Some Sage

You can obtain the computer algebra system Sage and its documentation from www.sagemath.org. An alternative to running Sage on prdile is to use SageMathCloud.

Zeta

Create the real field and complex field with 200 bits of precision:

```
RR = RealField(200)
CC = ComplexField(200)
```

The imaginary number I is predefined:

```
sage: I^2
-1
```

The Riemann zeta function:

```
zeta(2)       # a symbolic expression
zeta(2.0)     # approximation to standard precision
zeta(RR(2))   # approximation to precision of RR
zeta(2+3.0*I) # approximation to standard precision
zeta(CC(2,3)) # approximation to precision of CC
```

If the parameter real the computation is done using the MPFR library. When the input is not real, the computation is done using the PARI C library.

The optional Sage package database_odlyzko_zeta provides the imaginary parts of the first 2,001,052 non-trivial zeros of the Riemann zeta function, accurate to within $4 \cdot 10^{-9}$. Note to summer school participants: The package is already installed if you are using Sage on prdile.

```
zz = zeta_zeros()
zz[0]
zz[0:10]
```

Find the differences between the first 10 consecutive zeros.

```
zz = zeta_zeros()
[zz[i]-zz[i-1] for i in range(1,10)]
```

Alternating Zeta

This function evaluates the k-the derivative of the alternating zeta function using convergence acceleration:

```
def altzeta(k,s,N=0):
  if N==0: N=40+2*ceil(abs(imag(s)))+2*ceil(k)
  s=CC(s)
  one = RR(1)
  RN=RR(N)
  half = one/2

  d=exp(log(3+sqrt(RR(8)))*RN); d=(d+1/d)/2
  b=-one; c=-d; t=RR(0);
  for n in range(0,N-1):
    Rn = RR(n)
    Rn1=Rn+one
    Rn2=Rn1+one
    c = b-c; logn = log(Rn2); loglogn=log(logn)
    t=t+c*exp(loglogn*k)/exp(logn*s)
    b = (Rn+RN)*(Rn-RN)*b/((Rn+half)*(Rn1))
  if k==0:
   return one+(t/d)*exp(log(CC(-1))*(k+1))
  else:
    return (t/d)*exp(log(CC(-1))*(k+1))
```

Plotting

First we produce the background of the summer school poster with complex_plot.

```
complex_plot(zeta, (-9, 9), (-2, 16))
```

We plot the real part (in blue) and the imaginary part (in red) of ζ(σ+it) for a given t in the critical strip.

```
zz = zeta_zeros()

def horizontal_cut(t):
  G = plot(lambda s:zeta(s+I*t).imag(),(0,1),rgbcolor=(1.0,0,0))
  G+= plot(lambda s:zeta(s+I*t).real(),(0,1),rgbcolor=(0,0,1.0))
  return G
```

```
horizontal_cut(zz[0])
```

Root Finding

A function that using Newton's method for finding zeros of an analytic function can be easily implemented

```
def newton(start,f,df):
  z=CC(start)
  fz = f(z)
  N=0
  while abs(fz)>10**-6 and N<1000:
    N+=1
    dfz = df(z)
    z=z-fz/dfz
    fz = f(z)
  if N>1000:
    return false, z
  else:
    return true, z
```

We now use the function to find a zero of the alternating zeta function.

```
f = lambda z:altzeta(0,z)
df = lambda z:altzeta(1,z)

newton(1/2+13*I,f,df)
```

Numerical Integration

Using numerical integration we write a function that counts the zeros of an analytic function in a box.

```
RR = RealField(200)
def count_zeros(s,k,f,r=0.1):
# let f(k,s) be a function that returns the k-th derivative of an analytic function f at s
# return the number of zeros of the k-the derivative of f in a box around s whose sides
# have length 2r
  s1=s.real()-r
  s2=s.real()+r
  t1=s.imag()-r
  t2=s.imag()+r
  # top horizontal
  I1im=integral_numerical(lambda x:(f(k+1,CC(t2*I+x))/f(k,CC(t2*I+x))).imag(), s2 , s1)
  # vertical left
  I2re=integral_numerical(lambda x:(f(k+1,CC(x*I+s1))/f(k,CC(x*I+s1))).real(), t2, t1)
  # vertical right
  I3re=integral_numerical(lambda x:(f(k+1,CC(x*I+s2))/f(k,CC(x*I+s2))).real(), t1, t2)
  # bottom horizontal
  I4im=integral_numerical(lambda x:(f(k+1,CC(t1*I+x))/f(k,CC(t1*I+x))).imag(), s1 , s2)
  Icont = I1im[0]+I2re[0]+I3re[0]+I4im[0]
  return Icont/(2*RR(pi))
```

We count the zeros of the alternating zeta function in the box with side lengths 1/2 with center 1/2+14i.

```
count_zeros(CC(1/2+14*I),0,altzeta,1/4)
```

