TugasMetNum

September 3, 2024

**

Tugas Metode Numeris

**

0.1 Import Library

[ ]:
```
import numpy as np
```

0.2 Definisikan Fungsi Masalah

$$f(x) = \sin(x) - x^2 + 0.5$$

[ ]:
```
def f(x):
    return np.sin(x) - x**2 + 0.5

def g(x):
    return np.cos(x) - 2*x



def h(x):
    return np.arcsin(x**2-0.5)
```

0.3 Metode Bisection

[ ]:
```
def my_bisection(f, a, b, tol):
    """
    Find the root of a function f in the interval [a, b] using the bisection␣
 ↪method.

    Parameters
    ----------
    f : function
        The function to find the root of.
    a : float
        The lower bound of the interval.
    b : float
        The upper bound of the interval.
```

```python
    tol : float
        The desired accuracy of the root.

    Returns
    -------
    float
        The root of the function.

    Raises
    ------
    Exception
        If the scalars a and b do not bound a root.

    """
    print(f"{'Iterasi':>8} | {'xl':>20} | {'xu':>20} | {'xr':>20} | {'Error':
↪>25}")
    print("-" * 105)


    if np.sign(f(a)) == np.sign(f(b)):
        raise Exception("The scalars a and b do not bound a root")

    m_old = None  # m pertama kali belum ada nilainya
    iter_count = 1

    while True:
        m = (a + b) / 2

        if m_old is None:  # Pada iterasi pertama, set error ke infinity
            error = float('inf')
        else:
            error = np.abs(m - m_old) / np.abs(m)

        print(f"{iter_count:>8} | {a:>20} | {b:>20} | {m:>20} | {error:>25}")

        if error < tol and iter_count > 1:  # Abaikan error pada iterasi pertama
            return m

        m_old = m

        if np.sign(f(a)) == np.sign(f(m)):
            a = m
        else:
            b = m

        iter_count += 1
```

[ ]:
```
my_bisection(f, 0, 2, 0.001)
```

```
 Iterasi |                    xl |                    xu |                    xr |
Error
--------------------------------------------------------------------------------
-------------------------
       1 |                     0 |                     2 |                   1.0 |
inf
       2 |                   1.0 |                     2 |                   1.5 |
0.3333333333333333
       3 |                   1.0 |                   1.5 |                  1.25 |
0.2
       4 |                   1.0 |                  1.25 |                 1.125 |
0.1111111111111111
       5 |                 1.125 |                  1.25 |                1.1875 |
0.05263157894736842
       6 |                1.1875 |                  1.25 |               1.21875 |
0.02564102564102564
       7 |                1.1875 |               1.21875 |              1.203125 |
0.012987012987012988
       8 |                1.1875 |              1.203125 |             1.1953125 |
0.006535947712418301
       9 |             1.1953125 |              1.203125 |            1.19921875 |
0.003257328990228013
      10 |             1.1953125 |            1.19921875 |           1.197265625 |
0.0016313213703099511
      11 |             1.1953125 |           1.197265625 |          1.1962890625 |
0.0008163265306122449
```

[ ]: 1.1962890625

0.4 Metode Regula Falsi

[ ]:
```
def regular_falsi(f, a, b, tol):
    """
    Regular Falsi method to find the root of a non-linear equation.

    Parameters
    ----------
    f : function
        The function for which the root is to be found.
    a : float
        The lower limit of the bracket.
    b : float
        The upper limit of the bracket.
    tol : float
        The tolerance for the root.

```

```
    Returns
    -------
    float
        The root of the equation.

    Notes
    -----
    The regular falsi method is a modification of the bisection method.
    Instead of bisecting the interval, it uses the secant method to
    approximate the root. The method is more efficient than the bisection
    method but may not converge for all functions.
    """
    print(f"{'Iterasi':>8} | {'xl':>20} | {'xu':>20} | {'xr':>20} | {'Error':
↪>25}")
    print("-" * 105)
    step = 1
    condition = True
    m = m = b - (a-b) * f(b)/( f(a) - f(b) )
    while condition:
        m_old = m
        m = b - (a-b) * f(b)/( f(a) - f(b) )
        if step > 1 :
            error= abs(m - m_old) / abs(m)
        else:
            error = np.inf

        print(f"{step:>8} | {a:>20} | {b:>20} | {m:>20} | {error:>25}")

        if f(a) * f(m) < 0:
            b = m
        else:
            a = m

        step = step + 1
        condition = error > tol

    return m
```

