Problem 1)

A set of three integrals for testing numerical integration:

$$\int_0^1 x^3\,\mathrm{d}x = \frac{1}{4}, \quad \int_0^1 e^x\,\mathrm{d}x = e - 1, \quad \int_0^\pi \sin x\,\mathrm{d}x = 2. \tag{1}$$

(a) Write a program (that will be your code 1) that calculates an integral with a given integrand $f(x)$ in the region $[a, b]$ by one of the Newton-Cotes rules with $n \geq 1$ (the trapezoid rule, or 3-point Simpson's rule, or 5-point Boole's rule or higher) and test it on the set of three "test" integrals. Explore how the accuracy changes with the number of integrals.

(b) Upgrade/modify your code (that will be your code 2) to include the error estimate based on calculations for two step sizes h and $h/2$. You can find the error estimation in the lecture notes (see "Extrapolation and Romberg integration"). Test your code with the set of three test integrals. Now you have a tool to estimate accuracy of numerical integration.

(c) Write a program (that will be your code 3) that calculates an integral from $f(x)$ in the region $[a, b]$ by using Gauss quadratures for 10 points. Coefficients for Gauss quadratures can be found in Abramowitz and Tegun "Handbook of Mathematical Functions", or on the web. Again, test your code with the three test integrals.

(d) Find an integration routine (preferably adaptive) you can use with the language of your choice (Python, C++, or MatLab). That will be your code 4. Again, test your routine with the three test integrals.

(a) In general, we would like to approximate the integral

$$I = \int_a^b f(x)\,\mathrm{d}x\,, \tag{2}$$

which can be done as a weighted sum

$$I \approx \sum_i w_i f(x_i). \tag{3}$$

Of course, it befalls us to choose a good set of x_i and weights w_i in order to obtain a good approximation for I. A generic Newton-Cotes integration rule is derived by approximating the function $f(x)$ over the integration domain with an interpolating polynomial $L(x)$:

$$I = \int_a^b f(x)\,\mathrm{d}x \approx \int_a^b L(x)\,\mathrm{d}x\,. \tag{4}$$

In the Lagrange interpolation scheme, if we have a set of sample points $\{(x_0, f(x_0)), \ldots, (x_n, f(x_n))\}$, we can interpolate $f(x)$ with an n^{th} order polynomial

$$L(x) = \sum_{i=0}^{n} f(x_i) l_i(x), \tag{5}$$

where $\{l_i(x)\}$ is the interpolating basis and $l_i(x_j) = \delta_{ij}$. Thus, with the Newton-Cotes scheme

$$I \approx \sum_{i=0}^{n} f(x_i) \int_a^b l_i(x)\,\mathrm{d}x\,. \tag{6}$$

Thus, the weights are just the integrals of the interpolating polynomials. For this work, we implement the simplest Newton-Cotes rule, which is equivalent to the trapezoid rule. We first break the integration region into N intervals, giving us a sub-interval width $h = (b-a)/N$ and $N+1$ sample points $\{a = x_0, \ldots, x_i, x_{i+1}, \ldots, x_N = b\}$. Using a simple linear interpolation for the sub-interval $[x_i, x_{i+1}]$, our Lagrange basis is

$$l_1(x) = \frac{x - x_{i+1}}{x_i - x_{i+1}} = -\frac{x - x_{i+1}}{h}, \quad l_2(x) = \frac{x - x_i}{h}, \tag{7}$$

and

$$\int_{x_i}^{x_{i+1}} l_1(x)\,\mathrm{d}x = -\frac{(x - x_{i+1})^2}{h}\Big|_{x_i}^{x_{i+1}} = \frac{h}{2} \tag{8}$$

$$\int_{x_i}^{x_{i+1}} l_2(x)\,\mathrm{d}x = \frac{(x - x_i)^2}{h}\Big|_{x_i}^{x_{i+1}} = \frac{h}{2}. \tag{9}$$

Thus, over the sub-interval $[x_i, x_{i+1}]$

$$\int_{x_i}^{x_{i+1}} f(x)\,\mathrm{d}x \approx \frac{h}{2}(f(x_{i+1}) + f(x_i)). \tag{10}$$

If we now sum over all the sub-intervals, we find

$$\begin{aligned}\int_a^b f(x)\,\mathrm{d}x = \sum_{i=0}^{N} \int_{x_i}^{x_{i+1}} f(x)\,\mathrm{d}x &\approx \frac{h}{2} \sum_{i=0}^{N} [f(x_i) + f(x_{i+1})] \\ &= \frac{h}{2}[f(x_0) + 2f(x_1) + \ldots + 2f(x_{n-1}) + f(x_n)].\end{aligned} \tag{11}$$

Results using the Trapezoidal integration are shown in Fig. 1.

