Reševanje algebraičnih enačb

Datum: 07/10/2024

Avtor: Aleksander Grm

V zapiskih so uporabljeni primeri iz OnLine knjige Numerične metode v ekosistemu Pythona, Janko Slavič

Najprej naložimo celoten potreben Python ekosistem

In […
```python
import numpy as np              # orodja za numeriko
import matplotlib.pyplot as mpl # izdelava grafov
import numpy.polynomial as poly # paket za podporo polinomov
import scipy.optimize as opt    # uporaba fsolve() funkcije

from IPython.display import YouTubeVideo
```

Uvodoma si doma oglejte video predavanja o numeričnem reševanju enačb.

In […
```python
YouTubeVideo('17d55KE8SIU', width=800, height=300)
```

Uvod

V okviru reševanja enačb obravnavamo splošno funkcijo

$$y = f(x),$$

kjer iščemo njeno partikularno rešitev za enačbo $y = 0$. Rešitvam enačbe $f(x) = 0$ pravimo **koreni** enačbe (angl. *roots*). Koren enačbe $f(x) = 0$ je hkrati tudi ničla splošne funkcije $y = f(x)$.

Funkcija $y = f(x)$ ima lahko ničle stopnje:

- ničla prve stopnje: funkcija seka abscisno os pod neničelnim kotom,
- ničle sode stopnje: funkcija se dotika abscisne osi, vendar je ne seka,
- ničle lihe stopnje: funkcija seka abscisno os, pri ničli stopnje 3 in več imamo prevoj (tangenta je vzporedna z abscisno osjo).

Omejitveni kriteriji za $y = f(x)$

Dana funkcija $f(x)$ mora biti zvezna v okolici, kjer iščemo njen koren. Pravim, da je funkcija f zvezna na zaprtem intervali $[a, b]$, kjer je a spodnja meja in b zgornja meja intervala.

Velikokrat rešujemo neačbo ki je polinomske oblike

$$p_n(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$$

V numpy paketu imamo sedaj polynomial paket, ki vsebuje različne oblike predstavitve polinomov. Znotraj imamo tako razred Polynomial, ki predstavlja naš zgoraj zapisan matematičen polinom. Poglejmo si kako uporabimo funkcijo paketa in pa kako lahko to naredimo tudi sami.

Primer polinoma

Kot primer vzemimo polinom stopnje $n = 3$

$$p_3(x) = 5 - 10x^2 + x^3,$$

kjer imamo koeficiente $a_0 = 5$, $a_1 = 0$, $a_2 = -10$ in $a_3 = 1$.

Spodaj izvedemo inmplementacijo naše funkcije in funkcije, ki jo dobimo v numpy paketu.

In […
```python
# naša funkcija za polinom
def our_poly(x):
    return 5 - 10*x**2 + x**3

# polinom iz scipy paketa
sp_poly = poly.Polynomial([5,0,-10,1])
```

In […
```python
# Testirajmo kakšno vrednost dobimo za obe funkciji

xx = 2.5

print('Vrednost našega polinoma v x={:2f} je: {:.5f}'.format(xx,our_poly(xx)))
print(' Vrednost scipy polinoma v x={:2f} je: {:.5f}'.format(xx,sp_poly(xx)))
```

In […
```python
# izrišimo graf te funkcije na intervalu [-2,2]
xp = np.linspace(-2,2,100)

mpl.plot(xp, sp_poly(xp))
mpl.grid()
```

Funkcija ima tako na intervalu $[-2, 2]$ dve ničli, kar pomeni, da je potrebno poiskati rešitev enačbe $p_3(x) = 0$! Z uporabo Python paketa je to hitro rešeno

