Kernel: SageMath 10.7

The Prime Number Theorem

MATH 242 Modern Computational Math

Today we will continue our study of the *prime counting function* $\pi(x)$, which is defined to be the number of primes less than or equal to x.

Remember our big question from last time: ***What is the shape of the prime counting function?*** In other words, can we approximate the prime counting function with simpler functions?

Here is our sieve of Eratosthenes function from a previous class session:

In [4]:
```
# function: sieveEratos
# input: integer nMax, which must be at least 2
# output: a list of primes up to nMax
def sieveEratos(nMax):
    # create a list of integers up to nMax
    # if you start the list from zero, then each number in the list is EQUAL to its index in the
list, which is very convenient
    nums = list(range(nMax+1))

    # replace 1 in the list with 0
    nums[1] = 0

    # compute sqrt(nMax)
    nroot = floor(sqrt(nMax))

    # loop over each list item less than or equal to nroot
    i = 0
    while nums[i] <= nroot:
        # if nums[i] is not zero, then i is prime
        if nums[i] != 0:
            # i is prime, so set all of its multiples to zero
            j = i^2
            while j <= nMax:
                nums[j] = 0
                j += i
        i += 1

    # now extract all nonzero numbers from the list
    return [n for n in nums if n != 0]

# testing
print(sieveEratos(100))
```

Out[4]:
```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]
```

Here is our function from Friday that returns a list of values of the prime counting function $\pi(x)$.

In [5]:
```
# function: computePiVals
# input: integer nMax, which must be at least 2
# return: a list of values of the prime-counting function π(x), for integers x from 1 to nMax
def computePiVals(nMax):
    # compute a list of primes up to nMax
    primes = sieveEratos(nMax)

    # make a list of nMax+1 zeros
    piVals = [0]*(nMax+1)
```

```
    # track how many primes we've found so far
    count = 0

    # loop over integers i from 2 to nMax
    for i in range(2, nMax + 1):
        # if i is the next prime, then add 1 to our count
        if count < len(primes) and i == primes[count]:
            count += 1
        #print(f"i = {i}, count = {count}")

        # store the current count in piVals[i]
        piVals[i] = count


    # return the list of piVals
    return piVals

# testing
print(computePiVals(10))
```

Out[5]:
```
[0, 0, 1, 2, 2, 3, 3, 4, 4, 4, 4]
```

Let's make a big list of $\pi(x)$ values for integers x from 0 up to some large M.

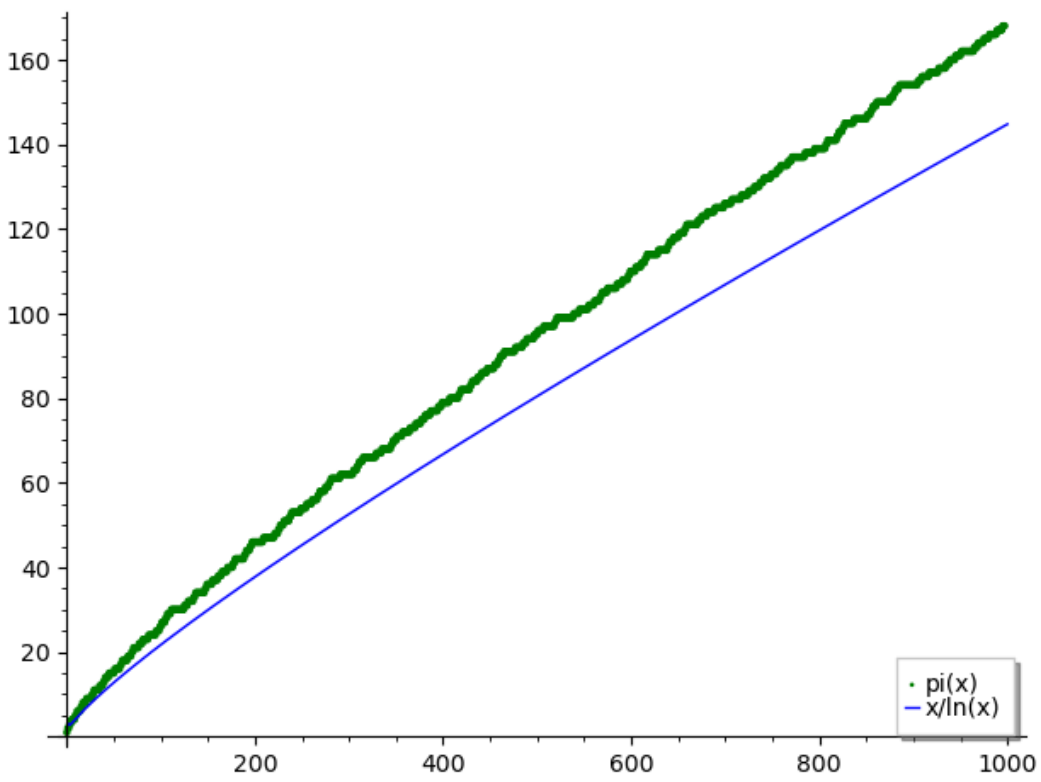
In [6]:
```
nMax = 200000
piVals = computePiVals(nMax)
piVals[:20]
```