(b) Romberg integration is a method which uses the following motivation. Let us denote our approximate integral, which is a function of the step-size h of our sub-intervals, as $I(h)$, where the exact result

$$I = I(h) + \sum_{k=0}^{\infty} a_k h^{n+km}, \tag{12}$$

where n, m are some integers. In particular, if we compute our integral at two step-sizes h and h/N, we have

$$I = I(h) + a_0 h^n + \sum_{k=1}^{\infty} a_k h^{n+km} \tag{13}$$

$$I = I(h/N) + a_0 (h/N)^n + \sum_{k=1}^{\infty} a_k (h/N)^{n+km}. \tag{14}$$

We then have

$$0 = I(h) - I(h/N) + a_0 h^n \left(1 - \frac{1}{N^n}\right) \Rightarrow a_0 h^n = \frac{N^n}{N^n - 1}\Big[I(h/N) - I(h)\Big], \tag{15}$$

where we have neglected $\mathcal{O}(h^{n+m})$ terms. This is effectively gives us an estimate of our trapezoidal integrations and an extrapolated value for the integral

$$I \approx I(h/N) + a_0 (h/N)^n = I(h/N) + \frac{1}{N^n - 1}\Big[I(h/N) - I(h)\Big]. \tag{16}$$

Results using the Trapezoidal integration are shown in Fig. 1.

(c) We now discuss Gauss-Legendre quadrature. First, we make the change of variables $x = \frac{b-a}{2}t + \frac{a+b}{2}$ such that

$$I = \int_a^b f(x)\,\mathrm{d}x = \int_{-1}^1 \frac{b-a}{2} f(x(t))\,\mathrm{d}t\,. \tag{17}$$

Again, we would like to find suitable weights and sample points $\{(w_i, x_i)\}$ such that

$$I \approx \sum_{i=1}^n w_i f(x_i). \tag{18}$$

Our inspiration here is the Taylor expansion

$$f(x(t)) = \sum_{n=0}^{2N-1} a_n t^n + \mathcal{O}(t^{2N}) \approx p_{2N-1}(t), \tag{19}$$

over the interval $[-1, 1]$. At this point, we would like to define our weights and sample points such that

$$\int_{-1}^1 p_{2N-1}(t)\,\mathrm{d}t = \sum_{i=1}^N w_i p_{2N-1}(t_i). \tag{20}$$

Following through, we have

$$\sum_{n=0}^{2N-1} a_n \frac{1 - (-1)^{n+1}}{n+1} = \sum_{i=0}^N w_i \sum_{n=0}^{2N-1} a_n t_i^n \Rightarrow \sum_{i=1}^N w_i t_i^n = \frac{1 + (-1)^n}{n+1} \tag{21}$$

for all $n \in \{0, \ldots, 2N-1\}$. We thus have $2N$ – not necessarily linear – equations for $2N$ unknowns. Let us solve this for a the simplest case of $N = 1$. First, for $N = 1$, we have only two equations

$$\begin{cases} w_1 + w_2 = 2 \\ w_1 t_1 + w_2 t_2 = 0 \\ w_1 t_1^2 + w_2 t_2^2 = \dfrac{2}{3} \\ w_1 t_1^3 + w_2 t_2^3 = 0. \end{cases} \tag{22}$$

Carrying out the algebra, one finds $w_1 = w_2 = 1$ and $t_1 = -t_2 = 1/\sqrt{3}$. One could continue producing weights and sample points for larger N, but it turns out that we can be a little bit more clever in how we determine them. While the details are beyond the scope of this report, we can choose our points t_i such that they are roots of the N^{th} Legendre polynomial, $P_N(t)$. That is $P_N(t_i) = 0$. These roots have several nice properties, including that they are symmetrically distributed about $t = 0$ and all distinct. The corresponding weights are given by

$$w_i = \frac{2}{(1 - t_i^2)[P_N'(t_i)]^2}. \tag{23}$$

These weights and roots are tabulated within the *numpy* library, and as such, we can pulled up to an arbitrarily large N. Before concluding, it should be noted that the sucess of this method depends on the behavior of our integrand. That is, our function and its derivatives should be reasonably smooth in order to be well approximated by a sufficiently low-order polynomial over the integration range of interest. Results using the fixed-order Gaussian quadrature discussed here is shown in Fig. 1.