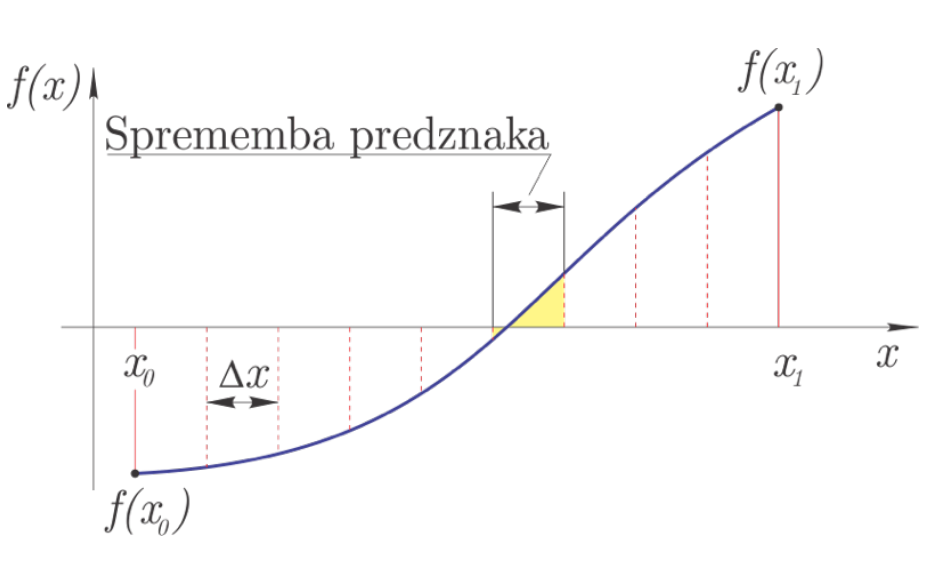
In […
```python
pr = sp_poly.roots()
print('koreni so:', pr)
```

In […
```python
mpl.axhline(0, color='g', lw=1)
mpl.plot(xp, sp_poly(xp))
mpl.plot(pr[0:2],sp_poly(pr[0:2]), 'ro')
mpl.grid()
```

Inkrementalna metoda

Inkrementalno reševanje temelji na ideji, da v kolikor ima funkcija $f(x)$ pri x_0 in x_1 različna predznaka, potem je vmes vsaj ena ničla. Zaprti interval $[x_0, x_1]$ razdelimo torej na odseke širine Δx; na odseku, kjer opazimo spremembo predznaka, je vsaj ena ničla funkcije.

Delovanje metode je prikazana na spodnji sliki.



Za ničlo zahtevamo:

$$|x_{i+1} - x_i| < \varepsilon \quad \text{in} \quad |f(x_{i+1})| + |f(x_i)| < D,$$

kjer je ϵ zahtevana natančnost rešitve in D izbrana majhna vrednost, ki prepreči, da bi kot ničlo razpoznali pol (kar sicer zaradi pogoja zveznosti ni mogoče).

Inkrementalna metoda ima nekatere slabosti:

- je zelo počasna,
- lahko zgreši dve ničli, ki sta si zelo blizu,
- večkratne sode ničle (lokalni ekstrem, ki se samo dotika abscise) ne zazna.

Inkrementalna metoda spada med t. i. *zaprte* (angl. *bracketed*) metode, saj išče ničle funkcije samo na intervalu $[x_0, x_1]$. Pozneje bomo spoznali tudi *odprte* metode, ki lahko konvergirajo k ničli zunaj podanega intervala.

Zaradi vseh zgoraj navedenih slabosti inkrementalno metodo pogosto uporabimo samo za izračun začetnega približka ničle.

