# From symbolic functions to numerical code in the context of AO Simulations

**Fabio Rossi, Cedric Plantet, Guido Agapito**

**email: fabio.rossi@inaf.it**

**Github: https://github.com/FabioRossiArcetri (https://github.com/FabioRossiArcetri)**

**This work was developed in the context of the simulations for the MAVIS/VLT instrument**

**MAVIS simulations team @Arcetri:**

- AO expertise: Cedric Plantet, Guido Agapito
- Software design, Python implementation: Fabio Rossi

***wfs2020 presentation, 13/10/2020***

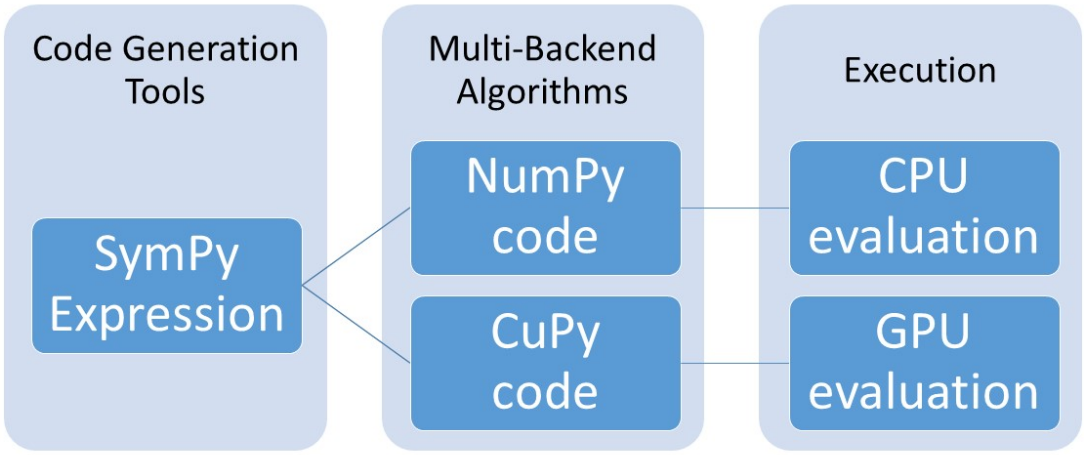
# Motivation

- Write high performace Python code
  - Exploit the **"Array Programming"** paradigm (NumPy/CuPy/ClPy) https://www.nature.com/articles/s41586-020-2649-2#article-info (https://www.nature.com/articles/s41586-020-2649-2#article-info)
- Enforce software engineering best practices
  - modular and reusable code
  - separation of problem statement, algorithms, parameters, input data
  - leverage exisiting libraries
- Critical computations should be easily protable (CPU/GPU execution)

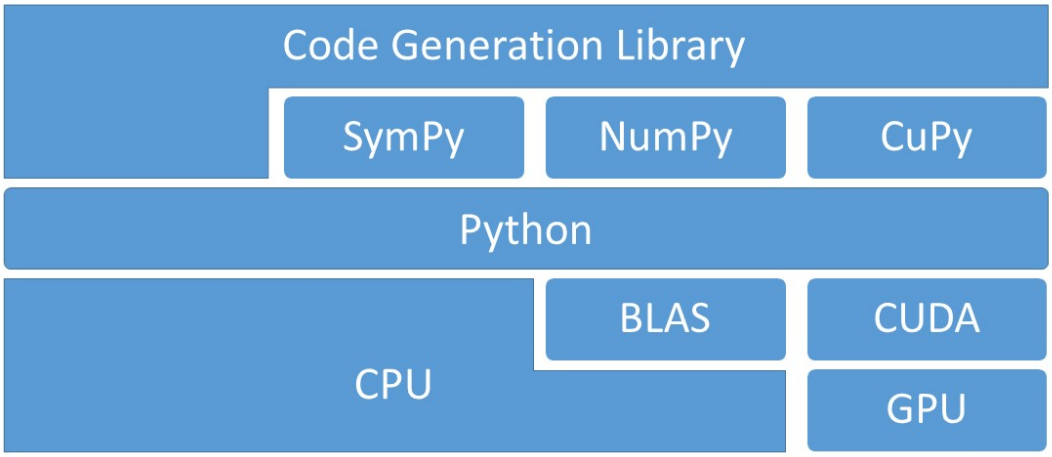
# Symbolic functions and Numerical code

- **Main idea**: Introduce symbolic expression definition and manipulation into Numerical Code development cycle
  - One initial step defining the symbolic functions
  - Symbolic functions can be manipulated in ways numerical functions can't
  - Easily import/export your math from/to other CAS softwares, languages or formats (including LaTex)
  - Symbolic functions can be translated automatically to numerical functions: **"lambdify"** mechanism!
- SymPy is the Python library implementing symbolic computation
  - it is a CAS, Computer Algebra System like Mathematica, MathCad, Maple, but developed in Python and OpenSource
- See appendix A for a 5 minutes introduction to SymPy

