Univerza v Ljubljani

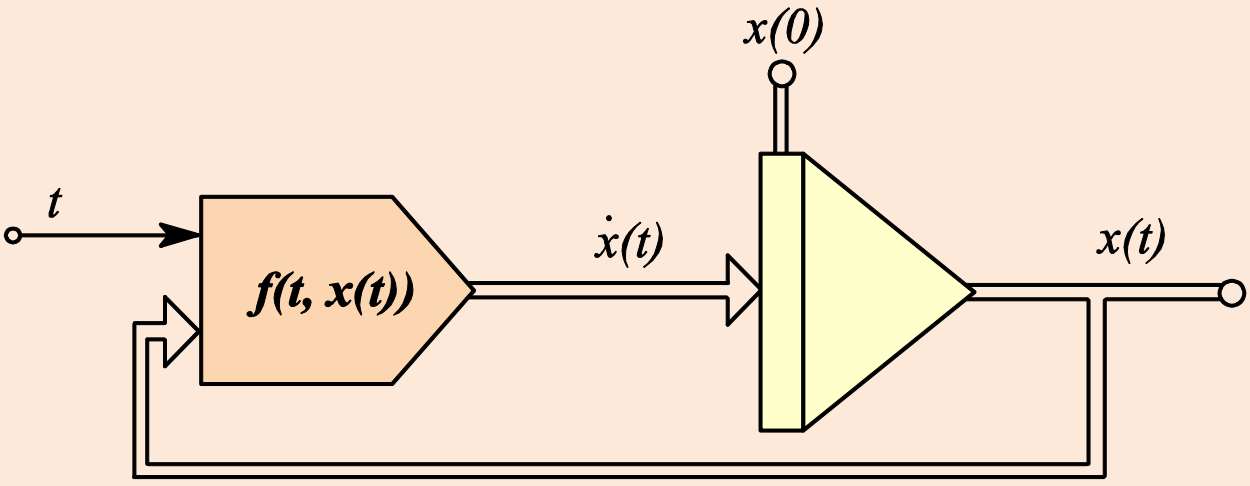
Fakulteta za elektrotehniko

Laboratorij za modeliranje, simulacijo in vodenje

Laboratorij za avtonomne mobilne sisteme

Računalniška simulacija

Borut Zupančič



Kazalo

1.

Uvod v simulacijo dinamičnih sistemov

Simulacija dinamičnih sistemov je ena najpomembnejših metod na področju analize in načrtovanja vodenja sistemov. Predstavlja sodobno metodologijo, ki direktno ali pa vsaj posredno spremlja vse sodobne metode analize in načrtovanja. Kljub izrednemu razvoju teorije vodenja sistemov je še vedno očitno, da najboljše rezultate dosegamo le z interaktivnim delom na računalniku, pri katerem imajo pomembno vlogo učinkoviti simulacijski jeziki in izkušnje uporabnika. Simulacija se uporablja praktično na vseh področjih, ki temeljijo na sistemskem pristopu obravnave procesov in sicer: pri vodenju sistemov, v robotiki in v računalništvu, v fiziki, biologiji, kemiji, medicini, farmaciji, pa tudi v ekonomskih, sociooloških in političnih vedah. Predstavlja pa navadno tudi eno od osnovnih metodologij pri reševanju interdisciplinarnih projektov.

Simulacija dinamičnih procesov je področje, ki je v zadnjih petdesetih letih takoj za signalnim procesiranjem najbolj drastično vplivalo na razvoj računalnikov. Obe področji namreč razvijata v smislu porabe računalniškega časa zelo potratne metode, ki so seveda učinkovite le ob uporabi sposobnih računalnikov.

Področje vodenja sistemov je eno prvih področij (takoj za letalsko industrijo), kjer so že pred več kot štiridesetimi leti s pridom uporabljali računalniško simulacijo. Orodje so takrat seveda predstavljali razni diferencialni analizatorji in pozneje analogni računalniki. Kasneje so se orodja zelo izpopolnila. Poleg analognih so se pojavili sposobni hibridni in digitalni računalniki skupno z ustrezno programsko opremo z ustreznimi simulacijskimi jeziki. Ker pa je bila računalniška

oprema, na kateri je bilo možno izvajati simulacijo, zelo draga in ker je simulacija veljala za računalniško zelo potratno metodo, se je pred razvojem cenenih mini in mikroračunalnikov uporabljala le kot skrajna možnost za reševanje omenjenih problemov.

V zadnjih desetletjih pa je simulacija zaradi izrednega razvoja materialne in programske opreme postala metodologija, ki jo načrtovalci sistemov vodenja običajno najprej uporabijo. Bistveno vlogo je imel pri tem razvoj sposobnih simulacijskih jezikov na cenenih mini in mikroračunalnikih (npr. osebni računalniki). To je omogočilo, da je simulacija postala dostopna vsakomur, tudi izobraževalnim institucijam in študentom. Zato je simulacija doživela nov vzpon in oživila številna teoretična področja sistemskih ved, pri čemer naj omenimo še zlasti področje nelinearnih sistemov. Raziskave kaotičnih sistemov predstavljajo aktualno področje nelinearnih sistemov, k razmahu tega področja pa je prispevala prav simulacija.

1.1 Definicije simulacije

Avtorji različno definirajo pojem simulacije. Omenili bomo nekatere definicije.

Simulacija dinamičnih sistemov je iterativna metoda, s pomočjo katere proučujemo funkcionalne lastnosti dinamičnih sistemov z eksperimentiranjem na ustreznem modelu realnega objekta. Kljub temu, da literatura mnogokrat bolj poudarja metode analize in sinteze, pa se pri reševanju realnih problemov vedno znova potrjuje velika vloga simulacije. Zato jo inženirji uspešno uporabljajo kot pomoč pri razumevanju problemov, kot orodje pri načrtovanju vodenja sistemov, za učenje osebja, ki nadzoruje vodene procese ter kot orodje, ki omogoča študirati nove ideje v fazi načrtovanja.

Pritsker (Pritsker, 1979) obravnava simulacijo in modeliranje kot dva nerazdružljiva pojma. Model predstavlja poenostavljen sistem, simulacija pa je posnemanje obnašanja sistema v realnem, skrčenem ali raztegnjenem času na ta način, da eksperimentiramo z modelom. Model, ki ga izvedemo na računalniku, je računalniški simulacijski model ali krajše simulacijski model.

Mc Leod (Mc Leod, 1987b) definira simulacijo kot uporabo modela za eksperimentiranje. Na ta način skušamo napovedati možno obnašanje sistema ali situacije, ki jo proučujemo. Podobno definicijo je uporabil tudi Korn (Korn, Wait, 1978).

Podobno kot Pritsker tudi Schmidt (Schmidt, 1986) obravnava modeliranje in simulacijo kot dva nerazdružljiva pojma. Simulacija je postopek, ki omogoča proučevanje realnega sistema s pomočjo drugega sistema, ki ga imenujemo realni model. Pomembno je, da imata realni sistem in realni model enak konceptualni model glede na značilnosti, ki jih proučujemo, vendar z realnim modelom laže eksperimentiramo. Simulacija pomeni torej določitev realnega (simulacijskega) modela in eksperimentiranje z njim z namenom študija oz. analize realnega sistema in napovedovanja njegovega obnašanja.

1.2 Splošno o simulaciji dinamičnih sistemov

Atherton (Atherton, 1986) omenja naslednja dva bistvena razloga, zakaj načrtovalci sistemov vodenja uporabljajo simulacijo:

- Analitične metode, pa čeprav jih lahko uporabljamo v obsežnih CACSD (Computer Aided Control System Design) paketih, so često zelo omejene, če jih uporabljamo za reševanje realnih problemov. Tipično je npr. področje nelinearnih sistemov in še bolj področje kaotičnih sistemov in stohastičnih sistemov. Simulacija predstavlja najuporabnejšo metodo tudi pri študiju vpliva sprememb parametrov na obnašanje sistema in s tem tudi pri študiju robustnosti.
- Za simulacijo v realnem času ob priključenih realnih objektih (hardware in the loop - HIL). Analogni računalniki so bili dolgo edino primerni za to vrsto simulacije. Z razvojem sodobne materialne in programske opreme lahko tudi digitalni računalniki izredno uspešno vršijo simulacijo v realnem času.

Fischlin (Fischlin, 1986) opisuje vlogo simulacije v inženirskem izobraževanju. Omenja, da je glavna prednost simulacije v tem, da omogoča skoraj na vseh področjih zamenjati realni svet, kompleksne eksperimente in pilotne naprave z enostavnim in cenenim mikroračunalnikom, na katerem poteka simulacija. S takim modelom je potem možno delati brez tveganja. Zaradi izredne ilustrativnosti je pristop primeren za začetnike, ki o realnem objektu ne vedo dosti, pa tudi za izkušenejše uporabnike.

Schmid (Schmid, 1988) omenja, da se simulacija uporablja prav v vsaki projektni fazi. Zato mora imeti vsak kvaliteten paket za CACSD vgrajen sposoben splošno

namenski simulator. Le-ta mora omogočati simulacijo zveznih in diskretnih sistemov.

Tudi Annino (Annino, 1979) ugotavlja, da je simulacija učinkovita metoda za preizkušanje modelov, pred razvojem in gradnjo dragih prototipov in implementacijo. V primerjavi z metodami analize je simulacija bolj realistična in laže razumljiva toda le, če je pravilno uporabljana. Izkušnje kažejo, da zaradi nepravilne uporabe večina simulacijskih projektov v sedemdesetih letih v Ameriki ni dala zadovoljivih rezultatov.

Mc Leod (Mc Leod, 1987b) opozarja, da je simulacija ena temeljnih ved modernih področij. Nekateri se sploh ne zavedajo, da uporabljajo simulacijo, drugim pa se zdi ta veda preveč klasična, da bi jo priznali. Te trditve avtor dokazuje na nekaterih sodobnih področjih. Tako pravi, da emulacija pomeni, da se en računalnik vede kot drugi, torej ga simulira. Celotno področje iger je osnovano na modelih in eksperimentiranju z njimi. Na področju umetne inteligence se uporablja modeliranje nekaterih aspektov mentalnih sposobnosti. Če ta model uporabimo za eksperimente, je to simulacija. Roboti so nastali kot modeli nekaterih človekovih aktivnosti in jih torej simulirajo. Razen tega robotika vsebuje elemente umetne inteligence in je torej zelo ozko povezana s simulacijo. Podobno velja za področje ekspertnih sistemov. Ekspertni sistem emulira človeka eksperta v procesu analize in načrtovanja. Torej simulira določene človekove akcije. Tudi navidezna resničnost (virtual reality) je osnovana na simuliranih modelih.

Herget (Herget, 1988) je z anketo pokazal, da je simulacija osrednja operacija na področju računalniškega načrtovanja vodenja sistemov. V anketi je sodelovalo 63 posameznikov in 47 ustanov, ki se ukvarjajo s področjem CACSD v ZDA. Anketa je pokazala, da je najpogosteje uporabljeni CACSD programski paket simulacijski jezik ACSL. Med prvimi desetimi paketi je še nekaj pretežno simulacijskih produktov. Anketa je tudi pokazala, da uporabniki na splošno svoje pakete največkrat uporabljajo za simulacijo (indeks 85), sledi področje načrtovanja sistemov vodenja (indeks 74) in pa področje identifikacije sistemov (indeks 43). Pri tem je treba omeniti, da se lahko simulacija implicitno skriva tudi v drugem in tretjem omenjenem področju. Podobne rezultate je dala tudi analiza uporabe paketov CACSD na Japonskem (Araki, 1985) in na Nizozemskem (Van den Bosch, Van den Boom, 1985).

