SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 6784

METODA KONAČNIH RAZLIKA I PRIMJENE U RJEŠAVANJU SUSTAVA OBIČNIH DIFERENCIJALNIH JEDNADŽBI

Dora Šegvić

Zagreb, lipanj 2020.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 6784

METODA KONAČNIH RAZLIKA I PRIMJENE U RJEŠAVANJU SUSTAVA OBIČNIH DIFERENCIJALNIH JEDNADŽBI

Dora Šegvić

Zagreb, lipanj 2020.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 13. ožujka 2020.

ZAVRŠNI ZADATAK br. 6784

Pristupnica:	**Dora Šegvić (0036506064)**
Studij:	Računarstvo
Modul:	Računalno inženjerstvo
Mentor:	doc. dr. sc. Dario Bojanjac

Zadatak: **Metoda konačnih razlika i primjene u rješavanju sustava običnih diferencijalnih jednadžbi**

Opis zadatka:

Metoda konačnih razlika je jedna od osnovnih metoda aproksimacije operatora derivacije u postupcima za numeričko rješavanje diferencijalnih jednadžbi. U ovisnosti o aproksimaciji derivacije postoje stabilne i nestabilne metode U radu je potrebno opisati metode aproksimacije operatora derivacije te razliku između implicitnih i eksplicitnih metoda. Navedene metode potrebno je testirati na populacijskim jednadžbama te na sustavima populacijskih jednadžbi i jednadžbama širenja zaraze. Potrebno je ispitati ponašanje metoda u ovisnosti o veličini koraka diskretizacije te ponašanje na sustavima s malim parametrima.

Rok za predaju rada: 12. lipnja 2020.

Sadržaj

Uvod 1

1. Metoda konačnih razlika 2

1.1. Eulerova metoda prema naprijed 3

1.2. Eulerova metoda prema natrag 4

1.3. Crank-Nicolsonova shema 6

1.4. Zajedničko θ pravilo 7

2. Implementacija θ pravila u rješavanju diferencijalnih jednadžbi 9

3. Računalni model širenja virusnih bolesti 15

3.1. Matematička podloga 15

3.2. Implementacija modela širenja gripe 18

Zaključak 21

Literatura 22

Sažetak 23

Summary 24

Uvod

U današnjem svijetu u kojem tehnologija ulazi u sve pore društva, iznimno industriju i znanost javlja se jasna i očita potreba da probleme iz „stvarnog“ svijeta prirode rješavamo uz pomoć te sveprisutne tehnologije. To postaje moguće stvaranjem različitih računalnih modela koji na neki način preuzimaju ulogu tradicionalnog laboratorija i time postaju savršen prostor za istraživanja i provođenje različitih eksperimenata. Možemo reći da nam moderno doba nudi pristup rješavanja klasičnih problema znanosti na brz, siguran, pouzdan i jeftin način.

Za dobar računalni model potrebna je jaka matematička osnova na kojoj se gradi daljnja funkcionalnost. Činjenica je da većina modela koji se danas koriste u osnovi predstavljaju neko računalno rješenje sustava diferencijalnih jednadžbi. Da bi mogli implementirati takvo neko numeričko rješenje matematičkog problema jasno nam je da je znanje iz područja matematike nezaobilazno i neizostavno.

Postoje brojne metode koje nam omogućavaju rješavanje diferencijalnih jednadžbi na numerički način, no ona koja je definitivno najčešća i najraširenija je metoda konačnih razlika. Njezina „popularnost“ leži u tome što je iznimno jednostavna, a opet ima raznoliku primjenu u rješavanju širokog spektra matematičkih problema.

Cilj ovog rada je korištenjem i implementacijom upravo ove navedene metode stvoriti jednostavan programski kod napisan programskim jezikom Python koji rješava obične diferencijalne jednadžbe.

Nakon toga, daljnjom razradom i nadogradnjom želimo stvoriti jedan računalni model. Taj bi model trebao olakšati analizu i predviđanje širenja zaraznih bolesti poput gripe.

Na ovaj ćemo način izravno vidjeti kako kombinacija matematike i osnovnih znanja iz područja računarstva može pomoći riješiti probleme iz različitih područja o kojima uopće ne trebamo imati mnogo predznanja. Tako jedan običan programski kod koji rješava sustav tri diferencijalne jednadžbe može postati sredstvo za proučavanje problema iz područja biologije i medicine.

1. Metoda konačnih razlika

Metoda konačnih razlika je jedna od osnovnih i najčešće korištenih metoda aproksimacije operatora derivacije u postupcima za numeričko rješavanje diferencijalnih jednadžbi. Jednadžba koju želimo riješiti ima oblik:

$$u'(t) = -a * u(t) \quad (1)$$

Funkcija $u(t)$ je skalarna funkcija vremena t, a je konstanta te pretpostavljamo da je $a > 0$, $u'(t)$ je derivacija funkcije $u(t)$ s obzirom na vrijeme.

Ova jednadžba zapravo pokazuje smanjenje količine. Neka kvantiteta prikazana funkcijom u kroz vrijeme opada.

Kada bi ovu diferencijalnu jednadžbu riješili dobili bi opće rješenje:

$$u(t) = C * e^{-a*t},$$

koje bi mogli precizirati kada bi znali početne uvijete, odnosno vrijednost $u(0)$.

No naša želja nije riješiti ovu jednadžbu integriranjem, već numerički.

Za to su nam potrebna sljedeća saznanja:

- jednadžba koju rješavamo ima oblik (1),
- rješavamo ju unutar intervala $t \in (0, T]$,
- $u(0) = I$.

Ovaj je problem vrlo očiti kontinuiran problem, odnosno t poprima bilo koju vrijednost od 0 do T. Numerička metoda se bazira na tome da kontinuiran problem pretvorimo u diskretan tako što ćemo odabrati **konačan** broj vrijednosti $u(t)$. To radimo tako što prvo odabiremo set vrijednosti $t_0,\ t_1, \dots, t_{N-1}, t_N$, a zatim kreiramo rekurzivni algoritam koji će izračunati $u_0, u_1, \dots, u_{N-1}, u_N$.

Prvi korak se naziva diskretizacija domene, a njegov produkt su točke $t_0,\ t_1, \dots, t_{N-1}, t_N$ koje predstavljaju takozvani *mesh* odnosno *grid* odnosno mrežu. Često uzimamo takve točke da su one ravnomjerno raspoređene između granica – u ovom slučaju 0 i T, odnosno razmak između dvije točke jest Δt, a $t_n = n * \Delta t$.