[ ]:
```
regular_falsi(f,0, 2, 0.001)
```

```
 Iterasi |                   xl |                   xu |                   xr |
Error
--------------------------------------------------------------------------------
-------------------------
       1 |                    0 |                    2 |   0.3235510296847963 |
inf
       2 |   0.3235510296847963 |                    2 |   0.6854591643992791 |
0.5279791321072363
```

```
      3 |    0.6854591643992791 |                       2 |    0.9533763890509503 |
0.28101936205738487
      4 |    0.9533763890509503 |                       2 |    1.0953106519233242 |
0.12958356848182082
      5 |    1.0953106519233242 |                       2 |    1.1569338164329626 |
0.05326420892392424
      6 |    1.1569338164329626 |                       2 |    1.1812919900232455 |
0.020619943075888963
      7 |    1.1812919900232455 |                       2 |    1.1905548509227786 |
0.0077802890747567105
      8 |    1.1905548509227786 |                       2 |     1.194024950729844 |
0.002906220514859657
      9 |     1.194024950729844 |                       2 |    1.1953176137668615 |
0.0010814389599296142
     10 |    1.1953176137668615 |                       2 |     1.195798134717781 |
0.00040184119456998215
```

[ ]: 1.195798134717781

0.5 Metode Simple Fixed Point Iteration

[ ]:
```python
def fixedPointIteration(f, a, tol, N=100):
    """
    Find the root of an equation using the Fixed Point Iteration method.

    Parameters
    ----------
    f : function
        The function for which to find the root.
    a : float
        The initial guess for the root.
    tol : float
        The desired accuracy of the root.
    N : int, optional
        The maximum number of iterations. Default is 100.

    Returns
    -------
    float
        The root of the equation.

    Notes
    -----
    The function `f` should take a single argument, and should return a single
    value. The function should be continuous in the region of the root.
    """
```

```
    print(f"{'Iterasi':>8} | {'xi':>20} | {'error':>20}")
    print("-" * 55)

    step = 1
    flag = 1
    condition = True
    a_old=None
    while condition:
        if a_old is None:
            error = np.inf
        else:
            error= abs(a_old- a) / abs(a_old)
            a = a_old

        print(f"{step:>8} | {a:>20} | {error:>20}")
        a_old = f(a)

        step = step + 1

        if step > N:
            flag=0
            break

        condition = error > tol

    if flag==1:
        print('\nRequired root is: %0.8f' % a_old)
    else:
        print('\nNot Convergent.')

    return a_old
```

[ ]:
```
fixedPointIteration(h, 1, 0.001)
```

```
 Iterasi |                   xi |                error
-------------------------------------------------------
       1 |                    1 |                  inf
       2 |   0.5235987755982989 |   0.9098593171027438
       3 | -0.22780966438754494 |   3.2984045782516227
       4 | -0.46464196892049614 |   0.5097092393164231
       5 |    -0.28807579657455 |   0.6129156786007646
       6 |  -0.4301557132963552 |   0.3302988018757743
       7 |  -0.3204208961969517 |  0.34247085131412064
       8 | -0.40860597232660517 |  0.21581935189915866
       9 |  -0.3395270291477253 |   0.2034563885893874
      10 |  -0.3949059686326181 |    0.140233229891777
      11 | -0.35122605913631694 |  0.12436409076169445
      12 |  -0.3861667850138658 |  0.09048091973082119
```

```
13 | -0.35850557642576303 |   0.0771569827835877
14 |  -0.3805958535689498 | 0.058041297444626126
15 |  -0.3630710957394681 | 0.048268116176499745
16 | -0.37705008679011925 |   0.0370746262642618
17 | -0.36594644587130487 | 0.030342256480663524
18 | -0.37479653820398895 | 0.023613057834240923
19 | -0.36776154502171615 | 0.019129224568211405
20 |  -0.3733658439063824 | 0.015010207752349881
21 | -0.36890889148243333 | 0.012081444841405475
22 |  -0.3724582489805446 | 0.009529544607553184
23 |  -0.3696347258465744 | 0.007638684724503218
24 | -0.37188279307219213 | 0.0060450961095726286
25 |  -0.3700941277270992 | 0.004833001150485293
26 | -0.37151805251184433 | 0.0038327203082540034
27 | -0.37038498441717543 | 0.0030591631473717924
28 |  -0.3712869204549271 | 0.002429215757577904
29 |  -0.3705691663319765 | 0.001936896504518147
30 |  -0.3711404754009183 | 0.0015393337746972206
31 | -0.37068581131717193 | 0.0012265483864376064
32 | -0.37104769636248136 | 0.0009753060020507414

Required root is: -0.37075969
```