Implementacija metode

In […
```python
def inkrementalna(fun, x0, x1, dx):
    """ Vrne prvi interval (x1, x2) kjer leži ničla

    :param fun: funkcija katere ničle iščemo
    :param x1:  spodnja meja iskanja
    :param x2:  zgornja meja iskanja
    :param dx:  inkrement iskanja
    """
    x_d = np.arange(x0, x1, dx)   # pripravimo x vrednosti
    f_d = np.sign(fun(x_d))       # pripravimo predznake funkcije
    f_d = f_d[1:]*f_d[:-1]        # pomnožimo sosednje elemente
    i = np.argmin(f_d)            # prvi prehod skozi ničlo

    # rezultat
    x0 = x_d[i]    # začetek iskanega segmenta
    x1 = x_d[i+1]  # konec iskanega segmenta
    D = np.abs(fun(x0)) + np.abs(fun(x1)) # vsota abs[f(x)]

    return [np.array([x0, x1]), D]
```

In […
```python
# Testiramo na primeru našega polinoma

rez_inkr, D = inkrementalna(our_poly, 0.5, 1., 0.001)
print('rezultat:', rez_inkr)
print('D:', D)
```

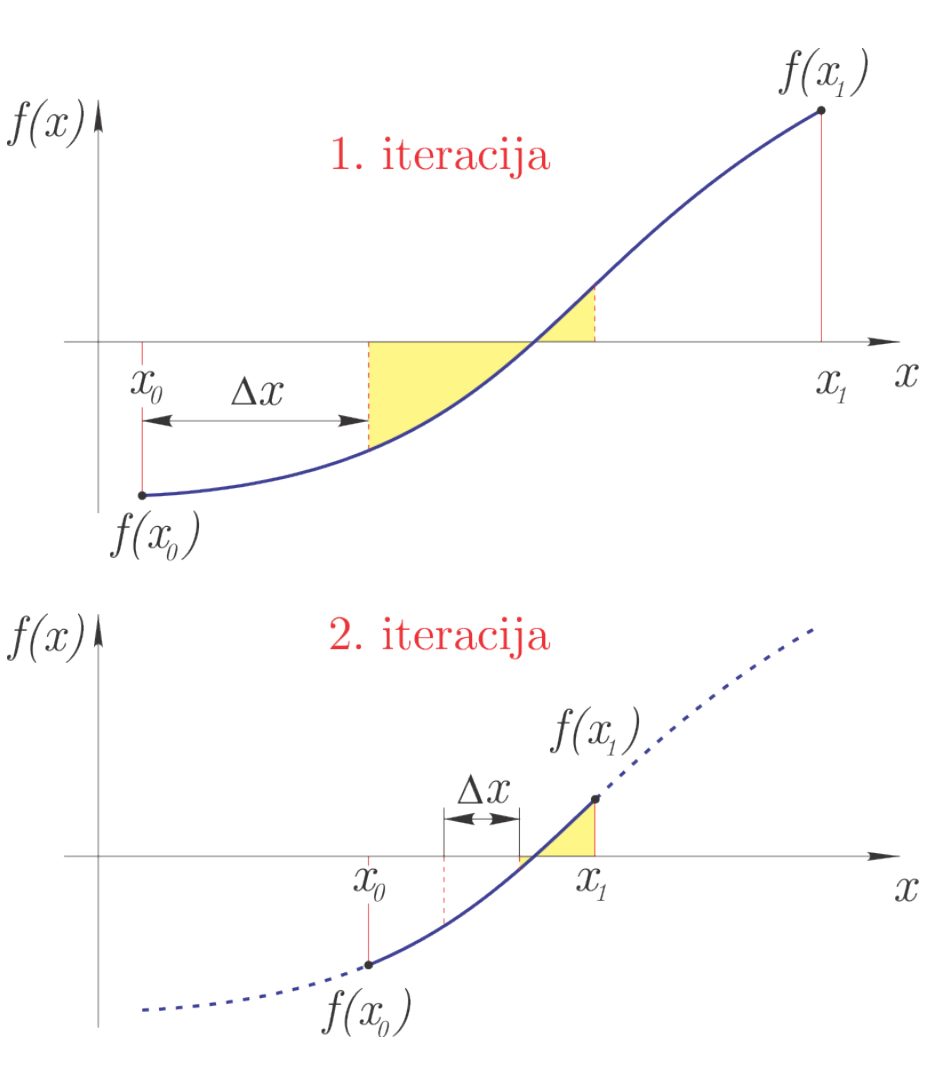
In […
```python
xp = np.linspace(rez_inkr[0]*0.9, rez_inkr[1]*1.1,100)

mpl.plot(xp, our_poly(xp), label='$f(x)$')
mpl.axhline(0, color='r', lw=0.5)     # horizontalna črta
mpl.axvline(rez_inkr[0], color='r', lw=0.5)     # vertikalna črta
mpl.axvline(rez_inkr[1], color='r', lw=0.5)     # vetrikalna črta
mpl.plot(rez_inkr, our_poly(rez_inkr), 'ro', label='Inkrementalna metoda')
mpl.xlim(0.7335,0.7355)
mpl.ylim(-0.025, 0.025)
mpl.legend()
mpl.grid()
```