# Approach outline



# Software Stack




# SEEING: Sympy Expressions Evaluation Implemented oN the GPU

- Github Repo: https://github.com/FabioRossiArcetri/SEEING (https://github.com/FabioRossiArcetri/SEEING)
  - feel free to checkout, experiment and give feedbacks
- SymPy expressions evaluation on CPU or GPU
  - **Extend the lambdify mechanism to generate CuPy code**
  - Implement functions that are not provided by NumPy and/or CuPy but do exist in SymPy
- Factor out common tasks when using SymPy expressions
  - Provide helper functions to handle groups of related SymPy expressions
  - evaluate, plot, substitute parameters, share variables
- Develop Backend agnostic Numerical Algorithms
  - use of xp.something calls (where xp can be numpy or cupy)
  - currently NumPy and CuPy are supported, in the future clpy or other numerical backends might be added
  - for now plain evaluation and a few Numerical Integration methods, over n-dimensional domains

# SEEING Overview

- Define a SymPy function (or just load some formulas someone else saved to disk)
- Specialize some parameters
- Evaluate a function or an integral

In [1]:
```
from seeing import *
propMethodsCartesian = Formulary.loadFromFile('Propagation100.frm')
approximations = ["Rayleigh-Sommerfeld", "Approximate Rayleigh-Sommerfeld", "Near Fresnel"]
propMethodsCartesian.display(approximations)
```

Rayleigh-Sommerfeld

$$\mathrm{E}_1\left(x_1, y_1, z_1\right) = \int_{-a}^{a}\int_{-a}^{a} -\frac{i E_0 z_1 \left(i\lambda \frac{1}{2\pi\sqrt{z_1^2+\left(-x_0+x_1\right)^2+\left(-y_0+y_1\right)^2}} + 1\right) e^{\frac{2i\pi\sqrt{z_1^2+\left(-x_0+x\right.}}{\lambda}}}{\lambda\left(z_1^2+\left(-x_0+x_1\right)^2+\left(-y_0+y_1\right)^2\right)}$$

Approximate Rayleigh-Sommerfeld

$$\mathrm{E}_1\left(x_1, y_1, z_1\right) = \int_{-a}^{a}\int_{-a}^{a} -\frac{i E_0 z_1 e^{\frac{2i\pi\sqrt{z_1^2+\left(-x_0+x_1\right)^2+\left(-y_0+y_1\right)^2}}{\lambda}}}{\lambda\left(z_1^2+\left(-x_0+x_1\right)^2+\left(-y_0+y_1\right)^2\right)}\, dx_0\, dy_0$$

Near Fresnel

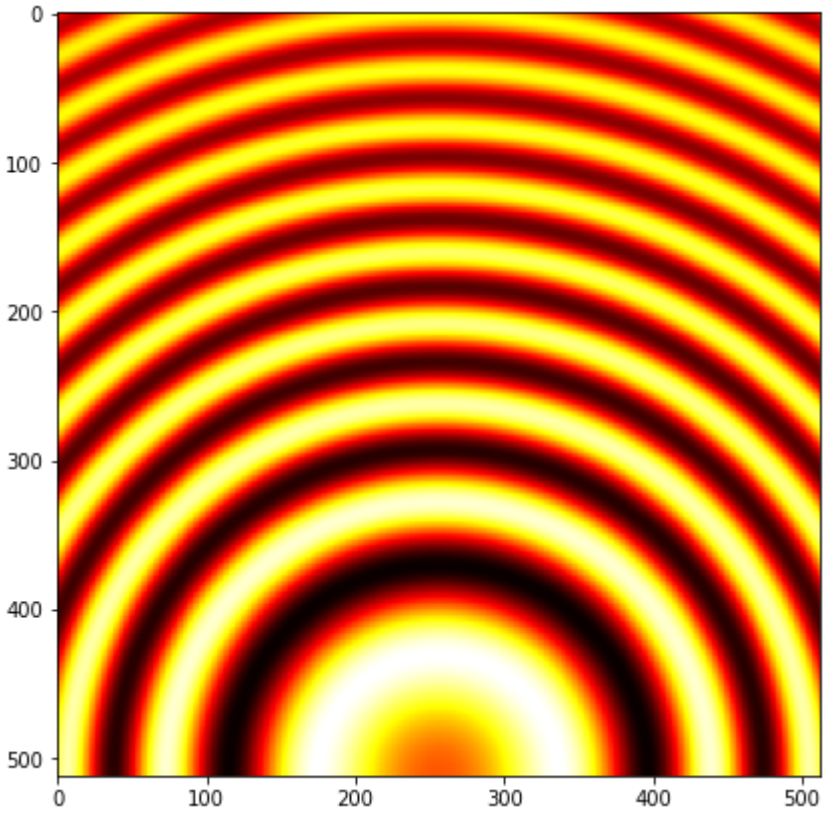
$$\mathrm{E}_1\left(x_1, y_1, z_1\right) = \int_{-a}^{a}\int_{-a}^{a} -\frac{i E_0 e^{\frac{2i\pi z_1}{\lambda}} e^{\frac{i\pi\left(\left(-x_0+x_1\right)^2+\left(-y_0+y_1\right)^2\right)}{\lambda z_1}}}{\lambda z_1}\, dx_0\, dy_0$$

