```python
from numba import njit
from scipy.optimize import newton
from numpy import matmul, array, zeros, float64, dot, empty
from numpy.linalg import norm




"""
  Temporal_schemes

    Inputs:
           U : state vector at tn
           dt: time step
           t : tn
           F(U,t) : Function dU/dt = F(U,t)

    Return:
           U state vector at tn + dt
"""


@njit
def Euler(U, dt, t, F):

    return U + dt * F(U, t)


def Inverse_Euler(U, dt, t, F):

    def Residual(X):
          return X - U - dt * F(X, t)

    return newton(func = Residual, x0 = U )

@njit
def Crank_Nicolson(U, dt, t, F ):

    def Residual_CN(X):

         return  X - a - dt/2 *  F(X, t + dt)

    a = U  +  dt/2 * F( U, t)
    return newton( Residual_CN, U )
```

```
@njit
def RK4(U, dt, t, F ):

     k1 = F( U, t)
     k2 = F( U + dt * k1/2, t + dt/2 )
     k3 = F( U + dt * k2/2, t + dt/2 )
     k4 = F( U + dt * k3,   t + dt   )

     return  U + dt * ( k1 + 2*k2 + 2*k3 + k4 )/6




@njit
def Embedded_RK( U, dt, t, F, q, Tolerance):

    #(a, b, bs, c) = Butcher_array(q)
    #a, b, bs, c = Butcher_array(q)

    N_stages = { 2:2, 3:4, 8:13  }
    Ns = N_stages[q]
    a = zeros( (Ns, Ns), dtype=float64)
    b = zeros(Ns); bs = zeros(Ns); c = zeros(Ns)

    if Ns==2:

     a[0,:] = [ 0, 0 ]
     a[1,:] = [ 1, 0 ]
     b[:]  = [ 1/2, 1/2 ]
     bs[:] = [ 1, 0 ]
     c[:]  = [ 0, 1]

    elif Ns==13:
       c[:] = [ 0., 2./27, 1./9, 1./6, 5./12, 1./2, 5./6, 1./6, 2./3 , 1./3,   1., 0.,

       a[0,:]  = [ 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0]
       a[1,:]  = [ 2./27, 0., 0., 0., 0., 0., 0.,  0., 0., 0., 0., 0., 0]
       a[2,:]  = [ 1./36 , 1./12, 0., 0., 0., 0., 0.,  0.,0., 0., 0., 0., 0]
       a[3,:]  = [ 1./24 , 0., 1./8 , 0., 0., 0., 0., 0., 0., 0., 0., 0., 0]
       a[4,:]  = [ 5./12, 0., -25./16, 25./16., 0., 0., 0., 0., 0., 0., 0., 0., 0]
       a[5,:]  = [ 1./20, 0., 0., 1./4, 1./5, 0., 0.,0., 0., 0., 0., 0., 0]
       a[6,:]  = [-25./108, 0., 0., 125./108, -65./27, 125./54, 0., 0., 0., 0., 0., 0.,
       a[7,:]  = [ 31./300, 0., 0., 0., 61./225, -2./9, 13./900, 0., 0., 0., 0., 0., 0]
       a[8,:]  = [ 2., 0., 0., -53./6, 704./45, -107./9, 67./90, 3., 0., 0., 0., 0., 0]
       a[9,:]  = [-91./108, 0., 0., 23./108, -976./135, 311./54, -19./60, 17./6, -1./12
       a[10,:] = [ 2383./4100, 0., 0., -341./164, 4496./1025, -301./82, 2133./4100, 45.
       a[11,:] = [ 3./205, 0., 0., 0., 0., -6./41, -3./205, -3./41, 3./41, 6./41, 0., 0
       a[12,:] = [ -1777./4100, 0., 0., -341./164, 4496./1025, -289./82, 2193./4100, 51

       b[:]  = [ 41./840, 0., 0., 0., 0., 34./105, 9./35, 9./35, 9./280, 9./280, 41./84
       bs[:] = [ 0., 0., 0., 0., 0., 34./105, 9./35, 9./35, 9./280, 9./280, 0          /840


