Univerza v Ljubljani
Fakulteta za elektrotehniko

MATLAB S SIMULINKOM
Priročnik za laboratorijske vaje

Simon Oblak
Igor Škrjanc

Ljubljana, 2008

PREDGOVOR

Programski paket MATLAB™ in orodje SIMULINK™ podjetja Mathworks sta nedvomno postala najbolj priljubljeni orodji pri reševanju inženirskih problemov. Čeprav sta bila sprva koristna pripomočka zgolj v raziskavah v univerzitetnem okolju, smo danes priča njuni vse večji uporabi v industriji. Na smeri Avtomatika se orodji že vrsto let uporabljata kot osnovna pripomočka za analizo in načrtovanje vodenja ter izvajanje simulacij. Študentje 3. letnika univerzitetnega študija in 2. letnika visokošolskega strokovnega študija se pri avditornih in laboratorijskih vajah spoznajo z uporabo orodij na omenjenih področjih, v višjih letnikih pa se nivo uporabe le še nadgrajuje. Ker je dobro poznavanje orodij temelj za uspešen študij problemov modeliranja, analize, simulacije in vodenja sistemov, pričujoči priročnik združuje podrobno razlago osnovnih in nekaterih naprednejših funkcij s primeri uporabe le-teh v izvedbah različnega obsega.

Povod za nastanek tega dela je povpraševanje študentov po priročniku, v katerem bi lahko našli vse potrebne informacije za reševanje problemov »tehnične« narave pri opravljanju laboratorijskih vaj. Kljub temu, da smo glede uvajanja uporabe Matlaba hitro sledili univerzam po svetu, ustreznega priročnika nismo izdali, tudi zato, ker so se nove različice izredno hitro menjavale in so vsi poskusi ostali v obliki delovnih verzij. Sodeč po izkušnjah upava, da bo delo dobrodošel pripomoček tako za študente, ki se prvič podajajo v reševanje omenjene problematike, kot tudi za študente višjih letnikov, ki želijo osvojiti ali obnoviti nekatere metode poglobljene uporabe orodij. Uporabnost priročnika pa ni omejena samo na študente avtomatike, namenjen je tudi vsem drugim študentom Fakultete za elektrotehniko in tudi študentom z drugih fakultet, ki na kakršenkoli način uporabljajo omenjeni orodji.

Priročnik je razdeljen na štirinajst poglavij, vsebinsko pa bi ga lahko razmejili na tri dele. V prvem delu (poglavja 1 do 7) so zbrani primeri uporabe osnovnih funkcij Matlaba: matrike, funkcije, m–datoteke in grafični prikaz. Drugi del (poglavji 8 in 9) podaja opis in primere uporabe funkcij iz knjižnice za analizo vodenja sistemov (Control System Toolbox) in knjižnice za simbolično računanje (Symbolic Toolbox). Tretji del (poglavja 10 do 14) pa se osredotoča na delo s Simulinkom in povezavo med obema okoljema. Poglavja so zasnovana tako, da je najprej podan pregled osnovnih sklopov funkcij, nato pa je znotraj vsakega sklopa razlaga funkcij in vedno tudi vsaj en primer uporabe. Primeri so izbrani v skladu s tematiko laboratorijskih vaj pri predmetih na smeri Avtomatika. Bralcu, ki ga zanima bolj poglobljena uporaba, pa je na voljo nekaj zahtevnejših primerov in programov, ki so večinoma zbrani v poglavjih Poglobljena uporaba Matlaba in Poglobljena uporaba Simulinka (poglavji 7 in 14).

Posebna zahvala gre sodelavcem Laboratorija za modeliranje, simulacijo in vodenje ter Laboratorija za avtomatizacijo in informatizacijo procesov, še posebej doc. dr. Alešu Beliču in prof. dr. Borutu Zupančiču, za dobrodošle napotke in nesebično pomoč pri nastajanju tega dela.

Simon Oblak in
Igor Škrjanc

VSEBINA

1. UVOD V MATLAB 7

MATLAB (ang. MATrix LABoratory) je interaktivno programsko orodje za numerično reševanje problemov. Razvijati so ga začeli v univerzitetnem okolju in je šele kasneje postal komercialni produkt. Njegov osnovni namen je bilo omogočiti enostaven dostop do programskih knjižnic linearne algebre, znanih pod imenoma LINPACK in EISPACK. Danes, v svoji sedmi izdaji, Matlabovo jedro uporablja knjižnici LAPACK in BLAS, ki predstavljata vrhunsko programsko opremo na področju matričnega računanja. Hitro in enostavno »rokovanje« z matrikami sodi še vedno med najmočnejša Matlabova orožja, še ena v vrsti njegovih odlik pa je tudi enostavna razširljivost s pomočjo programov, ki jih po njihovem podaljšku imen imenujemo m-datoteke. Vsako tako datoteko lahko obravnavamo kot novo funkcijo in jo uporabljamo na enak način kot vgrajene funkcije. Z združevanjem m-datotek v smiselno povezane celote lahko uporabnik sam zgradi obsežno orodje za namensko uporabo. Na tak način se je v letih od prve do trenutne različice Matlaba razvilo veliko število knjižnic funkcij (angl. toolbox), namenjenenih uporabi v specifičnih področjih, kot so načrtovanje vodenja sistemov, signalno procesiranje, statistična obdelava, umetna inteligenca itd.

Osnovni sistem Matlaba lahko razdelimo na 5 delov:

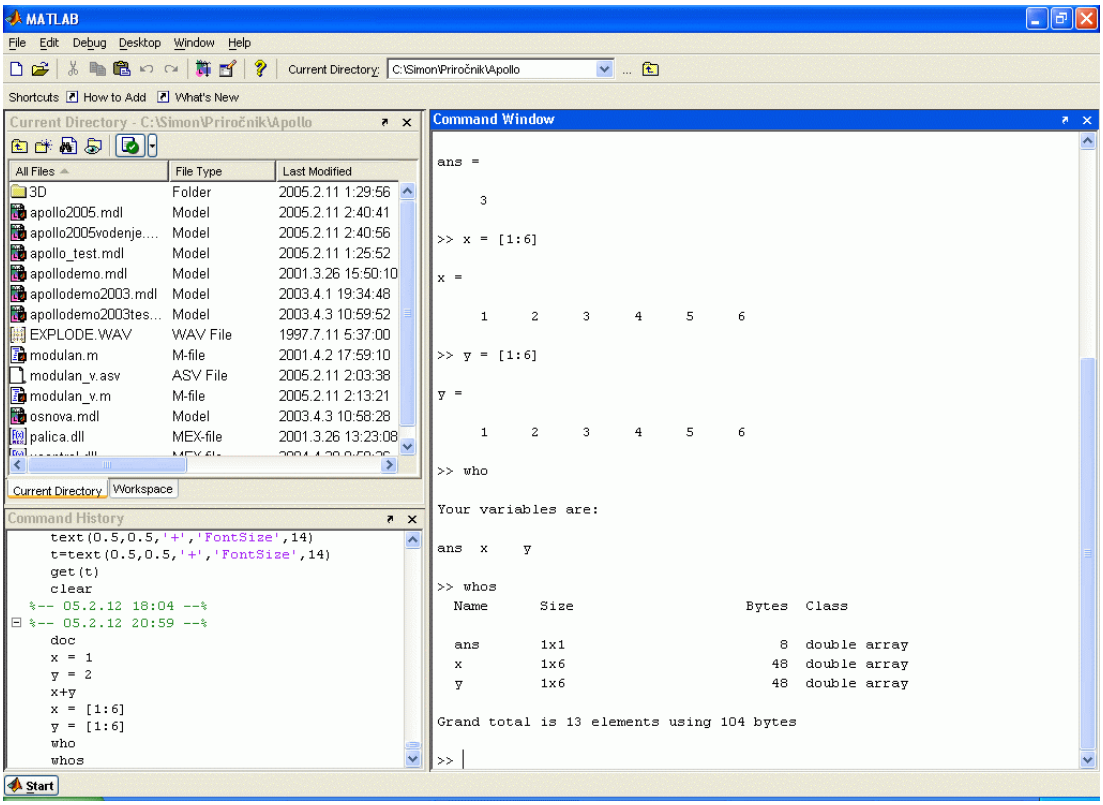
- **Razvojno okolje** je nabor orodij in okolij, ki omogočajo uporabo Matlabovih funkcij in datotek, in vključuje ukazno okno, ukazno zgodovino, tekstovni urejevalnik, razhroščevalnik ter okna za pomoč, delovni pomnilnik, delovno mapo in iskalnik.
- **Matematična funkcijska knjižnica** je obsežna zbirka računskih algoritmov, ki segajo od enostavnih (vsota, trigonometrija, kompleksna števila) do bolj kompleksnih funkcij (inverz matrik, Besslove funkcije in Fourierjevi transformi).
- **Programski jezik** je višjenivojski matrični jezik, ki vsebuje funkcije, podatkovne strukture in elemente vhodno-izhodnega in objektno orientiranega programiranja.
- **Grafika** – Matlab lahko vektorje in matrike predstavi v obliki dvo- in tridimenzionalnih diagramov. Vključuje tako višjenivojske ukaze za animacijo, obdelavo slik in prezentacijo kot nižjenivojske ukaze za oblikovanje izgleda grafičnih objektov.
- **Aplikacijski vmesnik** (ang. Application Program Interface (API)), ki omogoča pisanje programov v c-ju in fortranu, dinamično povezovanje Matlabovih funkcij in pisanje ter branje mat-datotek.

V priročniku se bomo dotaknili vsakega od omenjenih delov. V tem okviru bomo spoznali funkcije, ki jih v grobem razdelimo na vstavljanje podatkov, uporabo osnovnih Matlabovih funkcij, izvajanje skalarnih, vektorskih in matričnih operacij, uporabo zank *for* in *while* ter stavkov *if* in *switch,* pisanje m-datotek (ukaznih in funkcijskih), grafično prikazovanje rezultatov ter poglobljeno uporabo nekaterih funkcij analize, integracije in objektnega programiranja. Poseben poudarek pa bo namenjen tudi Matlabovemu uporabniškemu vmesniku, ki se od nekaterih prejšnjih verzij močno razlikuje.

2. KAKO ZAČETI Z DELOM V MATLABU

2.1 Delovni prostor

Po zagonu programa se nam prikaže kombinirani uporabniški vmesnik, ki v osnovni postavitvi omogoča delo v tekstovnem ukaznem oknu in sprotno spremljanje vsebine delovne mape in zgodovine ukazov, kot to kaže slika 2.1.



Slika 2.1 - Uporabniški vmesnik

Vloga okna *Command Window* se od prvih inačic Matlaba ni bistveno spremenila. Omogoča tekstovni vnos ukazov preko ukazne vrstice, ki je označena z '>>'.

V oknu *Current directory* imamo dostop do vseh datotek, ki se nahajajo v delovni mapi. S pomočjo miške ali tipkovnice lahko odpremo datoteke s končnicami **.m*, *.mat* in **.mdl*. Pomene končnic bomo razložili pozneje. Drugo okno na istem mestu je *Workspace*, v katerem so predstavljene vse delovne spremenljivke, ki se trenutno nahajajo v pomnilniku.

Ukazi, ki so že bili izvedeni, se shranjujejo na posebni lokaciji in so prikazani v oknu *Command History*. S pomočjo miške lahko ponovno izvedemo kateregakoli izmed njih, lahko pa jih uporabimo tudi več naenkrat, npr. pri pisanju m–datotek.

Privzeta delovna mapa ob zagonu Matlaba je C:\Matlab7\Work, vendar pa je priporočljivo, da ustvarimo novo delovno mapo izven osnovne in v njo shranjujemo vse podatke in datoteke. Do nje dostopamo preko okna *Current Directory* ali izbirnega menija na vrhu osnovnega okna. V ukaznem oknu to storimo z ukazom *cd*

```
>> cd mapa
```

ali pa podamo kar celotno pot do izbrane mape, npr.:

```
>> cd c:\priročnik\apollo
```

2.2 Ukazi, izrazi in spremenljivke

Matlab je interpretrski jezik. Vsak izraz, ki ga odtipkamo, je tolmačen (interpretiran) in ovrednoten posebej. Vnosi v Matlabu so ponavadi oblike:

```
>> spremenljivka = izraz
```

ali

```
>> izraz
```

Izrazi so sestavljeni iz operatorjev, funkcij in imen spremenljivk. Ko se izraz ovrednoti, Matlab zapiše rezultat v matriko in jo prikaže na zaslonu. Vrednost matrike se priredi določeni spremenljivki, katero lahko nato uporabljamo. Če ne navedemo imena spremenljivke in prireditvenega enačaja, priredi Matlab izraz spremenljivki z imenom *ans*, ki jo lahko kasneje uporabimo.

Vnos zaključimo s pritiskom na tipko Enter (CR), to je s prehodom v novo vrstico. Vendar pa v primeru, ko vnosa matrike še nismo zaključili z znakom ']', program čaka na dodatne vnose, ker pričakuje novo vrstico matrike. Če želimo nadaljevati izraz v naslednji vrstici, pa so vse matrike že zaključene, moramo pred znakom CR uporabiti tri oz. več zaporednih pik. Poglejmo primer.

```
>> y = [1:6]

y =

     1     2     3     4     5     6

>> x = 1 + 2*(y - 9) - 3*(2*y + 3) - ...
4*(-2*y + 2)

x =

   -30   -26   -22   -18   -14   -10
```

Če ne želimo izpisa rezultata (torej, če želimo izraz samo prirediti navedeni spremenljivki), moramo izraz zaključiti s podpičjem.

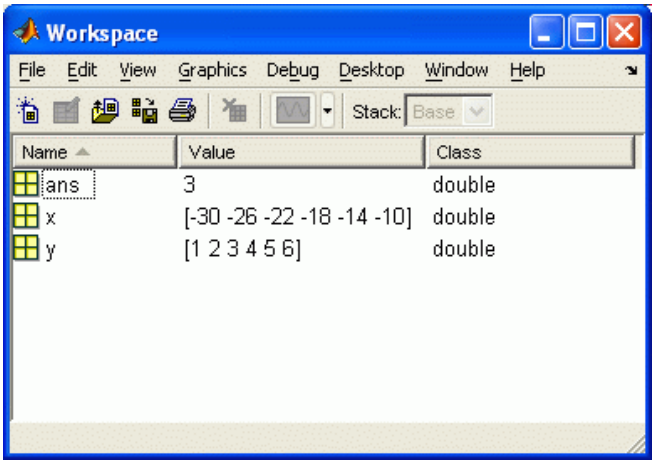
Stalno prisotna spremenljivka *eps* daje informacijo o natančnosti računalnika. Za večino računalnikov ima ta spremenljivka vrednost 10^{-16}. Spremenljivka *eps* je koristna pri določanju tolerance pri konvergenci iterativnih računskih procesov.

Matlab pozna v bistvu le eno vrsto objekta – pravokotno numerično matriko s kompleksnimi elementi. Vse spremenljivke so torej matrike. V posebnih primerih je matrika dimenzije 1×1 razumljena kot skalar, stolpična $(m\times1)$ ali vrstična $(1\times n)$ matrika pa kot vektor. Se pa matrike lahko med seboj ločijo po tipu podatkov posameznih elementov matrike. Tipe podatkov v Matlabu s primeri in razlago podaja tabela 2.1.

Tabela 2.1 - Tipi podatkov v Matlabu

Tip podatka	**Primer**	**Opis**
int, uint	65000 (uint16)	Pred- ali nepredznačena števila, ki potrebujejo manj pomnilnika kot *single* ali *double*
single	3 * 10^38	Števila z enojno natančnostjo. Zavzamejo manj prostora, a so manj natančna kot števila z dvojno natančnostjo in pokrivajo manjši obseg števil.
double	3 * 10^300, 5 + 6i	Števila z dvojno natančnostjo. Privzeti tip števil v Matlabu.
logical	A > 10	Logična števila z vrednostima 0 in 1, ki predstavljata Boolovi vrednosti 'napačno' in 'pravilno' ('false' in 'true'.)
char	'Živjo'	Znaki. Nizi so predstavljeni kot vektorji znakov, več nizov naenkrat pa zapišemo v celično polje..
cell array	a{1,1} = 12; a{1,2} = 'rdeča'; a{1,3} = A	Polje indeksiranih celic, vsaka od njih lahko vsebuje poljubno veliko polje kateregakoli tipa.
structure	a.dan = 12; a.barva = 'rdeča'; a.mat = A	Polje struktur, podobnim tistim v prog. jeziku c. Vsaka od njih je indeksirana in lahko vsebuje poljubno veliko polje kateregakoli tipa.
function handle	@sin	Kazalec na funkcijo. Kazalec ene funkcije lahko podamo tudi drugim funkcijam.

Spremenljivke, ki so trenutno naložene v delovnem prostoru, lahko vidimo v oknu *Workspace*, kot je prikazano na sliki 2.2.



Slika 2.2 - Okno *Workspace*

Na voljo so nam podatki o imenu, vrednostih in razredu spremenljivk. V ukaznem oknu dobimo seznam spremenljivk s pomočjo ukaza *who*:

```
>> who

Your variables are:

ans       x         y
```

Če želimo bolj natančno predstavitev spremenljivk delovnega prostora, pa vtipkamo *whos*:

```
>> whos
  Name      Size                    Bytes  Class

  ans       1x1                         8  double array
  x         1x6                        48  double array
  y         1x6                        48  double array

Grand total is 13 elements using 104 bytes

*******
```

Z ukazom *size* lahko poizvedujemo po dimenzijah matrik v delovnem prostoru:

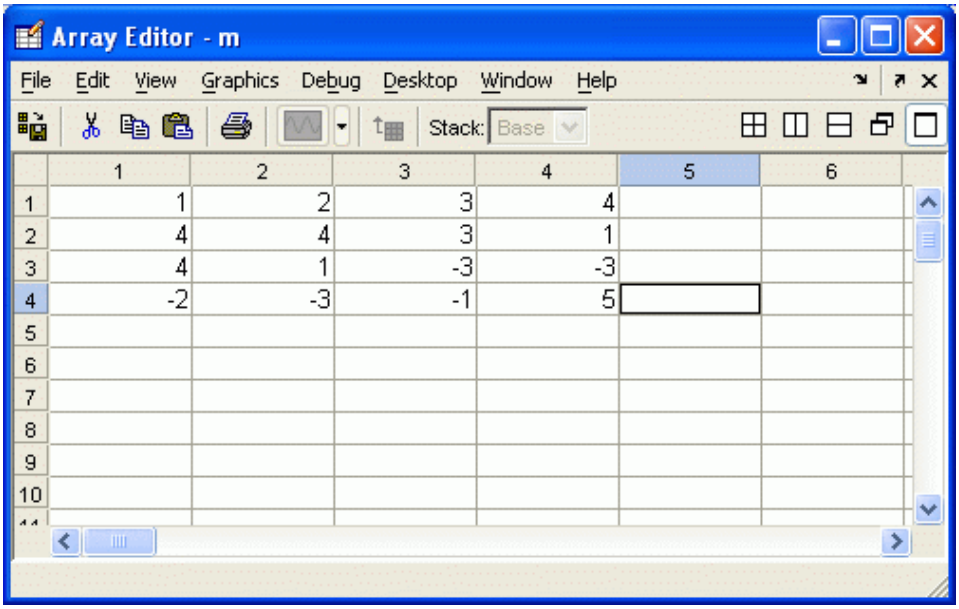
```
>> [m,n]=size(x)

m =

     1

n =

     6
```

Funkcija vrne izbrana parametra *m* in *n* obravnavane matrike, kjer *m* predstavlja število vrstic in *n* število stolpcev. V primeru, ko je spremenljivka, po kateri poizvedujemo, vektor, lahko uporabimo ukaz *length*:

```
>>  length(x)

ans =

      6
```

Potrebno je poudariti, da Matlab razlikuje med velikimi in malimi črkami (ang. case sensitive) v primeru spremenljivk in funkcij.

Matlab 7 omogoča tudi interaktivno urejanje elementov delovnega prostora. Če npr. z miško dvakrat kliknemo na spremenljivko *m* v oknu *Workspace*, se v delovnem oknu odpre urejevalnik *Array editor*. V tabelo lahko vnesemo poljubne vrednosti in s tem spremenimo tudi dimenzije objekta. Na sliki 2.3 vidimo, da je ob vnosu dodatnih 15 elementov spremenljivka *m* v oknu *Workspace* deklarirana kot <4x4 double>.



	1	2	3	4	5	6
1	1	2	3	4		
2	4	4	3	1		
3	4	1	-3	-3		
4	-2	-3	-1	5		
5						
6						
7						
8						
9						
10						

Slika 2.3 - Urejanje v okolju *Array editor*

Če želimo zbrisati elemente delovnega prostora, to storimo z ukazom *clear*:

```
>> clear
```

Večkrat pa se zgodi, da je potrebno zbrisati samo določen nabor spremenljivk, denimo *x* in *n*:

```
>> clear x n
```

V oknu *Workspace* to naredimo tako, da ju z miško označimo, kliknemo na eno od njiju z desno tipko in izberemo *Delete*. Brisanje spremenljivk je še posebej pomembno v primeru, ko želimo na novo izračunati vrednost vektorja *x*, ki je dimenzijsko manjši od predhodnega. Če prejšnjega vektorja ne zbrišemo, bo novi vektor enake dimenzije kot prejšnji, kjer bodo vrednosti elementov na mestih, ki presegajo dimenzijo novega vektorja, enake kot vrednosti predhodnjega vektorja *x*. Druga možnost v takem primeru je definicija prazne matrike:

```
>> x = []

x =

     []
```

Ta ukaz rezervira mesto v delovnem pomnilniku pod imenom spremenljivke. Uporabno vrednost tega ukaza bomo srečali kasneje.

Če želimo končati z delom v Matlabu in shraniti trenutne vrednosti spremenljivk delovnega prostora, potem to storimo z ukazom *save*, kjer je parameter ime datoteke *.mat, v katero hočemo shraniti celotno vsebino:

```
>> save ime_datoteke
```

Ta ukaz shrani spremenljivke v trenutno mapo. Če pa jih hočemo shraniti na poljubno mesto, le-to podamo v argumentu ukaza:

```
>> save c:\mapa\ime_datoteke
```

Če želimo shraniti samo določene spremenljivke, njihova imena podamo v argumentu funkcije.

```
>> save ime_datoteke x m n
```

Shranjene podatke lahko naložimo v delovni prostor z ukazom

```
>> load ime_datoteke
```

ali pa iz datoteke izberemo samo spremenljivke, ki jih potrebujemo.

```
>> load ime_datoteke spr1 spr2 …
```

Seveda lahko vse omenjene ukaze realiziramo tudi z miško v oknu *Workspace* na podoben način, kot izvedemo brisanje spremenljivke.

Seznam ukazov, ki smo jih podali v ukaznem oknu, lahko shranimo. To storimo z ukazom *diary*:

```
» diary ime_datoteke
```

Shranjevanje v datoteko lahko omogočimo z *diary on* ali onemogočimo z *diary off*. Sekvenco ukazov v datoteki lahko urejamo z ascii-urejevalnikom.

2.3 Oblika izpisa

Matlab za svoje računanje uporablja ti. *dvojno natančnost* (angl. double precision). Obliko izpisa pa lahko spreminjamo z naslednjimi ukazi

- **format short** – stalno mesto decimalne vejice in natančnost zapisa na 4 decimalna mesta (vnaprej nastavljena vrednost)
- **format long** – stalno mesto decimalne vejice in natančnost zapisa na 14 decimalnih mest
- **format short e** – zapis s plavajočo vejico (ang. floating point) s 3-mestnim eksponentom, natančnost zapisa na 4 decimalna mesta
- **format long e** – zapis s plavajočo vejico (ang. floating point) s 3-mestnim eksponentom, natančnost zapisa na 15 decimalnih mest
- **format hex –** zapis v šestnajstiškem številskem sestavu
- **format rat –** zapis z najboljšim približkom v obliki ulomka

Pri prvih štirih kombinacijah nam je na voljo še argument **g**, ki izbere kompaktnejšega od obeh zapisov (fiksna/plavajoča vejica in eksponentna oblika po potrebi). Format ostane enak, dokler ga ponovno ne spremenimo. Ukaz **format compact**, ki je neodvisen od formata izpisa, bo izločil večino presledkov na zaslonu, medtem ko **format loose** doda eno prazno vrstico po vsakem ukazu.

Kot primer smo vnesli število

```
>> a=0.4174859744578069
```

in preizkusili vse omenjene načine zapisa. Rezultati so zbrani v tabeli 2.2.

Tabela 2.2 - Izpis števila a = 0.4174859744578069 v različnih formatih

format short	0.4175
format short e	4.1749e-001
format short g	0.41749
format long	0.41748597445781
format long e	4.174859744578069e-001
format long g	0.417485974457807
format hex	3fdab81717b5702e
format rat	382/915

2.4 Pomoč v Matlabu

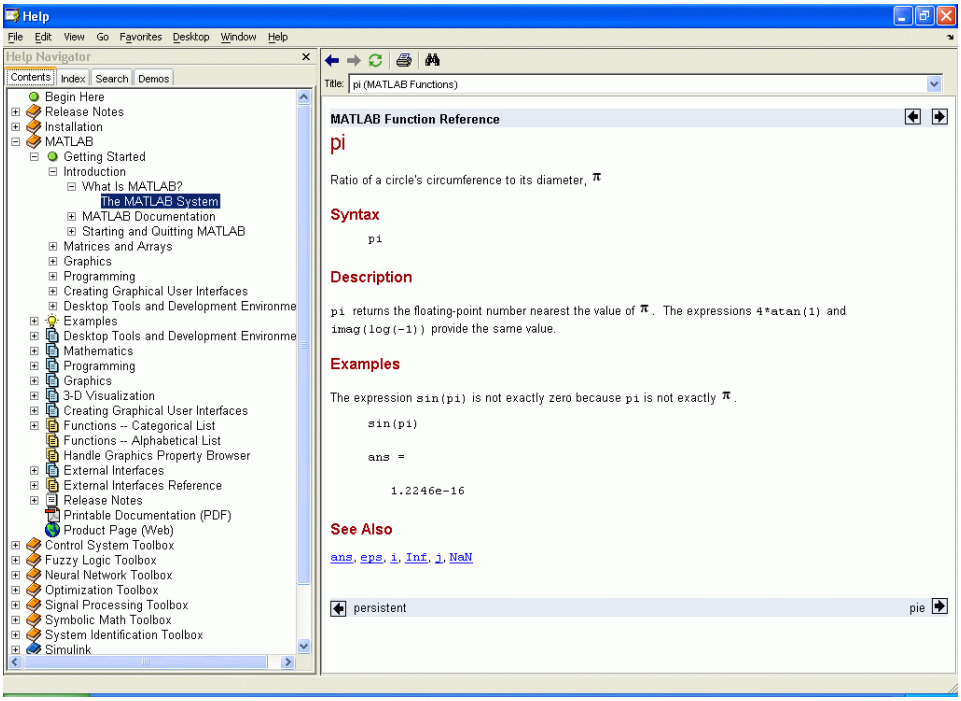
S pritiskom na tipko F1 se odpre okno *Help*, v katerem najdemo opis vseh funkcij v Matlabu, poleg tega pa nam je na voljo še indeksno iskanje in ogled demo-programov, ki v kompaktni obliki predstavijo delovanje večjih sklopov funkcij. Poleg tega pa Matlab nudi tudi sprotno (ang. on-line) pomoč v ukaznem oknu, saj vsaka funkcija vsebuje tudi opis svoje sintakse. Do nje dostopamo na dva načina: z ukazom *help* in ukazom *doc*. Poglemo si primer, kako Matlab obravnava število *pi* – najprej z ukazom *help*:

```
>> help pi
 PI     3.1415926535897....
    PI = 4*atan(1) = imag(log(-1)) = 3.1415926535897....

    Reference page in Help browser
       doc pi
```

Poskusimo še z ukazom *doc* – rezultat je prikazan na sliki 2.4:

```
>> doc pi
```

Okno *Help* uporablja pomoč v obliki *html*. Glavne rubrike, ki opisujejo uporabo funkcij, so *Syntax* (sintaksa ukaza), *Input arguments* (vrednosti, ki jih funkcija potrebuje za izvajanje, vendar ne nujno vseh), *Output arguments* (vrednosti, ki so na voljo kot izhod funkcije), *Algorithm* (matematično ozadje funkcije), *Examples* (primeri uporabe) in *See Als*o (nekatere funkcije s sorodno uporabo).



Slika 2.4 - Okno *Help* za ukaz *pi*

Seznam glavnih funkcij, ki nadzorujejo delo z operacijskim sistemom, urejajo delovanje v ukaznem oknu itd., lahko vidite tako, da si pomagate z ukazom:

```
>> help general
```

V primeru, ko ne poznamo natančnega imena funkcije, pa si lahko pomagamo z ukazom *lookfor*. Ukaz deluje kot indeks.

```
>> lookfor exponential
EXP     Exponential.
EXPINT Exponential integral function.
EXPM    Matrix exponential.
expmdemo.m: %% Matrix Exponentials
EXPMDEMO1  Matrix exponential via Pade approximation.
EXPMDEMO2  Matrix exponential via Taylor series.
EXPMDEMO3  Matrix exponential via eigenvalues and
eigenvectors.
EXPM    Symbolic matrix exponential.
```

3. MATRIKE IN POLJA ELEMENTOV

V tem poglavju bomo obravnavali osnovne operacije z matrikami in polji elementov.

3.1 Vstavljanje podatkov

Elemente matrike poljubnih dimenzij lahko vnesemo na več načinov. Ogledali smo si že *Array editor*, kjer so polja matrike predstavljena s tabelo elementov, kar nam omogoča poljubno vnašanje vrednosti in posledično tudi dimenzioniranje matrik. Ostali načini se izvajajo v delovnem oknu in so bolj primerni za skriptno izvajanje:

- kot zaporedje posameznih elementov
- z vgrajenimi izrazi ali funkcijami
- z m–datotekami
- iz zunanjih podatkovnih datotek

Če vnesemo izraz

```
>> A = [1 2 3;4 5 6;7 8 9]
```

ali izraz

```
>> A = [1  2  3
        4  5  6
        7  8  9]  ,
```

Matlab v obeh primerih oblikuje matriko velikosti 3×3 in jo priredi spremenljivki *A*:

```
A =

     1     2     3
     4     5     6
     7     8     9
```

Pri prvem zapisu lahko elemente v posamezni vrstici ločimo z vejico ali s presledkom, vrstice pa med seboj ločimo s podpičjem. Kadar vstavljamo matrike velikih dimenzij, je priporočljivo zapis shraniti v m–datoteko, ker lahko vsako napako enostavno popravimo z urejevalnikom.

Kadar napišemo število v eksponentni obliki (npr. 2.34e-4), se moramo izogibati presledkom, da ne bi prišlo do dvosmiselnih izrazov. Vnesimo vektor **x**:

```
>> x = [0.15 0.036 2.34e-4]

x =

    0.1500    0.0360    0.0002
```

Če napravimo pri zadnjem številu presledek med 4 in *e*, nam bo v primeru, ko smo *e* predhodno definirali kot spremenljivko z vrednostjo *e* = 5, program izpisal naslednji rezultat:

```
>> x = [0.15 0.036 2.34 e-4]

x =

    0.1500    0.0360    2.3400    1.0000
```

V vseh ostalih primerih pa bo javil napako v sintaksi, ker bo med 2.34 in *e* pričakoval znak za množenje.

Če hočemo, posebno v skriptnih programih, da nam Matlab javi neko sporočilo pred vnosom matrike in potem vnešeno preusmeri v želeno spremenljivko, uporabimo ukaz *input*:

```
>> A = input('Vnesi matriko A! ')
Vnesi matriko A! [3 2;4 5]

A =

     3     2
     4     5
```

Z ukazom *input* se bomo kasneje še srečali, saj omogoča tudi vnos poljubnega niza znakov in je zelo primeren za oblikovanje uporabniških pogojev.

Do posameznih elementov matrike ali vektorja lahko dostopamo z navedbo indeksov v oklepajih. `A(2,3) = 5` pomeni, da elementu v drugi vrstici in v tretjem stolpcu matrike *A* priredimo vrednost 5. Indeksi so lahko le pozitivna cela števila.

Najhitrejši način podajanja matrik ali vektorjev, ki imajo linearno odvisne elemente, je naslednji:

```
>> x = 1:2:11

x =
     1     3     5     7     9    11
```

Matlab generira vrstični vektor s 6 elementi, začetno vrednostjo 1, končno vrednostjo 11 in korakom 2. Seveda je možno narediti tudi padajoče zaporedje – v tem primeru je korak negativno število:

```
>> x = [14:-1.5:5.5]

x =

   14.0000   12.5000   11.0000    9.5000    8.0000    6.5000
```

Velja opozoriti, da v primeru, ko končna vrednost ni člen aritmetičnega zaporedja, določenega s korakom *k*, Matlab izbere za zadnji element vrednost prvega člena, ki je še manjši (naraščajoče zaporedje) ali večji (padajoče zaporedje) od končne vrednosti.

Če želimo preprečiti izpis spremenljivke, potem to storimo s podpičjem na naslednji način:

```
>> y = 5:5:50;
```

V tem primeru je vektor *y* definiran, ni pa izpisan v komandno okno. Vrednost tretjega elementa vektorja *y* lahko preberemo z ukazom:

```
>> y(3)

ans =

    15
```

Vrednost tega elementa se prenese v spremenljivko *ans*, ki se generira avtomatično pri izvedbi vsakega ukaza, kjer izhoda ne usmerimo v spremenljivko. Če sedaj spremenljivki *a* priredimo vrednost petega elementa vektorja **y**

```
a = y(5)

a =

    25
```

vrednost spremenljivke *ans* ostane nespremenjena:

```
>> ans

ans =

    15
```

Imaginarno število je predstavljeno s simboloma *i* ali *j*

```
>> i

ans =

        0 + 1.0000i

>> j

ans =

        0 + 1.0000i
```

Matrike so v splošnem definirane kot kompleksne, tako da vnos imaginarnega števila ne predstavlja večje ovire:

```
>> a = [sqrt(4), 1;2i+3, -5j]
```

```
a =

   2.0000             1.0000
   3.0000 + 2.0000i        0 - 5.0000i
```

3.2 Vstavljanje posebnih oblik matrik

Za oblikovanje nekaj posebnih vrst matrik si lahko pomagamo z Matlabovimi funkcijami

- **ones** – matrika enic
- **zeros** – matrika ničel
- **eye** – enotina matrika
- **rand** – matrika pozitivnih naključnih števil
- **randn** – matrika normalno porazdeljenih naključnih števil

Če vnesemo samo en vhodni argument *N*, dobimo kvadratno matriko dimenzije $N \times N$. Če podamo dva vhodna parametra, npr. *N* in *M*, bo rezultat matrika dimenzije $N \times M$. Poglejmo si primere.

```
>> M1 = ones(1,5)

M1 =

     1     1     1     1     1

>> M2 = zeros(2,3)

M2 =

     0     0     0
     0     0     0

>> M3 = rand(3)

M3 =

    0.3742    0.6700    0.8231
    0.9675    0.4562    0.5421
    0.6196    0.8380    0.0924

>> M4 = randn(3)

M4 =

   -1.3573   -0.3898    1.5532
   -1.0226   -1.3813    0.7079
    1.0378    0.3155    1.9574
```

```
>> M5 = eye(3)

M5 =

     1     0     0
     0     1     0
     0     0     1
```

Funkcija *eye* je izjema med naštetimi funkcijami, kajti ob neenakih vhodnih parametrih *N* in *M* ne dobimo enotine matrike. Diagonalni elementi so enaki 1, vsi ostali pa 0:

```
>> M6 = eye(3,4)

M6 =

     1     0     0     0
     0     1     0     0
     0     0     1     0
```

3.3 Operacije z matrikami

V Matlabu so vgrajene naslednje matrične operacije:

+	seštevanje,
–	odštevanje,
*	množenje,
^	eksponent,
'	transponiranje,
\	levo deljenje,
/	desno deljenje.

Vse te operacije lahko uporabimo tudi med skalarji. Če dimenzije matrik za določeno matrično operacijo ne ustrezajo, dobimo na zaslonu sporočilo o napaki. To se ne zgodi v primeru, če izvedemo operacijo med matriko in skalarjem (to velja za seštevanje, odštevanje, deljenje in množenje), ko se operacija izvede za vsak element matrike posebej.

Matrično deljenje si zasluži poseben komentar. Če je **A** invertibilna kvadratična matrika in **b** odgovarjajoči stolpni ali vrstni vektor, tedaj je

`>> x = A\b` rešitev enačbe $\mathbf{A}\cdot\mathbf{x}=\mathbf{b}$

in

`>> x = b/A` rešitev enačbe $\mathbf{x}\cdot\mathbf{A}=\mathbf{b}$.

Pri levem deljenju je v primeru, ko je matrika **A** kvadratna, le-ta faktorizirana s pomočjo Gaussove eliminacijske metode in nato uporabljena za rešitev enačbe $\mathbf{A}\cdot\mathbf{x}=\mathbf{b}$. Če matrika **A** ni kvadratna, je faktorizirana s Householderjevo ortogonalno faktorizacijo, nakar se z metodo najmanjših kvadratov rešuje predeterminiran ali

poddeterminiran sistem skalarnih enačb. Desno deljenje pa je izraženo z levim deljenjem na naslednji način:

```
b/A = (A'\b')'
```

3.4 Operacije s polji elementov

Matrični operaciji seštevanja in odštevanja delujeta le med istoležnimi matričnimi elementi, kar pa ne velja za ostale naštete matrične operacije. Poglejmo si primer za matriki **A** in **B**:

$$\mathbf{A} = \begin{bmatrix} 0.8060 & 0.0981 \\ -0.1447 & 0.4408 \end{bmatrix} \text{ in } \mathbf{B} = \begin{bmatrix} 0.1008 & 0.0091 \\ -0.0096 & 0.1073 \end{bmatrix}$$

```
>> A + B

ans =

    0.9068    0.1072
   -0.1543    0.5481

>> A*B

ans =

    0.0803    0.0178
   -0.0188    0.0460
```

Rezultat pri A*B ni produkt istoležnih elementov matrik **A** in **B**. Da bi operacije *, ^, \ in / uporabili in istoležnih elementih dveh matrik, postavimo med prvo matriko in operand piko:

```
>> A.*B

ans =

    0.0812    0.0009
    0.0014    0.0473
```

V tem primeru gre za skalarno operacijo

$$p(i,j) = a(i,j) \cdot b(i,j) . \tag{3.1}$$

Naredimo še preizkus, tako da pomnožimo elementa druge vrstice in prvega stolpca obeh matrik:

```
>> A(2,1)*B(2,1)

ans =

    0.0014
```

Ugotovimo lahko, da je dobljeni produkt enak elementu druge vrstice in prvega stolpca matrike A.*B. Podobno bi lahko pokazali tudi za ostale elemente.

3.5 Podmatrike, stolpci in vrstice

S pomočjo vektorjev in podmatrik lahko v Matlabu opravimo prenekatero kompliciranо operacijo s podatki. Stolpni zapis in indeksiranje sta ključ uspešnega manipuliranja s podatkovnimi strukturami. Z uporabo stolpnega zapisa se velikokrat znebimo nepotrebnega pisanja zank – le-te delovanje Matlaba izredno upočasnjujejo – in omogočimo enostaven in ločljiv zapis. Navaditi pa se moramo uporabljati vektorje namesto zank.

Izraz `1:5`, ki se, kakor bomo kasneje videli, uporablja tudi pri tvorjenju zank, je v bistvu vrstični vektor `[1 2 3 4 5]`. Kot smo videli v podpoglavju *Vstavljanje podatkov*, so elementi vektorja lahko poljubna števila in tudi korak ni omejen na ena. S sledečimi ukazi dobimo tabelo sinusov različnih kotov.

```
>> x = 0:0.1:1.2;
>> y = sin(x);
>> R = [x;y]

R =

  Columns 1 through 8

    0   0.1000  0.2000  0.3000  0.4000  0.5000  0.6000  0.7000
    0   0.0998  0.1987  0.2955  0.3894  0.4794  0.5646  0.6442

  Columns 9 through 13

    0.8000  0.9000  1.0000  1.1000  1.2000
    0.7174  0.7833  0.8415  0.8912  0.9320
```

Glede na to, da je *sin* skalarna funkcija, najprek vektor **x** preslikamo v vektor **y**, nato pa ga zapišemo v matriko **R**, ki predstavlja tabelo preslikave $\mathbf{y} = \sin(\mathbf{x})$

Stolpni zapis lahko uporabimo tudi za izločanje podmatrik iz celotnih matrik. Definirajmo matriko **A** kot

```
>> A = rand(5,5)

A =

    0.4175    0.5269    0.9103    0.3282    0.2470
    0.6868    0.0920    0.7622    0.6326    0.9826
    0.5890    0.6539    0.2625    0.7564    0.7227
    0.9304    0.4160    0.0475    0.9910    0.7534
    0.8462    0.7012    0.7361    0.3653    0.6515
```

`A(1:4,3)` je vrstični vektor, ki je sestavljen iz prvih štirih elementov tretjega stolpca matrike **A**:

```
>> A(1:4,3)

ans =

    0.9103
    0.7622
    0.2625
    0.0475
```

Lahko pa izločimo tudi celotno vrstico ali stolpec. `A(3,:)` predstavlja tretjo vrstico, z `A(1:3,:)` pa izločimo prve tri vrstice matrike **A**.

```
>> A(3,:)

ans =

    0.5890    0.6539    0.2625    0.7564    0.7227

>> A(1:3,:)

ans =

    0.4175    0.5269    0.9103    0.3282    0.2470
    0.6868    0.0920    0.7622    0.6326    0.9826
    0.5890    0.6539    0.2625    0.7564    0.7227
```

Poljubni celoštevilčni vektorji se lahko uporabljajo tudi za indeksiranje. Izraz `A(:,[2,4])` izloči 2. in 4. stolpec iz matrike **A** in ju zapiše v novo matriko:

```
>> A(:,[2,4])

ans =

    0.5269    0.3282
    0.0920    0.6326
    0.6539    0.7564
    0.4160    0.9910
    0.7012    0.3653
```

Tako indeksiranje lahko uporabimo na obeh straneh enačaja – omislimo si še matriko **B** dimenzije 5×5 in izvedimo operacijo `A(:,[2 4 5]) = B(:,1:3)`:

```
>> B = eye(5)

B =

     1     0     0     0     0
     0     1     0     0     0
     0     0     1     0     0
     0     0     0     1     0
     0     0     0     0     1
```

```
>> A(:,[2 4 5]) = B(:,1:3)

A =

    0.4175    1.0000    0.9103         0         0
    0.6868         0    0.7622    1.0000         0
    0.5890         0    0.2625         0    1.0000
    0.9304         0    0.0475         0         0
    0.8462         0    0.7361         0         0
```

Vidimo, da smo drugemu, četrtemu in petemu stolpcu matrike **A** priredili vrednosti prvih treh stolpcev matrike **B**. Na koncu pa izvedimo še matrično množenje – drugi in četrti stolpec nove matrike **A** pomnožimo z desne strani z matriko reda 2×2:

```
>> A(:,[2,4]) = A(:,[2,4])*[10 5;5 10]

A =

    0.4175   10.0000    0.9103    5.0000         0
    0.6868    5.0000    0.7622   10.0000         0
    0.5890         0    0.2625         0    1.0000
    0.9304         0    0.0475         0         0
    0.8462         0    0.7361         0         0
```

Pri matričnih operacijah moramo biti vedno pozorni na dimenzijsko ustreznost matrik in delov matrik, ki jih zapisujemo kot polja elementov.

Za konec pa si poglejmo še dodajanje elementov (večanje dimenzij vektorjev in matrik) ter uporabo argumenta *end*. Če želimo matriki **A** dodati nov stolpec na konec matrike in vse skupaj zapisati v matriko **C**, ne da bi pri tem spremenili originalno matriko, to naredimo na naslednji način:

```
>> C = [A [1;2;3;4;5]]

C =

    0.4175   10.0000    0.9103    5.0000         0    1.0000
    0.6868    5.0000    0.7622   10.0000         0    2.0000
    0.5890         0    0.2625         0    1.0000    3.0000
    0.9304         0    0.0475         0         0    4.0000
    0.8462         0    0.7361         0         0    5.0000
```

Podobno postopamo pri dodajanju vrstice:

```
>> D = [A;[1 2 3 4 5]]
```

```
D =

    0.4175   10.0000    0.9103    5.0000         0
    0.6868    5.0000    0.7622   10.0000         0
    0.5890         0    0.2625         0    1.0000
    0.9304         0    0.0475         0         0
    0.8462         0    0.7361         0         0
    1.0000    2.0000    3.0000    4.0000    5.0000
```

Kadar potrebujemo zadnje polje matrike ali vektorja, še posebej pri velikih dimenzijah matrik, si lahko pomagamo z ukazom

```
>> e = D(end,:)

e =

     1     2     3     4     5
```

Včasih se nam primeri, da moramo vrednosti vektorja **e** premakniti v desno, pri tem pa se dimenzija vektorja ne sme spremeniti. To lahko elegantno dosežemo z uporabo argumenta *end*, npr. za premik dveh vzorcev v desno

```
>> e = [0 0 e(1:end-2)]

e =

     0     0     1     2     3
```

Seveda pa uporaba takega indeksiranja pride resnično do izraza v kompleksnih programih, zapisanih v ukaznih in funkcijskih m–datotekah, z veliko zankami in matrikami velikih dimenzij.

4. FUNKCIJE, ZANKE IN RELACIJSKI OPERATORJI

4.1 Skalarne funkcije

Določene Matlabove funkcije delujejo le s skalarji. Če jih uporabimo na matrikah, se operacija izvrši na posameznih elementih matrike. Naštejmo nekaj tipičnih predstavnic:

- sin, sind, cos, cosd, tan, tand
- asin, acos, atan
- exp, pow2, log, log2, log10, sqrt
- abs, sign, imag, real, conj
- round, floor, ceil, fix
- rem, mod

Primer uporabe funkcije *sin* smo si že ogledali, zato se zdaj posvetimo funkcijama tangens (*tan*) in arkus tangens (*atan*), njenemu inverzu na intervalu $(-\pi/2, \pi/2)$. Izberimo si vektor **x**

```
>> x = -pi/4:pi/12:pi/4

x =

  -0.7854  -0.5236  -0.2618       0   0.2618   0.5236   0.7854
```

in izračunajmo $\mathbf{y} = \tan(\mathbf{x})$:

```
>> y = tan(x)

y =

  -1.0000  -0.5774  -0.2679       0   0.2679   0.5774   1.0000
```

Dobljene vrednosti pretvorimo nazaj in preverimo, če sta rezultata enaka:

```
>> z = atan(y)

z =

  -0.7854  -0.5236  -0.2618       0   0.2618   0.5236   0.7854

>> z-x

ans =

     0     0     0     0     0     0     0
```

Argument funkcije *tan* je podan v radianih. Če bi želeli uporabljati zapis v kotnih stopinjah, si lahko pomagamo s funkcijo *tand*:

```
>> x = -45:15:45

x =

   -45   -30   -15     0    15    30    45

>> y = tand(x)

y =

  -1.0000  -0.5774  -0.2679       0   0.2679   0.5774   1.0000
```

Rezultat je v obeh primerih enak. Analogno postopamo pri funkcijah *sin* in *cos*.

Definirajmo vektor **x** kot

```
>> x = [0.69315 7.3891 100 4];
```

in preizkusimo, kako delujejo funkcije *exp*, *log*, *log10* in *sqrt*:

```
>> y = [exp(x(1)) log(x(2)) log10(x(3)) sqrt(x(4))]

y =

     2     2     2     2
```

V vseh primerih dobimo rezultat 2. Prvi element v *x* je naravni logaritem števila 2, *exp* pa je funkcija e^x. Drugi element je enak e^2, *log* pa je naravni logaritem. *log10* je logaritem z osnovo 10, *sqrt* pa kvadratni koren. Poleg tega Matlab vključuje tudi funkciji *log2* (dvojiški logaritem, $\log_2 x$) in *pow2* (eksponentna funkcija z osnovo 2, tj. 2^x). Logaritme z ostalimi osnovami moramo računati po formuli za prehod na novo osnovo.

Funkciji *abs* (absolutna vrednost) in *sign* (signum) se uporabljata za spremembo predznaka skalarja ali elementov vektorja. Primer za signum:

```
>> x = [-3:3];

>> y = sign(x)

y =

    -1    -1    -1     0     1     1     1

>> z = abs(y)

z =

     1     1     1     0     1     1     1
```

Za neničelne elemente velja sign(x) = x / abs(x), sign(0) pa je enak 0. Funkciji *real* in *imag* vrneta realni in imaginarni del kompleksnega števila:

```
>> x = 2-3i

x =

   2.0000 - 3.0000i

>> y = [real(x) imag(x)]

y =

     2    -3
```

Podajmo primer za funkcije zaokroževanja *round*, *floor*, *ceil* in *fix*. Definirajmo vektor $\mathbf{x} = [-4.3287, 4.3287]^T$ in sestavimo matriko **Y** z naslednjim ukazom:

```
>> x = [-4.3287;4.3287];
>> Y = [round(x) floor(x) ceil(x) fix(x)]

Y =

    -4    -5    -4    -4
     4     4     5     4
```

Razlika med omenjenimi funkcijami je, da *round* zaokroži število proti najbližjemu celemu številu, *floor* na najbližje celo število proti $-\infty$, *ceil* na najbližje celo število proti $+\infty$ in *fix* na najbližje celo število proti 0.

Za konec si oglejmo še celoštevilsko deljenje in funkciji *mod* ter *rem*. Obe funkciji ponazarjata izraz

$$f(x, y) = x - n \cdot y \tag{4.1}$$

edina razlika pa je v interpretaciji koeficienta n, ki je pri *mod* definiran kot

$$n = \text{floor}(x./y), \quad y \neq 0, \tag{4.2}$$

pri *rem* pa

$$n = \text{fix}(x./y), \quad y \neq 0 \tag{4.3}$$

Če sta x in y istega predznaka, se rezultata ne razlikujeta, če pa sta različnega predznaka, ima rezultat pri *mod* isti predznak kot y, pri *rem* pa isti kot x. Še primer:

```
>> x = -10;,y = 3;
>> [mod(x,y) rem(x,y)]

ans =

     2    -1
```

4.2 Vektorske funkcije

Za razliko od skalarnih je pri vektorskih funkcijah vhodni argument vrstični ali stolpni vektor, izhodni argument pa skalar. Naštejmo nekatere izmed najbolj reprezentativnih vektorskih funkcij:

- max, min, median, mean
- sum, prod, cumsum, diff, sort
- any, all
- std, var

Če vektorsko funkcijo uporabimo na matriki velikosti $m \times n$, dobimo za rezultat vrstični vektor dolžine *n*, ki vsebuje rezultate za vsak vhodni stolpec posebej. Delovanje po vrsticah pa dobimo s transponiranjem vhodne matrike. Poglejmo si primere.