In [2]:
```
propMethodsCartesian.display(["Rayleigh-Sommerfeld Arg"])
```

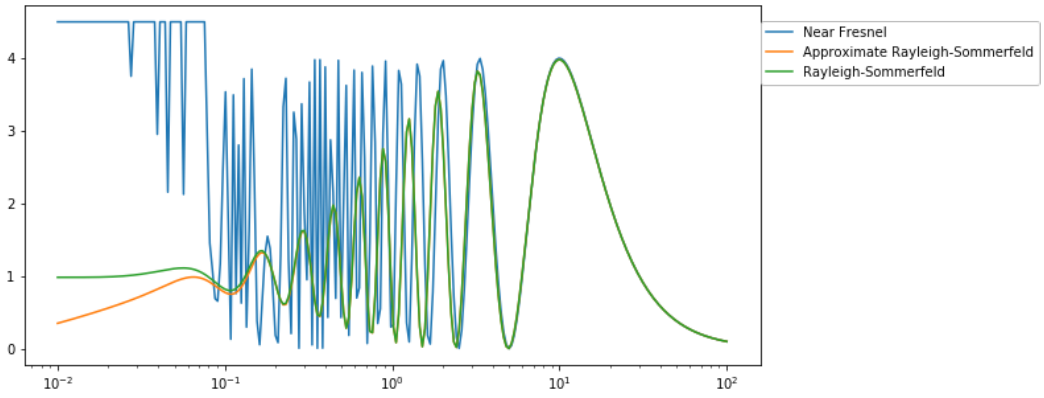
Rayleigh-Sommerfeld Arg

$$-\frac{i E_0 z_1 \left(i\lambda \frac{1}{2\pi\sqrt{z_1^2+\left(-x_0+x_1\right)^2+\left(-y_0+y_1\right)^2}} + 1\right) e^{\frac{2i\pi\sqrt{z_1^2+\left(-x_0+x_1\right)^2+\left(-y_0+y_1\right)^2}}{\lambda}}}{\lambda\left(z_1^2+\left(-x_0+x_1\right)^2+\left(-y_0+y_1\right)^2\right)}$$

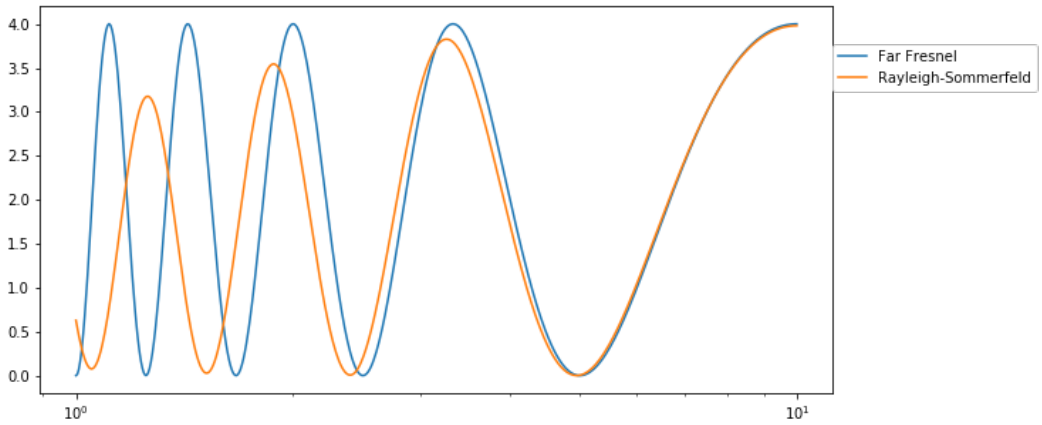
In [3]:
```
waveLength = 10e-6
apertureRadius =  waveLength*10
mCalc = Calculator(cp, cp.float64, 'intensity')
subdiv_points = 512
subsDictC = {'E_0': 1, 'z_1': 2*apertureRadius, 'x_0': 0, 'y_0
': apertureRadius, 'lambda': waveLength, 'a':apertureRadius}
paramAndRanges = [( 'x_1', -apertureRadius, apertureRadius, sub
div_points, 'linear' ),
                  ( 'y_1', -apertureRadius, apertureRadius, sub
div_points, 'linear' )]
propMethodsCartesian.plotFormula("Rayleigh-Sommerfeld Arg", sub
sDictC, paramAndRanges, mCalc)
```



In [4]:
```
subdiv_points = 256
ez0_c =  subsParamsByName( propMethodsCartesian.getFormula('xyC
ircle'), {'a':apertureRadius} )
subsDictC = {'E_0': ez0_c, 'x_1': 0, 'y_1': 0, 'lambda': waveLe
ngth, 'a':apertureRadius}
paramAndRange = ( 'z_1', 0.01*apertureRadius, apertureRadius*10
0, subdiv_points, 'geometric' )
fig, ax = plt.subplots(figsize=(10,5))
plt.xscale('log')
plt.yscale('linear')
for appr in reversed(approximations):
    eeq = propMethodsCartesian.getFormula(appr)
    xplot2, zplot2 = mCalc.IntegralEvalE(subsParamsByName(eeq,
subsDictC), [paramAndRange], None, 'trap')
    plt.plot(xplot2[0]/apertureRadius, np.clip(zplot2, -4.5, 4.
5), label=appr)
plt.legend(loc=(1.0, 0.78))
plt.show()
```