V primeru izračuna, je bilo potrebno 1000-krat izračunati vrednost $f(x)$, kjer je bil rezultat natančen $D = 0.01307$ namesto $D = 0$. Vidimo, da je inkrementalna metoda iskanja ničle zelo neučinkovita metoda, zato bomo iskali boljše načine. Ena od izboljšavje obstoječe metode je uporaba iterativnega procesa in s tem nadgradnja obstoječe metode na iterative inkrementalne metode.

Iterativna inkrementalna metoda

Iterativna inkrementalna metoda v prvi iteraciji z inkrementalno metodo omeji interval iskanja ničel pri relativno velikem koraku. Interval, najden v prvi iteraciji, se v drugi iteraciji razdeli na manjše intervale in ponovi se inkrementalno iskanje ničle. Tretja iteracija se nato omeji na interval določen v drugi in tako dalje. Z iteracijami zaključimo, ko smo dosegli predpisano natančnost rešitve ϵ.

Metoda je prikazana na spodnji sliki



Implementacija metode

In […
```python
def inkrementalna_super(fun, x0, x1, iteracij=3):
    """ Vrne interval (x0, x1) kjer leži ničla

    :param fun: funkcija katere ničlo iščemo
    :param x0:  spodnja meja iskanja
    :param x1:  zgornja meja iskanja
    :iteraci:   število iteracij inkrementalne metode
    """
    N = 10

    for i in range(iteracij):
        dx = (x1 - x0)/N  # delitev intervala na N segmentov
        [x0x1_new, D] = inkrementalna(fun, x0, x1, dx)  # poiščemo presečni segment
        [x0, x1] = x0x1_new # posodobimo nov interval z izračunanim presečnim segmentom

    return [np.array([x0, x1]), D]
```

In […
```python
# Testiranje za različna števila iteracij

iter = [3,5,8,10]

for it in iter:
    [rez, D] = inkrementalna_super(our_poly, 0., 1., it)
    print('  število iteracij: {:d}'.format(it))
    print('   segment rešitve:')
    print('                x0: {:.10g}'.format(rez[0]))
    print('                x1: {:.10g}'.format(rez[1]))
    print('natančnost rešitve: {:.2g}'.format(D))
    print(' -----------------')
```

V primeru iterativne metode smo z bistveno manj dela dobili bistveno boljši rezultat.