Definirajmo matriko **A** kot

$$\mathbf{A}=\begin{bmatrix} 3.5 & 6.4 & 1.3 & 9.8 \\ 3.4 & 6.1 & 1.5 & 9.6 \\ 3.2 & 6.6 & 1.2 & 9.4 \\ 3.3 & 6.5 & 1.4 & 9.9 \end{bmatrix}. \tag{4.4}$$

Funkcijo *max* lahko uporabimo na vrstici ali stolpcu matrike **A**,

```
>> [max(A(:,2)) max(A(2,:))]

ans =

    6.6000    9.6000
```

na stolpcih celotne matrike

```
>> max(A)

ans =

    3.5000    6.6000    1.5000    9.9000
```

ali na vrsticah, če uporabimo naslednjo obliko:

```
>> max(A')

ans =

    9.8000    9.6000    9.4000    9.9000
```

Na enak način uporabljamo funkciji *min* (najmanjši element vektorja) in *mean* (povprečna vrednost elementov vektorja). *Median* pa se od *mean* razlikuje glede statistične interpretacije rezultata – pri prvi je rezultat povprečni predstavnik množice elementov obravnavanega vektorja. Elemente funkcija najprej razvrsti po velikosti. Če vektor vsebuje liho število elementov, je rezultat srednji element, če pa jih je sodo število, potem je rezultat povprečje srednjih dveh elementov. Ponazorimo povedano v povezavi s funkcijo *sort*:

```
>> x = [0.1556 0.1911 0.4225 0.8560 0.4902 0.8159]
>> y = sort(x)

y =

    0.1556    0.1911    0.4225    0.4902    0.8159    0.8560

>> m1 = mean(y)

m1 =

    0.4886

>> m2 = median(y)

m2 =

    0.4564
```

Vidimo, da je število res povprečna vrednost 3. in 4. elementa vektorja **y**. Funkcijo *sort* pa lahko uporabimo tudi v obliki `[Y,I] = sort(X,DIM,MODE)`, kjer je *Y* sortirani vektor, *I* vektor z indeksi elementov na originalnih lokacijah, *X* vhodna matrika, *DIM* pove, ali ureja po stolpcih (*DIM* = 1) ali vrsticah (*DIM* = 2), *MODE* pa označuje smer urejanja (po naraščajočih ('ascend') ali padajočih ('descend') vrednostih). Pri funkcijah *sum*, *prod* in *cumprod* je na voljo matrična oblika `Y = f(X,DIM)`, ogledali pa si bomo vektorske oblike na vektorju *x* iz prejšnjega primera:

```
>> s = sum(x)

s =

    2.9313

>> p = prod(x)

p =

    0.0043

>> c = cumprod(x)

c =

    0.1556    0.0297    0.0126    0.0108    0.0053    0.0043
```

Funkcija *cumprod* poda kumulativni produkt, *i*-ti element rezultata je produkt *i*-tega in vseh predhodnjih elementov vhodnega vektorja.

Funkciji *any* in *all* kot rezultat vrneta boolovo spremenljivko, ki ima vrednost 1, če je katerikoli element vektorja različen od nič (*any*) ali če so vsi elementi različni od nič (*all*).

Funkcija *std* izračuna standardno deviacijo vektorja na dva načina po enačbi

$$\frac{1}{k}\sqrt{\sum_{i=1}^{N}(x_i - \frac{1}{N}\sum_{j=1}^{N} x_j)} \tag{4.5}$$

kjer je *k* ali *N* – 1 ali *N*. Drugi način dobimo tako, da dodamo 1 v vhodni argument – *y* = *std*(*x*,1). Varianca *var* je kvadrat standardne deviacije.

```
>> st = [std(x) std(x,1)]

st =

    0.2986    0.2726

>> v = [var(x) var(x,1)]

v =

    0.0892    0.0743
```

V drugem primeru je rezultat nižji zaradi deljenja s 6 namesto 5.

4.3 Posebne vektorske funkcije

Matlab nam ponuja tudi funkcije, ki vektor ali kombinacijo vektorjev preslika v vektor. To so

- poly karakteristični polinom
- roots koreni polinoma
- polyval izračun vrednosti polinoma v dani točki
- polyder analitični odvod polinoma
- polyfit aproksimacija danih podatkov s polinomom dane dimenzije

Oglejmo si primer uporabe funkcij *poly* in *roots*. Funkcijo *poly* se lahko uporablja na dva načina. Če je vhodni argument vektor, ki vsebuje *n* korenov polinoma, potem *poly* vrne vektor koeficientov polinoma *n*–tega reda. Denimo, da bi radi dobili koeficiente karakterističnega polinoma $p(x) = 0$, ki ima ničle pri vrednostih $x_1 = -0.34$, $x_2 = -1.45$, $x_3 = -2.67 + 5.46i$ in $x_4 = -2.67 - 5.46i$:

```
>> x = [-0.34,-1.45,-2.67+5.46i,-2.67-5.46i];
>> p = poly(x)

p =

    1.0000    7.1300   46.9921   68.7561   18.2117
```

Dobimo polinom $p(x) = x^4 + 7.1300x^3 + 46.9921x^2 + 68.7561x + 18.2117$. Če pa je vhodni argument matrika dimenzije $N \times N$, potem funkcija vrne vektor $N+1$ koeficientov karakterističnega polinoma $\det(\lambda \mathbf{I} - \mathbf{A})$. Za matriko **A**, podano z enačbo (4.4), tako dobimo:

```
>> p_lam = poly(A)

p_lam =

    1.0000  -20.7000   -3.9300    1.6130   -0.1766
```

oziroma polinom $p(\lambda) = \lambda^4 - 20.7\lambda^3 - 3.93\lambda^2 + 1.613\lambda - 0.1766$.

Funkcija *roots*(*x*) vrne vektor, v katerem so zapisani koreni polinoma, ki ga podamo v argumentu. Preizkusimo jo na polinomu $p(\lambda)$ iz prejšnjega primera:

```
>> r = roots(p_lam)

r =

  20.8845
  -0.4172
   0.1164 + 0.0820i
   0.1164 - 0.0820i
```

S pomočjo dobljenega rezultata lahko polinom $p(\lambda)$ zapišemo tudi kot:

$$p(\lambda) = (\lambda + 0.4172)(\lambda - 20.8845)(\lambda - 0.1164 + 0.082i)(\lambda - 0.1164 - 0.082i) \quad (4.6)$$

Za obravnavo ostalih treh funkcij si zadajmo sledečo nalogo – naj bo na voljo tabela podatkov (x, $f(x)$), kjer $f(x)$ dobimo po naslednji enačbi:

$$f(x) = \sin(0.2x)\cos(0.3x) \quad (4.7)$$

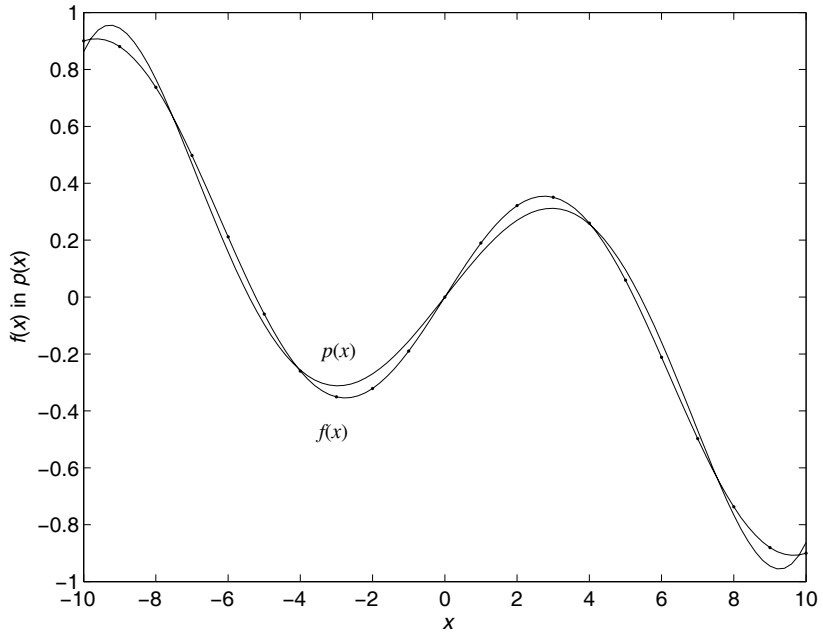
Naj ima x 21 elementov med –10 in 10. Potrebno je aproksimirati dane podatke s polinomsko funkcijo in izračunati strmino tangente na krivuljo v točki $x_0 = 2$. Začnimo s konstrukcijo funkcije $f(x)$.

```
>> x = [-10:10];
>> f = sin(0.2*x).*cos(0.3*x);
```

Poiščimo polinomsko krivuljo, ki se danim podatkom najbolje prilega v smislu najmanjših kvadratov. Izberimo red polinoma $n = 5$:

```
>> n = 5;
>> p = polyfit(x,f,n)

p =

    0.0000   -0.0000   -0.0068    0.0000    0.1613   -0.0000
```

4 koeficienti so v tem formatu izpisa enaki 0, vendar bodimo pozorni, kajti $p(1)$ = 4.2791e-005, $p(2)$ pa je enak –2.5466e-019. Po drugi strani pa je pomembna tudi interpretacija rezultatov, tako da bi pri splošni uporabi $p(2)$ zanemarili. Slika 4.1 prikazuje, koliko se dobljena polinomska funkcija razlikuje od osnovne.



Slika 4.1 - Primerjava polinomske in originalne funkcije

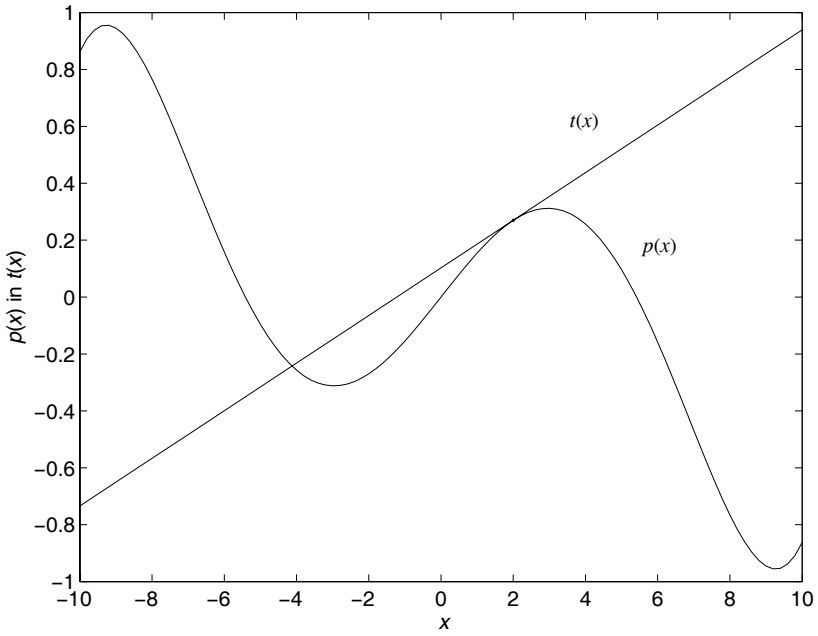
Funkcijske vrednosti polinoma smo ovrednotili z ukazom

```
>> xt = [-10:0.2:10];
>> pv = polyval(p,xt);
```

Zdaj pa poiščimo tangento na krivuljo. Odvod polinoma (*polyder*) in parametre tangente dobimo na naslednji način:

```
>> pd = polyder(p)

pd =

    0.0002   -0.0000   -0.0203    0.0000    0.1613

>> T = 2;
>> k = polyval(pd,T);
>> t = k*x + polyval(p,T) - k*(T);
```

Dobimo rezultat $k = 0.0837$. Končni izgled diagrama prikazuje slika 4.2.



Slika 4.2 - Tangenta na polinom v točki $T = 2$

4.4 Matrične funkcije

Ker je osnovni gradnik matrika, Matlab pride najbolj do izraza ob uporabi vgrajenih matričnih funkcij. Naštejmo nekatere izmed njih:

- eig lastne vrednosti in lastni vektorji,
- chol faktorizacija Cholesky,
- svd dekompozicija singularnih vrednosti,
- inv inverz matrike,
- lu LU-faktorizacija,
- qr QR-faktorizacija,
- hess Hessenbergova oblika,
- schur Schurova dekompozicija,
- rref reducirana vrstična oblika echelon,
- expm matrični eksponent,
- sqrtm kvadratni koren matrike,
- det determinanta matrike,
- size dimenzija matrike,
- norm norma-1, norma-2, norma-∞, norma-F (Frobenius),
- cond mera pogojenosti v normi-2,
- rank rang matrike.

Obravnava vseh funkcij zahteva poglobljeno matematično analizo in presega obseg tega dela, zato si bomo pogledali samo izbrane primere. Vzemimo matriko **A** iz prejšnjega primera in izračunajmo:

- lastne vrednosti (diagonala matrike **V**) in lastne vektorje (stolpci matrike **U**)

```
>> [U,V] = eig(A)
```

```
U =

  -0.5054      0.1704      0.6518             0.6518
  -0.4957     -0.4779      0.1875 - 0.2762i   0.1875 + 0.2762i
  -0.4909      0.8517      0.0907 - 0.5117i   0.0907 + 0.5117i
  -0.5078      0.1310     -0.3595 + 0.2537i  -0.3595 - 0.2537i

V =

  20.8845         0           0                    0
        0   -0.4172           0                    0
        0         0      0.1164 + 0.0820i          0
        0         0           0               0.1164 - 0.0820i
```

- determinanto,

```
>> D = det(A)

D =

   -0.1766
```

- inverz (če je determinanta različna od nič),

```
>> IN = inv(A)

IN =

    1.9479    2.3103    1.2514   -5.3567
   -1.9819    1.3703    2.8256   -2.0498
   -5.5606    5.7305    2.2707   -2.2084
    1.4383   -2.4802   -2.5934    3.5447
```

- rang matrike,

```
>> R = rank(A)

R =

     4
```

- in različne oblike norm.

```
>> [norm(A,1) norm(A,2) norm(A,inf) norm(A,'fro')]

ans =

   38.7000   24.3018   21.1000   24.3070
```

Razlike med oblikami norm so podane v tabeli 4.1. Način računanja norm je zapisan v Matlabovi kodi.

Tabela 4.1 - Različne oblike norm

norm(A,1)	max(sum(abs(A)))
norm(A,2)	max(eig(A))
norm(A,inf)	max(sum(abs(A')))
norm(A,'fro')	sum(diag(A'*A))

4.5 Relacijski operatorji

Matlab definira šest relacijskih operatorjev. To so:

<	manjše
>	večje
<=	manjše ali enako
>=	večje ali enako
==	enako
~=	različno

Ukaz `A<B` izvrši relacijo »manjše« na vseh istoležnih elementih matrik **A** in **B** in vrne matriko iste dimenzije kot **A** ali **B** z vrednostmi 1 na tistih lokacijah, kjer je element iz **A** manjši od istoležnega v **B**, in 0, kjer to ne drži. Poglejmo si primer. Izberimo si matriko **B**, matriko **A** pa dobimo tako, da od matrike **B** odštejemo enotino matriko.

```
>> B = rand(3)

B =

    0.4103    0.3529    0.1389
    0.8936    0.8132    0.2028
    0.0579    0.0099    0.1987

>> A = B - eye(3)

A =

   -0.5897    0.3529    0.1389
    0.8936   -0.1868    0.2028
    0.0579    0.0099   -0.8013

>> C = A<B


C =

     1     0     0
     0     1     0
     0     0     1
```

```
>> D = A==B

D =

     0     1     1
     1     0     1
     1     1     0
```

Ne smemo pozabiti, da je simbol '=' uporabljen kot prireditveni operator, simbol '==' pa kot relacija, ki izraža enakost elementov dveh matrik.

Relacije lahko vrednotimo ali povezujemo med seboj z logičnimi operatorji.

&	and (logični *in*)
|	or (logični *ali*)
~	not (logični komplement)
xor	ekskluzivni ali

Lahko jih uporabimo med skalarji, skalarjem in matriko ali med dvema matrikama istih dimenzij. Vrednosti relacij, povezanih z omenjenimi logičnimi operatorji, podaja tabela

Tabela 4.2 - Logične operacije

A	0	0	1	1
B	0	1	0	1
A&B	0	0	0	1
A|B	0	1	1	1
xor(A,B)	0	1	1	0

Komplement "~" se uporablja za komplementiranje logičnih vrednosti. Matrika ~**A** ima vrednosti 0 tam, kjer ima matrika **A** vrednosti 1, in obratno.

4.6 Stavek *for*

V svoji osnovni obliki Matlabovi krmilni stavki delujejo tako kot pri večini višjih programskih jezikov. Oglejmo si osnovno obliko ukaza *for ... end*. Za dano spremenljivko *n* bosta izraza

```
>> n = 6, x = []; for i = 1:n, x = [x,i^2]; end;
```

ali

```
>> x = [],n = 6;
>> for i = 1:n
x = [x,i^2];
end
>>
```

naredila vektor dimenzije *n*:

```
>> x

x =

     1     4     9    16    25    36
```

Obraten vrstni red elementov dosežemo z negativnim korakom v ukazu *for*:

```
>> x = [], for i = n:-1:1, x = [x,i^2], end

x =

    36    25    16     9     4     1
```

Druga oblika izraza je primerna za zapis v m–datoteko. Ukaz *for* se mora vedno zaključiti z *end*, tudi če uporabimo vgnezdene zanke, kot to prikazuje naslednji primer:

```
>> for i=1:2,
      for j=1:3
       H(i,j) = 1/(i+j-1);
      end
  end
>> H

H =

    1.0000    0.5000    0.3333
    0.5000    0.3333    0.2500
```

Dobili smo Hilbertovo matriko reda 2×3. Zaradi interpretrske narave Matlab po ukazu *for* čaka z izvedbo vnešenih ukazov, dokler ne vnesemo ukaza *end*, ki se nanaša na prvi ukaz *for* (prejšnji primer se zaključi z drugim ukazom *end* in šele takrat Matlab začne z izvajanjem obeh zank). Na ta način lahko enostavne primere napišemo kar v ukazno vrstico.

Za zahtevnejše funkcije je zaradi optimalne izrabe računalniškega časa bolj smiselno uporabljati vektorsko verzijo *for*–zank - običajne *for*–zanke so namreč časovno zelo potratne. Navedimo samo preprost primer za izračun funkcije *sin* v 20001 točki. S funkcijama *tic* in *toc* izmerimo čas, ki je pretekel od začetka do konca izvajanja operacije.

```
>> tic, x = 0:0.01:200; y = sin(x); toc
Elapsed time is 0.010000 seconds.
>> tic, for x = 1:20001, y(x) = sin((x-1)/100); end, toc
Elapsed time is 0.060000 seconds.
```

V drugem primeru je procesor Pentium 4 2 GHz porabil kar šestkrat več časa, če pa bi opustili podpičje v zanki (omogočili sprotni prikaz na zaslon), pa bi bila časa povsem neprimerljiva (v korist izvedbe z vektorjem).

4.7 Stavek *if*

Splošna oblika za ukaz *if...end* je

```
if relacija1
      izrazi1
elseif relacija2
      izrazi2
else
      izrazi3
end
```

Izrazi bodo izvršeni tedaj, ko bodo imeli vsi deli izraza *relacija1* nenegativne logične vrednosti. Deli *elseif* in *else* so opcionalni. Lahko uporabimo poljubno število izrazov *elseif*, izraz *else* pa je lahko samo eden in mora stati na koncu celotnega izraza. Prav tako pa je možna tudi uporaba vgnezdenih izrazov *if*. Relacije so največkrat v obliki

`(izraz r_op izraz) l_op ... l_op (izraz r_op izraz),`

kjer `r_op` označuje relacijski, `l_op` pa logični operator. Za primer si oglejmo, kako izvedemo funkcijo nasičenja. Vzemimo vektor **x**. Če je element iz **x** večji od 10 ali manjši od –10, naj ima istoležni element v izhodnem vektorju **y** vrednost 10 ali –10.

```
>> x=-30:5:30;
>> x_max=10; x_min=-10;
>> if x > x_max
y=x_max;
elseif x < x_min
y=x_min;
else
y=x;
end
```

Rezultat prikazuje slika 4.3.

4.8 Stavek *while*

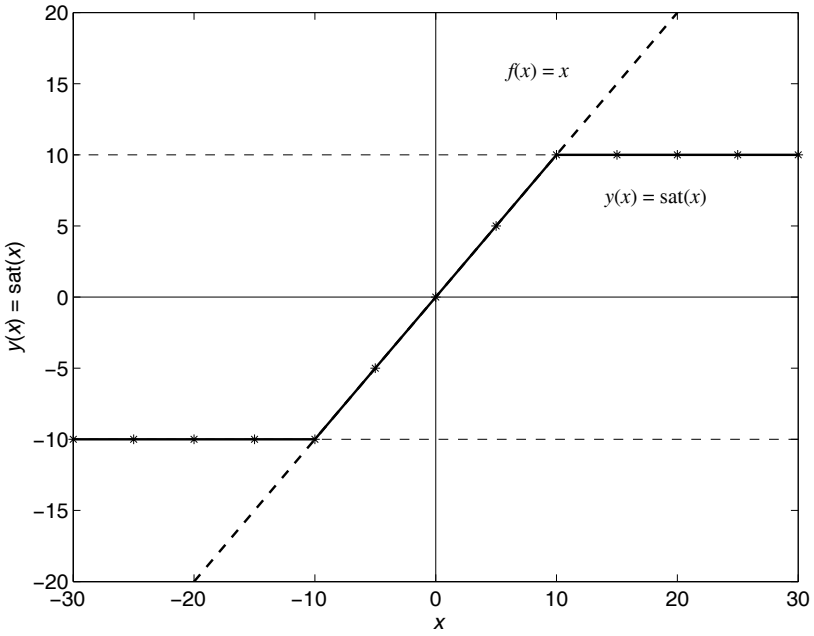
Zanka *while...end* se uporabi takrat, kadar ne vemo natančnega števila korakov izvajanja izrazov znotraj zanke. Splošna oblika je:

```
while relacija
       izrazi
end
```

Tudi tu so relacije največkrat oblike

`(izraz r_op izraz) l_op ... l_op (izraz r_op izraz),`

kjer `r_op` označuje relacijski, `l_op` pa logični operator. Izrazi se bodo izvajali toliko časa, dokler bo imela `relacija` logično vrednost 1.



Slika 4.3 - Funkcija nasičenja

Poglejmo si primer. Za dano število *a* bodo naslednji izrazi izračunali in prikazali najmanjšo nenegativno celo število *n*, tako da bo $2^n < a$:

```
>> a = 30;
>> n = 0;
>> while 2^n < a
n = n + 1;
end

>> n

n =

     5
```

Med izvajanjem zanke lahko vplivamo tudi na sam potek izvajanja z ukazoma *continue* in *break*. Prvi prekine izvajanje trenutnih izrazov in začne z novo iteracijo, drugi pa povzroči takojšen izstop iz zanke. Poglejmo si primera:

- Pri danem **a** izločimo samo pozitivne elemente in jih zapišimo v novi vektor **b**.

```
>> a = [-1.1878 -2.2023 0.9863 -0.5186 0.3274 0.2341 ...
0.0215 -1.0039 -0.9471 -0.3744];
>> i = 0; b = [];
>> while i<length(a)
i = i + 1;
if a(i)<0, continue, end
b = [b a(i)];
end
```

```
>> b

b =

    0.9863    0.3274    0.2341    0.0215
```

- Hočemo poiskati kvadratni koren elementov danega vektorja **c**. Ob tem pa nočemo koreniti negativnega števila – če naletimo na negativno število, takoj prekinemo z izvajanjem in javimo napako.

```
>> c = [3.4 2.5 4.7 9.8 5.6 -3.4 4.5 9.8];
>> i = 0;
>> while i<length(c)
i = i + 1;
if(c(i)<0), disp('Ne morem koreniti števila!'), break, end;
sqrt(c(i));
end
```

Ko poženemo program, pri šesti iteraciji Matlab naleti na negativno število, izpiše naše opozorilo in prekine z izvajanjem zanke. Poglejmo še vrednost spremenljivke *i*.

```
Ne morem koreniti števila!
>> i

i =

     6
```

Oba ukaza se lahko enakovredno uporabljata tudi v zanki *for*.

4.9 Stavek *switch*

Stavek *switch* uporabimo, kadar hočemo na neki točki v programu definirati več akcij naenkrat pri istem naboru podatkov, izbiro pa določimo s pogojnim izrazom. Osnovna oblika ukaza *switch...end* je:

```
switch s_izraz
      case c_izraz1,
            izrazi
      case c_izraz2
            izrazi
      ...
      otherwise,
            izrazi
end
```

Program bo izvršil prvi izraz *case*, pri katerem *c_izraz* ustreza vrednosti, ki jo ima *s_izraz*. Izvrševanje se nadaljuje z izrazom, ki sledi ukazu *end*. Če noben *c_izraz* nima prave vrednosti, potem se izvede izraz, podan v *otherwise*, seveda le, če smo ga podali. Izraz *s_izraz* je lahko številska spremenljivka ali niz znakov. Pogoj, ki se

izvrši pri vsaki vrstici *case*, je za številsko spremenljivko `s_izraz == c_izraz`, za niz pa `strcmp(s_izraz, c_izraz)` , ki vrne vrednost 1, če sta niza ista. Poglejmo še primer, kjer lahko z nastavljanjem spremenljivke *izb* odločamo o tem, katero operacijo hočemo izvesti nad matriko **A** iz enačbe (4.4).

```
>> izb = 3;
>> switch izb
case 1,
eig(A)
case 2,
norm(A)
case 3,
det(A)
otherwise
disp('Napačna izbira!')
end

ans =

   -0.1766
```

Ker je vrednost pogoja 3, je program izvedel tretjo funkcijo in vrnil determinanto matrike **A**. Poglejmo še, kako bi izvedli isti primer, če je spremenljivka *izb* niz:

```
>> izb = 'eig';
switch izb
case 'eig',
eig(A)
case 'norm',
norm(A)
case 'det',
det(A)
otherwise
disp('Napačna izbira!')
end

ans =

  20.8845
  -0.4172
   0.1164 + 0.0820i
   0.1164 - 0.0820i
```

5. M–DATOTEKE

Veliko uporabno vrednost Matlaba spoznamo, ko sekvenco ukazov, ki jih želimo izvesti, shranimo v datoteko s končnico *.m. Zaradi končnice jo imenujemo **m–datoteka**. Urejamo jih s pomočjo urejevalnika besedil, poganjamo pa jih iz ukazne vrstice (navedba imena datoteke) ali kar iz urejevalnika s pritiskom na tipko F5. Pri delu z Matlabom se velikokrat srečamo z oblikovanjem in popravljanjem sklopov ukazov ali pa pisanjem novih funkcij, zato je dobro poznavanje dela z m–datotekami pravzaprav nuja, ker nam prihrani veliko časa pri programiranju obsežnejših aplikacij.

Ločimo dva tipa m-datotek:

- **ukazne** ali **skriptne** datoteke, ki vsebujejo samo sekvence ukazov oz. vpisov, ne sprejemajo vhodnih in ne vračajo izhodnih parametrov, operirajo pa na podatkih v delovnem pomnilniku.
- **funkcijske** datoteke, ki sprejemajo vhodne parametre in vrnejo izhodne parametre, delujejo pa kot nove funkcije. Posebnost so interne spremenljivke, katere uporablja samo funkcija.

5.1 Ukazne m–datoteke

Ukazna m–datoteka je enostavna sekvenca ukazov, katero bi lahko vnesli tudi neposredno v ukazno vrstico. Če je sekvenca dolga ali se ponavlja večkrat, je bolj smiselno, da jo zapišemo v datoteko. Ukazne datoteke uporabljajo obstoječe podatke v delovnem pomnilniku ali pa naredijo nove podatke, na katerih potem izvajajo operacije. Kljub temu da ne vračajo izhodnih argumentov, se vse spremenljivke, ki so rezultat izvajanja ukazov v datoteki, shranijo v delovni pomnilnik in so nato na voljo drugim ukazom.

Kot primer si oglejmo ukazno m-datoteko za kvadratični populacijski model, podan z diferenčno enačbo

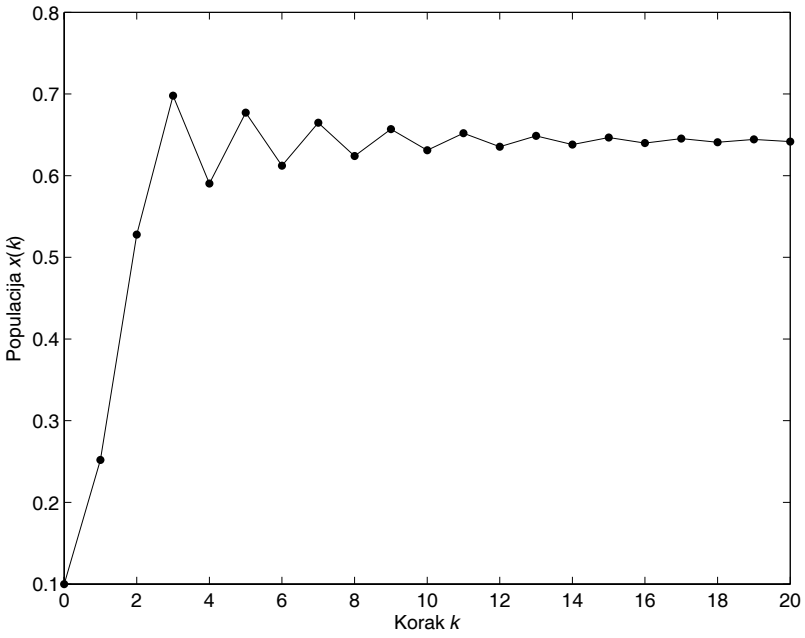
$$x(k+1) = \alpha x(k)\big(1 - x(k)\big), \tag{5.1}$$

kjer $x(k)$ predstavlja populacijo v k-tem trenutku. Datoteko z imenom *popmod.m* smo shranili v delovno mapo. Za zagon iz ukazne vrstice je potrebno vnesti ukaz *popmod*.

```
% popmod.m
% populacijski model, primer ukazne m-datoteke
%
x = [];
k = [];
time = [];
n = input('Vnesi stevilo korakov racunanja! ');
alfa = input('Vnesi parameter alfa  ');
xinit = input('Vnesi zacetno populacijo! ');
x(1) = xinit;
time(1) = 0;
for k = 2:(n+1)
```

```
    time(k) = k-1;
    x(k) = alfa*x(k-1)*(1-x(k-1));
end
plot(time,x,time,x,'.','markersize',15)
xlabel('Korak \it{k}','FontSize',12)
ylabel('Populacija \it{x}\rm{(}\it{k}\rm{)}','FontSize',12)
set(gca,'FontSize',12)
% konec ukazne m-datoteke
```

Oglejmo si dele datoteke. Prve tri vrstice imajo na začetku znak "%" za komentiranje ukazov. Teksta, ki sledi temu znaku do prvega znaka za prehod v novo vrstico, Matlab ne izvaja. V drugih treh vrsticah definiramo spremenljivke *x*, *k* in *time* kot prazne matrike. S tem se izognemo težavam, ki bi lahko nastale, če bi bile istoimenske spremenljivke že definirane v pomnilniku. V vrsticah 7, 8 in 9 s pomočjo funkcije *input* izvedemo uporabniški vnos želenih parametrov *n*, *alfa* in *xinit*. Nato spremenljivkama *x* in *time* priredimo začetne vrednosti in izvedemo zanko *for*, ki izračuna vrednosti po enačbi (5.1). Na koncu rezultat še narišemo s funkcijo *plot*, katere poglobljeno uporabo bomo spoznali kasneje. Podajmo rezultat za začetne vrednosti $n = 20$, $\alpha = 2.8$ in $xinit = 0.1$ – prikazuje ga slika 5.1.



Slika 5.1 - Populacijski model

5.2 Funkcijske m–datoteke

Eno najmočnejših orožij Matlaba je enostavno tvorjenje funkcijskih m–datotek. Tako napisana datoteka deluje kot nova Matlabova funkcija. Definiramo jo za poljubno veliko vhodnih in izhodnih argumentov, znotraj funkcije pa uporabljamo že obstoječe funkcije in tudi podprograme, kot smo to vajeni iz višjih programskih jezikov.

Začetnik v programiranju z Matlabom naj napiše novo funkcijo, ki jo hoče uporabljati v trenutni delovni mapi, in jo tja tudi shrani. Izkušeni uporabniki pa si lahko na ta način zgradijo obsežne knjižnic funkcij oz. nova orodja, ki jih dodajo v Matlabovo iskalno pot (ang. path) in uporabljajo iz katerekoli delovne mape.

Predelajmo populacijski model iz prejšnjega primera v funkcijo, ki ima tri vhodne argumente in dva izhodna argumenta, niso pa vsi nujni za izvajanje funkcije.

```
function [pop,kor] = pmod(alfa,xinit,n)
%   PMOD Kvadratični populacijski model
%   function [pop,t] = pmod(alpha,xinit,n)
%
%   Vrne vrednost populacije v n korakih
%
%   alfa = faktor hitrosti množenja populacije
%   xinit = začetna vrednosti populacije
%   n (opcionalno) = število korakov računanja, privzeto n=20
%   pop (opcionalno) = populacija v korakih računanja
%   kor (opcionalno) = koraki računanja
%
%   Če noben od izhodnih argumentov ni definiran, funkcija
%   samo izriše graf
%   Igor Škrjanc, Simon Oblak (27.10.1998 in 28.11.2004)
x = []; k = [];
time = []; t = [];
x(1) = xinit;
time(1) = 0;
if nargin<3
   n_end = 20;
else
   n_end = n;
end
for k = 2:n_end+1
   time(k) = k-1;
   x(k) = alfa*x(k-1)*(1-x(k-1));
end
if nargout==2
   kor = time;
   pop = x;
elseif nargout==1
   pop = x;
else
   figure
   plot(time,x,time,x,'.','markersize',15)
   xlabel('Korak \it{k}','FontSize',13)
   ylabel('Populacija \it{x}\rm{(}\it{k}\rm{)}','FontSize',13)
   set(gca,'FontSize',13)
end
% konec funkcije
```

Program shranimo v funkcijsko datoteko pod imenom *pmod.m*. Poglejmo si njene glavne dele. Prva vrstica definira klic funkcije z vsemi vhodnimi in izhodnimi

argumenti. Beseda *function* pove interpreterju, da ima opraviti s funkcijsko datoteko. Izhodni parametri so v oglatih oklepajih (če ima funkcija samo en izhodni parameter, ne uporabimo oklepajev), vhodni pa v okroglih. Beseda *pmod* definira ime funkcije, katerega kličemo v ukazni vrstici, in mora biti enako imenu datoteke, v kateri je funkcija shranjena. Sledi del, namenjen pomoči pri uporabi funkcije. Prva vrstica je ti. *H1-vrstica*, katero uporabljamo pri iskanju z ukazom *lookfor*. Ostale vrstice, ki se začnejo z znakom %, se izpišejo pri zagonu ukaza *help ime_funkcije*. Poglejmo, kaj se zgodi, če hočemo pomoč za našo novo funkcijo:

```
>> lookfor pmod
   PMOD Kvadratični populacijski model

>> help pmod

    PMOD Kvadratični populacijski model
    function [pop,t] = pmod(alpha,xinit,n)

    Vrne vrednost populacije v n korakih

    alfa = faktor hitrosti množenja populacije
    xinit = začetna vrednosti populacije
    n (opcionalno) = število korakov računanja, privzeto n=20
    pop (opcionalno) = populacija v korakih računanja
    kor (opcionalno) = koraki računanja

    Če noben od izhodnih argumentov ni definiran, funkcija
    samo izriše graf
    Igor Škrjanc, Simon Oblak (27.10.1998 in 28.11.2004)
```

Inicializacijski del je zapisan med vrstico `x = [];` in začetkom zanke *for*. Vidimo, da se tu precej razlikuje od prejšnjega primera. Začetnim spremenljivkam se priredijo vrednosti, podane z vhodnimi argumenti. Spremenljivke *alfa*, *xinit*, *n*, *x*, *k*, *time*, *t*, *pop* in *kor* so definirane kot **lokalne spremenljivke** in nimajo nobene povezave z ostalimi spremenljivkami delovnega pomnilnika. To pomeni, da po koncu izvajanja funkcije ne bodo več na voljo uporabniku. Novost je funkcija *nargin* v stavku *if*, ki vrne število vnešenih parametrov s strani uporabnika. Spremenljivki *nargin* in *nargout* sta dosegljivi znotraj izvajanja vsake funkcije in se generirata takoj po klicu funkcije. Tukaj smo s pogojem `if nargin<3` dosegli, da se začetni vrednosti pripiše privzeta vrednost $n = 20$, če v klicu funkcije nismo podali tretjega parametra. Računanje populacije je izvedeno na enak način kot v ukazni datoteki, pozorni pa moramo biti na določanje izhodnih vrednosti. Če smo zahtevali dve izhodni vrednosti, funkcija vrne tako vektor populacije kot vektor korakov računanja, če smo zahtevali samo eno izhodno vrednost, funkcija vrne samo vektor populacije, če pa smo izvedli klic funkcije brez izhodnih parametrov, nariše enak graf kot v prvem primeru. Izhodne vrednosti dobi funkcija iz lokalnih spremenljivk in jih priredi spremenljivkam, ki jih podamo v klicu funkcije. Poglejmo si vse primere uporabe.

- Za začetek hočemo vrednosti populacije v prvih šestih korakih računanja pri parametrih $\alpha = 2.8$ in $xinit = 0.1$. Rezultat zapišimo v spremenljivki *popul* in *korak*:

```
>> [popul,korak] = pmod(2.8,0.1,6)

popul =

  0.1000  0.2520  0.5278  0.6978  0.5904  0.6771  0.6122

korak =

     0     1     2     3     4     5     6
```

- Pri istih prvih dveh vhodnih parametrih zapišimo v vektor *popul* vrednosti populacije v petih korakih:

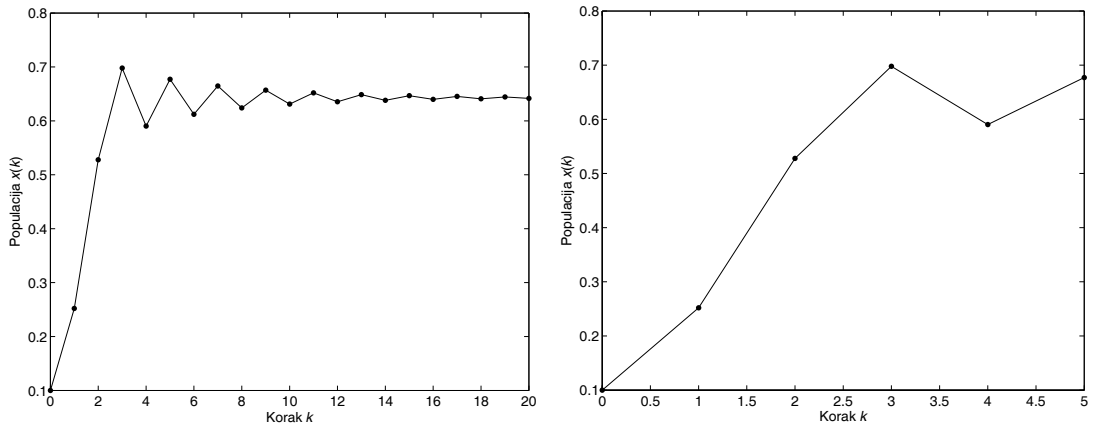
```
>> popul = pmod(2.8,0.1,5)

popul =

   0.1000   0.2520   0.5278   0.6978   0.5904   0.6771
```

- Zdaj pa primerjajmo, kaj se zgodi ob klicu funkcije brez izhodnih parametrov, enkrat s tremi vhodnimi parametri, drugič pa izpustimo zadnjega:

```
>> pmod(2.8,0.1)
>> pmod(2.8,0.1,5)
```

Rezultat sta diagrama, katera prikazuje slika 5.2. Vidimo, da se ob prvem klicu izračuna privzeto število korakov (20), v drugem pa podanih 5 korakov.

Če želimo iz ukazne vrstice pogledati, kako deluje naša funkcija, vtipkamo ukaz `type pmod` in vsebina datoteke »pmod.m« se v celoti izpiše na ekran.



Slika 5.2 - Diagrama populacije; levo - pmod(2.8,0.1), desno - pmod(2.8,0.1,5)

5.3 Funkcije kot podprogrami

Matlab omogoča tudi definicijo in izvedbo funkcij znotraj neke funkcije. Taka podfunkcija je na voljo samo delom znotraj iste datoteke, prav tako pa ni vidna zunaj datoteke, kjer je napisana, tako da je z ukazi pomoči (*help*, *lookfor*) ne moremo zaslediti. Za primer si poglejmo funkcijo *stat.m*, ki izračuna povprečno vrednost vhodnega vektorja, po potrebi pa še standardno deviacijo in varianco.

```
function [povpr,stdev,var] = stat(x)
% STAT statistika vektorja x
% function [povpr,stdev,var] = stat(x)
% x = vhodni vektor
% povpr = povprečna vrednost vhodnega vektorja
% stdev (opcionalno) = standardna deviacija vhodnega vektorja
% var (opcionalno) = varianca vhodnega vektorja
% Simon Oblak, 29.11.2004
n = length(x);
povpr = povpreci(x,n);
if nargout==2
    stdev = sqrt(sum((x-povpreci(x,n)).^2)/n);
elseif nargout==3
    var = sum((x-povpreci(x,n)).^2)/n;
    stdev = sqrt(var);
end

function avg = povpreci(x,n)
avg = sum(x)/n;
%
% Konec funkcije
```

V funkciji se za izračun srednje vrednosti uporablja klic `povpreci(x,n)`. Funkcija *povpreci* je definirana na koncu funkcije *stat* kot samostojna funkcija. Preizkusimo dva od možnih klicev funkcije *stat*:

```
>> x =[0.95 0.23 0.61 0.48 0.89 0.76 0.46 0.02 0.82 0.44];
>> stat(x)

ans =

    0.5660

>> [a,b,c] = stat(x)

a =

    0.5660

b =

    0.2832
```

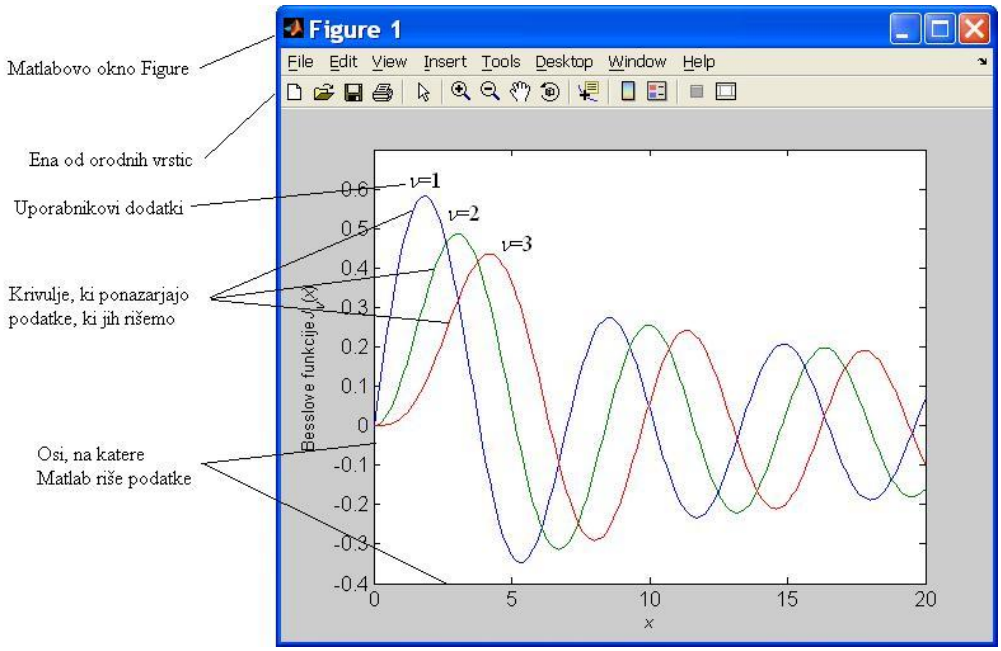
```
c =

    0.0802
```

V prvem delu se izhodna vrednost priredi spremenljivki *ans*, ker nismo definirali akcije za `nargin==0`. V drugem pa se izračunajo vse tri vrednosti.

6. GRAFIČNI PRIKAZ

Grafični vmesnik je eden tistih, ki je v novi verziji doživel največ sprememb, zato si ga bomo podrobneje ogledali. Najprej pa razložimo, kako sploh deluje Matlabov grafični prikazovalnik.

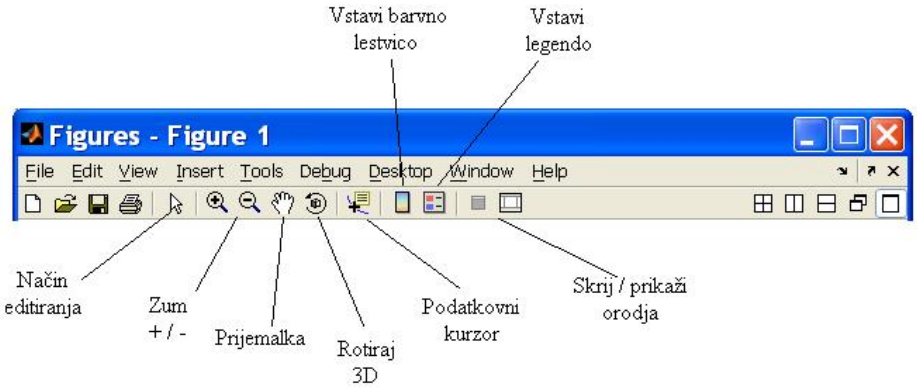
Orodja in funkcije za grafični prikaz svoj izhod usmerijo v posebno okno, ki je ločeno od ukaznega okna. Označeno je z besedo *Figure*. Glavne dele si lahko ogledamo na sliki 6.1.



Slika 6.1 - Okno Figure

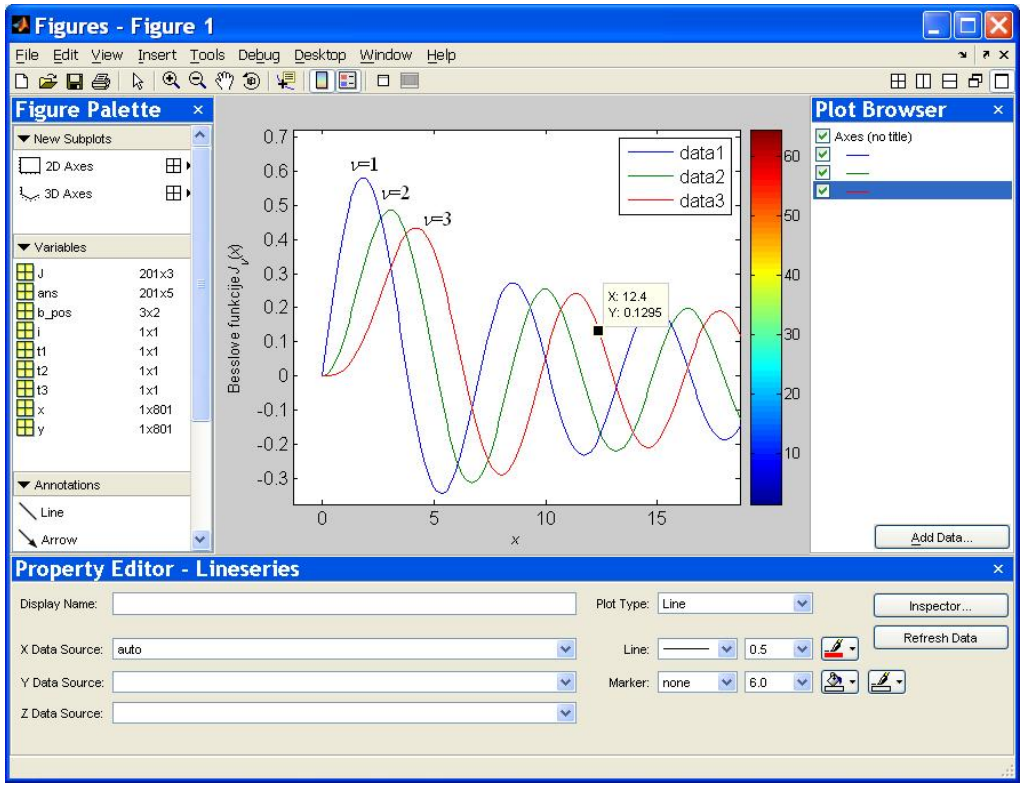
Slika prikazuje standardni dvodimenzionalni diagram podatkov v oknu z imenom *Figure 1*. Osi so izbrane glede na podatke, ki jih prikazujemo – v tem primeru je interval abscise **x** = [0, 20] in ordinate **y** = [–0.4, 0.7]. Razpon osi je vedno takšen, da se prikažejo vsi podatki, lahko pa ga spreminjamo tudi sami. V oknu je prostor tudi za dodatke s strani uporabnika, kot so črte, puščice, tekst (primer na sliki), legenda, barvna jakostna lestvica itd. Novost verzije 7 je v tem, da so nam vse omenjene funkcije na voljo v menijih in osnovni orodni vrstici. Pomen ikon orodne vrstice je predstavljen na sliki 6.2.

Način editiranja nam omogoči, da s pomočjo miške izberemo vsakega od objektov na diagramu in dostopamo do njegovih lastnosti v oknu, ki se nam odpre. *Zum* + / - nam poveča/pomanjša označeni del slike, s *Prijemalko* lahko sliko premikamo po diagramu (dobesedno: spreminjamo območje obeh koordinat), *Rotiraj 3D* najprej pretvori prikaz v tridimenzionalni diagram in nam omogoči vrtenje okoli poljubne osi, s *Podatkovnim kurzorjem* lahko označimo določeno točko na grafu in preberemo njene vrednosti, barvno lestvico in legendo pa vstavimo/zbrišemo z ustreznima ikonama.



Slika 6.2 - Ikone orodne vrstice

Zadnja ikona nas popelje v interaktivni način, kjer so nam na voljo orodja za napredno urejanje diagramov. Slika 6.3 prikazuje izgled diagrama po pritisku na tipko *Skrij / prikaži orodja*.