Heinrich (Heinrich, 1986) je naredil analizo, ki je pokazala, da je veliko število člankov na svetovno znanih konferencah iz področja vodenja (npr. IFAC) precej simulacijsko obarvanih, čeprav to niso simulacijsko orientirane konference in tudi nimajo simulacijskih sekcij. Pravi, da ni to nič presenetljivega, saj se vsaka

metoda lahko dokaže le na tri načine: z aplikacijo v realnem svetu, s simulacijo ali analitično "na papirju".

V naslednjih letih lahko pričakujemo, da bo simulacija postala še bolj razširjena. Superračunalniki se približujejo tudi po hitrosti analognim računalnikom. Še več si seveda obetamo od multiprocesorskih in paralelnoprocesorskih sistemov, ki bodo v povezavi z izredno sposobnimi grafičnimi zasloni ter potrebnimi perifernimi enotami predstavljali učinkovite simulacijske delovne postaje. Glavno prednost bodo te delovne postaje pokazale pri simulaciji v realnem času s priključenimi realnimi objekti (HIL).

1.3 Uporabnost simulacije in njene omejitve

Problematiko uporabe simulacije in njenih omejitev, je najbolj široko obdelal Neelamkavil (Neelamkavil, 1987). Simulacijo uporabljamo v naslednjih primerih:

- Realni sistem ne obstaja, pri tem pa je gradnja prototipov in eksperimentiranje drago in časovno zamudno. Včasih je tudi nemogoče graditi prototipe (npr. jedrski reaktor).
- Eksperimentiranje z realnim sistemom je drago, nevarno, dolgotrajno, zahteva kompleksno opremo in lahko povzroči resne motnje v delovanju (npr. transportni sistem, jedrski reaktor).
- Kaže se potreba po študiju obnašanja sistema v preteklosti, sedanjosti ali prihodnosti v realnem, skrčenem ali raztegnjenem času (npr. sistemi vodenja, naraščanje prebivalstva).
- Matematični modeli nimajo enostavne in praktično uporabne analitične rešitve (npr. nelinearne diferencialne enačbe, ki opisujejo kaotične sisteme).
- Možno je zadovoljivo ovrednotiti simulacijski model glede na podatke o realnem objektu - vrednotenje modela.
- V inženirskem izobraževanju predstavlja simulacija najbolj nazorno in hkrati tudi ceneno metodo, ki omogoča delo brez tveganja tako pri začetnikih kot pri izkušenih uporabnikih.

Seveda ima simulacija tudi nekatere slabosti, ki omejujejo njeno uporabo. Te so:

- Digitalna simulacija je relativno počasna, iterativna tehnika in zato računalniško potratna.
- Včasih nas vodi do suboptimalnih rešitev, če ne uporabljamo optimizacijskih postopkov.
- Brez dobrega poznavanja realnega procesa ali objekta je včasih težko ovrednotiti rezultate simulacijskih modelov.
- Analiza in interpretacija rezultatov zahteva v nekaterih primerih tudi dobro poznavanje teorije verjetnostnega računa in statistike.
- Rezultate lahko napačno interpretiramo.
- Včasih je težko odkriti, odkod napaka izvira.

1.4 Modeliranje in simulacija kot nerazdružljivi metodi

Vsak realni objekt, za katerega želimo načrtati vodenje, začnemo proučevati s pomočjo podatkov, ki so nam na voljo in s pomočjo eksperimentov, ki jih je možno izvajati na objektu. Na ta način si ustvarimo bazo podatkov o sistemu. S pomočjo analize podatkov o sistemu zgradimo t.i. konceptualni - matematični model (Schmidt, 1986). Le-ta predstavlja sliko realnega objekta po neki človekovi zamisli. Zgrajen mora biti tako, da se vede kot realni objekt, vendar le za tiste namene, za katere služi in v okviru določenih toleranc. Zato moramo pri gradnji konceptualnega (matematičnega) modela:

- jasno opredeliti namen modeliranja,
- definirati omejitve, znotraj katerih deluje model,
- izbrati lastnosti (atribute), ki jih bomo upoštevali in zanemariti nepomembne ter idealizirati določene realne zakonitosti,
- definirati strukturo in določiti parametre modela, t.j. definirati moramo povezave posameznih atributov v sistem ter opisati dinamiko posameznih atributov z diferencialnimi in diferenčnimi enačbami. Diferencialne enačbe dobimo ob upoštevanju masnega ali energijskega ravnotežja oz. ob upoštevanju zakonov o ohranitvi mase, energije in gibalne količine.

Ko zgradimo konceptualni (matematični) model, moramo zbrati čim več informacij o njegovem vedenju. Samo v enostavnih primerih lahko izpeljemo analitično rešitev konceptualnega modela. V splošnem pa uporabljamo pristop, da iz konceptualnega modela zgradimo t.i. realni ali simulacijski model, ki služi nato za preizkušanje realnega sistema. Pomembno je torej, da imata realni sistem in realni model enak konceptualni (matematični) model. Izgradnja realnega modela je prvi korak pri simulaciji. Drugi korak pa je eksperimentiranje s tem modelom. S pomočjo dedukcije (analitične obravnave) konceptualnega modela in eksperimentiranja z realnim ali simulacijskim modelom dobimo podatke o modelu. S pomočjo teh podatkov izvedemo vrednotenje modela, t.j. analizo ujemanja obnašanja realnega sistema in konceptualnega modela. Pri tem moramo poudariti, da vrednotenje modela ni isto kot verifikacija. Verifikacija je analiza ujemanja realnega in konceptualnega modela. Pri računalniški simulaciji to pomeni, da preverjamo, če je bil simulacijski program pravilno izveden.

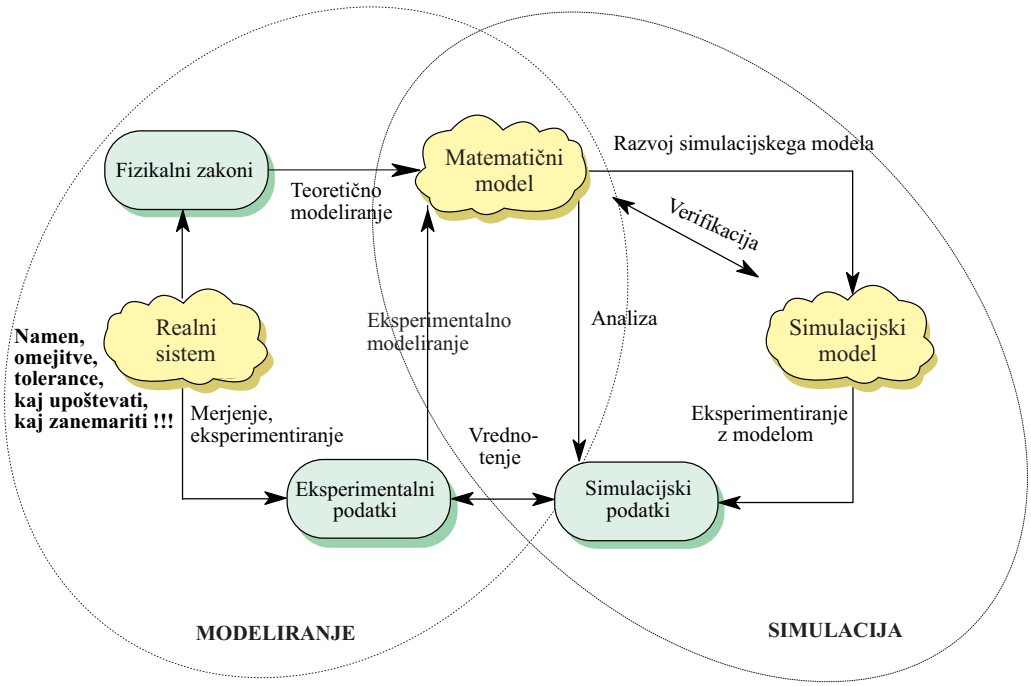
Analizo sistema, konstrukcijo modela, eksperimentiranje z modelom, vrednotenje in verifikacijo je običajno potrebno večkrat ponoviti, dokler ne pridemo do zahtevanih rezultatov. Zato Neelamkavil (Neelamkavil, 1987) simulacijo v širšem smislu definira kot počasno, iterativno in eksperimentalno orientirano tehniko.

Iterativni postopek modeliranja in simulacije prikazuje slika 1.1. Znano je, da za vsak mehanski ali hidravlični sistem lahko poiščemo električni ekvivalent. To pomeni, da so sistemi izrazljivi z enakimi diferencialnimi enačbami, torej imajo isti matematični model. V praksi je lahko eden izmed njih realni sistem, drugi pa realni model, ali obratno. Seveda za realne modele izberemo vedno take, s katerimi je možno enostavno eksperimentirati.

Čeprav sta modeliranje in simulacija tako neposredno povezani metodi, je namen tega učbenika pokriti zgolj simulacijski del. Da pa bo povezava vendarle dovolj poudarjena, bomo podali tri primere, v katerih bo poudarek na modeliranju. Te modele bomo uporabljali tudi v naslednjih poglavjih za prikaz simulacijskih metod in orodij.

Primer 1.1 Modeliranje avtomobilskega vzmetenja

Modeliranje poenostavljenega avtomobilskega vzmetenja predstavlja primer teoretičnega modeliranja na osnovi ravnotežja gibalnih količin. Običajno je model sistem dveh linearnih diferencialnih enačb drugega reda, ki opisujeta premike karoserije avtomobila in premike kolesa. Predpostavljamo torej model s koncentriranimi parametri. Pri tem upoštevamo vzmeti in dušilnike avtomobila ter



Slika 1.1: Iterativni postopek modeliranja in simulacije

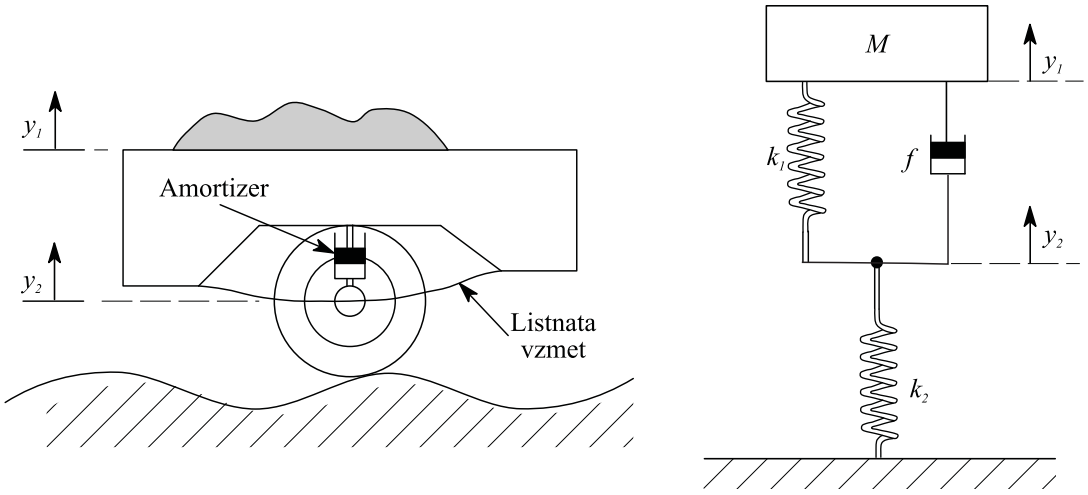
elastičnost pnevmatike. Oblika podlage predstavlja vzbujanje sistema. Model lahko služi kot dober pripomoček za spoznavanje modeliranega procesa, za študij vpliva posameznih elementov na obnašanje sistema, pa tudi za grobo dimenzioniranje elementov.