(d) The *scipy.integrate* module provides a generic adaptive integration routine *quad*. We use this to benchmark our other quadrature implementations. Results using this function are shown in Fig. 1.

The code below implements all the essential quadrature algorithms.

```
import numpy as np
from scipy.integrate import quad


def trap_quad(f,a,b,N=10):
    x_i = np.linspace(a,b,N+1)
    h   = (b-a)/N
    w_i = h/2*np.array([1]+(N-1)*[2]+[1])
    f_i = f(x_i)
    return np.sum(w_i*f_i)

def rom_quad(f,a,b,N=10,R=2):
    I1  = trap_quad(f,a,b,N)
    IR  = trap_quad(f,a,b,R*N)
    err = 1/(R**2 - 1)*(IR - I1)
    return IR+err

def gauss_quad(f,a,b,n=10):
    u_i,w_i = np.polynomial.legendre.leggauss(n)
    x_i = (b-a)*u_i/2 + (a+b)/2
    f_i = f(x_i)
    Jac = (b-a)/2
    return np.sum(Jac*w_i*f_i)

def gauss_quad_improper(f,a,b,n=10):
    f_ = lambda t: f(np.tan(t))/np.cos(t)**2
```

```
    if b == np.inf:
        b = np.pi/2
    else:
        b = np.arctan(b)

    if a == -np.inf:
        a = -np.pi/2
    else:
        a = np.arctan(a)

    return gauss_quad(f_,a,b,n)

def adaptive_gauss_quad(f,a,b,n=10,reps=1e-2,aeps=1e-2):
    I1 = gauss_quad(f,a,b)

    m  = (a+b)/2
    I2 = gauss_quad(f,a,m,n) + gauss_quad(f,m,b,n)

    adiff = np.abs(I2 - I1)
    rdiff = np.abs(adiff/(I1 + 1e-100))
    if adiff < aeps or rdiff < reps:
        return I2
    else:
        return gauss_quad(f,a,m,n)+gauss_quad(f,m,b,n)
```

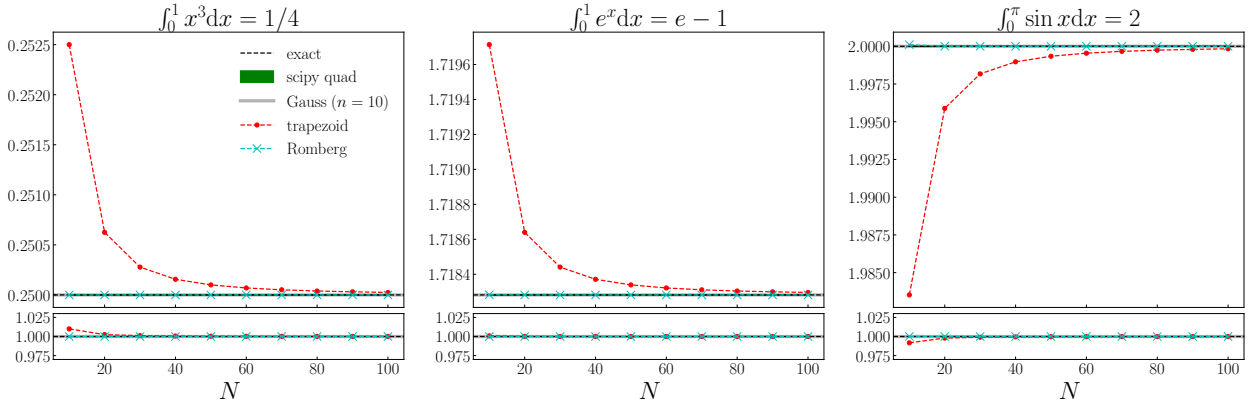

Figure 1: Results for the various quadrature routines discussed in this problem. The top row of panels displays the trapezoidal and Romberg integrations as functions of the number of intervals against the fixed-order Gaussian quadrature, *quad*, and exact results, and the bottom row of panels displays the ratio of each implemented method relative to the exact result.

