**Problem 1)**

Write three functions that numerically solves a first order ODE by using:

**(a)** Explicit Euler method: This is a very simple but educational exercise to get feeling about the method.

**(b)** Modified Euler method: The method is a bridge to methods used in particle simulations.

**(c)** The $5^{\text{th}}$ order Runge-Kutta-Fehlberg (RKF45). This is a very practical exercise leading to use a quite powerful method with an error estimate.

(a) The explicit Euler method is perhaps the most simple method for numerically solving a differential equation, typically introduced in a first-year calculus course. A generic first-order ordinary differential equation takes the form

$$\frac{\mathrm{d}x}{\mathrm{d}t} = f(t, x(t)), \tag{1}$$

where $t$ is our independent variable (e.g. time) and $x(t)$ is the function of interest. Of course, we can naively approximate the derivative at $t$ as the slope of the secant line between $t$ and $t + h$, where $h$ is small step. Doing, so we have

$$\frac{\mathrm{d}x}{\mathrm{d}t} \approx \frac{x(t+h) - x(t)}{h} = f(t, x(t)). \tag{2}$$

From this, we can approximate $x(t + h)$ by having some initial value for $x(t)$ through

$$x(t+h) \approx x(t) + hf(t, x(t)), \tag{3}$$

and therefore, we can define a recursive sequence of approximations by first specifying an initial condition $x(t_0) = x_0$ and computing

$$x_{n+1} = x_n + hf(t_n, x_n), \tag{4}$$

where $t_n = t + nh$ and $x_n = x(t_n)$, until we reach some final time $t_1$. Typically, the step-size is derived by specifying the number of steps $N$ between $t_0$ and $t_1$ as $h = (t_1 - t_0)/N$.

(b) As one probably knows from a differential equations class, the Euler method is only $\mathcal{O}(h^2)$ accurate, modulated by the size of the second-derivative of $x$, and hence suffers from some significant systematic defects unless our step size $h \ll (t_1 - t_0)$ (i.e. $N \gg 1$). There are two ways to make these errors smaller. Firstly, we can devise methods such that the local truncation error of order $\mathcal{O}(h^n)$ is such that $n$ is larger, or secondly, we can devise a method which adjusts the size of $h$ at each step. A first improvement to Euler's method is the modified Euler (or Heun's) method, which aims to mitigate the error as follows. If we adopt notation consistent with the general class of Runge-Kutta methods,

we define our recursive sequence as follows:

$$k = x_n + hf(t_n, x_n) \tag{5}$$

$$x_{n+1} = x_n + \frac{h}{2}\Big[f(t_n, x_n) + f(t_{n+1}, k)\Big]. \tag{6}$$

Indeed, by averaging the slopes in the above way, we increase by one power of $h$ the local truncation error to $\mathcal{O}(h^3)$.

(c) As alluded to in part (b) another way to obtain more accurate numerical schemes is to devise one such that our step size is adjusted at each step so that at each step our prescribed error is less than some tolerance. In the RKF45 scheme, we use the $5^{\text{th}}$ order Runge-Kutta scheme for our recursive sequence. Generically, an $N^{\text{th}}$ Runge-Kutta method is given by

$$\begin{aligned} k_1 &= hf(t_n, x_n) \\ k_2 &= hf(t_n + b_1 h, x_n + A_{11}k_1) \\ &\vdots \\ k_{N+1} &= hf(t_n + b_N h, x_n + A_{Ni}k_i) \\ x_{n+1} &= x_n + hc_i^{(N)}k_i, \end{aligned} \tag{7}$$

where the coefficients $b_i$, $c_i^{(N)}$, and $A_{ij}$ are method-specific. Note that for brevity we utilize Einstein summation convention where repeated indices are summed implicitly. Although we do not have the exact solution with which to compute an error, we can define an error as the difference between the $N^{\text{th}}$ and $(N-1)^{\text{th}}$ RK method:

$$e_{n+1} = \Big[c_i^{(N)} - c_i^{(N-1)}\Big]k_i, \tag{8}$$

where $d_i$ is again a vector of coefficients dependent on the method. With this estimate, if our error $e_{n+1} > \epsilon$, where $\epsilon$ is some tolerance, then we redefine our step size as