Out[6]:
```
[0, 0, 1, 2, 2, 3, 3, 4, 4, 4, 4, 5, 5, 6, 6, 6, 6, 7, 7, 8]
```

In the homework for today, we considered $\frac{x}{\ln(x)}$ as a possible approximation of $\pi(x)$. Make a plot comparing $\frac{x}{\ln(x)}$ and $\pi(x)$.

In [11]:
```
xMin = 2
xMax = 1000
plot1 = list_plot( piVals[xMin:xMax], color="green", legend_label="pi(x)" )
plot2 = plot( x/log(x), (x, xMin, xMax), legend_label="x/ln(x)" )
combined = plot1 + plot2
combined.show(legend_loc="lower right")
```

Out[11]:



Density of Primes

Roughly speaking, the *density* of primes near x is the proportion of integers near x that are prime. We can approximate the density of primes near x by fixing some length ℓ and computing the proportion of integers in the interval $(x, x+\ell)$ that are prime. In other words, the density of primes near x is approximately

$$\frac{\pi(x+\ell) - \pi(x)}{\ell}$$

Complete the following function that approximates the density of primes near x:

In [12]:
```
# function: primeDensity
# input: positive integers x and length
# return: estimate of the density of primes near x, computed using an interval of the specified
length
def primeDensity(x, length):
    return (piVals[x + length] - piVals[x])/length

primeDensity(10, 10)
```

Out[12]: 2/5

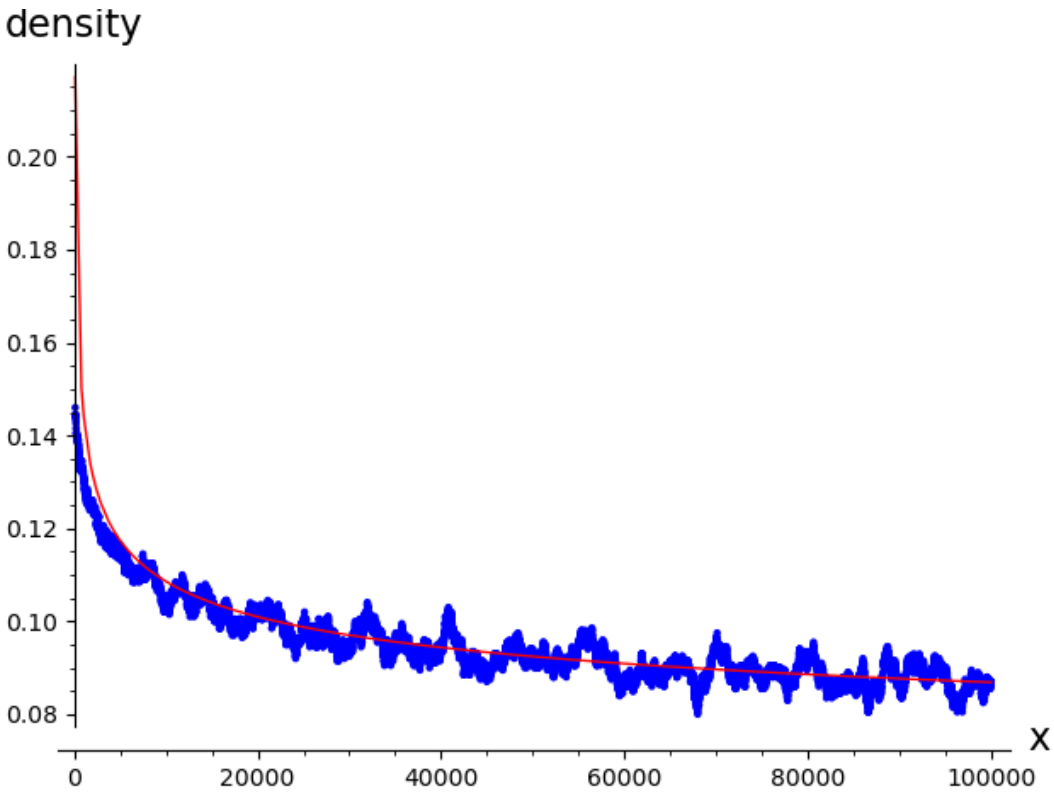
Make some plots of your prime density function for different values of `x` and `length`.