Problem 2)

Evaluate the following integrals using your codes 2, 3, and 4:

(a) $\displaystyle\int_0^1 \frac{1}{1-0.998x^2}\,dx$

(b) $\displaystyle\int_0^{2\pi} x\sin(30x)\cos(50x)$

(c) $\displaystyle\int_0^1 \frac{x}{e^x-1}\,\mathrm{d}x$

(d) $\displaystyle\int_0^1 x\sin\left(\frac{1}{x}\right)\mathrm{d}x$

Report if one of your codes fails to compute some of the integrals. Explain why.

The results are shown in Table 1. Most of the non-adaptive codes implemented in problem 1 fail to produce an accurate result across each integral here.

(a) The integrand has a singularity at $x = 1/\sqrt{0.998} \approx 1.001$ and is therefore very steep in this region. Thus, we would need a large number of sub-intervals to accurately capture the behavior in this region. Additionally, this function requires many terms in the Taylor expansion to accurately capture the behavior as $x \to 1$, meaning that while the Gaussian quadrature performs better than the Trapezoidal integration on the same number of function evaluations, we should crank up n in order to more accurately calculate the area under the curve.

(b) The integrand here oscillates very rapidly over the region of interest, making the quadrature routines fairly unstable for small numbers of function evaluations. While not done here, it would not be difficult to split the integration region between the integrand's zeros, apply the quadrature routines on these sub-intervals, and sum each contribution. This is the typical advice for periodic functions. It can be seen especially that the performance of the Gaussian quadrature implementation becomes much more accurate with a relatively small n on each sub-interval.

(c) Observe that

$$\frac{x}{e^x-1} \to 1 - \frac{x}{2} + \dots \tag{24}$$

as $x \to 0$, so in this case, instead of a singularity at $x = 0$ we have some indeterminacy. Of course, though, naively applying trapezoidal and Romberg integration on the provided integration domain provides non-finite results since we evaluate at exactly $x = 0$ and returns $0/0$ even though our function is finite throughout the integration domain. In practice, we could amend these issues by providing an alternate definition of our function such that at $x = 0$, we return 1, filling in the point discontinuity that is formally there.

With Gaussian integration, we avoid this subtle business since there is no function evaluation at the end points and obtain reasonably accurate results with a small number of function evaluations.

(d) The discussion here is similar to that of parts (b) and (c). Observe that our function

is formally undefined at $x = 0$, but this discontinuity is simply a hole that could be filled by redefining our function to take the limiting value at $x \to 0$. Additionally, we can observe that our function oscillates rapidly in this limit, and even if this oscillation is not periodic, we can analytically determine the location of the integrand's roots, which would allow us to simply split our integration domain (although this is not done here). It turns out that fixed-order Gaussian integration and adaptive integration do a reasonably good job of providing an accurate value for this integral.

Part	Trapezoid ($N = 100$)	Romberg ($N = 100$, R=2)	Gauss ($n = 10$)	*scipy*
a	5.362	4.633	3.175	3.804
b	0.272	0.319	3.303	0.112
c	nan	nan	0.778	0.778
d	nan	nan	0.387	0.379

Table 1: Results of quadrature routines applied to the integrals specified.

```python
import numpy as np
from scipy.integrate import quad
from main import *

import matplotlib.pyplot as plt
plt.rcParams.update({
    'text.latex.preamble': r'\usepackage{amsmath}',
    'text.usetex': True,
    'font.family': 'sans-serif',
    'font.sans-serif': ['Helvetica']
})

if __name__ == '__main__':

    sets = {
        1: [lambda t: 1/(1-0.998*t**2),[0,1]],
        2: [lambda t: t*np.sin(30*t)*np.cos(50*t),[0,2*np.pi]],
        3: [lambda t: t/(np.exp(t) - 1),[0,1]],
        4: [lambda t: t*np.sin(1/t),[0,1]]
    }

    N = 100
    results = {}
    for i in sets:
        f,ab = sets[i]
        results[i] = {}
        results[i]['trap']  = trap_quad(f,*ab,N=N)
        results[i]['rom']   = rom_quad(f,*ab,N=int(N/2))
        results[i]['gauss'] = gauss_quad(f,*ab)
        results[i]['scipy'] = quad(f,*ab)

    print(results)
```

Problem 3)

Evaluate numerically following improper integrals (you may need to modify one of your codes). There are multiple ways to evaluate improper integrals. Explain your choice and results.