$$h = \min\{0.9h(\epsilon/|e_{n+1}|)^{1/(N+1)}, 0.5h\} \tag{9}$$

and repeat the step. Alternatively, if $e_{n+1} \leq \epsilon$, then we accept the step and redefine our step size as

$$h = \min\{0.9h(\epsilon/|e_{n+1}|)^{1/(N+1)}, 2h\}. \tag{10}$$

Typically, the coefficients are specified by a Butcher tableau. For the RKF45 method, the Butcher tableau is as follows:

| | | | | | | |
|---|---|---|---|---|---|---|
| 1/4 | 1/4 | | | | | |
| 3/8 | 3/32 | 9/32 | | | | |
| 12/13 | 1932/2197 | -7200/2197 | 7296/2197 | | | |
| 1 | 439/216 | -8 | 3680/513 | -845/4104 | | |
| 1/2 | -8/27 | 2 | -3544/2565 | 1859/4104 | -11/40 | |
| | 16/135 | 0 | 6656/12825 | 28561/56430 | -9/50 | 2/55 |
| | 25/216 | 0 | 1408/2565 | 2197/4104 | -1/5 | 0 |

```python
import numpy as np

def step_Euler(func,t0,x0,h):
    t1 = t0 + h
    x1 = x0 + h*func(x0,t0)
    return np.array([t1,x1,h])


def step_mod_Euler(func,t0,x0,h):
    t1,xp,h = step_Euler(func,t0,x0,h)
    x1 = x0 + 0.5*h*(func(xp,t1) + func(x0,t0))
    return np.array([t1,x1,h])


def step_RKF45(func,t0,x0,h,tol,adaptive=True):
    A = np.array([
        [0,0,0,0,0,0],
        [1/4,0,0,0,0,0],
        [3/32,9/32,0,0,0,0],
        [1932/2197,-7200/2197,7296/2197,0,0,0],
        [439/216,-8,3680/513,-845/4104,0,0],
        [-8/27,2,-3544/2565,1859/4104,-11/40,0]
    ])
    B = np.array([0,1/4,3/8,12/13,1,1/2])

    K = np.zeros(6)
    K[0] = h*func(x0+np.sum(A[0]*K),t0+B[0]*h)
    K[1] = h*func(x0+np.sum(A[0]*K),t0+B[1]*h)
    K[2] = h*func(x0+np.sum(A[2]*K),t0+B[2]*h)
    K[3] = h*func(x0+np.sum(A[3]*K),t0+B[3]*h)
    K[4] = h*func(x0+np.sum(A[4]*K),t0+B[4]*h)
    K[5] = h*func(x0+np.sum(A[5]*K),t0+B[5]*h)

    t1  = t0 + h
    x1  = x0 + np.sum(np.array([16/135,0,6656/12825,28561/56430,-9/50,2/55])*K
    )
    err = np.abs(np.sum(np.array([1/360,0,-128/4275,-2197/75240,1/50,2/55])*K)
    + 1e-100)

    if err < tol or not adaptive:
        if adaptive:
            h1 = min(0.9*h*(tol/err)**(1/6),2*h)
        else:
            h1 = h
        return np.array([t1,x1,h1])
    else:
        h1 = min(0.9*h*(tol/err)**(1/6),h/2)
        return step_RKF45(func,t0,x0,h1,tol)


def solve_ODE(func,t0,x0,h,tf,tol=1e-4,method='RKF45',max_iter=int(1e3),
    adaptive=True):
    if method == 'RKF45':
```

```
        step_func = lambda t0_,x0_,h_: step_RKF45(func,t0_,x0_,h_,tol,adaptive
    )
    elif method == 'modified Euler':
        step_func = lambda t0_,x0_,h_: step_mod_Euler(func,t0_,x0_,h_)
    elif method == 'Euler':
        step_func = lambda t0_,x0_,h_: step_Euler(func,t0_,x0_,h_)

    results = [[t0,x0]]
    for i in range(max_iter):
        t1,x1,h = step_func(*results[-1],h)
        results.append([t1,x1])

        if t1 >= tf:
            break

    return np.array(results).T
```

**Problem 2)**

Solve numerically following ODEs (employing all three methods), and study the effect of step size by calculating the relative error either by comparing with analytic solution or with RKF45 (if analytic solution is not attainable).