Slika 6.3 - Interaktivni urejevalnik grafov

Interaktivni urejevalnik nam omogoča izvajanje naslednjih opravil:

- kreiranje različnih vrst diagramov
- izbira spremenljivk za prikaz neposredno iz delovnega prostora
- enostavno kreiranje in urejanje poddiagramov (ang. subplot)
- dodajanje oznak, kot so puščice, črte in tekst
- nastavljanje lastnosti grafičnih objektov

V oknu *Figure Palette* lahko izbiramo vrsto prikaza (2D, 3D), spremenljivke, katere želimo prikazati, in oznake, katere želimo dodati v okno diagrama. V glavnem oknu diagrama lahko izberemo katero od krivulj (če jih je seveda več) in njene parametre urejamo v oknu *Property Editor*. V oknu *Plot Browser* pa izbiramo med serijami zaporednih prikazov, ki so trenutno v glavnem oknu. Ob tem pa nam Matlab vse interaktivno izvedene funkcije pošlje še v obliki ukazov v delovni prostor, tako da so primerni za shranjevanje ali pisanje v m-datoteko.

Zavedati pa se moramo, da je glede ponovne uporabe in urejanja diagrame veliko enostavneje oblikovati kar s pomočjo ukaznih m-datotek, še posebej, če gre za večje serije podatkov in poskusov. Zato si bomo podrobno razlago funkcij urejanja grafičnega prikaza ogledali na primerih sekvenc ukazov, ki jih pišemo v ukazno vrstico.

6.1 Dvodimenzionalni diagrami

Kot smo že videli v nekaterih prejšnjih primerih, lahko z ukazom *plot* narišemo linearni x-y diagram dveh vektorjev. Če sta x in y vektorja enakih dolžin, tedaj ukaz `plot(x,y)` odpre grafično okno, v katerem se nariše diagram z elementi vektorja x na abscisi in elementi vektorja y na ordinati.

Ob razlagi grafičnega vmesnika na slikah 6.1-6.3 smo uporabili diagram treh krivulj $J_v(x)$. Krivulje predstavljajo Besselove funkcije za $v = 1, 2, 3$ na intervalu $0 \leq x \leq 20$. Podatke smo zapisali v spremenljivki x in J po izvedbi naslednjih ukazov:

```
>> x=(0:.1:20)';
>> J = besselj(1:3,x);
```

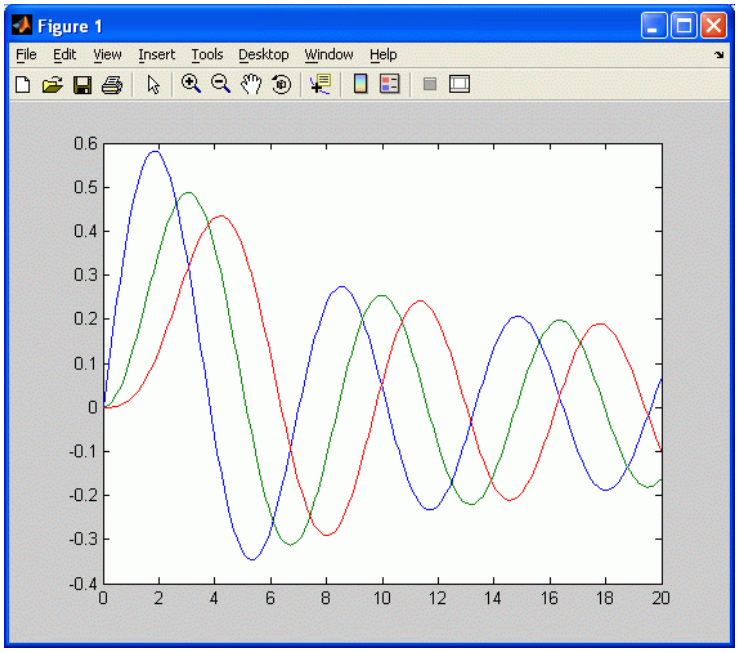
Spremenljivka **x** je stolpni vektor z 201 elementom, matrika **J** pa je dimenzije 201×3. Dolžina stolpcev se torej ujema z dolžino vektorja **x**. Zdaj bomo po korakih opisali glavne funkcije, ki nas pripeljejo do zgledno urejenega in označenega diagrama. Začnimo torej z ukazom *plot*:

```
>> plot(x,J)
```

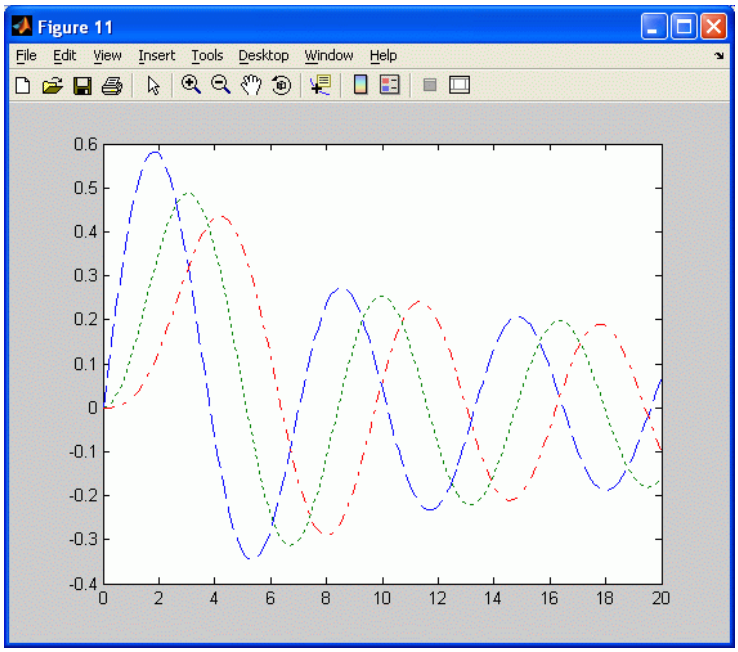
Na sliki 6.4 je prikazano okno *Figure 1*, ki se avtomatsko odpre ob klicu tega ukaza. Številka 1 predstavlja parameter, po katerem ločimo diagrame med seboj, če jih je odprtih več naenkrat. Če hočemo narediti okno z zaporedno številko 11, pred ukazom *plot* izvedemo ukaz `figure(11)`. Na sliki 6.4 vidimo, da so se naenkrat izrisali vsi trije stolpci matrike **J**. Dejansko Matlab ustvari 3 krivulje, jih nariše v diagram z istimi koordinatnimi osmi in jih kot objekte po sistemu *roditelj–otrok* pripiše diagramu. Vsak od objektov pa ima svoje parametre, katere lahko tudi spreminjamo. V tem delu si bomo ogledali samo nekatere od njih.

Denimo, da želimo vsako krivuljo predstaviti z drugačnim tipom črte. Zapišimo ukaz, ki bo prvo krivuljo narisal črtkano, drugo pikčasto in tretjo v načinu črta–pika (slika 6.5). Vse skupaj podamo z enim samim ukazom *plot*, v katerega lahko vključimo poljubno število parov spremenljivk z dodanimi parametri.

```
>> figure(11)
>> plot(x,J(:,1),'--',x,J(:,2),':',x,J(:,3),'-.')
```



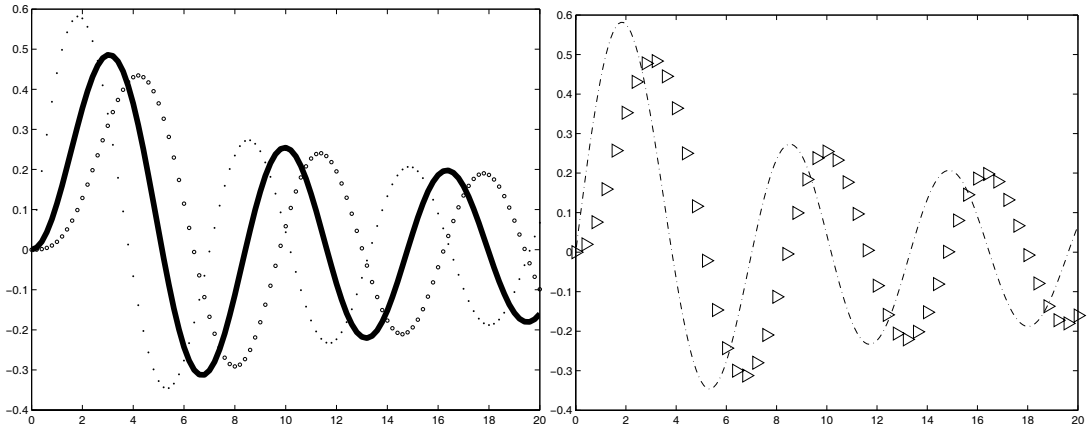
Slika 6.4 - Prikaz Besslovih funkcij – osnovni prikaz



Slika 6.5 - Prikaz Besslovih funkcij – trije tipi krivulj

Tretji način prikazovanja več krivulj naenkrat na istem diagramu je z uporabo ukazov *hold on* / *off*. Privzeti način risanja je »zamenjava« (ang. replace) – ob naslednjem klicu funkcije *plot* se vsi predhodnji prikazi izbrišejo in »naredijo prostor« novemu. Z ukazom *hold on* se temu izognemo in do preklica lahko dodamo poljubno število krivulj. Pri naslednjem primeru privzemimo, da hočemo drugo funkcijo še posebej poudariti z debelejšo črto, medtem ko preostali dve prikažemo samo s 50 točkami in 50 krogci. Rezultat je prikazan na sliki 6.6 levo.

```
>> figure
>> plot(x(1:2:end),J(1:2:end,1),'.','markersize',5)
>> hold on
>> plot(x(1:2:end),J(1:2:end,2),'-','linewidth',5)
>> plot(x(1:2:end),J(1:2:end,3),'o','markersize',3)
>> hold off
```



Slika 6.6 - Predstavitev različnih tipov izrisov krivulj

Pozorni moramo biti, da imena parametrov in tekstovne argumente vnašamo v navednicah, razen številčnih vrednosti, kot na primer `... 'linewidth',5...` Če hočemo elemente vektorjev prikazati z znaki (1. in 3. funkcija), imamo na voljo

'+'	plus	'v'	trikotnik (dol)
'x'	križec	'^'	trikotnik (gor)
'.'	točka	'>'	trikotnik (desno)
'x'	križec	'<'	trikotnik (levo)
'*'	zvezdica	'p'	petkotnik
's'	kvadrat	'h'	šestkotnik
'd'	karo		

Poleg tega z dodatnim parametrom *markersize* določimo velikost znaka. Krivulje lahko tudi obarvamo. Po enakem principu kot pri navedbi spremenljivk dodamo parameter v navednicah (lahko tudi v kombinaciji z znakom ali tipom črte):

'b'	modra	'm'	magenta
'g'	zelena	'y'	rumena
'r'	rdeča	'k'	črna
'c'	cian		

Uporabo kombinacije parametrov ponazorimo z ukazom

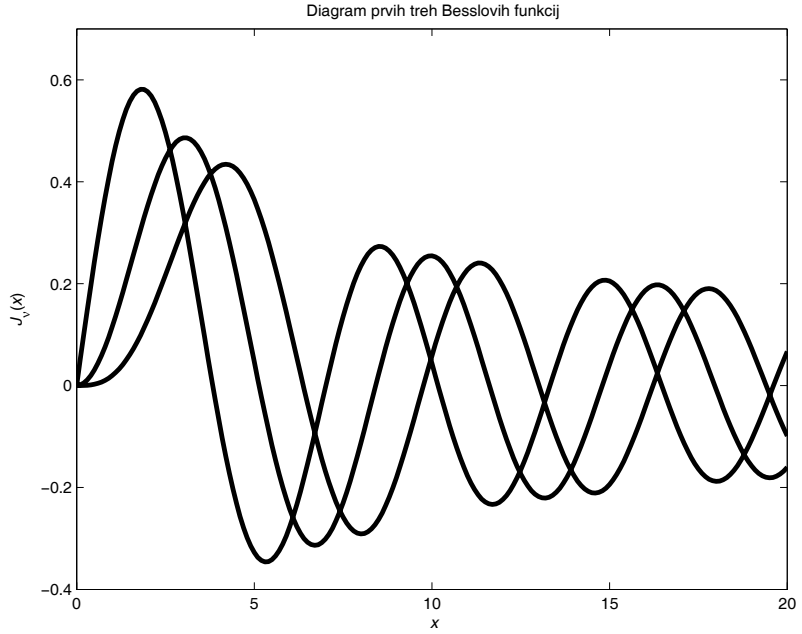
```
>> plot(x,J(:,1),'c-.',x(1:4:end),J(1:4:end,2),'k>',...
'markersize',10)
```

in predstavimo na sliki 6.6 desno.

Zdaj pa se posvetimo skaliranju osi in oblikovanju oznak. Zamislimo si, da bi radi nad krivulje napisali vrednost parametra ν za vsako od njih. Zato bi potrebovali malo več prostega prostora na vrhu diagrama. To storimo tako, da z ukazom `axis([x_sp, x_zg, y_sp, y_zg])` podamo novi zgornji in spodnji meji obeh osi. Radi bi tudi označili imeni osi na diagramu in diagram naslovili. Omenjeno storimo z ukazi `xlabel('tekst')`, `ylabel('tekst')` in `title('tekst')`.

Ker zna Matlab interpretirati tudi kodo, napisano v jeziku TeX, lahko vključimo tudi grške črke in tekst oblikujemo z ukazi, kot npr. \it{} (ležeče). Vse spremembe, ki jih naredimo na oseh, se shranijo v spremenljivke objekta, do katerega dostopamo z ukazom *gca*. Ker bomo objektno programiranje spoznali kasneje, omenimo samo parametra *xtick* in *ytick* – številske oznake na oseh, katere lahko poljubno izberemo. Poglejmo si zdaj program, ki bo vključil opisane funkcije za oblikovanje osi diagrama:

```
>> figure(13)
>> plot(x,J,'-','linewidth',3)
>> axis([0 20 -0.4 0.7])
>> set(gca,'xtick',[0 5 10 15 20])
>> set(gca,'ytick',[-0.4 -0.2 0 0.2 0.4 0.6])
>> xlabel('{\it x}')
>> ylabel('{\itJ}_{\nu}({\itx})')
>> title('Diagram prvih treh Besslovih funkcij')
```



Slika 6.7 - Diagram s popravki in oznakami osi

Za konec pa upoštevajmo še naslednje predpostavke:

- označimo krivulje s parametrom
- vključimo legendo
- povečajmo velikost črk in spremenimo stil vseh oznak glede na sliko 6.7
- narišimo koordinatno mrežo

Za izvedbo prve alineje uporabimo ukaz `text(x,y,'...tekst...')`, kjer sta *x* in *y* koordinati, kamor bi želeli postaviti podano besedilo. Koordinate odčitamo iz prejšnjega diagrama – glede na osi – ali pa si pomagamo z ukazom `a = ginput(n)`, kjer *n* pomeni število točk, ki jih želimo odčitati. Ob izvedbi ukaza se izriše križ, katerega s pomočjo miške pozicioniramo po oknu diagrama. S pritiskom na levo tipko odčitamo trenutne koordinate glede na osi diagrama. Če je $n > 1$, potem

odčitamo več točk zaporedoma. Koordinate se zapišejo v spremenljivko **a**, ki je tako dimenzije $n \times 2$. Uporabimo jih lahko neposredno v klicu funkcije *text*, npr.

```
>> text(a(1,1),a(1,2),'J_1(x)')
```

Če pa spremenljivko **a** shranimo v mat-datoteko, potem nam ob ponovnem risanju istega diagrama ne bo potrebno znova odčitavati koordinat. Za sprotno izpisovanje teksta z miško pa je primeren tudi ukaz `gtext('...tekst...')`, ki izpiše '...tekst...' na mesto, kjer pritisnemo levo miškino tipko.

Legendo vključimo z ukazom

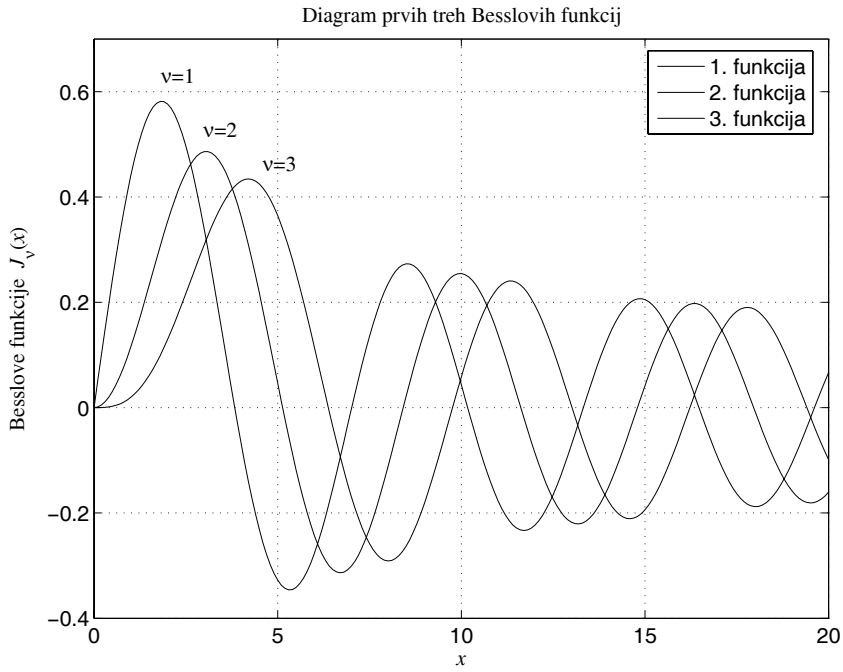
```
>> legend('Ime_1','Ime_2',...,'Ime_n',i),
```

kjer morajo imena slediti zaporedju vnosov krivulj, katere predstavljajo. Število *i* ponazarja kvadrant, v katerega naj se legenda izpiše – pri *i* = 1 bo mesto izpisa legende desni zgornji kot (prvi kvadrant).

Velikost črk določimo s parametrom `'FontSize'`. V argument ukaza enostavno dodamo omenjeni niz in številčno vrednost. Pisavo spremenimo s parametrom `'FontName'`, kateremu sledi znakovni niz v navednicah, npr. 'Times New Roman'. Koordinatno mrežo narišemo z ukazom `grid on`.

Poglejmo si zdaj končni izgled našega diagrama in program, ki stoji za tem.

```
figure(14)
% Funkcija
x = (0:.1:20)'; J = besselj(1:3,x);
plot(x,J)
% Oznake
xlabel('\it{x}','FontSize',13,'FontName','Times New Roman')
ylabel('Besslove funkcije  \it{J}_{\nu}\rm{(}\it{x}\rm{)}',...
'FontSize',13,'FontName','Times New Roman')
title('Diagram prvih treh Besslovih funkcij','FontSize',13,...
'FontName','Times New Roman')
% Spremembe osi
set(gca,'FontSize',13)
set(gca,'xtick',[0 5 10 15 20])
set(gca,'ytick',[-0.4 -0.2 0 0.2 0.4 0.6])
axis([0 20 -0.4 0.7])
% Ostalo
grid on
legend('1. funkcija','2. funkcija','3. funkcija',1)
% Tekstovne oznake
load b_pos
t1 = text(b_pos(1,1),b_pos(1,2),'\it{\nu}\rm=1')
t2 = text(b_pos(2,1),b_pos(2,2),'\it{\nu}\rm=2')
t3 = text(b_pos(3,1),b_pos(3,2),'\it{\nu}\rm=3')
set(t1,'FontName','Times New Roman','FontSize',13,...
'HorizontalAlignment','center')
set(t2,'FontName','Times New Roman','FontSize',13,...
'HorizontalAlignment','center')
set(t3,'FontName','Times New Roman','FontSize',13,...
'HorizontalAlignment','center')
```



Slika 6.8 - Končni izgled diagrama

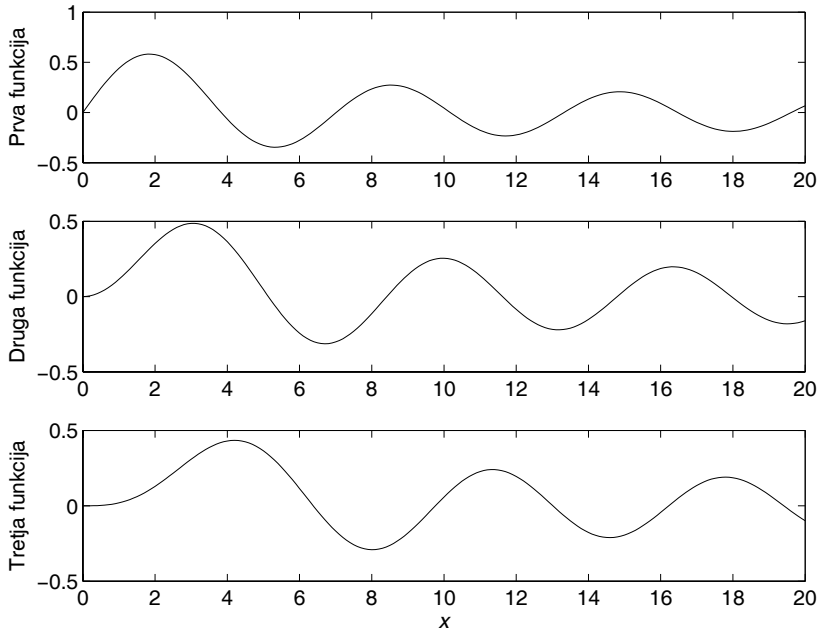
Vrednosti, zapisane v datoteki *b_pos.mat*, smo dobili z ukazom `b_pos = ginput(3)`. Dodaten parameter pri ukazu *text* je `'HorizontalAlignment'`, ki določi, kako se postavi tekst glede na točko, podano s koordinatama.

Če želimo prikazati več diagramov na eni sami sliki, uporabimo funkcijo *subplot*(*m*,*n*,*p*) (vejice lahko tudi izpustimo). S tem ukazom se diagram razdeli na prostor diagramov dimenzije $m \times n$, p pa pomeni zaporedno številko trenutnega aktivnega diagrama. Ponazorimo uporabo s prikazom vsake od Besslovih funkcij na svojem poddiagramu (slika 6.9).

```
subplot(311)
set(gca,'FontSize',13)
plot(x,J(:,1))
ylabel('Prva funkcija','FontSize',13)
subplot(312)
set(gca,'FontSize',13)
plot(x,J(:,2))
ylabel('Druga funkcija','FontSize',13)
subplot(313)
set(gca,'FontSize',13)
plot(x,J(:,3))
ylabel('Tretja funkcija','FontSize',13)
xlabel('\it{x}','FontSize',13)
```

6.2 Tridimenzionalni diagrami

Za funkcije dveh spremenljivk uporabimo enega od tridimenzionalnih prikazov, ki jih ponuja Matlab. Obravnavali bomo funkcije *plot3*, *mesh* in *surf* kot najbolj značilne predstavnice.



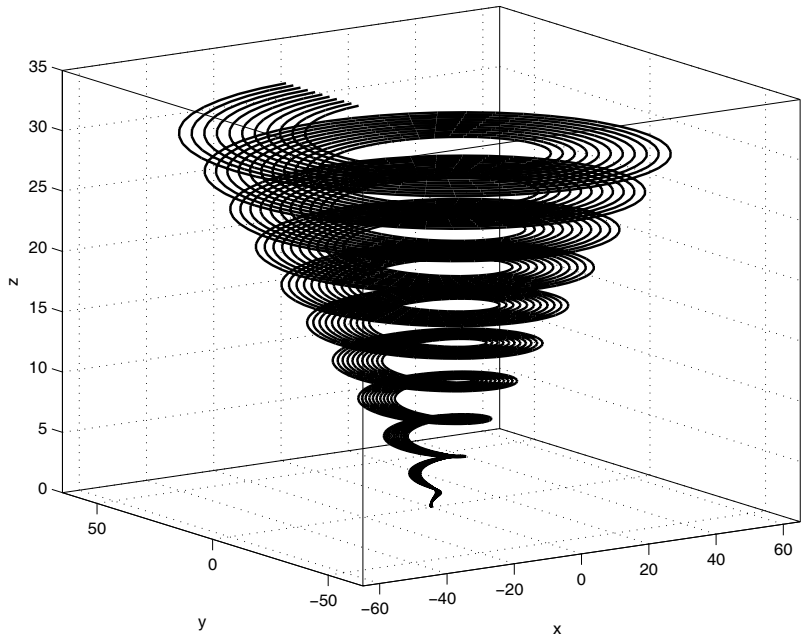
Slika 6.9 - Trije ločeni diagrami na eni sliki

Funkcija *plot3* je ekvivalent funkcije *plot* v treh dimenzijah. Uporabimo jo, če imamo na voljo trojico enako dolgih vektorjev, katerih elementi so koordinate točk funkcije dveh neodvisnih spremenljivk. Če kot vhodne argumente podamo tri matrike, funkcija nariše toliko krivulj, kot je stolpcev v matrikah. Za primer si poglejmo, kako bi v obrnjeni stožec narisali enajst spiral (slika 6.10).

```
figure
t = [0:pi/50:10*pi]';
plot3(t.*sin(2*t)*[1:0.1:2],t.*cos(2*t)*[1:0.1:2],...
t*ones(1,11),'linewidth',1.5);
axis([-65 65 -65 65 0 35]);
grid on, box on;
xlabel('x'), ylabel('y'), zlabel('z');
```

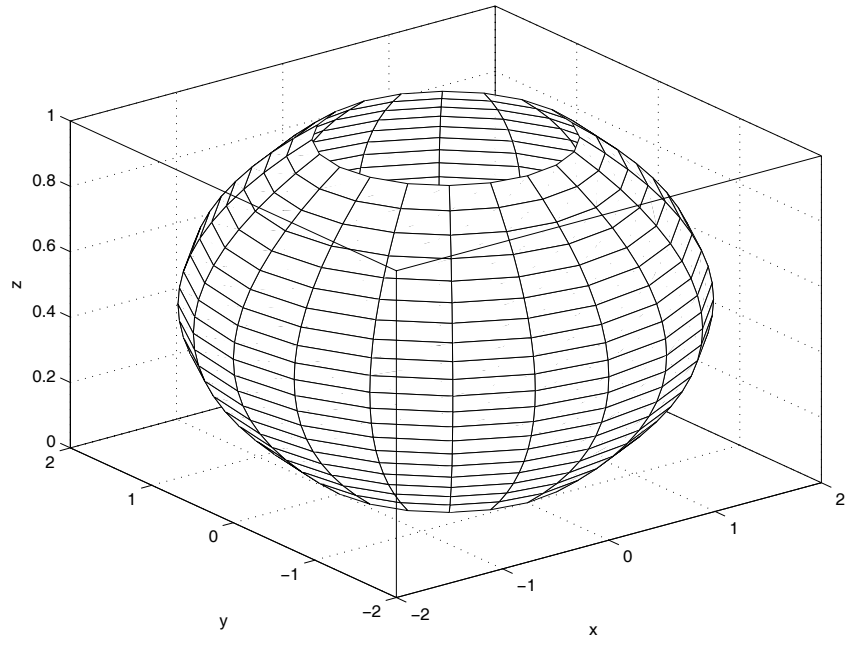
Vhodni argumenti so tu matrike z enajstimi stolpci in 501 vrstico.

Funkcija *mesh* nariše mrežo, podano s tremi matrikami koordinat (*X*, *Y*, *Z*) ali dvema vektorjema koordinat **x** in **y** ter matriko koordinat **Z**. V drugem primeru mora biti matrika dimenzij $m \times n$, kjer je *m* dimenzija vektorja **y**, *n* pa dimenzija vektorja **x**. Četrti vhodni parameter je matrika vrednosti barvne lestvice **C**. Če je ne podamo, se za intenzivnost barve vzame skalirane vrednosti matrike **Z**. Za primer narišimo pokončen cilinder, kjer stranski rob opiše funkcija $2 - t^2$, $-1 \le t \le 1$ (slika 6.11).



Slika 6.10 - Primer uporabe funkcije *plot3* za diagram funkcije dveh spremenljivk

```
figure
t =-1:0.1:1;
[x,y,z] = cylinder(2-t.^2);
mesh(x,y,z)
grid on, box on;
xlabel('x'), ylabel('y'), zlabel('z');
colormap gray
```

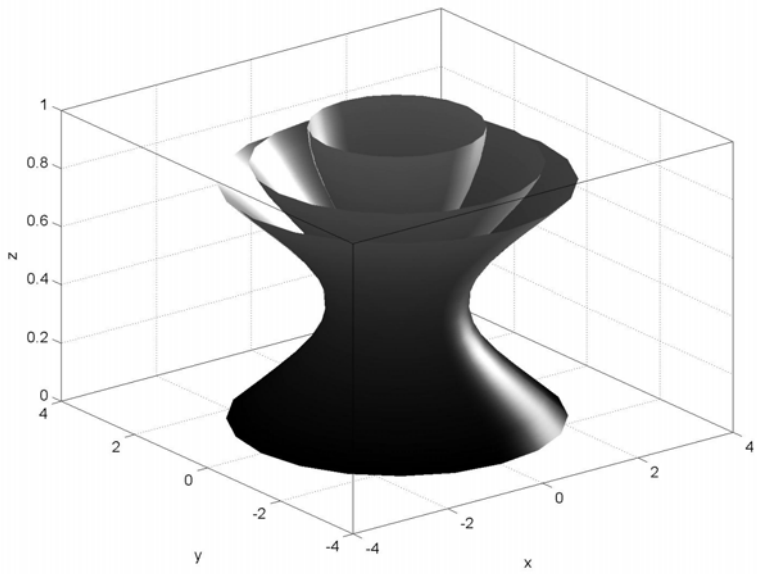


Slika 6.11 - Primer uporabe funkcije *mesh* za diagram cilindra

Ukaz `colormap gray` pretvori barvno lestvico v lestvico sivin od črne ([0 0 0]) do bele barve ([1 1 1]) v 256 korakih.

Poglejmo si še primer uporabe funkcije *surf*, ki nariše barvno mrežo z istimi vhodnimi argumenti kot pri funkciji *mesh*. Z ukazom *shading* lahko spremenimo vrsto senčenja površine – `shading faceted` je privzeta nastavitev (mreža), `shading flat` odpravi pokončne črte mreže, tako da dobimo nivoje senčenja glede na vertikalno os, `shading interp` pa interpolira nivoje po *z*-osi. Posebnost funkcije *surf* je tudi v tem, da lahko v diagram dodamo osvetlitev z ukazom *light*. Podati moramo koordinate namišljenega svetila in tip osvetlitve, kar naredimo z ukazom *lighting*. Na voljo imamo načine *none*, *flat*, *phong* in *gouraud*. Poglejmo si primer (slika 6.12).

```
figure
t = 3*pi/8:pi/10:2*pi;
[x,y,z] = cylinder(0.5*(2+cos(t)))
surf(x,y,z)
hold on
[x,y,z] = cylinder(0.9*(2+cos(t)))
surf(x([1:end-2],:),y([1:end-2],:),z([1:end-2],:))
[x,y,z] = cylinder(1.2*(2+cos(t)))
surf(x([1:end-3],:),y([1:end-3],:),z([1:end-3],:))
hold off
grid on, box on;
xlabel('x');
ylabel('y');
zlabel('z');
shading interp
colormap gray
light('Position',[1 0 0],'Style','infinite')
lighting phong
```



Slika 6.12 - Primer uporabe funkcije *surf* za diagram sestavljenega telesa

7. POGLOBLJENA UPORABA MATLABA

To poglavje je namenjeno spoznavanju s funkcijami, katerih se ne uporablja tako pogosto, pa vendar so pomembna orodja pri reševanju bolj zapletenih problemov. To so funkcije za analizo drugih funkcij (in imajo te funkcije za argumente), za reševanje sistemov diferencialnih enačb, numerično integracijo in optimizacijo. Na koncu pa sledi še nekaj uporabnih nasvetov za programiranje ukaznih m–datotek.

7.1 Definicija funkcij

Poleg pisanja funkcij v ukazne in funkcijske m–datoteke nam Matlab ponuja še dva načina podajanja funkcij v ukazni vrstici. Prvi je s pomočjo **kazalca na funkcijsko datoteko**. Denimo, da imamo napisano funkcijo *izracunaj* in da definiramo kazalec *fe* na omenjeno funkcijo.

```
function y = izracunaj(x)
%
y = 1-exp(-x.^2/9).*cos(2*x)
%
% Konec funkcije

>> fe = @izracunaj

fe =

    @izracunaj
```

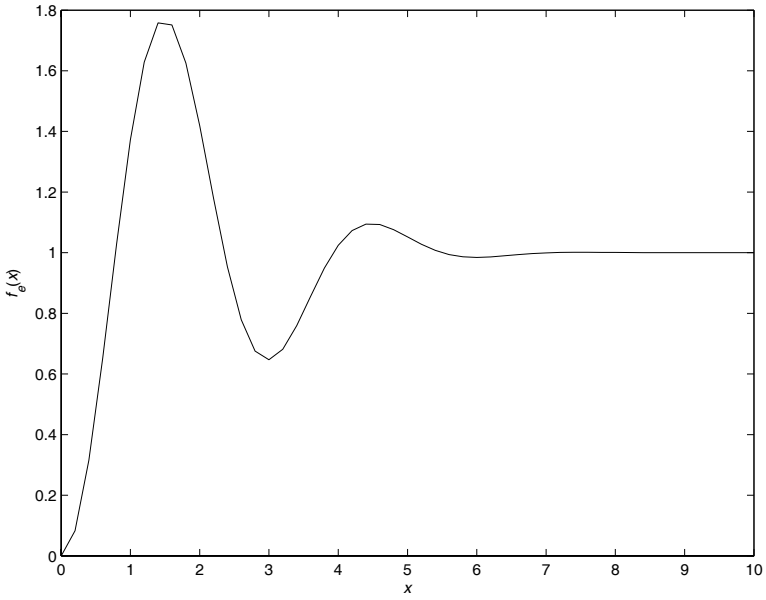
Izračunajmo in na sliki 7.1 prikažimo funkcijske vrednosti, ko gre x od 0 do 10.

```
>> x = [0:0.2:10];
>> figure
>> plot(x,fe(x))
```

Drugi način je definicija ti. **anonimnih funkcij** (ang. anonymous functions). V definiciji kazalca najprej podamo argumente funkcije, nato pa še matematični zapis funkcije. Poglejmo primer za funkcijo z dvema vhodnima argumentoma

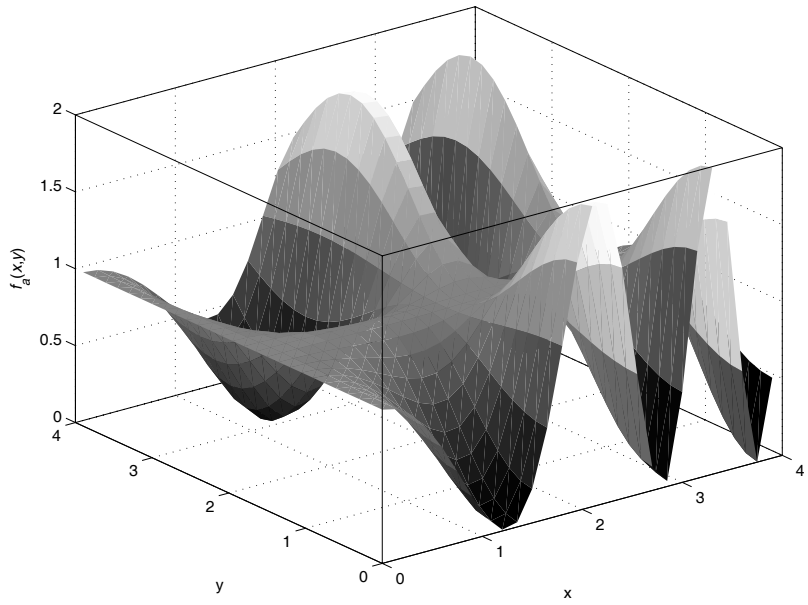
$$f(x,y) = 1 - \cos^2(2x) \cdot \sin(x^2). \tag{7.1}$$

Napišimo program in rezultat prikažimo na diagramu na sliki 7.2. V dvojni zanki s klicem funkcije *fa* izračunamo z–koordinate točk, podanih s pari (x,y).



Slika 7.1 - Izračun funkcijskih vrednosti preko kazalca na funkcijo

```
fa = @(x,y) 1-cos(x).^2.*sin(y.^2)
z = []; x = [0:0.15:4], y = [0:0.15:4];
for j = 1:length(y)
    for i = 1:length(x)
        z(i,j) = fa(x(i),y(j));
    end
end
figure, surf(x,y,z)
xlabel('x'); ylabel('y');
zlabel('\it{f}_a\rm(\it{x}\rm,\it{y}\rm)')
grid on; box on; shading flat; colormap gray
```



Slika 7.2 - Anonimna funkcija dveh spremenljivk

7.2 Orodja za analizo funkcij

Predstavili bomo nekatere važnejše funkcije, ki se uporabljajo za analizo drugih funkcij. To so *fplot*, *fzero*, *fsolve*, *fminbnd* in *fminsearch*.

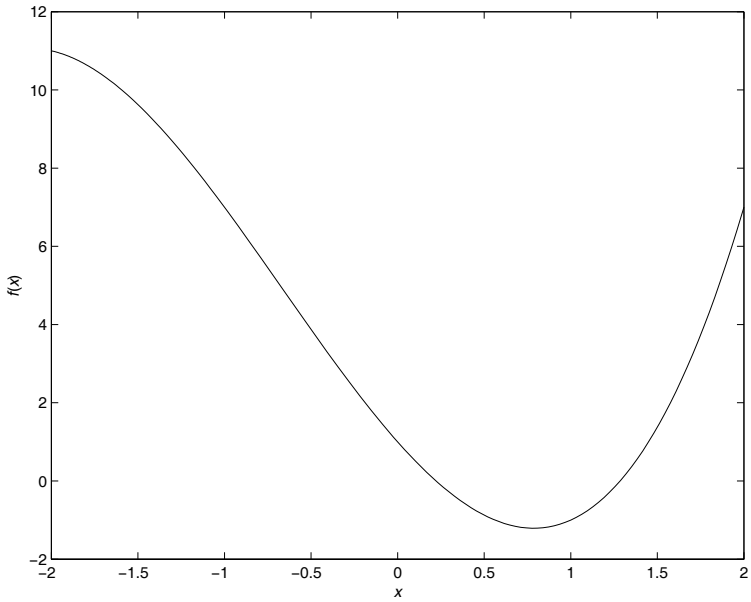
Imejmo funkcijo

$$f(x) = x^3 + 2x^2 - 5x + 1 \tag{7.2}$$

in narišimo njen graf na intervalu $-2 \leq x \leq 2$ s pomočjo funkcije *fplot*. Vhodni parametri funkcije so definicija funkcije, katero želimo narisati, območje neodvisne spremenljivke in relativna toleranca ali število točk. Poglejmo si klica z dvema različnima definicijama funkcije $f(x)$.

```
>> fplot('x^3+2*x^2-5*x+1',[-2 2])

>> f = @(x) x^3+2*x^2-5*x+1;
>> fplot(f,[-2 2])
```

Rezultat je prikazan na sliki 7.3.



Slika 7.3 - Diagram, narejen s funkcijo *fplot*

Ničle funkcije na danem intervalu poiščemo s pomočjo funkcije *fzero*. Vhodni argumenti so definicija funkcije, začetna vrednost in struktura *options*, v kateri so shranjene nastavitve iskalne metode. Izhodni parametri so lokacija ničle *x*, vrednost funkcije *fun* v tej točki, informacija o zaključku iskanja (uspešno ali vzrok neuspešnosti, kodiran z negativnim celim številom med –1 in –5) in končno izhodno poročilo.

```
>> [x,fun,exitflag,out] = fzero(f,3)

x =

    1.2851
```

```
fun =

     0

exitflag =

     1

out =

   intervaliterations: 10
         iterations: 8
         funcCount: 28
         algorithm: 'bisection, interpolation'
         message: 'Zero found in the interval [1.08, 4.35765]'
```

Če imamo več ničel, moramo za iskanje ostalih spremeniti začetni pogoj.

```
>> [x2,fun2] = fzero(f,-2)

x2 =

   -3.5070

fun2 =

     0
```

Poglejmo dejanske ničle funkcije:

```
>> roots([1 2 -5 1])

ans =

   -3.5070
    1.2851
    0.2219
```

Sistem nelinearnih enačb rešujemo s funkcijo *fsolve*. Funkcija rešuje enačbe oblike

$$F(x) = 0, \tag{7.3}$$

kjer sta F in x lahko vektorja ali matriki. Denimo, da imamo sistem dveh nelinearnih enačb z dvema neznankama x_1 in x_2 in parametrom a

$$\begin{aligned} 3x_1 - 5x_2 - e^{ax_1^2} &= 0 \\ -2x_1 - 3x_2 - e^{ax_2^2} &= 0 \end{aligned} \tag{7.4}$$

in želimo poiskati rešitev sistema pri začetnih pogojih $x_1 = x_2 = -3$ in parametru $a = -0,5$. Parametrizirano funkcijo obravnavamo s klicem anonimne funkcije, pred tem pa moramo definirati vrednost parametra.

```
>> a = -0.5;
>> x = fsolve(@(x) sistem(x,a),[-3;-3])
Optimization terminated: first-order optimality is less than
options.TolFun.

x =

   -4.8601
   -2.9161
```

Minimum funkcije na danem intervalu iščemo s pomočjo funkcije *fminbnd*. Izhodni parametri so enaki kot pri funkciji *fzero*, vhodni parametri pa definicija funkcije in pa minimalna ter maksimalna vrednost intervala neodvisne spremenljivke, na katerem iščemo minimum. Poiščimo ga za funkcijo $f(x)$ iz enačbe (7.2) na intervalu $-3 \leq x \leq 3$.

```
>> [x,fun,exitflag,opt] = fminbnd(f,-3,3)

x =

    0.7863

fun =

   -1.2088

exitflag =

     1

opt =

   iterations: 9
   funcCount: 12
   algorithm: 'golden section search, parabolic interpolation'
   message: [1x112 char]
```

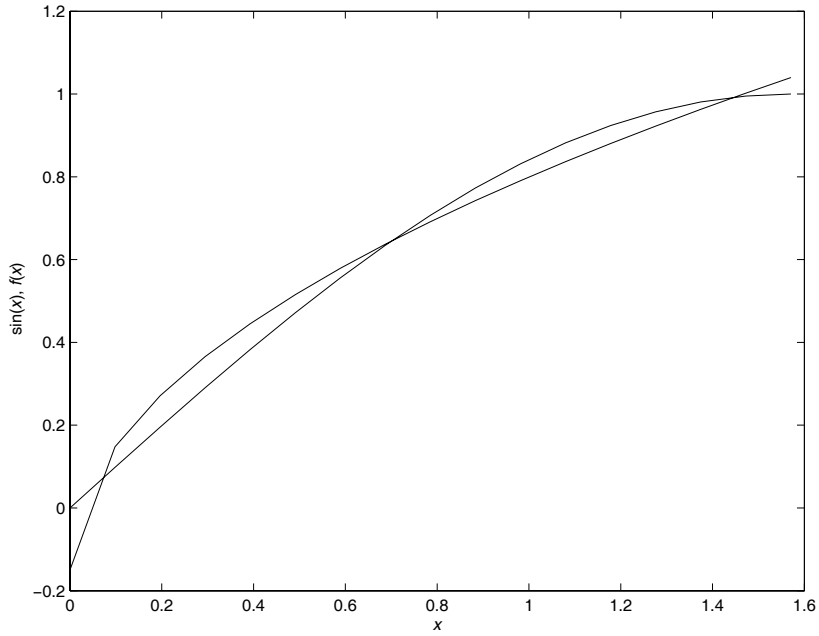
Če imamo **večdimenzionalni problem**, uporabimo funkcijo *fminsearch***,** ki išče minimum dane funkcije z neomejeno Nead–Melderjevo simpleksno metodo. Ogledali si bomo problem aproksimacije dane funkcije s funkcijo z dvema parametroma. Funkcija z imenom *razlika* naj računa vsoto kvadratov razlike obeh funkcij v vsakem koraku optimizacije.

```
function err = razlika(par)
%
x = [0:pi/32:pi/2];
dif = sin(x) - (par(1)*sqrt(x) + par(2));
err = sum(dif.*dif);
```

V ukaznem oknu najprej definiramo nekatere opcije, ki so nam na voljo, s pomočjo že omenjene strukture *options*, nato pa poženemo funkcijo *fminsearch*:

```
>> options = optimset('Display','iter','TolX',...
1e-3,'TolFun',1e-3,'MaxIter',50);
>> par0 = [1,-0.2]
>> par = fminsearch(@razlika,par0,options)

Iteration   Func-count     min f(x)         Procedure
     0            1        0.0587218
     1            3        0.0587218         initial simplex
     2            5        0.0587218         contract inside
     3            7        0.0587218         contract inside
     4            9        0.0587218         contract outside
     5           11        0.0563424         expand
     6           13        0.0563424         contract outside
     7           15        0.0550665         expand
     8           17        0.0527244         expand
     9           18        0.0527244         reflect
    10           19        0.0527244         reflect
    11           20        0.0527244         reflect
    12           22        0.0527244         contract inside
    13           24        0.0526728         contract inside
    14           26        0.0525956         contract inside
    15           28        0.0525375         contract inside
    16           30        0.0525375         contract outside
    17           32        0.0525375         contract inside
    18           33        0.0525375         reflect
    19           35        0.0525336         contract inside
    20           37        0.0525333         contract inside
    21           39        0.0525323         contract inside
    22           41        0.0525321         contract inside

Optimization terminated:
 the current x satisfies the termination criteria using
OPTIONS.TolX of 1.000000e-003
 and F(X) satisfies the convergence criteria using
OPTIONS.TolFun of 1.000000e-003

par =

    0.9490   -0.1496
```

Z *options* smo dosegli, da funkcija pri vsaki iteraciji na zaslon izpiše trenutne akcije ('*Display*','*Iter*') in da se optimizacija konča, ko sprememba razlike doseže določeno mejo (*TolX* = *TolFun* = 0.001). Poglejmo še rezultat aproksimacije z dobljenima parametroma $a = 0.949$ in $b = -0.1496$ na sliki 7.4.



Slika 7.4 - Rezultat aproksimacije s funkcijo fminsearch

7.3 Funkcije za numerično integracijo

Obravnavali bomo dva pristopa numerične integracije:

- integracija funkcije s ti. **kvadraturno formulo,**
- integracija diferencialne enačbe s pomočjo funkcij **ode***xy*.

Pri prvem pristopu računamo določeni integral funkcije

$$A = \int_a^b f(x)dx \tag{7.5}$$

s pomočjo ukaza `a = quad(fun,a,b,tol)`, kjer so vhodni argumenti definicija funkcije *fun*, začetna vrednost *a*, končna vrednost *b* in absolutna toleranca *tol* (privzeta vrednost je 10^{-6} in če je nočemo spremeniti, parametra ne podamo). Izračunajmo integral funkcije, podane z enačbo (7.2) in prikazane na sliki 7.3.

```
>> f = @(x) x.^3+2*x.^2-5*x+1;
>> a = quad(f,-2,2)

a =

   14.6667
```

Pri drugem pristopu mora biti funkcija podana v obliki navadne diferencialne enačbe

$$y^{(n)} = f(y^{(n-1)},\ldots,y'',y',y) \tag{7.6}$$

pri znanih začetnih pogojih

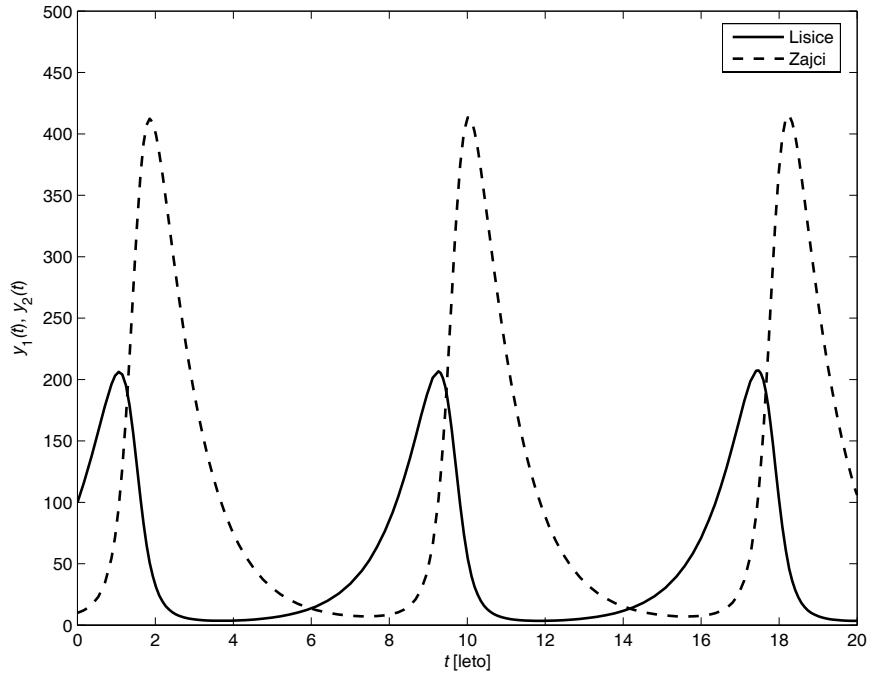
$$y(0) = k_0, y'(0) = k_1,\ldots,y^{(n-1)}(0) = k_{n-1}, \tag{7.7}$$

tako da integral praktično pomeni analitični potek funkcije $y(t)$. Ker funkcije *odexy* rešujejo sisteme navadnih diferencialnih enačb, se enačbe n-tega reda prevedejo v sistem n diferencialnih enačb prvega reda. Naš primer bo tokrat Lotka–Volterrov sistem **plenilec-plen** (z njim lahko predstavimo popularen model dinamike populacije zajčkov in lisičk). Opisuje ga enačba

$$\begin{aligned} \dot{y}_1 &= (1-\alpha y_2) y_1 \\ \dot{y}_2 &= (-1+\beta y_1) y_2 \end{aligned}, \tag{7.8}$$

v Matlabu pa je predstavljen z vgrajeno funkcijo *lotka*. Izračunajmo in narišimo časovna poteka dinamik obeh populacij pri začetnih pogojih $y_1(0) = 100$ in $y_2(0) = 10$.

```
>> [t,y] = ode45('lotka',[0 20],[100 10]);
>> plot(t,y(:,1),t,y(:,2),'--','linewidth',1.5)
>> axis([0 20 0 500])
>> legend('Lisice','Zajci')
>> xlabel('\it{t}\rm [leto]')
>> ylabel('\it{y}_{\rm1}(\it{t}\rm),\it{y}_{\rm2}(\it{t}\rm)')
```



Slika 7.5 - Dinamiki populacij lisic in zajcev

Še nekaj besed o x in y v funkciji *odexy*. Gre namreč za red metode računanja. Pri zadnjem primeru smo uporabili funkcijo *ode45*, ki je osnovana na metodi Runge–Kutta (4,5). Od bolj uporabnih omenimo še *ode23* in *ode115* za toge sisteme, primerov pa ne bomo navajali.