In [20]:
```
xVals = range(100, 100001, 10)
length = 2000
densityVals = [primeDensity(x, length) for x in xVals]

#print(densityVals)

plot1 = list_plot( list(zip(xVals, densityVals)) , axes_labels=["x","density"])
plot2 = plot( 1/log(x), (x, 100, 100000), color="red")
combined = plot1 + plot2
combined.show()
```

Out[20]:



In [0]:

In the late eighteenth century, using printed tables of prime numbers, Gauss conjectured that the density of primes near x is approximately $\frac{1}{\ln(x)}$. Can you provide computational evidence for or against Gauss's conjecture?

In [0]:

In [0]:

The Logarithmic Integral

If the density of primes near x is approximately $\frac{1}{\ln(x)}$, then the *count* of primes up to x should be approximately the integral $\int_0^x \frac{1}{\ln(t)} dt$. This integral has a special name and notation:

Definition: The *logarithmic integral*, $\mathrm{li}(x)$ is defined

$$\mathrm{li}(x) = \int_0^x \frac{1}{\ln(t)} dt.$$

This integral is a bit tricky to compute, but fortunately it is already implemented in Sage as `li()` and also as `log_integral()`.

In [21]:
```
li?
```

Out[21]:
```
Signature:      li(self, coerce=True, hold=False, dont_call_method_on_arg=False, *args)
Type:           Function_log_integral
String form:    log_integral
File:           /ext/sage/10.7/src/sage/functions/exp_integral.py
Docstring:
   The logarithmic integral \operatorname{li}(z) defined by

      \operatorname{li}(x) = \int_0^z \frac{dt}{\ln(t)} =
```

```
   \operatorname{Ei}(\ln(x))

for x > 1 and by analytic continuation for complex arguments z (see
[AS1964] 5.1.3).

EXAMPLES:

Numerical evaluation for real and complex arguments is handled
using mpmath:

   sage: N(log_integral(3))
   2.16358859466719
   sage: N(log_integral(3), digits=30)
   2.16358594667191976287692236735
   sage: log_integral(ComplexField(100)(3+I))
   2.2879892769816826157078450911 + 0.87232935488528370139883806779*I
   sage: log_integral(0)
   0

Symbolic derivatives and integrals are handled by Sage and Maxima:

   sage: x = var('x')
   sage: f = log_integral(x)
   sage: f.diff(x)
   1/log(x)
   sage: f.integrate(x)
   x*log_integral(x) - Ei(2*log(x))

