```python
# falkner_skan.py
# By Peter Sharpe

from scipy import optimize, integrate
import numpy as np
import matplotlib.pyplot as plt


def falkner_skan(m, verbose=False):
    """
    Solves the Falkner-Skan equation for a given value of m.
    See Wikipedia for reference: https://en.wikipedia.org/wiki/Falkner-Skan_boundary_layer

    :param m: power-law exponent of the edge velocity (i.e. u_e(x) = U_inf * x ^ m)
    :param verbose: boolean about whether you want to print detailed output (for debugging)
    :return: eta, f0, f1, and f2 as a tuple of 1-dimensional ndarrays.

    Governing equation:
    f''' + f*f'' + beta*( 1 - (f')^2 ) = 0, where:
    beta = 2 * m / (m+1)
    f(0) = f'(0) = 0
    f'(inf) = 1

    Syntax:
    f0 is f
    f1 is f'
    f2 is f''
    f3 is f'''

    """

    # Assign beta
    beta = 2 * m / (m + 1)

    ### Figure out what f2(0) is with the shooting method:
    f2_init_guess = 1.233  # Dear god whatever you do don't change this initial guess, it took so much trial and error to find a initial guess that's stable for all values of m

    # Nelder-Mead simplex optimization algorithm
    opt_result = optimize.minimize(
```

```
        fun=falkner_skan_error_squared,
        x0=f2_init_guess,
        args=(beta,),
        method='nelder-mead',
        options={
            'fatol': 1e-12
        }
    )
    f2_init = opt_result.x
    if verbose:
        print("f''_init: %f" % f2_init)
        print("Residual: %f" % opt_result.fun)

    ### Calculate the solution
    eta = np.linspace(0, 10, 1001)  # values of eta that you want data at
    f_init = [0, 0, f2_init]  # f(0), f'(0), and f''(0)
    soln = integrate.solve_ivp(
        fun=lambda eta, f: falkner_skan_differential_equation(eta, f, beta),
        t_span=(0, 10),
        y0=f_init,
        t_eval=eta,
        method='BDF'  # More stable for stiff problems
    )

    ### Format and return the output
    f0 = soln.y[0, :]
    f1 = soln.y[1, :]
    f2 = soln.y[2, :]
    return eta, f0, f1, f2


def falkner_skan_error_squared(f2_init, beta):
    """
    For a given guess of f''(0) and fixed parameter beta, returns the square of the error of the Falkner-Skan solution
    :param f2_init: Guess of f''(0)
    :param beta: The Falkner-Skan beta parameter (beta = 2 * m / (m + 1) )
    :return: The square of the difference between f'(infinity) and 1, since 1 is the boundary condition that should be enforced.
    """
    eta, f0, f1, f2 = falkner_skan_solution(f2_init, beta)
```

```python

    f1_inf = f1[
        -1]  # Gets the last value of f1 that was calculated (typically at eta = 20, considered far enough to be infinity).

    error_squared = (f1_inf - 1) ** 2

    if f2_init < 0:
        error_squared = np.Inf  # Eliminate separated solutions by adding a "penalty function" for negative f''(0) values.
        # Negative f''(0) values imply negative shear stress at the wall, or separation.
        # This is implemented like this because the Nelder-Mead simplex algorithm in scipy.optimize doesn't support constraints.
        # (This is sort of like a barrier method)

    return error_squared


def falkner_skan_solution(f2_init, beta):
    """
    Returns the Falkner-Skan solution for a given guess f''(0) and fixed parameter beta.
    :param f2_init: Guess of f''(0)
    :param beta: The Falkner-Skan beta parameter (beta = 2 * m / (m + 1) )
    :return: eta, f0, f1, and f2 as a tuple of 1-dimensional ndarrays.
    """
    f_init = [0, 0, f2_init]  # f(0), f'(0), and f''(0)
    raw_soln = integrate.solve_ivp(
        fun=lambda eta, f: falkner_skan_differential_equation(eta, f, beta),
        t_span=(0, 20),
        y0=f_init,
        method='BDF'  # More stable for stiff problems
    )
    eta = raw_soln.t
    f0 = raw_soln.y[0, :]
    f1 = raw_soln.y[1, :]
    f2 = raw_soln.y[2, :]

```

```python
    return eta, f0, f1, f2


def falkner_skan_differential_equation(eta, f, beta):
    """
    The governing differential equation of the Falkner-Skan boundary layer solution.
    :param eta: The value of eta. Not used; just set up like this so that scipy.integrate.solve_ivp() can use this function.
    :param f: A vector of 3 elements: f, f', and f''.
    :param beta: The Falkner-Skan beta parameter (beta = 2 * m / (m + 1) )
    :return:  The derivative w.r.t. eta of the input vector, expressed as a vector of 3 elements: f', f'', and f'''.
    """
    dfdeta = [
        f[1],
        f[2],
        -f[0] * f[2] - beta * (1 - f[1] ** 2)
    ]

    return dfdeta


if __name__ == "__main__":
    # Run through a few tests to ensure that these functions are working correctly.
    # Includes all examples in Table 4.1 of Drela's Flight Vehicle Aerodynamics textbook, along with a few others.
    # Then plots all their velocity profiles.
    m_tests = [-0.0904, -0.08, -0.05, 0, 0.1, 0.3, 0.6, 0.8, 1, 1.2, 1.4, 1.6, 1.8, 2]
    for m_val in m_tests:
        eta, f0, f1, f2 = falkner_skan(m=m_val)
        plt.plot(f1, eta)
    plt.ion()
    plt.grid(True)

```