Zatim želimo derivaciju, odnosno $u'(t)$ zamijeniti izrazom konačnih razlika.

Za rješavanje ovog problema koristimo tri sheme:

1. *Forward Euler shema (Eulerova metoda prema naprijed)*
2. *Backward Euler shema (Eulerova metoda prema natrag)*
3. *Crank-Nicolsonova shema*

1.1. Eulerova metoda prema naprijed

U ovoj shemi $u'(t)$ aproksimira se sljedećim izrazom:

$$u'(t_n) \approx \frac{u^{n+1} - u^n}{t_{n+1} - t_n} \tag{2}$$

Sada rješavamo jednadžbu:

$$\frac{u^{n+1} - u^n}{t_{n+1} - t_n} = -a * u^n \qquad u \in [0, N-1] \tag{3}$$

$$u^{n+1} = (1 - a(t_{n+1} - t_n)) * u^n \tag{4}$$

Uvrštavanjem početnog uvjeta dobivamo

$$u(0) = I$$

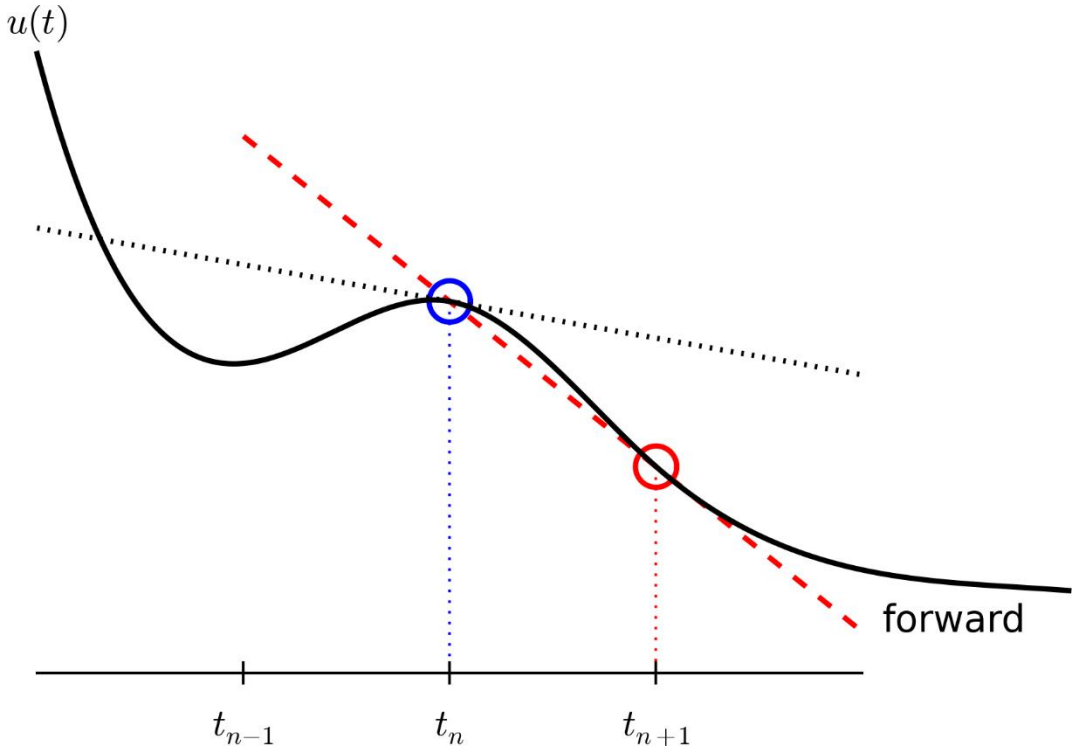
$$u(1) = u(0) - a * (t_1 - t_0) * u(0) = I(1 - a * \Delta t)$$

$u(2)$ postaje $I(1 - a * \Delta t)^2$ itd.

Zaključak je, a ujedno i rješenje

$$u^n = I(1 - a * \Delta t)^n$$

Aproksimacija se jasno vidi na slici (**Slika 1**).



Slika 1. Eulerova metoda prema naprijed [4]

1.2. Eulerova metoda prema natrag

Kao i za Eulerovu metodu prema naprijed prvo trebamo na neki način aproksimirati $u'(t)$.

Njezina aproksimacija ima sljedeći oblik:

$$u'(t_n) \approx \frac{u^n - u^{n-1}}{t_n - t_{n-1}} \tag{5}$$

Jednadžba koju sada rješavamo ima oblik:

$$\frac{u^n - u^{n-1}}{t_n - t_{n-1}} = -a * u^n \qquad u \in [1, N] \tag{6}$$

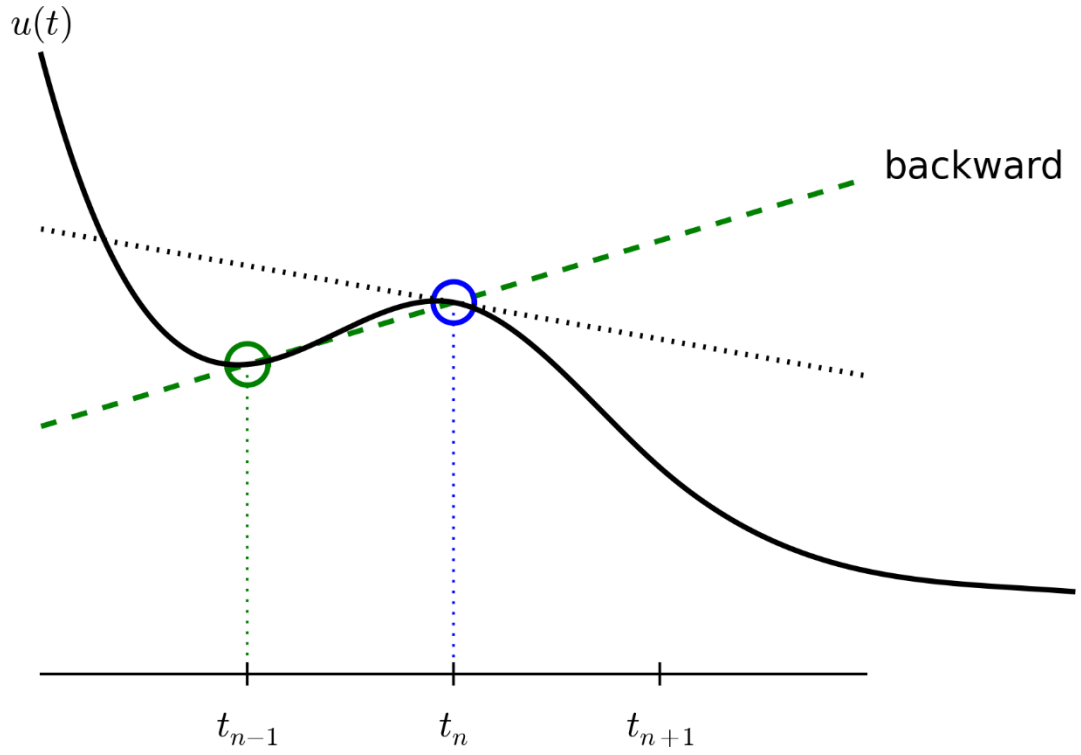
$$u^n = u^{n-1} - a * (t_n - t_{n-1}) * u^n \tag{7}$$