V našem primeru pa bomo uvedli še eno predpostavko. Ker smo zainteresirani le za obliko gibanja karoserije, bomo zanemarili še maso kolesa in obese, ki je majhna v primeri s četrtino mase avtomobila. Tako lahko obravnavani sistem prikažemo kot kombinacijo translatornih mehanskih elementov na sliki 1.2.

Na sliki 1.2 predstavlja M četrtino mase avtomobila, k_1 je konstanta togosti vzmeti avtomobila, f je konstanta dušenja dušilnika avtomobila, k_2 je konstanta togosti pnevmatike, y_1 je premik karoserije, y_2 pa premik kolesa iz mirovne lege.

Osnovno relacijo za začetek teoretičnega modeliranja predstavlja *drugi Newtonov zakon*, saj predstavlja eno od oblik zakona o ravnotežju gibalnih količin

$$\Sigma F = Ma = M\ddot{y} \tag{1.1}$$



Slika 1.2: Poenostavljen prikaz vzmetenja avtomobila

kjer je M masa [kg], a je pospešek [ms^{-2}] in F je sila [N]. Glede na to enačbo moramo najti ustrezne relacije še za ostale sile v sistemu (njihove povezave s premikom, ki nas zanimajo). Za silo vzmeti je na razpolago Hook-ov zakon, ki podaja silo, ki je potrebna, da raztegnemo (ali pa stisnemo) vzmet za razdaljo y glede na njeno osnovno dolžino

$$F_s = ky \tag{1.2}$$

pri čemer je k konstanta vzmeti [Nm^{-1}].

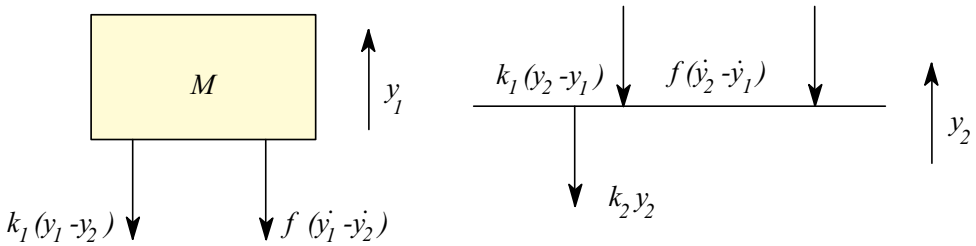
Silo, ki je potrebna, da premaknemo en konec viskoznega dušilnika s hitrostjo v relativno glede na drugi konec, podaja zveza

$$F_D = fv = f\dot{y} \tag{1.3}$$

kjer je f dušilni koeficient [Nsm^{-1}].

Tako se moramo odločiti le še o predznaku sil, kar najlaže naredimo s pomočjo diagrama sil. Pri tem se dogovorimo, da so sile v smeri predpostavljenega premika (največkrat ga izberemo v smeri sile, ki vzbuja sistem) pozitivne. Zavedati pa se

moramo, da se sile na vzmeteh in dušilnikih vedno upirajo kakršnemkoli premiku. Prav tako je jasno, da premiki v mehanskem vezju niso vezani le na mase, ki so vključene vanj, temveč se, kot v našem primeru, lahko pojavijo tudi ob zaporednih vezavah elementov, ali pa če vzbujevalna sila deluje na vzmet ali na dušilnik, ne pa na maso. Kadar je vsak konec elementa podvržen različnima premikoma (ni vezan na mirujočo točko), se namesto premika y oz. odvoda $\dot{y}$ v zvezah pojavi razlika premikov oz. odvodov (v tej razliki ima prva spremenljivka isti indeks, kot ga ima premik točke, v kateri delujejo obravnavane sile). Glede na povedano lahko za naš primer narišemo diagram sil na sliki 1.3.



Slika 1.3: Diagram sil za sistem avtomobilskega vzmetenja

Iz diagrama sil na sliki 1.3 lahko direktno napišemo enačbi za oba premika tako, da sile s predznakom, kot ga narekuje diagram sil, vpišemo na desno stran enačbe (1.1)

$$M\ddot{y}_1 = -f(\dot{y}_1 - \dot{y}_2) - k_1(y_1 - y_2) \tag{1.4}$$
$$0 = -f(\dot{y}_2 - \dot{y}_1) - k_1(y_2 - y_1) - k_2 y_2 \tag{1.5}$$

Enačbi (1.4) in (1.5) že predstavljata matematični model našega procesa. Ker pa nas zanima le y_1, eliminirajmo y_2. Vsota enačb (1.4) in (1.5) daje

$$M\ddot{y}_1 = -k_2 y_2 \tag{1.6}$$

iz česar izrazimo y_2 kot

$$y_2 = -\frac{M}{k_2}\ddot{y}_1 \tag{1.7}$$

Če y_2 sedaj vnesemo v enačbo (1.4), dobimo

$$\frac{fM}{k_2}\dddot{y}_1 + M(1+\frac{k_1}{k_2})\ddot{y}_1 + f\dot{y}_1 + k_1 y_1 = 0 \tag{1.8}$$

in končno

$$\dddot{y}_1 + \frac{k_1+k_2}{f}\ddot{y}_1 + \frac{k_2}{M}\dot{y}_1 + \frac{k_1 k_2}{Mf}y_1 = 0 \tag{1.9}$$

Dobili smo torej navadno linearno diferencialno enačbo tretjega reda s konstantnimi koeficienti, ki opisuje gibanje karoserije avtomobila.

Do sedaj nismo rekli še ničesar o vzbujanju sistema. Kot vemo, je sistem lahko vzbujan z vhodnim signalom, ali pa ima v trenutku, ko ga začnemo opazovati, določeno začetno stanje. Zunanji vhod bi lahko predstavljala podlaga po kateri vozi avto. Kakšna ovira ali jama na podlagi bi vzbudila sistem, pri čemer pa ne smemo pozabiti, da je glede na predpostavko naše kolo infinitezimalno majhno. Izberimo začetno stanje

$$y_1(0) = -y_{10} \tag{1.10}$$

Začetno stanje si lahko predstavljamo tako, da voznik vstopi v avtomobil, v trenutku $t = 0$ pa izstopi.

Predpostavimo naslednje konkretne podatke avtomobilskega vzmetenja:

M=500 kg, k_1=7500 N/m, k_2=150000 N/m, f=2250 Ns/m in y_{10}=0.05 m.

Tako lahko enačbo (1.9) zapišemo v splošni obliki

$$\dddot{y}_1 + a\ddot{y}_1 + b\dot{y}_1 + cy_1 = 0 \tag{1.11}$$

pri čemer so

$a = 70$, $b = 300$, $c = 1000$ in $y_1(0) = -y_{10} = -0.05$

□

Primer 1.2 Modeliranje regulacije gretja prostora

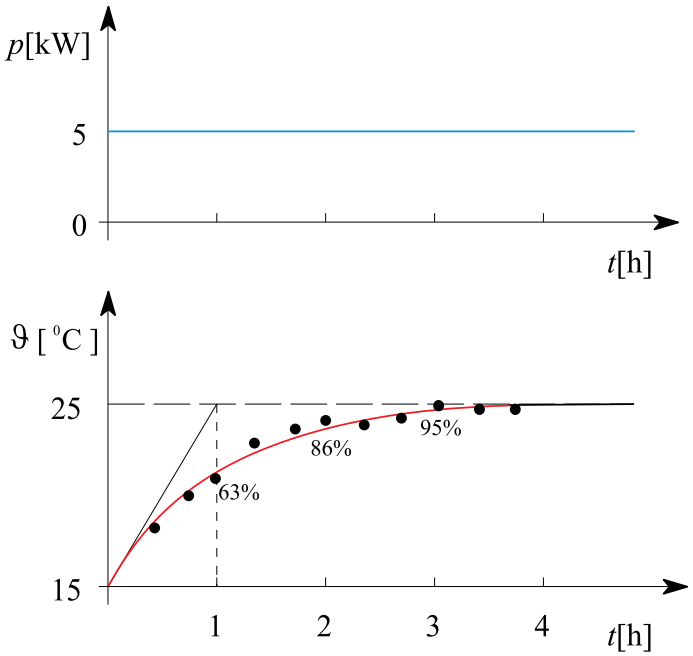
Oglejmo si še možnost "intuitivnega" pristopa k modeliranju gretja prostora z električnim grelom, ki ga prižiga ali ugaša termostatski regulator. To je klasična in cenena možnost regulacije temperature v prostoru, ki jo v vsakdanjem življenju največkrat uporabljamo. Učinki regulacije zavisijo v glavnem od treh faktorjev:

- moči grelnega telesa,
- histereze termostatskega regulatorja in
- zakasnitev, ki so prisotne v procesu.

Ti faktorji določajo nihanja temperature okrog želene vrednosti. Ker želimo majhna nihanja, bi morala biti tudi histereza termostata majhna, kar pa po drugi strani povzroča pogosto prižiganje in ugašanje grela in zopet predstavlja nezaželen pojav. Seveda pa je za čas, v katerem dosežemo želeno temperaturo, odločilna moč grela. Načrtovanje sistema gretja (dimenzioniranje gretja in izbira ter postavitev termostata) bi lahko potekalo v samem prostoru, vendar bi potrebne meritve zahtevale ogromno časa, saj so časovne konstante procesa v razredu ur. Pristop z natančno matematično analizo pa je tudi problematičen saj histereza, ki predstavlja nelinearnost, že v zelo enostavni povratnozančni strukturi in pri do skrajnosti poenostavljenem modelu procesa povzroči kompliciranо dinamično obnašanje obravnavanega sistema. Zato je pristop z modeliranjem in simulacijo najustreznejši. Ko se odločimo za enostaven model gretja prostora, lahko namreč celotno regulacijsko zanko simuliramo s pomočjo primernega simulacijskega orodja in tako mnogo hitreje kot pa na realnem sistemu s poskušanjem dobimo potrebne informacije o zaprtozančnem obnašanju sistema.

Omenjena dejstva so pomenila osnovne podatke o sistemu in namenu modeliranja. Glede na to skušajmo priti do modela gretja prostora z grelom na najenostavnejši "intuitivni"način, pri čemer pa mora dobljeni model vseeno zadovoljiti naše zahteve.

Zato v prostoru pri relativno konstantni temperaturi (npr. 15^oC) prižgemo grelo moči 5KW ($t = 0$, $p = 5k$W) in v določenih časovnih intervalih (npr. 20 min) merimo temperaturo. Tako dobimo rezultate na sliki 1.4.



Slika 1.4: Gretje prostora ob vklopu grela (p...moč gretja, ϑ...temperatura v prostoru). Pike pomenijo merjene vrednosti, krivulja pa odziv modela.

