Anexo 3. `wexler1.py`

Código utilizado para calcular la solución analítica del ejemplo de transporte.

```
    import numpy as np

class Wexler1d:
    """
    Analytical solution for 1D transport with inflow at a
    ↪  concentration of 1.
    at x=0 and a third-type bound at location l.
    Wexler Page 17 and Van Genuchten and Alves pages 66-67
    """

    def betaeqn(self, beta, d, v, l):
        return beta / np.tan(beta) - beta**2 * d / v / l + v * l
        ↪  / 4.0 / d

    def fprimebetaeqn(self, beta, d, v, l):
        """

        f1 = cotx - x/sinx2 - (2.0D0*C*x)
        """
        c = v * l / 4.0 / d
        return 1.0 / np.tan(beta) - beta / np.sin(beta) ** 2 -
        ↪  2.0 * c * beta

    def fprime2betaeqn(self, beta, d, v, l):
        """
        f2 = -1.0D0/sinx2 - (sinx2-x*DSIN(x*2.0D0))/(sinx2*sinx2)
        ↪  - 2.0D0*C

        """
        c = v * l / 4.0 / d
        sinx2 = np.sin(beta) ** 2
        return (
            -1.0 / sinx2
            - (sinx2 - beta * np.sin(beta * 2.0)) / (sinx2 *
            ↪  sinx2)
            - 2.0 * c
        )

    def solvebetaeqn(self, beta, d, v, l, xtol=1.0e-12):
        from scipy.optimize import fsolve

        t = fsolve(
```

```
            self.betaeqn,
            beta,
            args=(d, v, l),
            fprime=self.fprime2betaeqn,
            xtol=xtol,
            full_output=True,
        )
        result = t[0][0]
        infod = t[1]
        isoln = t[2]
        msg = t[3]
        if abs(result - beta) > np.pi:
            raise Exception("Error in beta solution")
        err = self.betaeqn(result, d, v, l)
        fvec = infod["fvec"][0]
        if isoln != 1:
            print("Error in beta solve", err, result, d, v, l,
            ↪  msg)
        return result

    def root3(self, d, v, l, nval=1000):
        b = 0.5 * np.pi
        betalist = []
        for i in range(nval):
            b = self.solvebetaeqn(b, d, v, l)
            err = self.betaeqn(b, d, v, l)
            betalist.append(b)
            b += np.pi
        return betalist

    def analytical(self, x, t, v, l, d, tol=1.0e-20, nval=5000):
        sigma = 0.0
        betalist = self.root3(d, v, l, nval=nval)
        concold = None
        for i, bi in enumerate(betalist):
            denom = bi**2 + (v * l / 2.0 / d) ** 2 + v * l / d
            x1 = (
                bi
                * (bi * np.cos(bi * x / l) + v * l / 2.0 / d *
                ↪  np.sin(bi * x / l))
                / denom
            )

            denom = bi**2 + (v * l / 2.0 / d) ** 2
            x2 = np.exp(-1 * bi**2 * d * t / l**2) / denom
```

```
            sigma += x1 * x2
            term1 = 2.0 * v * l / d * np.exp(v * x / 2.0 / d -
            ↪  v**2 * t / 4.0 / d)
            conc = 1.0 - term1 * sigma
            if i > 0:
                assert concold is not None
                diff = abs(conc - concold)
                if np.all(diff < tol):
                    break
            concold = conc
        return conc

    def analytical2(self, x, t, v, l, d, e=0.0, tol=1.0e-20,
    ↪  nval=5000):
        """
        Calculate the analytical solution for one-dimension
        ↪  advection and
        dispersion using the solution of Lapidus and Amundson
        ↪  (1952) and
        Ogata and Banks (1961)

        Parameters
        ----------
        x : float or ndarray
            x position
        t : float or ndarray
            time
        v : float or ndarray
            velocity
        l : float
            length domain
        d : float
            dispersion coefficient
        e : float
            decay rate

        Returns
        -------
        result : float or ndarray
            normalized concentration value

        """
        u = v**2 + 4.0 * e * d
        u = np.sqrt(u)
        sigma = 0.0
```

```
        denom = (u + v) / 2.0 / v - (u - v) ** 2.0 / 2.0 / v / (u
        ↪  + v) * np.exp(
            -u * l / d
        )
        term1 = np.exp((v - u) * x / 2.0 / d) + (u - v) / (u + v)
        ↪  * np.exp(
            (v + u) * x / 2.0 / d - u * l / d
        )
        term1 = term1 / denom
        term2 = 2.0 * v * l / d * np.exp(v * x / 2.0 / d - v**2 *
        ↪  t / 4.0 / d - e * t)
        betalist = self.root3(d, v, l, nval=nval)
        concold = None
        for i, bi in enumerate(betalist):
            denom = bi**2 + (v * l / 2.0 / d) ** 2 + v * l / d
            x1 = (
                bi
                * (bi * np.cos(bi * x / l) + v * l / 2.0 / d *
                ↪  np.sin(bi * x / l))
                / denom
            )

            denom = bi**2 + (v * l / 2.0 / d) ** 2 + e * l**2 / d
            x2 = np.exp(-1 * bi**2 * d * t / l**2) / denom

            sigma += x1 * x2

            conc = term1 - term2 * sigma
            if i > 0:
                assert concold is not None
                diff = abs(conc - concold)
                if np.all(diff < tol):
                    break
            concold = conc
        return conc

def sol_analytical_t(i, x, atimes, mesh, pparams,
↪  x_axis_time=True):

    a1 = Wexler1d().analytical2(x, atimes,
                                pparams["specific_discharge"] /
                                ↪  pparams["retardation_factor"],
                                mesh.row_length,

                                ↪  pparams["dispersion_coefficient"],
                                pparams["decay_rate"])
```

```
idx = 0

if x_axis_time:
    if idx == 0:
        idx_filter = a1 < 0
        a1[idx_filter] = 0
        idx_filter = a1 > 1
        a1[idx_filter] = 0
        idx_filter = atimes > 0
        if i == 2:
            idx_filter = atimes > 79
    elif idx > 0:
        idx_filter = atimes > 0
else:
    if idx == 0:
        idx_filter = x > mesh.row_length
        if i == 0:
            idx_filter = x > 6
        if i == 1:
            idx_filter = x > 9
        a1[idx_filter] = 0.0

return a1, idx_filter
```