$$u^n = \frac{1}{1 + a * (t_n - t_{n-1})} * u^{n-1} \tag{8}$$

Jednakost (8) možemo zapisati na drukčiji način tako da umjesto $n \in [1, N]$ uvrstimo $n \in [0, N-1]$. Tada jednakost dobiva oblik:

$$u^{n+1} = \frac{1}{1 + a * (t_{n+1} - t_n)} * u^n \tag{9}$$

Aproksimaciju ovom shemom vidimo na slici (**Slika 2**)



Slika 2. Eulerova metoda prema natrag [3]

1.3. Crank-Nicolsonova shema

Eulerove metode prema naprijed i prema natrag su obje jednostrane aproksimacije derivacije. Kada bi imali centralnu aproksimaciju ona bi svakako bila točnija i preciznija od jednostranih. Upravo to nam nudi Crank-Nicolsonova shema.

Uzimamo diskretnu točku u vremenu koja ima oblik:

$$t_{n+\frac{1}{2}} = \frac{1}{2} * (t_n + t_{n+1}) = \left(n + \frac{1}{2}\right) * \Delta t$$

Aproksimacija tada ima oblik:

$$u'\left(t_{n+\frac{1}{2}}\right) \approx \frac{u^{n+1} - u^n}{t_{n+1} - t_n} \tag{10}$$

Sada rješavamo jednadžbu:

$$\frac{u^{n+1} - u^n}{t_{n+1} - t_n} = -a * u^{n+\frac{1}{2}} \tag{11}$$

Jednakost (11) zahtjeva još jednu aproksimaciju. S obzirom da je n nekakav cijeli broj između 0 i N-1, zaključujemo kako $n + \frac{1}{2}$ to sigurno nije. Zato uvodimo aproksimaciju:

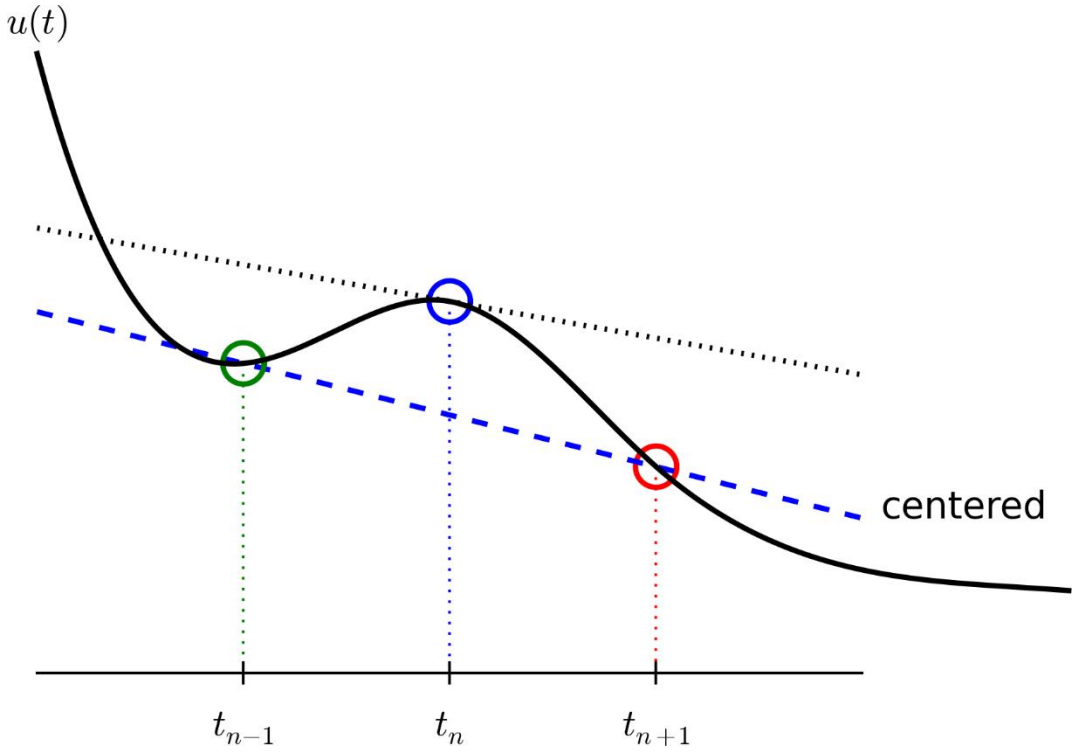
$$u^{n+\frac{1}{2}} \approx \frac{1}{2} * (u^n + u^{n+1}) \tag{12}$$

Rješavamo dalje:

$$\frac{u^{n+1} - u^n}{t_{n+1} - t_n} = -a * \frac{1}{2} * (u^n + u^{n+1}) \tag{13}$$

$$u^{n+1} = \frac{1 - \frac{1}{2} * a * (t_{n+1} - t_n)}{1 + \frac{1}{2} * a * (t_{n+1} - t_n)} * u^n \tag{14}$$

Aproksimaciju je predstavljena slikom (**Slika 3**)



Slika 3. Crank-Nicolsonova shema [5]

1.4. Zajedničko θ pravilo

Primjećujemo da se sve tri sheme mogu zamijeniti jednom zajedničkom jednadžbom uvodeći novi parametar θ. Unoseći različite vrijednosti za taj parametar $(0,1,\frac{1}{2})$ dobivamo upravo jednadžbe Eulerove metode unaprijed, unatrag i Crank-Nicolsonove sheme.

Jednadžba sada konačno ima oblik:

$$u^{n+1} = \frac{1-(1-\theta)*a*(t_{n+1}-t_n)}{1+\theta*a*(t_{n+1}-t_n)} * u^n \tag{15}$$

Usporedba procjene derivacije za različite sheme dana je u tablici (Tablica 1) gdje jasno možemo vidjeti veću preciznost Crank-Nicolsonove metode u odnosu na Eulerovu metodu prema naprijed i prema natrag. Pri izračunu početnu vrijednost smo postavili na 1, a parametar a na 2, dok razlika $t_{n+1} - t_n$ iznosi 0.8.