Primes

We find the maximum gap between any two consecutive primes up to 10^6.

```
def max_prime_gap(N):
  maxgap = 0
  p1 = 2; p2 = 3
  while p2 <= N:
    if p2-p1 > maxgap:
      maxgap = p2-p1
      print p2,'-',p1,'=',maxgap
    p1 = p2
    p2 = next_prime(p2)
  return maxgap

max_prime_gap(10^6)
```

mpmath

The Python library mpmath provides an extensive set of transcendental functions, unlimited exponent sizes, complex numbers, interval arithmetic, numerical integration and differentiation, root-finding, linear algebra, and much more. In particular it provides fast evaluation of Hurwitz zeta functions and their derivatives, which is used for evaluation zeta and Dirichlet L-functions. mpmath is distributed with Sage but must be loaded explicitely.

```
import mpmath

mpmath.mp.dps=100     # set mpmath precision

mpmath.hurwitz(3,1,2) # the second derivative of the riemann zeta function
                      # (hurwitz zeta with a=1) at 3

                      # find a root of the second derivative of zeta near 1+400i.
mpmath.findroot(lambda z:mpmath.hurwitz(10,1,2),1+400*I)
```

Fredrik's example 1

```
from mpmath import *
mp.dps = 50; mp.pretty = True

mp.dps = 500
quad(lambda t: t**(1.5-1) * exp(-t), [0,inf], verbose=True) / sqrt(pi)

mp.dps = 30
nsum(lambda n: 1/n**4, [1,inf], verbose=True)
nsum(lambda n: 3**n, [1,inf], verbose=True)

fp.plot(fp.siegelz, [1e9, 1e9+10], file="a.png")
zetazero(10000000)
```

Fredrik's example 2

Download ore_algebra-0.2.spkg
from www.risc.jku.at/research/combinat/software/ore_algebra/
Install with "sage -i ore_algebra-0.2.spkg"

```
from ore_algebra import *
R.<x> = QQ['x']
A.<Dx> = OreAlgebra(R, 'Dx')

# annihilator of exp(-x^2)
print (Dx - 1).annihilator_of_composition(-x^2)

# annihilator of integral
L = (Dx + 2*x).annihilator_of_integral()
```

```
print L

# annihilator of power series
print L.to_S(OreAlgebra(QQ['n'], 'Sn'))

# first few terms
print L.power_series_solutions(10)

# asymptotic series
print L.annihilator_of_composition(1/x).generalized_series_solutions(10)

# Stirling's series for the gamma function
B.<Sx> = OreAlgebra(R, 'Sx')
print (Sx - x).generalized_series_solutions()
```

Fredrik's example 3

```
# must use either RealField or mp.mpf (I use mp.mpf) since
# the numbers are too large for floats
from mpmath import mp
mp.dps = 15
ns = []
gammas = []
gammas_approx = []
gammas_approx_magnitude = []

# download this file from http://fredrikj.net/math/hurwitz_zeta.html
infile = open("stieltjes100k20d.txt", "r")
for n, line in enumerate(infile):
   gammas.append(mp.mpf(line.strip()))

infile.close()

def appr(n):
   if n < 1:
      return (mp.nan, mp.nan, mp.nan)
   if n % 10000 == 0:
      print n
   # this part done using fp (hardware) arithmetic for speed
   from mpmath import fp
   n = float(n)
   log2pi = fp.log(2r * fp.pi)
   v = fp.findroot(lambda v: log2pi + v*fp.tan(v) - fp.log(n*fp.cos(v)/v),
      [1e-6, fp.pi/2r-1e-6], solver='bisect', maxsteps=100)
   u = v*tan(v); u2v2 = u*u + v*v
   A = 0.5*fp.log(u2v2) - u/(u2v2)
   B = 2r*fp.sqrt(2*fp.pi)*fp.sqrt(u2v2)/((u+1r)**2r+v*v)**0.25r
   a = fp.atan(v/u) + v/u2v2
   b = fp.atan(v/u) - 0.5r*fp.atan(v/(u+1r))
   approx_sign = mp.cos(a*n+b)
   approx_magnitude = B / mp.sqrt(n) * mp.exp(n*A)
   return (approx_sign, approx_sign * approx_magnitude, approx_magnitude)

for n, g in enumerate(gammas):
   approx_sign, approx, approx_magnitude = appr(n)
   gammas_approx.append(approx)
   gammas_approx_magnitude.append(approx_magnitude)

nn = range(len(gammas))
err1 = [abs((gammas[i] - gammas_approx[i])/gammas[i]) for i in nn]
err2 = [abs((gammas[i] - gammas_approx[i])/gammas_approx_magnitude[i]) for i in nn]



import matplotlib.pyplot as plt
plt.clf()
plt.loglog(nn, err1, color="blue"); plt.ylim([1e-10, 1e1]); plt.grid(True)
plt.show()

plt.clf()
plt.loglog(nn, err2, color="blue"); plt.ylim([1e-10, 1e1]); plt.grid(True)
plt.show()
```