**(a)** $x' = x + e^{-t}$ with $x(0) = 0$ in the interval $t \in [0, 1]$.

**(b)** $x' = x + 2\cos t$ with $x(0) = 1$ in the interval $t \in [0, 1]$.

**(c)** $x' = tx^2$ with $x(0) = 1$ in the interval $t \in [0, 1]$.

**(d)** $x' = 1.5 \sin 2x - x \cos t$ with $x(0) = 1$ in the interval $t \in [0, 10]$.

We can obtain analytic solutions for parts (a) – (c), which are as follows:

$$(a) - x(t) = \sinh t \tag{11}$$

$$(b) - x(t) = 2e^t + \sin t - \cos t \tag{12}$$

$$(c) - x(t) = \frac{2}{2 - t^2}. \tag{13}$$

Numeric results are displayed in Fig. 1. Across the board, we can see that the RKF45 and modified Euler methods perform quite well at a step size of $h = 0.1$, being within 1% of the analytic values, while the Euler method tends to start diverging from the exact or RKF45 solution closer to $t = 1$, being as much as roughly 15% from the analytic result in part (c).

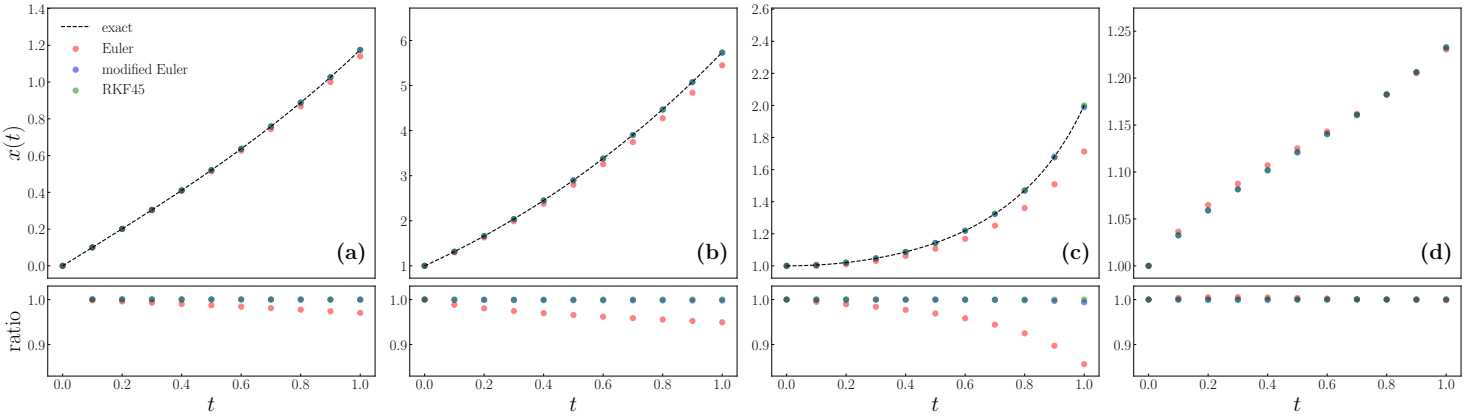


**Figure 1:** In the top row of panels, a comparison of the exact (when an analytic solution is possible to obtain), Euler, modified Euler, and RKF45 (without adaptive step sizes) solutions to the respective differential equation given in problem 2 is shown, and in the lower panel, we plot the ratio of the results of the numerical scheme to the value provided by the exact solution (or RKF45 result when an analytic solution is notavailable).

```
import numpy as np
from scipy.interpolate import interp1d
from scipy.optimize import bisect
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__':

    setups = [
        [lambda x,t: x + np.exp(-t), 0, 0],
        [lambda x,t: x + 2*np.cos(t), 0, 1],
        [lambda x,t: t*x**2, 0, 1],
        [lambda x,t: 1.5*np.sin(2*x) - x*np.cos(t), 0, 1]
    ]
    h  = 0.1
    tf = 1

    results = []
    for setup in setups:
        temp = {}
        for method in ['Euler','modified Euler','RKF45']:
            temp[method] = solve_ODE(*setup,h,tf,method=method,adaptive=
    False)
        results.append(temp)
```

```
exact = [
    lambda t: np.sinh(t),
    lambda t: 2*np.exp(t) + np.sin(t) - np.cos(t),
    lambda t: 2/(2-t**2)
]

T = np.linspace(0,1)
exact_ = [_(T) for _ in exact]

nrows,ncols=2,4
fig,ax = plt.subplots(nrows=nrows,ncols=ncols,figsize=(7*ncols,8),
gridspec_kw={'height_ratios':[6,2]})

color  = ['r','b','g']
labels = [r'$\rm Euler$',r'$\rm modified~Euler$',r'${\rm RKF}45$']
for i in range(4):
    temp = results[i]

    if i == 3:
        interp = interp1d(temp['RKF45'][0],temp['RKF45'][1],kind='cubic
')

    if i != 3:
        ax[0][i].plot(T,exact_[i],'k--',label=r'$\rm exact$')

