs.luc.edu/python/hands-on/3.1/handsonHtml/functions.html>
- More on functions
 - http://www.tutorialspoint.com/python/python_functions.htm

What is a Function

- In math: a function is a mapping:
 - input ==> unique output
- In Python:
 - input ==> unique (or no) output
 - However, something must happen
 - change a global variable
 - output to a file, a pipe, to the web

Kinds of Functions

- Simplest regular function
 - `def funct(args):`
`pass`
- Simplest nontrivial Lambda function
 - `a = lambda x: x+2`
`a(5)    # ==> 7`

Python supports the creation of anonymous functions (i.e. functions that are not bound to a name) at runtime, using a construct called **lambda**.

Function Arguments

- `def add(x,y):`
 `return x+y`
- `def add(a,b,*lst):`
 `return a + b + sum(lst)`
`add(5,3,7,8,9)   # ==> 32`

Keyword arguments

- `def return_args(a,b,c):`
 `return a,b,c` `# also return (a,b,c)`
- `return_args(b=27,a=2,c=10)` `# ==> (2, 27, 10)`
- `return_args(2,c=5,b=3)` `# ==> (2, 3, 5)`
- `return_args(2,c=5,a=3)` `# error`
 - `b` not defined
- `return_args(b=3,a=2,3)` `# SyntaxError: non-keyword arg after keyword arg`
 - non-assigned args must come first

Default Arguments

- `def add(x,y=3):`
 `return x+y`
- `add(5)` `# returns 8`
- `add(5,8)` `# returns 13`
- if less arguments are specified than the number of arguments, the missing arguments take default values
- missing arguments can take default values *if all* given arguments use keywords
- cannot have default arguments and at the same time have arbitrary number of arguments! Why?
- useful for very long argument lists

Variable Number of Arguments

- A function as 2 arguments, but 4 arguments are passed
- `def add(x,y,*kw):` # add all arguments
 `return x+y+sum(kw)`
- `add(3,4,5,6)` # returns 18
- `add(3,4,5,6,7)` # returns 25
- Cannot use keyword arguments if there are more than 2 arguments since `kw` is a list of non-keyword arguments, and keywords come after non-keyword arguments

```
>>> def add(x,y,*kw):  
...     return x+y+sum(kw)  
...  
>>> add(x=3,y=2,1,2,3)  
File "<stdin>", line 1  
SyntaxError: non-keyword arg after keyword arg  
>>> add(3,2,1,2,3)  
11  
>>> add(3,2)  
5  
>>> add(x=3,y=2)  
5
```


Dictionary argument

- `def add(a,b,**kww):`
 `try:`
 `print "N:", kww['N']`
 `except:`
 `print "no keyword N"`
 `return a+b`
- `kww` returns all excess key-value pairs
- `add(3,5,N=35,h=2)`
 - returns ==> N: 35
- `add(3,5,NN=35,h=2)`
 - returns ==> 'no keyword N'
 - type of NN is a string

Functions are references

- ```
def add(a,b):
 return a+b
def sub(a,b):
 return a-b
```
- ```
v = add  
v(3,4)    # ==> 7  
v = sub  
v(3,4)    # ==> -1
```
- ```
v = [sub, add]
for i in v:
 print i(5,9) # returns -4 and 14
```

# Functions are references

- `def calc(fct, a, b):  
 return fct(a,b)`
- `v = [add, sub]  
for i in v:  
 calc(i,5,8)`
- returns 13 and -3

# Lambda expressions

- useful to define functions directly in arguments
- elegant, but not necessary
- `lambda x,y : body of function`
- `fct = lambda x,y : x+y`  
`fct(3,4)     # returns 7`
- `fct` is used like any other function

# Recursive Functions

```
In [1]: def factorial(n):
 # assume n > 0
 if (n <= 1):
 return 1
 else:
 return n*factorial(n-1)
```

# Use of Recursive Functions

- Any problem where a subset has a similar structure to the whole
  - fractals
  - trees, binary trees, quad-trees, oct-trees
  - solution to recursions
    - $a[n+1] = a[n] + a[n-1]$   
 $a[0] = 0, a[1] = 1$
- Harder to program without recursion, although possible
- Use recursion if easier. If too slow, remove recursion at a later stage of development.

