- **Desarrollo Pregunta 1**:

  (a) **[10 puntos]**

  - Si se define $\alpha = 1$ y $\beta = 0$ entonces la solución de la ecuación diferencial será $\cos(t)$.
  - Si se define $\alpha = 0$ y $\beta = 1$ entonces la solución de la ecuación diferencial será $\sin(t)$.

  (b) **[15 puntos]**

  a) Para resolver el IVP asociado, se procederá a hacer el cambio de variables respectivo:

  $$\begin{aligned} y_1(t) &= y(t), \\ y_2(t) &= \dot{y}(t). \end{aligned}$$

  Entonces, omitiendo la dependencia temporal,

  $$\begin{aligned} \dot{y}_1 &= \dot{y} = y_2, \\ \dot{y}_2 &= \ddot{y} = -y = -y_1. \end{aligned}$$

  Por lo tanto,

  $$\begin{bmatrix} \dot{y}_1 \\ \dot{y}_2 \end{bmatrix} = \dot{\mathbf{y}} = \mathbf{F}(t, \mathbf{y}) = \begin{bmatrix} y_2 \\ -y_1 \end{bmatrix},$$
  $$\begin{bmatrix} y_1(0) \\ y_2(0) \end{bmatrix} = \begin{bmatrix} \alpha \\ \beta \end{bmatrix}.$$

  b) Se obtiene $K = \left\lfloor \dfrac{\tau}{h} \right\rfloor$.

  c) Ahora con el sistema dinámico $\dot{\mathbf{y}} = \mathbf{F}(t, \mathbf{y})$, condiciones iniciales y K definidos. Podemos aplicar cualquiera de los métodos indicados hasta el *time-step* K. En particular los métodos disponibles son: *Backward-Euler*, RK2 o RK4.

  d) Finalmente se debe avanzar al tiempo $t = \tau$ desde el tiempo t_K, esto significa avanzar con un $h_{\text{final}} = \tau - K\,h$. En este caso, para cumplir con lo solicitado, se debe avanzar en el tiempo con el mismo algoritmo seleccionado anteriormente.

(c) **[25 puntos]**

```
'''
input:
    tau     : (float) Value for Tau
    h       : (float) Temporal discretization h
    outputSelector   : (int)
                0 -> returns cos(tau),
                1 -> returns sin(tau),
                2 -> returns alpha*cos(tau)+beta*sin(tau)
    alpha   : (float) Value for alpha if used.
    beta    : (float) Value for beta if used.

    output:
    out_value  : (double) If:
               outputSelector=0, it returns cos(tau)
               outputSelector=1, it returns sin(tau)
               outputSelector=2, it returns alpha*cos(tau)+beta*sin(tau)
'''
def find_cos_sin_tau(tau, h=1e-5, outputSelector=0, alpha=1.0, beta=1.0):

    # Defining 'initial conditions'. This is from part (a) of question.
    y0 = np.zeros(2)
    if outputSelector==0:
        y0[0] = 1
        y0[1] = 0
    elif outputSelector==1:
        y0[0] = 0
        y0[1] = 1
    elif outputSelector==2:
        y0[0] = alpha
        y0[1] = beta

    # Defning f(t,y) for dynamical system
    f = lambda t,y: np.array([y[1],-y[0]])

    # Computing number of steps before reaching t=tau
    K = np.floor(tau/h)

    # This is from part (b) of question.
    # h for final step
    h_final = tau-K*h

    # Time-steps until t=K*h
    y_previous = y0
    y_next = np.zeros(2)
    for k in np.arange(1,K+1):
        y_next = RK4_one_step(y_previous,k*h,f,h)
        y_previous = y_next
    # This is from part (b) of question.
    # Final step.
    y_final = RK4_one_step(y_previous,K*h,f,h_final)

    # Required approximation
    out_value = y_final[0]

    return out_value
```

- **Desarrollo Pregunta 2**:

  (a) **[25 puntos]**

  a) La ecuación diferencial (2) se aproxima numéricamente en cada punto x_k con $k \in \{1, \ldots, n\}$ y $h = 1/(n+1)$ de la siguiente forma:

$$\begin{aligned} y''(x_k) &\approx \frac{y(x_{k-1}) - 2\,y(x_k) + y(x_{k+1})}{h^2} \\ &\approx \frac{y_{k-1} - 2\,y_k + y_{k+1}}{h^2} \end{aligned}$$

  Aplicando en la ecuación diferencial se obtiene que:

$$\begin{aligned} \frac{y_{k-1} - 2\,y_k + y_{k+1}}{h^2} &= 2\,\exp(-2y_k)\,(1 - x_k^2) \\ \frac{y_{k-1} - 2\,y_k + y_{k+1}}{h^2} - 2\,\exp(-2y_k)\,(1 - x_k^2) &= 0 \\ y_{k-1} - 2\,y_k + y_{k+1} - 2\,h^2\,\exp(-2y_k)\,(1 - x_k^2) &= 0 \end{aligned}$$

  Entonces para cada aproximación numérica y_k con $k \in \{1, \ldots, n\}$ se tiene que:

$$\begin{aligned} -2\,y_1 + y_2 - 2\,h^2\,\exp(-2y_1)\,(1 - x_1^2) &= 0 \qquad \text{para } k = 1 \\ y_{k-1} - 2\,y_k + y_{k+1} - 2\,h^2\,\exp(-2y_k)\,(1 - x_k^2) &= 0 \qquad \text{para } k \in \{2, \ldots, n-1\} \\ y_{n-1} - 2\,y_n - 2\,h^2\,\exp(-2y_n)\,(1 - x_n^2) &= 0 \qquad \text{para } k = n \end{aligned}$$

  con las condiciones iniciales $y_0 = y_{n+1} = 0$. Por lo tanto, se debe resolver el sistema de ecuaciones no lineales descrito anteriormente para encontrar la aproximación numérica y_k con $k \in \{1, \ldots, n\}$.

  b) Se define el vector $\mathbf{y}$ como la aproximación numérica para cada punto x_k con $k \in \{1, \cdots, n\}$, es decir:

$$\mathbf{y} = [y_1, y_2, \ldots, y_{n-1}, y_n]^\top$$

  Luego, el sistema de ecuaciones no lineales descrito anteriormente se puede escribir como:

$$\mathbf{F}(\mathbf{y}) = \begin{bmatrix} f_1(\mathbf{y}) \\ f_2(\mathbf{y}) \\ \vdots \\ f_k(\mathbf{y}) \\ \vdots \\ f_{n-1}(\mathbf{y}) \\ f_n(\mathbf{y}) \end{bmatrix} = \begin{bmatrix} -2\,y_1 + y_2 - 2\,h^2\,\exp(-2y_1)\,(1 - x_1^2) \\ y_1 - 2\,y_2 + y_3 - 2\,h^2\,\exp(-2y_2)\,(1 - x_2^2) \\ \vdots \\ y_{k-1} - 2\,y_k + y_{k+1} - 2\,h^2\,\exp(-2y_k)\,(1 - x_k^2) \\ \vdots \\ y_{n-2} - 2\,y_{n-1} + y_n - 2\,h^2\,\exp(-2y_{n-1})\,(1 - x_{n-1}^2) \\ y_{n-1} - 2\,y_n - 2\,h^2\,\exp(-2y_n)\,(1 - x_n^2) \end{bmatrix} = \mathbf{0}$$

  Luego resolviendo el problema de búsqueda de raíz $\mathbf{F}(\mathbf{y}) = \mathbf{0}$ se encuentra la solución de la aproximación numérica y_k para $k \in \{1, \ldots, n\}$.