    for j,method in enumerate(temp):
        _ = temp[method]

        ax[0][i].scatter(_[0],_[1],marker='o',s=50,color=color[j],
edgecolor=color[j],alpha=0.5,label=labels[j] if i==0 else '')

        if i != 3:
            r = _[1]/exact[i](_[0])
            ax[1][i].scatter(_[0],r,marker='o',s=50,color=color[j],
edgecolor=color[j],alpha=0.5)
        else:
            cond = _[0] < _[0][-1]
            r = _[1][cond]/interp(_[0][cond])
            ax[1][i].scatter(_[0][cond],r,marker='o',s=50,color=color[j
],edgecolor=color[j],alpha=0.5)

    ax[0][i].text(s=r'\boldmath $({\rm %s})$'%['a','b','c','d'][i],x
=0.975,y=0.05,size=30,ha='right',va='bottom',transform=ax[0][i].
transAxes)



    for k in range(2):
        ax[k][i].tick_params(axis='both',which='major',direction='in',
labelsize=20)
        ax[k][i].set_xlim(-0.05,1.05)
    ax[1][i].set_xlabel(r'$t$',size=30)
    ax[0][i].set_xticks([],[])
```

```
        ax[1][i].set_ylim(0.83,1.03)

    ax[0][0].set_ylabel(r'$x(t)$',size=30)
    ax[1][0].set_ylabel(r'$\rm ratio$',size=30)
    ax[0][0].legend(fontsize=20,loc='upper left',frameon=False)

    fig.align_labels()
    plt.tight_layout()
    plt.show()
    # fig.savefig(r'prob2.pdf',bbox_inches='tight')
```

**Problem 3)**

Using the error estimation in the $5^{\text{th}}$ order Runge-Kutta-Fehlberg method, you can upgrade your code to include adaptive step control. Apply your upgraded code to problem 2.4. Report your observations when you start either with a very small step or very large step and compare with the results when you do not employ adaptive step size.

From Fig. 2, we can see the performance of our RKF45 with static and adaptive step sizes for three different initial step sizes: $10^{-2}, 10^{-1}$, and 1. It is clear that the solutions cannot be properly reconstructed with one static step, but it is not too bad when we take 10 steps and certainly a more resolved construction when we take 100 static steps. When one permits the step-size to change with a tolerance of $\epsilon = 10^{-4}$, we can see that a step size of about 1/6 is approximately optimal across the time interval to construct the solution, and therefore there is only a slight adjustment of the step sizes when $h_0 = 0.1$. When $h_0 = 1$, the first step is not accepted until a much smaller step of roughly $h = 0.3$ is attained and decrease subsequently. On the other hand, while the steps are accepted initially for $h_0 = 0.01$, the code makes the subsequent steps larger because such small steps are not necessary to accurately produce the result.