(a) $\displaystyle\int_0^\infty \frac{e^{-x^2}}{x^2+1}\,\mathrm{d}x$

(b) $\displaystyle\int_0^\infty \frac{x\sin x}{x^2+1}\,\mathrm{d}x$

(c) $\displaystyle\int_0^\infty e^{-\sqrt{x}}\cos(2x)\,\mathrm{d}x$

While there are nicer methods to handle improper integrals such as these, we can make a simple substitution to map our infinite domain of integration onto a finite domain:

$$x = \tan u \Rightarrow \int_a^b f(x)\,\mathrm{d}x = \int_{\arctan(a)}^{\arctan(b)} \frac{f(\tan u)}{\cos^2 u}\,\mathrm{d}u\,. \tag{25}$$

Notice that if $a = \infty$, then $\arctan a = \pi/2$, or if $b = -\infty$, then $\arctan b = -\pi/2$. With this, we can employ the integration techniques from Problem 1 with some caveats. Results are shown in Table **??**. While our fixed-order integration provides reasonably accurate results for the integrals of parts (a) and (c), it fails to do so for part (b)'s integral, primarly because of the oscillation. For these kinds of integrals, it may be best to simply observe that this function dies off as $1/x$ as $x \to \infty$ and integrate over successive intervals (defined by the roots of the integrand) until some specified tolerance is achieved since over each such interval, our function is well-behaved.

Part	Fixed Gauss	*scipy*
a	0.672 ($n = 10$)	0.672
b	-2.11 ($n = 10$)	1.255
c	0.122 ($n = 1000$)	0.123

Table 2: Results of a modified fixed-order Gaussian quadrature routine and adaptive quadrature applied to the integrals specified above.

```
import numpy as np
from scipy.integrate import quad
from main import *

import matplotlib.pyplot as plt
plt.rcParams.update({
    'text.latex.preamble': r'\usepackage{amsmath}',
    'text.usetex': True,
    'font.family': 'sans-serif',
    'font.sans-serif': ['Helvetica']
})
```

```
if __name__ == '__main__':

    func = lambda t: np.exp(-t**2)/(1+t**2)
    print(gauss_quad_improper(func,0,np.inf,n=10))
    print(quad(func,0,np.inf))
    print()

    func = lambda t: t*np.sin(t)/(1 + t**2)
    print(gauss_quad_improper(func,0,np.inf,n=10))
    print(quad(func,0,np.inf))
    print()

    func = lambda t: np.exp(-np.sqrt(t))*np.cos(2*t)
    print(gauss_quad_improper(func,0,np.inf,n=1000))
    print(quad(func,0,np.inf))
    print()
```

Problem 4)

Evaluate numerically following principal value integrals using one of techniques for princpal value integrals (again, you may need to modify one of your codes).

(a) $\displaystyle\int_0^1 \frac{1}{x^{1/3}}\,\mathrm{d}x$

(b) $\displaystyle\int_{-1}^1 \left(1+\frac{1}{x}\right)\mathrm{d}x$

(c) $\displaystyle\int_{-\infty}^{\infty} \frac{1}{(x-1)(x^2+1)}\,\mathrm{d}x$

(a) We can employ Gaussian quadrature directly on the integral here since our troubles are only at the endpoint $x = 0$ but the routines never evaluate there. It is observed that our routines are quite accurate here.

(b) Unlike in part (a), since our integral has difficulties around $x = 0$, we split explicitly the integration domain into two sub-intervals $[-1, 0] \cup [0, 1]$. Doing so analytically, it is simple to see that the pathological piece $1/x$ cancels in the principal value prescription since it is odd about $x = 0$, and because of this, our integration routines are quite accurate.

(c) This integral can be performed analytically by utilizing partial fraction expansion. It is a simple matter then to see that the singular piece cancels in the principal value prescription, leaving us with a finite Lorentzian. It is interesting to note that fixed-order Gaussian integration performs quite well for part (c) but the *quad* function requires large but not infinite (*np.inf*) bounds to be inserted. In the code shown below, we replace the

bounds with ± 100.