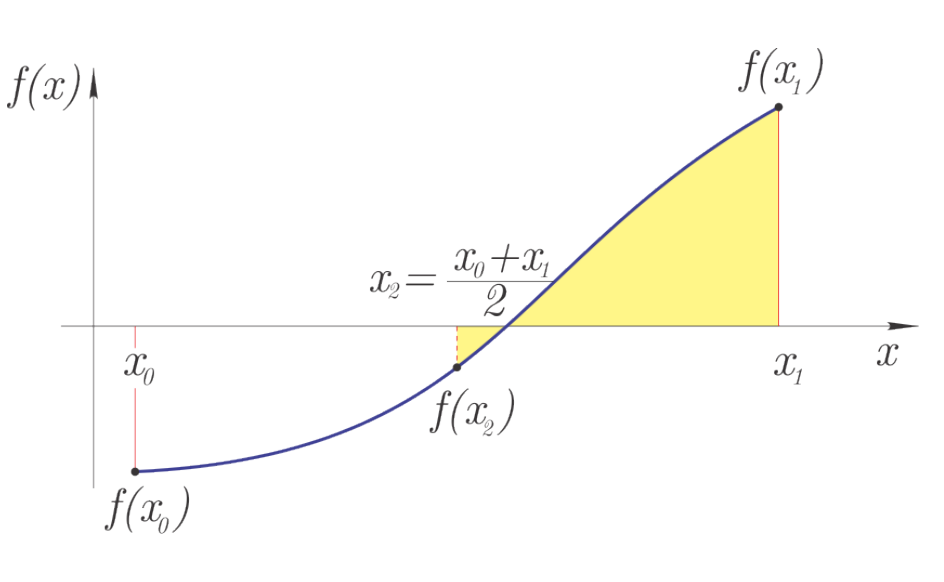
Bisekcijska metoda

Na intervalu $[x_0, x_1]$, kjer vemo, da obstaja ničla funkcije (predznaka $f(x_0)$ in $f(x_1)$ se razlikujeta), lahko uporabimo *bisekcijsko metodo*.

Ideja metode je:

- interval $[x_0, x_1]$ razdelimo na pol (od tukaj ime: *bi-sekcija*): $x_2 = (x_0 + x_1)/2$,
- če imata $f(x_0)$ in $f(x_2)$ različne predznake, je nov interval iskanja ničle $[x_0, x_2]$, sicer pa: $[x_2, x_1]$,
- glede na predhodni korak definiramo nov zaprt interval $[x_0, x_1]$ in nadaljujemo z iterativnim postopkom, dokler ne dosežemo želene natančnosti $|x_1 - x_0| < \varepsilon$.

Na spodnji sliki je prikazana slika metode



Bisekcijska metoda spada med *zaprte* metode, ki vrne ničlo funkcije na podanem intervalu $[x_0, x_1]$.

Implementacija metode

In […
```python
def bisekcija(fun, x0, x1, tol=1e-3, Dtol=1e-1, izpis=True):
    """ Vrne ničlo z natančnostjo tol

    :param fun: funkcija katere ničlo iščemo
    :param x0:  spodnja meja iskanja
    :param x1:  zgornja meja iskanja
    :param tol: zahtevana natančnost
    :param Dtol:največja vsota absolutnih vrednosti rešitve
    :izpis:     ali na koncu izpiše kratko poročilo
    """

    if np.sign(fun(x0))==np.sign(fun(x1)):
        raise Exception('Napaka (ERROR): Ničla ni izolirana!')

    n = np.ceil( np.log(np.abs(x1-x0)/tol)/np.log(2) ).astype(int) # število iteracij

    for i in range(n):
        x2 = (x0 + x1) / 2
        f1 = fun(x0)
        f3 = fun(x2)
        f2 = fun(x1)
        if np.sign(fun(x2)) != np.sign(fun(x0)):
            x1 = x2
        else:
            x0 = x2

    D = np.abs(fun(x0)) + np.abs(fun(x1))

    if D > Dtol:
        raise Exception('Opozorilo (WARNING): Verjetnost pola ali več ničel!')

    r = (x0+x1)/2

    if izpis:
        print('Rešitev: {:.5g}, število iteracij: {:d}, D: {:.5g}'
              .format(r,n,D))

    return r
```

In […
```python
# Testiranje metode na naši funkciji
eps = 1e-8
bisekcija(our_poly, 0., 1., eps);
```

Poizkusi različne vrednosti za ε in primerjaj rezultat. Izriši graf primerjave iteracije/napaka.

Kaj pa napaka metode?

