hw10_p1p2_code

July 2, 2021

[2]:
```
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt
%matplotlib inline
```

1 Problem 1: Nonlinear Shooting with Newton's Method

[72]:
```
# Problem 1: Nonlinear shooting with Newton's Method

def prob1_rhs(w, x):
    """ computes right hand side (see solution derivation). """
    w1, w2, w3, w4 = w[0], w[1], w[2], w[3]
    return np.array([w2, (-w2**2)-w1+np.log(x), w4, -w3-2*w2*w4])


def rk4(w0, h, rhs, x_start=1, x_end=2):
    """ standard Runge-Kutta integrator.

    Input:
        w0                  initial condition
        h                   stepsize
        rhs                 function to compute rhs, takes in w
        x_start, x_end      specify domain for integration
    Output:
        w_history           full dynamics of w from x_start to x_end
        x_grid              grid generated during integration
    """
    # number of grid points
    N = int((x_end - x_start) / h)
    x_grid = np.linspace(x_start, x_end, N+1)
    w_history = np.zeros([len(w0), N+1])
    w_history[:, 0] = w0
    for i in range(N):
        k1 = h * rhs(w_history[:, i], x_grid[i])
        k2 = h * rhs(w_history[:, i] + k1/2, x_grid[i] + h/2)
        k3 = h * rhs(w_history[:, i] + k2/2, x_grid[i] + h/2)
        k4 = h * rhs(w_history[:, i] + k3, x_grid[i] + h)
```

```python
        k = (1/6) * (k1 + 2*k2 + 2*k3 + k4)
        # advance one step
        w_history[:, i+1] = w_history[:, i] + k
    return w_history, x_grid


def nonlinear_shooting(a, b, h, alpha=0, beta=np.log(2), tol=1e-8):
    """ nonlinear shooting algorithm for souling two point BVP.
    Reinterpreted version of alg. 11.2 from Textbook.

    Input:
        a, b                    spatial domain x_start, x_end
        alpha, beta             boundary conditions for y
        h                       step size for integrator
        tol                     tolerance, default to 1e-08

    Output:

    """
    # initialize params
    t0 = ( beta - alpha ) / (b - a)
    w0 = np.array([0, t0, 0, 1])
    w_k = w0
    t_k = t0
    t_history = []
    t_history.append(t_k)
    # loop, breaks loop when error is small
    while True:
        # shoot
        w_k_history, x_grid = rk4(w0, h, prob1_rhs, x_start=1, x_end=2)
        # take approximated right end point, compare with beta
        y_approx = w_k_history[0, w_k_history.shape[1]-1]
        z_approx = w_k_history[2, w_k_history.shape[1]-1]
        end_error = y_approx - beta
        if abs(end_error) < tol:
            print("> solution converged")
            break
        # otherwise adjust initial slope for re-shooting
        t_k = t_k - (end_error / z_approx)
        t_history.append(t_k)
        w0[1] = t_k
    # after looping, return best w_k, and adjustment history
    final_w = w_k_history

    return x_grid, final_w, t_history
```