  c) Se utiliza el método de Newton en $\mathbb{R}^n$ para resolver el problema de búsqueda de raíz $\mathbf{F}(\mathbf{y}) = \mathbf{0}$. Para esta tarea se define un *initial guess* $\mathbf{y}_0 = \mathbf{0}$ y se actualiza la aproximación numérica $\mathbf{y}$ mediante la ecuación:

$$\mathbf{y}_{i+1} = \mathbf{y}_i - \underbrace{J_{\mathbf{F}}^{-1}(\mathbf{y}_i)\,\mathbf{F}(\mathbf{y}_i)}_{\mathbf{w}_i} \qquad \text{para } i \in \{0, \ldots, M\}$$

  En cada iteración se debe resolver el siguiente sistema de ecuaciones:

$$J_{\mathbf{F}}(\mathbf{y}_i)\,\mathbf{w}_i = \mathbf{F}(\mathbf{y}_i)$$

  el cual se resuelve con el método de GMRes utilizando $\text{afun}(\mathbf{v}) = J_{\mathbf{F}}(\mathbf{y})\,\mathbf{v}$ y considerando la aproximación de primer orden entregada, por lo tanto:

$$\text{afun}(\mathbf{v}) = \frac{\mathbf{F}(\mathbf{y} + \varepsilon\,\mathbf{v}) - \mathbf{F}(\mathbf{y})}{\varepsilon}, \qquad \varepsilon = 10^{-8}$$

BONUS (10 pts.) La matriz Jacobiana viene dada por:

$$J_{\mathbf{F}}(\mathbf{y}) = \begin{bmatrix} z_1 & 1 & & & & & \\ 1 & z_2 & 1 & & & & \\ & \ddots & \ddots & \ddots & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & \ddots & \ddots & \ddots & \\ & & & & 1 & z_{n-1} & 1 \\ & & & & & 1 & z_n \end{bmatrix}$$

donde

$$\begin{aligned} z_1 &= -2 + 4\,h^2\,\exp(-2y_1)(1 - x_1^2) \\ z_2 &= -2 + 4\,h^2\,\exp(-2y_2)(1 - x_2^2) \\ &\vdots \\ z_{n-1} &= -2 + 4\,h^2\,\exp(-2y_{n-1})(1 - x_{n-1}^2) \\ z_n &= -2 + 4\,h^2\,\exp(-2y_n)(1 - x_n^2) \end{aligned}$$

Luego, el cálculo del producto **matriz-vector** entre la matriz Jacobiana y un vector arbitrario $\mathbf{v}$ de forma **exacta** es:

$$J_{\mathbf{F}}(\mathbf{y})\,\mathbf{v} = \begin{bmatrix} z_1 & 1 & & & & & \\ 1 & z_2 & 1 & & & & \\ & \ddots & \ddots & \ddots & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & \ddots & \ddots & \ddots & \\ & & & & 1 & z_{n-1} & 1 \\ & & & & & 1 & z_n \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ \vdots \\ \vdots \\ v_{n-1} \\ v_n \end{bmatrix} = \begin{bmatrix} z_1\,v_1 + v_2 \\ v_1 + z_2\,v_2 + v_3 \\ \vdots \\ \vdots \\ \vdots \\ v_{n-2} + z_{n-1}\,v_{n-1} + v_n \\ v_{n-1} + z_n\,v_n \end{bmatrix}$$

La implementación viene dada por:

```
def afun_exact(v,y):
    out = np.zeros(n)
    x_vec = np.linspace(0.,1.,n + 2)
    x = x_vec[1:-1]
    z = -2. + 4.*np.power(h,2.)*np.exp(-2.*y)*(1. - np.power(x,2))
    out[1:-1] = v[:-2] + z[1:-1]*v[1:-1] + v[2:]
    out[0] = z[0]*v[0] + v[1]
    out[-1] = v[-2] + z[-1]*v[-1]
    return out
```