7.4 Nekaj napotkov za boljše programiranje

V zadnjem podpoglavju bo opisanih nekaj praktičnih napotkov, s katerimi si bo bralec lahko pomagal pri programiranju zahtevnejših aplikacij:

- pretvorbe tipov podatkov,
- osnovni postopki obravnave grafičnih objektov s pomočjo ročic (ang. handle graphics),
- snemanje videoposnetkov,
- dodajanje vmesniških krmilnih in menijskih elementov,
- primer preprostega grafičnega uporabniškega vmesnika.

7.4.1 Operacije z nizi znakov

Niz znakov je vektor, ki ima v svojih elementih zaporedje znakov. Zato običajne matematične vektorske operacije pri nizih niso možne. Seštevanje dveh nizov se imenuje *združevanje* (ang. string concatenation) in se izvede z ukazom *strcat.* Poglejmo si primer.

```
>> s1 = 'Dober'

s1 =

Dober

>> s2 = 'dan!'

s2 =

dan!

>> s3 = strcat(s1,s2)

s3 =

Doberdan!
```

Vidimo, da bomo presledek morali dodati ročno v enega od nizov.

```
>> s2 = ' dan!'

s2 =

 dan!

>> s5 = strcat(s1,s2)

s5 =

Dober dan!
```

Če hočemo sestaviti niz iz znakovnih nizov in števil, moramo slednja najprej pretvoriti v znakovne nize z ukazom *num2str*. Uporaba tega postopka pride najbolj do izraza pri shranjevanju serij podatkov, katere pridobivamo v zankah.

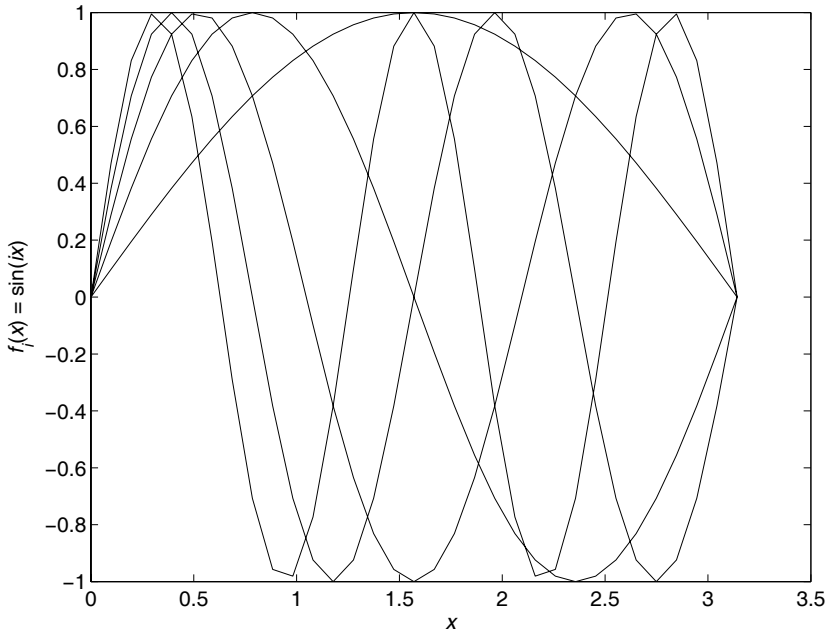
```
>> a = 20;
>> class(a)

ans =

double

>> b = num2str(a)

b =

20

>> class(b)

ans =

char
```

Z uporabo funkcije *class* smo ugotovili, da smo število 20 spremenili v niz znakov '2' in '0'. Zdaj pa naredimo primer shranjevanja serij podatkov.

```
>> x = 0:pi/32:pi;
>> for i = 1:5
       y = sin(i*x);
       save(strcat('sinus',num2str(i)),'x','y')
   end
```

Ta program nam kreira 5 datotek: *sinus1.mat*, *sinus2.mat*, *sinus3.mat*, *sinus4.mat* in *sinus5.mat*. Če hočemo vse meritve naknadno prikazati na istem diagramu, postopamo na podoben način. Uporabili bomo še dva ukaza, ki nam bosta služila kot programski klic vprašanja, ali naj se program izvede ali ne. To sta *odg* = *input*('*Vprašanje*', *tip_odgovora*) in *strcmp*('*niz1*', '*niz2*'). Po izvedbi prvega ukaza Matlab čaka na uporabnikov odziv preko tipkovnice, z drugim ukazom pa primerjamo odgovor z želenim odgovorom. Poglejmo primer programa in diagram na sliki 7.6.

```
vpr = input('Ali hočeš prikazati meritve na diagramu (d/n)? ','s');
if strcmp(vpr,'d')
    figure
    hold on
    for i = 1:5
        load(strcat('sinus',num2str(i)));
        plot(x,y);
    end
end
```



Slika 7.6 - Diagram vseh shranjenih funkcij

7.4.2 Upravljanje grafičnih objektov z ročicami

Z ročicami smo se nekajkrat že srečali, pa vendar celovite obravnave še nismo podali. **Ročica objekta** (ang. handle) je spremenljivka s pozitivno realno vrednostjo, ki kaže na objekt, hkrati pa vsebuje tudi strukturo, v kateri so shranjene lastnosti objekta. S prirejanjem ali spreminjanjem vrednosti lastnosti lahko povzročimo takojšnje spremembe izgleda in delovanja objekta. Bistvo tega načina dela je, da imamo z definicijo ročice ob kreiranju novega objekta takoj na voljo tudi vse njegove lastnosti in smo zato pri pisanju programa veliko bolj fleksibilni. Poglejmo si primer. Ustvarimo novo okno diagrama z ukazom *figure*, narišimo krivuljo

$$f(x) = 1 - \cos(2x) \cdot e^{-x} \tag{7.9}$$

in določimo lastnosti vseh objektov.

```
>> rocica_f = figure

rocica_f =

     1

>> x=0:0.1:5;
>> rocica_p = plot(x,1-cos(2*x).*exp(-x))

rocica_p =

  152.0023
```

Definirali smo dva objekta in v ročicah *rocica_f* ter *rocica_p* imamo shranjene njune lastnosti. Osnova obeh objektov je glavno okno, ki ima po definiciji ročico 0. Pravimo, da je **roditelj** (ang. parent) vseh ostalih objektov, ki iz njega izhajajo. Vsem grafičnim objektom (diagramom) se, če sami ne določimo drugače, priredi zaporedno naravno število, ki je hkrati tudi ročica na dani diagram. Poglejmo lastnosti roditelja.

```
>> get(0)

CallbackObject = []
      CommandWindowSize = [100 44]
      CurrentFigure = [1]
      Diary = off
      DiaryFile = diary
      Echo = off
      FixedWidthFontName = Courier
      Format = short
      FormatSpacing = loose
      Language = slovenian
      MonitorPositions = [0 0 1280 1024]
      More = off
      PointerLocation = [654 323]
      PointerWindow = [0]
      RecursionLimit = [500]
      ScreenDepth = [32]
      ScreenPixelsPerInch = [96]
      ScreenSize = [1 1 1280 1024]
      ShowHiddenHandles = off
      Units = pixels

      BeingDeleted = off
      ButtonDownFcn =
      Children = [1]
      Clipping = on
      CreateFcn =
      DeleteFcn =
      BusyAction = queue
      HandleVisibility = on
      HitTest = on
      Interruptible = on
      Parent = []
      Selected = off
      SelectionHighlight = on
      Tag =
      Type = root
      UIContextMenu = []
      UserData = []
      Visible = on
```

Z izvedbo ukaza *get*(0) smo poizvedli po lastnostih ukaznega okna, ki je baza ali roditelj (ang. parent) vseh ostalih objektov. Vidimo, da je vrednost lastnosti *Children* [1], kar pomeni, da ima glavni objekt enega otroka z ročico 1. Ta objekt je naš

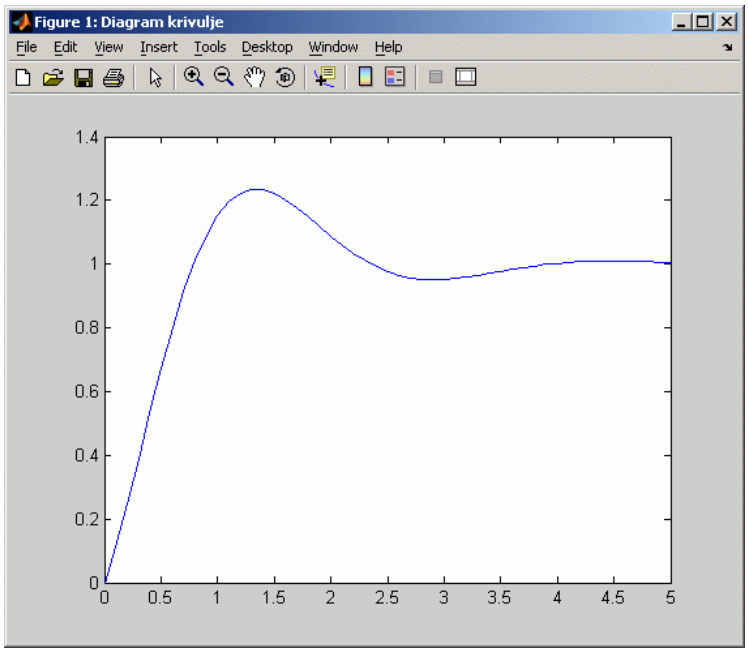
diagram, ki je od roditelja podedoval nekaj lastnosti, nakaj pa ima tudi lastnih. Po posamezni lastnosti poizvemo z ukazom *get*(*ročica*,'*lastnost*').

```
>> get(rocica_f,'children')

ans =

  151.0023
```

Dobimo isto vrednost, kot smo jo dobili pri definiciji ročice krivulje. Zdaj pa spremenimo ime okna diagrama. Uporabimo ukaz *set*(*ročica*,'*lastnost*',*vrednost*), kjer je *vrednost* skalar, matrika ali niz znakov, odvisno od tipa lastnosti.

```
>> set(rocica_f,'name','Diagram krivulje')
```

Okno diagrama si lahko ogledamo na sliki 7.7.



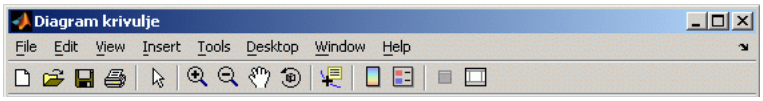
Slika 7.7 - Okno s spremenjenim imenom

Preverimo še lastnost '*number*'.

```
>> set(rocica_f,'number')
[ {on} | off ]
```

Pri nekaterih lastnostih z ukazom *set* dobimo vse možne nastavitve in v oklepaju privzeto oziroma trenutno izbrano nastavitev. Nastavimo to lastnost na '*off*' in prikažimo razliko na sliki 7.8.

```
>> set(rocica_f,'number','off')
```



Slika 7.8 - Glava diagrama brez številke

Kadar delamo s tridimenzionalnimi diagrami, je ena od pomembnih lastnosti *pogled* (ang. view), to je kot, pod katerim gledamo diagram. Z nastavljanjem pogleda v *for*-zanki lahko objekt poljubno zavrtimo. Ob tem pa si bomo še pogledali, kako iz zaporedja slik posnamemo film v formatu *.avi. Narišimo 5 funkcij iz enačbe (7.9), objekt zavrtimo za 360° in slike posnemimo v datoteko.

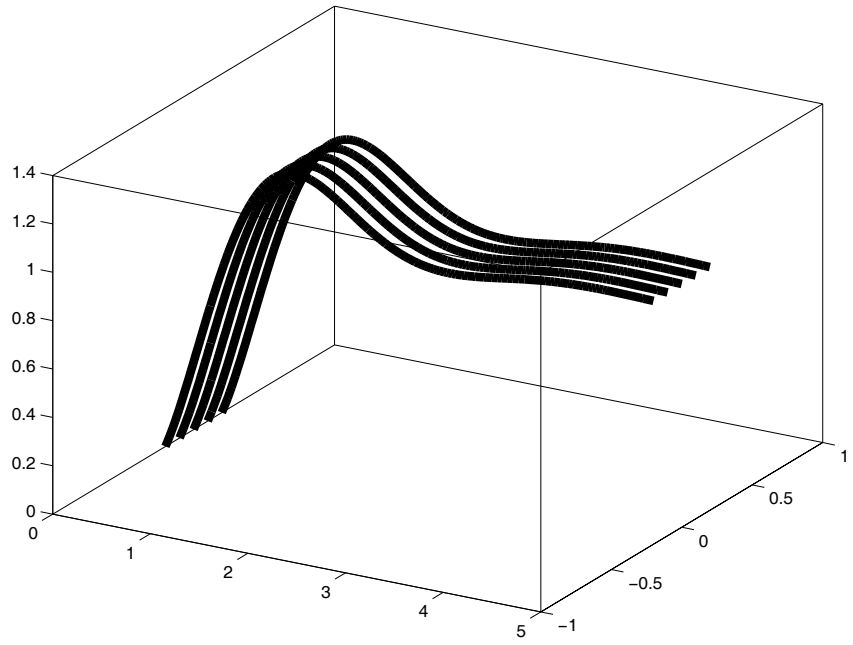
```
% Naredimo novo okno in rezerviramo spomin
fig = figure
set(fig,'DoubleBuffer','on');

% Narišemo 5 funkcij na tridimenzionalni diagram
for k = -0.2:0.1:0.2
    plot3(x,k*ones(1,length(x)),...
          1-cos(2*x).*exp(-x),'linewidth',5)
    hold on
end
set(gca,'Ylim',[-1 1])

% Inicializiramo objekt za snemanje filma in izhodno datoteko
mov = avifile('vrtiljak.avi')

% V zanki obračamo diagram, naredimo slike in jih shranimo v
% datoteko
for i = 1:360
    set(gca,'view',[i-1,30])
    M = getframe;
    mov = addframe(mov,M);
end

% Zdaj pa še zapremo datoteko
mov = close(mov)
```

Po končanem snemanju filma javi Matlab naslednje sporočilo:

```
Adjustable parameters:
               Fps: 15.0000
       Compression: 'Indeo5'
           Quality: 75
    KeyFramePerSec: 2.1429
         VideoName: 'vrtiljak.avi'

Automatically updated parameters:
        Filename: 'vrtiljak.avi'
     TotalFrames: 360
           Width: 436
          Height: 344
```

```
          Length: 0
       ImageType: 'Truecolor'
    CurrentState: 'Closed'
```

Parametre s seznama *Adjustable parameters* lahko nastavljamo v klicu `mov = avifile('parameter','vrednost',...)`. Prikažimo še enega izmed posnetkov – slika 7.9.



Slika 7.9 - Eden od posnetkov pri snemanju avi-datoteke

7.4.3 Grafični uporabniški vmesnik

Namen grafičnih uporabniških vmesnikov (ang. graphical user interface – GUI) je omogočiti uporabniku, da bodoče akcije izbira s pomočjo miške, programerju pa ponuditi enostavno intuitivno orodje za kombiniranje različnih objektov. V tem podpoglavju bomo obravnavali osnove uporabniških krmilnih vmesniških elementov (ang. user interface control, UI control) in uporabniških menijskih vmesniških elementov (ang. user interface menu, UI menu).

Matlab pozna devet različnih krmilniških elementov GUI:

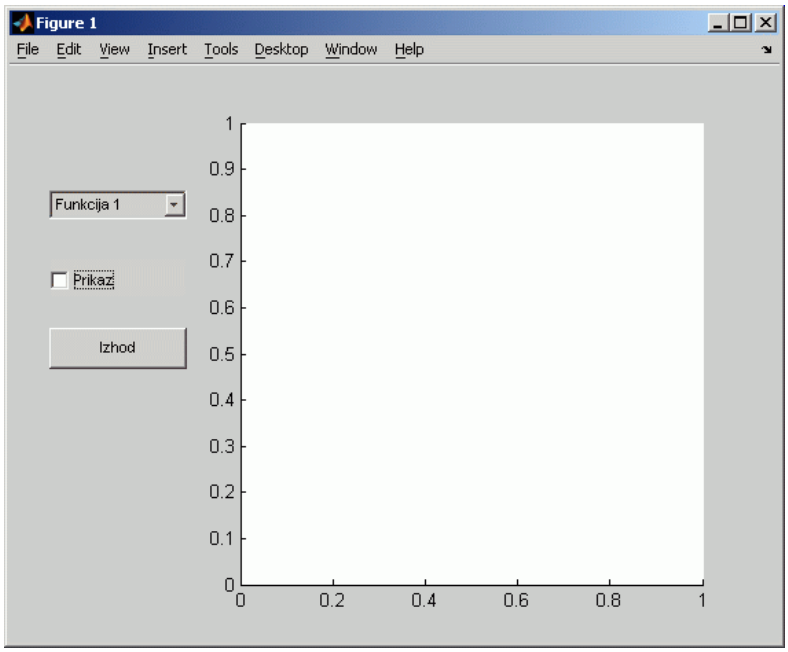
- **check box** – stikalo (kvadratne oblike),
- **editable text** – vnosno polje,
- **pop-up menu** – izbirnik (padajoči meni),
- **push button** – gumb,
- **radio button** – stikalo (okrogle oblike),
- **slider** – drsni izbirnik,
- **static text** – tekstovna oznaka,
- **list box** – seznam,
- **toggle button** – preklopno stikalo.

Krmilni element kreiramo z ukazom `ročica = uicontrol('lastnost 1', nastavitev 1, ..., 'lastnost n', nastavitev n)`. Kot prvi parameter lahko navedemo tudi ročico diagrama, v katerem naj se element nariše. Po lastnostih

poizvedujemo z ukazom *get*, nastavljamo pa jih z ukazom *set*. Poglejmo si primer diagrama s tremi krmilnimi elementi.

```
fig = figure
ax = axes('Position',[0.3 0.1 0.6 0.8])
k_el1 = uicontrol('Position',[30 300 100 30],...
                  'Style','PopupMenu',...
                  'String','Funkcija 1|Funkcija 2');
k_el2 = uicontrol('Position',[30 250 100 30],...
                  'Style','Checkbox',...
                  'String','Prikaz')
k_el3 = uicontrol('Position',[30 200 100 30],...
                  'Style','PushButton',...
                  'String','Izhod')
```

Slika 7.10 prikazuje osnovni izgled diagrama s tremi elementi.



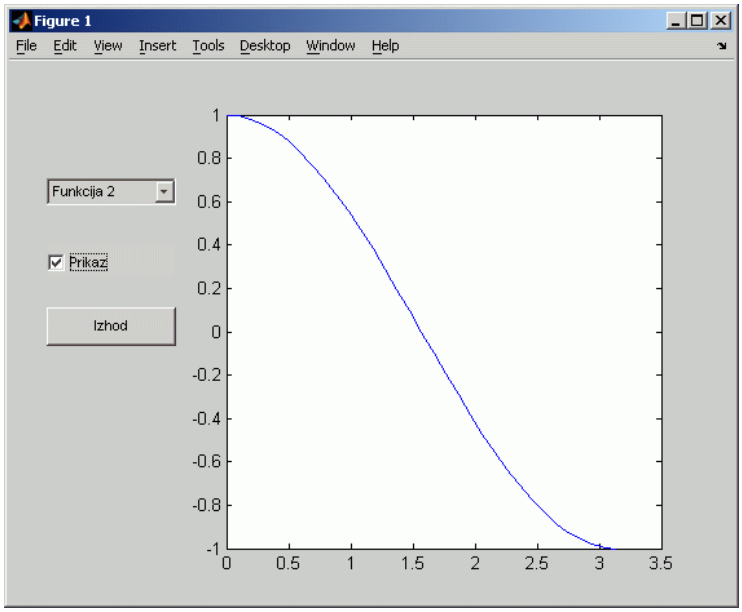
Slika 7.10 - Diagram s tremi vmesniškimi krmilnimi elementi

Zdaj pa nastavimo nekaj lastnosti elementov, ki bodo povečale uporabnost grafičnega vmesnika. Najpomembnejši lastnosti vsakega elementa sta *Callback* ali povratni klic in *Value* oz. vrednost.

```
x = 0:pi/32:pi;
y = [sin(x);cos(x)];
p = plot(x,y(1,:),'Visible','off');
set(k_el1,'Callback','set(p,''YData'',y(get(k_el1,''Value''),:))')
set(k_el2,'Callback','set(p,''Visible'',''on'')')
set(k_el3,'Callback','delete(fig)')
```

Prvi povratni klic nariše izbrano vrstico matrike *y*. V prvi vrstici je funkcija *sinus*, v drugi pa *kosinus* danega intervala *x*. Na ta način izbiramo med funkcijama. Ker

imamo klic podan kot niz znakov, morajo biti nizi znotraj njega označeni z dvojnimi navednicami. Drugi klic nam izbrano funkcijo prikaže na diagramu. Tretji klic pa zbriše ročico glavnega okna. Vmesni izgled diagrama je na sliki 7.11.



Slika 7.11 - Izgled diagrama po opravljenih izbirah

Zdaj pa si bomo pogledali še, kako naredimo svoje menije in uporabimo vnaprej pripravljene objekte. Na diagram poleg že obstoječih menijev dodamo novega z ukazom `ročica_menija_1 = uimenu('label','Oznaka menija 1')`, postavko v meni pa vključimo z `ročica_postavke_1 = uimenu(ročica_menija_1, 'label', 'Postavka 1')`. Podpostavko naredimo na podoben način, le da prvo ročico zamenjamo z ročico podmenija. Na enak način se lotimo tudi menijev, ki jih dobimo ob kliku na diagram z desno tipko miške – ukaz se glasi

```
ročica_kontekstnega_menija = uicontextmenu('label','Oznaka 1').
```

Narejeni objekti, ki so nam na voljo, so:

- **msgbox** – okno s sporočilom uporabniku
- **errordlg** – okno z opozorilom na napako
- **questdlg** – okno z vprašanjem in tremi možnimi izbirami
- **uigetfile** – odpre okno za izbiro datoteke, ki se naj odpre
- **uiputfile** – odpre okno za izbiro datoteke za shranjevanje

V naš diagram bomo dodali meni s podmenijem in objekta *questdlg* ter *uiputfile*. V meni *Uporabnik* dodajmo postavko *Shrani*, v povratni klic za zaprtje diagrama pa dodajmo vprašanje *Ali ste 100% prepričani?*. V ta namen napišimo dve m-datoteki, *povr_klic1.m* in *povr_klic2.m*, ki nam bosta služili kot argumenta povratnih klicev *Callback*. Poglejmo si kodo novega programa.

```
fig = figure
ax = axes('Position',[0.3 0.1 0.6 0.8])
k_el1 = uicontrol('Position',[30 300 100 30],...
                  'Style','PopupMenu',...
                  'String','Funkcija 1|Funkcija 2');
```

```
k_el2 = uicontrol('Position',[30 250 100 30],...
                  'Style','Checkbox',...
                  'String','Prikaz')
k_el3 = uicontrol('Position',[30 200 100 30],...
                  'Style','PushButton',...
                  'String','Izhod')
x = 0:pi/32:pi;
y = [sin(x);cos(x)];
p = plot(x,y(1,:),'Visible','off');
% Dodamo meni in podmeni ter definiramo povratni klic
m1 = uimenu('Label','Uporabnik');
m2 = uimenu(m1,'Label','Shrani','Callback','povr_klic1')
set(k_el1,'Callback','set(p,''YData'',y(get(k_el1,''Value''),:))')
set(k_el2,'Callback','set(p,''Visible'',''on'')')
% Tukaj povratni klic preusmerimo v datoteko povr_klic2.m
set(k_el3,'Callback','povr_klic2')
```

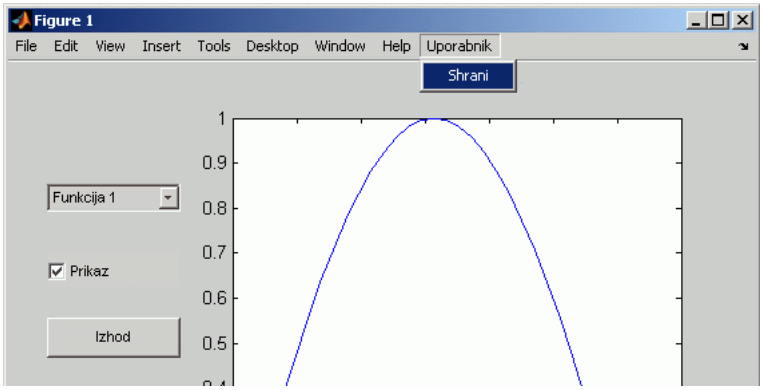
Datoteka *povr_klic1.m*:

```
[ime,pot] = uiputfile('*.fig','Shrani funkcijo!','privzet.fig');
if ischar(ime)
    saveas(fig,ime)
end
```

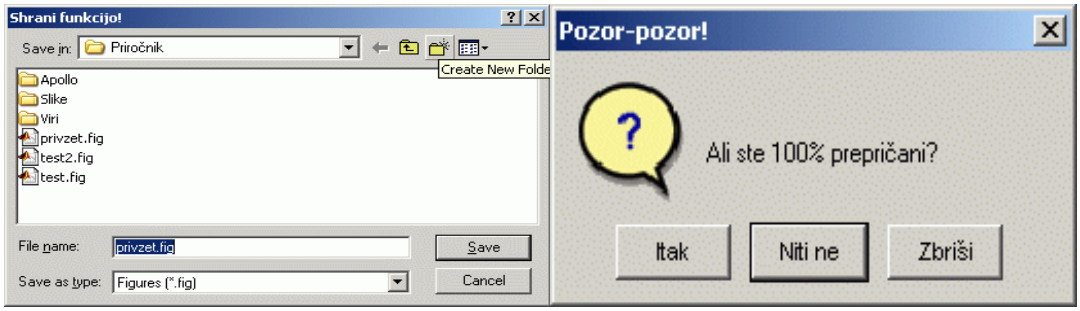
Datoteka *povr_klic2.m*:

```
vpr = questdlg('Ali ste 100% prepričani?','Pozor-pozor!',...
               'Itak','Niti ne','Zbriši','Niti ne');
if (strcmp(vpr,'Itak'))
    delete(fig)
end
```

Novi izgled uporabniškega vmesnika je prikazan na sliki 7.12. Po kliku na postavko *Shrani* se nam odpre dialog za shranjevanje z naslovom *Shrani funkcijo!*. Ko končamo z delom, pritisnemo na gumb *Izhod* in odpre se nam novi dialog s prej podanimi vprašanji in odgovori. Klik na *Itak* zapre okno diagrama.



Slika 7.12 - Novi izgled vmesnika z novim menijem



Slika 7.13 - Izbirnika, ki sledita povratnima klicema

7.4.4 Primer grafičnega programa

Vse do sedaj povedano o objektih bomo uporabili pri pisanju programa relativno preprostega uporabniškega vmesnika, ki nam omogoča prikazovanje in premikanje treh funkcij. Opozorimo naj, da je načinov reševanja tega problema več, predstavili bomo samo najosnovnejše ogrodje takega vmesnika, ki bo ponazoril uporabo ročic, grafičnih objektov in pretvorbe tipov podatkov.

Naj bo primer zasnovan na naslednji način:

- vmesnik naj omogoča izbiranje med tremi funkcijami $f_i(x)$, $0 \le x \le 5$:

$$f_1(x) = 1 - \cos(2x) \cdot e^{-x} \tag{7.10}$$

$$f_2(x) = \sin(x)\cos(x) \tag{7.11}$$

$$f_3(x) = -0.08\left(x^6 - 15x^5 + 85x^4 - 225x^3 + 274x^2 - 120x\right) \tag{7.12}$$

- s pomočjo miške naj uporabnik s klikom na gumba *Gor* in *Dol* poljubno premakne izbrano funkcijo v vertikalni smeri iz osnovne lege; obe funkciji, tako osnovna kot prenaknjena, naj se izrišeta v vsakem koraku, tako da dobimo področje, omejeno z obema funkcijama;
- uporabnik naj ima možnost izbire velikosti premika v vsakem koraku;
- na voljo naj bosta tudi izbira senčenja tega področja in pa barva senčenega področja, če je le–to vključeno;
- vmesnik naj vsebuje tipko za izhod, ki zbriše celoten vmesnik.

Podana bo celotna koda programa s komentarji za vsak posamezni sklop vmesnika. Program zaradi svoje zasnove ne potrebuje nobene zanke, ločen pa je tudi po konstrukcijskih korakih – izračun funkcij, risanje vmesnika, definicija začetnih pogojev in definicija povratnih klicev.

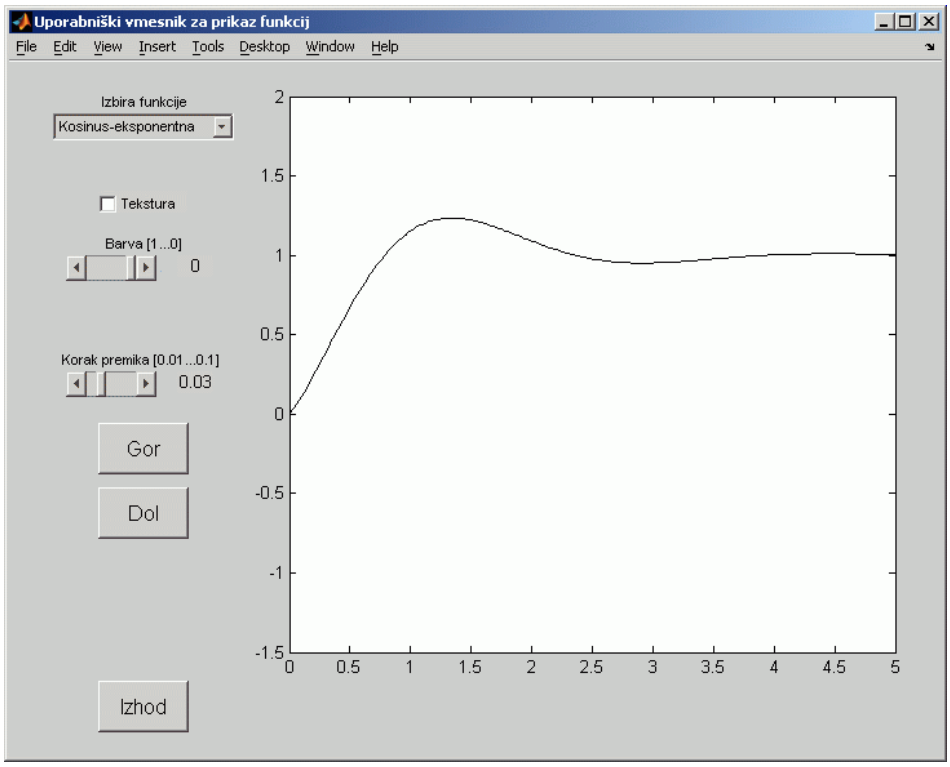
```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Preprost grafični vmesnik %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Začnemo z zapisom treh funkcij v vrsticah matrike yz
x = 0:0.02:5;
yz = [(1-cos(2*x).*exp(-x));...
      (sin(x).*cos(x));...
       -0.08*(x.^6-15*x.^5+85*x.^4-225*x.^3+274*x.^2-120*x)];
```

```
%
% Začetno matriko zapišemo v yn, ki jo bomo potem spreminjali
yn = yz;
%
% Najprej naredimo osnovno okno, ga oblikujemo in določimo osi
% diagrama
rocica_f = figure;
set(rocica_f,'NextPlot','add',...
             'Name','Uporabniški vmesnik za prikaz funkcij',...
             'Position',[198 385 722 539],...
             'Number','off');
ax = axes('position',[0.3 0.15 0.65 0.80]);
%
% Nato naredimo meni, kjer bomo izbirali funkcije
l1 = uicontrol('Style','Popupmenu','Position',[35 470 140 30]);
set(l1,'string','Kosinus-eksponentna|Sinus-kosinusna|Polinomska');
t1 = uicontrol('Style','Text','Position',[65 500 80 15],...
               'String','Izbira funkcije');
set(t1,'BackgroundColor',get(rocica_f,'Color'));
%
% Sledi izbirnik za senčenje z napisom nad njim
t2 = uicontrol('Style','Text','Position',[65 390 80 15],...
               'String','Barva [1...0]');
set(t2,'BackgroundColor',get(rocica_f,'Color'));
ch = uicontrol('Style','CheckBox','Position',[70 420 70 20]);
set(ch,'String','Tekstura');
%
% Tole bosta drsni izbirnik za svetlost senčenja in pa prikazovalnik
% vrednosti
s1 = uicontrol('Style','Slider','Position',[45 370 70 20]);
set(s1,'Min',0,'Max',1,'SliderStep',[0.05 0.05],'Value',1)
td1 = uicontrol('Style','Text','Position',[130 370 30 20]);
set(td1,'String',num2str(0),'FontSize',10);
%
% Izbirnik za korak premika in prikazovalnik vrednosti
t3 = uicontrol('Style','Text','Position',[32 300 140 15],...
               'String','Korak premika [0.01...0.1]',...
               'BackgroundColor',get(rocica_f,'Color'));
s2 = uicontrol('Style','Slider','Position',[45 280 70 20]);
set(s2,'Min',0.01,'Max',0.1,'SliderStep',[0.01 0.01],'Value',0.03)
td2 = uicontrol('Style','Text','Position',[130 280 30 20]);
set(td2,'String',num2str(get(s2,'Value')),'FontSize',10);
%
% Gumba za premikanje funkcij navzgor in navzdol
b1 = uicontrol('Style','Pushbutton','Position',[70 220 70 40]);
set(b1,'String','Gor','FontSize',12);
b2 = uicontrol('Style','Pushbutton','Position',[70 170 70 40]);
set(b2,'String','Dol','FontSize',12);
%
% Pa še gumb za izhod
iz = uicontrol('Style','Pushbutton','Position',[70 20 70 40]);
```

```
set(iz,'String','Izhod','FontSize',12);
%
% Definicije začetnih parametrov
% Izb bo parameter za izbiranje funkcije
izb = 1;
% Parameter premika
p = 0;
% Parameter za barvo senčenja
bar = 1;
% Začetni korak
kor = get(s2,'Value');
%
% Senčenje bomo izvedli z ukazom fill, kjer podamo zaključeno
% površino s točkami v vektorjih
rocica_fi = fill([x x(end:-1:1)],...
                 [yz(izb,1:end) yz(izb,end:-1:1)],[1 1 1]);
set(rocica_fi,'erasemode','background')
%
% Nastavimo območji abscise in ordinate diagrama
set(ax,'XLim',[0 5],'YLim',[-1.5 2]);
%
% Tu se začnejo povratni klici, ki izvedejo vsako spremembo z
% uporabniške strani
%
% Sprememba izbirnika funkcij vpliva na izbiro vrstice v matriki yn
set(l1,'Callback',['izb = get(l1,''Value''); '...
      'set(rocica_fi,''YData'',[yz(izb,1:end) yn(izb,end:-1:1)],'...
      '''FaceColor'',[bar bar bar])']);
% Sprememba pozicije drsnika za barvo senčenja vpliva na parameter
% bar, ki ga uporabljajo tudi drugi klici
set(s1,'Callback',['bar=1-get(ch,''Value'')*get(s1,''Value''); '...
      'set(td1,''String'',num2str(bar)); '...
      'set(rocica_fi,''YData'',[yz(izb,1:end) yn(izb,end:-1:1)],'...
      '''FaceColor'',[bar bar bar])']);
% Podobno velja tudi za spremembo pozicije drsnika za korak
set(s2,'Callback',['kor = get(s2,''Value''); '...
       'set(td2,''String'',num2str(get(s2,''Value'')))']);
% Klik na enega od gumbov Gor/Dol pove, koliko naj bo parameter
% premika p
set(b1,'Callback',['p = kor; '...
      'yn(izb,:) = yn(izb,:) + p; '...
      'set(rocica_fi,''YData'',[yz(izb,1:end) yn(izb,end:-1:1)],'...
      '''FaceColor'',[bar bar bar]);']);
set(b2,'Callback',['p = -kor; '...
      'yn(izb,:) = yn(izb,:) + p; '...
      'set(rocica_fi,''YData'',[yz(izb,1:end) yn(izb,end:-1:1)],'...
      '''FaceColor'',[bar bar bar]);']);
% Kljukica v kvadratku za senčenje postavi vrednost s1 na 0, zato je
% barva senčenja bela ali [1 1 1]
set(ch,'Callback',['bar = 1-get(ch,''Value'')*get(s1,''Value'');'...
      'set(rocica_fi,''YData'',[yz(izb,1:end) yn(izb,end:-1:1)],'...
```

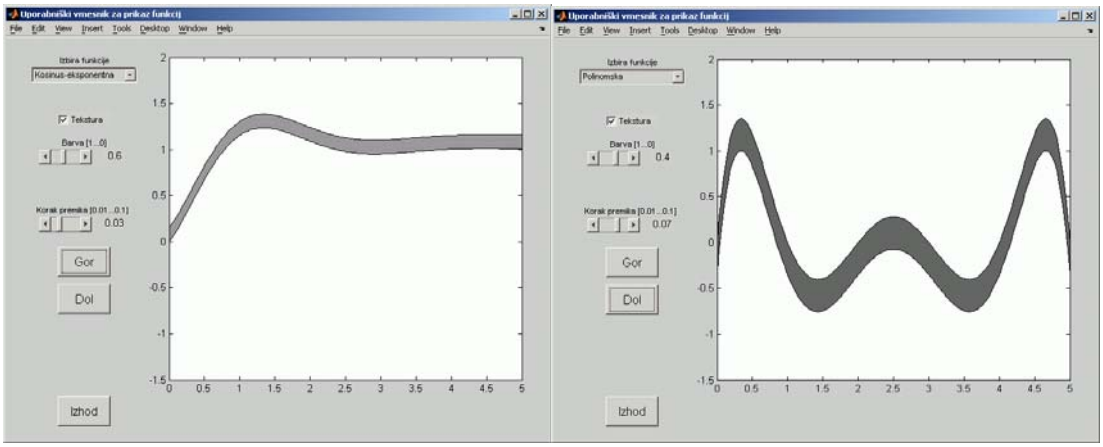
```
      '''FaceColor'',[bar bar bar])']);
% Klik na tipko za izhod zbriše ročico okna in s tem tudi vse
% njegove otroke
set(iz,'Callback','delete(rocica_f)');
```

Po vnosu programa in prvem klicu se nam na zaslon izriše vmesnik, kot je prikazano na sliki 7.14.



Slika 7.14 - Začetni izgled grafičnega vmesnika

Obkljukajmo izbiro teksture, izberimo vrednost barve 0,6 in premaknimo funkcijo za 5 korakov navzgor (slika 7.15 levo). Nato izberimo polinomsko funkcijo, nastavimo vrednost barve na 0,4, izberimo korak premika 0.07 in naredimo 5 korakov navzdol (slika 7.15 desno).



Slika 7.15 - Levo - premaknjena eksponentna funkcija, desno - premaknjena polinomska funkcija

Bistvo, ki ga želimo tu poudariti, je v tem, da je obstoječi program samo ogrodje za večje aplikacije. Ko je program zastavljen modularno, ga lahko na enostaven način razširimo. Bralec lahko poizkusi v program dodati oznake funkcij na diagramu (z ukazom *text* in z imenom ter koordinatami, ki jih dobi v vsakem koraku od trenutno izbrane funkcije) ali pa podatka o maksimalni in minimalni vrednosti trenutno izbrane funkcije, ki se izpišeta v okno na levi strani.

8. KNJIŽNICA FUNKCIJ ZA ANALIZO VODENJA SISTEMOV (CONTROL SYSTEM TOOLBOX)

Matlab vključuje širok nabor funkcij, prikladnih za raziskovalce tako s področja teorije vodenja kot s področja sistemske teorije. Aritmetika kompleksnih števil, lastne vrednosti, iskanje korenov enačb, inverzija matrik in hitra Fourierjeva transformacija (*fft*) so le nekateri od primerov pomembnih numeričnih orodij, ki se uporabljajo v omenjenih področjih. Knjižnica orodij za analizo vodenja izkorišča te funkcije pri gradnji algoritmov, večinoma v obliki funkcijskih m–datotek, s katerimi izvajamo običajne postopke analize, sinteze in modeliranja v sistemih vodenja. Na voljo so predstavitve sistemov v prostoru stanj, s prenosnimi funkcijami v polinomski in faktorizirani obliki, v zveznem in diskretnem času, omogočeni pa so tudi direktni prehodi med različnimi zapisi. Grafično lahko predstavimo časovne in frekvenčne odzive, Bodejeve in Nyquistove diagrame ter diagrame lege korenov. Nenazadnje pa lahko tudi naredimo nove m–datoteke s poljubnim zaporedjem funkcij za obvladovanje kompleksnejših aplikacij.

8.1 Zapisi linearnega časovno nespremenljivega modela

Linearni časovno nespremenljivi proces (ang. linear time-invariant, LTI), za katerega želimo načrtati sistem vodenja, je podan s sistemom diferencialnih enačb, prenosno funkcijo ali v prostoru stanj. Osnovne funkcije, ki jih uporabimo za zapis matematičnega modela procesa, so:

- **zpk(Z,P,K)** (zapis prenosne funkcije v faktorizirani obliki)
- **tf(num,den)** (zapis prenosne funkcije v polinomski obliki)
- **ss(A,B,C,D)** (zapis v prostoru stanj)

Začnimo torej z zapisom modela v obliki prenosne funkcije. Proces, ki ga podaja enačba

$$G(s) = \frac{3(s+2)(s+4)}{(s+1)^2(s+5)}\ , \tag{8.1}$$

zapišimo s prenosno funkcijo v faktorizirani obliki s pomočjo ukaza *zpk*.

```
>> G = zpk([-2 -4],[-1 -1 -5],3)

Zero/pole/gain:
3 (s+2) (s+4)
--------------
(s+1)^2 (s+5)
```

Klic funkcije vrne objekt *G* tipa *ZPK*. S tem zapisom obravnavamo prenosne funkcije oblike

$$G(s) = \frac{K \cdot \prod_{i=1}^{m}(s+z_i)}{\prod_{j=1}^{n}(s+p_i)} \ , \tag{8.2}$$

kjer so elementi modela ničle (z_i), poli (p_i) in ojačenje (K) – prva dva podamo z vektorjema Z in P, ojačenje pa je od nič različen skalar iz množice realnih ali kompleksnih števil. Opozoriti velja, da moramo razlikovati K od ojačenja sistema v stacionarnem stanju K_{SS}, ki je v tem primeru

$$K_{SS} = \frac{K \cdot \prod_{i=1}^{m} z_i}{\prod_{i=1}^{n} p_i} \tag{8.3}$$

Dimenzija vektorja Z naj bo manjša ali kvečjemu enaka dimenziji vektorja P. Če sistem nima ničel, vnesemo prazen vektor.

```
>> G2 = zpk([],[-1 -1 -5],3)

Zero/pole/gain:
      3
-------------
(s+1)^2 (s+5)
```

Spremenljivka *s*, ki nastopa v modelu, je kompleksna spremenljivka Laplaceove transformiranke časovno zveznega sistema. Če jo definiramo kot ZPK-objekt, lahko za zapis modela uporabimo zapis z racionalno funkcijo. Poglejmo si primer.

```
>> s = zpk('s')

Zero/pole/gain:
s
```

S tem ukazom omogočimo vnos sistema *G* na naslednji način:

```
>> G = 3*(s+2)*(s+4)/((s+1)^2*(s+5))

Zero/pole/gain:
3 (s+2) (s+4)
-------------
(s+1)^2 (s+5)
```

Zdaj pa obravnavajmo še zapis obeh sistemov v polinomski obliki

$$G_{tf}(s) = \frac{n(s)}{d(s)} = \frac{b_m s^m + b_{m-1} s^{m-1} + \ldots b_0}{a_n s^n + a_{n-1} s^{n-1} + \ldots a_0} \ . \tag{8.4}$$

Funkcija *tf* potrebuje za vhodna argumenta vektorja koeficientov polinomov števca (*num*) in imenovalca (*den*) prenosne funkcije. Najprej izračunamo vektorja iz ničel in polov prenosne funkcije z ukazom *poly*.

```
>> num = 3*poly([-2 -4])

num =

     3    18    24

>> den = poly([-1 -1 -5])

den =

     1     7    11     5
```

Sedaj pa zapišimo prenosno funkcijo G_{t1} s pomočjo dobljenih rezultatov.

```
>> Gt1 = tf(num,den)

Transfer function:
  3 s^2 + 18 s + 24
----------------------
s^3 + 7 s^2 + 11 s + 5
```

Na podoben način kot prej lahko spremenljivko *s* definiramo tudi kot objekt tipa *TF*.

```
>> s = tf('s')

Transfer function:
s

>> Gt2 = (3*s^2+18*s+24)/(s^3+7*s^2+11*s+5)

Transfer function:
  3 s^2 + 18 s + 24
----------------------
s^3 + 7 s^2 + 11 s + 5
```

Opozorimo pa naj na razliko med definicijama spremenljivke *s*. Tip modela, ki ga bomo podali z racionalno funkcijo spremenljivke *s*, bo sledil definiciji tipa spremenljivke. Tako bo prenosna funkcija G_{t2} pretvorjena v model tipa *ZPK*, če bo *s* definirana na naslednji način:

```
>> s = zpk('s')

Zero/pole/gain:
s

>> Gt2 = (3*s^2+18*s+24)/(s^3+7*s^2+11*s+5)

Zero/pole/gain:
3 (s+4) (s+2)
-------------
(s+5) (s+1)^2
```

Za zapis v prostoru stanj izberemo linearno diferencialno enačbo drugega reda

$$\frac{d^2\varphi(t)}{dt^2}+3\frac{d\varphi(t)}{dt}+4\varphi(t)=5u(t) \tag{8.5}$$

in za stanji določimo $x_1=\varphi$ in $x_2=\dot{\varphi}$. Na ta način sistem zapišemo z enačbami prostora stanj

$$\begin{aligned}\dot{\mathbf{x}}&=\begin{bmatrix}\dot{x}_1\\ \dot{x}_2\end{bmatrix}=\mathbf{A}\mathbf{x}+\mathbf{b}u\\ y&=\mathbf{c}\mathbf{x}+du\end{aligned}, \tag{8.6}$$

kjer so matrike in vektorji

$$\mathbf{A}=\begin{bmatrix}0 & 1\\ -4 & -3\end{bmatrix},\ \mathbf{b}=\begin{bmatrix}0\\ 5\end{bmatrix},\ \mathbf{c}=\begin{bmatrix}0 & 1\end{bmatrix},\ d=0. \tag{8.7}$$

Ukaz v Matlabu, ki vrne objekt G_{ss} tipa *ss*, oblikujemo na naslednji način:

```
>> Gss = ss([0 1;-4 -3],[0;5],[0 1],0)

a =
       x1  x2
   x1   0   1
   x2  -4  -3


b =
       u1
   x1   0
   x2   5


c =
       x1  x2
   y1   0   1


d =
       u1
   y1   0

Continuous-time model.
```

8.2 Pretvorbe med tipi zapisov

Če imamo sistem zapisan v eni od opisanih oblik, potrebujemo pa zapis v drugi obliki, nam Matlab ponuja neposredno pretvorbo objektov s pomočjo ukazov *zpk*, *tf* in *ss* ter funkcije za posredno pretvorbo med tipi *TF*, *ZPK* in *SS*. Te funkcije so

- **zp2tf** (iz prenosne funkcije tipa *ZPK* v prenosno funkcijo tipa *TF*),
- **zp2ss** (iz prenosne funkcije tipa *ZPK* v prostor stanj),

- **tf2ss** (iz prenosne funkcije tipa *TF* v prostor stanj),
- **tf2zp** (iz prenosne funkcije tipa *TF* v prenosno funkcijo tipa *ZPK*),
- **ss2zp** (iz prostora stanj v prenosno funkcijo tipa *ZPK*),
- **ss2tf** (iz prostora stanj v prenosno funkcijo tipa *TF*).

Razlog za pretvorbo pa ni samo v estetiki zapisa, upoštevati moramo namreč tudi, da zapisi med seboj niso enakovredni glede numerične pogojenosti. V primeru prenosnih funkcij višjega reda velikokrat pride do nenatančne predstavitve, zato je bolje sistem zapisati v prostoru stanj. Lahko se zgodi, da se poli prenosne funkcije v polinomski obliki razlikujejo od polov enakovrednih predstavitev v faktorizirani obliki in prostoru stanj. Po drugi strani pa pretvorba v prostor stanj ni enoznačno definirana in ne zagotavlja minimalne realizacije. Zato se v praksi izogibamo pretiranemu pretvarjanju iz enega zapisa v drugega.