```

```
    k = RK_stages( F, U, t, dt, a, c )
    Error = dot( b-bs, k )

    dt_min = min( dt, dt * ( Tolerance / norm(Error) ) **(1/q) )
    N = int( dt/dt_min  ) + 1
    h = dt / N
    Uh = U.copy()

    for i in range(0, N):

        k = RK_stages( F, Uh, t + h*i, h, a, c )
        Uh += h * dot( b, k )

    return Uh

@njit
def RK_stages( F, U, t, dt, a, c ):

     k = zeros( (len(c), len(U)), dtype=float64 )

     for i in range(len(c)):

        for  j in range(len(c)-1):
          Up = U + dt * dot( a[i, :], k)

        k[i, :] = F( Up, t + c[i] * dt )

     return k









@njit
def Butcher_array(q):

    N_stages = { 2:2, 3:4, 8:13  }

    N =  N_stages[q]
    a = zeros((N, N), dtype = float64);
    b = zeros((N)); bs = zeros((N)); c = zeros((N))
```

```
    if q==2:

     a[0,:] = [ 0, 0 ]
     a[1,:] = [ 1, 0 ]
     b[:]  = [ 1/2, 1/2 ]
     bs[:] = [ 1, 0 ]
     c[:]  = [ 0, 1]

    elif q==3:

      c[:] = [ 0., 1./2, 3./4, 1. ]

      a[0,:] = [  0., 0., 0.,  0        ]
      a[1,:] = [ 1./2, 0., 0., 0        ]
      a[2,:] = [ 0., 3./4, 0., 0       ]
      a[3,:] = [ 2./9,  1./3,  4./9, 0    ]

      b[:]  = [ 2./9,   1./3,  4./9,  0. ]
      bs[:] = [ 7./24,  1./4,  1./3,  1./8 ]

    elif q==8:

       c[:] = [ 0., 2./27, 1./9, 1./6, 5./12, 1./2, 5./6, 1./6, 2./3 , 1./3,   1., 0

       a[0,:]  = [ 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0]
       a[1,:]  = [ 2./27, 0., 0., 0., 0., 0., 0.,  0., 0., 0., 0., 0., 0]
       a[2,:]  = [ 1./36 , 1./12, 0., 0., 0., 0., 0.,  0.,0., 0., 0., 0., 0]
       a[3,:]  = [ 1./24 , 0., 1./8 , 0., 0., 0., 0., 0., 0., 0., 0., 0., 0]
       a[4,:]  = [ 5./12, 0., -25./16, 25./16., 0., 0., 0., 0., 0., 0., 0., 0., 0]
       a[5,:]  = [ 1./20, 0., 0., 1./4, 1./5, 0., 0.,0., 0., 0., 0., 0., 0]
       a[6,:]  = [-25./108, 0., 0., 125./108, -65./27, 125./54, 0., 0., 0., 0., 0.,
       a[7,:]  = [ 31./300, 0., 0., 0., 61./225, -2./9, 13./900, 0., 0., 0., 0., 0.,
       a[8,:]  = [ 2., 0., 0., -53./6, 704./45, -107./9, 67./90, 3., 0., 0., 0., 0.,
       a[9,:]  = [-91./108, 0., 0., 23./108, -976./135, 311./54, -19./60, 17./6, -1.
       a[10,:] = [ 2383./4100, 0., 0., -341./164, 4496./1025, -301./82, 2133./4100,
       a[11,:] = [ 3./205, 0., 0., 0., 0., -6./41, -3./205, -3./41, 3./41, 6./41, 0.
       a[12,:] = [ -1777./4100, 0., 0., -341./164, 4496./1025, -289./82, 2193./4100,

       b[:]  = [ 41./840, 0., 0., 0., 0., 34./105, 9./35, 9./35, 9./280, 9./280, 41.
       bs[:] = [ 0., 0., 0., 0., 0., 34./105, 9./35, 9./35, 9./280, 9./280, 0., 41./

    else:
             print("Butcher array  not avialale for order =", q)
             exit()

    #return (a, b, bs, c)
    return a, b, bs, c
```

# Esquemas temporales

Un esquema temporal es una forma de avanzar en el tiempo para resolver ecuaciones diferenciales, como por ejemplo, como cambia la posición y la velocidad de un objeto que se está moviendo. En vez de calcularlo todo de golpe lo hacemos paso a paso en intervalos de tiempo pequeños.

En este código se implementan diferentes esquemas temporales:

- Euler: toma el estado actual y avanza un paso usando las derivadas actuales
- Euler inverso: Versión mas avanzada, en lugar de usar el estado actual para calcular el siguiente estado, usa un método iterativo para encontrar el próximo estado, lo que lo hace más preciso
- Crank-Nicolson: Esquema mas complej que promedia la tasa de cambio entre el estado actual y el siguiente
- Runge-Kutta de 4º orden (RK4): Toma varias aproximaciones dentro de un mismo paso de tiempo(4aproximaciones),calculando promedios de varias derivadas intermedias. No tienen control de error adaptativo, por lo que no ajusta automáticamente el tamaño de dt.
- Runge-Kutta embebido: es una variante mas avanzada que calcula dos soluciones simultáneamente en cada paso, una de mayor orden y otra de menor orden, usa estos dos resultados para estimar el error del paso actual y ajustar el tamaño del paso de tiempo dt. Si el error es muy grande reduce dt, si es pequeño puede aumentarlo, lo que lo hace mas eficiente y estable

**Código**

El código implementa **esquemas numéricos** (como Euler, Runge-Kutta, etc.) para resolver **ecuaciones diferenciales** mediante la integración temporal. Estos métodos permiten simular cómo evoluciona un sistema dinámico (como órbitas, sistemas físicos, etc.) avanzando en pequeños pasos de tiempo. El usuario elige el esquema que quiere usar y define la función que describe el sistema a resolver.
En el código no está incluida la **función que describe el sistema** (la ecuación diferencial). El código espera que esa función se pase como un argumento cuando uses los **esquemas temporales**.