Kot vidimo na sliki 1.4, je proces proporcionalen, kar smo tudi pričakovali, saj višanje temperature povzroča tudi večanje izgub toplote (skozi zidove, vrata, okna itd.). Izgube so namreč proporcionalne razliki med temperaturama prostora in okolice ($\vartheta - \vartheta_e$). Tako pridemo do ustaljenega stanja pri približno 25^oC (pri tej temperaturi je toplota, ki jo proizvaja grelo, enaka izgubam). Glede na zakon o energijskem ravnotežju lahko predpostavimo, da je odvod (sprememba) temperature proporcionalen razliki gretja in izgub, kar lahko zapišemo v obliki naslednje diferencialne enačbe:

$$k_1\dot{\vartheta} = k_2 p - k_3(\vartheta - \vartheta_e) \tag{1.12}$$

Če enačbo (1.2) delimo s konstanto k_1, namesto preostalih konstant pa vpeljemo časovno konstanto T in ojačenje k, dobimo diferencialno enačbo prvega reda

$$\dot{\vartheta} + \frac{1}{T}(\vartheta - \vartheta_e) = \frac{k}{T}p \tag{1.13}$$

kjer je pomen oznak naslednji:

ϑ - temperatura v prostoru [^{o}C],
ϑ_e - temperatura okolice [^{o}C], ki je približno konstantna in znaša 15^{o}C,
p - moč grela [kW] v našem primeru 5 kW,
k - ojačenje sistema prvega reda,
T - časovna konstanta sistema prvega reda.

Tako dobljeni linearni model dobro opisuje realni proces za temperaturni interval $15^o\text{C} \leq \vartheta \leq 25^o\text{C}$ pri konstantni temperaturi okolice. Običajno pa uporabljamo tako imenovane deviacijske modele, ki podajajo razmere relativno na delovno točko (v našem primeru je to temperatura okolice). To storimo s pomočjo spremenljivke ϑ_w, ki jo definiramo kot razliko

$$\vartheta_w = \vartheta - \vartheta_e \tag{1.14}$$

kar daje končno obliko modela ($\dot{\vartheta} = \dot{\vartheta}_w$)

$$\dot{\vartheta}_w + \frac{1}{T}\vartheta_w = \frac{k}{T}p \tag{1.15}$$

Model, ki ga podaja enačba (1.15) lahko zapišemo še v obliki prenosne funkcije

$$\frac{\theta_w(s)}{P(s)} = \frac{k}{Ts+1} \tag{1.16}$$

Postavitev enačbe (1.15) predstavlja teoretični del modeliranja. Konstanti k in T pa skušajmo določiti iz merjenih rezultatov, kar pomeni neko vrsto eksperimentalnega modeliranja. To kaže, da smo pri tem problemu uporabili pravzaprav kombinirano modeliranje. Za naš sistem prvega reda lahko časovno konstanto določimo s pomočjo tangente na krivuljo ob času t=0 (kot kaže slika 1.4). Kjer tangenta seka premico ustaljenega stanja, odčitamo časovno konstanto T. Le-ta je v našem primeru ocenjena na

$$T = 1\text{h}$$

Ojačenje sistema pa določa razmerje med spremembo temperature v ustaljenem stanju in spremembo vhodnega signala (v našem primeru gretja)

$$k = \frac{\Delta\vartheta}{\Delta p} = \frac{25-15}{5-0} = 2^o\mathrm{C/kW} \tag{1.17}$$

Če dobljene vrednosti vstavimo v enačbo (1.15), dobimo model ogrevanja prostora

$$\dot{\vartheta}_w + \vartheta_w = 2p \tag{1.18}$$

oz. v obliki prenosne funkcije

$$\frac{\theta_w(s)}{P(s)} = \frac{2}{s+1} \tag{1.19}$$

V praksi pa se izkaže, da bi lahko ta model uporabili le v primeru, ko bi bil termostat, ki zaznava temperaturo prostora, zelo blizu grelnega telesa. Če temu ni tako, bi problem bolje opisovala diferencialna enačba višjega reda, ki jo lahko zadovoljivo aproksimiramo z modelom prvega reda v kombinaciji z mrtvim časom. V tem primeru je potrebno zapis (1.15) spremeniti v zapis z dvema enačbama

$$\begin{aligned} \dot{\vartheta}_w + \frac{1}{T}\vartheta_w &= \frac{k}{T}p_d \\ p_d(t) &= p(t-T_d) \end{aligned} \tag{1.20}$$

kjer je T_d mrtvi čas. Prenosna funkcija takega sistema je

$$\frac{\theta_w(s)}{P(s)} = \frac{k}{Ts+1}e^{-T_d s} \tag{1.21}$$

Zapis s pomočjo prenosne funkcije ponuja zelo ugodno in ilustrativno možnost predstavitve sistema s pomočjo bločnih diagramov. Pri študiju regulacijskih problemov (kot je to v našem primeru) lahko tako jasno prikažemo celotno strukturo regulacijske zanke.

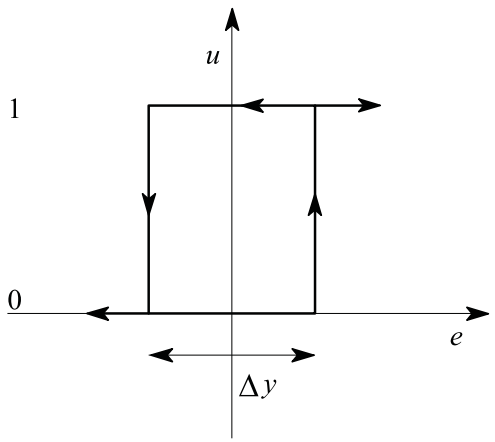
Zgradimo bločni diagram regulacije gretja prostora. Želeno temperaturo prostora (ϑ_r) nastavimo na termostatskem regulatorju. Vhod v slednjega je tudi temperatura v prostoru ϑ. Termostatski regulator ustvari razliko

$$e = \vartheta_r - \vartheta \tag{1.22}$$

iz katere preko histerezne karakteristike

$$u(t) = \begin{cases} 0, & e < -\frac{\Delta y}{2} \\ 1, & e > \frac{\Delta y}{2} \\ u(t-\Delta t), & -\frac{\Delta y}{2} \leq e \leq \frac{\Delta y}{2} \end{cases} \tag{1.23}$$

določi regulirni signal. Ustrezno zakonitost prikazuje slika 1.5. Pri tem je $u(t-\Delta t)$ pretekla vrednost regulirnega signala (seveda 0 ali 1, pri tem je Δt poljubno majhen).



Slika 1.5: Histereza termostata

Regulirni signal u vklaplja in izklaplja grelec (npr. preko releja, kontaktorja). Letega lahko modeliramo kar z nekim konstantnim ojačenjem p_{max}, oz. z enačbo

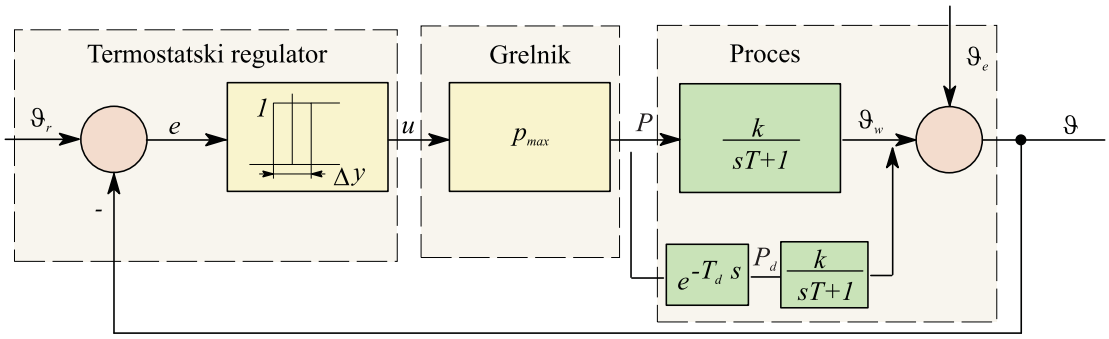
$$p = p_{max} u \tag{1.24}$$

Dobljeni signal p predstavlja vhod v model procesa (enačba (1.15) za $T_d = 0$ in enačbi (1.20) za $T_d \neq 0$). Ker smo razvili deviacijski model, s simulacijo pa bi radi

opazovali absolutne iznose spremenljivk, dobimo absolutni iznos temperature z vsoto temperature v okolici delovne točke (ϑ_w) in temperature delovne točke, ki je kar temperatura okolice (ϑ_e)

$$\vartheta = \vartheta_w + \vartheta_e \tag{1.25}$$

Bločni diagram regulacijske zanke pri gretju prostora tako prikazuje slika 1.6. V gradniku za proces sta navedeni obe možnosti: model prvega reda in model prvega reda z mrtvim časom.



Slika 1.6: Bločni diagram regulacije temperature prostora

Tako je problem pripravljen za nadaljnjo obravnavo. Naštejmo nekaj možnosti za študij različnih aspektov našega problema s pomočjo razvitega modela:

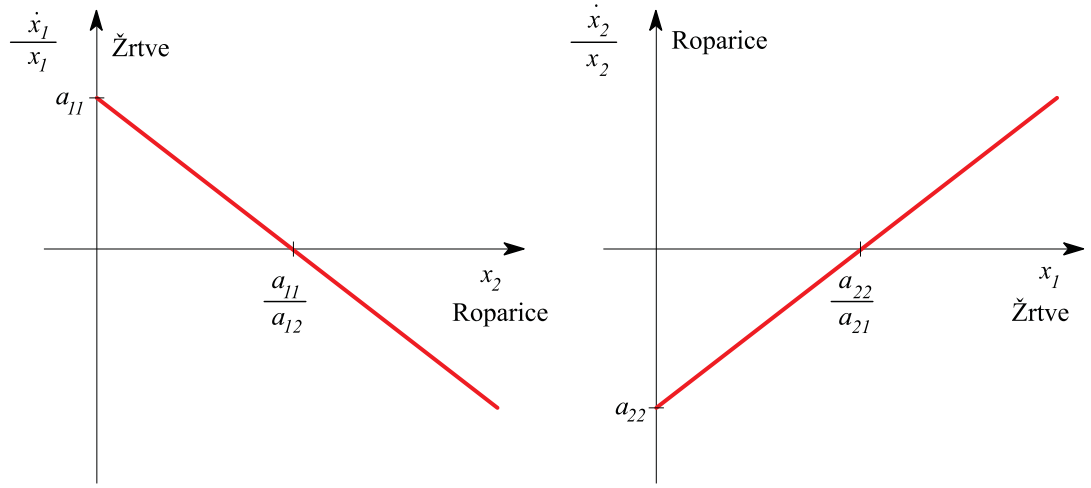
- določitev širine histereze termostata, ki daje majhna nihanja temperature ob ne preveč pogostem preklapljanju grela,
- določitev minimalne moči grela, ki še zagotavlja zadovoljivo regulacijo (sprejemljivo hitrost prilagajanja temperature novi želeni vrednosti, neobčutljivost na motnje itd.),
- študij vpliva motenj na regulirano temperaturo (možno je enostavno simulirati različne motnje na vhodu in izhodu procesa ali pa tudi na kakšnem drugem mestu),
- določitev optimalnega poteka želene temperature, ki obenem s primernim temperaturnim profilom v prostoru zagotavlja optimalno izrabo energije,
- študij vpliva lege termostata na temperaturo prostora (kar je posebno pomembno pri slabem mešanju zraka v prostoru),

- študij vpliva različnih časovnih konstant gretja in hlajenja prostora na temperaturo v prostoru itd.