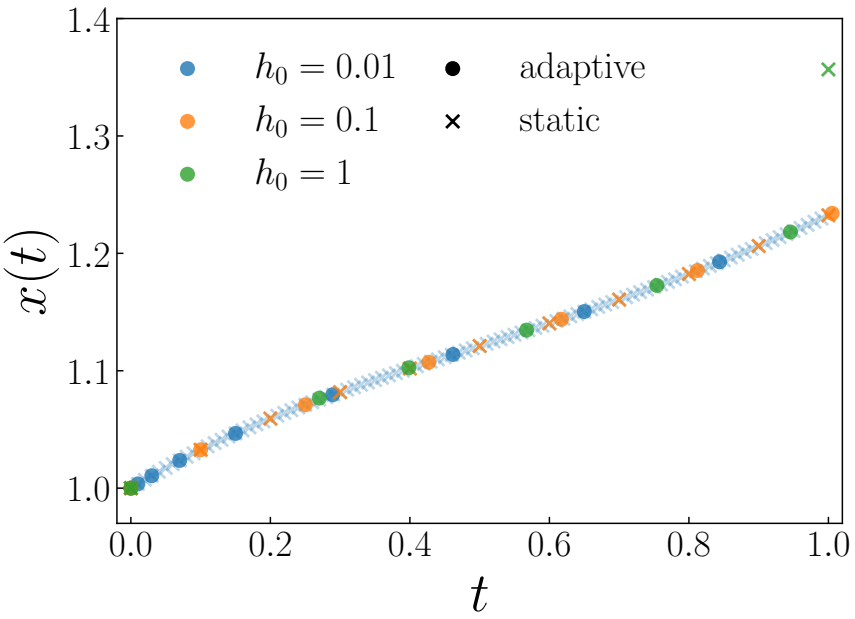


**Figure 2:** Comparison of solutions using the RKF45 method with and without adaptive step sizes for an array of initial step sizes $h_0$.

```
import numpy as np
from scipy.interpolate import interp1d
from scipy.optimize import bisect
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__':

    f = lambda x,t: 1.5*np.sin(2*x) - x*np.cos(t)
    h = [1e-2,0.1,1]
    results  = []
    results1 = []
    for _ in h:
        results.append(solve_ODE(f,0,1,_,1,method='RKF45'))
        results1.append(solve_ODE(f,0,1,_,1,method='RKF45',adaptive=False))


    nrows,ncols = 1,1
    fig,ax = plt.subplots(nrows=nrows,ncols=ncols,figsize=(7*ncols,5*nrows)
    )

    for i,_ in enumerate(h):
```

```
        t,x = results[i]
        ax.scatter(t,x,marker='o',s=50,color='C%d'%i,edgecolor='C%d'%i,
    alpha=0.8,label=r'$h_0=%.1g$'%_)

        t,x = results1[i]
        ax.scatter(t,x,marker='x',s=50,color='C%d'%i,alpha=[0.3,0.8,0.8][i
    ])


    ax.scatter([],[],marker='o',s=50,color='k',label=r'$\rm adaptive$')
    ax.scatter([],[],marker='x',s=50,color='k',label=r'$\rm static$')

    ax.tick_params(axis='both',which='major',direction='in',labelsize=20)
    ax.legend(fontsize=20,loc='upper left',frameon=False,ncol=2,
    columnspacing=0.5)
    ax.set_xlabel(r'$t$',size=30)
    ax.set_ylabel(r'$x(t)$',size=30)
    ax.set_xlim(-0.02,1.02)
    ax.set_ylim(0.97,1.4)

    plt.show()
    # fig.savefig(r'prob3.pdf',bbox_inches='tight')
```

**Problem 4)**

**(a)** Under certain conditions the rate of change of the temperature of a body, immersed in a medium whose temperature (kept constant) differs from it, is proportional to the difference in termperature between it and the medium. In mathematical symbols, this statement is written as

$$\frac{\mathrm{d}T}{\mathrm{d}t} = -k(T - T_M), \tag{14}$$

where $k > 0$ is a proportionality constant, $T$ is the temperature of the body at any time $t$, and $T_M$ is the constant temperature of the medium. Let's say you have a cup of coffee at 90°C initially ($T_0 = 90$) and its placed in a room where the temperature is 20$^{\mathrm{circ}}$C initially ($T_M = 20$). The constant $k$ is given as 0.05 (per minute). We want to find the temperature of the coffee after 10 minutes. Think, how can you test your code?

**(b)** A projectile of mass $m = 1$ kg is thrown vertically upward from the ground with an initial velocity $v_0 = 20$ m/s. The projectile is the subject to linear air resistance so that the force of air resistance is given $f_d = -bv$ with $b = 1$ kg/s. Find its velocity as a function of time. For how long will it rise?

Note: Newton's second law reads

$$m\frac{\mathrm{d}v}{\mathrm{d}t} = -mg - bv, \tag{15}$$

where $m$ is the mass, $g$ is the free-fall acceleration, and $b$ is the linear drag coefficient. Think, how can you test your code?