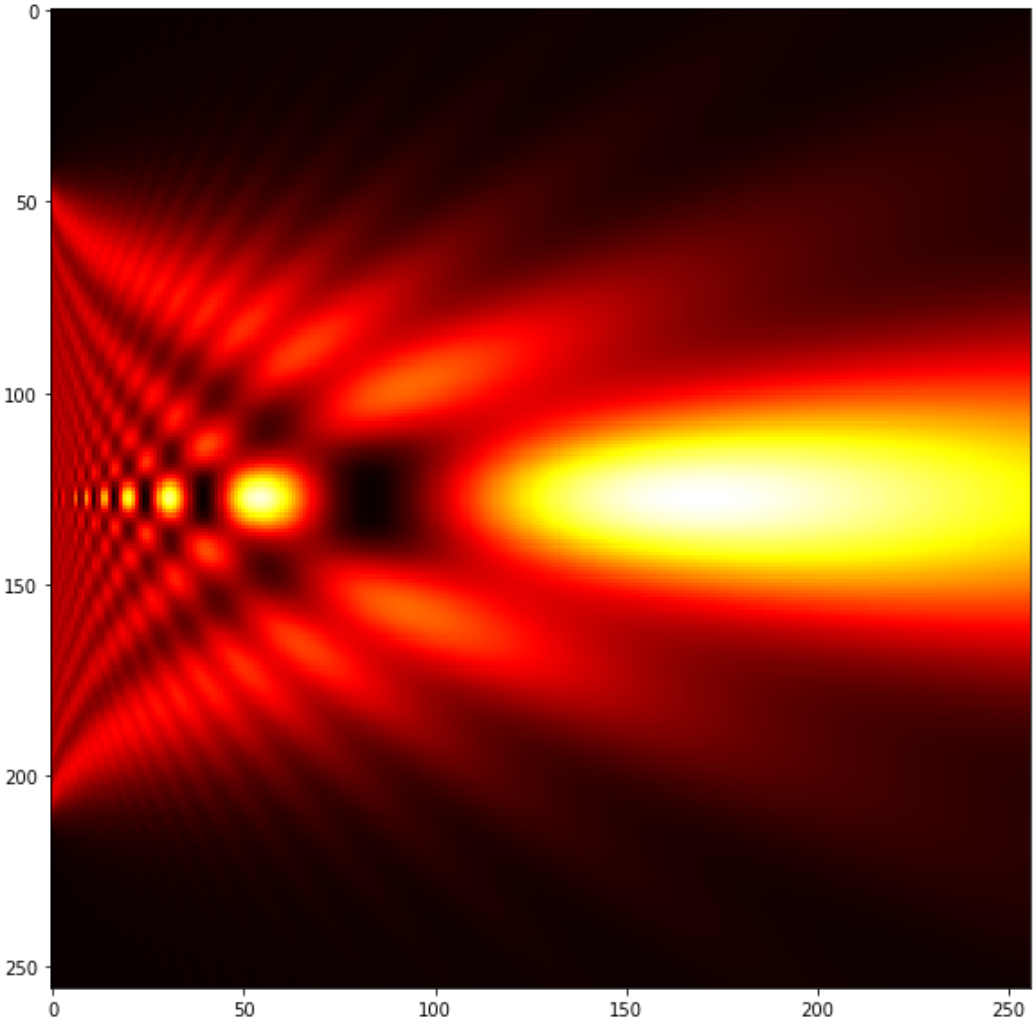


In [5]:
```
approximationsF = ["Rayleigh-Sommerfeld", "Far Fresnel"]
fig, ax = plt.subplots(figsize=(10,5))
plt.xscale('log')
plt.yscale('linear')
paramAndRange = ( 'z_1', apertureRadius, apertureRadius*10, 50
0, 'geometric' )
subsDictC = {'E_0': ez0_c, 'x_1': 0, 'y_1': 0, 'lambda': waveLe
ngth, 'a':apertureRadius}
for appr in reversed(approximationsF):
    eeq = propMethodsCartesian.getFormula(appr)
    xplot2, zplot2 = mCalc.IntegralEvalE(subsParamsByName(eeq,
subsDictC), [paramAndRange], [(subdiv_points, 'linear'), (subdi
v_points, 'linear')], 'rect')
    plt.plot(xplot2[0]/apertureRadius, np.clip(zplot2, -4.5, 4.
5), label=appr)
plt.legend(loc=(1.0, 0.78))
plt.show()
```