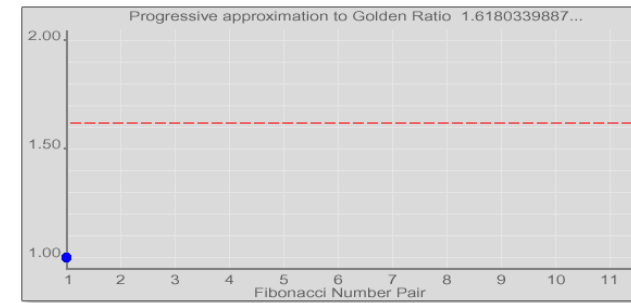
# Fibonacci sequence

```
"""
 Solve $a[n+1] = a[n] + a[n-1]$
 $a[0] = 0$
 $a[1] = 1$
"""

def recursion(n):
 if n == 0: return 0
 elif n == 1: return 1
 else:
 return recursion(n-1) + recursion(n-2)

#-----

if __name__ == "__main__":
 for i in xrange(100):
 print recursion(i),
```



Pair 1:  $\frac{1}{1} = 1.000$

0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946 17711 28657 46368 75025  
832040 1346269 2178309 3524578 5702887 9227465 14930352 24157817 39088169 63245986



# Function methods

```
a = lambda x: 1
print dir(a)
```

```
['__call__', '__class__', '__closure__', '__code__', '__defaults__', '__delattr__', '__dict__', '__doc__', '__format__', '__get__', '__getattr__', '__globals__', '__hash__', '__init__', '__module__', '__name__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'func_closure', 'func_code', 'func_defaults', 'func_dict', 'func_doc', 'func_globals', 'func_name']
```

```
print (dir(a.func_code))
```

```
['__class__', '__cmp__', '__delattr__', '__doc__', '__eq__', '__format__', '__ge__', '__getattr__', '__gt__', '__hash__', '__init__', '__le__', '__lt__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'co_argcount', 'co_cellvars', 'co_code', 'co_consts', 'co_filename', 'co_firstlineno', 'co_flags', 'co_freevars', 'co_lnotab', 'co_name', 'co_names', 'co_nlocals', 'co_stacksize', 'co_varnames']
```



# Context

- All variables inside a function are local
- Global variables can be accessed **but not changed except** via the **global** declaration  
(the web: “If you want to simply access a global variable you just use its name. However to **change** its value you need to use the **global** keyword.”)
- A global variable can be defined from within a function!

```
def example():
```

```
 global v
```

```
 v = 5
```

```
print v # returns 5. Exception without global
```

```
total = 9
globvar = 11
```

```
def add(a, b):
 total = 3 # does not affect global variable
 return a+b+total+globvar
```

```
def add_glob(a, b):
 global total
 total = 3 # changes the global variable total
 return a+b+total
```

```
print add(3,5) # returns 22 (3+5+3+11)
print total # returns 9
print add_glob(3,5) # returns 17
print total # returns 3
```

# Variable Types

- `locals()` : local variables
- `globals()` : global variables
- `vars(obj)` : object dictionary
- `vars()` : same as `locals()`

# Output

```
total, newvar = 9, 11
```

```
def add(a, b):
 total = 3
 print "locals= ", locals()
 print "globals= ", globals()
 return a+b+total+newvar
```

```
def add_glob(a, b):
 global total
 total = 3 # changes the global total
 print "locals= ", locals()
 print "globals= ", globals()
 return a+b+total
```

```
add(3,4)
print "-----"
add_glob(3,4)
```

```
locals= {'a': 3, 'total': 3, 'b': 4}
globals= {'add_glob': <function
add_glob at 0x379830>, '__builtins__':
<module '__builtin__' (built-in)>,
 '__file__': 'context.py', '__package__':
None, 'add': <function add at
0x379870>, '__name__': '__main__',
'total': 9, 'newvar': 11, '__doc__': None}
```

```

locals= {'a': 3, 'b': 4}
globals= {'add_glob': <function
add_glob at 0x379830>, '__builtins__':
<module '__builtin__' (built-in)>,
 '__file__': 'context.py', '__package__':
None, 'add': <function add at
0x379870>, '__name__': '__main__',
'total': 3, 'newvar': 11, '__doc__': None}
```

# vars()

```
a = lambda x: x + 3
a.func_dict['location'] = 'scs_class'
print "\nvars= ", vars(), '\n'
print "vars(a)= ", vars(a)
```

## Output

python vars\_demo.py

```
vars= {'a': <function <lambda> at 0x1004d5848>, '__builtins__': <module
'__builtin__' (builtin)>, '__file__': 'temp.py', '__package__': None, '__name__':
'__main__', '__doc__': None}
```

```
vars(a)= {'location': 'scs_class'}
```