[ ]: -0.37075968969468565

0.6 Metode Newton Raphson

[ ]:
```python
def newtonRaphson(f,g,a,tol,N=100):
    """
    Finds the root of f(x) using Newton Raphson method

    Parameters
    ----------
    f : function
        The function to find the root of
    g : function
        The derivative of f
    a : float
        The initial guess
    tol : float
        The tolerance of the root
    N : int, optional
        The maximum number of iterations. Default is 100.

    Returns
    -------
    float
```

```
        The root of f(x)
    """

    print(f"{'Iterasi':>8} | {'xi':>20} | {'xi+1':>20} | {'error':>20}")
    print("-" * 80)
    step = 1
    flag = 1
    condition = True
    while condition:
        if g(a) == 0.0:
            print('Divide by zero error!')
            break

        b = a - f(a)/g(a)
        error= abs(b- a) / abs(b)
        print(f"{step:>8} | {a:>20} | {b:>20} | {error:>20}")
        a = b
        step = step + 1

        if step > N:
            flag = 0
            break

        condition = error > tol

    return b
```

[ ]:
```
newtonRaphson(f,g,0,0.001)
```

```
 Iterasi |                   xi |                 xi+1 |                error
--------------------------------------------------------------------------------
       1 |                    0 |                 -0.5 |                  1.0
       2 |                 -0.5 |    -0.37780801587057 |   0.32342348228866247
       3 |    -0.37780801587057 |   -0.3709105514033993 | 0.018596031957228105
       4 |   -0.3709105514033993 | -0.37088734037553595 | 6.258242149716885e-05
```

[ ]: -0.37088734037553595

[ ]:
```
newtonRaphson(f,g,2,0.001)
```

```
 Iterasi |                   xi |                 xi+1 |                error
--------------------------------------------------------------------------------
       1 |                    2 |   1.4133567861163074 |   0.41507085800726945
       2 |   1.4133567861163074 |   1.2223605259485244 |     0.156251986311137
       3 |   1.2223605259485244 |   1.1965641529553526 | 0.021558704503605815
       4 |   1.1965641529553526 |    1.196082201285628 | 0.0004029419292474218
```

[ ]: 1.196082201285628

0.7 Metode Secant

[ ]:
```python
def secant(f,a,b,e,N=100):
    """
    Secant method for finding roots of a function.

    Parameters
    ----------
    f : function
        The function to find the root of.
    a : float
        The lower bound of the initial interval.
    b : float
        The upper bound of the initial interval.
    e : float
        The desired accuracy of the root.
    N : int, optional
        The maximum number of iterations. Default is 100.

    Returns
    -------
    float
        The root of the function.

    Notes
    -----
    The secant method is a root-finding algorithm that uses the slope of the
    function at two points to approximate the root. The algorithm starts with
    an interval [a, b] containing the root, and uses the slope of the function
    at a and b to approximate the root. The algorithm iterates until the
    desired accuracy is reached or the maximum number of iterations is reached.
    """
    print(f"{'Iterasi':>8} | {'xi-1':>20} | {'xi':>20} | {'xi+1':>20} |␣
↪{'error':>20}")
    print("-" * 105)
    step = 1
    condition = True
    while condition:
        if f(a) == f(b):
            print('Divide by zero error!')
            break

        m = a - (b-a)*f(a)/( f(b) - f(a) )
        error= abs(m- a) / abs(m)
        print(f"{step:>8} | {a:>20} | {m:>20} | {b:>20} | {error:>20}")
        a = b
        b = m
```

```
        step = step + 1

        if step > N:
            print('Not Convergent!')
            break

        condition = error > e
    return m
```