**(c)** A projectile of mass $m = 1$ kg is thrown vertically upward from the ground with an initial velocity $v_0 = 20$ m/s. The projectile is the subject to linear air resistance so that the force of air resistance is given $f_d = -cv^2$ with $c = 0.1$ kg/m. Find its velocity as a function of time. For how long will it rise?

Note: Newton's second law reads

$$m\frac{\mathrm{d}v}{\mathrm{d}t} = -mg - cv^2, \tag{16}$$

where $m$ is the mass, $g$ is the free-fall acceleration, and $b$ is the linear drag coefficient. Think, how can you test your code?

(a) The exact solution to the differential equation is

$$T(t) = T_0 + T_M(1 - e^{-kt}), \tag{17}$$

so

$$T(10 \text{ min}) \approx 47.543°\text{C}. \tag{18}$$

With the RKF45 method with $\epsilon = 10^{-4}$, we obtain $T(10) \approx 47.504°$C. Howver, with $\epsilon = 10^{-6}$, we have $T(10) \approx 47.542°$C. Although our result is quite accurate in both cases, a lesson to be wary of is that the error can propagate to a noticeable and perhaps significant level over a few steps, so one must always balance computational efficiency and accuracy.

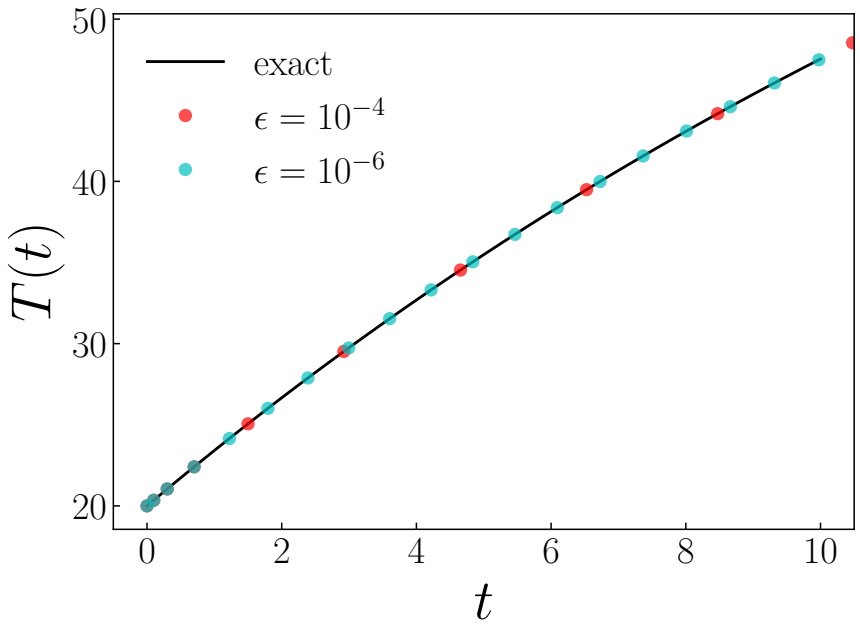


**Figure 3:** Results using RKF45 method to solve Newton's law of cooling equation with the prescribed initial conditions and physical constants.

```python
import numpy as np
from scipy.interpolate import interp1d
from scipy.optimize import bisect
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__':

    k  = 0.05
    TM = 90

    f  = lambda T,t: -k*(T - TM)

    T0 = 20
    tf = 10
    h  = tf/100
    results1 = solve_ODE(f,0,T0,h,tf,method='RKF45',adaptive=True,tol=1e-4)
    results2 = solve_ODE(f,0,T0,h,tf,method='RKF45',adaptive=True,tol=1e-6)

    T = interp1d(*results1)
    print(T(10))
```

```
T = interp1d(*results2)
print(T(10))

print(TM + (T0 - TM)*np.exp(-k*tf))


nrows,ncols = 1,1
fig,ax = plt.subplots(nrows=nrows,ncols=ncols,figsize=(7*ncols,5*nrows)
)

t = np.linspace(0,10)
ax.plot(t,TM + (T0 - TM)*np.exp(-k*t),'k',label=r'$\rm exact$')
ax.plot(results1[0],results1[1],'ro',alpha=0.7,label=r'$\epsilon
=10^{-4}$')
ax.plot(results2[0],results2[1],'co',alpha=0.7,label=r'$\epsilon
=10^{-6}$')

ax.legend(fontsize=20,frameon=False,loc='upper left')
ax.set_xlabel(r'$t$',size=30)
ax.set_ylabel(r'$T(t)$',size=30)
ax.tick_params(axis='both',which='major',direction='in',labelsize=20)
ax.set_xlim(-0.5,10.5)

plt.show()
# fig.savefig(r'prob4a.pdf',bbox_inches='tight')
```