# Returning a function

- Very straightforward
- Function is a reference variable
- Given a function, one can find out how many arguments it has and other properties

```
def add(x,y):
 return x + y
```

```
def add_more(fct, n):
 return lambda x,y: fct(x,y)+n
```

**Return function**

```
add_10 = add_more(add, 10)
print add_10(1,2) # ==> 13 = 1+2+10
```

```
add_7 = add_more(add, -7)
print add_7(1,2) # ==> -4 = 1+2-7
```

```
def add_list(fct, lst):
 def add_list(lst):
 return fct(lst[0], lst[1])
 return add_list
```

**Return function**

```
add_list = add_list(add, [3,5])
print add_list([5,2]) # ==> 7
```

13

-4

7



# Function Properties

```
def add(*kw, **kww):
 def newfunc(a, *lst):
 z = 3
 return a + sum(kw) + sum(lst)
 return newfunc

newfunc = add(3,5,a=10)
print newfunc(10,20,30) # 68 = 3+5+10+20+30
print dir(newfunc)
print "name: ", newfunc.func_name
print "globals: ", newfunc.func_globals
print "freevars: ", newfunc.func_code.co_freevars
print "arg count: ", newfunc.func_code.co_argcount # ==> 1
print " etc ..."
```

68

```
['__call__', '__class__', '__closure__', '__code__', '__defaults__', '__delattr__',
['__dict__', '__doc__', '__format__', '__get__', '__getattribute__', '__globals__',
['__hash__', '__init__', '__module__', '__name__', '__new__', '__reduce__',
['__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
'func_closure', 'func_code', 'func_defaults', 'func_dict', 'func_doc', 'func_globals',
'func_name']
name: newfunc
globals: {'__builtins__': <module '__builtin__' (built-in)>, '__file__': 'temp.py',
'__package__': None, 'add': <function add at 0x1006d5848>, '__name__': '__main__',
'__doc__': None, 'newfunc': <function newfunc at 0x1006d55f0>}
freevars: ('kw',)
arg count: 1
etc ...
```



# Example of callback function: First order Differential Equation

```
""" Solve an ODE via a first order Euler method.
 Compute the error as a function of time as:
 We solve $dy/dt = rhs(y,t)$ for x in $[a,b]$
 rhs(y,t): right hand side
 """

from pprint import pprint
#-----
def ode(interv, sol0, max_iter, rhs, dt):
 y = [sol0]
 a = interv[0]
 b = interv[1]
 t = 0.
 iter = 0

 while t < b and iter < max_iter:
 ynew = y[-1] + dt * rhs(y[-1],t)
 y.append(ynew)
 iter += 1
 t += dt
 return y
```

```

def rhs1(y,t):
 return(-y)

def rhs2(y,t):
 return(-3.*y)

def rhs3(y,t):
 return(-15.*y)
#-----
if __name__ == "__main__":
 interv, sol0, max_iter, rhs , dt = \
 ([1,5], 1., 1000, rhs3, .2)

 tests = [rhs1, rhs2, rhs3]
 for rhs in tests:
 sol = ode(interv, sol0, max_iter, rhs, dt)
 ti = [dt*i for i in range(0,len(sol))]
 sol = [[t,s] for t,s in zip(ti,sol)]
 pprint(sol)
 print "=====
```

```

[[0.0, 1.0],
 [0.2, 0.8],
 [0.4, 0.64],
 [0.6000000000000001, 0.512],
 [0.8, 0.4096],
 [1.0, 0.32768],
 [1.2000000000000002, 0.26214400000000004],
 [1.4000000000000001, 0.20971520000000005],
 [1.6, 0.16777216000000003],
 [1.8, 0.13421772800000004],
 [2.0, 0.10737418240000003],
 [2.2, 0.08589934592000002],
 [2.4000000000000004, 0.06871947673600001],
 [2.6, 0.05497558138880001],
 [2.8000000000000003, 0.04398046511104001],
 [3.0, 0.035184372088832],
 [3.2, 0.028147497671065603],
 [3.4000000000000004, 0.02251799813685248],
 [3.6, 0.018014398509481985],
 [3.8000000000000003, 0.014411518807585589],
 [4.0, 0.01152921504606847],
 [4.2, 0.009223372036854777],
 [4.4, 0.007378697629483821],
 [4.6000000000000005, 0.005902958103587057],
 [4.8000000000000001, 0.004722366482869646],
 [5.0, 0.0037778931862957168]]
```