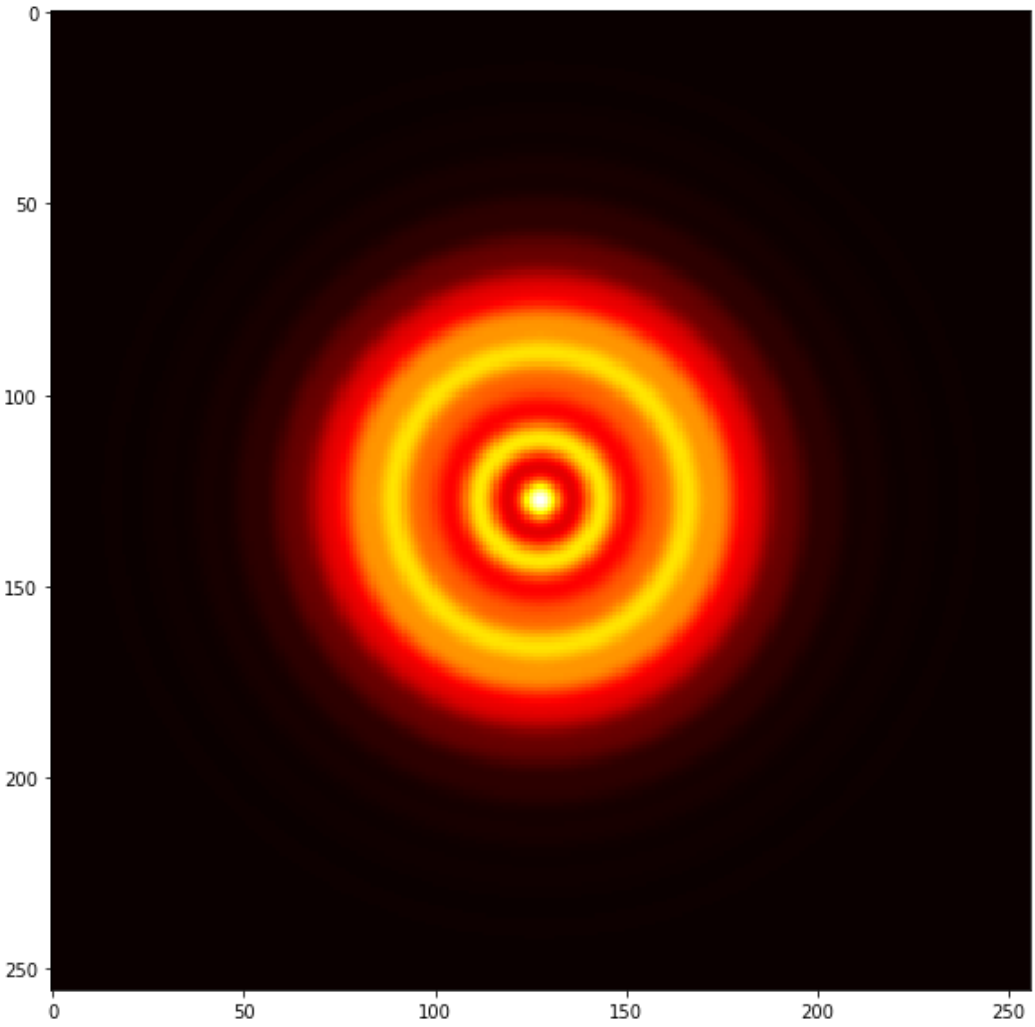
In [6]:
```
subdiv_points = 256
subsDict = {'E_0': ez0_c, 'y_1': 0, 'lambda': waveLength, 'a':a
pertureRadius}
paramsAndRanges = [ ('x_1', -1.5*apertureRadius, 1.5*apertureRa
dius, subdiv_points, 'linear'), ('z_1',  waveLength, 15*apertur
eRadius, subdiv_points, 'linear')]
eeq = propMethodsCartesian.getFormula("Rayleigh-Sommerfeld")
xplot, fplot1 = mCalc.IntegralEvalE(subsParamsByName(eeq, subsD
ict), paramsAndRanges)
fig, ax = plt.subplots(figsize=(10,10))
ax.imshow( fplot1, cmap='hot' )
```

Out[6]: <matplotlib.image.AxesImage at 0x7f910c0c42b0>



In [7]:
```
subsDict = {'E_0': ez0_c, 'z_1': 2*apertureRadius, 'lambda': wa
veLength, 'a':apertureRadius}
paramsAndRanges = [ ('x_1', -2*apertureRadius, 2*apertureRadiu
s, subdiv_points, 'linear'), ('y_1', -2*apertureRadius, 2*apert
ureRadius, subdiv_points, 'linear')]
eeq = propMethodsCartesian.getFormula("Rayleigh-Sommerfeld")
xplot, fplot1 = mCalc.IntegralEvalE(subsParamsByName(eeq, subsD
ict), paramsAndRanges)
fig, ax = plt.subplots(figsize=(10,10))
ax.imshow( np.log(fplot1+1), cmap='hot' )
```

Out[7]: <matplotlib.image.AxesImage at 0x7f918f70da90>




# Using SEEING for MAVIS Simulations

- We are using the approach described so far and the SEEING library to develop simulations related to the development of the MAVIS instrument for the VLT telescope
  - Development of a tool for Natural Guide Star asterism selection
  - Synthetic PSF estimation (both for design purposes and observation planning/evaluation)
- Quite a lot, quite huge expressions!
- Example: Covariance of two Zernike Modes