Tablica 1. Usporedba metoda

vrijeme	**F. Euler**	**greška**	**B. Euler**	**greška**	**C.-N.**	**greška**
0	1	0	1	0	1	0
0.8	-0.6	$8.019*10^{-1}$	0.384615	$-1.827*10^{-1}$	0.111111	$9.079*10^{-2}$
1.6	0.36	$-3.192*10^{-1}$	0.147929	$-1.072*10^{-1}$	0.012346	$2.8421*10^{-2}$
2.4	-0.216	$2.242*10^{-1}$	0.0568958	$-4.867*10^{-2}$	0.001372	$6.858*10^{-3}$
3.2	0.1296	$-1.279*10^{-1}$	0.021883	$-0.022*10^{-2}$	0.000152	$1.509*10^{-3}$
4.0	-0.077	$7.810*10^{-2}$	0.0084165	$-0.081*10^{-3}$	$1.69351*10^{-5}$	$3.185*10^{-4}$

2. Implementacija θ pravila u rješavanju diferencijalnih jednadžbi

Jednadžba (15) upravo je ona koju ćemo koristiti kako bi napisali jednostavan programski kod u Pythonu namijenjen rješavanju diferencijalne jednadžbe (1).

Mijenjanjem parametra θ želimo dobiti različite sheme i rezultate, za zadane parametre a, T, u^0 *i* Δt.

Funkcija koju implementira nazvana je solver i ima oblik kao u kodu (**Slika 4**).

T je u ovom slučaju prikazano argumentom period, u^0 je initial, dt je Δt, a theta odgovara parametru θ.

```
def solver(a, initial, period, dt, theta):
    """
    Solve u'=-a*u
    initial condition u(0)=initial
    for t in (0,period] with steps of dt.
    theta = 0 --- Forward Euler Scheme
    theta = 1 --- Backward Euler Scheme
    theta = 1/2 --- Crank-Nicolson Scheme

    """
    dt = float(dt)
    num_t = int(round(period / dt))  # no of time intervals
    period = num_t * dt  # adjust T to fit time step dt
    u = zeros(num_t + 1)  # array of u[n] values
    t = linspace(0, period, num_t + 1)  # time mesh
    u[0] = initial  # assign initial condition
    for n in range(0, num_t):  # n=0,1,...,Nt-1
        u[n + 1] = (1 - (1 - theta) * a * dt) / (1 + theta * dt * a) * u[n]
    return u, t
```

Slika 4. Solver funkcija

Funkciju pozivamo naredbom

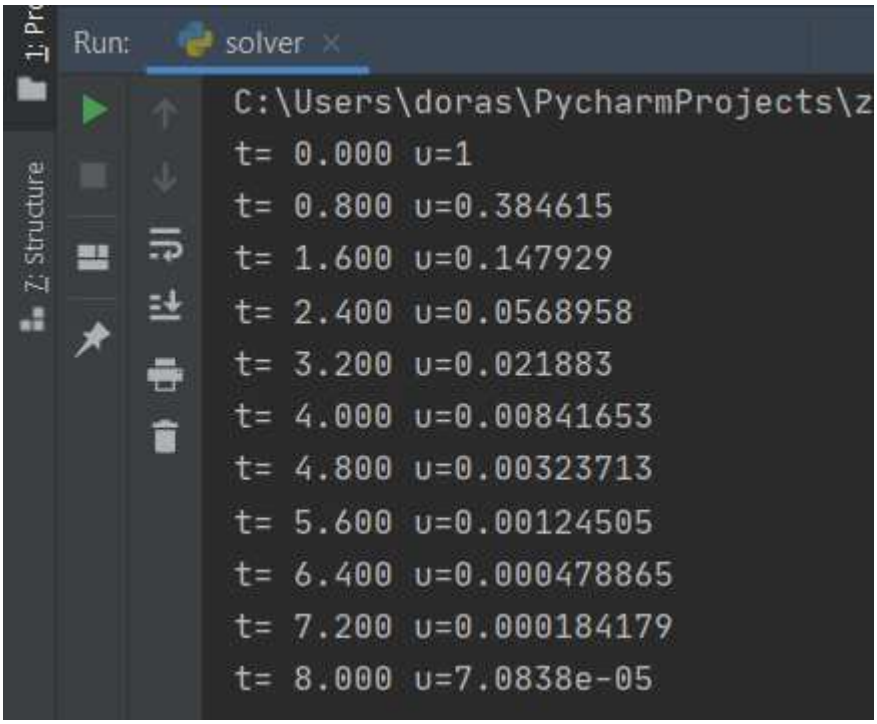
```
u, t = solver(a=2, initial=1, period=8, dt=0.8, theta=1)
```

gdje theta može biti 0, 1 ili 0.5. Parametar a je u ovom slučaju postavljen na 2, u^0, odnosno initial je 1, period T je postavljen na 8 , a Δt, odnosno razlika između t_{n+1} i t_n iznosi 0.8.

Ako želimo ispisati vrijednosti to činimo ovim dijelom koda nakon poziva funkcije.

```
for i in range(len(t)):
    print('t=%6.3f u=%g' % (t[i], u[i]))
    # or print 't={t:6.3f} u={u:g}'.format(t=t[i], u=u[i])
```

Pri tome dobivamo vrijednosti računa nakon pokretanja programa kao na slici (**Slika 5**). Ispis nam pokazuje za svaki trenutak t u intervalu [0, 8], gdje je razmak između t_{n+1} i t_n jednak zadanom Δt (u ovom slučaju to je 0.8).



Slika 5. Ispis nakon pokretanja

Ukoliko bi htjeli provjeriti računa li naša solver funkcija točne vrijednosti to bi mogli učiniti tako što ručno izračunamo prvih nekoliko vrijednosti i usporedimo odgovaraju li izračunatima. Uz pretpostavku da solver radi točno nastavljamo pisati kod dalje.