□

Primer 1.3 Primer modeliranja populacijske dinamike

Oglejmo si še primer modeliranja enostavnega netehniškega problema. Pri populacijski dinamiki gre za študij rasti ali upadanja različnih populacij. Ta problem je dobro znan iz literature in če ga razširimo na obravnavo dveh populacij, ga imenujemo *model ekosistema Lhotka-Volterra*, še večkrat pa *problem roparic in žrtev*. Če jemljemo populaciji kot zvezni spremenljivki, je logaritmična rast ene odvisna le od druge populacije. Logaritmična rast populacije roparic je sorazmerna populaciji žrtev in logaritmična rast populacije žrtev je sorazmerna populaciji roparic. Ob predpostavki, da je sorazmernost linearna, dobimo odvisnosti, kot jih prikazuje slika 1.7. Pri tem smo označili populacijo žrtev z x_1, populacijo roparic z x_2, $a_{11}, a_{12}, a_{21}, a_{22}$ pa so konstante modela.



Slika 1.7: Logaritmični rasti populacij žrtev in roparjev

S pomočjo slike 1.7 lahko napišemo naslednji matematični model

$$\begin{aligned} \frac{\dot{x}_1}{x_1} &= a_{11} - a_{12}x_2 \\ \frac{\dot{x}_2}{x_2} &= a_{21}x_1 - a_{22} \end{aligned} \tag{1.26}$$

ali

$$\begin{aligned} \dot{x}_1 &= (a_{11} - a_{12}x_2)x_1 \\ \dot{x}_2 &= (a_{21}x_1 - a_{22})x_2 \end{aligned} \tag{1.27}$$

Pri tem zanemarimo faktorje, ki bi povzročali upadanje rasti neke populacije in bi izvirali v tej isti populaciji. Tako se pri odsotnosti žrtev populacija roparic zmanjšuje, pri odsotnosti roparic pa se populacija žrtev zvečuje. Pozitivni konstanti a_{11} in a_{22} zavisita od hitrosti rasti, pozitivni konstanti a_{12} in a_{21} pa predstavljata vzajemne faktorje zadrževanja rasti in sta proporcionalni velikosti druge populacije.

Če bi naša razmišljanja prenesli na zajce (žrtve) in lisice (roparice), lahko konstante v enačbah (1.27) ocenimo na teoretičen način s pomočjo naslednjih predpostavk:

1. Vsak par zajcev ima povprečno 10 mladičev letno.
2. Vsaka lisica poje povprečno 25 zajcev letno.
3. Povprečna starost lisic je 5 let, kar pomeni, da letno umre 20 % lisic.
4. V povprečju število mladih lisic, ki preživijo, zavisi od dosegljive hrane (število preživelih mladih lisic je torej enako številu zajcev deljeno s 25).

Za časovno enoto eno leto lahko torej izračunamo konstanti a_{11} in a_{22} iz predpostavk 1 in 3 na naslednji način:

$$\begin{aligned} a_{11} &= \left.\frac{\dot{x}_1\prime}{x_1}\right|_{a_{12}=0} \approx \frac{\Delta x_1\prime/\Delta t}{x_1} = \frac{10}{2} = 5 \\ a_{22} &= \left. -\frac{\dot{x}_2\prime}{x_2}\right|_{a_{21}=0} \approx \frac{\Delta x_2\prime/\Delta t}{x_2} = -\frac{-1}{5} = 0.2 \end{aligned} \tag{1.28}$$

Kot smo omenili, sta konstanti a_{12} in a_{21} odvisni tudi od področja na katerem opazujemo populaciji, kar pomeni, da sta odvisni od povprečnega števila zajcev in lisic. Če npr. opazujemo območje 50 km^2, kjer je povprečno število zajcev $\overline{x}_1 = 500$ in lisic $\overline{x}_2 = 100$, predpostavka 2 daje

$$a_{12} = -\left.\frac{\dot{x}_1''}{\overline{x}_1\overline{x}_2}\right|_{a_{11}=0} \approx \frac{\Delta x_1''/\Delta t}{\overline{x}_1\overline{x}_2} = -\frac{25\overline{x}_2}{\overline{x}_1\overline{x}_2} = \frac{25}{500} = 0.05 \quad (1.29)$$

predpostavka 4 pa

$$a_{21} = \left.\frac{\dot{x}_2''}{\overline{x}_1\overline{x}_2}\right|_{a_{22}=0} \approx \frac{\Delta x_2''/\Delta t}{\overline{x}_1\overline{x}_2} = \frac{\overline{x}_1/25}{\overline{x}_1\overline{x}_2} = \frac{1}{25\overline{x}_2} = \frac{1}{2500} = 0.0004 \quad (1.30)$$

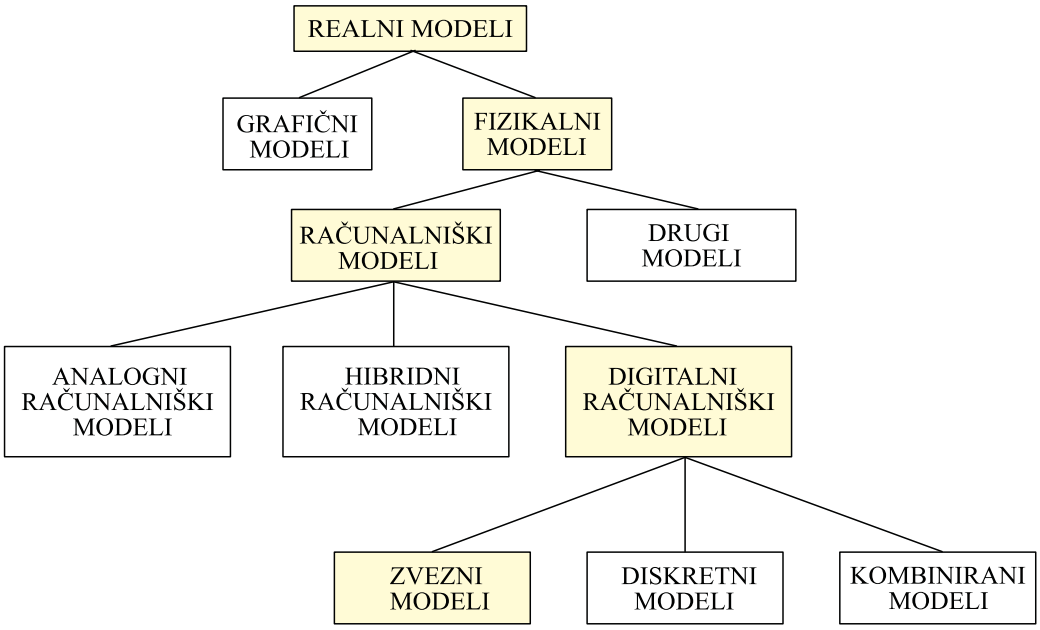
Tako smo prišli do matematičnega modela za obravnavani primer:

$$\begin{aligned} \dot{x}_1 &= 5x_1 - 0.05x_1x_2 \\ \dot{x}_2 &= 0.0004x_1x_2 - 0.2x_2 \end{aligned} \quad (1.31)$$

Model predstavlja sistem dveh navadnih nelinearnih diferencialnih enačb prvega reda s konstantnimi koeficienti. □

1.5 Razvrstitev realnih oz. simulacijskih modelov

Slika 1.7 prikazuje razvrstitev realnih oz. simulacijskih modelov. Pri grafičnih modelih elemente in funkcije konceptualnega modela predstavljamo v grafični obliki (npr. x-y krivulja, skica arhitekta, ...). Fizikalni modeli pa so opisani s fizikalnimi zakoni. Najbolj uporabna podmnožica fizikalnih modelov so računalniški modeli. Glede na to, s kakšnimi računalniki jih izvedemo, jih delimo na analogne, hibridne in digitalne računalniške modele. Analogni računalniški modeli so ugodni za simulacijo zveznih dinamičnih sistemov, odlikuje jih predvsem velika hitrost simulacije in so zato tudi zelo primerni za simulacijo v realnem času. Digitalni računalniški modeli so zlasti primerni za simulacijo diskretnih dogodkov in diskretnih sistemov (diskretni modeli), toda z uporabo numerične integracije se uspešno uporabljajo za simulacijo zveznih dinamičnih sistemov (zvezni modeli) in za simulacijo kombiniranih sistemov (kombinirani modeli). Odlikuje jih večja natančnost, široka možnost uporabe in dobre zmožnosti vhodno-izhodnih operacij. Hibridni računalniški modeli izkoriščajo dobre lastnosti analognih in digitalnih računalnikov.



Slika 1.8: Klasifikacija modelov

1.6 Načrtovanje sistemov vodenja, varnosti in zanesljivosti

Zaradi izrednega napredka znanja in tehnologije so računalniške obravnave za vodenje procesov, ki so še pred nekaj leti izgledale utopične, postale rutinsko delo. Načrtovanje sistemov vodenja pa ne more biti ločena naloga inženirja za načrtovanje vodenja, ampak je le del kompleksnega postopka, v katerem simulacija kot metoda za načrtovanje vodenja, varnosti in zanesljivosti igra pomembno vlogo. Simulacija se uporablja v naslednjih projektnih fazah (Tyso, 1985):

- za načrtovanje vodenja sistemov,
- za odkrivanje in spoznavanje napak,
- za varen zagon in ustavitev procesa,
- za vadbo operaterjev in
- kot orodje za pomoč operaterjem pri odločanju.

Uporaba simulacije za načrtovanje vodenja sistemov

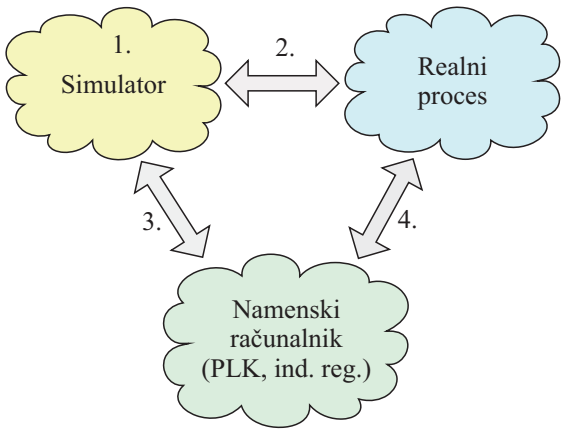
Simulacija se v povezavi z načrtovanjem vodenja procesov v glavnem uporablja za razvoj metode ter za razvoj konkretne rešitve vodenja in eksperimentiranje (Heinrich, 1986).

Klasični primer uporabe simulacije kot orodja za razvoj metode predstavlja razvoj postopkov za načrtovanja regulatorjev PID, ki so jih podali Chien, Hrones in Reswick ter Ziegler in Nichols v začetku petdesetih let. Te metode se še danes uporabljajo. Novejše metode vodenja so seveda bolj zahtevne, imajo pa tudi več omejitev, ki jih v praksi ponavadi težko izpolnimo. Zato je simulacija primerna za preverjanje uspešnega delovanja metode, za analizo stabilnosti, konvergentnosti in robustnosti. Razen tega je simulacija vgrajena tudi v številne druge metode za analizo in načrtovanje vodenja sistemov (npr. optimalni, adaptivni sistemi, načrtovanje multivariabilnih regulatorjev, ...).