[ ]:
```
secant(f,-2, 0, 0.001)
```

```
 Iterasi |                    xi-1 |                       xi |                     xi+1 |
error
--------------------------------------------------------------------------------
-------------------------
       1 |                      -2 | -0.20369513456971244 |                        0 |
8.81859485365136
       2 |                       0 | -0.41778277958994275 | -0.20369513456971244 |
1.0
       3 | -0.20369513456971244 |  -0.3667082223143116 | -0.41778277958994275 |
0.4445307681289949
       4 | -0.41778277958994275 | -0.37079417270488946 |  -0.3667082223143116 |
0.12672423232080013
       5 |  -0.3667082223143116 |  -0.3708875311315772 | -0.37079417270488946 |
0.011268399356846853
       6 | -0.37079417270488946 | -0.37088734010328567 |  -0.3708875311315772 |
0.00025120134424179734
```

[ ]: -0.37088734010328567

[ ]:
```
secant(f,0, 2, 0.001)
```

```
 Iterasi |                    xi-1 |                       xi |                     xi+1 |
error
--------------------------------------------------------------------------------
-------------------------
       1 |                       0 |  0.32355102968479627 |                        2 |
1.0
       2 |                       2 |   0.6854591643992791 |  0.32355102968479627 |
1.9177522219762788
       3 |  0.32355102968479627 |   5.4783563382325795 |   0.6854591643992791 |
0.94094012698174
       4 |   0.6854591643992791 |   0.7883364888280808 |   5.4783563382325795 |
0.13049925493330428
       5 |   5.4783563382325795 |   0.8777682011730086 |   0.7883364888280808 |
5.241233540827247
       6 |   0.7883364888280808 |   1.3797644021717623 |   0.8777682011730086 |
0.4286441311377278
```

```
       7 |    0.8777682011730086 |    1.1497278951292276 |    1.3797644021717623 |
0.23654265944869604
       8 |    1.3797644021717623 |    1.1904565469923398 |    1.1497278951292276 |
0.15902122228459686
       9 |    1.1497278951292276 |    1.1962775308437756 |    1.1904565469923398 |
0.038912070580908606
      10 |    1.1904565469923398 |    1.1960812349965801 |    1.1962775308437756 |
0.004702596980594216
      11 |    1.1962775308437756 |    1.1960820331842839 |    1.1960812349965801 |
0.0001634483706534053
```

[ ]:
```
1.1960820331842839
```

0.8 Modified Secant Method

[ ]:
```python
def mod_secant(f,a,delta,e,N=100):
    """
    Modified Secant method for finding roots of a function.

    Parameters
    ----------
    f : function
        The function to find the root of.
    a : float
        The lower bound of the initial interval.
    delta : float
        The upper bound of the initial interval.
    e : float
        The desired accuracy of the root.
    N : int, optional
        The maximum number of iterations. Default is 100.

    Returns
    -------
    float
        The root of the function.

    Notes
    ------
    The secant method is a root-finding algorithm that uses the slope of the
    function at two points to approximate the root. The algorithm starts with
    an interval [a, b] containing the root, and uses the slope of the function
    at a and b to approximate the root. The algorithm iterates until the
    desired accuracy is reached or the maximum number of iterations is reached.

    """
    print(f"{'Iterasi':>8} | {'xi-1':>20} | {'xi+1':>20} | {'error':>20}")
    print("-" * 80)
```

```
    step = 1
    condition = True
    while condition:
        if f(a+delta) == f(a):
            print('Divide by zero error!')
            break

        m = a - delta*f(a)/(f(a+delta) - f(a))
        error= abs(m- a) / abs(m)
        print(f"{step:>8} | {a:>20} | {m:>20} | {error:>20}")
        a = m
        step = step + 1

        if step > N:
            print('Not Convergent!')
            break

        condition = error > e

    return m
```

[ ]:
```
mod_secant(f,0, 0.01, 0.001)
```

```
 Iterasi |                 xi-1 |                 xi+1 |                error
--------------------------------------------------------------------------------
       1 |                    0 |  -0.5050590076848889 |                  1.0
       2 |  -0.5050590076848889 |  -0.3778034957192951 |   0.33682989545481484
       3 |  -0.3778034957192951 |  -0.3708769337808227 | 0.018676173435379514
       4 |  -0.3708769337808227 |  -0.3708873914195287 | 2.8196263739165807e-05
```