Kako bi provjerili preciznost same funkcije solver za različite vrijednosti θ prvo ćemo napisati funkciju koja računa egzaktnu vrijednost diferencijalne jednadžbe (1) za zadanu varijablu t. Time ćemo uvijek imati egzaktnu vrijednost s kojom dalje možemo uspoređivati

numerički dobivenu. Kod se nalazi na slici (**Slika 6**), a vrijednost koju vraća upravo je rješenje diferencijalne jednadžbe (1) koje dobijemo integriranjem i uvrštavanjem početnih uvjeta.

```
def u_exact(time, initial, a):
    """

    :param time: time
    :param initial: initial condition
    :param a: constant
    :return: exact function
    """
    return initial * exp(-a * time)
```

Slika 6. Kod za funkciju u_exact

Pišemo funkciju explore čiji je zadatak za različite parametre theta koje zadamo (inicijalno postavljeno u 0.5) iscrtavati graf egzaktno i numerički riješene diferencijalne jednadžbe. Funkciju vidimo na slici (). Ona poziva prethodno napisane funkcije solver i u_exact te vraća pogrešku E.

```
def explore(initial, a, period, dt, theta=0.5, makeplot=True):
    """
    Run a case with the solver, compute error measure,
    and plot the numerical and exact solutions (if makeplot=True).
    """
    u, t = solver(a, initial, period, dt, theta)   # Numerical solution
    u_e = u_exact(t, initial, a)
    e = u_e - u
    E = sqrt(dt*sum(e**2))
    if makeplot:
        figure()                    # create new plot
        t_e = linspace(0, period, 1001)      # fine mesh for u_e
        u_e = u_exact(t_e, initial, a)
        plot(t,   u,   'r--o')          # red dashes w/circles
        plot(t_e, u_e, 'b-')            # blue line for exact sol.
        legend(['numerical', 'exact'])
        xlabel('t')
        ylabel('u')
        title('theta=%g, dt=%g' % (theta, dt))
        theta2name = {0: 'FE', 1: 'BE', 0.5: 'CN'}
        savefig('%s_%g.png' % (theta2name[theta], dt))
        savefig('%s_%g.pdf' % (theta2name[theta], dt))
        show()
    return E
```

Slika 7. Kod za funkciju explore

Funkcija main (**Slika 8**) služi nam za poziv funkcije explore s različitim vrijednostima parametra θ. Ona ujedno ispisuje pogrešku koja nastaje pri numeričkom računu u odnosu na egzaktno rješenje.

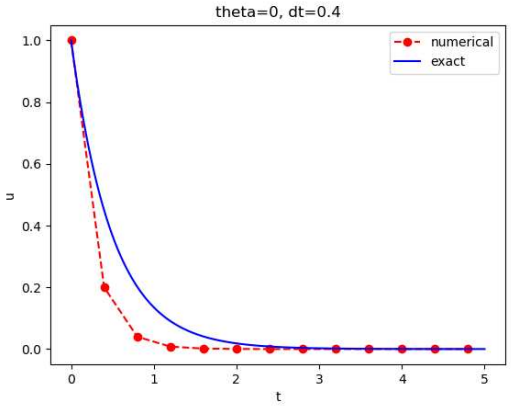
```
def main(initial, a, period, dt_values, theta_values=(0, 0.5, 1)):
    print("theta   dt      error")  # Column headings in table
    for theta in theta_values:
        for dt in dt_values:
            E = explore(initial, a, period, dt, theta, makeplot=True)
            print('%4.1f %6.2f: %12.3E' % (theta, dt, E))
```

Slika 8. Kod za funkciju main

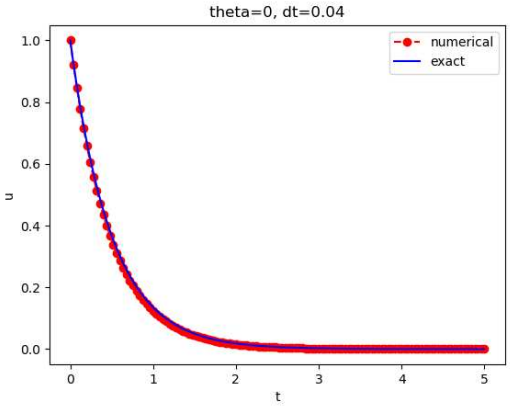
Kao što vidimo funkcija main također omogućava unos različitih Δt vrijednosti pa na grafovima možemo vidjeti i kako promjena tog parametra utječe na preciznost. Funkciju konačno pozivamo na sljedeći način:

```
main(initial=1, a=2, period=5, dt_values=[0.4, 0.04])
```

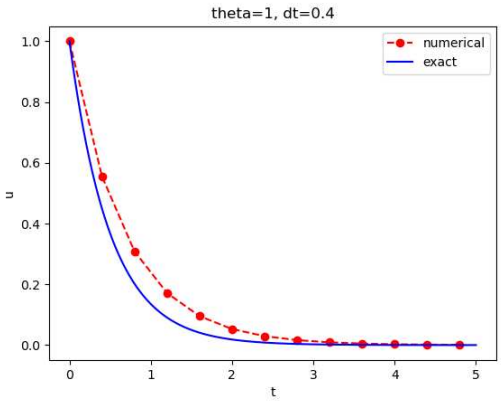
Grafovi koje dobijemo pokretanjem programa vidimo na slikama (**Slika 10**) i (**Slika 9**) za Eulerovu metodu prema naprijed (θ je 0), (**Slika 11**) i (**Slika 12**) Eulerovu metodu prema natrag (θ je 1), (**Slika 14**) i (**Slika 13**) Crank-Nicolsonovu shema (θ je 0.5). Plava linija predstavlja rezultat funkcije u_exact, a crvena isprekidana rezultat funkcije solver.



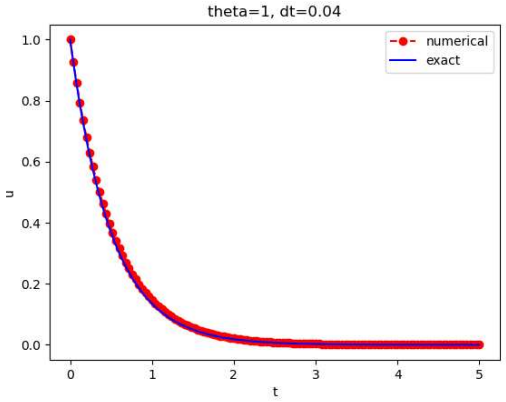
Slika 9. Forward Euler (Δt = 0.4)