(b) The analytic solution to this differential equation is

$$v(t) = (v_0 + v_t)e^{-t/\tau} - v_t, \tag{19}$$

where $v_t = mg/b$ and $\tau = m/b$ are the terminal speed and characteristic damping time of our system. Solving this equation for $T$ where

$$v(T) = 0 \Rightarrow T = \tau \ln\left(1 + \frac{v_0}{v_t}\right) = 1.112 \text{ s} \tag{20}$$

for the input physical parameters and initial condition.

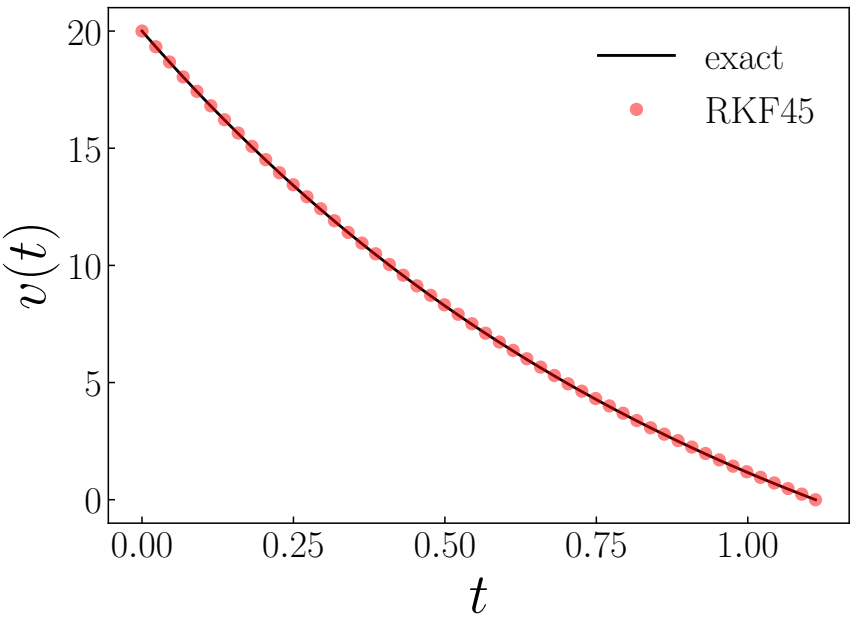


**Figure 4:** Results using the RKF45 method to solve Newton's 2nd law for a vertically launched projectile under a linear drag force with initial velocity $v_0 = 20$ m/s.

```
import numpy as np
from scipy.interpolate import interp1d
from scipy.optimize import bisect
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__':

    m  = 1
    g  = 9.80665
    b  = 1

    a  = lambda v,t: -g - b/m*v
    t0 = 0
    v0 = 20

    results = [[t0,v0]]
    h   = 1e-3
    tol = 1e-4
    while results[-1][1] > 0:
        temp = step_RKF45(a,*results[-1],h,tol)
        results.append(list(temp[:-1]))
```

```
        h = temp[-1]
    results = np.array(results).T

    v  = interp1d(*results)
    tp = bisect(v, results[0][0], results[0][-1])
    print(tp)
    print(m/b*np.log(1+v0/(m*g/b)))

    nrows, ncols = 1,1
    fig, ax = plt.subplots(nrows=nrows, ncols=ncols, figsize=(7*ncols,5*nrows)
    )

    T = np.linspace(*results[0][[0,-1]])
    v_exact = (v0 + m*g/b)*np.exp(-b*T/m) - m*g/b
    ax.plot(T, v_exact, 'k-', label=r'$\rm exact$')
    ax.plot(T, v(T), 'ro', label=r'$\rm RKF45$')

    ax.legend(frameon=False, loc='upper right', fontsize=20)
    ax.set_xlabel(r'$t$', size=30)
    ax.set_ylabel(r'$v(t)$', size=30)
    ax.tick_params(axis='both', which='major', labelsize=20, direction='in')

    plt.show()
    # fig.savefig(r'prob4b.pdf', bbox_inches='tight')
```