[ ]: -0.3708873914195287

[ ]:
```
mod_secant(f,2, 0.01, 0.001)
```

```
 Iterasi |                 xi-1 |                 xi+1 |                error
--------------------------------------------------------------------------------
       1 |                    2 |   1.4152818844542523 |   0.4131460467122571
       2 |   1.4152818844542523 |   1.2238422576726502 |  0.15642508303779115
       3 |   1.2238422576726502 |    1.196807659702185 | 0.022588924587257794
       4 |    1.196807659702185 |   1.1960876181677065 | 0.0006019973148636158
```

[ ]: 1.1960876181677065

0.9 Modified Newton-Raphson Method

[ ]:
```
def mod_newtonRaphson(f,g,a,tol,N=100):
    """
    Finds the root of f(x) using Newton Raphson method
```

```python
    Parameters
    ----------
    f : function
        The function to find the root of
    g : function
        The derivative of f
    a : float
        The initial guess
    tol : float
        The tolerance of the root
    N : int, optional
        The maximum number of iterations

    Returns
    -------
    float
        The root of f(x)

    Notes
    ------
    The function `f` and `g` should take a single argument, and should return a␣
↪single value.
    The function `f` should be continuous in the region of the root.
    """

    print(f"{'Iterasi':>8} | {'xi':>20} | {'xi+1':>20} | {'error':>20}")
    print("-" * 80)
    step = 1
    flag = 1
    condition = True
    while condition:
        if g(a) == 0.0:
            print('Divide by zero error!')
            break

        b = a - f(a)*g(a)/(g(a)**2-f(a)*g(a))
        error= abs(b- a) / abs(b)
        print(f"{step:>8} | {a:>20} | {b:>20} | {error:>20}")
        a = b
        step = step + 1

        if step > N:
            flag = 0
            break

        condition = error > tol
```

```
    return b
```

[ ]:
```
mod_newtonRaphson(f,g,0, 0.001)
```

```
 Iterasi |                   xi |                 xi+1 |                   error
--------------------------------------------------------------------------------
       1 |                    0 |                 -1.0 |                     1.0
       2 |                 -1.0 |   -0.654417998075753 |    0.5280753324945133
       3 |   -0.654417998075753 |  -0.4509621248977419 |   0.45115955851978873
       4 |  -0.4509621248977419 |  -0.3792528573724579 |   0.18908036190445765
       5 |  -0.3792528573724579 | -0.37099003200763886 |   0.02227236489375246
       6 | -0.37099003200763886 |  -0.3708873558134567 | 0.00027683929520040337
```

[ ]: -0.3708873558134567

[ ]:
```
mod_newtonRaphson(f,g,2, 0.001)
```

```
 Iterasi |                   xi |                 xi+1 |                   error
--------------------------------------------------------------------------------
       1 |                    2 |   0.5807824291564252 |    2.4436303503619414
       2 |   0.5807824291564252 |    1.266836188597014 |    0.5415489118607942
       3 |    1.266836188597014 |   1.1945040818612478 |   0.06055408921086324
       4 |   1.1945040818612478 |    1.196081346190124 | 0.001318693192482551
       5 |    1.196081346190124 |   1.1960820332970041 | 5.744646779498571e-07
```