Here is a test from the mpmath documentation. There are
1,925,320,391,606,803,968,923 many prime numbers less than 1e23.
The value of "log_integral(1e23)" is very close to this:

   sage: log_integral(1e23)
   1.92532039161405e21

ALGORITHM:

Numerical evaluation is handled using mpmath, but symbolics are
handled by Sage and Maxima.

REFERENCES:

* https://en.wikipedia.org/wiki/Logarithmic_integral_function

* mpmath documentation: logarithmic-integral
Init docstring:
See the docstring for "Function_log_integral".

EXAMPLES:

   sage: log_integral(3)
   log_integral(3)
   sage: log_integral(x)._sympy_()
   li(x)
   sage: log_integral(x)._fricas_init_()
   'li(x)'
Call docstring:
Evaluate this function on the given arguments and return the
result.

EXAMPLES:

   sage: exp(5)
   e^5
   sage: gamma(15)
   87178291200
```

```
Python float, Python complex, mpmath mpf and mpc as well as numpy
inputs are sent to the relevant "math", "cmath", "mpmath" or
"numpy" function:

   sage: cos(1.r)
   0.5403023058681398
   sage: assert type(_) is float
   sage: gamma(4.r)
   6.0
   sage: assert type(_) is float

   sage: cos(1jr)  # abstol 1e-15
   (1.5430806348152437-0j)
   sage: assert type(_) is complex

   sage: import mpmath
   sage: cos(mpmath.mpf('1.321412'))
   mpf('0.24680737898640387')
   sage: cos(mpmath.mpc(1,1))
   mpc(real='0.83373002513114902', imag='-0.98889770576286506')

   sage: import numpy
   sage: if int(numpy.version.short_version[0]) > 1:
   ....:     __ = numpy.set_printoptions(legacy="1.25")                    # needs numpy

   sage: sin(numpy.int32(0))
   0.0
   sage: type(_)
   <class 'numpy.float64'>
```

Compute some values of $\mathrm{li}(x)$. How do they compare to $\pi(x)$ and $\frac{x}{\ln(x)}$?

In [23]:
```
x = 1000
print( piVals[x] )
print( n(x/log(x)) )
print( n(li(x)) )
```

Out[23]:
```
168
144.764827301084
177.609657990152
```

In [24]:
```
x = 100000
print( piVals[x] )
print( n(x/log(x)) )
print( n(li(x)) )
```

Out[24]:
```
9592
8685.88963806503
9629.80900105080
```

In [30]:
```
x = 200000
print( piVals[x] )
print( n(x/log(x)) )
print( n(li(x)) )
```

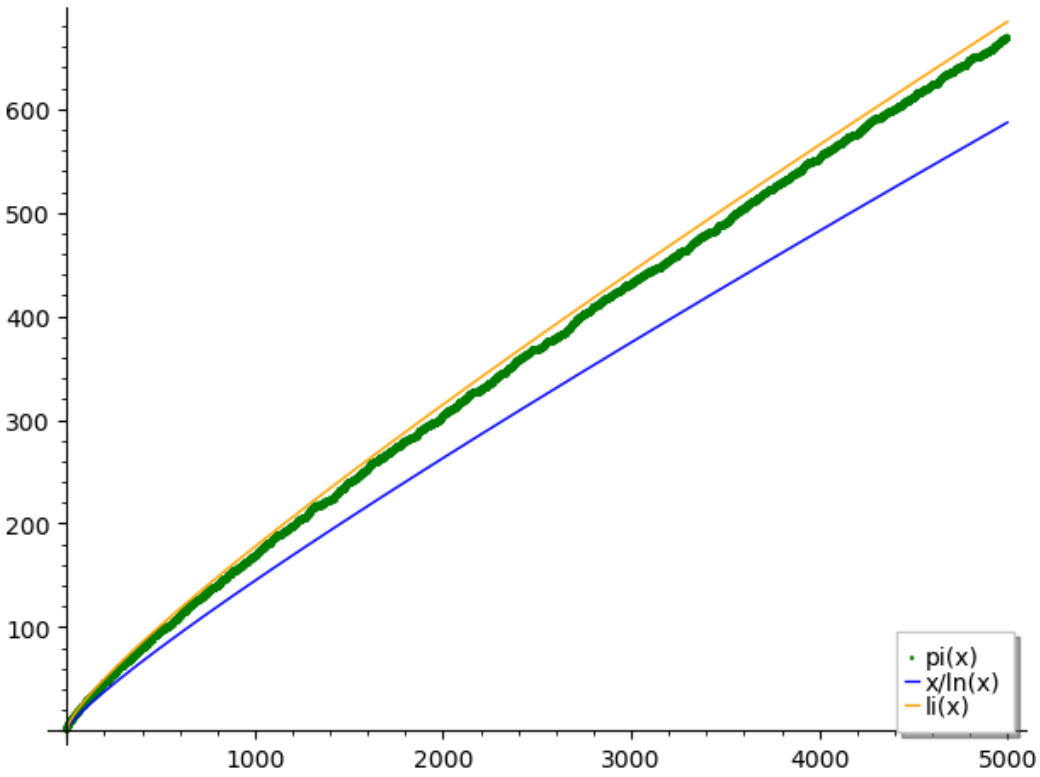
Out[30]:
```
17984
16385.2867181844
18036.0521218970
```