In [8]:
```
from seeing import *
from zernike import *
mf = Formulary.loadFromFile('Mavis.frm')
mIt = Integrator(cp, cp.float64, 'none')
covIntegral = mf['ZernikeCovarianceI']
display(covIntegral)
```

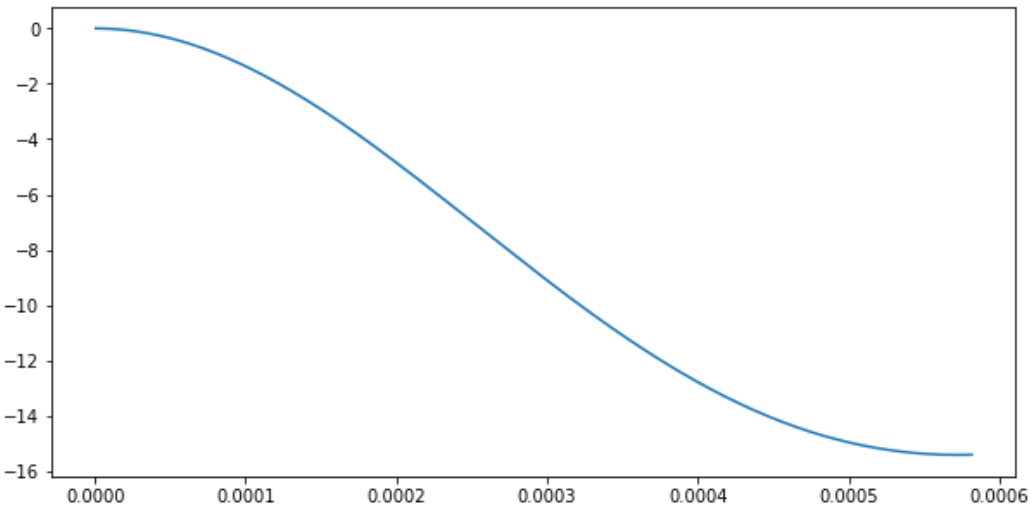
$$0.0229(-1)^{m_k} 2^{(-0.5)\delta_{0m_j}-0.5\delta_{0m_k}+1} i^{n_j+n_k} \sqrt{\left(n_j+1\right)\left(n_k+1\right)} \left(i^{3m_j+}$$

$$\mathrm{W}_{\phi}\left(\rho\right) = \int_{f_{min}}^{f_{max}} \frac{+ i^{3|m_j-m_k|} \cos\left(\theta\left(m_j-m_k\right) + \frac{\pi\left(\left(1-\delta_{0m_j}\right)\left((-1)^j-1\right)-\left(1-\delta_{0m_k}\right)\left((-1)^k-\right.\right.}{4}}{\pi R_1 R_{2.}}$$

In [9]:
```
r0_Value = 0.15 # [m]
L0_Value = 25.0 # [m]
TelescopeDiameter = 8.0 #[m]
DM_height = 10000 # [m]
rho_max = 120.0/206265.0
paramsDict = {'f_min':0.001, 'f_max':10, 'theta': np.pi/4.0, 'L
_0': L0_Value,
              'r_0': r0_Value, 'R_1': TelescopeDiameter/2.0, 'R
_2': TelescopeDiameter/2.0, 'h': DM_height }
_integral1 = subsParamsByName( cov_expr_jk(covIntegral, 2, 3),
paramsDict)
display(_integral1)
```

$$\mathrm{W}_{\phi}\left(\rho\right) = \int_{0.001}^{10} -\frac{0.1351937846348 99 J_2^2\left(8.0\pi f\right) J_2\left(20000\pi f\rho\right)}{\pi f\left(f^2+0.0016\right)^{\frac{11}{6}}}\, df$$

In [10]:
```
xplot1, zplot1 = mIt.IntegralEvalE(_integral1, [('rho',  0.0, r
ho_max, 1000, 'linear')], [(1024, 'linear')], method='rect')
fig, ax = plt.subplots(figsize=(10,5))
ax.plot(xplot1[0], zplot1)
plt.show()
```



# This is interesting, now what?

- Read the reminder of this presentation, including the Appendixes
- Refresh your NumPy skills (ufuncs, array programming)
  - remember: loops are bad !!!
- Start using SymPy: don’t need to become an expert, just need to able to define your expressions/functions
- Understand basic concepts of GPU programming by experimenting with CuPy: if you know NumPy it’s going to be almost straightforward
- “SEEING” library could help you to easily manipulate/lambdify/evaluate your math on CPU/GPU

# Be relieved! You don’t need to learn C/C++ and CUDA if you don't want to!

# More details about NumPy, Cupy, Sympy and the development of SEEING

# More on Sympy Lambdify

- Sympy provides the lambdify method to transform a sympy expression to a lambda which is also a composition of ufuncs, for different backends ("math", "mpmath", "numpy", "numexpr", "scipy" etc)
- What about a CuPy?
- We can pass to lambdify a dictionary which maps sympy functions to cupy functions
- The dictionary will include many obvious translations like: gpulib = { ... 'sin': cp.sin …