Pri razvoju konkretnega vodenja in pri eksperimentiranju uporabljamo simulacijo, ker eksperimenti na realnem objektu običajno niso možni, saj so mnogokrat povezani z velikimi stroški ali tveganjem. Uspešnost preizkušanja konkretne rešitve s simulacijo seveda v največji meri zavisi od vernosti simulacijskega modela. Prav zato, ker je potrebno zgraditi kvaliteten model, se danes v industrijskem okolju relativno malo uporabljajo paketi CACSD, čeprav jih je veliko na trgu. Postavitev kvalitetnega modela zahteva veliko znanja, izkušenj, časa ter ustrezne instrumentacije, kar je povezano s stroški, ki pa so mnogo manjši od potrebnih vlaganj za gradnjo prototipov ali pilotnih obratov, kar predstavlja alternativno možnost modeliranju in simulaciji.

Slika 1.9 prikazuje, kako lahko na tri načine uporabljamo simulacijsko okolje pri načrtovanju konkretne rešitve. Redko se sicer uporabljajo vsi opisani načini, a pri kompleksnih sistemih vodenja je tak pristop smiseln:

1. Začnemo v simulacijskem okolju, kjer se nahaja shema vodenja in model.
2. V simulacijskem okolju obdržime shemo vodenja in jo preizkusimo na realnem procesu. Simulacijsko okolje mora delovati v realnem času.
3. V simulacijskem okolju obdržimo shemo modela, namenski računalnik vodi simulirani model. Simulacijsko okolje mora delovati v realnem času.
4. Šele ko predhodni eksperimenti dajo pričakovane rezultate, gremo v končno izvedbo, ko namenska oprema vodi realni proces.



Slika 1.9: Uporaba simulacije pri načrtovanju konkretne rešitve vodenja

Uporaba simulacije za odkrivanje in spoznavanje napak

Odkrivanje in spoznavanje napak (FDD - fault detection and diagnosis) predstavlja sodobno področje pri vodenju sistemov. S sprotnim spremljanjem dogajanja (parametrov) v sistemu vodenja je možno pravočasno odkriti oz. predvideti napako, preden povzroči padec kvalitete proizvodov ali celo prekinitev proizvodnje. To pa lahko pomeni precejšnje prihranke. Napredni sistemi uporabljajo simulacijski model, ki deluje paralelno z realnim objektom. Tako je možno delati primerjavo med realnimi meritvami in simuliranimi vrednostmi. Razen tega je možno s sodobnimi metodami izvesti tudi ocenjevanje stanj in parametrov, ki niso merljivi in napovedovanje določenih spremenljivk. Ob nenormalnih situacijah sistem za odkrivanje lahko sproži delovanje sistema za spoznavanje napak, ki na podlagi ugotovljenih dejstev pri sprotnem spremljanju ugotovi napako. Metode spoznavanja v zadnjem času temeljijo na pristopih umetne inteligence (npr. spoznavanje z nevronsko mrežo). Sistem za odkrivanje in spoznavanje napak pravočasno opozori operaterja, ki nato posreduje.

Uporaba simulacije za varen zagon in ustavitev procesa

Pri zagonu in ustavitvi procesa nastanejo prehodni pojavi, ki igrajo pomembno vlogo pri projektiranju sistema vodenja. S simulacijo zagona ali ustavitve lahko preizkušamo razne možne načine delovanja v nenormalnih pogojih. Preizkušamo lahko tudi zanesljivost in robustnost sistemov ob morebitnih napakah senzorjev in aktuatorjev ob upoštevanju njihove dinamike.

Uporaba simulacije za vadbo operaterjev

Dobro izurjeni operater je nujno potreben za nemoteno obratovanje vsakega realnega kompleksnega procesa, čeprav le-ta že vsebuje sistem vodenja. Računalniška simulacija omogoča hitro učenje operaterja. Na simuliranem sistemu lahko operater v enem dnevu doživi več težkih pogojev, v katerih mora poseči v proces, kot kasneje morda v več letih na realnem objektu. Pomembno pa je, da je simulator čim bolj verna reprodukcija dejanskega objekta z vsemi detajli, sicer bi lahko bila izurjenost operaterja le navidezna.

Uporaba simulacije za pomoč operaterju pri odločanju

Čeprav je vodenje v nekaterih primerih povsem avtomatizirano, lahko pride do situacij, ko mora operater ročno poseči v proces. Ročno vodenje pa je problematično pri procesih z velikimi zakasnitvami, pri fazno neminimalnih ter multivariabilnih sistemih ipd. V takih primerih je potrebno, da je paralelno k procesu priključen simulator, ki deluje hitreje kot v realnem času. Z njegovo pomočjo lahko operater preizkusi brez tveganja vse potrebne akcije.

1.7 Razlogi za morebitno neuspešnost simulacijskih projektov

Simulacija je sicer učinkovita metoda za preizkušanje modelov preden gremo v gradnjo dragih prototipov in njih dejansko implementacijo. V primerjavi z drugimi metodami analize in načrtovanja je simulacija bolj realistična in lažje razumljiva, toda le, če je pravilno uporabljena. Annino (Annino, 1979) omenja, da v sedemdesetih letih veliko izredno dragih simulacijskih projektov ni dalo želenih rezultatov. Omenili bomo nekaj glavnih razlogov za neuspešno uporabo simulacije.

Slabo definiran končni cilj

Cilj simulacijskega projekta ne sme biti nikoli model zaradi modela. Najbolj pomembno je natančno definirati, za kaj bodo služili rezultati simulacije. Zato

je skrbna postavitev končnih ciljev najpomembnejša naloga. Načrtovalci prav tu največkrat zagrešijo napako.

Slabo sestavljena ekipa

Simulacijski projekt lahko zadovoljivo opravi le ekipa, ki ima izkušnje iz različnih področij:

- vodenje projekta (sposobnost motiviranja, vodenja),
- modeliranje (zmožnost razviti konceptualni model, ki naj čim bolje opisuje sistem, toda le do ustreznega nivoja podrobnosti),
- programiranje (zmožnost pretvoriti konceptualni model v simulacijski model v obliki dobro čitljivega programa z možnostmi za enostavno spreminjanje),
- znanje o realnem sistemu (zmožnost vrednotenja rezultatov simulacije).

Najbolj problematična je prav zadnja točka, ki pomeni povezavo med načrtovalci in uporabniki rezultatov.

Neustrezni nivo podrobnosti

Model vedno predstavlja poenostavljen sistem, zato naj upošteva le tiste značilnosti, ki so pomembne za uporabnike rezultatov. Načrtovalci običajno nekatere dele sistema poznajo in razumejo bolj kot druge. Pri tem je nevarno, da bolj podrobno modelirajo tiste dele, ki jih bolje razumejo. Zato je potrebno ugotoviti podrobnosti, ki v večji meri vplivajo na rezultate in te vključiti v model. Torej je zelo pomembno v vsakem simulacijskem projektu definirati ustrezni nivo podrobnosti.

Slabo komuniciranje

Pomembno je dobro komuniciranje med člani moštva, saj morajo vsi sodelovati pri postavitvi končnih ciljev. Pri tem ima lahko velik učinek izbira ustreznega problemsko orientiranega simulacijskega jezika, ki omogoča dobro in enostavno

komuniciranje med člani ekipe tudi preko simulacijskega modela, hkrati pa seveda zelo olajša programiranje.

Izbira neustreznega računalniškega jezika

Nekateri menijo, da ima visok problemsko orientiran simulacijski jezik bistvene prednosti, drugi pa menijo, da je možno uporabiti splošno namenske jezike kot npr. Matlab (le kot programski jezik, brez simulacijskih dodatkov), Basic, Visual Basic, C++, Fortran, Java script, Java, ASP, Phyton, Visual Studio, itd. in s tem priti tudi do določenih prednosti. Izkušnje kažejo, da problemsko orientiran simulacijski jezik bistveno skrajša čas za razvoj modela in simulacijskega projekta, omogoča pa tudi večjo fleksibilnost in čitljivost.

Pomanjkljiva dokumentacija

Dokumentacija simulacijskega projekta je zelo pomembna, zahteva veliko časa in navadno povzroča precej težav. V končni fazi, ko je dokumentacija v glavnem že izdelana, še vedno prihaja do manjših ali večjih sprememb, včasih tudi konceptualnih. Ponavadi je laže popraviti program kot pa priročnike. Za same razvijalce je zelo pomembno tudi komentiranje v izvornih programih, ki ga lahko relativno enostavno in hitro sproti dopolnjujemo.

Uporaba nepreverjenega simulacijskega modela

Zelo pomembno je verificirati simulacijski model (program) s konceptualnim modelom. To najbolj učinkovito naredita načrtovalec simulacijskega modela in oseba, ki je sodelovala pri snovanju konceptualnega modela in dobro pozna realni objekt. Zato pa je izredno pomembno, da je izvorni program dobro čitljiv.

Napake zaradi neuporabe modernih programskih orodij za vodenje in načrtovanje velikih projektov

Vzrok za kasnitev pri mnogih velikih projektih je v tem, da razvijajo simulacijski model, še predno je dokončan konceptualni model ali ker je plan razvoja preveč

optimističen (ne predvidi se časa za nepredvidljive stvari, ki v določeni fazi zahtevajo veliko časa). Moderna programska orodja za vodenje in načrtovanje velikih projektov (orodja programskega inženirstva) lahko precej pomagajo pri premostitvi takih problemov in lahko zelo skrajšajo čas projekta (Steppard, 1983).

Neprimerna oblika rezultatov simulacije

Rezultati simulacije morajo biti v taki obliki, da jih uporabnik lahko na enostaven način poveže in primerja z realnim sistemom. V nasprotnem primeru uporabnik ne dobi zaupanja v model.

1.8 Zgodovinski razvoj simulacijskih orodij, metod in organiziranosti

Različni modeli, ki so ponazarjali realne objekte (npr. zgradbo atoma) so znani že iz prejšnjih stoletij. Šele z razvojem računalnikov pa lahko govorimo o sodobni simulaciji. Leta 1930 sta Howard Aiken iz Harvarda in George Stibitz iz Bell Telephone Laboratory razvila prvi električni relejski računalnik. Leta 1946 pa sta John Mauckley in Prisper Eckert iz Univerze v Pensylvaniji končala z razvojem sistema ENIAC (Electronic Numerical Integrator and Calculator), ki je imel namesto relejev elektronske cevi. To je bil prvi elektronski digitalni računalnik. Razvijalci so ustanovili računalniško podjetje, iz katerega je leta 1950 prišel na trg Eckert-Mauckleyev UNIVAC. Toda ti prvi računalniki so bili za simulacijo neprimerni zaradi skromnih računskih zmožnosti, majhne kapacitete pomnilnika in zaradi izredno slabih vhodno - izhodnih naprav. K sreči pa je za uporabnike tistega časa Georg A. Philbrick iz podjetja Foxboro že v letih 1937-38 razvil "An automatic control analyser", ki je bil prvi analogni računalnik za posebne namene. V letu 1943 so John R. Regazzini, Robert H. Randall in Frederick A. Russell s kolumbijske univerze razvili analogni računalnik, ki so ga imenovali "An electronic system for obtaining an engineering solution for integrodifferential equations of physical systems". Okoli leta 1945 je bila tudi že znana simulacija po metodi Monte Carlo. John von Neuman in Stanislav Ulam iz Laboratorija v Los Alamosu sta z njo reševala problem difuzije neutronov.