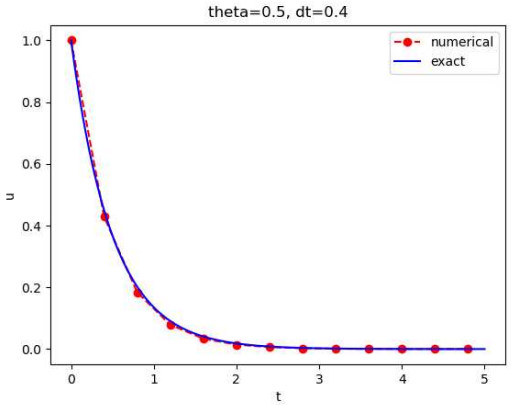
Slika 10. Forward Euler (Δt = 0.04)



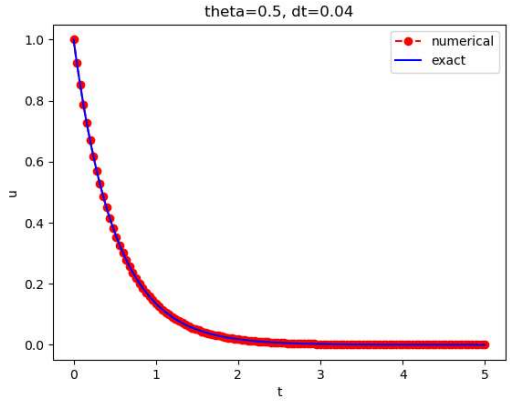
Slika 11. Backward Euler (Δt = 0.4)



Slika 12. Backward Euler (Δt = 0.04)

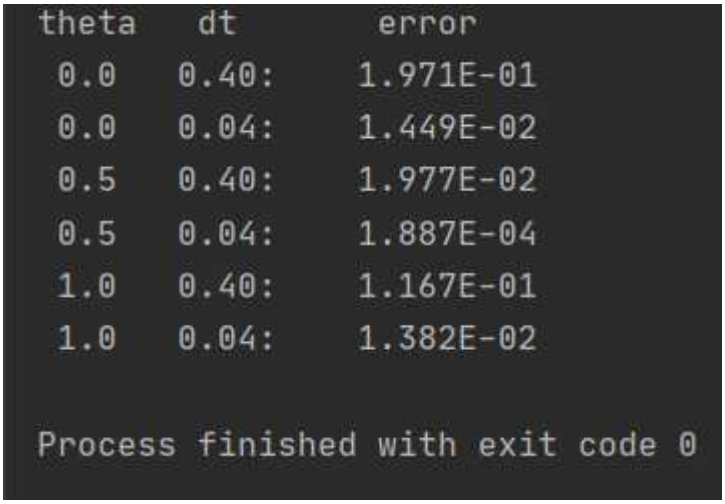


Slika 13. Crank-Nicolson (Δt = 0.4)



Slika 14. Crank-Nicolson (Δt = 0.04)

Kao što smo rekli main ispisuje i pogrešku s obzirom na θ i Δt. Taj ispis nakon pokretanja programa vidimo na slici (**Slika 15**).



Slika 15. Ispis funkcije main

S ovim možemo reći da smo u potpunosti implementirali θ pravilo kao način rješavanja diferencijalnih jednadžbi.

Također smo dobili i vizualnu reprezentaciju sve tri objašnjene sheme. Iz grafova (9) – (14), te sa slike (15) jasno uočavamo da je najpreciznija Crank-Nicolsonova shema što je i očekivan zaključak, s obzirom da je upravo ta shema jedina obostrana aproksimacija derivacije.

3. Računalni model širenja virusnih bolesti

Kroz rad smo se do sada upoznali s numeričkim aproksimacijama derivacija koje su nam potrebne za rješavanje diferencijalnih jednadžbi na diskretnoj razini. Takav način rješavanja uspjeli smo i implementirati. Sljedeći cilj bio bi iskoristiti takvu jednu implementaciju kako bi stvorili jedan matematički i računalni model koji će naći svoju primjenu u promatranju i boljem shvaćanju nekih od prirodnih procesa i pojava. Primjer kojeg ovaj rad uzima je širenje virusnih bolesti među populacijom.

3.1. Matematička podloga

Ovaj model zapravo je sustav 4 diferencijalne jednadžbe koje je potrebno riješiti.

Za početak uvodimo oznaku nekoliko kategorija, odnosno funkcija koje će opisati određene grupe populacije.

$S(t)$ je funkcija koja prikazuje broj jedinki koje se mogu zaraziti virusnom bolesti (za primjer uzmimo gripom) unutar vremena t.

$I(t)$ predstavlja broj inficiranih jedinki unutar vremena t.

Konačno imamo $R(t)$, funkciju koja prikazuje broj jedinki koje više nisu inficirane, a bile su, odnosno ili su stekle imunitet ili su umrle. Te jedinke svakako ne mogu napustiti kategoriju $R(t)$.

Kasnije ćemo uvesti još jednu funkciju, no za sad možemo proučiti ove tri navedene. Slika (**Slika 16**) nam pokazuje migracije jedinki između svake od funkcija.

Slika 16. SIR model

Jasno je da će se unutar nekog vremena određeni broj jedinki zaraziti, pa izvodimo diferencijalnu jednadžbu oblika:

$$S'(t) = -\beta * S(t) * I(t) \tag{16}$$

Gdje je β parametar koji nam govori koliko lako se jedinke zaraze. Možemo ga izračunati po formuli:

$$\beta = \frac{N}{T * S_0 * I_0} \tag{17}$$

Gdje je S_0 broj jedinki za koje sumnjamo da su zaražene, I_0 broj jedinki koje se zaraze u periodu T, a N broj jedinki za koje se sumnjalo da su zaražene i postale su zaražene unutra perioda T (odnosno prešle su iz S u I kategoriju).

Na sličan način izvodimo ostale diferencijalne jednadžbe:

$$I'(t) = \beta * S(t) * I(t) - \gamma * I(t) \tag{18}$$

Prvi dio ove jednadžbe odgovara izrazu suprotnom od onog u jednadžbi (16) što je dakako i logično jer broj jedinki koje su napustile kategoriju S prešao je u kategoriju I. Drugi dio odgovara broju jedinki koje napuštaju kategoriju I te prelaze u kategoriju R. γ pri tom predstavlja parametar koji nam govori koliko lako jedinke stječu imunitet i prestaju biti zaražene.

Posljednja jednadžba logično ima oblik:

$$R'(t) = \gamma * I(t) \tag{19}$$

Ukoliko želimo zaključiti je li došlo do izbijanja epidemije promatramo uvjet

$$\frac{\beta * S_0}{\gamma} > 1$$

Ako je uvjet istinit, tada možemo reći da je izbila epidemija.