Napako lahko ocenimo. Recimo, če v začetku pričnemo z intervalom $\Delta x = |x_1 - x_0|$, potem je natančnost bisekcijske metode po prvem koraku:

$$\varepsilon_1 = \Delta x/2,$$

po drugem koraku:

$$\varepsilon_2 = \Delta x/2^2,$$

in seveda po poljubno n korakih:

$$\varepsilon_n = \Delta x/2^n.$$

Ponavadi zahtevamo, da je rešitev podana z določeno natančnostjo, ki jo podamo z ε. Iz zgornje enačbe lahko izpeljemo število potrebnih korakov bisekcijske metode:

$$n = \frac{\log\left(\frac{\Delta x}{\varepsilon}\right)}{\log(2)}.$$

Seveda je število korakov n **celo število**.

Primerjava z metodo scipy.optimize.bisect

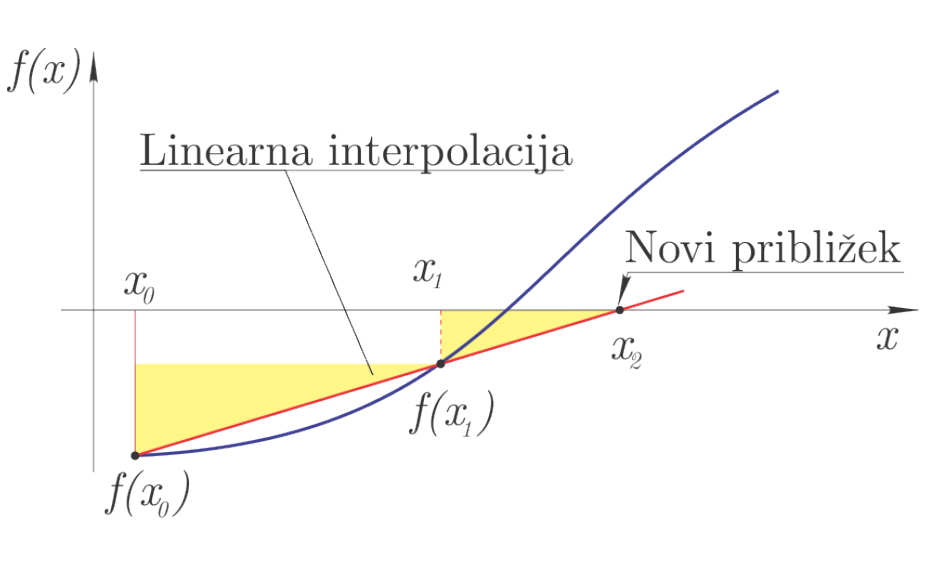
V paketu **scipy.optimize** imamo tudi metode iskanja ničle, med njimi je tudi bisekcijska metoda, ki se nahaja na povezavi scipy.optimize.bisect

In […
```python
# uporaba funkcije bisect

root, r = opt.bisect(our_poly, 0.0, 1.0, full_output=True)

print('Rešitev:', root)
print('Izpis postopeka:\n', r)
```

Sekantna metoda

Sekantna metoda zahteva dva začetna približka x_0 in x_1 in funkcijo $f(x)$. Ob predpostavki linearne interpolacije med točkama $x_0, f(x_0)$ in $x_1, f(x_1)$ (skozi točki potegnemo *sekanto*, od tukaj tudi ime), se določi x_2, kjer ima linearna interpolacijska funkcija ničlo. x_2 predstavlja nov približek ničle.

Delovanje metode je prikazano na spodnji sliki



Zapišimo razmerje stranic (podobna trikotnika sta na sliki označena z rumeno):

$$\frac{f(x_1)}{x_2 - x_1} = \frac{f(x_0) - f(x_1)}{x_1 - x_0}.$$

Od tod sledi, da je nov približek za ničlo funkcije:

$$x_2 = x_1 - f(x_1)\,\frac{x_1 - x_0}{f(x_1) - f(x_0)}.$$

V naslednjem koraku pri sekantni metodi izvedemo sledeče zamenjave: $x_0 = x_1$ in $x_1 = x_2$.

Sekantna metoda spada med **odprte** metode, saj lahko najde ničlo funkcije, ki se nahaja zunaj območja $[x_0, x_1]$.