V tistih časih so bili analogni računalniki edino primerno orodje za simulacijo zveznih sistemov, digitalni računalniki pa so se uporabljali za simulacijo diskret-

nih dogodkov. Glavna odlika analognih računalnikov je bila velika hitrost, digitalne računalnike pa je odlikovala velika natančnost. V Evropi pomeni eno prvih simulacijskih orodij večjih sposobnosti analogni računalnik TRIDAC, ki je začel delovati leta 1955 v Angliji. Imel je elektronske, mehanične in hidravlične komponente.

Že zgodaj pa so se pokazale tudi slabosti obeh vrst računalnikov. Zato so se že v petdesetih letih pojavile ideje o združitvi analognega in digitalnega računalnika. Sredi petdesetih let pa se pojavijo prvi programi za digitalne računalnike, ki omogočajo reševanje diferencialnih enačb z numerično integracijo. Leta 1955 je Selfridge prvi podal idejo bločno orientiranega simulacijskega jezika, s pomočjo katerega bi probleme na digitalnem računalniku reševali podobno kot na analognem. Leta 1959 pa so Stein, Rose in Parker poročali o prevajalniškem konceptu digitalne simulacije.

Šestdeseta leta predstavljajo velik napredek na področju analognih in digitalnih računalnikov. Z njihovo pomočjo je bilo že možno izvajati kompleksne simulacije. Leta 1963 je Electronic Associetes (EAI) dal na trg hibridni računalnik HYDAC, t.j. prvi računalnik z digitalnim krmiljenjem in logiko. V sredini šestdesetih let so se pojavili prvi hibridni sistemi s povezavo splošno namenskega analognega in splošno namenskega digitalnega računalnika. Prav tako so takrat prišli na trg prvi sposobni simulacijski jeziki, ki so temeljili na jeziku FORTRAN. Leta 1965 je bil dokončan DSL 90, prvi IBM-ov bločno orientirani simulacijski jezik, ki je služil za razvoj kasnejših verzij jezikov CSMP. Pravi mejnik glede na sposobnosti pa predstavlja CSMP 360, ki je prišel na trg leta 1966. Že leta 1967 pa je Simulation Council (predhodnik današnje The Society for Computer Simulation) v reviji Simulation (Strauss, 1967) objavil standard, po katerem naj bi bili izdelani prihodnji jeziki za simulacijo zveznih dinamičnih sistemov. Standard se je uspešno obdržal vse do danes.

V sedemdesetih letih so se začeli digitalni računalniki mnogo bolj množično uporabljati kot analogni zaradi bistveno nižje cene ter vedno večje računske sposobnosti in interaktivnosti. Analogni in hibridni računalniki so se največ uporabljali v posebne namene (simulacija v realnem času, uporaba v visokotehnoloških letalskih, vesoljskih in vojaških projektih, zahtevne regulacije, jedrski reaktorji) in pa v pedagoške namene. Najbolj znani hibridni računalniki tega obdobja so računalniki firme Electronic Asociates (EAI 580, EAI 680, EAI 1000, EAI 2000). Sredi sedemdesetih let so pri ADI (Applied Dynamics International) ocenili, da hibridni sistemi nimajo prave prihodnosti v vedno bolj kompleksnem modeliranju, pri katerem se zahteva tudi velika natančnost. Odločili so se, da gredo v razvoj povsem digitalnega večprocesorskega simulacijsko

specializiranega procesorja AD-10. Tako je že konec sedemdesetih let prišel na trg izredno sposoben večprocesorski simulacijski sistem. Tudi digitalni simulacijski produkti so doživeli velik razvoj. Na ta razvoj je razen simulacijskega standarda vplival tudi razvoj numeričnih metod, matematičnih knjižnic in kasneje obsežnih paketov CACSD. Zato je bil pri načrtovanju jezikov večji poudarek na numerični robustnosti, fleksibilnosti in interaktivnosti. Najbolj znan produkt tega obdobja je simulacijski jezik CSSL.

V osemdesetih letih je prišlo do ekspanzije mikroračunalnikov in osebnih računalnikov. Ker so le - ti postajali vedno bolj zmogljivi, so začeli na njih prenašati simulacijske pakete, ki so do takrat delovali le na večjih računalnikih. Najbolj znana splošno namenska simulacijska jezika sta CSSL IV in ACSL. V tem obdobju se začenja kazati težnja po novem standardu CSSL in na podlagi priporočil dveh delovnih komisij (pri IMACS -International Association for Mathematics and Computers in Simulation in pri SCS - The Society for Computer Simulation) se sredi 80 let začnejo razvijati simulacijski jeziki nove generacije (ESL, SYSMOD, COSMOS). Simulacijska orodja so postala dostopna vsakomur in množična uporaba simulacije je močno vplivala na razvoj nekaterih teoretičnih področij. Prav tako je za to obdobje značilen velik razmah modernih paketov za analizo in načrtovanje vodenja sistemov, v katerih je vedno vključena tudi simulacija. Značilen za to obdobje je tudi razvoj superračunalnikov, vrstičnih procesorjev in paralelnih procesorjev. Nova materialna oprema je zelo vplivala na razvoj simulacijskih orodij. Pojavljajo se simulacijske delovne postaje z vrstičnimi procesorji, z barvnimi grafičnimi zasloni velike ločljivosti, z veliko stopnjo interaktivnosti ter z izrednimi zmožnostmi za delo v realnem času (po hitrosti se približujejo analognim računalnikom). Tudi izdelovalci hibridnih sistemov niso ostali na nivoju sedemdesetih let. Electronic Associates je dala leta 1983 na trg paralelnoprocesorski sistem SIMSTAR, ki konceptualno predstavlja povsem novo vrsto hibridnega sistema. Povezava računalnikovih elementov v program se izvrši avtomatsko, programira pa se s pomočjo simulacijskega jezika.

Devetdeseta leta so prinesla velik napredek predvsem pri razvoju uporabniških vmesnikov simulacijskih orodij. Tako na osebnih računalnikih kot na sposobnih delovnih postajah prevladujejo koncepti okenskih vmesnikov. Ena najpomembnejših lastnosti je opis modela na grafični način. Take vmesnike so razvili za starejše simulacijske produkte (npr. za ACSL), imajo pa jih seveda vsi na novo razviti produkti (npr. SIMULINK v okolju MATLAB, SYSTEM_BUILT v okolju MATRIXx, MODEL-C v okolju CONROL-C, paket CC, EASY5 - to so vse paketi za računalniško podprto načrtovanje vodenja sistemov, kjer pa je simulacija osrednja metoda). Nesluten razvoj je naredilo programsko okolje MATLAB tudi na področju simulacij zlasti s paketom SIMULINK. MATLAB-SIMULINK je postalo

standardno okolje na vseh akademskih institucijah, kasneje pa se je izdatno začelo uporabljati tudi v industriji. Kasneje so za SIMULINK razvili razne namensko uporabne dodatke, eno je npr. STATE FLOW, ki omogoča modeliranje dogodkovnih procesov. Velik je tudi poudarek na objektni orientiranosti (npr.Xmath). Nekatera orodja uvajajo tudi večjo podporo v smislu modeliranja (npr. DYMOLA z orodji DYMODRAW, DYMOVIEW, DYMOSIM).

V novem tisočletju je simulacija najbolj zaznamovana z objektno orientiranim in več domenskim jezikom Modelica. Podobno idejo objektne orientiranosti vsebujejo tudi Bond grafi - grafična modelerska tehnika, ki opisuje pretok energije med komponentami (podpira npr. simulacijski paket 20-sim). Modelica postaja priznani standard za modeliranje zveznih pa tudi diskretnih in hibridnih dinamičnih sistemov. Omogoča zlasti veliko podporo modeliranju, saj ni potrebno izraziti odvodov stanj, kot v primeru večine konvencionalnih simulacijskih orodij. Zlasti je pomemben način povezovanja komponent, ki omogoči gradnjo knjižnic ponovno uporabljivih komponent. Povezujemo pa lahko komponente različnih področij, kar je zlasti pomembno v mehatroniki, robotiki, v avtomobilski industriji, v vodenju sistemov ipd. Zlasti sposobni okolji, ki podpirata jezik Modelica, sta Dymola in MathModelica. Podoben način je uvedel tudi MathWorks l. 2008 z okoljem Simscape v sklopu programskega paketa Matlab-Simulink. Žal pa ta način ne spoštuje standarda Modelica.

Tabela 1.1 predstavlja pregled razvoja simulacijskih metod in orodij.

Tabela 1.1: Pregled razvoja simulacijskih metod in orodij

1930	prvi električni relejski računalnik - Howard Aiken (Harvard) in George Stibitz (Bell Telephone Laboratory)
1938	prvi analogni računalnik - "An automatic control analyser- Georg A. Philbrick (Foxboro)
1943	"An electronic system for obtaining an engineering solution for integrodifferential equations of physical systems- John R. Regazzini, Robert H. Randall in Frederick A. Russell (univerza Kolumbija)
1946	ENIAC (Electronic Numerical Integrator and Calculator) - John Mauckley in Prisper Eckert (Univerza v Pennsylvaniji)
1950	začetek razvoja splošnonamenskih analognih računalnikov
1955	SELFRIDGE - prvi simulacijski jezik
1960	prvi hibridni sistemi (EAI)
1965	DSL 90 - prvi prevajalniški enačbno orientirani jezik
1967	CSMP 360, CSMP III - prvi sposobni simulacijski jeziki, funkcijski generatorji, 60 operatorjev, reševanje algebrajske zanke
1967	**standard CSSL 67**
1968	MIMIC - prvi jezik po vzoru CSSL 67
1969	CSSL III - prvi sposobni jezik CSSL
1970	hibridni računalniki (EAI 580, EAI 680, EAI 2000)
1972	CSSL IV - v preteklosti eden najsposobnejših simulacijskih jezikov
1972	HYSIM - kombinirana simulacija, razvoj na takratni Fak. za el. in rač., UL
1975	SIMNON - sposoben interaktivni jezik, prevajanje direktno na strojni nivo, uporabniku ni potrebno integrirati odvodov
1975	ACSL - vsestransko dober, komercialno uspešen simulacijski jezik
1975	AD-10, AD-100 - simulacijsko specializirana digitalna procesorja (Applied Dynamics International)
1980	ekspanzija mikroračunalnikov, CACSD paketov, superračunalnikov, paralelnoprocesorskih sistemov
1983	SIMSTAR - hibridni večprocesorski sistem
1983	HYBSIS, STARTRAN - simulacijska jezika za hibridne sisteme, simulacija v realnem času
1984	ADSIM, PARSIM - simulacijska jezika za namenske simulacijske računalnike, simulacija v realnem času
1984	ESL, SYSMOD, COSMOS - simulacijski jeziki nove generacije
1988	Xanalog- simulacijska delovna postaja, grafifični vnos modela
1989	SIMCOS - zvezna in diskretna simulacija, eksperimentiranje, delovanje v realnem času, razvoj na takratni Fak. za el. in rač., UL
1990	MATRIXx, SYSTEM-Built, CONTROL-C, MODEL-C, EASY5 - kompleksni CACSD sistemi
1990	SIMULINK - grafični uporabniški vmesnik, delovanje v okolju MATLAB
1992	okolje DYMOLA - podpora za modeliranje, objektna orientiranost
1996	jezik MODELICA - poizkus standardizacije jezika za objektno orientirano več domensko modeliranje - nov standard po CSSL'67
2008	Simscape, rešitev podjetja Mathworks za več domensko objektno orientirano modeliranje

Hkrati z razvojem simulacijskih orodij in simulacijskih metod se je razvijala tudi simulacijska organiziranost, vendar predvsem v ZDA. Tam se je že okoli leta 1950 pokazalo, da različni laboratoriji zaradi slabe povezave delajo na enakih ali podobnih problemih. John Mc Leod iz Naval Air Missile Test Centra v Kaliforniji je skušal združiti laboratorije pod eno organizacijo, vendar ni uspel. Zato je leta 1952 ustanovil lastno zvezo, ki je bila predhodnica današnje "The Society for Modeling and Simulation International (SCS)- Simulation Council. Tej zvezi se je kmalu priključilo veliko laboratorijev. Začela je izdajati tudi revijo Newsletter, ki je pozneje prerasla v revijo na področju simulacije SIMULATION.