Make a plot showing the values of $\pi(x)$, $\frac{x}{\ln(x)}$, and $\mathrm{li}(x)$. What do you observe?

In [32]:
```
reset("x")
xMin = 2
xMax = 5000
```

```
plot1 = list_plot( piVals[xMin:xMax], color="green", legend_label="pi(x)" )
plot2 = plot( x/log(x), (x, xMin, xMax), legend_label="x/ln(x)" )
plot3 = plot( li(x), (x, xMin, xMax), legend_label="li(x)", color="orange")
combined = plot1 + plot2 + plot3
combined.show(legend_loc="lower right")
```

Out[32]:



In [0]:

The Riemann Zeta Function

To find an even better approximation to the prime counting function, we must learn about a function called the *Riemann zeta function*. This function leads to one of the most important open questions in mathematics, the *Riemann Hypothsis*, which is a statement about the zeros of the Riemann zeta function.

The Riemann zeta function, $\zeta(s)$, is defined

$$\zeta(s) = \sum_{n=1}^{\infty} \frac{1}{n^s} = \frac{1}{1^s} + \frac{1}{2^s} + \frac{1}{3^s} + \frac{1}{4^s} + \frac{1}{5^s} + \cdots$$

The Riemann zeta function converges for all $s > 1$. Moreover, it converges for all *complex* numbers s with real part greater than 1. However, for $s = 1$ the sum diverges, so $\zeta(1)$ is undefined.

Write a Python function below that approximates $\zeta(s)$ by computing a partial sum of `numterms` terms.

We didn't get to this in class, but the following code cell contains an implementation of the zeta function.

In [33]:

```
# function: zeta
# input: positive value s, positive integer numTerms
# return: an approximation of ζ(s)
def zeta(s, numTerms):
    total = 0
    for m in range(1, numTerms):
        total += 1/(m^s)
    return total
```

Use your `zeta` function to compute decimal approximations of $\zeta(2), \zeta(3), \zeta(4), \ldots$. What do you notice? Can you find closed-form expressions for any of these values?

In [38]:
```
print( zeta(2, 10000).n() )
print( n(pi^2/6) )
```

Out[38]:
```
1.64483406184806
1.64493406684823
```

In [0]:

In [0]:

One connection between the Riemann zeta function and the prime numbers has to do with the following product:

$$\prod_{p \text{ prime}} \frac{1}{1 - p^{-s}}$$

This is a product over all prime numbers p.

Write a Python function below that computes partial products of this infinite product. Take the primes in order, starting with the smallest prime.

In [0]:
```
# function: primeProduct
# input: positive value s, positive integer numFactors
# return: the product of 1/(1-p^(-s)), where p takes the values of the first numFactors prime numbers
def primeProduct(s, numFactors):
```

In [0]:

In [0]:

What is the connection between $\zeta(s)$ and $\prod_{p \text{ prime}} \frac{1}{1-p^{-s}}$?