Imejmo sistem, podan z enačbo (8.1), zapisan v obliki *ZPK*.

```
>> z1 = [-2 -4];
>> p1 = [-1 -1 -5];
>> k1 = 3;
>> Gzz1 = zpk(z1,p1,k1)

Zero/pole/gain:
3 (s+2) (s+4)
-------------
(s+1)^2 (s+5)
```

Najprej si oglejmo neposredno pretvorbo v zapisa *TF* in *SS*. Funkciji uporabimo na način, kot je bil predstavljen v predhodnjem razdelku, edina razlika je v podajanju vhodnega argumenta, ki mora biti zdaj objekt tipov *ZPK*, *TF* ali *SS*.

```
>> Gtf1 = tf(Gzz1)

Transfer function:
  3 s^2 + 18 s + 24
----------------------
s^3 + 7 s^2 + 11 s + 5

>> Gss1 = ss(Gzz1)

a =
          x1     x2     x3
   x1     -1  0.866    0.5
   x2      0     -1  1.732
   x3      0      0     -5

b =
          u1
   x1      0
   x2      0
   x3  3.464
```

```
c =
          x1      x2      x3
   y1  1.732     1.5   0.866

d =
       u1
   y1   0

Continuous-time model.
```

Sedaj pa se posvetimo še posredni pretvorbi. Obravnavali bomo prvo, tretjo in peto pretvorbo od zgoraj omenjenih. Sistem najprej pretvorimo v prenosno funkcijo polinomskega tipa.

```
>> [num,den] = zp2tf(z1',p1',k1)

num =

     0     3    18    24


den =

     1     7    11     5
```

Kratka opomba – zakaj smo uporabili transponirana vektorja znotraj klica funkcije *zp2tf* (z1',p1',k1)? Enostavno zato, ker morata biti prva argumenta funkcije stolpna vektorja ničel in polov, mi pa smo prej uporabljali vrstična vektorja (funkciji *zpk* in *tf* ne razlikujeta med vrstičnimi in stolpnimi argumenti). Rezultat pretvorbe sta polinoma števca (*num*) in imenovalca (*den*) prenosne funkcije G_{tf}, ki jo nato lahko zapišemo z ukazom `Gtf = tf(num,den)`. Dobljeni rezultat zdaj pretvorimo v zapis v prostoru stanj. Pretvorbo ponazorimo z enačbo (8.8)

$$\mathbf{G}(s) = \mathbf{D} + \mathbf{C}(s\mathbf{I} - \mathbf{A})^{-1}\mathbf{B} \ , \tag{8.8}$$

kjer so **A, B, C** in **D** matrike modela v prostoru stanj (v splošnem z več vhodi in več izhodi), **G** pa je matrika prenosnih funkcij.

```
>> [A,B,C,D] = tf2ss(num,den)

A =

    -7   -11    -5
     1     0     0
     0     1     0

B =

     1
     0
     0
```

```
C =

     3    18    24

D =

     0
```

Iz dobljenih matrik pa znova sestavimo prenosno funkcijo v faktorizirani obliki.

```
>> [z2,p2,k2] = ss2zp(A,B,C,D)

z2 =

   -4.0000
   -2.0000

p2 =

   -5.0000
   -1.0000
   -1.0000

k2 =

    3.0000
```

Vidimo, da smo dobili enake parametre kot na začetku, opozorimo pa naj na to, da sta vektorja polov in ničel stolpna vektorja. Ostalih pretvorb se lotimo na analogen način – navedimo samo sintakso:

- [z,p,k] = tf2zp(num,den),
- [A,B,C,D] = zp2ss(z,p,k),
- [num,den] = ss2tf(A,B,C,D).

8.3 Lastnosti objektov

V prejšnjih podpoglavjih smo si ogledali, kako zapišemo LTI-sistem na 3 različne načine in kako med njimi prehajamo neposredno. Seveda pa je na voljo kar nekaj funkcij za obravnavanje lastnosti zapisanega sistema. V tem podpoglavju bo govora o tem, kako spreminjamo splošne in posebne lastnosti objektov LTI ter kako dobimo informacijo o sistemskih lastnostih obravnavanega objekta.

8.3.1 Splošne lastnosti objektov

Modeli LTI vsebujejo določeno število lastnosti, ki so vključene v vseh tipih modelov. Te lastnosti so polja objekta, katerim so prirejene vrednosti določenega tipa. Seznam in opis polj je prikazan v tabeli 8.1.

Tabela 8.1 - Tabela splošnih lastnosti LTI-modelov

Naziv lastnosti	Opis	Tip vrednosti
ioDelay	Vhodno-izhodne zakasnitve	Matrika
InputDelay	Zakasnitve vhodov	Vektor
InputGroup	Skupine vhodov	Struktura
InputName	Imena vhodov	Celični vektor nizov
Notes	Beležke o zgodovini modela	Tekst
OutputDelay	Zakasnitve izhodov	Vektor
OutputGroup	Skupine izhodov	Struktura
OutputName	Imena izhodov	Celični vektor nizov
Ts	Čas vzorčenja	Skalar
Userdata	Dodatni podatki	Poljubno

Polji *Notes* in *Userdata* se uporabljata za dodajanje podatkov o sistemu, *InputGroup* in *OutputGroup* pa omogočata povezovanje vhodov in izhodov v skupine z istim imenom. Podrobneje jih ne bomo obravnavali. *Ts* je čas vzorčenja sistema in ima po definiciji vrednost 0, če obravnavamo sistem z zvezno časovno spremenljivko, –1, če je čas nedoločena diskretna spremenljivka, in ≥0 za diskretni sistem s časom vzorčenja $T_s \geq 0$. Ostala polja razložimo na primeru sistema, zapisanega z enačbo (8.5), ki predstavlja tokovno gnan električni motor. Vhod v sistem je električni tok $i(t)$, izhod pa kotna hitrost gredi $\omega(t)$. Priredimo ti dve imeni tudi ustreznima poljema objekta:

```
>> Gss = ss([0 1;-4 -3],[0;5],[0 1],0);
>> set(Gss,'InputName','i(t)');
>> set(Gss,'OutputName','w(t)');
>> Gss

a =
       x1  x2
   x1   0   1
   x2  -4  -3

b =
       i(t)
   x1     0
   x2     5

c =
         x1  x2
   w(t)   0   1

d =
         i(t)
   w(t)     0

Continuous-time model.
```

Zdaj pa predpostavimo, da med nastopom spremembe vhoda in spremembo na izhodu sistema mine T_d = 0,2 s. To je mrtvi čas sistema in ga v objekt vključimo s pomočjo ukaza

```
>> set(Gss,'ioDelay',0.2) .
```

Poglejmo, kako se dodani mrtvi čas odraža na novem zapisu in prenosni funkciji sistema. Vtipkamo `Gss`, nato pa spremenimo zapis iz *SS* v *TF*.

```
>> Gss

a =
       x1  x2
   x1   0   1
   x2  -4  -3

b =
       i(t)
   x1     0
   x2     5

c =
         x1  x2
   w(t)   0   1

d =
         i(t)
   w(t)     0

I/O delay time (for all I/O pairs): 0.2

Continuous-time model.

>> Gtf = tf(Gss)


Transfer function from input "i(t)" to output "w(t)":
                  5 s
exp(-0.2*s) * -------------
              s^2 + 3 s + 4
```

Za vnos vhodnih ali izhodnih zakasnitev je postopek enak.

8.3.2 Posebne lastnosti objektov

Vsak od objektov ima tudi svoje posebne lastnosti. Ogledamo si jih lahko s pomočjo ukaza

```
>> ltiprops('tip_modela').
```

Tako na primer pri zapisu v prostoru stanj imena stanj določa polje z imenom *StateName*. Vzemimo prejšnji primer in poimenujmo stanji x_1 in x_2.

```
>> set(Gss,'StateName',{'fi(t)';'w(t)'})
>> Gss

a =
           fi(t)    w(t)
   fi(t)       0       1
   w(t)       -4      -3


b =
           i(t)
   fi(t)      0
   w(t)       5


c =
          fi(t)    w(t)
   w(t)       0       1


d =
          i(t)
   w(t)      0

I/O delay time (for all I/O pairs): 0.2

Continuous-time model.
```

Parameter *StateName* mora biti celično polje nizov dimenzije vektorja stanj (dim **x**). Celično polje vnesemo na enak način kot vektor, le da uporabimo zavite oklepaje.

8.3.3 Sistemske lastnosti objektov

Nekatere karakteristične lastnosti sistema lahko izračunamo s pomočjo vgrajenih funkcij. Pri tem ni pomembno, v kateri obliki imamo sistem zapisan. Izračunajmo najprej **ničle** in **pole** sistema

$$G_{tf}(s) = \frac{3\,s^2 + 18\,s + 24}{s^3 + 7\,s^2 + 11.25\,s + 6.25} \tag{8.9}$$

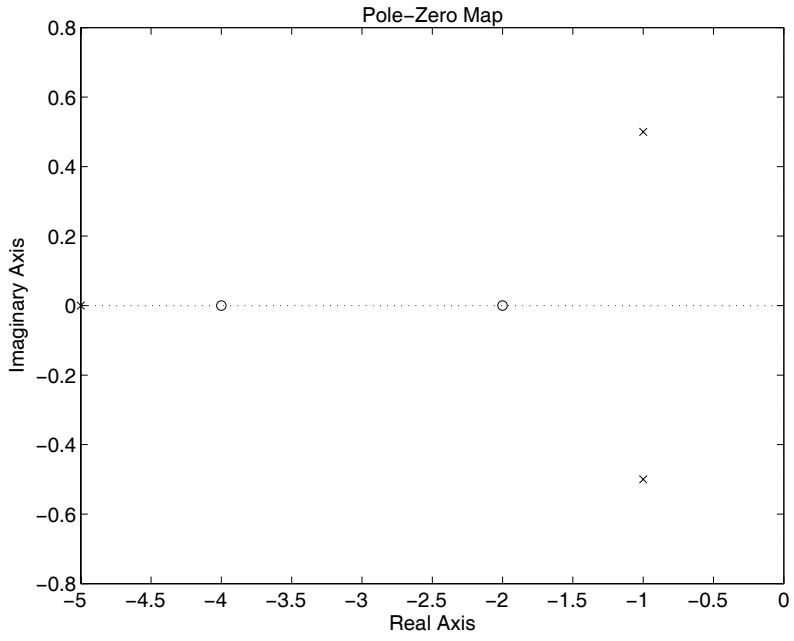
s pomočjo ukazov *zero* in *pole*.

```
>> z = zero(Gtf)

z =

    -4
    -2
```

```
>> p = pole(Gtf)

p =

  -5.0000
  -1.0000 + 0.5000i
  -1.0000 - 0.5000i
```

Obe funkciji lahko nadomestimo s funkcijo *pzmap*, ki, če podamo izhodna argumenta, vrne vektorja polov in ničel, drugače pa izriše **diagram polov in ničel** (slika 8.1).

```
>> [p,z] = pzmap(Gtf)

p =

  -5.0000
  -1.0000 + 0.5000i
  -1.0000 - 0.5000i

z =

    -4
    -2

>> pzmap(Gtf)
```



Slika 8.1 - Diagram polov in ničel prenosne funkcije

Ničle so označene s krožci, poli pa s križci.

S pomočjo funkcije *damp* izračunamo **naravne frekvence** (ω_n) in pripadajoče **koeficiente dušenja (ζ)** polov prenosne funkcije sistema. Funkcija brez izhodnih argumentov izpiše tabelo lastnih vrednosti sistema s pripadajočimi naravnimi frekvencami in koeficienti dušenja, razvrščenih po naraščajoči frekvenci. Ogledali pa si bomo klic funkcije z izhodnimi vektorji.

```
>> [wn,zeta] = damp(Gtf)

wn =

    1.1180
    1.1180
    5.0000


zeta =

    0.8944
    0.8944
    1.0000
```

Podamo lahko tudi dodatni izhodni parameter, ki nam poleg že omenjenih prinese še vektor polov sistema.

Ojačenje sistema v **ustaljenem stanju** podaja enačba

$$k_{ss} = \lim_{s \to 0} s \cdot G(s), \tag{8.10}$$

izračunamo pa ga s pomočjo ukaza *dcgain*.

```
>> Kss = dcgain(Gtf)

Kss =

    3.8400
```

Frekvenčno karakteristiko sistema

$$G(j\omega) = G(s)\big|_{s=j\omega} \tag{8.11}$$

izračunamo in predstavimo s funkcijama *freqresp* in *bode*. Prva vrne kompleksno vrednost frekvenčne karakteristike v točki $\omega = \omega_0$, druga pa isti kompleksor predstavi z dvajsetkratno vrednostjo desetiškega logaritma absolutne vrednosti in faznim kotom (v kotnih stopinjah). Podana frekvenca mora biti nenegativna vrednost v radianih na sekundo. Izračunajmo na primer vrednost frekvenčne karakteristike sistema $G_{tf}(j\omega)$ pri $\omega = 0{,}2$ rad/s

```
>> fr1 = freqresp(Gtf,0.2)

fr1 =

   3.7041 - 0.7880i
```

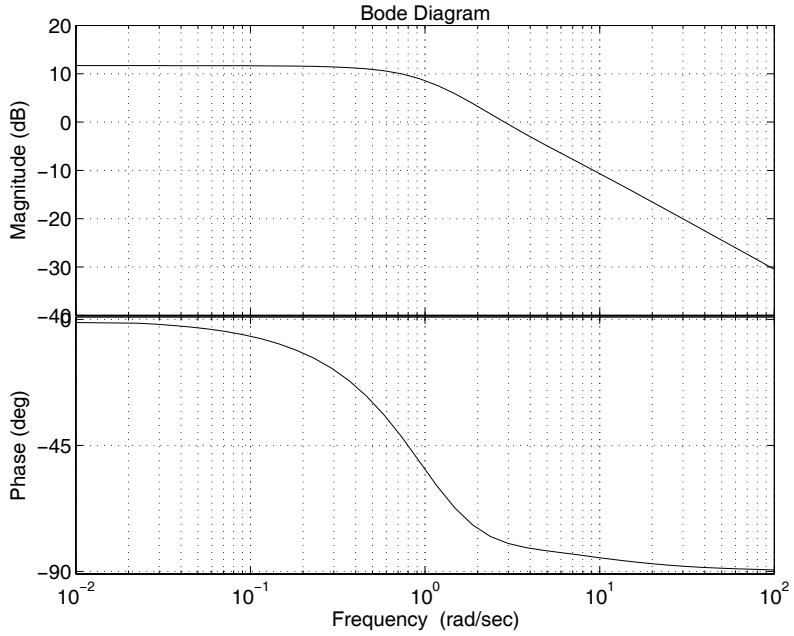
```
>> [ab,faz] = bode(Gtf,0.2)

ab =

    3.7870

faz =

  -12.0104
```

Seveda pa se uporabna vrednost obeh funkcij pokaže, če podamo vektor frekvenc znotraj frekvenčnega območja, v katerem želimo izračunati frekvenčno karakteristiko. Če je dimenzija vektorja frekvenc dim w = N, vrneta funkciji tridimenzionalno polje dimenzij (št. izhodov sistema)x(št. vhodov sistema)xN. Funkcija *bode* brez izhodnih argumentov in brez podanega vektorja frekvenc nariše t.i. **Bodejev diagram**, kjer program sam izbere najprimernejše frekvenčno območje. Poglejmo si primera. Najprej izračunajmo frekvenčno karakteristiko med frekvencama $\omega_1 = 10^{-3}$ rad/s in $\omega_2 = 10^3$ rad/s v stotih logaritemsko porazdeljenih točkah (uporabimo funkcijo *logspace*) in prikažimo vrednosti v 50. točki karakteristike.

```
>> w = logspace(-3,3,100);
>> fr = freqresp(Gtf,w);
>> [ab,faz] = bode(Gtf,w);
>> size(fr)

ans =

     1     1   100

>> w(50)

ans =

    0.9326

>> fr(1,1,50)

ans =

   1.7705 - 2.1801i

>> [ab(1,1,50) faz(1,1,50)]

ans =

   2.8084   -50.91899
```

Pri frekvenci ω = 0.9326 rad/s torej dobimo ojačenje sistema 2.8084 dB in fazni kot –50.9189°. Narišimo še Bodejev diagram (slika 8.2).

```
>> bode(Gtf)
```



Slika 8.2 - Bodejev diagram prenosne funkcije G_{tf}

Pasovno širino sistema, tj. frekvenco, kjer ojačenje ustaljenega stanja pade za 3 dB, izračunamo s pomočjo funkcije *bandwidth*.

```
>> wb = bandwidth(Gtf)

wb =

    0.9841
```

Nyquistov diagram ponazarja funkcija *nyquist*, ki izriše Nyquistovo frekvenčno karakteristiko v frekvenčnem območju, določenem na podlagi polov in ničel sistema. Tudi tu lahko frekvenčno območje podamo kot vhodni vektor, prav tako pa funkcija vrne vektorje točk (brez izrisa diagrama), če podamo izhodne argumente. Narišimo Nyquistov diagram sistema G_{tf} (slika 8.3).

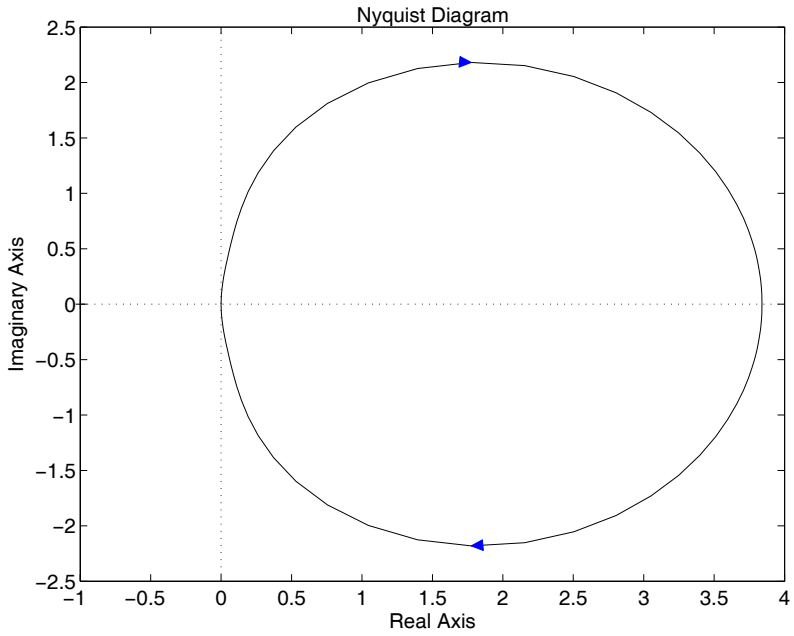
```
>> nyquist(Gtf)
```

Diagram lege korenov narišemo s pomočjo funkcije *rlocus*. Funkcija nariše potek zaprtozančnih polov sistema v ravnini *s*, ko gre parameter *K* odprtozančnega sistema

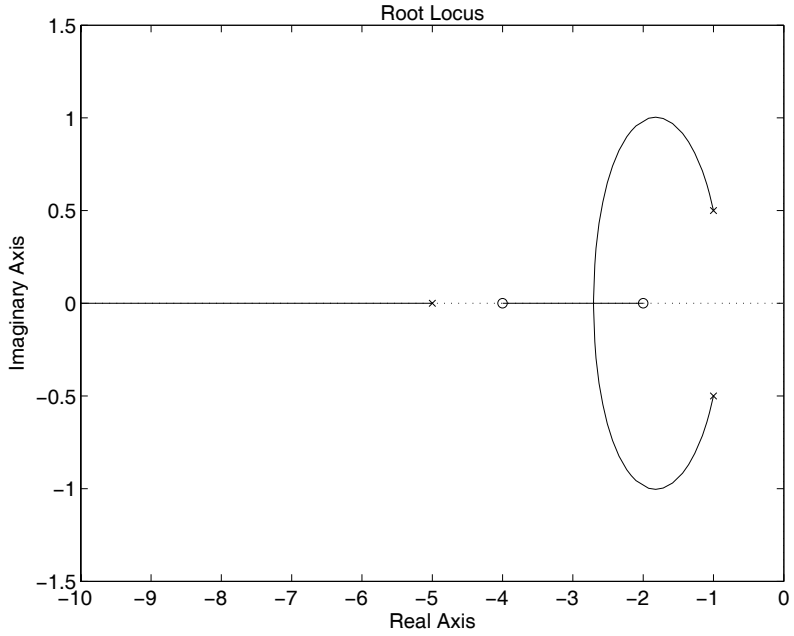
$$K \cdot GH(s) \tag{8.12}$$

od 0 do ∞. Če želimo narisati diagram samo za določeno območje *K*, to podamo z vektorjem vrednosti v vhodnih argumentih funkcije. Za dani sistem narišimo DLK in ga prikažimo na sliki 8.4.

```
>> rlocus(Gtf)
```



Slika 8.3 - Nyquistov diagram prenosne funkcije G_{tf}



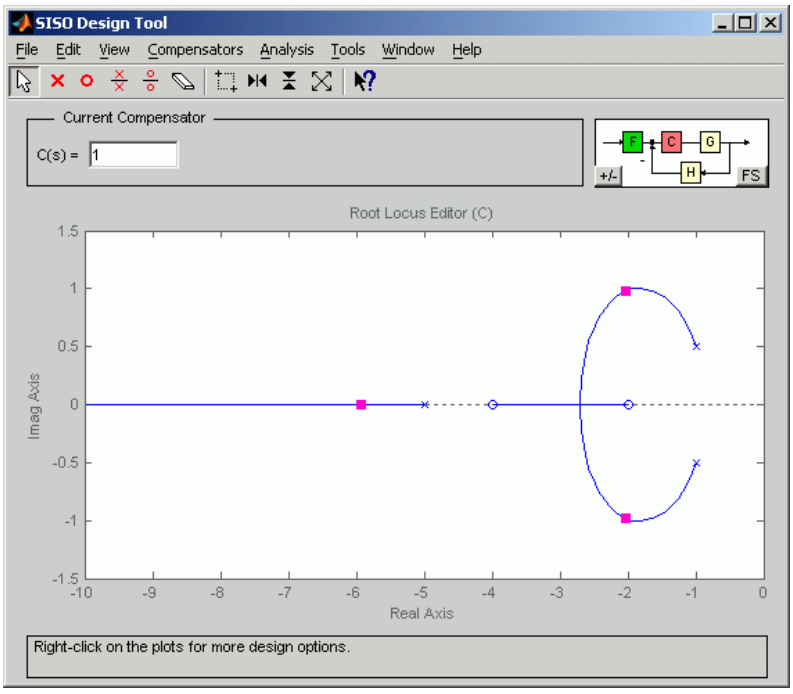
Slika 8.4 - Diagram lege korenov za dani sistem

Klic funkcije `[r,k] = rlocus(Gtf)` vrne vektor ojačenj k in matriko r, kjer so lokacije zaprtozančnih polov pri ustreznih ojačenjih parametra K.

Matlab pa ponuja tudi uporabniški vmesnik za risanje diagrama lege korenov *rltool*, ki je del širšega uporabniškega analitično-načrtovalskega vmesnika *sisotool*, katerega pa ne bomo posebej predstavljali. Orodje *rltool* je prikladno za načrtovanje kompenzatorjev po metodi DLK in za analizo zaprtozančnega sistema pod določenimi omejitvami. Poglejmo si primer uporabe. Za odprtozančni sistem G_{tf} želimo načrtati P–regulator v direktni veji, da bo dušenje $\zeta < 0.91$ (maksimalen

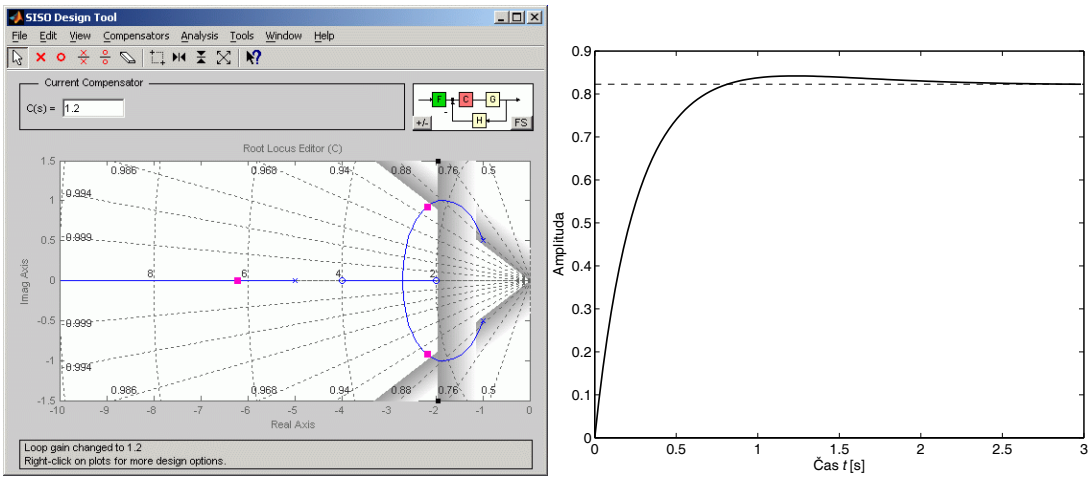
prevzpon < 10%) in umiritveni čas T_u < 2 s. S pomočjo orodja *rltool* lahko določimo območje zaprtozančnih polov v skladu z omejitvama. Najprej narišemo DLK G_{tf}.

```
>> rltool(Gtf)
```



Slika 8.5 - Izgled DLK v uporabniškem vmesniku *rltool*

V podmeniju *Edit/Root Locus/Design Constraints* izberemo *New* in v oknu, ki se nam odpre, definiramo *Settling time* = 2 in *Damping* = 0,91. Na levem delu slike 8.6 je prikazano območje, kjer naj ležijo zaprtozančni poli. S pomočjo miške pole nato enostavno premaknemo v želeno območje in odčitamo vrednost ojačenja. Desni del slike 8.6 prikazuje odziv zaprtozančnega sistema na stopnico po vnosu predlaganega ojačenja K = 1,2.

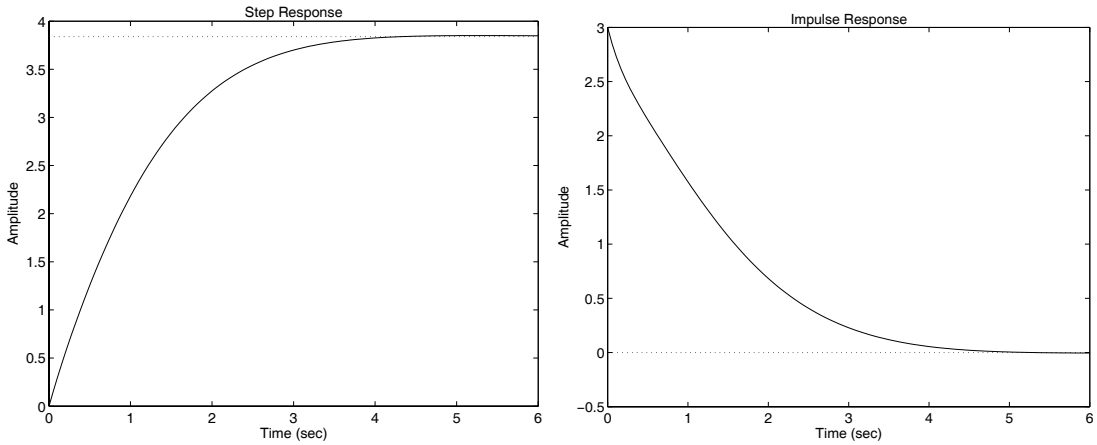


Slika 8.6 - Levo - DLK z omejitvami in podatkom o ojačenju regulatorja, desno – odziv zaprtozančnega sistema na stopnico po vnosu predlaganega ojačenja

8.4 Časovni odziv

Osnovna odziva prenosne funkcije v časovnem prostoru sta odziv na stopnico in impulzni odziv. Prvega dobimo s pomočjo funkcije *step*, drugega pa s klicem funkcije *impulse*. Tako kot že nekajkrat prej nam klic funkcij brez izhodnih argumentov nariše diagram odziva, lahko pa izhoda funkcij preusmerimo v vektorja odziva in časa. Za G_{tf} narišimo oba odziva in ju prikažimo na sliki 8.7.

```
>> step(Gtf)
>> impulse(Gtf)
```



Slika 8.7 - Levo - odziv na stopnico, desno - impulzni odziv

Če pa hočemo izračunati odziv na poljubno vzbujanje, uporabimo funkcijo *lsim*. Kot vhodne argumente podamo sistem, vektor vzbujanja in vektor pripadajočih časov, izhod pa spet usmerimo na diagram ali v izhodna vektorja odziva in pripadajočega časa. Pri tvorjenju periodičnih vhodnih signalov si lahko pomagamo s funkcijo [*u*,*t*] = *gensig*('*tip*',*tau*,*Tkon*,*Ts*), ki naredi periodični signal s periodo *tau*, dolžine *Tkon***Ts* in razmakom med dvema vzorcema *Ts*. Podprti tipi signalov so kvadratni periodični signal ('*square*'), sinusoida ('*sin*') in pulzni signal ('*pulse*'). Poglejmo si primer. Izračunajmo odziv sistema G_{tf} na kvadratni signal s periodo $\tau = 1$ s, končnim časom $T_{kon} = 10$ s in časom vzorčenja $T_s = 0{,}02$ s ter ga predstavimo na sliki 8.8.

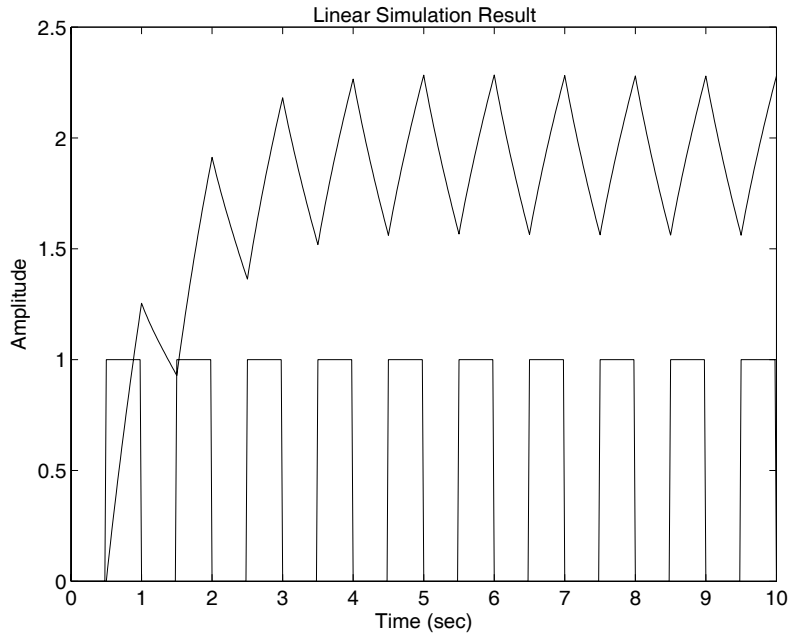
```
>> [u,t] = gensig('square',1,10,0.02);
>> lsim(Gtf,u,t)
```

8.5 Vezave sistemov

Poglejmo si še medsebojne vezave sistemov. Pri vseh funkcijah, ki jih bomo obravnavali, je vseeno, v kakšni obliki so podani sistemi. **Zaporedno vezavo** dveh sistemov LTI dobimo s funkcijo *series* ali množenjem objektov. Znak za množenje je tu definiran kot množenje dveh prenosnih funkcij. Vzemimo prenosni funkciji

$$G_1(s) = \frac{1}{(s+2)^2}, \quad G_2(s) = \frac{5}{(s+2)^2} \tag{8.13}$$

in ju povežimo serijsko.



Slika 8.8 - Odziv sistema G_{tf} s pomočjo funkcije *lsim*

```
>> G1 = zpk([],[-2],1);
>> G2 = zpk([],[-2],5);
>> Gs = series(G1,G2)

Zero/pole/gain:
   5
-------
(s+2)^2
```

Vzporedno vezavo sistemov izvedemo z ukazom *parallel* ali s seštevanjem/odštevanjem prenosnih funkcij. Naredimo primer vzporedne vezave za obe operaciji.

```
>> Gp1 = parallel(G1,G2)

Zero/pole/gain:
6 (s+2)
-------
(s+2)^2

>> G1-G2

Zero/pole/gain:
-4 (s+2)
--------
(s+2)^2
```

Povratno zanko dobimo z uporabo ukaza *feedback*. Vhodni argumenti so sistem v direktni veji, sistem v povratni veji in predznak povezave. Privzeta vrednost

povezave je –1 (negativna povratna vezava), če pa želimo pozitivno povratnozančno strukturo, uporabimo klic funkcije

```
sis = feedback(sis1,sis2,1).
```

Funkcija *feedback* vrne povratnozančno prenosno funkcijo

$$G_{pv} = \frac{G_{sis1}}{1 \pm G_{sis1} G_{sis2}}. \tag{8.14}$$

Naredimo za dana sistema negativno povratno vezavo.

```
>> Gpv = feedback(G1,G2)

Zero/pole/gain:
     (s+2)
---------------
(s^2  + 4s + 9)
```

8.6 Načrtovanje regulatorja stanj

Za konec si oglejmo še funkciji, ki pomagata določiti ojačenja regulatorja stanj za sistem, zapisan v obliki *SS*. To sta *place* in *acker*. Pri obeh podamo sistemsko matriko **A,** vhodni vektor **b** (za sisteme MISO in MIMO vhodno matriko **B**) in vrstični vektor želenih zaprtozančnih polov **p**, funkciji pa vrneta vektor ojačenj **k**, da je izpolnjena enačba

$$\mathrm{eig}(\mathbf{A} - \mathbf{bk}) = \mathbf{p} \tag{8.15}$$

Edina razlika je v načinu računanja – funkcija *acker* za izračun ojačenj uporablja Ackermannovo formulo

$$\mathbf{k} = \begin{bmatrix} 0 & 0 & \dots & 0 & 1 \end{bmatrix} \mathbf{Q}_v^{-1} \mathbf{\Phi}(\mathbf{A}), \tag{8.16}$$

kjer je $\mathbf{Q}_v$ vodljivostna matrika, **A** sistemska matrika, $\mathbf{\Phi}(\mathbf{A})$ pa funkcija, ki zadostuje Cayley-Hamiltonovemu teoremu in določa želeni karakteristični polinom. Pri obravnavi velikih sistemov (več kot 10 stanj) in slabo vodljivih sistemov je funkcija *acker* lahko numerično slabše pogojena, zato se v primeru, ko se dobljeni poli preveč razlikujejo od želenih, na zaslon izpiše opozorilo.

Za sistem iz enačbe (8.7) načrtajmo regulator stanj, ki bo premaknil pola v želeni legi $s_1 = -3$ in $s_2 = -4$.

```
>> Gss = ss([0 1;-4 -3],[0;5],[0 1],3);
>> p = [-3 -4];
>> k1 = place(Gss.a,Gss.b,p)

k1 =

    1.6000    0.8000

>> k2 = acker(Gss.a,Gss.b,p)
```

```
k2 =

    1.6000    0.8000
```

Preverimo s pomočjo formule v enačbi (8.15), ali smo res dobili želene lastne vrednosti sistemske matrike zaprtozančnega sistema.

```
>> eig(Gss.a-Gss.b*k1)

ans =

   -3.0000
   -4.0000
```

9. KNJIŽNICA FUNKCIJ ZA SIMBOLIČNO RAČUNANJE (SYMBOLIC TOOLBOX)

Matlabovo orodje za simbolično računanje (Symbolic toolbox) je namejeno simbolični obravnavi funkcij in spremenljivk. Po funkcionalnosti je podoben programskemu paketu Mathematica, le da je njegova uporaba bolj preprosta. Uporabljamo ga za reševanje navadnih in diferencialnih enačb, računanje Laplaceove in inverzne Laplaceove transformacije in še za mnoge druge analitične postopke.

Spremenljivke, ki jih uporabljamo, moramo deklarirati kot simbolične, šele nato lahko z njimi sestavljamo različne izraze in rešujemo podane enačbe. Največ težav pa lahko pričakujemo pri končnih rešitvah zapletenih sistemov, saj Matlab ne zna poenostavljati izrazov tako dobro kot človek, kar lahko privede do nenavadno zapletenih izrazov. Zato je sprotna interpretacija dobljenih rezultatov še vedno zelo pomembna.

V tem poglavju si bomo pogledali osnovne funkcije in podali kratke ilustrativne primere uporabe, nato pa bomo s pomočjo podanih funkcij rešili konkretno matematično nalogo na analitičen način.

9.1 Opis osnovnih funkcij

Poglejmo si opis nekaj najbolj pomembnih funkcij. Z uporabo osnovnih funkcij lahko praviloma rešimo večino analitičnih nalog, na katere naletimo v okviru laboratorijskih vaj.

S funkcijo **syms** deklariramo neodvisno spremenljivko kot simbolično.

```
>> syms t
```

V naslednjem koraku definiramo funkcijo neodvisne spremenljivke. Izberemo lahko poljubno izmed funkcij, ki smo jih že obravnavali.

```
>> y = sin(t)

y =

sin(t)
```

Brez klica funkcije *syms* bo Matlab ob klicu funkcije spremenljivke *t* izvedel numerično računanje odvisne spremenljivke *y*.

S pomočjo funkcije **diff** lahko izračunamo odvod funkcije, npr.:

```
>> x = diff(y)

x =

cos(t)
```

V klicu funkcije lahko podamo tudi spremenljivko, po kateri naj izraz odvaja. Poglejmo si primer preproste funkcije dveh spremenljivk $f(x,t) = 2x^2t$.

```
>> syms x t
>> diff(2*x^2*t,t)

ans =

2*x^2

>> diff(2*x^2*t,x)

ans =

4*x*t
```

Funkcija **int** omogoča izračun simboličnega integrala funkcije.

```
>> y = 1/(1+x^2);
>> x = int(y)

x =

atan(x)
```

S funkcijo **limit** izračunamo limito simboličnega izraza.

```
>> limit((x-3)/(x^2-7),2)

ans =

1/3

>> syms h
>> limit((sin(x+h)-sin(x))/h,h,0)

ans =

cos(x)
```

Drugi primer nam oriše, kako sta povezana limita in analitični odvod funkcije v določeni točki.

Funkcija **simplify** služi poenostavitvi izrazov. Uporabljamo jo v glavnem zato, da razultat prikažemo z bolj preglednim izpisom.

```
>> simplify((sin(x)^2 + cos(x)^2)*sin(2*x))

ans =

2*sin(x)*cos(x)
```

S pomočjo funkcije **factor** dano število razcepimo na prafaktorje. Funkcija vrne vektor prafaktorjev, zapisanih v naraščajočem zaporedju.

```
>> factor(84)

ans =

     2     2     3     7
```

Funkcija **simple** na podanem izrazu preizkusi različne metode poenostavljanja (vseh ne bomo obravnavali, omenimo le *combine*, *collect*, *convert* in *expand*), rezultate prikaže na zaslon in vrne najkrajši možen rezultat. Če v izhodnih parametrih podamo tudi drugi argument *how*, nam izpiše samo najboljši rezultat in metodo, po kateri je do njega prišla.

```
>> [r,how] = simple(cos(x)^2-sin(x)^2)

r =

cos(2*x)


how =

combine
```

Če hočemo dolg in nepregleden vrstični zapis funkcije polepšati, uporabimo funkcijo **pretty**.

```
>> pretty((1-tan(1/2*x)^2)^2/(1+tan(1/2*x)^2)^2-...
4*tan(1/2*x)^2/(1+tan(1/2*x)^2)^2)

                          2 2                   2
                 (1 - tan(1/2 x) )          tan(1/2 x)
                 ------------------ - 4 ------------------
                          2 2                     2 2
                 (1 + tan(1/2 x) )       (1 + tan(1/2 x) )
```

S funkcijo **solve** izračunamo analitično rešitev algebrajske enačbe ali sistema algebrajskih enačb. Če analitična rešitev ne obstaja, funkcija izračuna rešitev numerično.

```
>> solve('sin(t)=1',t)

ans =

1/2*pi
```

S funkcijo **dsolve** izračunamo rešitev diferencialne enačbe ali sistema diferencialnih enačb. Odvod funkcije označimo z veliko črko *D*, višje odvode pa s številko, ki sledi črki *D*, npr. *D2*. Če podamo začetne pogoje enačbe, dobimo celotno rešitev enačbe.

```
>> syms y
>> dsolve('D3y+3*D2y+2*Dy=10','y(0)=0,Dy(0)=1,D2y(0)=2')

ans =

5*t-5-exp(-2*t)+6*exp(-t)
```

Če začetnih pogojev ne podamo, dobimo samo splošno rešitev, ki je definirana do konstant natančno.

```
>> syms y
>> dsolve('D3y+3*D2y+2*Dy=10')

ans =

5*t+C1+C2*exp(-2*t)+C3*exp(-t)
```

Funkcija **laplace** omogoča izračun Laplaceove transformacije izraza.

```
>> syms t
>> laplace(sin(t))

ans =

1/(s^2+1)
```

Inverzno Laplaceovo transformacijo izračunamo s pomočjo funkcije **ilaplace**.

```
>> syms s
>> ilaplace(1/(s^2+1))

ans =

sin(t)
```

Funkcija **ezplot** omogoča risanje grafov simboličnih izrazov. Ukaz `ezplot(y)` nariše izraz, ki se skriva pod spremenljivko *y* v mejah, ki jih določi algoritem glede na izraz.

S funkcijo **subs** zamenjamo spremenljivke v izrazu.

```
>> subs(y,'t','m')

ans =

sin((m))
```

V izrazu $y = \sin(t)$ smo neodvisno spremenljivko *t* zamenjali z *m.*

Funkcija **vpa** prisili orodje k računanju z omejeno natančnostjo. V podanem izrazu izračuna vse konstante na podano število mest natančno v zapisu s plavajočo vejico.

```
>> vpa(pi,20)

ans =

3.1415926535897932385
```

9.2 Ilustrativen primer uporabe

S pomočjo simboličnega orodja bomo rešili sistem enačb in rešitev narisali. Privzemimo, da so začetni pogoji sistema enaki nič.

$$\begin{aligned} &3\ddot{y}_1(t)+12\left(\dot{y}_1(t)-\dot{y}_2(t)\right)+3\left(y_1(t)-y_2(t)\right)=0 \\ &5\ddot{y}_2(t)+12\left(\dot{y}_2(t)-\dot{y}_1(t)\right)+2\dot{y}_2(t)+3\left(y_2(t)-y_1(t)\right)=2u(t) \\ &u(t)=\begin{cases}0;\ t\le 0\\ 2;\ t>0\end{cases} \end{aligned} \qquad (9.1)$$

Najprej na sistemu enačb izvedemo Laplaceovo transformacijo, da se izognemo reševanju sistema diferencialnih enačb. Ker orodje ne omogoča neposredne pretvorbe diferencialnih enačb s pomočjo Laplaceove transformacije v navadne algebrajske enačbe, jih najenostavneje pretvorimo preko tabel.

$$\begin{aligned} &3s^2Y_1(s)+12s\left(Y_1(s)-Y_2(s)\right)+3\left(Y_1(s)-Y_2(s)\right)=0 \\ &5s^2Y_2(s)+12s\left(Y_2(s)-Y_1(s)\right)+2sY_2(s)+3\left(Y_2(s)-Y_1(s)\right)=2U(s) \\ &U(s)=\frac{2}{s} \end{aligned} \qquad (9.2)$$

Enačbe sedaj prepišemo v Matlabovo ukazno vrstico.

```
>> syms t s
>> en1 = '3*s^2*Y1+12*s*(Y1-Y2)+3*(Y1-Y2)=0';
>> en2 = '5*s^2*Y2+12*s*(Y2-Y1)+2*s*Y2+3*(Y1-Y2)=2*U';
```

V naslednjem koraku izračunamo rešitve sistema enačb.

```
>> Y = solve(en1,en2,'Y1','Y2')

Y =

      Y1:[1x1 sym]
      Y2:[1x1 sym]
```

Dobimo torej rešitvi za $Y_1(s)$ in $Y_2(s)$, ki si ju lahko ogledamo z naslednjimi ukazi.

```
>> Y.Y1

ans =

2*U*(4*s+1)/s/(5*s^3+34*s^2+10*s+2)

>> Y.Y2

ans =

2*U/s/(5*s^3+34*s^2+10*s+2)*(s^2+4*s+1)

>> pretty(Y.Y1)

                          U (4 s + 1)
                 2 --------------------------
                          3       2
                    s (5 s  + 34 s  + 10 s + 2)
>> pretty(Y.Y2)

                            2
                       U (s  + 4 s + 1)
                 2 --------------------------
                          3       2
                    s (5 s  + 34 s  + 10 s + 2)
```

Rešitev ob definiranem signalu $U(s)$ pa dobimo tako, da v rešitvi zamenjamo spremenljivko U z $2/s$.

```
>> Y.Y1 = subs(Y.Y1,'U','2/s');
>> Y.Y2 = subs(Y.Y2,'U','2/s');
>> pretty(Y.Y1)

                             4 s + 1
                 4 --------------------------
                     2     3       2
                    s  (5 s  + 34 s  + 10 s + 2)
```

V naslednjem koraku izračunamo inverzno Laplaceovo transformacijo.

```
>> y1 = ilaplace(Y.Y1);
>> y2 = ilaplace(Y.Y2);
```

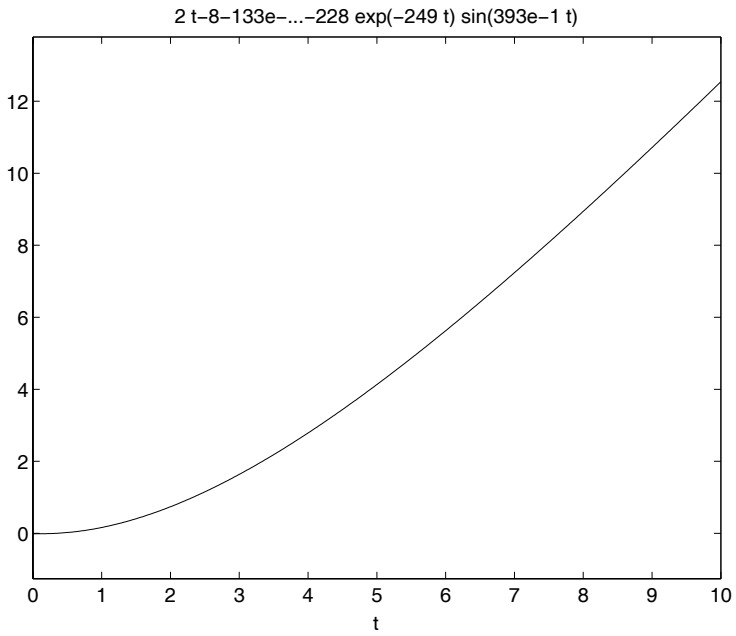
Vendar pa rešitev ni nujno v najbolj pregledni obliki (rešitev za y_1 ima kar 122 znakov). Največkrat se težava skriva v dejstvu, da simbolično orodje podaja rezultate v absolutno točni obliki, kar v veliko primerih povroči neuporaben izpis. Temu se izognemo tako, da zahtevamo rezultat z določeno natančnostjo (ukaz **vpa**), nato pa izraz poenostavimo.

```
>> y1 = vpa(y1,3);
>> y2 = vpa(y2,3);
```

```
>> y1 = simple(y1);
>> y2 = simple(y2);
>> pretty(y1)

2. t - 2. - .0117 exp(-6.50 t) + 2.01 exp(-.149 t) cos(.198 t)

          - 8.96 exp(-.149 t) sin(.198 t)
```

Sedaj lahko krivuljo še narišemo v časovnem intervalu [0,10].

```
>> ezplot(y1,[0,10])
```



Slika 9.1 - Analitična rešitev funkcije y_1, narisana z **ezplot**

10. UVOD V SIMULINK 6.0

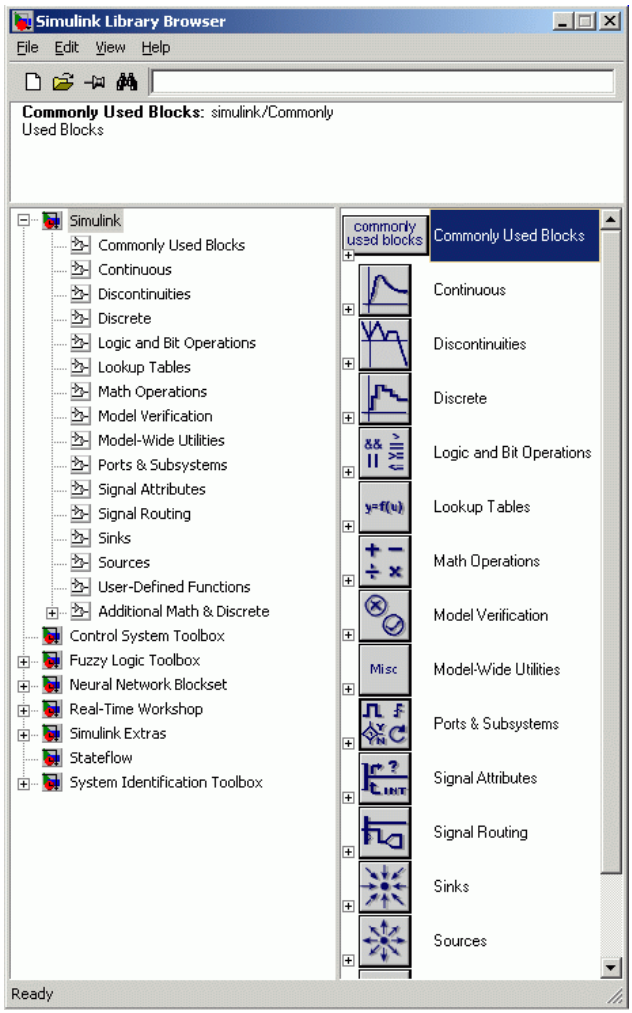
SIMULINK® 6.0 je programski paket za modeliranje, simulacijo in analizo dinamičnih sistemov, ki podpira tako linearne kot nelinearne časovno-zvezne in diskretne sisteme. Za gradnjo modelov ponuja grafični uporabniški vmesnik, kjer s pomočjo miške bloke povezujemo v kompleksnejše diagrame. Vključuje bogato knjižnico signalnih virov in ponorov, linearnih in nelinearnih elementov ter konektorjev. Modele lahko zgradimo modularno in z odpiranjem sklopov posegamo na nižje nivoje. Ko je model definiran, ga simuliramo s pomočjo ene od mnogih integracijskih metod iz Simulinkovega okna ali iz ukazne vrstice Matlaba. Delo v oknu je interaktivno, druga opcija pa je bolj primerna za serije simulacij v različnih zankah, npr. pri optimizaciji. S pomočjo prikazovalnikov lahko med potekom simulacije spremljamo, kaj se dogaja s signali, izhode pa lahko preusmerimo v delovni prostor Matlaba v spremenljivke, ki so na voljo za nadaljno obdelavo. Analiza vključuje predvsem orodja za linearizacijo (*linmod*) in iskanje ravnotežnih točk (*trim*). In ker sta Simulink in Matlab tesno povezana, lahko modele vsak trenutek simuliramo, analiziramo in popravljamo v kateremkoli od obeh okolij.

V priročniku bomo obravnavali glavne sklope Simulinka in nekatere najbolj uporabne bloke vsakega od sklopov. Podali bomo pregled map z elementi in pri vsaki omenili glavni namen uporabe. Za elemente, ki se najbolj uporabljajo, bo uporaba orisana s preglednimi eksperimenti. Spoznali se bomo tudi s postopki gradnje nelinearnih modelov in pokazali, kako lahko na učinkov način uporabljamo kombinacijo metod Matlaba in Simulinka. Na koncu bomo razložili še, kako zgradimo in uporabljamo s–funkcije. S programiranjem s–funkcij in vključevanjem kode, napisane v jezikih fortran ali c, v Simulinkove sheme se nam možnosti simulacije in prikazovanja rezultatov razširijo do neslutenih meja.

Poglavja o Simulinku bodo bralcu podala dovolj znanja, da bo sam lahko rešil prenekateri problem modeliranja, simulacije, vodenja sistemov, optimizacije in identifikacije. Za nekatere bolj poglobljene funkcije ali parametre funkcij pa je seveda na voljo pomoč v Matlabu v obliki html-dokumenta.