Implementacija metode

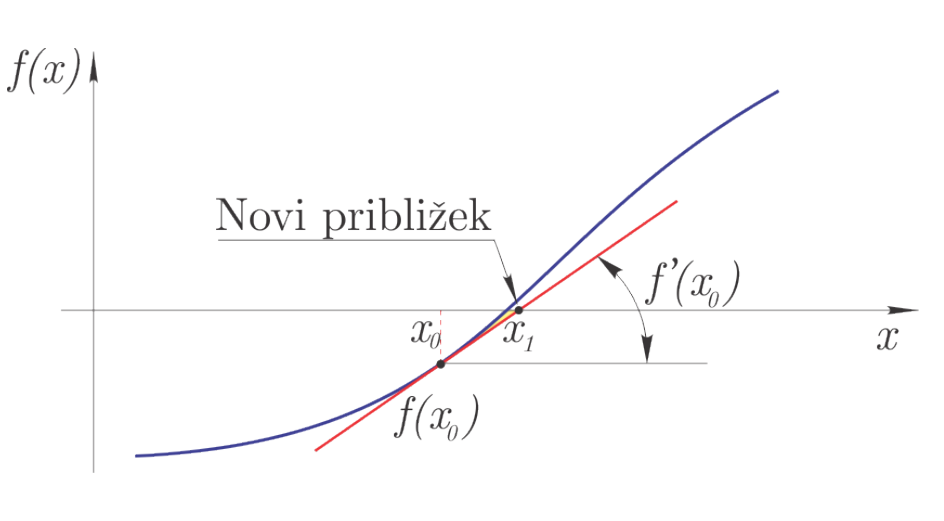
In […
```python
def sekantna(fun, x0, x1, tol=1e-3, Dtol=1e-1, max_iter=50, izpis=True):
    """ Vrne ničlo z natančnostjo tol

    :param fun: funkcija katere ničlo iščemo
    :param x0:  spodnja meja iskanja
    :param x1:  zgornja meja iskanja
    :param tol: zahtevana natančnost
    :max_iter:  maksimalno število iteracij preden se izvajanje prekine
    :param Dtol:največja vsota absolutnih vrednosti rešitve
    :izpis:     ali na koncu izpiše kratko poročilo
    """

    if np.sign(fun(x0))==np.sign(fun(x1)):
        raise Exception('Napaka: Ničla ni izolirana!')

    for i in range(max_iter):
        f0 = fun(x0)
        f1 = fun(x1)
        x2 = x1 - f1 * (x1 - x0)/(f1 - f0)
        x0 = x1
        x1 = x2
        if izpis:
            print('{:g}. korak: x0={:g}, x1={:g}.'.format(i+1, x0, x1))
        if np.abs(x1-x0)<tol:
            r = (x0+x1)/2
            D = np.abs(fun(x0)) + np.abs(fun(x1))
            if D > Dtol:
                raise Exception('Opozorilo: Verjetnost pola ali več ničel!')
            r = (x0+x1)/2
            if izpis:
                print('Rešitev: {:.8g}, D: {:.5g}'.format(r,D))
            return r
```

In […
```python
# Testiramo metodo

sekantna(our_poly, 0, 1., tol=1.e-8, izpis=True);
```

Newton-Raphson metoda

V literaturi najdemo za to metodo dve imeni, **tangentna** in **Newton-Raphsonova** metoda. Potrebuje en začetni približek x_0, poleg definicije funkcije $f(x)$ pa tudi njen odvod $f'(x)$.