(c) In this case, the analytic solution (for purely upward motion – the sign is flipped on the drag force when the projectile moves down) is

$$v(t) = v_c \tan\left[\arctan\left(\frac{v_0}{v_c}\right) - \frac{t}{\tau}\right], \tag{21}$$

where $v_c = \sqrt{mg/c}$ and $\tau = v_c/g$. Solving for the time $T$ where the velocity $v(T) = 0$ we find

$$T = \tau \arctan\left(\frac{v_0}{v_c}\right) \approx 1.122 \text{ s} \tag{22}$$

with the physical parameters and initial speed as given.

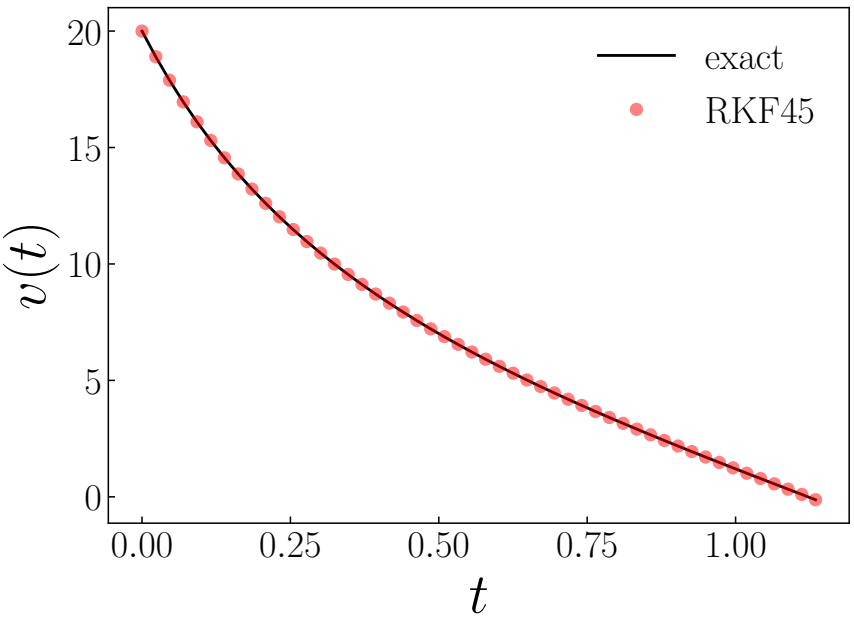


**Figure 5:** Results using the RKF45 method to solve Newton's 2nd law for a vertically launched projectile under a quadratic drag force with initial velocity $v_0 = 20$ m/s.

```python
import numpy as np
from scipy.interpolate import interp1d
from scipy.optimize import bisect
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__':

    m  = 1
    g  = 9.80665
    c  = 0.1

    a  = lambda v,t: -g - c/m*v**2
    t0 = 0
    v0 = 20

    results = [[t0,v0]]
    h   = 1e-3
    tol = 1e-6
    while results[-1][1] > 0:
        temp = step_RKF45(a,*results[-1],h,tol)
        results.append(list(temp[:-1]))
```

```
        h = temp[-1]
    results = np.array(results).T

    v  = interp1d(*results)
    tp = bisect(v,results[0][-2],results[0][-1])
    print(tp)

    vt  = np.sqrt(m*g/c)
    tau = np.sqrt(m/c/g)
    print(tau*np.arctan(v0/vt))

    nrows,ncols = 1,1
    fig,ax = plt.subplots(nrows=nrows,ncols=ncols,figsize=(7*ncols,5*nrows)
    )

    T = np.linspace(*results[0][[0,-1]])

    v_exact = vt*np.tan(np.arctan(v0/vt) - T/tau)
    ax.plot(T,v_exact,'k-',label=r'$\rm exact$')
    ax.plot(T,v(T),'ro',label=r'$\rm RKF45$')
    ax.set_xlabel(r'$t$',size=30)
    ax.set_ylabel(r'$v(t)$',size=30)
    ax.tick_params(axis='both',which='major',direction='in',labelsize=20)
    ax.legend(fontsize=20,loc='upper right',frameon=False)

    plt.show()
    # fig.savefig('prob4c.pdf',bbox_inches='tight')
```