In [1]:
```
import inspect
from seeing import *

x = sp.symbols('x')
fsympy = sp.sin(x) * sp.exp(x) / (1+sp.sqrt(sp.Abs(x)) )
display(fsympy)
fnumpy = sp.lambdify(x, fsympy, "numpy" )
print("NumPy ufunc:")
print(inspect.getsource(fnumpy))
print(fnumpy.__globals__['sin'])
print()
print()
fcupy = sp.lambdify(x, fsympy, gpulib )
print("CuPy ufunc:")
print(inspect.getsource(fcupy))
print(fcupy.__globals__['sin'])
```

$$\frac{e^{x} \sin{(x)}}{\sqrt{|x|} + 1}$$

```
NumPy ufunc:
def _lambdifygenerated(x):
    return (exp(x)*sin(x)/(sqrt(abs(x)) + 1))

<ufunc 'sin'>


CuPy ufunc:
def _lambdifygenerated(x):
    return (exp(x)*sin(x)/(sqrt(abs(x)) + 1))

<ufunc 'cupy_sin'>
```

# Problem

- What about SymPy functions which do not have a CuPy analogue?
  - i.e. CuPy only provides the Bessel Functions of the first order of index 0 and 1... but we can implent and use in the lambdify process our own version for general n

In [12]:
```
# somewhere we defined:
def besselj__n(n, z):
    if n==0:
        return cupyx.scipy.special.j0(z)
    elif n==1:
        return cupyx.scipy.special.j1(z)
    elif n>=2:
        return 2*besselj__n(n-1, z)/z - besselj__n(int(n)-2, z)

# and in gpulib dictionary we have: ... 'besselj: 'besselj__n'
...
```

In [13]:
```
from seeing import *

n = sp.symbols('n', integer=True)
fsympy = sp.besselj(n, x)
display(fsympy)
fnumpy = sp.lambdify((x, n), fsympy, "scipy" )
print(inspect.getsource(fnumpy))
print(fnumpy.__globals__['jv'])
fcupy = sp.lambdify((x,n), fsympy, gpulib )
print(inspect.getsource(fcupy))
print(fcupy.__globals__['besselj'])
```

$J_n\left(x\right)$

```
def _lambdifygenerated(x, n):
    return (jv(n, x))

<ufunc 'jv'>
def _lambdifygenerated(x, n):
    return (besselj(n, x))

<function besselj__n at 0x7f910e9e1598>
```

# Array programming

- Do as much as possible using Numpy built-in functions and avoid loops and element-wise access
  - understand what ufuncs are
  - See Appeindix C for a glimpse of performance gains
- Performances: Python-like vs C-like !
- Simplyfied transition to different backends (i.e. CuPy)
- Crucial for GPU programming

# What is a ufunc

- A (basic) ufunc in NumPy (or CuPy) is a function that can be applied to vectors, working an element-wise basis.
  - for example: np.sin, np.cos, np.power
  - np.meshgrid() is not!
  - np.sum() is a reduction ufunc...
  - more details here: https://numpy.org/doc/stable/reference/ufuncs.html (https://numpy.org/doc/stable/reference/ufuncs.html)

# Array programming on the GPU with CuPy

- CuPy API tries to be as similar as possible to NumPy's, https://docs.cupy.dev/en/stable/ (https://docs.cupy.dev/en/stable/)
- Comparison table of the two libraries: https://docs.cupy.dev/en/stable/reference/comparison.html (https://docs.cupy.dev/en/stable/reference/comparison.html)
- Important issue to take care of: CPU<->GPU data transfer
  - if the data set you are performing your computation on is quite small, depending on how complex is your computations, it might get less efficient to transfer it to the GPU and get the result back than simply perform your computation on the CPU
- For a 5 minutes introduction to CuPy see Appendix B

# What is a lambda?

- A lambda is just a (usually small) anonymous function:

In [14]:
```
f = lambda a,b,c : a+b*c
print(f(5, 6, 2))
```

```
17
```

In [15]:
```
# also useful when you want to do have a function that returns
a function:
def myfunc(n):
    return lambda a:a*n

mydoubler = myfunc(2)
print(mydoubler(11))
```

```
22
```

# Adding some functionalities to SymPy

- When manipulating an expression imported from another module, we would like an easy way to access its symbols, symbols' names

In [16]:
```
# lets say we have this function returning a SymPy expression d
efined somewhere
def niceExpr():
    x0, y0, ss = sp.symbols('x_0 y_0 sigma')
    return sp.exp(-x0/ss) + sp.log(1+sp.Abs(y0)*ss)

# then in out main program we could have:
anExpr=niceExpr()
display(anExpr)
```

$$\log\left(\sigma\left|y_0\right|+1\right)+e^{-\frac{x_0}{\sigma}}$$

- just looking at the expression, we can tell there is a parameter named sigma
- we would like to substitute it with some specific real value, how to do it?
- we might not have an easy way to access to the 'ss' variable!!!

