Accelerate integrate.quad

In [1]:

```
from numba import njit, cfunc
from numba.types import intc, float64, CPointer
import numpy
import scipy
from scipy import integrate
```

In [2]:

```
# !conda install --yes numba
```

Example to create operator A where

$$A_{ij} = \begin{cases} \frac{1}{2\pi} \int_0^L \frac{\left(x_i - x_j(s)\right) \cos(\beta) + \left(y_i - y_j(s)\right) \sin(\beta)}{\left(x_i - x_j(s)\right)^2 + \left(y_i - y_j(s)\right)^2} ds & \text{if} \quad i \neq j \\ \frac{1}{2} \quad \text{if} \quad i = j \end{cases}$$

Classical usage

In [3]:

```
def integral_vanilla(x, y, xa, ya, beta, length):
    def _integrand(s):
        xb, yb = xa - s * numpy.sin(beta), ya + s * numpy.cos(beta)
        return (((x - xb) * numpy.cos(beta) +
                  (y - yb) * numpy.sin(beta)) /
                 ((x - xb)**2 + (y - yb)**2))
    return integrate.quad(_integrand, 0.0, length)[0]
```

Accelaration using Numba

In [4]:

```
def jit_integrand(integrand):
    jitted_integrand = njit(integrand)

    @cfunc(float64(intc, CPointer(float64)))
    def _wrapped(n, x):
        return jitted_integrand(x[0], x[1], x[2], x[3], x[4],
                                x[5], x[6])

    return scipy.LowLevelCallable(_wrapped.ctypes)
```

In [5]:

```
@jit_integrand
def integrand(s, *args):
    x, y = args[0], args[1]
    xa, ya = args[2], args[3]
    beta = args[4]
    xb, yb = xa - s * numpy.sin(beta), ya + s * numpy.cos(beta)
    return (((x - xb) * numpy.cos(beta) +
              (y - yb) * numpy.sin(beta)) /
             ((x - xb)**2 + (y - yb)**2))
```

In [6]:

```
def integral(x, y, xa, ya, beta, length):
    args = (x, y, xa, ya, beta)
    return integrate.quad(integrand, 0.0, length, args=args)[0]
```

Function to create A

In [7]:

```
def createA(n, integral_func):
    x = numpy.linspace(0.0, 1.0, num=n)
    y = numpy.linspace(0.0, 1.0, num=n)

    beta = numpy.pi / 4
    length = 1.0

    A = numpy.empty((n, n))
    numpy.fill_diagonal(A, 0.5)
    for i in range(n):
        for j in range(n):
            if i != j:
                args = (x[i], y[i], x[j], y[j], beta, length)
                A[i, j] = 0.5 / numpy.pi * integral_func(*args)
    return A
```

Time estimations

In [8]:

```
n = 100  # A will be a n x n matrix
```

In [9]:

```
%%timeit
createA(n, integral_vanilla)
```

```
3.56 s ± 157 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

In [10]:

```
%%timeit
createA(n, integral)
```

```
70.6 ms ± 298 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

In [11]:

```
A = createA(n, integral_vanilla)
A2 = createA(n, integral)
numpy.allclose(A, A2)
```

Out[11]:

```
True
```

In [ ]:

```
```