Delovanja metode je prikazano na spodnji sliki



Metodo bi lahko izpeljali grafično (s slike), tukaj pa si poglejmo izpeljavo s pomočjo Taylorjeve vrste:

$$f(x_{i+1}) = f(x_i) + f'(x_i)\ (x_{i+1} - x_i) + O^2\ (x_{i+1} - x_i)\,,$$

če naj bo pri x_{i+1} vrednost funkcije nič, potem velja:

$$0 = f(x_i) + f'(x_i)\ (x_{i+1} - x_i) + O^2\ (x_{i+1} - x_i)\,.$$

Naredimo napako metode in zanemarimo člene višjega reda v Taylorjevi vrsti. Lahko izpeljemo:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}.$$

x_{i+1} je tako nov približek iskane ničle.

Algoritem Newtonove metode je:

1. izračunamo nov približek x_{i+1},
2. računanje prekinemo, če je največje število iteracij doseženo (rešitve enačbe nismo našli),
3. če velja $|x_{i+1} - x_i| < \varepsilon$ računanje prekinemo (izračunali smo približek ničle), sicer povečamo indeks i in gremo v prvi korak.

Opombi:

- ε je zahtevana absolutna natančnost,
- *Newtonova* metoda lahko divergira, zato v algoritmu predpišemo največje število iteracij.

Zgoraj smo omenili, da je Newtonova metoda ena izmed boljših metod za iskanje ničel funkcij. Ima pa tudi nekaj slabosti/omejitev:

- spada med *odprte* metode,
- kvadratična konvergenca je zagotovljena le v dovolj majhni okolici rešitve enačbe,
- poznati moramo odvod funkcije.

Red konvergence metode

Red konvergence Newtonove metode je kvadraten (ker smo upoštevali še odvod funkcije):

$$\varepsilon_n = C\,\varepsilon_{n-1}^2,$$

kjer je C:

$$C = -\frac{f''(x)}{2f'(x)}.$$

Konvergenca je torej hitra, v vsaki novi iteraciji se število točnih števk v približku podvoji.

Implementacija metode

In […
```python
def newtonova(fun, dfun, x0, tol=1e-3, Dtol=1e-1, max_iter=50, izpis=True):
    # ime `newtonova` zato ker je `newton` vgrajena funkcija v `scipy`
    """ Vrne ničlo z natančnostjo tol

    :param fun:  funkcija katere ničlo iščemo
    :param dfun: f'
    :param x0:   začetni približek
    :param tol:  zahtevana natančnost
    :max_iter:  maksimalno število iteracij preden se izvajanje prekine
    :param Dtol:največja vsota absolutnih vrednosti rešitve
    :izpis:     ali na koncu izpiše kratko poročilo
    """

    for i in range(max_iter):
        x1 = x0 - fun(x0)/dfun(x0)
        if np.abs(x1-x0)<tol:
            r = (x0+x1)/2
            D = np.abs(fun(x0)) + np.abs(fun(x1))
            if D > Dtol:
                raise Exception('Opozorilo: Verjetnost pola ali več ničel!')
            if izpis:
                print('Rešitev: {:.5g}, število iteracij: {:d}, D: {:.8g}'.format(x1,i,D))
            return x1
        x0 = x1
    raise Exception('Napaka: Metoda po {:d} iteracijah ne konvergira!'.format(max_iter))
```

In […
```python
# Zapišimo naš polinom in njegov odvod

def f(x):
    return 5 - 10*x**2 + x**3

def dfdx(x):
    return -20*x + 3*x**2
```

In […
```python
# Testiramo metodo na naši funkciji

newtonova(f, dfdx, 1.0, 1e-8);
```

Tako se vidi, da je potrebno bistveno manj iteracij za doseganje enake natančnosti rezultata. NR metoda v kombiniaciji z bisekcijo se dejansko uporablja kot metoda za reševanje enačb!

Uporaba metode fsolve (scipy.optimize.fsolve)

V **scipy.optimize** knjižnjici imamo splošno metodo za reševanje enačb **fsolve**, ki se nahaja na povezavi scipy.optimize.fsolve.

In […
```python
[root, output, flag, msg] = opt.fsolve(f, 1.0, full_output=True)

print('Rešitev: {:8g}'.format(root[0]))
print('Izpis sporočila:\n')
print(output)
```

In […