11. KAKO ZAČETI Z DELOM V SIMULINKU

V Matlabovem oknu kliknemo na Simulinkovo ikono ali v ukazni vrstici napišemo ukaz *simulink* in odpre se nam začetno okno grafičnega uporabniškega vmesnika, kot je prikazano na sliki 11.1.



Slika 11.1 - Začetno Simulinkovo okno

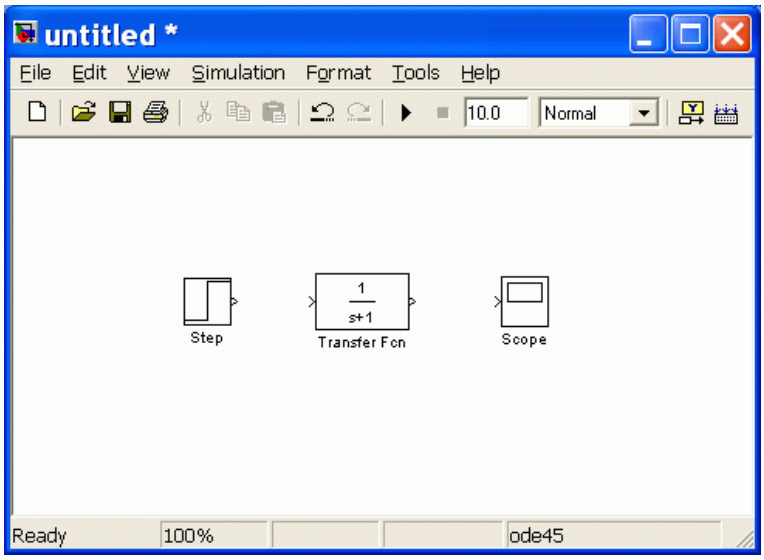
V glavni mapi imamo mapo *Simulink* (osnovna mapa Simulinka) in ostale mape, ki se nanašajo na vključene knjižnice orodij. Simulink ima 17 osnovnih podmap in vsebino večine si bomo pobliže ogledali. Odpremo jih na dva načina; s klikom z levo tipko miške na ime ali ikono mape dobimo seznam elementov na desni strani okna, s klikom z desno miškino tipko in izbiro *Odpri* pa se celotna podmapa odpre v novem oknu. Okno delovne sheme odpremo v meniju *File*/*New*/*Model* ali z bližnjico Ctrl–N ali s klikom na ikono v osnovnem oknu. V shemo nato z miško prenašamo elemente iz podmap osnovnega okna in jih med sabo z miško povezujemo v sheme.

11.1 Tvorjenje preproste simulacijske sheme

Za primer si poglejmo, kako dobimo odprtozančni odziv procesa

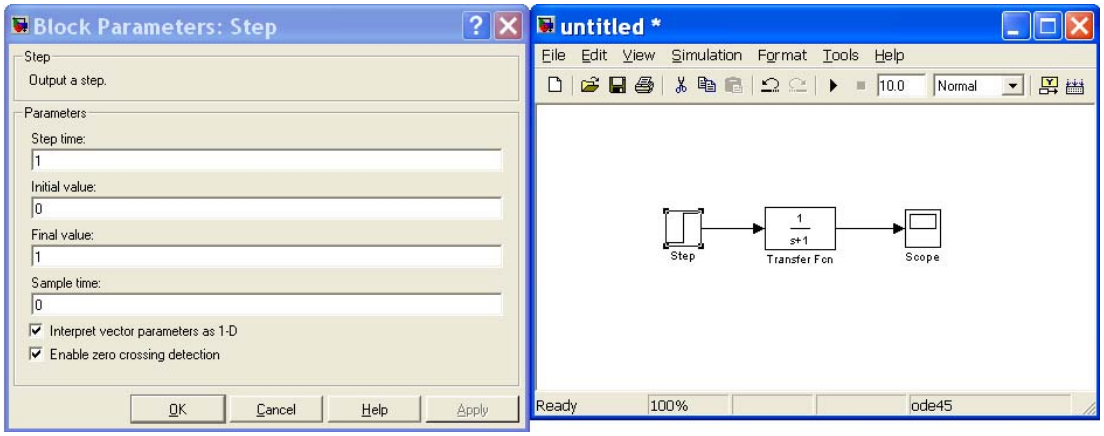
$$G_p = \frac{1}{s+1} \tag{11.1}$$

na vzbujanje s stopnico, ki pri T = 5 s zavzame vrednost R = 2. Želimo opazovati časovni potek signala izhoda procesa. Najprej odpremo mapo *Continuous* in ikono *Transfer function* z miško odvlečemo v naš diagram. Ker je prizveza nastavitev prava, ne potrebujemo nastavljanja parametrov elementa. Nato iz mape *Sources* prenesemo element *Step* in iz mape *Sinks* element *Scope* (slika 11.2)

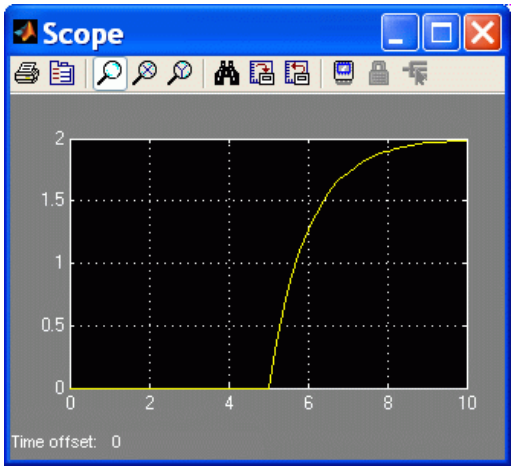


Slika 11.2 - Osnovni elementi v diagramu

Elemente moramo sedaj povezati v smislu poteka signalov. Izvir in ponor signala sta na vsakem elementu označena s puščico, ki kaže iz elementa (izhod bloka) ali v element (vhod bloka). Prenosna funkcija ima oba znaka, ki določata, v kateri smeri jo je treba povezati. To storimo tako, da z miško potegnemo črto od znaka izvora do znaka ponora na elementih. Ta verzija Simulinka pa nam ponuja tudi bližnjico za povezovanje – najprej kliknemo na element, iz katerega želimo potegniti črto, nato pa ob pritisku na tipko Ctrl z miško kliknemo na element, do katerega želimo potegniti črto, in Matlab to naredi namesto nas. Bližnjica je uporabna predvsem v velikih shemah z veliko povezavami. Ko smo povezali bloke med seboj, nastavimo parametre izvora. Dvakrat kliknemo na objekt in odpre se nam okno s parametri (slika 11.3). V rubriko *Step time* vnesemo čas, ko se vrednost stopnice spremeni od začetne v končno vrednost, to je 5, v rubriko *Final value* pa končno vrednost 2. Simulacijo poženemo s klikom na ikono ▶ ali v meniju *Simulation*/*Start* ali z bližnjico Ctrl-T. nato z dvoklikom odpremo objekt *Scope* in kliknemo ikono *Autoscale* (🔭) za skaliranje osi diagrama s skladu z vrednostmi odziva. Na sliki 11.4 je prikazano okno objekta *Scope*.



Slika 11.3 - Povezan diagram z odprtim oknom lastosti objekta *Step*



Slika 11.4 - Okno *Scope* za prikaz časovnih potekov signalov

11.2 Delo z objekti

Pogledali si bomo osnovne operacije z bloki v shemah Simulinka. Dobro poznavanje teh operacij nam v veliki meri skrajša čas, potreben za izvedbo simulacije glede na predpisane postavke.

11.2.1 Oblika kurzorja

Oblika kurzorja se spreminja glede na njegovo trenutno funkcijo. Osnovna oblika je puščica, ki kaže z desne na levo navzgor. Funkcije kurzorja naslednje:

- **puščica** v levo navzgor – pripravljen za naslednjo akcijo ali premikanje bloka;
- **puščica s plusom** – kopiranje objekta, tako da original z desnim klikom potegnemo na drugo lokacijo**;**
- **križ** – risanje povezave (pritisnjena leva miškina tipka) ali bližnjica povezave (pritisnjena tipka Ctrl in kurzor miške na objektu, do katerega želimo povezavo potegniti)
- **puščica v smeri raztega** – spreminjanje velikosti bloka

11.2.2 Izbira objektov

Določene funkcije delujejo le, če objekt označimo. To storimo tako, da kliknemo nanj z levo miškino tipko. Izbrani objekt je označen z majhnimi črnimi kvadratki v ogliščih ikone. Za izbiro več objektov je potrebno držati pritisnjeno tipko Shift in z levim gumbom izbrati želene objekte, za izbiro vseh objektov v aktivnem oknu pa lahko uporabimo ukaz *Select All* v meniju *Edit*. Množico objektov lahko izberemo tudi tako, da s pritisnjenim levim miškinim gumbom označimo polje, kjer se nahajajo objekti, ki jih želimo izbrati.

11.2.3 Premikanje in kopiranje blokov

Bloke lahko premikamo ali kopiramo iz enega okna v drugo z vlečenjem (ang. dragging) s pomočjo miške ali z uporabo ukazov *Cut*, *Copy* in *Paste* v meniju *Edit*.

Z miško izberemo blok, ki ga želimo premakniti, in ga s pritisnjenim levim gumbom izvlečemo na izbrani položaj znotraj istega okna, nato pa sprostimo gumb. Rezultat akcije je premik bloka, vse povezave med bloki pa ostanejo nespremenjene. V primeru, ko je končni položaj v drugem oknu, se blok kopira v novo okno. Kopiranje znotraj istega okna lahko izvedemo s premikom, kjer namesto leve uporabimo desno miškino tipko ali pa pritisnemo tipko Ctrl. V tem primeru dobi ime bloka podaljšek v obliki zaporedne številke.

11.2.4 Brisanje blokov

Bloke brišemo tako, da najprej izberemo tiste, ki jih želimo brisati, in nato pritisnemo tipko Delete ali izberemo opcijo *Edit/Cut.*

11.2.5 Urejanje blokov

Spreminjanje velikosti – velikost lahko spreminjamo z miško tako, da vlečemo rob ali oglišče izbranega bloka v smeri spremembe dimenzije.

Urejanje imen – vsi bloki znotraj okna morajo imeti različna imena. Ime je v osnovnem položaju bloka niz znakov pod blokom in je lahko viden ali pa skrit, kar lahko določimo v stilski opciji *Format/Hide Name*. Ime lahko urejamo tako, da izberemo niz z imenom in ga nadomestimo z novim ali da postavimo kurzor na staro ime in napišemo novo.

Odpiranje blokov – vsak blok odpremo z dvojnim klikom. Če blok ni maskiran, se odpre okno s parametri bloka, ki jih lahko urejamo, drugače pa se pojavi okno, ki je določeno v parametrih maske.

Rotacija blokov – po definiciji potekajo signali skozi blok od leve proti desni, tako da je vhod v osnovni postavitvi vedno na levi, izhod pa na desni strani. Orientacijo pa lahko spreminimo z ukazoma *Format/Flip Block* in *Format/Rotate Block*. Ukaz *Rotate* z bližnjico Ctrl-R obrne blok za 90° v smeri urinega kazalca, ukaz *Flip* z bližnjico Ctrl-I pa za 180°.

11.2.6 Povezovanje blokov

Omenili smo že, kako povežemo dva bloka med seboj. Če na vhod bloka ne pripeljemo nobene povezave, je izhod vedno enak nič in v Matlabu se nam izpiše sporočilo, da smo pustili vhodno povezavo prazno. Število povezav iz izhoda posameznega bloka je neomejeno (razcepišče signala), medtem ko je na en vhod dovoljena le ena sama signalna linija. V primeru, da potrebujemo več signalov na

istem vhodu, uporabimo multipleksiranje. Seveda pa ne moremo povezovati vhodov in izhodov med seboj.

Brisanje povezav – povezavo brišemo tako, da jo izberemo z levo miškino tipko in nato pritisnemo tipko Delete ali pa izberemo *Edit/Cut*.

Segmentiranje in dodajanje povezav – če želimo določeno povezavo razstaviti na več lomljenih segmentov, potem postavimo kurzor na mesto, kjer želimo prelom, in pritisnemo tipko Shift. Hkrati pritisnemo levo miškino tipko in povlečemo kurzor na želeno mesto.

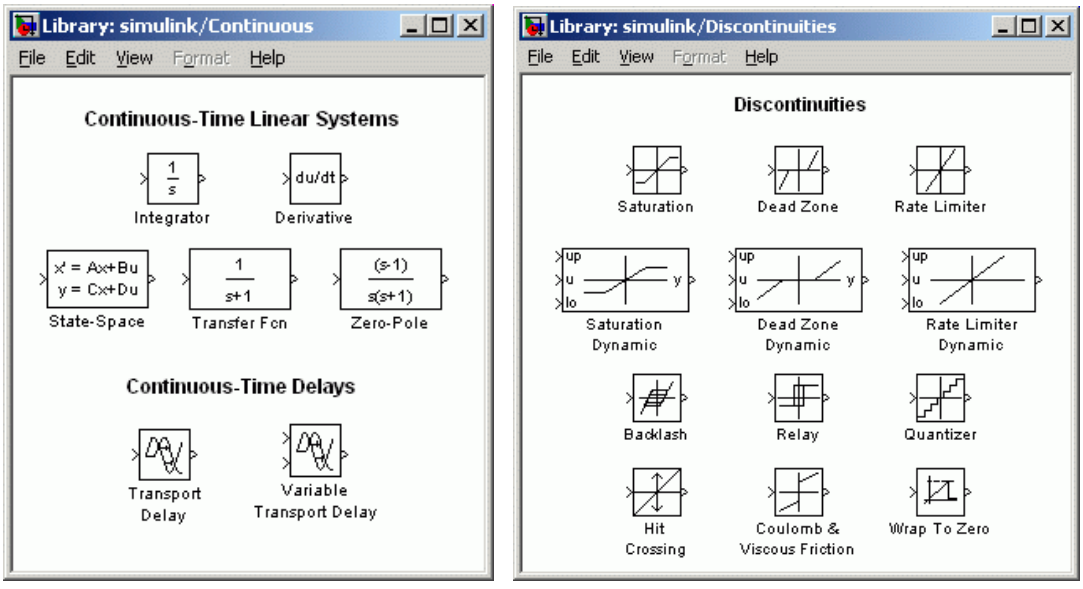
Novo povezavo lahko dodamo v katerikoli točki obstoječe povezave. Kurzor postavimo na mesto, kjer želimo odjemati signal, in s pritisnjeno desno miškino tipko izvlečemo novo povezavo. Vrednost signala na izhodu iz povezave je enaka vrednosti na odjemnem mestu. Enako lahko storimo s pritiskom levega gumba in hkratnim pritiskom tipke Ctrl.

11.2.7 Ostale funkcije v meniju *Format*

Bloke, povezave in glavno okno lahko poljubno pobarvamo (*Foreground*, *Background* in *Screen Color*). S podpostavkami postavke *Port/Signal Displays* odebelimo povezave, kjer je signal vektor, omogočimo izpisovanje dimenzij signalov in tipa signala itd. Z opcijo *Show Drop Shadow* pa lahko na primer blokom narišemo senco, tako da izgledajo tridimenzionalni.

11.3 Pregled najpomembnejših sklopov elementov

Pregledali bomo glavne podmape mape *Simulink* in pri vsaki podali kratek opis uporabe. V nadaljevanju pa bomo še razložili uporabo nekaterih najpomembnejših elementov oz. tistih, ki se jih največ uporablja, pa njihova uporaba ni neposredno razvidna iz imena elementa.

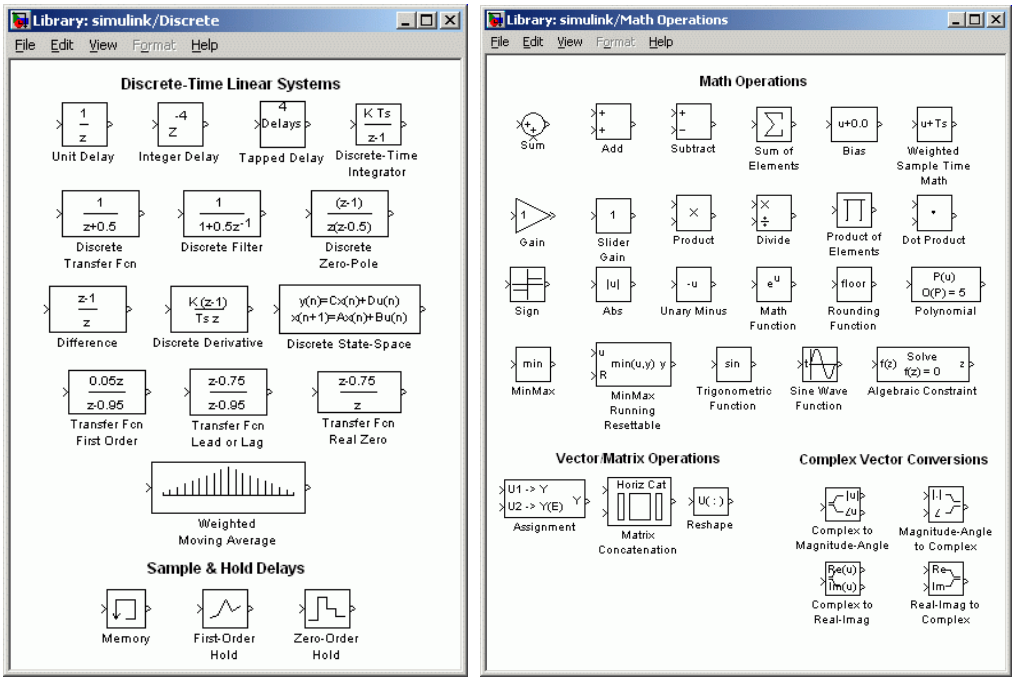


Slika 11.5 - Mapi **Contiunous** (levo) in **Discontinuities** (desno)

Začeli bomo z mapo **Continuous**, ki je predstavljena na levem delu slike 11.5. V njej so elementi, s katerimi modeliramo linearne časovno-nespremenljive procese.

Glavni element je *Integrator*, ki predstavlja numerični integrator funkcije odvoda, katero pripeljemo na njegov vhod. Proces, zapisan v prostoru stanj ali s prenosno funkcijo v polinomski in faktorizirani obliki, lahko izvedemo s pomočjo blokov *State-space*, *Transfer Fcn* in *Zero-Pole*. Z zadnjima blokoma pa v sistem dodamo časovno zakasnitev.

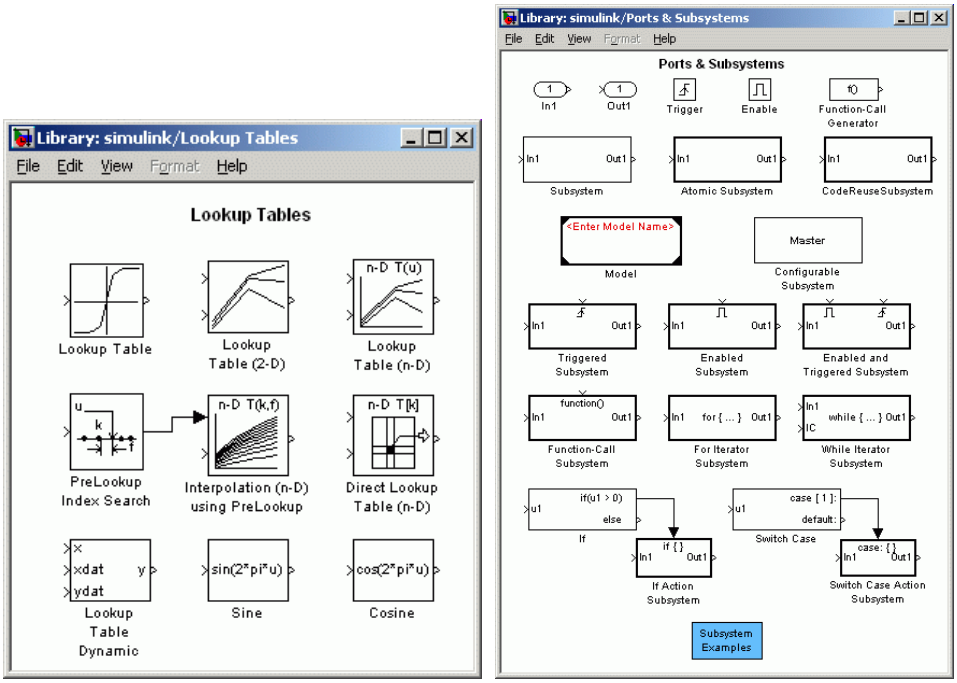
Bloke mape **Discontinuities** (slika 11.5 desno) uporabljamo za preoblikovanje signala (nasičenja, mrtve cone itd.) in za dodajanje nezveznih sprememb v signal (histereza, kvantizacija itd.).



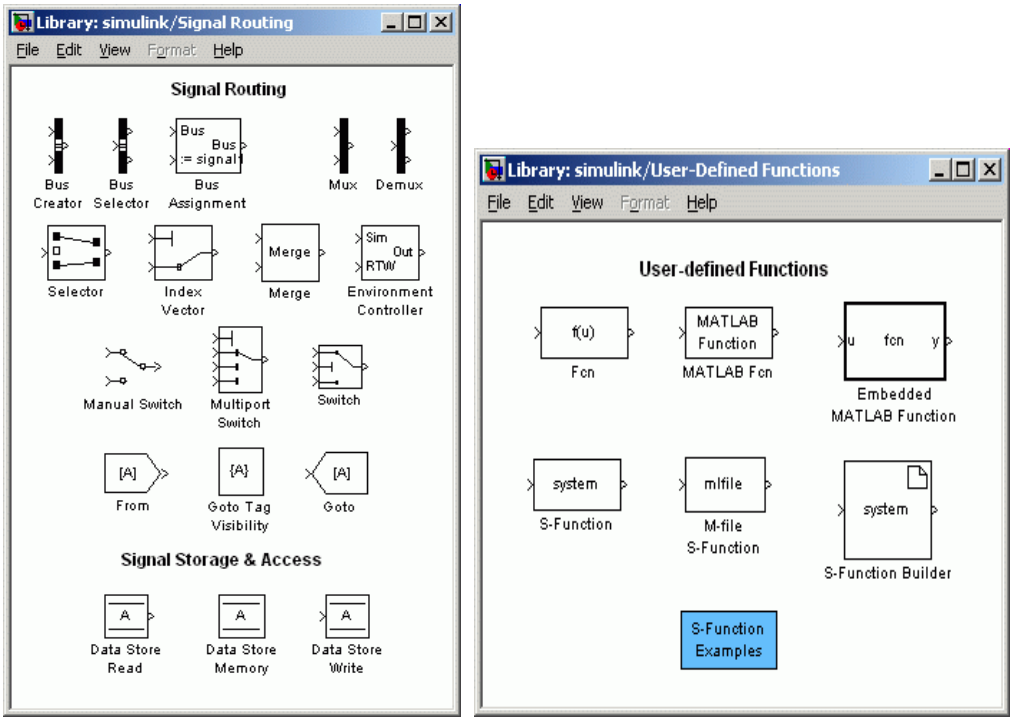
Slika 11.6 - Mapi **Discrete** (levo) in **Math operations** (desno)

Na sliki 11.6 levo je prikazana mapa **Discrete**, katere ne bomo podrobneje predstavljali, funkcionalno pa je precej podobna mapi **Continuous** – omogoča simulacijo diskretnih sistemov. Pač pa je pomembnejša mapa **Math Operations** (slika 11.6 desno), v kateri so zbrani vsi matematični operatorji in nekatere matematične funkcije, ki jih potrebujemo pri simulaciji dinamičnih sistemov. Največkrat uporabljamo bloke *Sum*, *Gain*, *Product* in *Trigonometric Function*.

V mapi **Lookup Tables** (slika 11.7 levo) so bloki, s katerimi ponazarjamo statične preslikave dveh ali več spremenljivk. Nelinearne vhodno-izhodne preslikave zapišemo z dvema vektorjema v bloku *Lookup Table*, katerega si bomo kasneje tudi pobliže pogledali. Na desnem delu slike 11.7 pa je predstavljena mapa **Ports & Subsystems**, v katerem je potrebno izpostaviti bloka *In* in *Out*. Uporabljamo jih za označevanje, kje v modelu se nahajajo vhodi in izhodi, in za tvorjenje podsistemov (podmodelov) v uporabniških blokih.



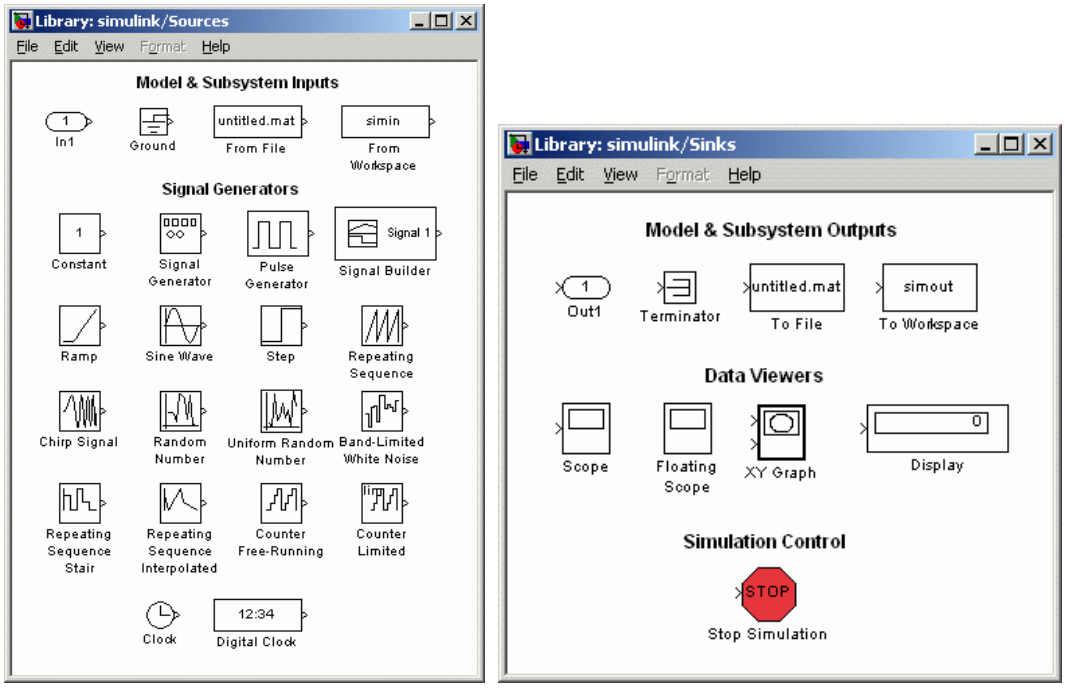
Slika 11.7 - Mapi **Lookup Tables** (levo) in **Ports & Subsystems** (desno)



Slika 11.8 - Mapi **Signal Routing** (levo) in **User-Defined Functions** (desno)

Na levem delu slike 11.8 je prikazana mapa **Signal Routing**, v kateri so predvsem bloki za usmerjanje in pretvorbo signalov. Najbolj uporabni elementi so *Mux* in *Demux*, ki se uporabljata za tvorjenje multipleksiranih vektorjev, ter *Switch* in *Manual Switch*, ki pripomoreta k izbiranju med večimi signali ročno ali na avtomatiziran način. Na desnem delu iste slike pa imamo mapo uporabniško

definiranih funkcij. Bloki se razlikujejo med seboj po okolju, ki podpira napisano funkcijo – blok *Fcn* vsebuje Matlabovo kodo, kot bi jo napisali v ukazni vrstici, *MATLAB Fcn* vsebuje ime m-datoteke, *S-Function* pa sistemske s-funkcije, katere funkcionalnost bomo še spoznali.



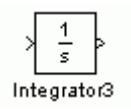
Slika 11.9 - Mapi **Sources** (levo) in **Sinks** (desno)

Zadnji mapi, ki ju bomo obravnavali, sta **Sources** in **Sinks** (slika 11.9). Prva vsebuje vire signalov v sistem, od katerih se največ uporablja *From File*, *From Workspace*, *Step* in *Ramp*, zanimivi pa so tudi generatorji šuma. V drugi pa najdemo ponore signalov, ki nam pomagaj pri shranjevanju podatkov (v mat-datoteko ali delovni pomnilnik) in prikazu (bloki **Data Viewers**).

12. PRIMERI UPORABE GLAVNIH ELEMENTOV

Posvetimo se glavnim elementom simulacijskih shem. Pri vsakem bosta podana razlaga parametrov in primer uporabe. Razlage najbolj enostavnih elementov, kot sta npr. *Gain* in *Sum*, zaradi intuitivne uporabe ne bomo podajali – bralec bo lahko s pomočjo primerov pridobil osnovne informacije o uporabi le-teh.

12.1 Integrator



Integrator je glavni element pri simulaciji sistemov, podanih v obliki diferencialnih enačb. Na vhod bloka pripeljemo signal odvoda funkcije, na izhodu pa dobimo integral odvoda oziroma časovni potek funkcije od začetnega do končnega časa, katera podamo v simulacijskih parametrih. Nastavljamo lahko tudi omejitve veličine na izhodu integratorja, začetno vrednost integrala in pa način proženja (interni in eksterni). Pri drugem načinu lahko vrednost integrala resetiramo z nekim zunanjim signalom, kar je še posebej uporabno pri tvorbi žagastih signalov. Poglejmo si primer uporabe.

Poskušajmo narediti simulacijsko shemo za numerično rešitev diferencialne enačbe

$$2\dddot{y} + 5\ddot{y} + 3\dot{y} + 4y = 2u\,, \tag{12.1}$$

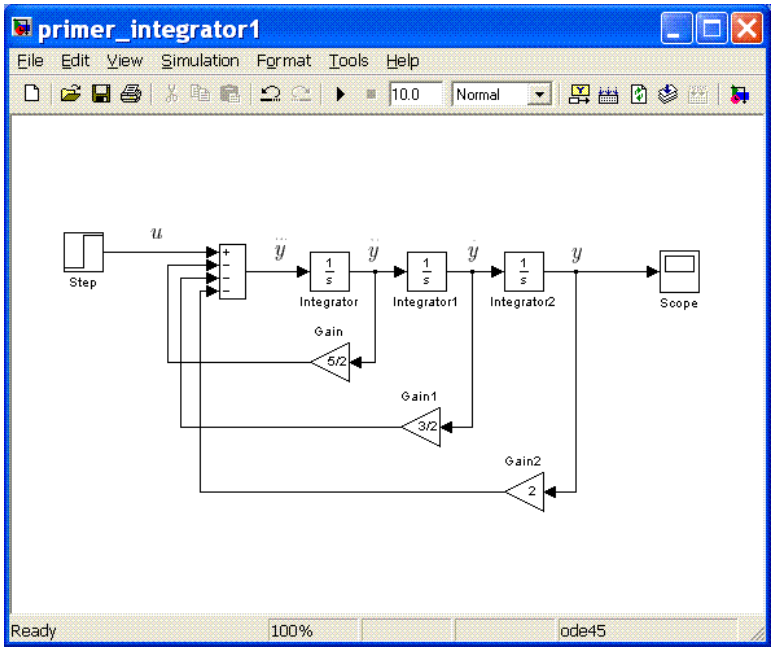
kjer je vzbujanje $u(t)$ stopnica z amplitudo 1 in časom nastopa T = 1 s. Najprej enačbo (12.1) preoblikujemo v obliko, kjer izrazimo najvišji odvod kot funkcijo vseh ostalih členov,

$$\dddot{y} = -\frac{5}{2}\ddot{y} - \frac{3}{2}\dot{y} - 2y + u\,, \tag{12.2}$$

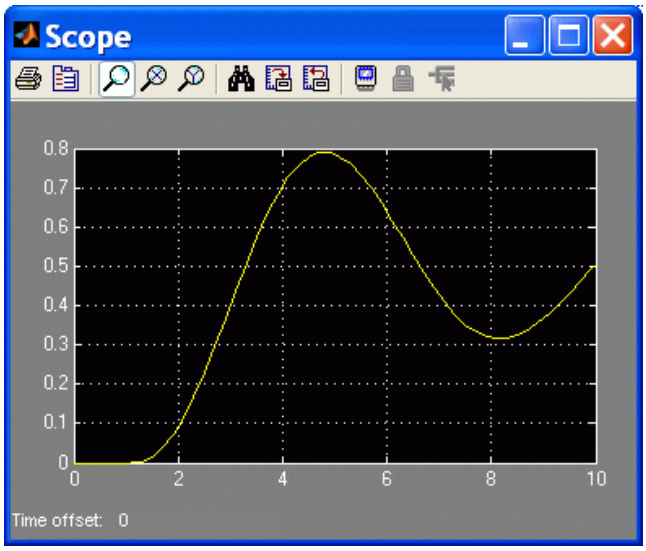
nato pa zgradimo shemo, v kateri na blok *Sum* pripeljemo vse člene desne strani enačbe (12.2) in dobljeno integriramo s tremi zaporedno vezanimi integratorji. Shemo, na kateri so označeni vsi signali, prikazuje slika 12.1.

Izhodi integratorjev so nižji odvodi, izhod zadnjega pa je iskana funkcija y. Izberemo tri bloke *Gain*, vhode povežemo s signali na izhodih integratorjev, izberemo pravilne vrednosti ojačenj in izhode pripeljemo na sumacijski blok, kateremu v lastnostih nastavimo *Icon Shape* na 'rectangular' in *List of Signs* na '+---'. Poženemo simulacijo in dobimo naslednji rezultat (slika 12.2).

Vidimo, da se funkcija v podanem časovnem okvirju ne ustali. Želeli bi videti, kakšen je nadaljni potek funkcije izhoda in, če konvergira k ustaljeni vrednosti, katera vrednost je to. Poleg tega pa bi radi bolj natančno izračunavanje numeričnega integrala, kar bi nam prineslo bolj gladko funkcijo, ter prikaz prvega odvoda funkcije na istem prikazovalniku.

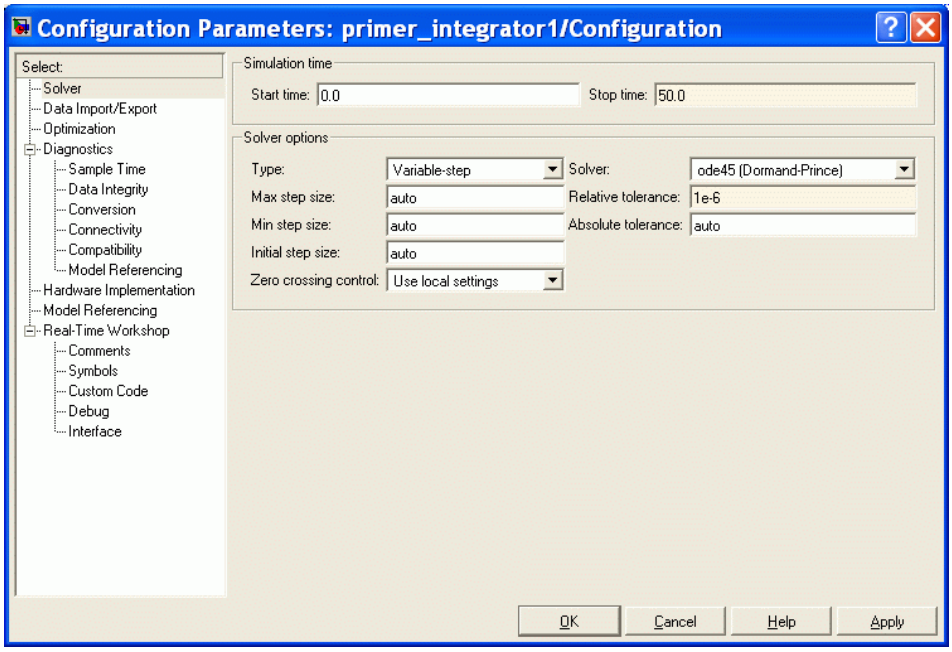


Slika 12.1 - Simulacijska shema za enačbo (12.2)

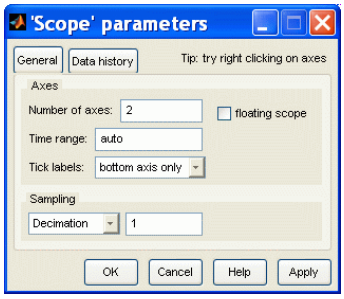


Slika 12.2 - Prvi rezultat simulacije

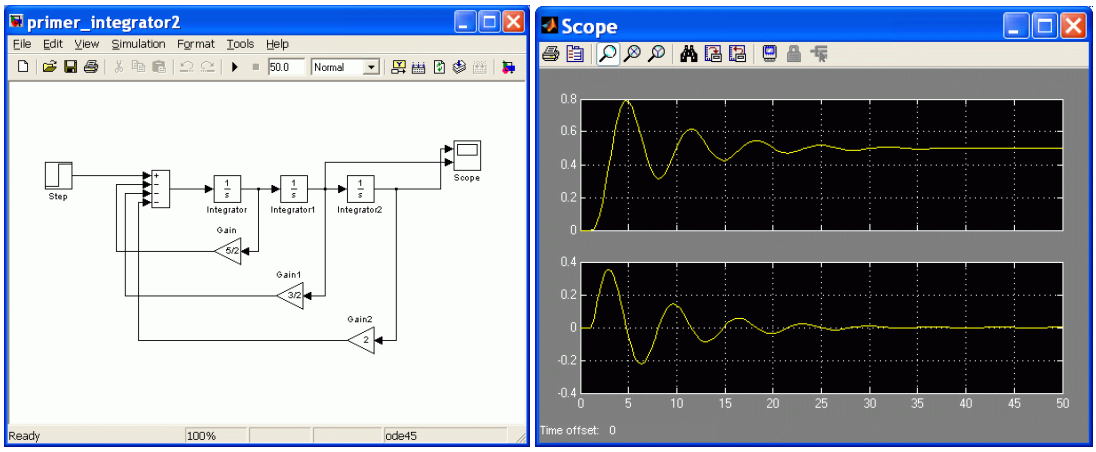
Najprej odprimo *Simulation*/*Configuration Parameters* in vnesimo vrednosti *Stop time* = 50 in *Relative tolerance* = 1e-6. Drugi parameter pomeni, da zahtevamo natančnost izračunavanja stanj 0.0001%. Vrednosti vseh ostalih parametrov Simulink prireja med samo simulacijo, lahko pa jih poda tudi uporabnik. *Max step size* in *Min step size* sta maksimalni in minimalni korak integracije, *Absolute tolerance* pa je največji dopustni absolutni pogrešek simulacijskega koraka. Slika 12.3 prikazuje okno *Configuration* z vnešenimi parametri.



Slika 12.3 - Okno *Configuration* za urejanje nastavitev simulacije



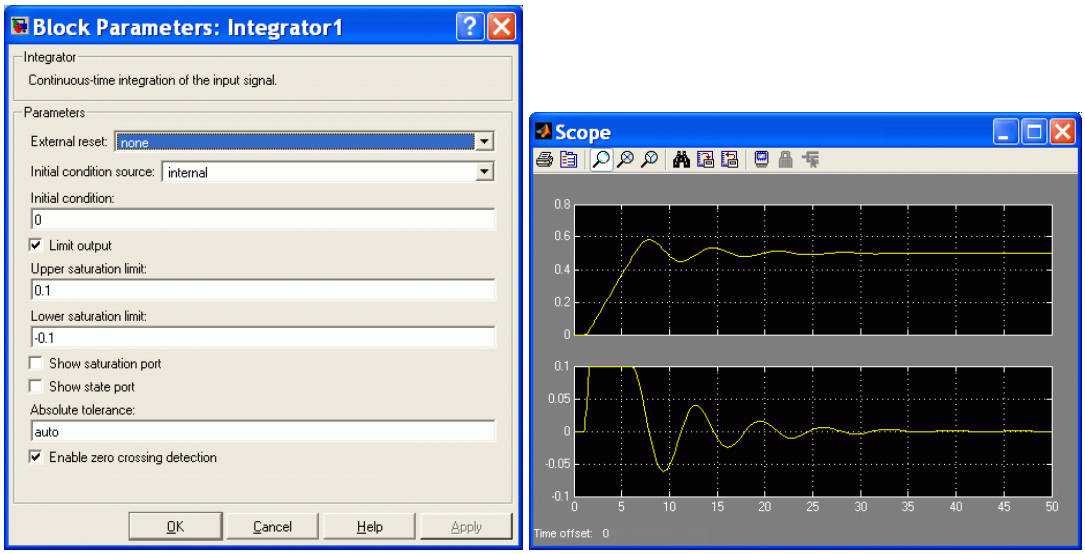
Če hočemo imeti na enem prikazovalniku več diagramov oz. osi, z dvoklikom odpremo okno *'Scope' parameters* in pri *Number of axes* vnesemo želeno število. V našem primeru bo to število 2, ostalih parametrov pa ne spreminjamo. Slika 12.4 prikazuje shemo s popravljenimi parametri in prikazom funkcije ter odvoda (levo) in rezultat simulacije (desno). Zgornji diagram predstavlja vrednost funkcije y, spodnji pa odvod $\dot{y}$.



Slika 12.4 - Shema s popravljenimi parametri (levo) in rezultat simulacije (desno)

Zdaj pa se posvetimo dodatnim parametrom bloka *Integrator*. Na sliki 12.5 levo je prikazano okno s parametri. Nastavljamo lahko zunanji vir resetiranja (*External reset*) in začetne vrednosti (*Initial condition source*). Slednje lahko podamo tudi

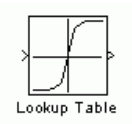
znotraj okna v rubriki *Initial condition*. Če izberemo *Limit output*, lahko podamo zgornjo in spodnjo mejo vrednosti stanja integratorja.



Slika 12.5 - Okno parametrov bloka *Integrator* (levo) in rezultat simulacije z novimi parametri

Preizkusimo, kaj se zgodi z rezultatom simulacije, če vrednost odvoda omejimo na ±0,1. V drugi blok integratorja ti dve meji vpišemo na ustrezni mesti, kot je to prikazano na sliki 12.5, in poženemo simulacijo. Rezultat prikazuje diagram na desnem delu iste slike. Vidimo, da je vrednost odvoda omejena pri 0,1, zato funkcija ne doseže prejšnje vrednosti, kar pa takoj vpliva tudi na odvod v negativnem delu, ki sploh ne doseže meje –0,1.

12.2 Look-up table



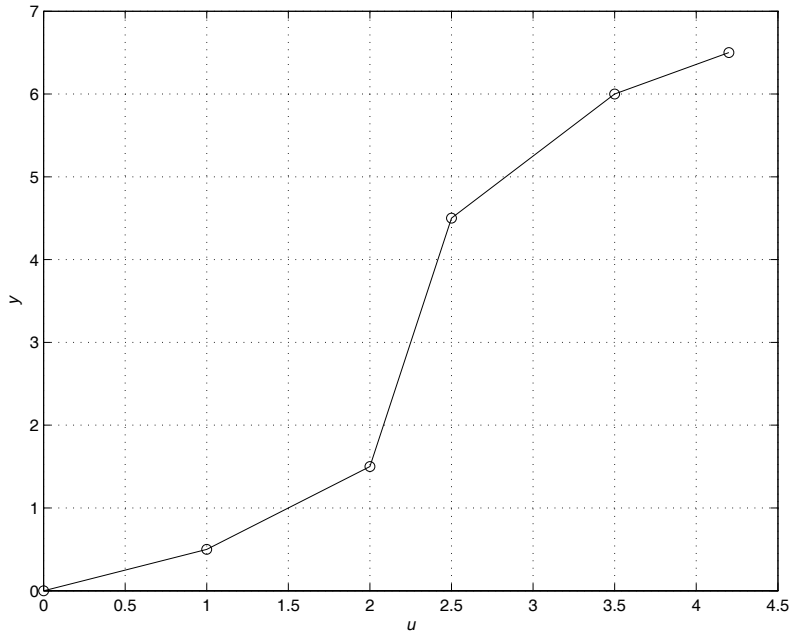
Statično nelinearno preslikavo izvedemo z blokom *Lookup Table*, ki se nahaja v mapi **Lookup Tables**. Podatke vnesemo v obliki točk dveh vektorjev, enega za absciso in enega za ordinato preslikave. Kot opozorilo naj povemo, da sta v osnovni izvedbi, kot je prikazana na sliki na levi strani, privzeti vrednosti abscise vektor [-5:5] in ordinate vektor tanh([-5:5]), kar pomeni enajst točk hiperboličnega tangensa.

Obravnavali bomo primer nelinearnega dinamičnega sistema, ki ga lahko modeliramo na naslednji način – dinamični del je linearen in ga lahko zapišemo s prenosno funkcijo

$$G(s) = \frac{1}{100s+1}, \tag{12.3}$$

kar pomeni, da je časovna konstanta T = 100 s neodvisna od delovne točke, nelinearnost pa je podana s statično preslikavo na sliki 12.6. Simulirajmo odziv sistema na stopničasto vzbujanje $u(t)$

$$u(t) = \begin{cases} 1, & 0 \le t \le 500 \\ 2, & 500 < t \le 1000 \\ 3, & 1000 < t \le 1500 \end{cases}. \tag{12.4}$$



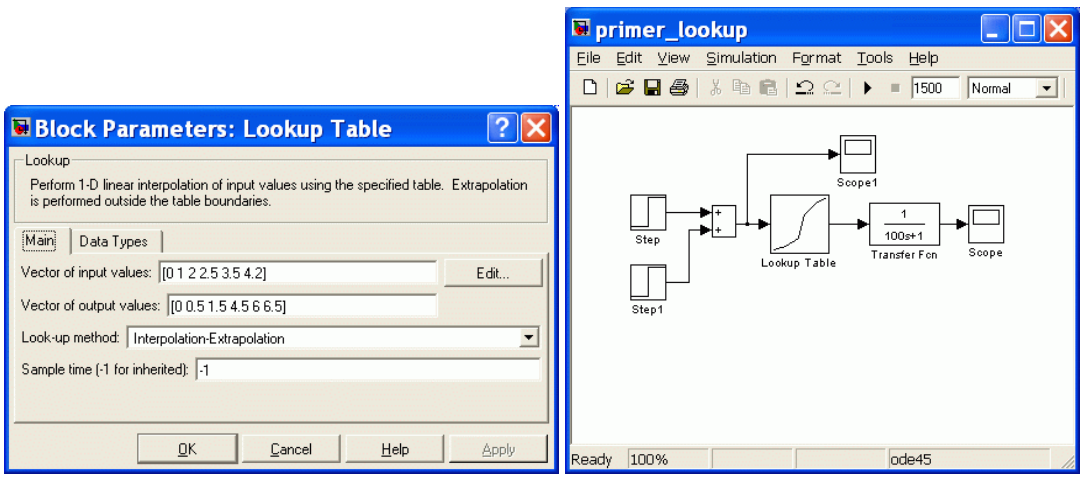
Slika 12.6 - Nelinearna statična preslikava

Iz diagrama lahko razberemo statično preslikavo, podano s tabelo 12.1.

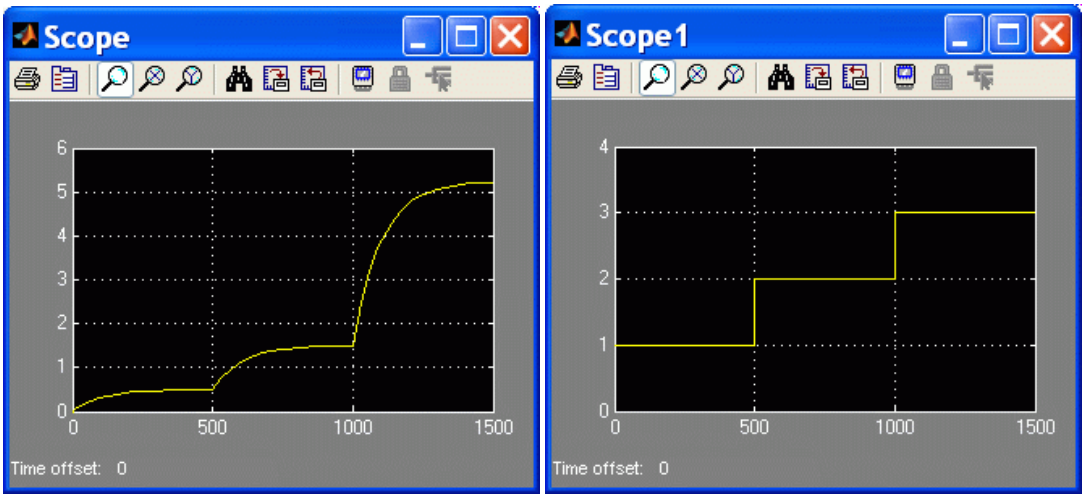
Tabela 12.1 - Točke nelinearne statične preslikave

vhod *u*	0	1	2	2.5	3.5	4.2
izhod *y*	0	0.5	1.5	4.5	6	6.5

Na ta način se ojačenje procesa skriva samo v razmerju izhodov in vhodov točk na statični preslikavi. Odpremo nov model, v njega vnesemo blok *Transfer Fcn* in v *Denominator* napišemo [100 1]. Nato dodamo blok *Lookup table* in vnesemo vektorja, kot je prikazano na sliki 12.7 levo. Vhodni signal izvedemo z vsoto dveh stopnic, pri katerih s prvo izvedemo spremembo iz 1 na 2 v času 500 s, pri drugi pa iz 0 na 1 v času 1000 s. Dodamo še dva bloka *Scope*, enega za vhodni in enega za izhodni signal, in povežemo, kot je prikazano na sliki 12.7 desno. Rezultat simulacije je prikazan na sliki 12.8.



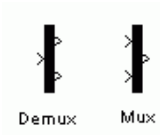
Slika 12.7 - Parametri bloka *Lookup Table* (levo) in shema modela (desno)



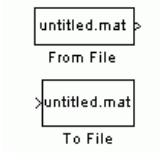
Slika 12.8 - Rezultata simulacije –izhodni signal (levo) in pripadajoči vhodni signal (desno)

Iz slike je razvidno, da je odziv sistema nelinearen, ker na enake spremembe vhodnega signala odgovori z različnimi spremembami izhoda.

12.3 Mux, Demux, From File in To File



Bloka *Mux* in *Demux* uporabljamo za multi- in demultipleksiranje signalov. Edini parameter, ki ga nastavljamo, je dimenzija vhodnega/izhodnega signala. Multipleksor se velikokrat uporablja v kombinaciji s funkcijskimi bloki, ki zahtevajo več spremenljivk naenkrat, ali pa za prikaz spremenljivk na istem diagramu.