Ako sada proširimo model tako što kažemo da određene jedinke kroz vrijeme mogu izgubiti imunitet i ponovno prijeći u kategoriju S jednadžbe dobivaju nove oblike:

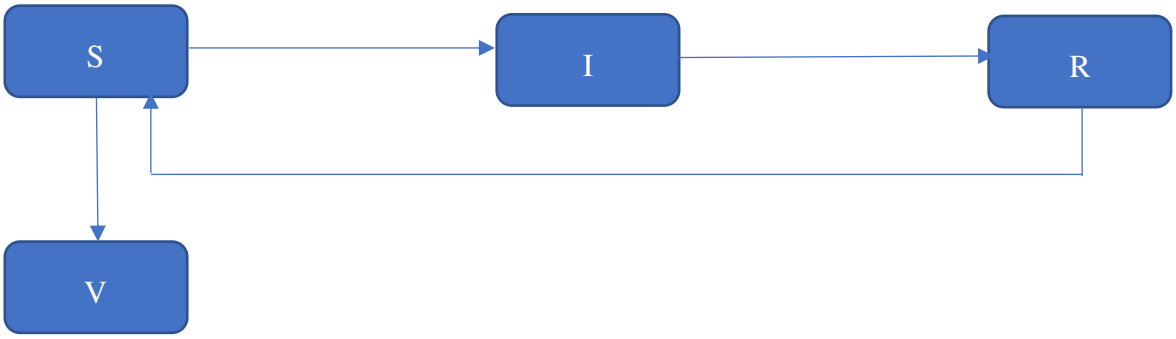
$$S'(t) = -\beta * S(t) * I(t) + \upsilon * R(t) \tag{20}$$

$$I'(t) = \beta * S(t) * I(t) - \gamma * I(t) \tag{21}$$

$$R'(t) = \gamma * I(t) - \upsilon * R(t) \tag{22}$$

Gdje je υ^{-1} tipično vrijeme potrebno da jedinka izgubi imunitet.

Dodajmo posljednju jednadžbu u naš model. Ona se odnosi na pojavu cjepiva. Time je omogućeno da pojedine jedinke nikada ne uđu u kategoriju I, već odmah iz S prelaze u novu kategoriju cijepljenih (V). To je prikazano na slici (**Slika 17**).



Slika 17. SIRV model

Konačne jednadžbe tada imaju oblik:

$$S'(t) = -\beta * S(t) * I(t) + \upsilon * R(t) - p * S(t) \quad (23)$$

$$I'(t) = \beta * S(t) * I(t) - \gamma * I(t) \quad (24)$$

$$R'(t) = \gamma * I(t) - \upsilon * R(t) \quad (25)$$

$$V'(t) = p * S(t) \quad (26)$$

Time je naš matematički model potpun i vrijeme je za implementaciju.

3.2. Implementacija modela širenja gripe

U implementaciji kao aproksimaciju derivacije koristimo Eulerovu metodu prema naprijed objašnjenu u poglavlju 1.1. , a implementiranu (između ostalog) u poglavlju 2. Program ponovno pišemo u programskom jeziku Python.

Prvi nam je zadatak definirati jednadžbe koje smo opisali u poglavlju 3.1., točnije jednadžbe (23), (24), (25) i (26). To činimo funkcijom (**Slika 18**) f u kojoj također definiramo parametre $\beta, \gamma \; i \; \upsilon$.

```
def f(u, t):
    S, V, I, R = u
    beta = 10. / (40 * 8 * 24)  # factor for affected
    beta /= 4
    gamma = 3. / (15 * 24)  # factor foe recovered
    nu = 1. / (24 * 50)  # factor for lost immunity
    v = p(t)
    return [-beta * S * I + nu * R - v * S, v * S, beta * S * I - gamma * I, gamma * I - nu * R]
```

Slika 18. Funkcija f

Primjećujemo kako faktor p (u funkciji izražen varijablom v) računamo pomoću posebne funkcije p(t). Tu funkciju vidimo na slici (). Ta nam funkcija kaže da 6. dan zaraze započinje cijepljenje i traje 10 dana.

```
def p(t):
    return 0.005 if (6 * 24 <= t <= 15 * 24) else 0
```

Slika 19. Funkcija p (kampanja cijepljenja)

Sada treba napisati funkciju (**Slika 20**) koja će numerički Forward Euler shemom rješavati zadane jednadžbe. Vidimo da funkcija rekurzivno poziva funkciju f omotanu lamda izrazom f_ koji pretvara rezultate u polje.

```
def solver(f, initial, period, dt):
    """
    Solve u'=f(u,t)
    using Forward Euler step

    """
    dt = float(dt)
    f_ = lambda u, t: asarray(f(u, t))
    num_t = int(round(period / dt))  # no of time intervals
    period = num_t * dt  # adjust T to fit time step dt
    u = zeros((num_t + 1, len(initial)))  # array of u[n] values
    t = linspace(0, period, len(u))  # time mesh
    u[0] = initial  # assign initial condition
    for n in range(0, num_t):  # n=0,1,...,Nt-1
        u[n + 1] = u[n] + dt * f_(u[n], t[n])
    return u, t
```

Slika 20. solver funkcija

Konačno želimo dobiti grafove na kojima ćemo najlakše iščitavati što se točno događa mijenjanjem parametara. To čini funkcija (**Slika 21**) plot_SVIR. Vidimo da funkcija prvo poziva solver, a zatim iscrtava na istu sliku sve četiri funkcije. Također provjerava ispravnost i konzistentnost računa.

```
def plot_SVIR(f, initial, dt, period):
    """Compare the numerical and exact solution in a plot."""
    u, t = solver(f, initial, period, dt)
    S = u[:, 0]
    V = u[:, 1]
    I = u[:, 2]
    R = u[:, 3]
    plot(t, S, 'r-', t, V, 'g-', t, I, 'b-', t, R, 'y-')
    legend(['S', 'V', 'I', 'R'])
    xlabel('hours')
    title('flu')
    savefig('plot_flu.png')
    show()
    # Consistency check:
    N = S[0] + V[0] + I[0] + R[0]
    eps = 1E-12  # Tolerance for comparing real numbers
    for n in range(len(S)):
        sum = S[n] + V[n] + I[n] + R[n]
    if abs(sum - N) > eps:
        print('***consistency check failed: S + I + R = % g != % g' % (sum, N))
```