In [17]:
```
print(anExpr.free_symbols)
# cannot do this: ss = anExpr.free_symbols[1]

# we can do this:
ss = sp.symbols('sigma')
anExprSpec = anExpr.subs(ss, 0.5)
display(anExprSpec)

# ugly, what if sigma was complex?
```

```
{y_0, sigma, x_0}
```

$$\log\left(0.5\left|y_0\right|+1\right)+e^{-2.0x_0}$$

- We want to have group of formulas sharing the same symbols, this is not obviosly done in SymPy
- We want to implement numerical algorithms that can have in input SymPy expressions and run relying on NumPy or CuPy (same algorithm AND code, different backend)
  - Think about a Monte Carlo integration method that can work using calls like:
    - sp.lambdify(integrand_functions, xp), xp.sum(), xp.random.uniform() etc etc, where xp is a variable point to either np or cp

# Appendix A: 5 minutes intro to Sympy

- Define some symbols
- Define a function
- .. or an integral
- Specialize
- Evaluate

In [18]:
```
import sympy as sp

#Define some symbols
x, y, a, b = sp.symbols("x y a b")
# Define a function
f1 = sp.exp(-a*x**2-b*y**2)
# see how nicely it is displayed
display(f1)
```

$$e^{-ax^2-by^2}$$

In [19]:
```
# Or define an integral
i1 = sp.Integral(f1, (x,-sp.oo,+sp.oo), (y,-sp.oo,+sp.oo))
display(i1)
```

$$\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} e^{-ax^2-by^2}\,dx\,dy$$

In [20]:
```
# Specialize
i2 = i1.subs([(a, 1), (b,2)])
# Evaluate
display(i2.doit())
```

$$\frac{\sqrt{2}\pi}{2}$$

In [21]:
```
#You could also do real symbolic computations… :
fdev = sp.diff(f1, y)
display(fdev)
```

$$-2bye^{-ax^2-by^2}$$

# Appendix B: 5 minutes intro to CuPy

## How to:

- Allocate an array, compose and use some ufuncs
- Move data from the CPU to the GPU
- Move data from the GPU to the CPU

In [22]:
```
import numpy as np
import cupy as cp

npoints = int(1e6)
fcpu = lambda x : np.sum( np.sin(x) * np.exp(x) / (1+np.sqrt(n
p.abs(x)) ) ) / npoints
x_cpu = np.linspace(-1.0, 1.0, npoints)
result = fcpu(x_cpu)
print(type(result), result)

x_gpu = cp.linspace(-1.0, 1.0, npoints)
fgpu = lambda x : cp.sum( cp.sin(x) * cp.exp(x) / (1+cp.sqrt(c
p.abs(x)) ) ) / npoints
result = fgpu(x_gpu)
print(type(result), result)
```

```
<class 'numpy.float64'> 0.17955110645016475
<class 'cupy.core.core.ndarray'> 0.1795511064501647
```

In [23]:
```
x_cpu = np.array([1,2,3])
print(type(x_cpu), x_cpu)
x_gpu = cp.asarray(x_cpu)  # move the data to the current devic
e.
print(type(x_gpu), x_gpu)
```

```
<class 'numpy.ndarray'> [1 2 3]
<class 'cupy.core.core.ndarray'> [1 2 3]
```

In [24]:
```
x_gpu = cp.array([1, 2, 3])  # create an array in the current d
evice
print(type(x_gpu), x_gpu)
x_cpu = cp.asnumpy(x_gpu)  # move the array to the host.
print(type(x_cpu), x_cpu)
```

```
<class 'cupy.core.core.ndarray'> [1 2 3]
<class 'numpy.ndarray'> [1 2 3]
```

# Appendix C: Performance comparison: Python vs NumPy vs Cupy

In [25]:
```
import numpy as np
import cupy as cp
import math

def mySquaredSum(xx):
    r = 0.0
    for i in range(xx.shape[0]):
        r += xx[i]*xx[i]*math.exp(x[i])*math.sin(x[i])**3
    return r

npoints = int(1e5)

x_gpu = cp.linspace(0, 1, npoints)
x = np.linspace(0, 1, npoints)
print(mySquaredSum(x))
```

```
26717.919072457928
```

In [26]:
```
f = lambda x: np.sum(np.square(x)*np.exp(x)*np.sin(x)**3)
print(f(x))
```

```
26717.919072458004
```

In [27]:
```
fgpu = lambda x: cp.sum(cp.square(x)*cp.exp(x)*cp.sin(x)**3)
print(fgpu(x_gpu))
```

```
26717.919072458
```

In [28]:
```
%%timeit
y = mySquaredSum(x)
```

```
143 ms ± 3.32 ms per loop (mean ± std. dev. of 7 runs, 1 loop
each)
```

In [29]:
```
%%timeit
y = f(x)
```

```
9.27 ms ± 2.18 µs per loop (mean ± std. dev. of 7 runs, 100 lo
ops each)
```

In [30]:
```
%%timeit
y = fgpu(x_gpu)
```

```
189 µs ± 93.6 ns per loop (mean ± std. dev. of 7 runs, 10000 l
oops each)
```