Bloka *From File* in *To File* uporabljamo, kadar želimo delati s serijami podatkov, shranjenih v mat-datotekah. Pri *From File* v bloku podamo ime datoteke, v kateri se nahaja spremenljivka s podatki. Pri *To File* podamo poleg imena izhodne datoteke tudi ime spremenljivke, v katero se bo signal shranil. Privzeti imeni sta *untitled.mat* in *ans*. Standarden zapis je vrstična matrika, kjer prva vrstica predstavlja vrednosti neodvisne spremenljivke, ostale vrstice pa serije vrednosti odvisne spremenljivke. Pri *To File* je prva vrstica vedno simulacijski čas (vstavi jo Simulink), ostale vrstice pa pripeljemo na vhod bloka.Če hočemo shraniti več signalov, jih združimo z blokom *Mux* in pripeljemo na vhod bloka *To File*.

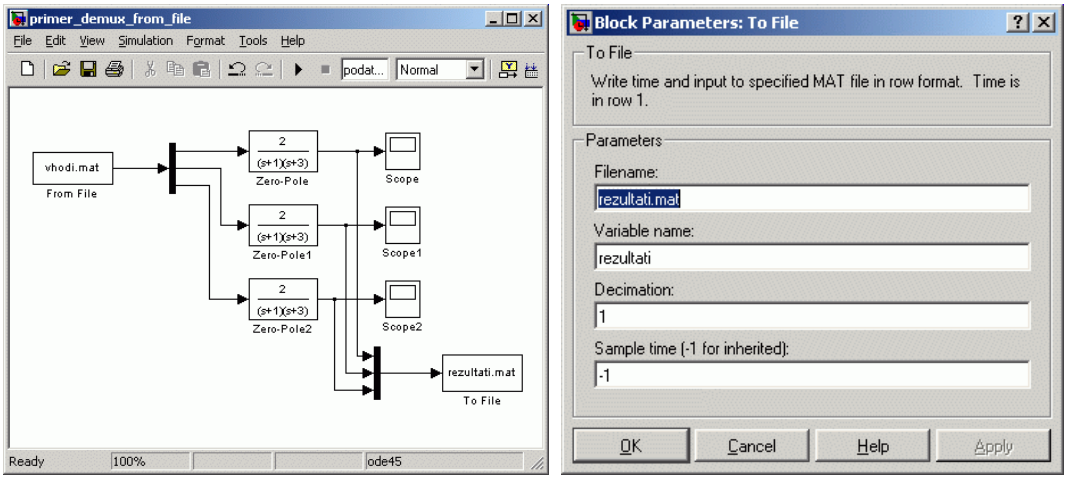
Poglejmo si primer, kjer proučujemo odziv sistema

$$G(s) = \frac{2}{(s+2)(s+3)} \tag{12.5}$$

na sinusno nihanje naraščajoče frekvence $\sin(k\omega t)$, $k \in \{1,3,5\}$, $\omega t \in [0, 4\pi]$. Vhodne signale pripravimo v Matlabu in shranimo v datoteko *vhod.mat*, nato izvedimo simulacijo z vsemi vhodnimi signali in rezultate shranimo v datoteko *rezultati.mat*. Priprava signalov je naslednja:

```
x = [0:pi/32:4*pi];
y = [];
for i = 1:3
    y(i,:) = sin((2*i-1)*x);
end
podatki = [x;y];
save vhodi podatki
```

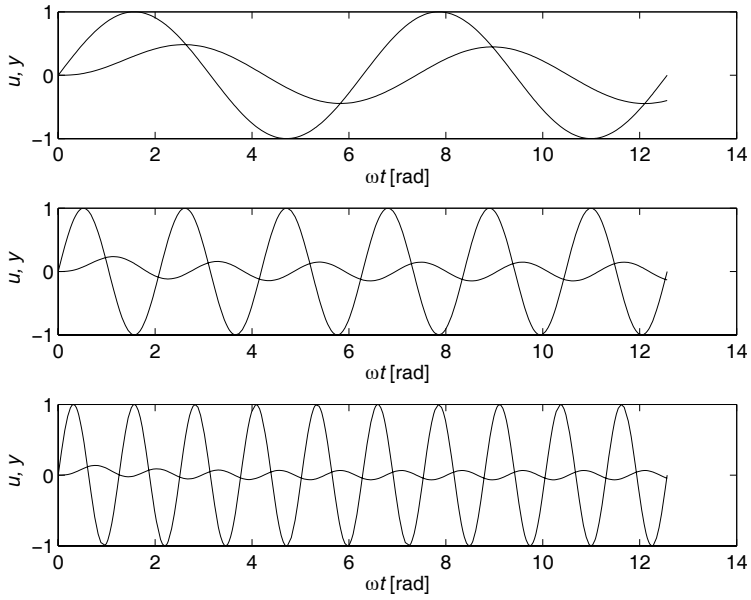
V Simulinku sestavimo shemo, kot jo prikazuje levi del slike 12.9, in v blok *To File* vpišemo ime izhodne datoteke *rezultati.mat* in ime spremenljivke *rezultati*, kot je prikazano na desnem delu iste slike. Nato poženemo simulacijo.



Slika 12.9 - Shema za simulacijo odzivov sistema na sinusna vzbujanja

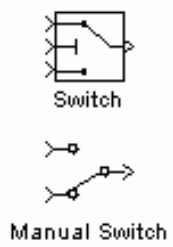
Rezultati so zdaj shranjeni v datoteki. Naložimo jih v delovni prostor in prikažimo na sliki s tremi diagrami (slika 12.10).

```
load rezultati
figure
for i=1:3
    subplot(3,1,i)
    set(gca,'FontSize',13)
    % Signal vzbujanja lahko naredimo iz prve vrstice
    plot(rezultati(1,:),sin((2*i-1)*rezultati(1,:)))
    hold on
    % Prva vrstica spr. rezultati je neodvisna spremenljivka
    plot(rezultati(1,:),rezultati(i+1,:))
    hold off
end
```

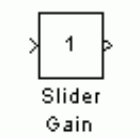


Slika 12.10 - Rezultati simulacije

12.4 Switch, Manual Switch in Slider Gain



Oba bloka uporabljamo za preklapljanje med dvema signaloma, ki ju pripeljemo na vhode blokov. Razlika je le v tem, da pri bloku *Manual Switch* preklapljamo ročno s klikom z miško, pri avtomatskem pa odloča logika na podlagi signala na srednjem vhodu. Na voljo imamo tri možnosti za odločanje, kdaj naj blok preklopi iz prevajanja prvega na prevajanje drugega signala – če je odločitveni signal strogo večji od mejne vrednosti, katero vpišemo v rubriko *Treshold* ($u(2) > Treshold$), če je večji ali enak tej vrednosti ($u(2) \geq Treshold$) in če je različen od 0 ($u(2) \neq 0$).

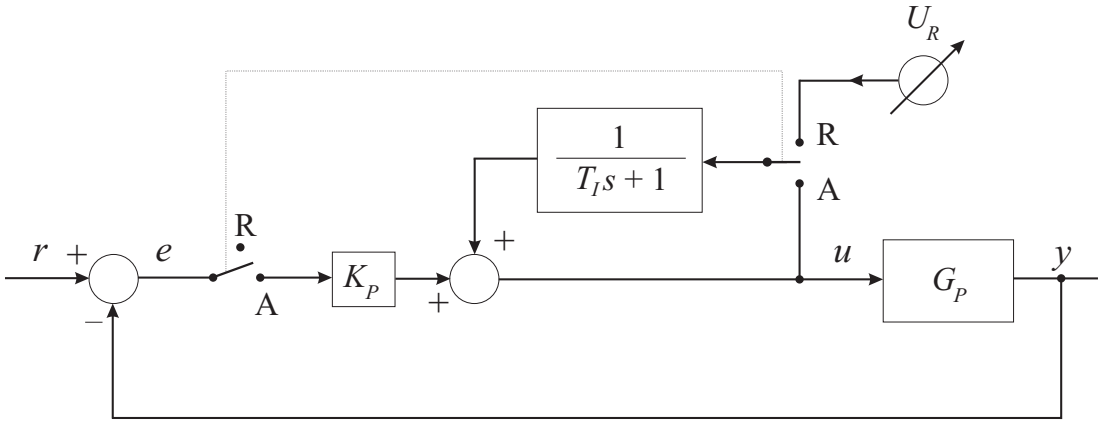


Slider Gain je blok, ki mu lahko med simulacijo nastavljamo vrednost ojačenja in je zato primeren za izvedbo stopničastega signala spremenljive amplitude. Na vhod povežemo konstanto 1, odpremo lastnosti bloka, nastavimo začetno in končno vrednost ojačenja ter nato z miško premikamo drsnik, da dobimo trenutne vrednosti izhoda bloka.

Primer uporabe opisanih elementov je ena od izvedb brezudarnega preklopa **ročno-avomatsko** v industrijskem regulatorju. Slika 12.11 prikazuje shemo take izvedbe. Denimo, da bi želeli regulirati proces

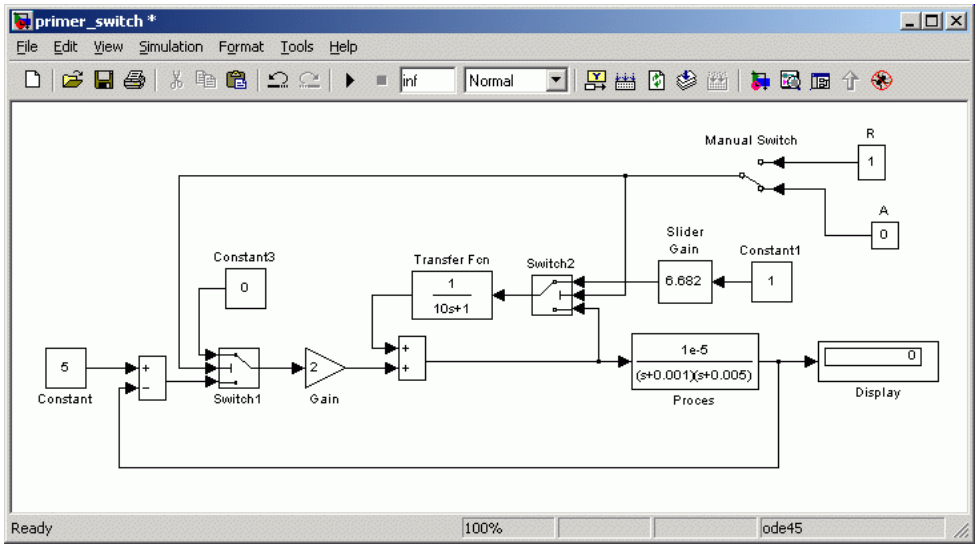
$$G_P(s) = \frac{10^{-5}}{(s+0.002)(s+0.005)} \tag{12.6}$$

s PI-regulatorjem, ki ima ojačenje $K_P = 2$ in časovno konstanto I-člena $T_I = 1000$.



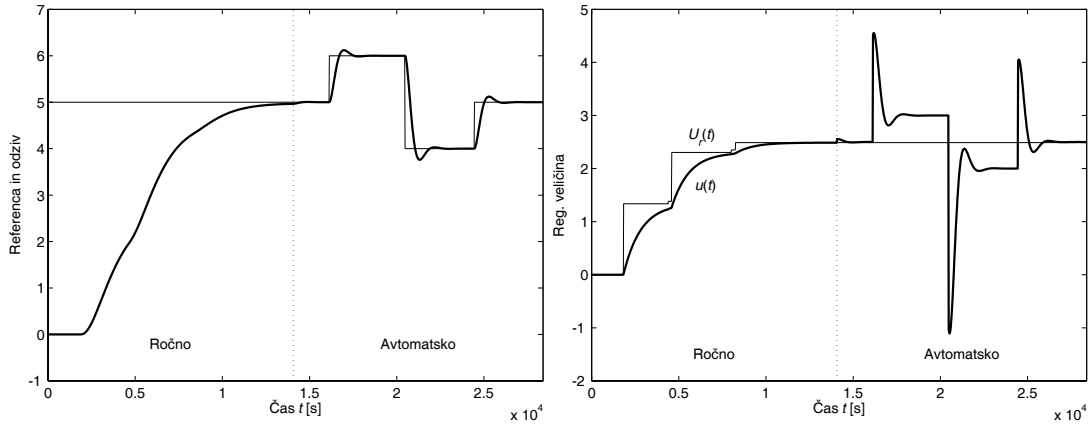
Slika 12.11 - Shema izvedbe regulatorjem z brezudarnim preklopom ročno-avtomatsko

Ker na sliki 12.11 vidimo, da sta preklopnika povezana med seboj, preklopa ne moremo izvesti z dvema ročnima preklopnikoma, ampak uporabimo enega ročnega in dva avtomatska, ki preklapljata glede na izhod ročnega preklopnika. Tako dosežemo hkraten preklop na P- in I-členu regulatorja. Slika 12.12 kaže shemo izvedbe v Simulinku.



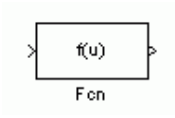
Slika 12.12 - Shema izvedbe regulatorja z brezudarnim preklopom v Simulinku

Najprej z nastavljanjem vrednosti ojačenja bloka *Slider Gain* pripeljemo proces v okolico delovne točke $r = 5$, nato pa preklopimo *Manual Switch* v položaj A. Potem izvedemo pozitivno, dvakratno negativno in ponovno pozitivno spremembo reference amplitude 1. Slika 12.13 prikazuje rezultate poskusa. Na desnem diagramu vidimo potek regulirne veličine, ki jo do preklopa upravljamo ročno s signalom U_r (regulirna veličina u spremembam sledi po sistemu prvega reda časovno konstanto 1000 s), po preklopu pa vključimo še P-člen in spremembe so vidno hitrejše. Na levem delu slike pa se pokaže, da v trenutku preklopa ni posledic udara regulirne veličine.

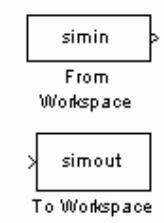


Slika 12.13 - Odziv in regulirna veličina pri simulaciji sistema vodenja z brezudarnim preklopom

12.5 Fcn, From Workspace in To Workspace



Funkcijski blok iz mape **User-defined Functions** uporabimo, kadar hočemo neposreden zapis nelinearne funkcije, ki bi sicer ob izvedbi z bloki shemo naredila nepregledno. Na vhod bloka pripeljemo vse spremenljivke, ki nastopajo v enačbi, v obliki vektorja (*Mux*), znotraj bloka pa napišemo enačbo, kot bi jo v Matlabovo ukazno vrstico, na spremenljivke pa se referenciramo s komponentami vhodnega vektorja $u(i)$. Opozoriti velja, da funkcijski blok na izhodu vedno vrača **skalarno** vrednost.



Bloka *From* in *To Workspace* se uporabljata na podoben način kot bloka *From* in *To File*. Edina razlika je, da tu kot izvor ali ponor signala podamo spremenljivko v Matlabovem delovnem pomnilniku. Pri *From Workspace* podamo v rubriko *Data* vhodne podatke v obliki strukture ali vektorjev. Pri vektorski obliki podajamo vektorja neodvisne (navadno čas) in odvisne spremenljivke v strukturi [čas,podatki]. Pri *To Workspace* podamo ime spremenljivke in njen tip, ki je lahko *Structure* (struktura), *Structure with time* (struktura s časom) in *Array* (vektor ali matrika). Slednja je najbolj uporabna za nadaljno obdelavo.

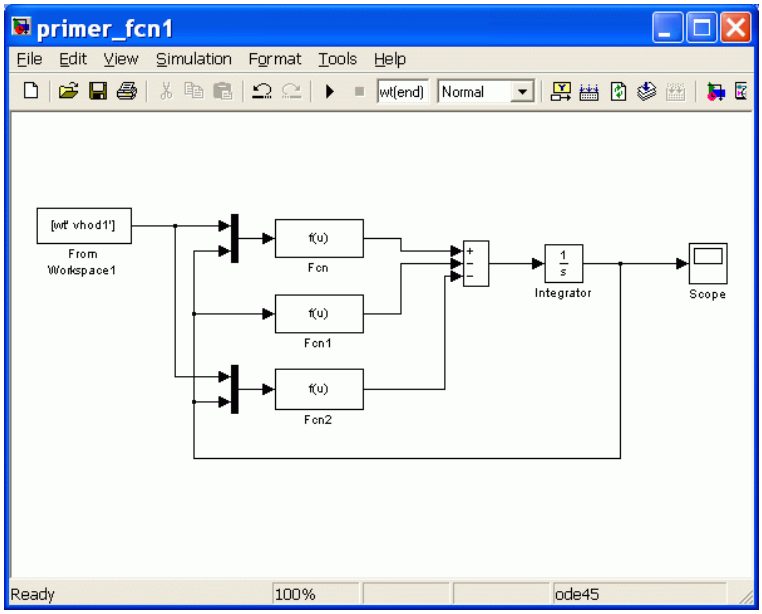
Za primer si poglejmo, kako bi simulirali odziv nelinearnega sistema

$$\frac{dx(t)}{dt} = C_1 V_1 \left(X_1 - x(t)\right)^2 \cdot u(t) - C_2 V_2^2 \sqrt{X_2 S - x(t)} + C_1 V_3 \frac{1}{U_0 - u(t)} \cdot x(t) \quad (12.7)$$

na sinusno vzbujanje $u(t) = 1 + \sin \omega t,\ 0 \le \omega t \le 8\pi$. Vrednosti konstant so C_1 = 1/2, C_2 = 1/4, V_1 = 1/3, $V_2 = V_3$ = 1/6, $X_1 = X_2$ = 5, S = 4 in U_0 = 5. Najprej pripravimo integrator in sumator treh elementov. Vsakega od členov na desni strani enačbe (12.7) bomo zapisali v svoj funkcijski blok. Vidimo, da bomo potrebovali pri prvem in tretjem členu dve spremenljivki, zato dodamo pred bloke dva multipleksorja z dvema vhodoma. Za vnos spremenljivk imamo dve možnosti – lahko pišemo njihove vrednosti neposredno v enačbo v funkcijskem bloku, lahko pa jih definiramo v Matlabovem pomnilniku in v enačbo zapišemo njihova imena, kar bo veliko bolj pregledno, še posebej, če imamo veliko spremenljivk. Poglejmo si drugi primer. V m-datoteko zapišemo vrednosti konstant in naredimo vhodni vektor.

```
% Konstante
C1 = 1/2, C2 = 1/4;
V1 = 1/3, V2 = 1/6, V3 = 1/6;
X1 = 5, X2 = 5;
S = 4, U0 = 5;

% Vzbujanje
wt = [0:pi/64:8*pi];
vhod1 = 1 + sin(wt);
```

Če zapišemo konstante v datoteko, **ne pozabimo pognati datoteke v Matlabovi ukazni vrstici!** Nato povežemo shemo, kot to prikazuje slika 12.14.



Slika 12.14 - Shema vezave z uporabo blokov *From Workspace* in *Fcn*

V blok *From Workspace* vpišemo imeni vhodne in izhodne spremenljivke (levi del slike 12.15)., Ker sta *wt* in *vhod1* vrstična vektorja, potrebujemo pa par stolpnih vektorjev, uporabimo znak za transponiranje vektorjev, tako da je oblika [wt' vhod1']. Na desnem delu iste slike vidimo, kako zapišemo funkcijo v blok *Fcn*. Paziti moramo, da se oznake elementov vhodnega vektorja ujemajo s signali na multipleksorju. Funkcije, ki smo jih zapisali v funkcijske bloke, so

```
C1*V1*(X1-u(2))^2*u(1)
```

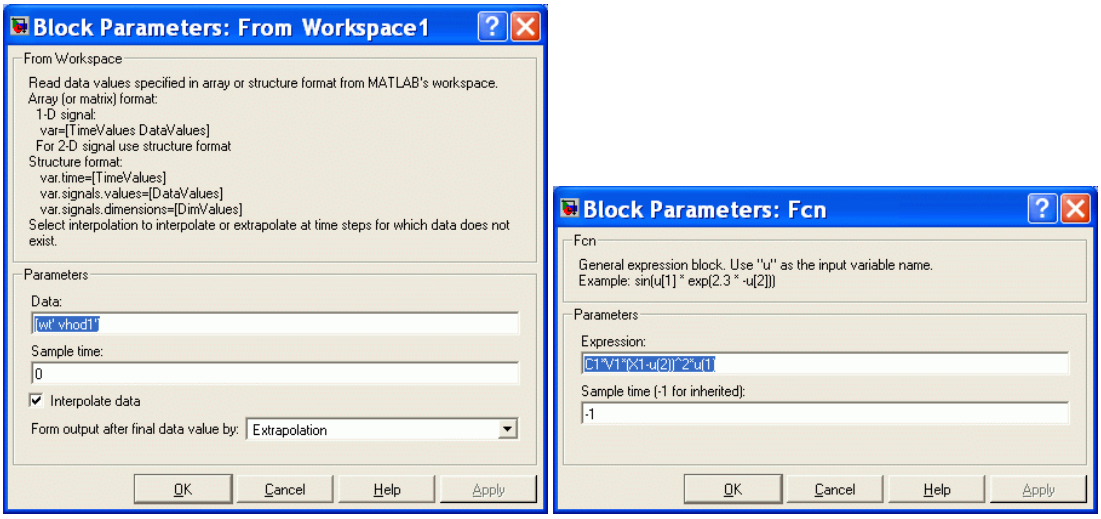
v prvem,

```
C2*(V2)^2*sqrt(abs(X2*S-u(1)))*sgn(X2*S-u(1))
```

v drugem in

```
C1*V3/(U0-u(1))*u(2)
```

v tretjem bloku.

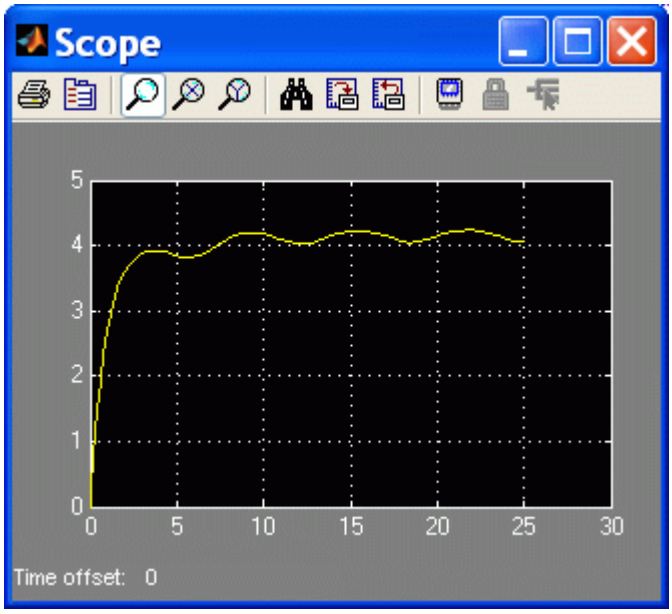


Slika 12.15 - Parametri blokov *From Workspace* in *Fcn*

Opozorimo še na sintakso drugega bloka. Ker potrebujemo korensko karakteristiko, vrednost pod korenom pa ne sme biti negativna, to najelegantneje rešimo na naslednji način: vzamemo absolutno vrednost izraza pod korenom in pomnožimo s signumom istega izraza, ali drugače,

$$\sqrt{|X_2S-x(t)|}\,\operatorname{sgn}\left(X_2S-x(t)\right)=\begin{cases}\sqrt{X_2S-x(t)}, & \text{če } X_2S-x(t)\geq 0\\ -\sqrt{-\left(X_2S-x(t)\right)}, & \text{če } X_2S-x(t)<0\end{cases}. \quad (12.8)$$

Čas simulacije nastavimo na končno vrednost vhodnega vektorja *wt*, kar napišemo *wt*(end). Končno lahko poženemo simulacijo in dobimo rezultat, prikazan na sliki 12.16.



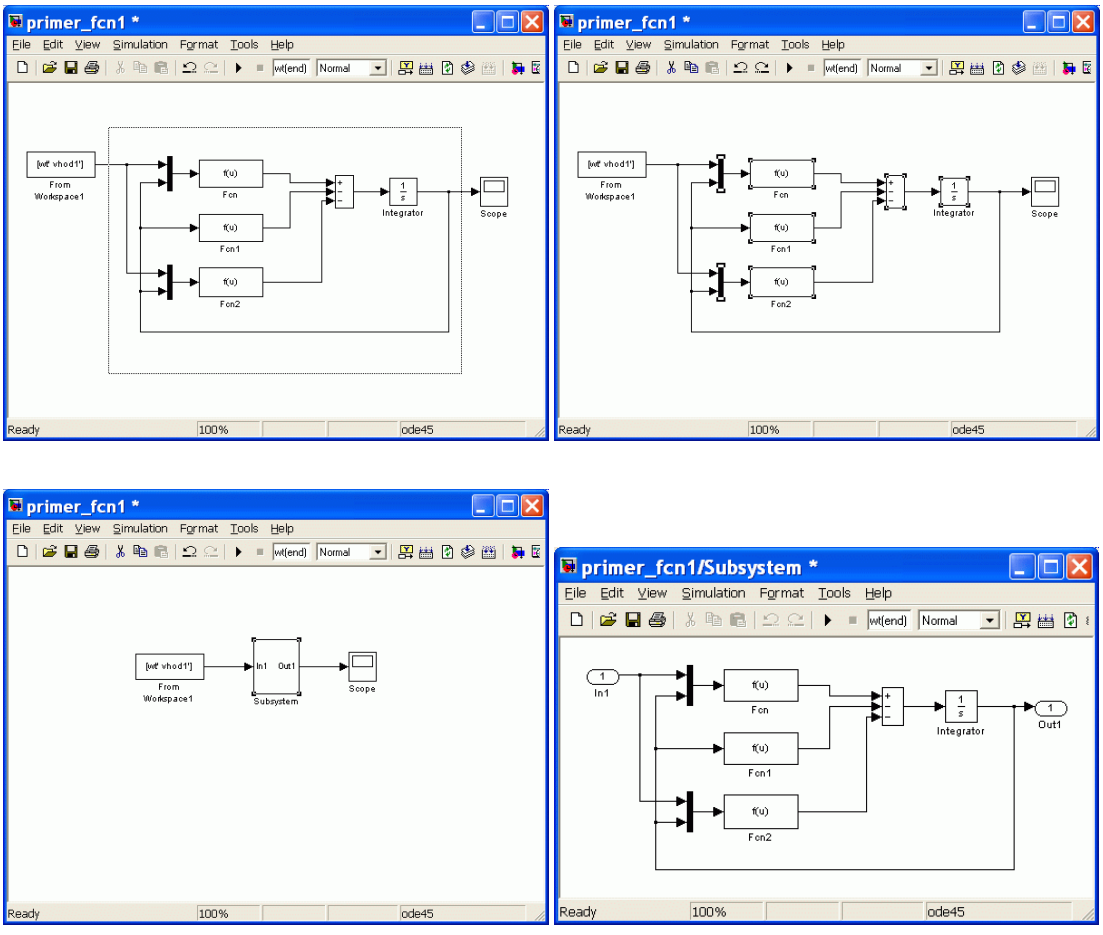
Slika 12.16 - Rezultat simulacije

12.6 Kako naredim podsistem in ga maskiram?

Obravnavajmo isti primer kot v prejšnjem podpoglavju, vendar pa bi zdaj v isti simulacijski shemi radi dobili tudi odziva na sinusni nihanji s trojno in petkratno frekvenco. Matlabovo datoteko moramo torej preurediti tako, da dobimo tri vhodne vektorje.

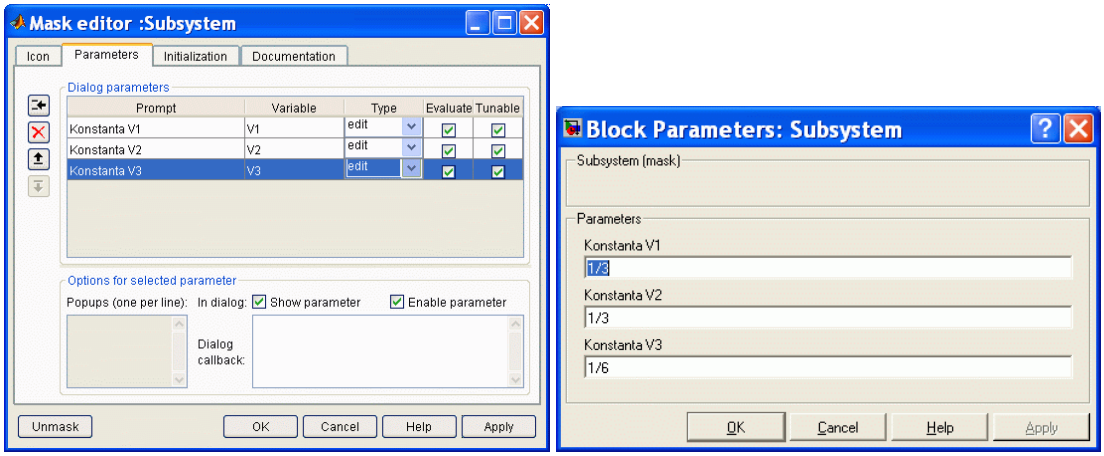
```
y = []; wt = [0:pi/64:8*pi];
for i = 1:3
    y(i,:) = sin((2*i-1)*wt) + 1;
end
vhod1 = y(1,:);
vhod2 = y(2,:);
vhod3 = y(3,:);
```

Ker bi morali imeti tri enake sklope blokov nelinearnega procesa, bi shema izgledala nepregledna. Zato proces maskiramo v podsistem, blok z enim vhodom in enim izhodom. To naredimo tako, da s pritisnjeno levo miškino tipko označimo pravokotnik, v katerem se nahajajo vsi bloki in povezave, ki bi jih radi imeli v podsistemu. Ko imamo želeno področje označimo, izberemo *Edit/Create Subsystem*, pritisnemo bližnjico Ctrl-G ali z desnim klikom na enega od izbranih blokov v kontekstnem meniju izberemo enako opcijo. Vsi koraki so prikazani na sliki 12.17.



Slika 12.17 - Koraki izdelave podsistema

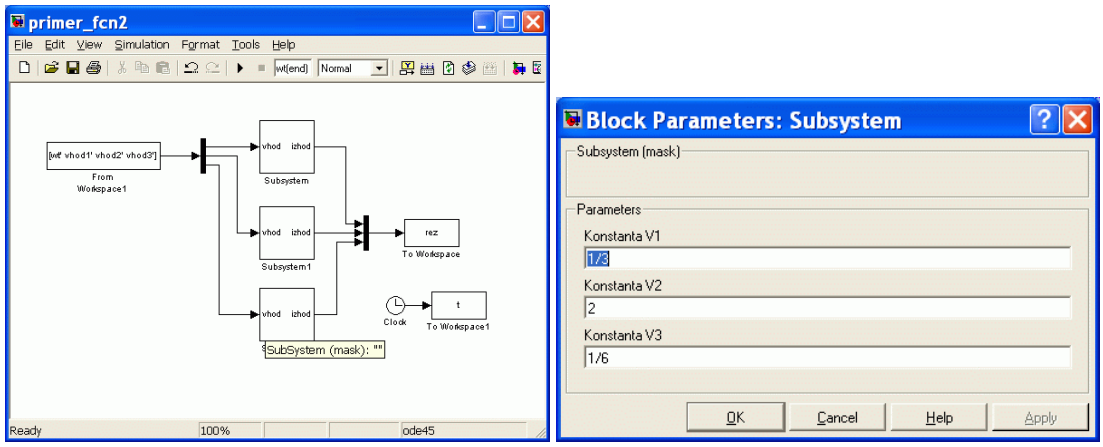
S poimenovanjem blokov *In* in *Out* dobimo ti imeni tudi na vhodu in izhodu bloka podsistema. Preimenujmo ju v *vhod* in *izhod*. Nato podsistem maskirajmo, tako da bomo lahko urejali izbrane parametre znotraj dialoga, ki ga sami oblikujemo. Blok označimo in pritisnemo Ctrl-M ali izberemo *Edit/Mask Subsystem*. Odpre se nam okno, ki ga prikazuje levi del slike 12.18. V meniju *Icon* lahko oblikujemo izgled ikone, ki predstavlja podsistem. V meniju *Parameters* pa s pomočjo ikone odpremo vrstico, v kateri označimo ime parametra, kot bo prikazano v oknu parametrov podsistema, in ga povežemo s spremenljivko modela, katere vrednost želimo urejati. V tem primeru hočemo spreminjati vrednosti konstant V_1, V_2 in V_3.



Slika 12.18 - Okno za maskiranje sistema

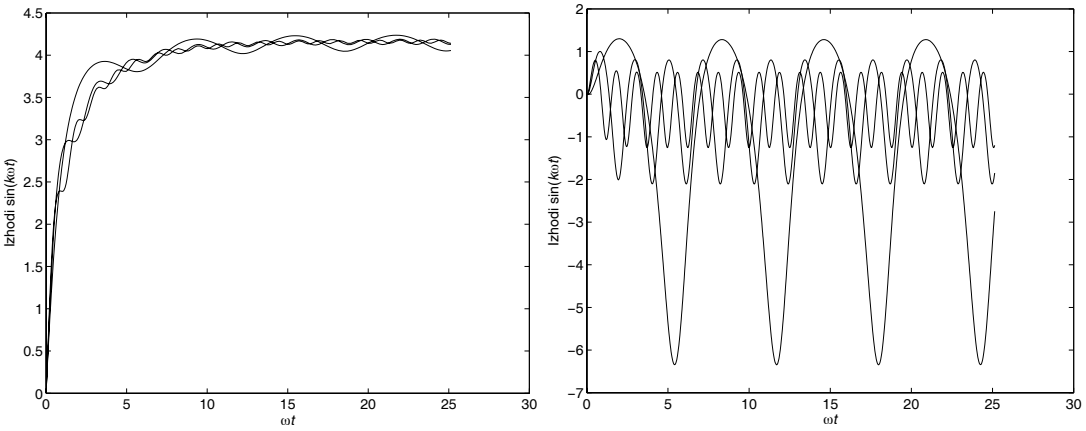
Ko smo blok maskirali, z dvoklikom odpremo novo okno s parametri, katere smo določili v meniju *Parameters* (desni del slike 12.18). Če želimo urejati parametre maske, moramo izbrati *Edit/Edit Mask*, če pa želimo dostopati do podsistema, moramo izbrati *Edit/Look Under Mask*.

Zdaj pa dokončajmo zadani eksperiment. V *From Workspace* vpišemo vse štiri spremenljivke (*wt*, *vhod1*, *vhod2* in *vhod3*), z demultipleksorjem dobimo tri signale, podsistem dvakrat prekopiramo, povežemo in z multipleksorjem izhode povežemo v *To Workspace*, v katerem izberemo ime *rez* in obliko *Array*. Poleg tega v mapi **Sources** poiščemo blok za simulacijski čas in ga preusmerimo v izhod *t*. Shema je prikazana na levem delu slike 12.19.



Slika 12.19 - Shema (levo) in novi parametri (desno)

Izvedemo simulacijo in narišemo diagram z ukazom `plot(t,rez)`, ki nariše vse tri odzive na isti diagram. Nato v vseh podsistemih spremenimo vrednost parametra V_2 na 2, kot to kaže desni del slike 12.19. Ponovno izvedemo simulacijo in narišemo diagram. Oba sta prikazana na sliki 12.20. S pomočjo maskiranih blokov lahko na enostaven način nastavljamo parametre podsistema in preučujemo njihove vplive na odziv sistema (še posebej uporabno se to izkaže pri nastavljanju parametrov regulatorja).

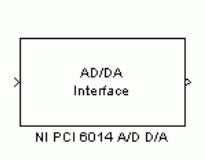


Slika 12.20 - Rezultati simulacije z osnovnimi parametri (levo) in po spremembi V_2 (desno)

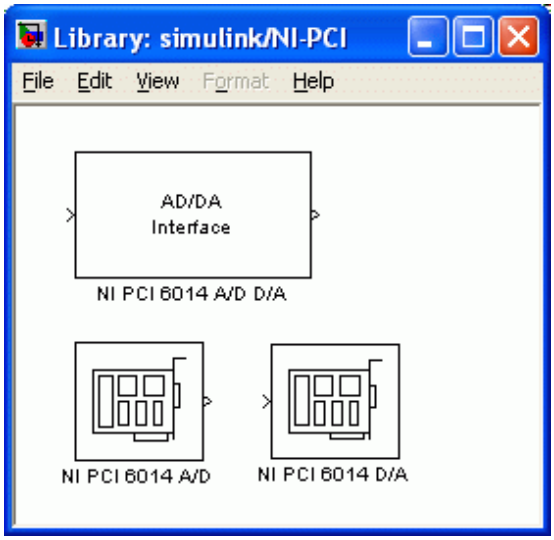
12.7 Bloki za komunikacijo s procesnim vmesnikom NI–PCI 6014

Realne naprave na računalnik priključimo preko procesnega vmesnika, ki analogne signale pretvori v digitalno obliko, ki je primerna za nadaljno obdelavo. Ogledali si bomo, kako uporabimo povezavo s pomočjo procesnega vmesnika National Instruments NI–PCI 6014 (http://www.ni.com). Vmesnik omogoča analogno–digitalno pretvorbo (A/D), digitalno–analogno pretvorbo (D/A), zajem digitalnih vhodov in izhodov, realizacijo digitalnih števcev in sinhronizacijo z realnim časom. Na njegove vhode lahko priključimo do 8 analognih vhodnih signalov, na izhodih pa vmesnik generira 2 analogna časovno–zvezna signala. Delovno območje vhodnih in izhodnih signalov je ±10 V.

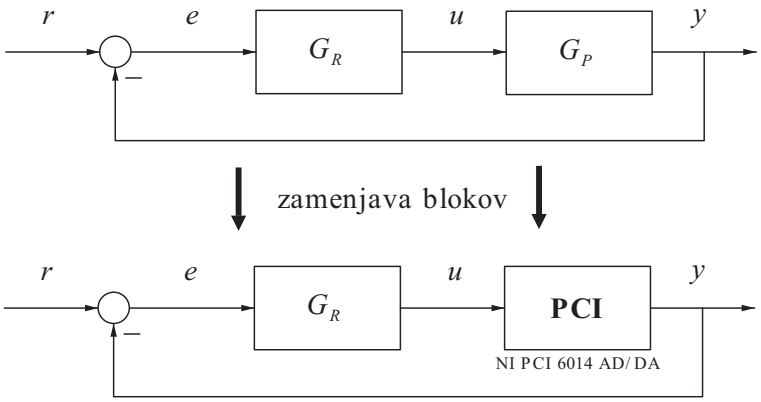
Za komunikacijo Simulinka z vmesnikom NI–PCI 6014 uporabljamo bloke iz knjižnice **NI–PCI**, ki se nahaja v osnovni mapi Simulinka (seveda mora biti knjižnica predhodno nameščena v okolju Simulink). Z dvojnim klikom na ikono NI–PCI odpremo mapo, v kateri so trije bloki, kot je prikazano na sliki 12.21. Prvi blok se imenuje *NI PCI 6014 A/D D/A* in omogoča tako analogno–digitalno kot digitalno-analogno pretvorbo. Ostala dva sta NI PCI 6014 A/D ter NI PCI 6014 D/A in ju uporabljamo samo za A/D– ali pa D/A–pretvorbo.



Blok *NI PCI 6014 A/D D/A* v simulacijski shemi dejansko predstavlja zunanji proces. Simulacijski signal, ki ga pripeljemo na njegov vhod, se z D/A–pretvornikom pretvori v analogno obliko in predstavlja vhod v proces, signal, ki ga dobimo na izhodu bloka, pa je rezultat A/D–pretvorbe in predstavlja izhod procesa.



Slika 12.21 - Izgled mape **NI-PCI**



Slika 12.22 - Ekvivalentni zaprtozančni shemi

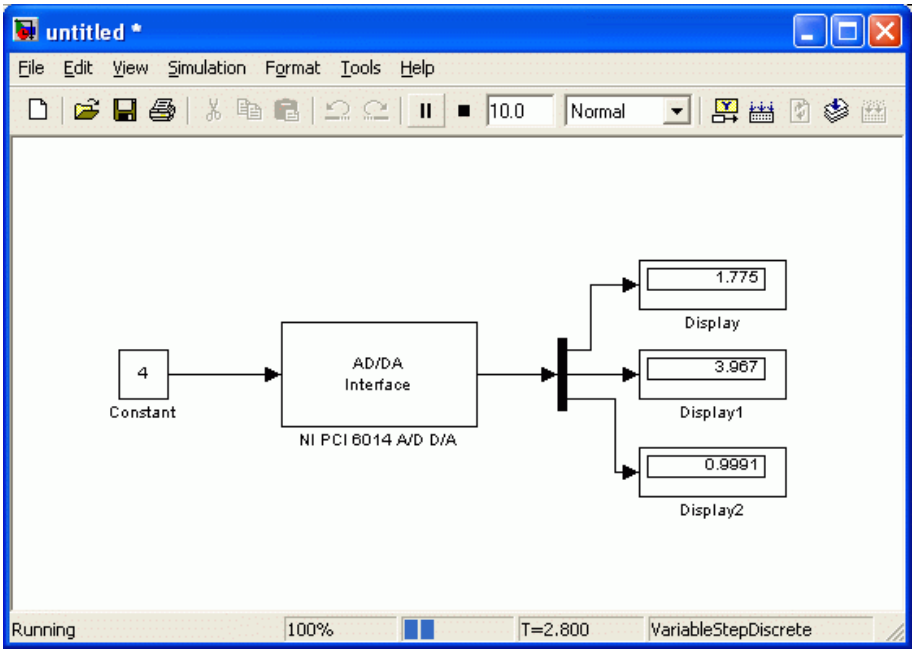
Ker lahko preko A/D– in D/A–pretvornika zajemamo več signalov oziroma kanalov, lahko z enim blokom *NI PCI 6014 A/D D/A* generiramo oziroma merimo več signalov naenkrat. V prvem primeru moramo pred vhodom v blok signale združiti z multipleksorjem, v drugem pa na izhodu bloka signal z demultipleksorjem razcepimo in na ta način dosežemo signale s posameznih kanalov. V primeru zaprtozančnega vodenja lahko blok uporabimo tako, da ga postavimo na mesto, kjer imamo v običajni simulacijski shemi model procesa. Slika 12.22 ponazarja omenjeno zamenjavo.

Parametri pogovornega okna (predstavljeni so na sliki 12.24) so naslednji:

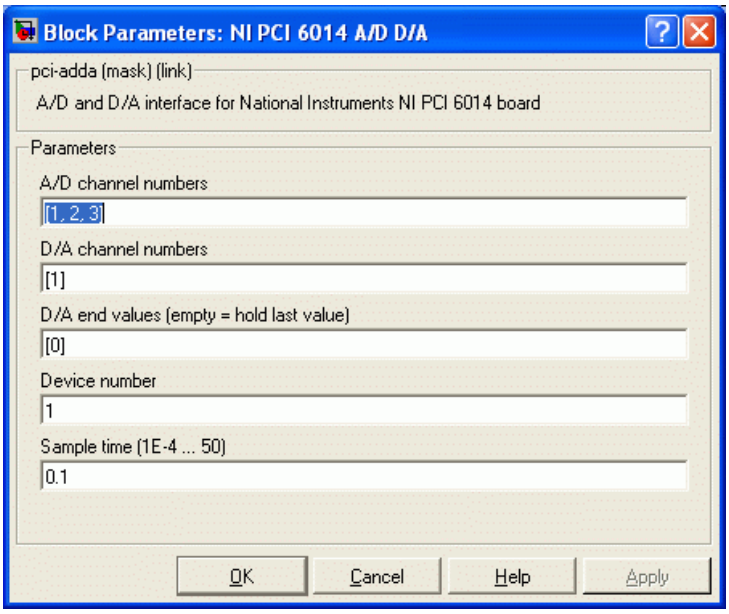
- **A/D channel numbers** – vektor s številkami kanalov A/D–pretvornika, na katere imamo priključene signale, ki jih želimo meriti. Številke kanalov so lahko med 1 in 8 in so enake številkam priključnih sponk na vmesniškem modulu.
- **D/A channel numbers** – vektor s številkami kanalov D/A–pretvornika, na katerih želimo generirati izhodna signala. Številki kanalov sta lahko 1 in 2 in sta enaki številkama priključnih sponk na vmesniškem modulu.

- **D/A end values** – vrednosti, ki jih zavzamejo podani D/A–kanali po končanem simulacijskem teku. Dimenziji vektorjev *D/A channel numbers* in *D/A end values* se morata ujemati, drugače nam sistem javi napako.
- **Device number** – številka priključene kartice, s katere zajemamo signale. Nastavljena vrednost je 1 in je običajno ni potrebno spreminjati.
- **Sample time** – čas vzorčenja oziroma čas med dvema zaporednima A/D–pretvorbama. Najmanjši možni čas vzorčenja je odvisen od hitrosti računalnika in od kompleksnosti simulacijske sheme. Če izberemo prekratek čas vzorčenja, pride do izpuščanja vzorcev, kar pomeni, da program ne uspe dovolj hitro izvajati obeh pretvorb. Če pride do izpuščanja, programski vmesnik vsako sekundo javi odstotek izgubljenih vzorcev, na koncu simulacije pa poda končno poročilo o številu izgubljenih vzorcev. Če se opozorilo o izgubljanju ponavlja, je potrebno povečati čas vzorčenja! Do izpuščanja vzorcev pa lahko pride tudi v primeru, ko med simulacijskim tekom premikamo simulacijsko okno, odpiramo ali premikamo druga pogovorna okna in podobno, saj takrat okolje Windows blokira izvajanje vseh programov, torej tudi program za zajem podatkov.

Poglejmo si dva primera uporabe opisanega bloka. V prvem naj ima proces en vhod in tri izhode. Denimo, da ga vzbujamo s konstantnim signalom $u = 4$ in da želimo sproti spremljati vse tri izhodne veličine. Shemo sestavimo, kot kaže slika 12.23. Na izhodu s pomočjo demultipleksorja signale razcepimo in vsakega povežemo na svoj prikazovalnik. Z dvoklikom odpremo nastavitveno okno (slika 12.24) in vnesemo predlagane nastavitve. Merjene signale iz procesa moramo fizično povezati s sponkami 1, 2 in 3.

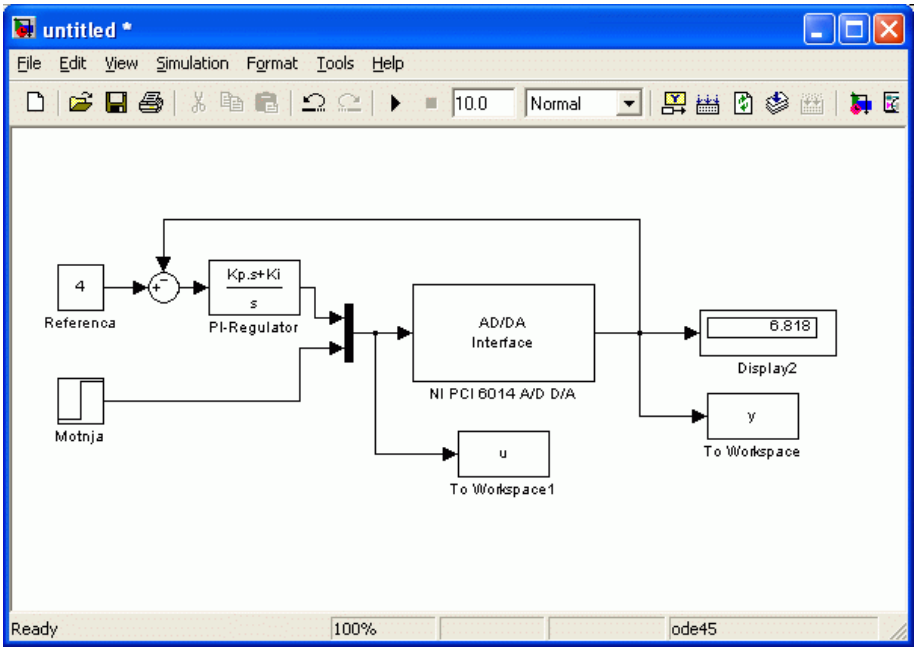


Slika 12.23 - Shema za prvi primer uporabe vmesniškega bloka *NI-PCI 6014 A/D D/A*



Slika 12.24 - Nastavitve vmesniškega bloka *NI-PCI 6014 A/D D/A*

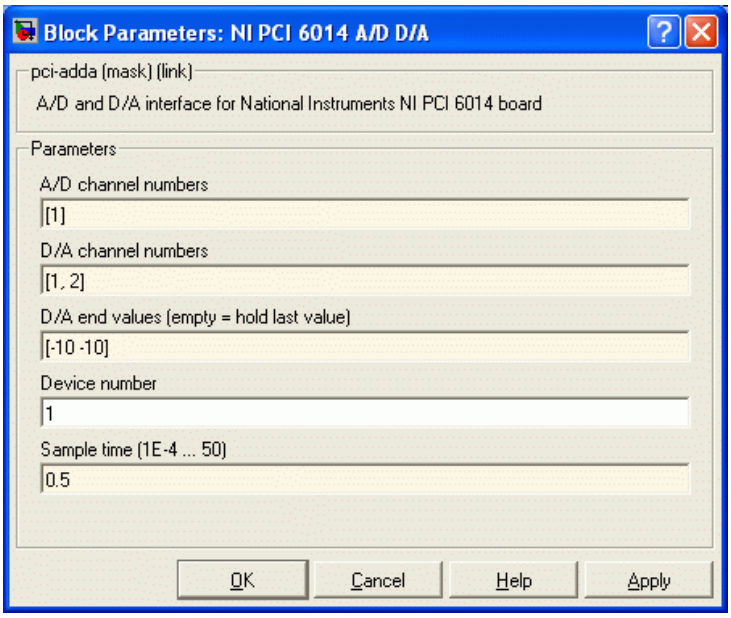
V drugem primeru predpostavimo, da hočemo eksperimentalno preizkusiti kvaliteto regulacijskega delovanja sistema vodenja. Proces naj ima tokrat en izhod in dva vhoda, od katerih je prvi namenjen regulirni veličini *u*, drugi pa predstavlja motnjo. Slednjo realiziramo tako, da na drugi vhod multipleksorja pripeljemo stopničasti signal in ga skupaj z regulirnim signalom povežemo na vhod procesa, tako kot je prikazano na sliki 12.25.



Slika 12.25 - Shema za drugi primer uporabe vmesniškega bloka

V tem primeru se spremenijo tudi nastavitve vmesniškega bloka, kar prikazuje slika 12.26. Sedaj imamo vektorja pri *D/A channel numbers* in *D/A end values*. Vrednosti slednjega sta –10, kar pomeni, da hočemo to vrednost na vhodu v proces, ko se bo

simulacijski tek končal (navadno to pomeni, da ni vzbujanja; druga zelo pogosta vrednost je 0, odvisno seveda od območja signalov). Spremenili smo tudi vrednost časa vzorčenja, ker ima naš namišljeni proces časovno konstanto $T = 5$ sekund.