[ ]: 1.1960820332970041

0.10 Brent's Method

[ ]:
```
def brents(f, a, b, e=1e-5, max_iter=50):

    """
    Finds the root of a function f in the interval [a, b] using Brent's method.

    Parameters
    ----------
    f : function
        The function to find the root of.
    a : float
        The lower bound of the interval.
    b : float
        The upper bound of the interval.
    e : float, optional
        The desired accuracy of the root. Defaults to 1e-5.
    max_iter : int, optional
        The maximum number of iterations. Defaults to 50.
```

```
    Returns
    -------
    float
        The root of the function.
    int
        The number of iterations taken.

    Notes
    -----
    The function `f` should take a single argument, and should return a single␣
↪value.
    The function should be continuous in the region of the root.
    """
    print(f"{'Iterasi':>8} | {'xl':>20} | {'xu':>20} | {'xr':>20} | {'Error':
↪>25}")
    print("-" * 105)
    fa = f(a)
    fb = f(b)
    error = np.inf

    assert (fa * fb) <= 0, "Root not bracketed"

    if abs(fa) < abs(fb):
        a, b = b, a
        fa, fb = fb, fa

    x2, fx2 = a, fa
    mflag = True
    steps_taken = 0

    while steps_taken < max_iter and abs(b - a) > e:
        fa = f(a)
        fb = f(b)
        fx2 = f(x2)

        if fa != fx2 and fb != fx2:
            L0 = (a * fb * fx2) / ((fa - fb) * (fa - fx2))
            L1 = (b * fa * fx2) / ((fb - fa) * (fb - fx2))
            L2 = (x2 * fb * fa) / ((fx2 - fa) * (fx2 - fb))
            new = L0 + L1 + L2
        else:
            new = b - ((fb * (b - a)) / (fb - fa))

        if ((new < ((3 * a + b) / 4) or new > b) or
            (mflag == True and (abs(new - b)) >= (abs(b - x2) / 2)) or
            (mflag == False and (abs(new - b)) >= (abs(x2 - d) / 2)) or
            (mflag == True and (abs(b - x2)) < e) or
```

```
            (mflag == False and (abs(x2 - d)) < e)):
            new = (a + b) / 2
            mflag = True
        else:
            mflag = False

        fnew = f(new)
        d, x2 = x2, b


        if (fa * fnew) < 0:
            b = new
        else:
            a = new

        if abs(fa) < abs(fb):
            a, b = b, a

        # Hitung error
        error = abs(b - a)

        # Tampilkan iterasi
        print(f"{steps_taken+1:>8} | {a:>20} | {b:>20} | {new:>20} | {error:
↪>25}")

        steps_taken += 1

    return b
```

[ ]:
```
brents(f, -2, 0, 0.001, 100)
```

```
 Iterasi |                   xl |                   xu |                   xr |
Error
--------------------------------------------------------------------------------
-------------------------
       1 |                   -2 |   -0.2036951345697124 |   -0.2036951345697124 |
1.7963048654302876
       2 |   -0.4060256049169877 |   -0.2036951345697124 |   -0.4060256049169877 |
0.2023304703472753
       3 | -0.30486036974335007 |   -0.4060256049169877 | -0.30486036974335007 |
0.10116523517363762
       4 |   -0.3554429873301689 |   -0.4060256049169877 |   -0.3554429873301689 |
0.05058261758681881
       5 |   -0.3807342961235783 |   -0.3554429873301689 |   -0.3807342961235783 |
0.025291308793409406
       6 |    -0.370884369625688 |   -0.3807342961235783 |    -0.370884369625688 |
0.009849926497890293
       7 | -0.37580933287463314 |    -0.370884369625688 | -0.37580933287463314 |
```

```
0.004924963248945147
       8 | -0.37580933287463314 |  -0.3708873400308677 |  -0.3708873400308677 |
0.0049219928437654326
       9 | -0.37580933287463314 |   -0.370887340111992 |   -0.370887340111992 |
0.004921992762641159
      10 | -0.37334833649331256 |   -0.370887340111992 | -0.37334833649331256 |
0.0024609963813205793
      11 | -0.37211783830265227 |   -0.370887340111992 | -0.37211783830265227 |
0.0012304981906602896
      12 | -0.37150258920732215 |   -0.370887340111992 | -0.37150258920732215 |
0.0006152490953301726
```

[ ]: -0.370887340111992

[ ]:
```
brents(f, 0, 2, 0.001, 100)
```

```
 Iterasi |                   xl |                   xu |                   xr |
Error
--------------------------------------------------------------------------------
-------------------------
       1 |                    2 |                  1.0 |                  1.0 |
1.0
       2 |                  1.5 |                  1.0 |                  1.5 |
0.5
       3 |                 1.25 |                  1.0 |                 1.25 |
0.25
       4 |                1.125 |                 1.25 |                1.125 |
0.125
       5 |                1.125 |   1.1970480507412633 |   1.1970480507412633 |
0.0720480507412633
       6 |     1.19607820360712 |   1.1970480507412633 |     1.19607820360712 |
0.0009698471341432757
```

[ ]: 1.1970480507412633