(b) **[25 puntos]**

```
'''
input:
    n        : (int)     number of interior points to obtain the numerical
                         aproximations, notice that y(x_0) and y(x_{n+1}) are the
                         boundary conditions.
    gmres_th : (float)   value for threshold used in GMRes method.
    rel_tol  : (float)   value for relative tolerance where norm(F(y))/norm(F(0)) <= rel_tol.
                         If it is achieved, the code needs to return the approximation.
    n_newton : (int)     max number of iterations used in Newton method.
    output:
    y        : (ndarray) numerical approximation for y(x).
'''
def bvp_meteor(n,gmres_th=1e-6,rel_tol=1e-7,n_newton=100):
    # Initialization of output variable.
    # Notice that y[0] and y[-1] must be kept equal to 0 since they are
    # the boundary conditions.
    y = np.zeros(n+2)

    # Defining step
    h = 1./(n + 1)

    #Implementation of F(y)
    def F(y):
        out = np.zeros(n)
        x_vec = np.linspace(0.,1.,n + 2)
        x = x_vec[1:-1]
        # y''(x_i)
        z1 = y[:-2] -2.*y[1:-1] + y[2:]
        # Non-linear component
        z2 = -2.*np.power(h,2.)*np.exp(-2.*y)*(1. - np.power(x,2))
        out[1:-1] = z1 + z2[1:-1]
        out[0] = - 2.*y[0] + y[1] + z2[0]
        out[-1] = y[-2] - 2.*y[-1] + z2[-1]
        return out

    eps = 1e-8
    i = 0
    # n interior points --> [y_1,...,y_n]
    y_int = np.zeros(n)
    # initial guess for GMRes
    w_i = np.zeros(n)
    norm_b0 = np.linalg.norm(F(np.zeros(n)))
    while i < n_newton:
        b = F(y_int)
        # approximation of Jacobian matrix
        afun = lambda v: (F(y_int + eps*v) - F(y_int))/eps
        if np.linalg.norm(b)/norm_b0 < rel_tol:
            break
        w_i = GMRes_matrix_free(afun,b,w_i,gmres_th)
        y_int = y_int - w_i
        i += 1

    y[1:-1] = y_int
    return y
```

Implementación con el cálculo del producto **matriz-vector** entre la matriz Jacobiana y un vector arbitrario **v** de forma **exacta**.

```
def bvp_meteor(n,gmres_th=1e-6,rel_tol=1e-7,n_newton=100):
    # Initialization of output variable.
    # Notice that y[0] and y[-1] must be kept equal to 0 since they are
    # the boundary conditions.
    y = np.zeros(n+2)

    # Defining step
    h = 1./(n + 1)

    #Implementation of F(y)
    def F(y):
        out = np.zeros(n)
        x_vec = np.linspace(0.,1.,n + 2)
        x = x_vec[1:-1]
        # y''(x_i)
        z1 = y[:-2] -2.*y[1:-1] + y[2:]
        # Non-linear component
        z2 = -2.*np.power(h,2.)*np.exp(-2.*y)*(1. - np.power(x,2))
        out[1:-1] = z1 + z2[1:-1]
        out[0] = - 2.*y[0] + y[1] + z2[0]
        out[-1] = y[-2] - 2.*y[-1] + z2[-1]
        return out

    def afun_exact(v,y):
        out = np.zeros(n)
        x_vec = np.linspace(0.,1.,n + 2)
        x = x_vec[1:-1]
        z = -2. + 4.*np.power(h,2.)*np.exp(-2.*y)*(1. - np.power(x,2))
        out[1:-1] = v[:-2] + z[1:-1]*v[1:-1] + v[2:]
        out[0] = z[0]*v[0] + v[1]
        out[-1] = v[-2] + z[-1]*v[-1]
        return out

    eps = 1e-8
    i = 0
    # n interior points --> [y_1,...,y_n]
    y_int = np.zeros(n)
    # initial guess for GMRes
    w_i = np.zeros(n)
    norm_b0 = np.linalg.norm(F(np.zeros(n)))
    while i < n_newton:
        b = F(ya)
        # exact computation of Jacobian matrix
        afun2 = lambda x: afun_exact(x,y_int)
        if np.linalg.norm(b)/norm_b0 < rel_tol:
            break
        w_i = GMRes_matrix_free(afun2,b,w_i,gmres_th)
        y_int = y_int - w_i
        i += 1

    y[1:-1] = y_int
    return y
```