Slika 12.26 - Nastavitve vmesniškega bloka *v* drugem primeru

13. ANALIZA MODELOV

V tem delu bo prikazano numerično orodje, ki omogoča analizo modelov, narejenih s pomočjo Simulinkovega grafičnega uporabniškega vmesnika. Sestavljajo ga funkcije za simulacijo in linearizacijo modelov ter funkcije za iskanje ravnotežnih točk.

Simulink lahko uporabljamo na tri načine. Najbolj interaktivni izmed njih je izvajanje in nadzorovanje simulacije s pomočjo **grafičnega vmesnika** in prikazovanje rezultatov na grafih *Scope*. To je najenostavnejši in hkrati uporabniku najbolj prijazen način, saj ima dostop do vseh parametrov modela in simulacijske metode. Slabost tega postopka je nezmožnost izvajanja velikih serij simulacijskih tekov.

Drugi način je zaganjanje simulacije iz **Matlabove ukazne vrstice**. Ta metoda ne omogoča takšne interaktivnosti z uporabnikom, je pa veliko bolj prilagodljiva. Rezultate simulacije, ki jih dobimo na ta način, lahko zelo hitro in enostavno analiziramo s pomočjo m–funkcij, zelo enostavno pa je tudi izvesti serijo simulacijskih tekov z uporabo *for*–zank.

Najkompleksnejši in hkrati najbolj prilagodljiv način simulacije pa je neposredno dostopanje do **s–funkcije modela**. Modeli, zgrajeni v Simulinku, so v Matlabu predstavljeni z s–funkcijami, v katerih je zapisano dinamično obnašanje modelov. S–funkcijo lahko kličemo na več različnih načinov. Vsebuje informacijo o: številu vhodov, izhodov in stanj (diskretnih in zveznih) modela ter vrednosti stanj in izhodov v vsakem simulacijskem koraku. Vse funkcije, ki omogočajo analizo modelov, zgrajenih v Simulinku (npr. *linmod* in *trim*), delujejo na tem principu.

13.1 Simulacija modelov

Simulacija je v svojem bistvu reševanje sistemov diferencialnih enačb z numerično integracijo. Simulink nudi več različnih integracijskih metod, ker z eno samo ne moremo uspešno, natančno in dovolj hitro rešiti vseh možnih simulacijskih problemov. Razloga sta ponavadi različno dinamično obnašanje modelov in zahtevana natančnost simulacije. Prav ustrezna izbira integracijske metode in njenih parametrov pa je ključni del celotnega postopka simulacije modela.

Pogledali smo si že, kako zaženemo simulacijo v grafičnem vmesniku. Glavni parametri so *Start* in *Stop time* (začetni in končni čas simulacije), *Min step size* (minimalni korak integracije), *Max step size* (maksimalni korak integracije) ter *Relative* in *Absolute tolerance* (maksimalni relativna in absolutna napaka pri integraciji). Z zadnjimi štirimi določamo korak integracije. Integracijski algoritem ne uporabi manjše vrednosti od minimalnega koraka, razen če model ne vsebuje blokov iz mape **Discrete**, ki zahtevajo manjši čas vzorčenja. V določenih primerih lahko simulacijski algoritem vrne rezultate, ki so natančni, vendar neprimerni za risanje, ker ne dobimo gladke krivulje zaradi prevelikega koraka integracije. To se zgodi v primeru linearnih sistemov, ki jih vzbujamo z odsekoma zveznimi vhodi. Zato je takrat potrebno ustrezno definirati parameter *Max step size*. Večina integracijskih metod, ki jih določimo v parametru *Solver*, pa uporablja spremenljivi korak

integracije (velikost koraka se spreminja glede na napako pri integraciji), zato natančnost reguliramo s tolerančnima parametroma.

Vsako simulacijo, ki jo poganjamo iz grafičnega vmesnika, pa lahko zaženemo tudi iz ukazne vrstice z uporabo ukaza *sim*

```
>> [t,x,y] = sim('model',cas_interval,opcije,ut),
```

kjer je *model* ime Simulinkove simulacijske sheme (brez končnice), *cas_interval* vektor z elementoma začetnega in končnega časa simulacije, *opcije* struktura, kjer podamo parametre simulacijske metode, in *ut* vektor zunanjega vzbujanja modela. Obvezen parameter je samo ime modela, medtem ko druge lahko opuščamo. Če želimo vnesti samo ime modela in *opcije*, časovnega intervala pa ne (simuliramo s privzetim intervalom [0 10]), moramo med oba vnesti prazno matriko: `[t,x,y] = sim('model',[],opcije)`. Izhodni argumenti funkcije *sim* pa so simulacijski čas (*t*), stanja (*x*) in izhodi modela (*y*).

Simulacija, ki jo zaganjamo iz ukazne vrstice, ima naslednje prednosti pred simulacijo v uporabniškem vmesniku:

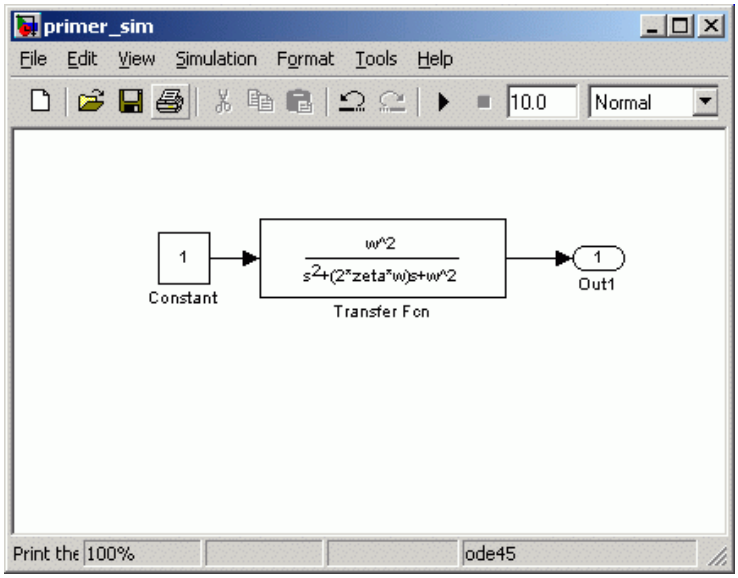
- lahko spremenimo začetna stanja,
- če ne definiramo izhodnih spremenljivk funkcije, nam funkcija avtomatično izriše diagram časovnega poteka izhodne spremenljivke ali stanja,
- poleg Simulink–modelov lahko simuliramo tudi m– in mex–datoteke (to so modeli, napisani v jeziku c),
- simulacijo lahko izvajamo v zanki, kjer parametre modela iterativno spremenjamo,
- za majhne modele se simulacija izvaja hitreje.

Pogledali si bomo preprost primer, kjer bodo vidne nekatere omenjene prednosti. Denimo, da hočemo simulirati odziv sistema drugega reda

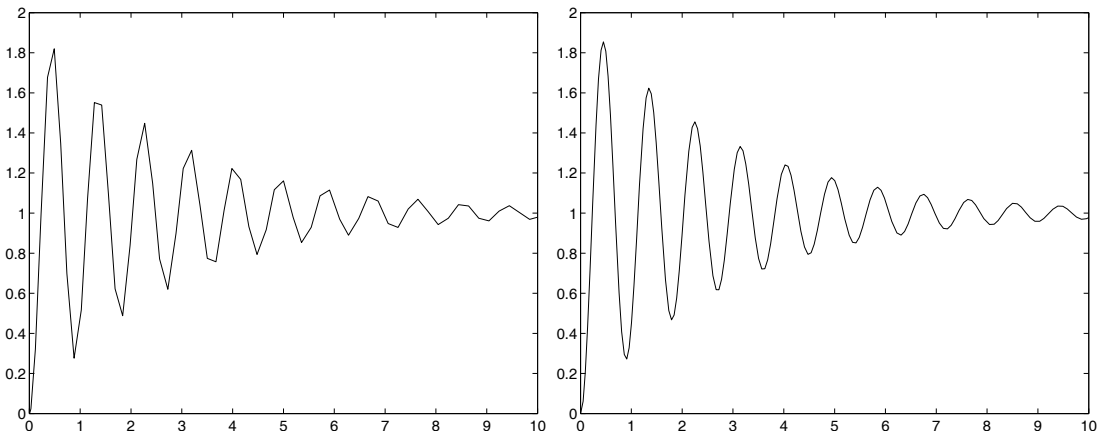
$$G(s) = \frac{\omega^2}{s^2 + 2\zeta\omega s + \omega^2} \tag{13.1}$$

s parametroma $\zeta = 0.05$ in $\omega = 7$ na stopničasto vzbujanje $r = 1$ pri $t = 0$ s. Zanima nas tudi, kako parameter ζ vpliva na potek izhoda sistema. Najprej naredimo shemo v Simulinku, kot jo prikazuje slika 13.1. Izhod *Out1* moramo vključiti v shemo, če hočemo dobiti rezultate v izhodni parameter *y*. Nato v ukazni vrstici definiramo potrebne parametre in poženemo simulacijo s pomočjo funkcije *sim*.

```
>> w = 7;
>> zeta = 0.05;
>> [t,x,y] = sim('primer_sim');
>> size(y)

ans =

    65     1

>> plot(t,y)
```



Slika 13.1 - Shema v Simulinku, ki jo uporabljali iz Matlabove ukazne vrstice



Slika 13.2 - Rezultat simulacije z osnovnimi parametri (levo) in nižjo relativno toleranco (desno)

Dobili smo odziv sistema v 65 točkah. V stanjih *x* dobimo matriko 65x2, ker je sistem drugega reda in ima zato dve stanji. Na levem delu slike 13.2 je prikazan graf časovnega poteka izhoda. Vidimo lahko, da krivulja ni gladka, kljub temu da je prave eksponentno padajoče sinusne oblike. Znižati moramo maksimalni korak integracije ali eno od toleranc. To naredimo s pomočjo strukture *opcije*, ki jo dobimo s klicem *simget*, parametre pa nastavljamo s klicem *simset*. Poglejmo primer.

```
>> opcije = simget

opcije =

                    AbsTol: []
                     Debug: []
                Decimation: []
             DstWorkspace: []
            FinalStateName: []
```

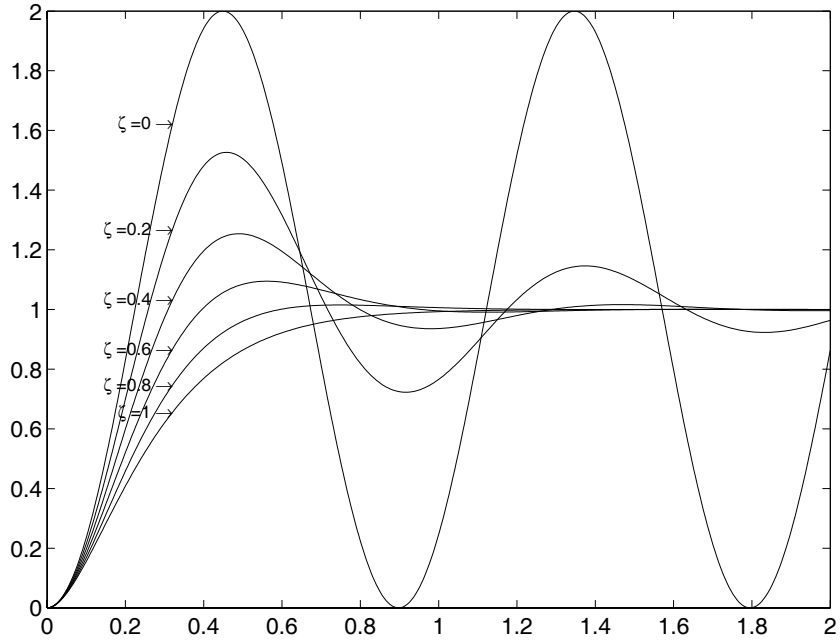
```
                    FixedStep: []
                  InitialState: []
                   InitialStep: []
                      MaxOrder: []
                    SaveFormat: []
                 MaxDataPoints: []
                       MaxStep: []
                       MinStep: []
                  OutputPoints: []
              OutputVariables: []
                       Refine: []
                       RelTol: []
                       Solver: []
                  SrcWorkspace: []
                         Trace: []
                    ZeroCross: []
            ExtrapolationOrder: []
         NumberNewtonIterations: []

      >> opcije = simset('RelTol',1e-6);
      >> [t,x,y] = sim('primer_sim',[],opcije);
      >> size(y)

      ans =

         149     1

      >> plot(t,y)
```

Zdaj smo dobili 149 izračunanih točk, kar pomeni, da je algoritem zmanjšal korak na kritičnih delih. Rezultat je bolj gladka krivulja, kar je prikazano na desnem delu slike 13.2.

Poglejmo, kako bi simulirali odziv sistema od $t = 0$ s do $t = 3$ s, če se parameter ζ spreminja od 0 do 1 v korakih 0,2. V zanki v vsakem koraku napravimo simulacijo odziva, vrišemo vsak rezultat na diagram in ga označimo z vrednostjo parametra in puščico, ki kaže na ustrezno krivuljo. Tokrat definirajmo parameter *MaxStep*.

```
opcije = simset('MaxStep',5e-3);
figure
set(gca,'FontSize',13,'NextPlot','Add')
for zeta = 0:0.2:1
    [t,x,y] = sim('primer_sim',[0 2],opcije);
    plot(t,y);
    % Hočemo oznake v levem delu slike
    len = round(length(t)/6);
    text(t(len),y(len),...
         strcat('\zeta = ',num2str(zeta),...
         ' \rightarrow'),'HorizontalAlignment','right');
end
```

Slika 13.3 prikazuje rezultat. Oznake imajo zdaj približno isto koordinato abscise, ker je število točk določeno z maksimalnim korakom integracije (če tega parametra ne nastavimo dovolj nizko, algoritem korak po potrebi zmanjša glede na tolerance).



Slika 13.3 - Študija vpliva parametra ζ na odziv sistema drugega reda

Po potrebi lahko izberemo tudi drugo integracijsko metodo. Pri togih sistemih se na primer uporabljajo ti. Stiff-metode. Primerov ne bomo navajali, povejmo le, da v *opcije* v lastnost *solver* zapišemo *ode45*, *ode23*, *ode113*, *ode15s* ali *ode23* za metode s spremenljivim korakom ter *ode5*, *ode4*, *ode3*, *ode2* ali *ode1* za metode s fiksnim korakom integracije.

13.2 Linearizacija modelov

Simulink omogoča linearizacijo splošnega nelinearnega modela okrog delovne točke. Linearizirani model je zapisan v prostoru stanj. Ukaz ima obliko

```
[A,B,C,D] = linmod('model',X,U),
```

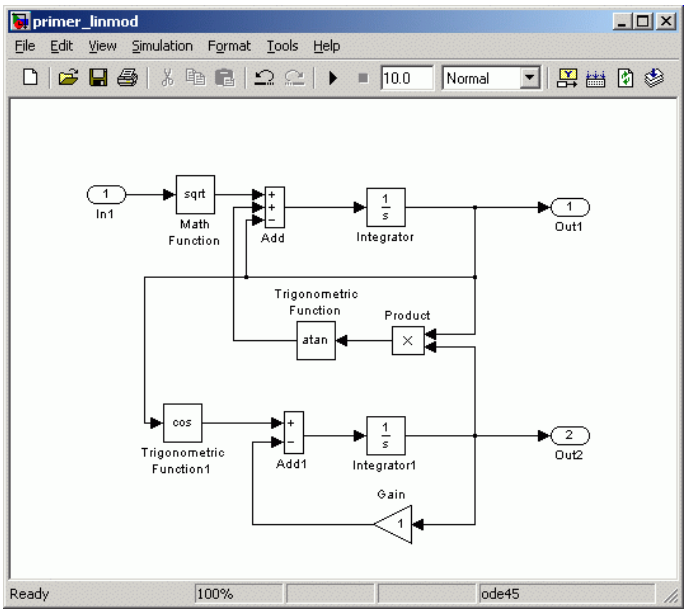
kjer je *model* ime sistema nelinearnih diferencialnih enačb, podanega v Simulink–modelu ali s–funkciji, *X* vektor vrednosti stanj sistema, *U* vektor vrednosti vhodov v sistem, *A*, *B*, *C* in *D* pa so matrike lineariziranega sistema v prostoru stanj. Če parametrov *X* in *U* ne podamo, bo *linmod* opravil linearizacijo okrog delovne točke **X** = **0** in **U** = **0**. Za primer linearizirajmo sistem

$$\begin{aligned} \dot{x}_1 &= \arctan\left(x_1 \cdot x_2\right) - x_1 + \sqrt{u} \\ \dot{x}_2 &= -2x_2 + \cos x_1 \end{aligned} \tag{13.2}$$

okoli delovne točke

$$\begin{aligned} \mathbf{x} &= \begin{bmatrix} x_1 & x_2 \end{bmatrix}^T = \begin{bmatrix} 1.6322 & -0.0614 \end{bmatrix}^T \\ u &= 3 \end{aligned} . \tag{13.3}$$

Najprej zgradimo shemo, kot je prikazano na sliki 13.4. Vhode in izhode sistema moramo označiti s temu namenjenimi bloki.



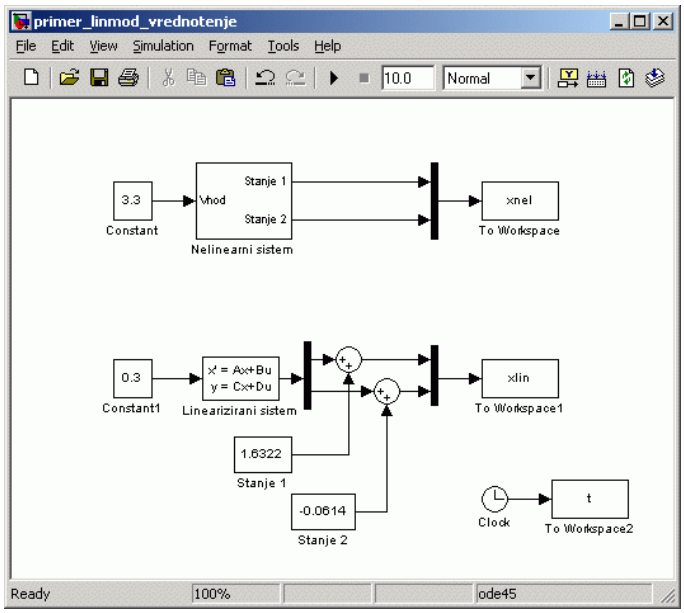
Slika 13.4 - Shema modela za linearizacijo s funkcijo *linmod*

Nato izvedemo naslednji ukaz:

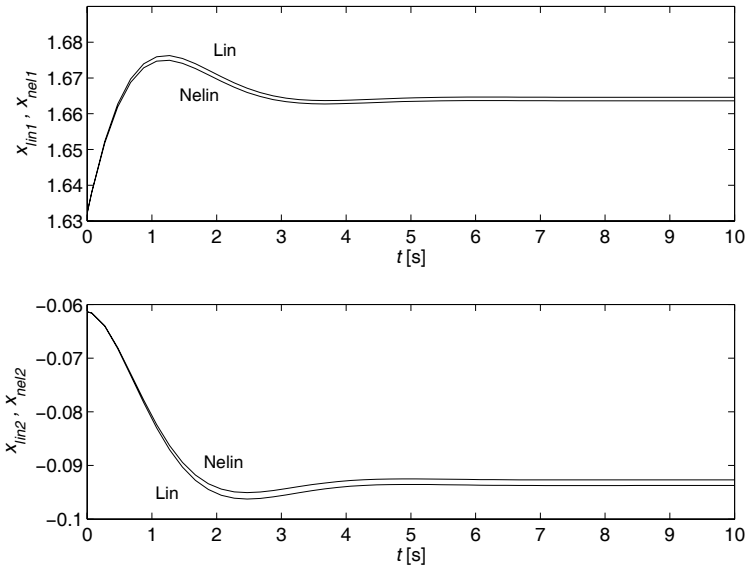
```
>> [A,B,C,D] = linmod('primer_linmod',[ 1.6322 -0.0614],3)

A =

   -1.0608    1.6160
   -0.9981   -1.0000

B =

    0.2887
         0

C =

    1.0000         0
         0    1.0000

D =

     0
     0
```

Dobili smo matrike lineariziranega sistema. Primerjajmo zdaj odzive linearnega in originalnega nelinearnega sistema. Prvotno shemo preuredimo tako, da naredimo podsistem nelinearnega modela in v začetni stanji integratorjev zapišemo stanji delovne točke. Dodamo še blok *State space* z matrikami lineariziranega modela in ga vzbujamo z $\Delta u = 0{,}3$. Izhodoma linearnega modela prištejemo vrednosti izhodov v

delovni točki. Slika 13.5 prikazuje shemo v Simulinku, slika 13.6 pa rezultate primerjave odzivov nelinearnega in linearizirranega modela. Vidimo, da sta odziva zelo podobna, vendar se že pri relativno majhnem odmiku od delovne točke (10%) razlikujeta, kar dokazuje, da je linearizirani sistem uporaben samo v ozki okolici delovne točke.



Slika 13.5 - Shema za vrednotenje linearizirranega modela



Slika 13.6 - Rezultati vrednotenja

Namesto funkcije *linmod* lahko uporabimo funkcijo *linmod2*, ki uporablja naprednejši algoritem za zmanjševanje pogreška pri linearizaciji. Druga razlika je v tem, da *linmod* linearizira vsak nelinearni blok posebej, medtem ko *linmod2* dela majhne odmike (perturbacije) na vhodih in stanjih modela.

13.3 Iskanje ravnotežne točke

S pomočjo Simulinkove funkcije *trim* lahko poiščemo ravnotežno točko sistema diferencialnih enačb, v kateri je sistem v ustaljenem ali stacionarnem stanju. Iz matematičnega vidika je to točka, v kateri so vrednosti odvodov enake 0.

Funkcija *trim* v obliki

```
[x,u,y] = trim('model')
```

poišče ravnotežno točko x, u in y, ki je najbližja začetnim stanjem modela. Slednja lahko tudi podamo v vhodnih argumentih.

```
[x,u,y,dx] = trim('model',x0,u0,y0)
```

Algoritem s pomočjo kvadratne optimizacijske metode išče najbližje ravnotežno stanje; če ga ne najde, vrne vrednosti stanj iz točke, kjer so bili odvodi najmanjši. Podatke o končnih vrednostih odvoda lahko dobimo, če podamo izhodni parameter *dx*. Za primer izračunajmo ravnotežno točko sistema iz enačbe (13.2), če za začetno vrednost vhoda podamo $u_0 = 4{,}64$.

```
>> [x,u,y,dx] = trim('primer_linmod',[],4.64)

x =

    1.6330
   -0.0621

u =

    3.0070

y =

    1.6330
   -0.0621

dx =

  1.0e-012 *

   -0.0913
   -0.1102
```

14. POGLOBLJENA UPORABA SIMULINKA

V tem poglavju bomo natančneje predstavili, kako deluje Simulink na nivoju s–funkcij. Pogledali si bomo zgradbo in delovanje s–funkcij ter podali primer gradnje take funkcije. Na koncu bomo skozi problem optimizacije pokazali še, kako lahko s kombinirano uporabo Matlaba in Simulinka zgradimo zmogljive algoritme, ki združujejo prednosti obeh okolij.

Gradnja modelov v Simulinku je smiselna predvsem s stališča enostavnosti in prijaznosti uporabniku, ni pa optimalna glede simulacije. Prvi Matlabov korak je generiranje podatkovne strukture, ki je optimalna z vidika simulacije. V tem koraku Matlab uredi vrstni red blokov, jih nadomesti s številskimi vrednostmi itd. Po končanem urejanju modela je podatkovna struktura primerna za zagon simulacije. Sistem zgradi t.i. **s–funkcijo**, ki je uporabniku dostopna iz Matlabove ukazne vrstice. V tej funkciji sta podana vhodno–izhodno obnašanje in celotna dinamika modela. Tako definiran model lahko simuliramo s pomočjo ene od metod numerične integracije, linearizirramo in iščemo ravnotežne točke modela.

14.1 Kako napišemo s–funkcijo

S–funkcija deluje na enak način kot katerakoli Matlabova funkcija, ima pa določeno sintakso

```
sys = proces(t,x,u,flag),
```

kjer je *proces* ime modela, parameter *flag* pa določa informacijo, ki jo funkcija vrne v izhodni spremenljivki *sys*. Ob vrednosti *flag* = 1 na primer funkcija vrne odvode stanj modela v delovni točki ob času *t*, pri vektorju stanj *x* in vhodni spremenljivki *u*. S–funkcijo lahko napišemo v obliki navadne m–datoteke ali mex–datoteke (c–jevska ali fortranova funkcija). Način podajanja s–funkcij je odvisen od zahtevnosti problema, uporabe in hitrosti, ki je zahtevana za izvajanje.

Uporabniško s–funkcijo lahko podamo znotraj Simulinkove sheme, tako da vstavimo blok *S–function* iz mape **User–defined Functions** in vanj vpišemo ime te funkcije. Z maskiranjem lahko takemu bloku dodamo še običajni vmesnik, ki ga imajo ostali Simulinkovi bloki, in definiramo lastne parametre. Na ta način je uporabniku omogočeno generiranje lastnih blokov in podmodelov.

Zgradba s–funkcij omogoča izgradnjo modelov za reševanje problemov s časovno zveznimi, diskretnimi in hibridnimi sistemi. Ravno zaradi tega ima s–funkcija točno določeno strukturo. Ključni element s–funkcije je parameter *flag*, ki definira informacijo izhoda funkcije:

- *flag* = 0 : s–funkcija vrne velikosti parametrov in začetnih pogojev
- *flag* = 1 : s–funkcija vrne vrednosti odvodov stanj *dx/dt*
- *flag* = 2 : s–funkcija vrne diskretna stanja $x(n+1)$
- *flag* = 3 : s–funkcija vrne vrednosti izhodnih spremenljivk
- *flag* = 4 : s–funkcija vrne čas naslednjega časovnega intervala

Vsaka od vrednosti zastavice posreduje del informacije, ki je potrebna za izvajanje simulacije.

Delovanje s–funkcije si bomo podrobneje pogledali na primeru omejenega integratorja. V komentarjih med klici ukazov znotraj funkcije bodo podane osnovne informacije o elementih, ki jih uporabljamo pri gradnji take funkcije.

```
function [sys,x0,str,ts] = limintm(t,x,u,flag,lb,ub,xi)
%
switch flag
  case 0
    [sys,x0,str,ts] = mdlInitializeSizes(lb,ub,xi);
  case 1
    sys = mdlDerivatives(t,x,u,lb,ub);
  case {2,4,9}
    sys = []; % neuporabljene zastavice
  case 3
    sys = mdlOutputs(t,x,u);
  otherwise
    error(['Napačna zastavica = ',num2str(flag)]);
end
%
%=============================================================================
% mdlInitializeSizes
% Vrne vektor sizes, začetne pogoje in čase vzorčenja za funkcijo
%=============================================================================
function [sys,x0,str,ts] = mdlInitializeSizes(lb,ub,xi)

% Priprava vektorja sizes
sizes = simsizes;    % Inicializira vektor sizes
sizes.NumContStates  = 1;
sizes.NumDiscStates  = 0;
sizes.NumOutputs     = 1;
sizes.NumInputs      = 1;
sizes.DirFeedthrough = 0;
sizes.NumSampleTimes = 1;
sys = simsizes(sizes);
%
% Inicializacija
str = [];
x0  = xi;        % Začetni pogoji (podani od zunaj)
ts  = [0 0];   % Čas vzorčenja: [perioda, premik]
%
%=============================================================================
% mdlDerivatives
% Izračuna odvode zveznih stanj - tu je jedro funkcije
%=============================================================================
function sys = mdlDerivatives(t,x,u,lb,ub)

if (x <= lb & u < 0)  | (x>= ub & u>0 )
  sys = 0;
else
  sys = u;
end
```

```
%=============================================================================
% mdlOutputs
% Vrne vektor izhodov s-funkcije
%=============================================================================
function sys = mdlOutputs(t,x,u)

sys = x;
%
% Konec definicij funkcij
```

Razložimo najprej klic funkcije. Vhodni parametri so *t*, *x*, *u*, *flag* (standardni parametri) ter *lb* (spodnja meja), *ub* (zgornja meja) in *xi* (začetni pogoji), ki so zunanji parametri. To pomeni, da jih je treba podati ob vsakem klicu te funkcije. V prvem delu je stavek *switch*, ki nas usmeri na ustrezno podfunkcijo *mdlInitializeSizes* (kliče se samo enkrat), *mdlDerivatives* (odvodi) ali *mdlOutputs* (podajanje izhodov).

Ključna elementa funkcije *mdlInitializeSizes* sta vektor x_0 (vektor začetnih stanj) in vektorj *sizes*, ki mora vsebovati naslednje elemente:

- sizes(1) – število zveznih stanj
- sizes(2) – število diskretnih stanj
- sizes(3) – število izhodnih spremenljivk
- sizes(4) – število vhodnih spremenljivk
- sizes(5) – mora biti 0 (rezervirano za iskanje korenov)
- sizes(6) – zastavica direktne povezave – če uporabimo v funkciji izhoda tudi vektor *u*, jo postavimo na 1, sicer pa 0
- sizes(7) – število časov vzorčenja (lahko izpustimo, če imamo samo enega)

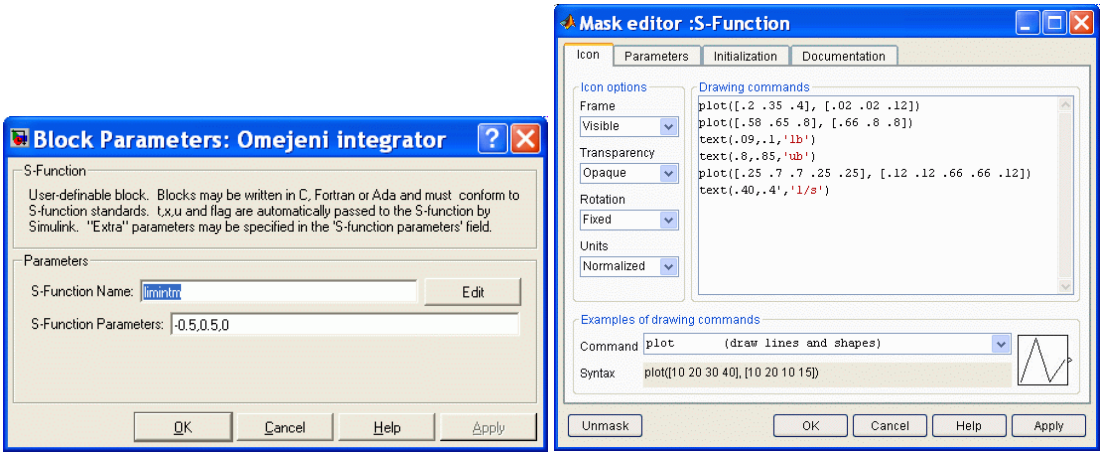
Nato s klicem `sys = simsizes(sizes)` pripravimo podatke, ki jih funkcija vrne po končanju inicializacije. S klicem `T0 = [0 0]` definiramo, da je naš sistem časovno zvezen.

V funkciji *mdlDerivatives* izračunamo vrednosti odvodov stanj – vhod v funkcijo *u* je izhod te podfunkcije, če stanje *x* ne krši zgornje ali spodnje meje. V tem primeru vrnemo vrednost odvoda 0, kar pomeni, da se vrednost stanj ne bo spremenila. V primeru zveznega sistema sta za simulacijo potrebni le informaciji o odvodih stanj v vsakem trenutku simulacije (*flag* = 1) in izhodih sistema (*flag* = 3). Opozoriti velja, da se za razliko od klica funkcije z zastavico *flag* = 0 ostali klici pojavljajo v vsakem računskem koraku simulacije.

Katerokoli obliko s–funkcije lahko pretvorimo v blok v Simulinkovi shemi. V ta namen je potrebno v uporabniški vmesnik funkcijskega bloka *S–function* vpisati ime s–funkcije in njene parametre. Blok namreč omogoča prenos določenih parametrov, ki sledijo formalni strukturi parametrov *t*, *x*, *u* in *flag*. Na koncu lahko tak blok še maskiramo. Maskiranje bo prikazano na primeru.

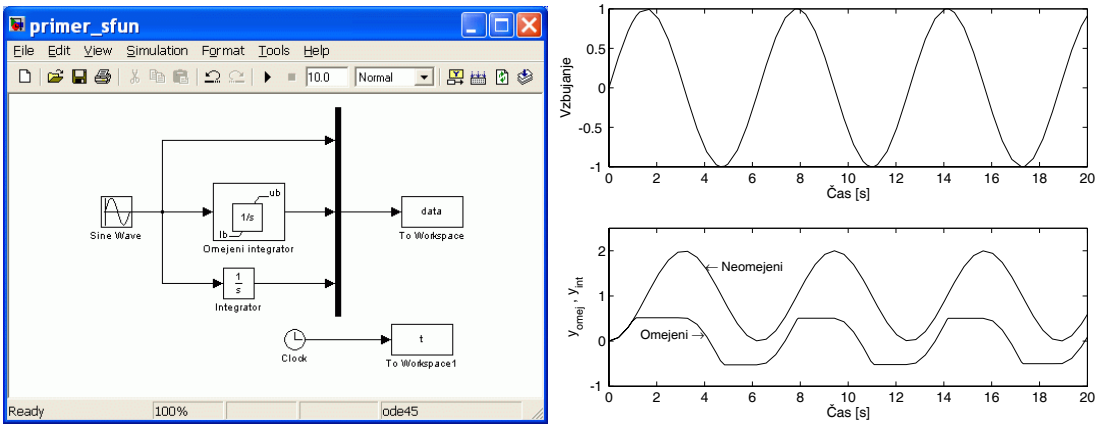
V shemo vnesemo izvor sinusnega nihanja ter bloka *S–function* in *Scope*. Odpremo prvi blok in v polji vnesemo ime m–datoteke, v kateri se nahaja s–funkcija, in zunanje vhodne parametre. Denimo, da hočemo omejiti integrator pri *lb* = –0.5, *ub* = 0.5 in *x0* = 0. Levi del slike 14.1 prikazuje okno parametrov bloka *S–function*. Nato maskiramo blok po že omenjenem postopku in definiramo vse tri parametre kot spremenljivke v dialogu z imeni *Spodnja meja*, *Zgornja meja* in *Zacetno stanje*. Pred tem moramo v bloku s–funkcije parametre spremeniti v *lb*, *ub* in *xi*, ker jih bomo podajali v dialogu maskiranega bloka. Tokrat bomo pokazali še, kako priredimo izgled ikone. V meniju *Mask Editor/Icon* v polje *Drawing Commands* vnesemo ukaze, kot bi risali sliko na diagram, imamo pa dve možnosti – ali uporabljamo

absolutne koordinate in v parametru *Units* izberemo *Autoscale* ali pa uporabljamo koordinate od 0 do 1 in pri istem parametru izberemo *Normalized*. Na desnem delu slike 14.1 je prikazan primer kode za risanje preproste ikone v normaliziranih koordinatah.



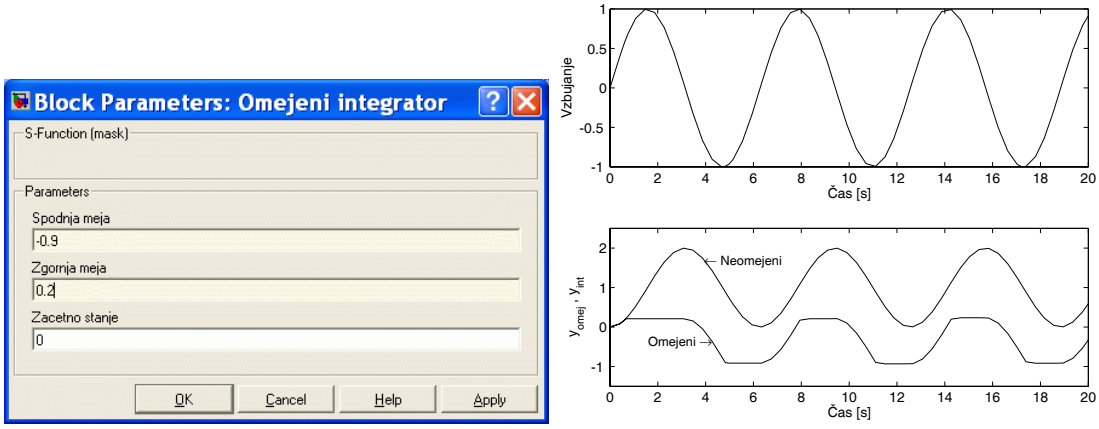
Slika 14.1 - Parametri bloka *S-function* (levo) in dialog za risanje ikone (desno)

Na istem diagramu bomo poleg odziva omejenega integratorja prikazali še izvorni signal in odziv navadnega integratorja, zato vse tri signale združimo na multipleksorju in peljemo na izhod *To Workspace* (levi del slike 14.2)



Slika 14.2 - Shema z vključeno maskirano s-funkcijo (levo) in rezultati simulacije (desno)

Rezultate simulacije prikazuje desni del slike 14.2. Vidimo, kako omejitev deluje na izhod integratorja. Zdaj pa spremenimo vrednosti meja, kot to kaže levi del slike 14.3, in ponovimo simulacijo. Izhod je zdaj v nasičenju pri vrednostih 0,2 in –0.9.



Slika 14.3 - Sprememba parametrov (levo) in novi rezultati (desno)

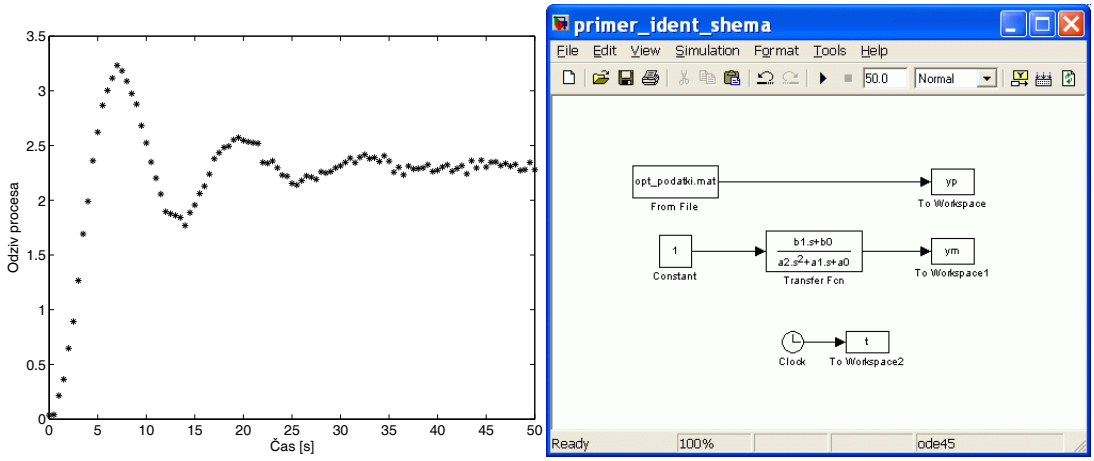
14.2 Optimizacija z uporabo modelov v Simulinku

Predstavili bomo primer optimizacije, ko imamo na voljo model v Simulinku, ki vrača vrednost kriterijske funkcije v vsakem simulacijskem teku, in funkcijo v Matlabu, ki poganja optimizacijski tek. Primer je zanimiv predvsem zato, ker gre za združevanje Matlabovih funkcij in Simulinkovih shem. Optimizacijski problem lahko v tem primeru razdelimo na dve skupini: **identifikacijo parametrov procesa s pomočjo prilagajanja modela** in **prilagajanje parametrov regulatorja za optimalen odziv glede na cenilko**.

Prilagajanje modela je metoda, ki s spreminjanjem parametrov modela odziv poskuša čimbolj približati posnetim podatkom. Metoda je dokaj uveljavljena predvsem pri simulacijsko usmerjenih problemih z zveznimi modeli. Njena poglavitna prednost je, da ni omejena na linearne sisteme. Bistvo metode je v minimizaciji kriterijske funkcije **izhodnega pogreška** oz. pogreška med izhodom modela in izhodom merjenega procesa. V splošnem lahko optimiramo tako strukturo kakor tudi parametre modela, zato je metoda uporabna tako za strukturno identifikacijo kakor tudi za ocenjevanje parametrov. Prav tako je metoda uporabna za sprotno in nesprotno identifikacijo, vendar teorija o tem presega okvir tega dela, zato bomo podali samo primer za nesprotno ocenjevanje parametrov.

Pri **iskanju optimalnih parametrov regulatorja** moramo podati obliko kriterijske funkcije, ki največkrat zajema izhodni pogrešek in regulirno veličino. Velikokrat uporabljamo optimizacijo z omejenimi parametri (*fminbnd*), ker ne želimo velikih ojačenj regulatorja oz. želimo doseči konsenz med kvaliteto zaprtozančnega vodenja in umirjenim potekom regulirne veličine.

Pogledali si bomo primer, kako identificirati parametre modela, če imamo odziv procesa posnet v datoteki. Denimo, da imamo v datoteki *opt_podatki.mat* posnet odziv procesa na stopnico amplitude 1 v 101 vzorcu s časom vzorčenja T_s = 0.5 s. Prikazuje ga levi del slike 14.4.



Slika 14.4 - Odziv procesa na stopnico v 101 točki (levo) in shema v Simulinku (desno)

V Simulinku zgradimo shemo, kot je prikazano na desnem delu slike 14.4. Simulacijski tek nam prinese vektor y_m, ki je odziv prenosne funkcije s trenutnimi parametri. Iz oblike odziva y_p vidimo, da je proces višjega reda in da je moten s šumom. Najprej izberemo prenosno funkcijo modela v obliki

$$G_m(s) = \frac{b_1 s + b_0}{a_2 s^2 + a_1 s + a_0} \tag{14.1}$$

in vsem parametrom priredimo začetno vrednost 1. Identifikacija parametrov a_2, a_1, a_0, b_1 in b_0 s pomočjo prilagajanja modela je izvedena s pomočjo naslednjega programa:

```
global a0 a1 a2
global b0 b1
%
% Definiramo začetne parametre
a2 = 1; a1 = 1; a0 = 1;
b1 = 1; b0 = 1;
%
% Nastavitve metode
opts = optimset('TolX',1e-3,'TolFun',1e-3);
%
% Odpremo diagram
figure
%
% Zagon optimizacije, kjer dobimo optimalne parametre
optimal = fminsearch('ident_pogr',[a2 a1 a0 b1 b0],opts);
%
% Simulacija končnega rezultata
a2 = optimal(1); a1 = optimal(2); a0 = optimal(3);
b1 = optimal(4); b0 = optimal(5);
sim('primer_ident_shema')
figure
plot(t,yp,'*',t,ym,'r')
tf([b1 b0],[a2 a1 a0])
```

Program uporablja funkcijo *fminsearch* za iskanje minimuma kriterijske funkcije, ki jo v vsakem optimizacijskem koraku dobi s klicem funkcije *ident_pogr*. Slednja izgleda nekako takole:

```
function err = ident_pogr(par)
%
% Parametra morata biti globalna
global a0 a1 a2
global b0 b1
%
a2 = par(1); a1 = par(2); a0 = par(3);
b1 = par(4); b0 = par(5);
%
% Simuliramo našo shemo z novimi parametri
sim('primer_ident_shema');
%
% Rišemo graf, da lahko sproti spremljamo dogajanje
plot(t,yp,'*',t,ym,'r')
drawnow
%
% Vrnemo vrednost cenilke
err = sum((yp-ym).^2);
```

Funkcija od *fminsearch* dobi trenutne parametre *par*, jih zapiše v spremenljivke a_0 – b_1 in izvede simulacijski tek sheme *primer_ident_shema.mdl*, prikazane na desnem delu slike 14.4. Opozorimo naj na to, da morajo biti spremenljivke definirane kot globalne spremenljivke, drugače se spremembe njihovih vrednosti znotraj funkcije *ident_pogr* ne bi odražale v shemi v Simulinku. Vrednost kriterijske funkcije je vsota kvadratov razlik med točkami procesa in točkami, dobljenimi s simulacijo modela. Da bi dobili enako število točk, kot jih je v procesu, v bloke *To Workspace* parametru *Sample time* priredimo vrednost 0.5 (čas vzorčenja). Funkcija *fminsearch* toliko časa spreminja parametre modela, dokler vrednost kriterijske funkcije ni minimalna oz. v mejah podanih toleranc. Rezultat optimizacije je prenosna funkcija modela

$$G_m(s) = \frac{-0.4008s + 0.8742}{1.5166s^2 + 0.4059s + 0.3815}, \tag{14.2}$$

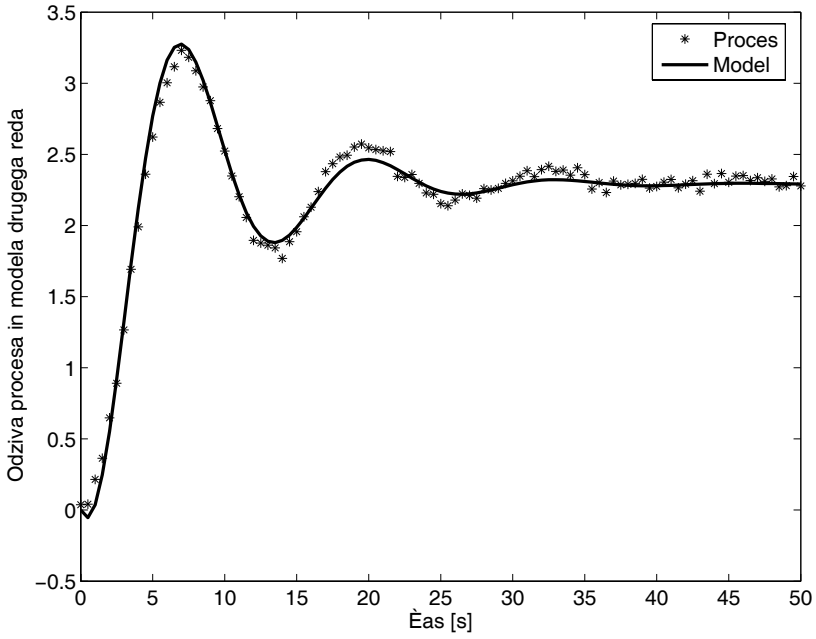
primerjavo odzivov pa prikazuje slika 14.5. Vidimo, da je prileganje odziva modela ustrezno, moti pa začetni del odziva, ki gre v negativno vrednost, kar pomeni, da smo dobili fazno neminimalen sistem. Zato poskusimo povečati red prenosne funkcije na 3, vendar pa s tem v splošnem povečamo tudi red optimizacijskega problema za dva parametra.

$$G_m(s) = \frac{b_2 s^2 + b_1 s + b_0}{a_3 s^3 + a_2 s^2 + a_1 s + a_0} \tag{14.3}$$

Ponovno izvedemo optimizacijo, s tem da moramo ustrezno spremeniti vrstico, kjer poganjamo optimizacijo.

```
% Zagon optimizacije, kjer dobimo optimalne parametre
optimal = fminsearch('ident_pogr_r3',[a3 a2 a1 a0 b2 b1 b0],opts);
```

Prav tako moramo v funkciji *ident_pogr* upoštevati, da imamo sedaj 7 parametrov.

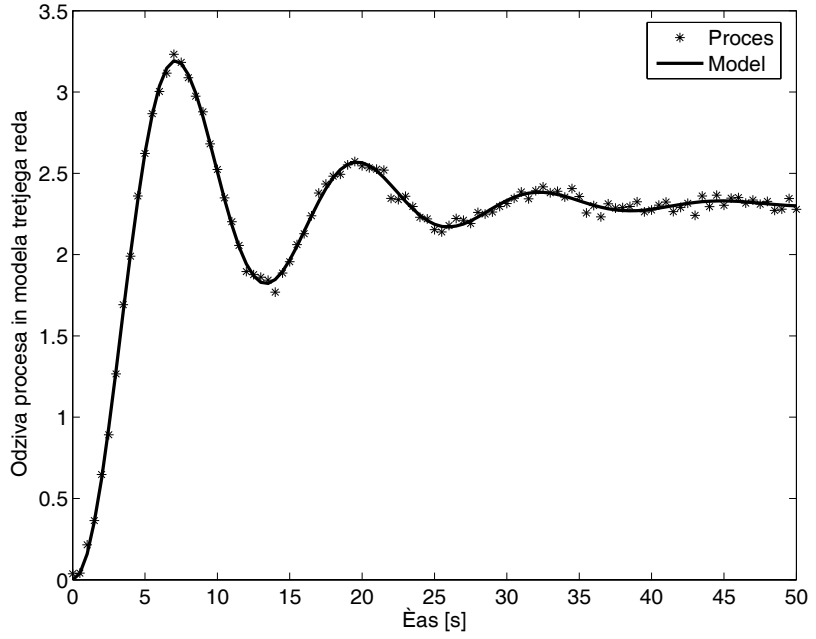


Slika 14.5 - Primerjava odziva modela drugega reda z odzivom procesa

Rezultat optimizacije modela tretjega reda predstavljata prenosna funkcija

$$G_m(s) = \frac{-0.0011s^2 + 0.5796s + 0.5448}{1.9007s^3 + 1.2885s^2 + 0.6735s + 0.2359} \tag{14.4}$$

in diagram na sliki 14.6. Vidimo, da se prileganje še izboljša in da ni več začetnega negativnega odziva.



Slika 14.6 - Primerjava odziva modela tretjega reda z odzivom procesa

Analizirajmo še ničle in pole dobljenega modela.

```
>> roots([b2 b1 b0])

ans =

  551.2171
   -0.9384

>> roots([a3 a2 a1 a0])

ans =

  -0.0994 + 0.4992i
  -0.0994 - 0.4992i
  -0.4791
```

Dobimo hitro ničlo v desni in počasno v levi *s*–polravnini, kar pomeni, da prvo ničlo lahko zanemarimo. Imamo dva konjugirano kompleksna pola in en realen pol, vsi so v levi polravnini. Prenosna funkcija originalnega procesa je bila

$$G_p(s) = \frac{0.3(s+1)}{(s+0.5)(s+0.1-0.5i)(s+0.1+0.5i)}, \tag{14.5}$$

kar pomeni, da smo z optimizacijo dobili ustrezno predstavitev procesa z modelom.

15. SEZNAM UPORABLJENIH FUNKCIJ

A

B

C

K

L

M

N

O

P

Q

R

S