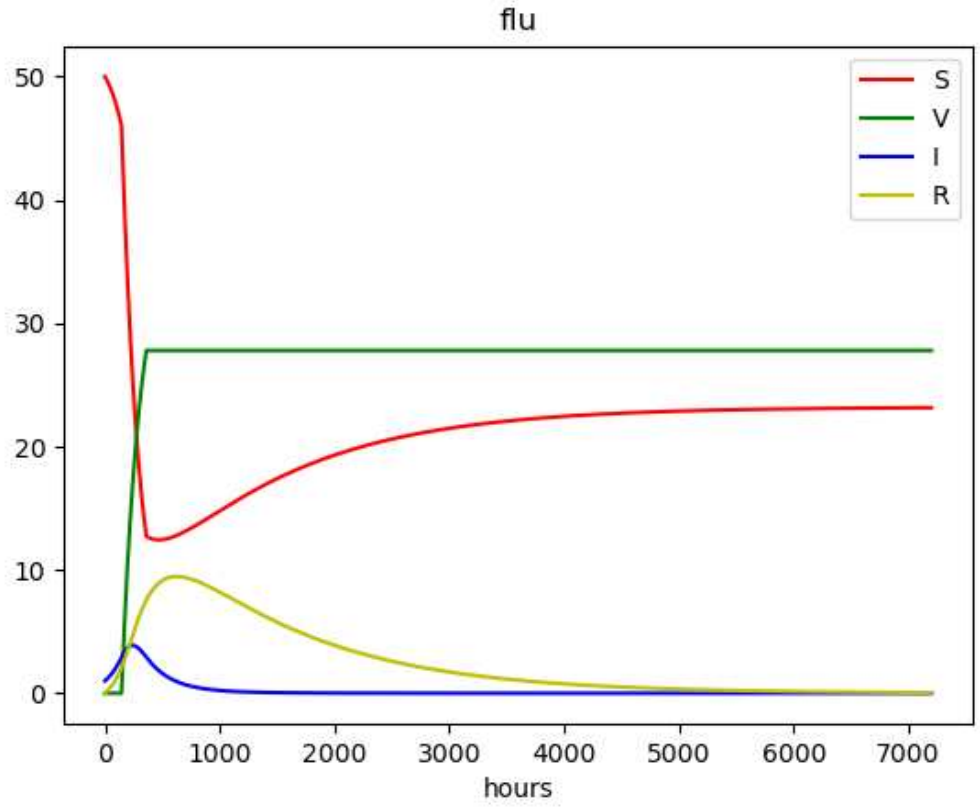
Slika 21. plot_SVIR funkcija

Prije pokretanja programa potrebno je još postaviti neke početne uvjete i nakon toga možemo pokrenuti program pozivom funkcije plot_SVIR.

```
dt = 0.1  # 6 min
D = 300  # Simulate for D days
n_t = int(D * 24 / dt)  # Corresponding no of hours
period = dt * n_t  # End time
initial = [50, 0, 1, 0]  # S,V,I,R

plot_SVIR(f, initial, dt, period)
```

Rezultat je kao što smo rekli graf (**Slika 22**). Crvena linija prikazuje broj jedinki u populaciji koje mogu biti zaražene, odnosno S. Plava linija bi bili zaraženi (kategorija I), žuta izliječeni (kategorija R) te konačno, zelena cijepljeni (kategorija V).



Slika 22. graf funkcija S, I, R, V

Promjenom početnih uvjeta mogli bi vidjeti kako se mijenja stanje, a promjenom parametara $\beta, \gamma, \upsilon \; i \; p$ mogli bi proučavati širenje drugih zaraznih bolesti.

Ovim možemo reći da je naš model kompletan.

Zaključak

Kroz čitav rad proučavamo ideju stvaranja računalnih modela koji nam omogućuju i olakšavaju proučavanje određenih pojava iz svijeta biologije, medicine, znanosti općenito, industrije i tako dalje.

Na primjeru širenja virusnih bolesti poput gripe među jedinkama populacije implementirali smo jedan takav model. Za to nam je bila potrebna dobra podloga iz područja matematike s naglaskom na rješavanje diferencijalnih jednadžbi, obzirom da se upravo na njima bazira naš model. Zbog jednostavne implementacije, poseban fokus obratili smo na numerički način rješavanja običnih diferencijalnih jednadžbi - konkretno na metodu konačnih razlika.

Ta nam je metoda bila alat, a njezinim razumijevanjem stvaranje samog modela postalo je jednostavnije uz osnovna znanja programiranja u programskom jeziku Python.

Upravo smo tako u ovom radu počeli od temelja – matematičkih izraza, jednadžbi i izvoda te smo preko implementacije same metode konačnih razlika naposlijetku došli do potpunog matematičkog i računalnog modela koji predviđa širenje gripe uzimajući u obzir brojne parametre poput pojave cjepiva, trajanja imuniteta lakoću širenja zaraze i tako dalje.

Možemo konačno zaključiti da smo počevši od metode konačnih razlika koja je u svijet matematike uvedena u 18. stoljeću od strane poznatog švicarskog matematičara Eulera uspjeli doći do nečega što nalazi široku primjenu u današnjem svijetu znanosti i tehnologije koja je daleko naprednija od onog u kojem je Euler živio.

Na neki način uistinu se možemo složiti s Gaussovom rečenicom, matematika zaista je kraljica znanosti.

Literatura

[1] Langtangen, H.P., *Finite Difference Computing with Exponential Decay Models*. Springer Open

[2] Linge, S., Langtangen, H.P., *Programing for Computations - Python*, Springer Open

[3] http://hplgit.github.io/INF5620/doc/pub/sphinx-decay/_images/fd_backward.png

[4] http://hplgit.github.io/INF5620/doc/pub/sphinx-decay/_images/fd_forward.png

[5] http://hplgit.github.io/INF5620/doc/pub/sphinx-decay/_images/fd_centered.png

Sažetak

Naslov: Metoda konačnih razlika i primjene u rješavanju sustava običnih diferencijalnih jednadžbi

Metoda konačnih razlika je jedna od osnovnih metoda aproksimacije operatora derivacije u postupcima za numeričko rješavanje diferencijalnih jednadžbi. U ovisnosti o aproksimaciji derivacije postoje stabilne i nestabilne metode koje ovaj rad proučava.

Navedene metode rad kasnije testira na sustavima populacijskih jednadžbi kao i jednadžbama širenja zaraze. Također se ispituje ponašanje metoda s obzirom na korak diskretizacije i ostale male parametre.

Summary

Title: Finite difference method and applications to systems of ordinary differential equations

Finite difference method is one of the main methods used to approximate differentation numerically. The method is used to solve differential equations. Depending on approximation there are two kind of methods: stable and unstable.

This paper tests this method on systems of population and disease spreading equations. It also analyses the effects of changing discretization step and other small parameters.