The results are as follows

Part	Fixed Gauss	*scipy*	exact
a	1.499 ($n = 100$)	1.5	1.5
b	2.0 ($n = 10$)	2.0	2
c	-1.021 ($n = 100$)	-1.571	$-\pi/2$

```
import numpy as np
from scipy.integrate import quad
from main import *

func = lambda t: t**(-1/3)
print(gauss_quad(func,0,1,n=100))

func = lambda t: 1 + 1/t
print(gauss_quad(func,-1,0,n=10) + gauss_quad(func,0,1,n=10))

func = lambda t: 1/((t-1)*(t**2+1))
print(gauss_quad_improper(func,-np.inf,1,n=100) + gauss_quad_improper(func
    ,1,np.inf,n=100))
print(quad(func,-100,100)[0])
```

Problem 5)

Evaluate numerically following double integrals.

(a) $\displaystyle\int_{-1}^{1} \mathrm{d}x \int_{0}^{2} \mathrm{d}y \sin\left(x^2 + y^2\right)$

(b) $\displaystyle\int_{0}^{1} \mathrm{d}x \int_{0}^{1-x} \mathrm{d}y \frac{1}{\sqrt{x+y}(1+x+y)^2}$

For these multi-dimensional integrations, we utilize the iterative intregration scheme. That is, our integral of interest is

$$I = \int_a^b \mathrm{d}x \int_{y_1(x)}^{y_2(x)} \mathrm{d}y\, f(x,y) = \int_a^b \mathrm{d}x\, F(x), \tag{26}$$

where

$$F(x) = \int_{y_1(x)}^{y_2(x)} \mathrm{d}y\, f(x,y). \tag{27}$$

Now we can set up $F(x)$ as a function, which employs one integration over the variable y at a given x, and set up a second integration of $F(x)$. While Gaussian quadrature performs

nicer than Monte Carlo integration in one dimension over a large set of functions, one can appreciate here the utility of Monte Carlo methods for higher-dimensional integrations. If one uses Gaussian quadrature with n points for each integration dimension, then in total we will require n^D points for an integral over D variables, which is exponentially poor. However, as we have learned previously, if we sample the integrand, we may hope to reduce the number of points needed to evaluate the integral.

(a) Using the iterated integral presription above and the fixed-order Gaussian quadrature from problem (1), we obtain

$$I \approx 1.753, \tag{28}$$

which is quite accurate since our function is well-behaved over the integration domain.

(b) Using the iterated integral prescription above and the adaptive quadrature function from *scipy*, we obtain

$$I \approx 0.285. \tag{29}$$

Note that we required adaptive quadrature since there is a singularity at the origin.

```
import numpy as np
from scipy.integrate import quad
from main import *

if __name__ == '__main__':

    print(gauss_quad(lambda x: gauss_quad(lambda y: np.sin(x**2 + y**2)
    ,0,2,n=100),-1,1,n=100))

    def f(x,y):
        return 1/np.sqrt(x+y)/(1+x+y)**2

    I1 = lambda x: quad(f,0,1-x,args=(x,))[0]
    print(quad(I1,0,1)[0])
```

Problem 6)

Write a program that employs either adaptive or automatic numerical integration. Appy your code to integrals in part 2.

We implement a simple adaptive routine via recursion. That is, we first compute the integral

$$I_1 = \int_a^b f(x)\,\mathrm{d}x \tag{30}$$

and the two integrals

$$I_2 = \int_a^{(a+b)/2} f(x)\,\mathrm{d}x + \int_{(a+b)/2}^b f(x)\,\mathrm{d}x\,. \tag{31}$$

If the relative and absolute difference between these two integrals are within prescribed tolerances we return I_2. Otherwise, we further split the integration domains of each integration and repeat the conditional checks until they are satisfied or the maximum recursion depth is achieved. The results of implementing this method with fixed Gaussian quadrature as our base function are shown in the following table. It can be see that they match reasonably with our results from problem 2 obtained with the *scipy* adaptive routine.

Part	Adaptive Gauss ($n = 100$)
a	3.804
b	0.118
c	0.778
d	0.379

```python
import numpy as np
from scipy.integrate import quad
from main import *

if __name__ == '__main__':

    sets = {
        1: [lambda t: 1/(1-0.998*t**2),[0,1]],
        2: [lambda t: t*np.sin(30*t)*np.cos(50*t),[0,2*np.pi]],
        3: [lambda t: t/(np.exp(t) - 1),[0,1]],
        4: [lambda t: t*np.sin(1/t),[0,1]]
    }

    results = {}
    for i in sets:
        f,ab = sets[i]
        print(f'{i}: {adaptive_gauss_quad(f,*ab,n=100)}')
```