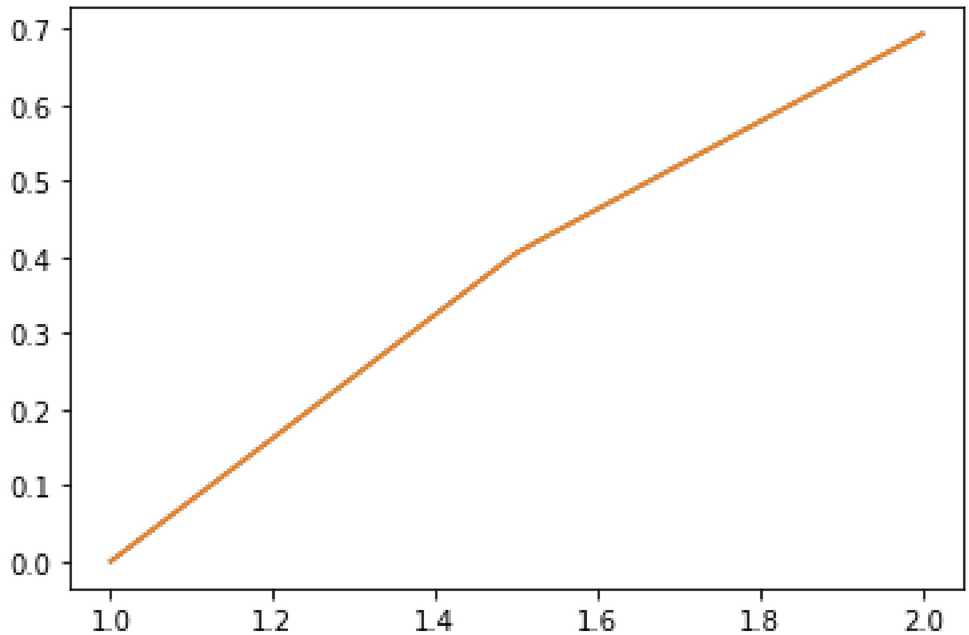
[73]:
```python
# computations
```

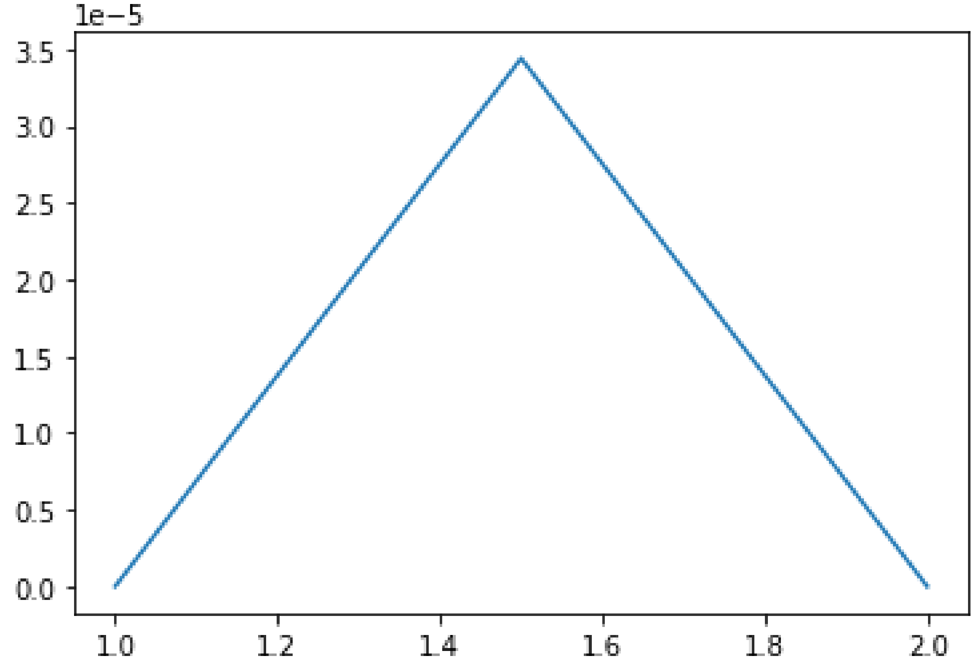
[81]:
```python
x_grid, w, test2 = nonlinear_shooting(1, 2, 0.5, alpha=0, beta=np.log(2),␣
↪tol=1e-8)
```

```
> solution converged
```

[84]:
```python
# plotting
y_dynamics = w[0, :]
plt.plot(x_grid, y_dynamics, x_grid, np.log(x_grid))
plt.figure(2)
plt.plot(x_grid, abs(y_dynamics-np.log(x_grid)))
```

[84]:
```
[<matplotlib.lines.Line2D at 0x7f8e4ab92fa0>]
```





2 Problem 2: Finite Difference Method for Nonlinear BVP

[128]:
```
def prob2_rhs(x, y, y_p):
    """ right hand side for y'' = f(x, y, y'). """
    return -(y_p**2) - y + np.log(x)

def J(w, h):
    """ Helper function to generate Jacobian matrix. """
    # scipy.sparse.diags
    N = len(w)-2 # exclude boundary
    # compute w_prime using centered difference
    print(w[1+1:(N+1)+1])
    print( w[1-1:(N+1)-1])
    w_prime = (1/(2*h)) * ( w[1+1:(N+1)+1] - w[1-1:(N+1)-1] )
    upp_diag = -1 - h*w_prime
    low_diag = -1 + h*w_prime
    diag = (2 - h**2) * np.ones(N)
    J = sp.sparse.diags()
    return J


def fdfbvp_solver(a, b, alpha, beta, h, tol=1e-8):
    """ finite difference BVP solver using Newton's Iteration.
        Generates an equally spaced grid with step size h
```

```python
    Input:
        a, b                        x_start, x_end
        alpha, beta                 boundary conditions
        h                           step size
        tol                         vector error tolerance default 1e-8
    Output:
        y_approx                    numerical solution on grid points
    """
    # number of grid points
    N = int(((b-a)/h))-1
    # initialize w
    w = np.zeros(N+2) # w0, w1, ..., wn, wn+1
    w[0] = alpha
    slope = (beta-alpha)/(b-a)
    w[1:N+2] = alpha + np.arange(1, N+2) * slope * h






    return w
```

[129]:
```python
fdfbvp_solver(a=1, b=2, alpha=0, beta=np.log(2), h=0.1, tol=1e-8)
```

[129]:
```
array([0.        , 0.06931472, 0.13862944, 0.20794415, 0.27725887,
       0.34657359, 0.41588831, 0.48520303, 0.55451774, 0.62383246,
       0.69314718])
```

[130]:
```python
test = np.arange(10)
J(test, 1)
```

```
[2 3 4 5 6 7 8 9]
[0 1 2 3 4 5 6 7]
```

[ ]:
```python
test[1:10]
```