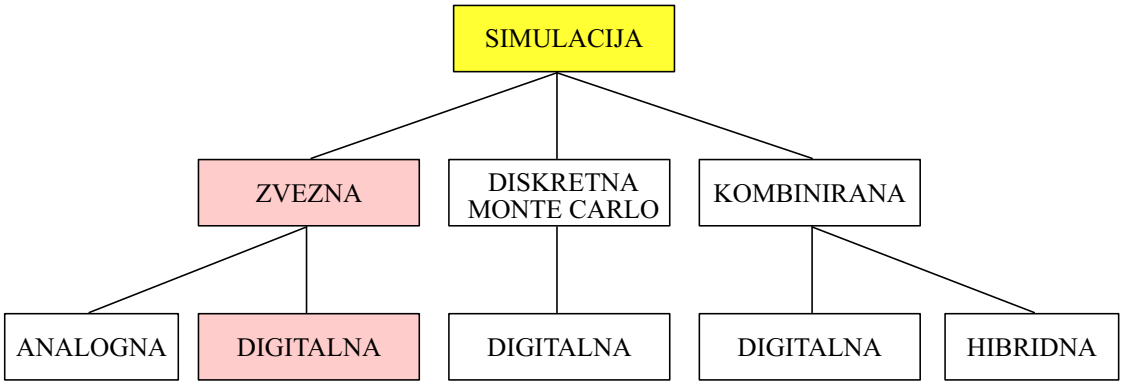
V Evropi je SCS šele leta 1985 odprla t.i. evropski urad v Ghentu. Leta 1989 pa se je ustanovilo evropsko združenje EUROSIM. Osnovna naloga je koordinacija dela, konferenc, simpozijev in drugih dejavnosti nacionalnih simulacijskih združenj: ASIM – Arbeitsgemeinschaft Simulation (Avstrija, Nemčija, Švica) CROSSIM – Croatian Society for Simulation Modelling, CSSS – Czech and Slovak Simulation Society, DBSS – Dutch Benelux Simulation Society (Belgija, Nizozemska), FRANCOSIM – Societe Francophone de Simulation (Belgija, Francija), HSS – Hungarian Simulation Society, ISCS – Italian Society for Computer Simulation, PSCS – Polish Society for Computer Simulation, SIMS – Simulation Society of Scandinavia (Danska, Finska, Norveška, Švedska), SLOSIM – Slovenian Society for Simulation and Modelling, UKSIM – United Kingdom Simulation Society (Velika Britanija, Irska), CEA SMSG Spanish Modelling and Simulation Group, LSS-Latvian Society for Simulation in ROMSIM - Romanian Society for Modelling and Simulation. Pod okriljem EUROSIMa izdaja založba Elsevier revijo Simulation Modelling Practice and Theory, zveza ASIM pa časopis/revijo Simulation News Europe.

Zveze prirejajo številne konference. V Evropi so najpomembnejši simulacijski kongresi EUROSIM (l. 2007 ga je v Ljubljani organiziralo slovensko društvo SLOSIM) v zvezi s simulacijo na področju vodenja pa so zlasti pomembne razne konference, ki jih organizira mednarodna zveza za avtomatsko vodenje IFAC. Prav tako so številne konference organizirane pod okriljem mednarodne elektrotehniške zveze IEEE.

2.

Vrste simulacije

Simulacijo kot metodologijo za analizo in načrtovanje sistemov delimo glede na vrste modelov na zvezno, diskretno, simulacijo po metodi Monte Carlo in kombinirano simulacijo. V odvisnosti od orodja oz. tehnike, s katero se izvaja (vrsta računalnika), pa jo delimo na analogno, digitalno in hibridno simulacijo. Glede na to razdelitev je zvezna simulacija lahko analogna ali digitalna, diskretna in simulacija po metodi Monte Carlo je vedno digitalna, kombinirana simulacija pa je digitalna ali hibridna. Vrste simulacije prikazuje slika 2.1.



Slika 2.1: Vrste simulacije

Hitrost izvrševanja simulacije z ozirom na realni čas, v katerem deluje realni sistem, določa, kako model simuliramo:

- počasneje kot v realnem času,

- v realnem času,
- hitreje kot v realnem času.

Simulacija, ki ne poteka v realnem času, je najbolj običajna in se lahko izvaja na splošnonamenskih računalnikih. Ali se izvaja hitreje ali počasneje kot v realnem času, je odvisno od časovnih konstant realnega sistema ter od sposobnosti simulacijskega orodja, s katerim želimo dani sistem ponavadi čim hitreje simulirati. Simulacija v realnem času pa uvaja precej novih problemov, saj si pod tem pojmom predstavljamo običajno tudi priključitev realnega sistema na računalnik (HIL -hardware in the loop). Učinkovita simulacija v realnem času je možna le s specifično programsko in materialno opremo (moderne simulacijske delovne postaje, analogno-hibridni računalniki).

2.1 Zvezna simulacija

Zvezna simulacija omogoča simulirati sisteme, ki jih lahko opišemo z linearnimi ali nelinearnimi navadnimi (ODE) ali parcialnimi (PDE) diferencialnimi enačbami s konstantnimi ali spremenljivimi koeficienti. Pri tem pa je pogoj, da so spremenljivke stanj in njeni odvodi zvezni preko celotnega simulacijskega teka, v katerem je neodvisna spremenljivka običajno čas. To področje predstavlja najstarejšo pa tudi najnaravnejšo obliko simulacije, ki se je sprva izvajala na analognih, kasneje pa tudi na digitalnih računalnikih. Osnovni princip pri reševanju z zvezno simulacijo je uporaba integracije, s pomočjo katere iz višjih odvodov spremenljivk stanj v modelu izračunamo spremenljivke stanj, le-te pa z uporabo ustreznih operacij povežemo v paralelni simulacijski model. Običajno delimo zvezno simulacijo glede na vrsto uporabljenega računalnika na

- analogno simulacijo in
- digitalno simulacijo.

Sistemi, ki jih simuliramo z zvezno simulacijo, so zvezni in paralelni. Zato analogni način simulacije predstavlja najbolj naravno obliko. Pri digitalni simulaciji pa je potrebno integrator zamenjati z diskretnim numeričnim algoritmom, paralelno strukturo modela pa reševati zaporedno. Tak postopek zahteva več računalniškega časa, vendar je možno s primerno numerično integracijsko metodo in ustreznim vrstnim algoritmom problem zadovoljivo rešiti.

2.2 Diskretna simulacija

Pri diskretni simulaciji se stanja sistema spreminjajo v diskretnih trenutkih. Tem spremembam pravimo diskretni dogodki. Le-ti se lahko izvajajo periodično v natančno določenih trenutkih (npr. regulirni signal diskretnega regulatorja) ali pa nesinhrono v odvisnosti od pogojev, ki jih določajo vrednosti spremenljivk stanj. Prva oblika je bolj običajna v teoriji avtomatskega vodenja. Metodologija te vrste diskretne simulacije ima veliko skupnega z zvezno simulacijo. Drugi obliki, ki je bolj značilna za diskretno simulacijo, pa navadno pravimo simulacija diskretnih dogodkov. Lastnosti, ki jih bomo omenili v nadaljevanju, so značilne predvsem za to vrsto simulacije.

Simulacija diskretnih dogodkov omogoča določanje stanj sistema v odvisnosti od časa, zbiranje podatkov in ustrezno statistično analizo. Statistika je ena temeljnih operacij pri tej vrsti diskretne simulacije, kajti modeli, ki jih na ta način simuliramo, so običajno stohastične (naključne) narave.

Klasični primer, ki ga največkrat obravnava literatura (npr. Neelamkavil, 1987), je poštni urad z enim strežnikom in čakajočo vrsto. Stranke prihajajo naključno v sistem in so postrežene po sistemu FIFO (first in - first out). Bistveni parametri, ki jih proučujemo z diskretno simulacijo, so:

- povprečni časi med prihodoma zaporednih strank,
- povprečni časi streženja strank,
- povprečno število streženj na časovno enoto,
- izkoriščenost strežnika,
- povprečna dolžina vrste,
- povprečno število strank v vrsti,
- povprečno število strank v sistemu,
- povprečni časi čakanja strank,
- povprečni časi zadrževanja strank v sistemu.

V splošnem je možno tak problem rešiti z metodo opazovanja realnega sistema, s teoretično metodo ali z diskretno simulacijo.

Z metodo opazovanja realnega sistema v dovolj dolgem časovnem intervalu zgradimo tabelo, v kateri so naslednji podatki: indeks stranke, čas prihoda, čas med dvema zaporednima prihodoma, čas začetka strežbe, čas konca strežbe, čas strežbe, čas čakanja v vrsti, čas stranke v sistemu. Razen tega sestavimo tabele kumulativnih časov (čas, ko ni v vrsti nobene stranke, ko je v vrsti ena stranka, dve stranki, ..., n_{max} strank, čas, ko ni v sistemu nobene stranke, ko je ena stranka, dve stranki, ..., n_{max} strank). S pomočjo teh tabel lahko enostavno izračunamo karakteristične parametre diskretnega sistema. Če so kateri od parametrov socialno ali ekonomsko nesprejemljivi, se mora zgoraj navedeni urad preurediti.

Pri teoretični metodi uporabimo za izračun statistike analitične postopke. Le-ti so dobro razviti le za sistem z enim strežnikom in eno čakajočo vrsto in veljajo ob določenih predpostavkah:

- neomejena dolžina vrste,
- neskončni vir strank,
- opazovanje preko dolgega časovnega intervala in
- eksponencialni verjetnostni porazdelitvi za čas med dvema prihodoma in za čas streženja ($f(t) = \lambda e^{-\lambda t}$).

Z metodo opazovanja izračunamo parametre porazdelitve (λ), vendar moramo upravičenost eksponencialne porazdelitve statistično preveriti (npr. s pomočjo Chi square testa). Nato uporabimo analitične enačbe za izračun parametrov diskretnega sistema.

S simulacijo na digitalnem računalniku lahko problem rešujemo, če sta poznani eksponencialni verjetnostni porazdelitvi za čas med dvema prihodoma in za čas streženja. S pomočjo znanih porazdelitev računalnik generira naključna števila, ki predstavljajo čase med dvema prihodoma in čase streženja. Računalniški program torej simulira prihode strank v sistem in delovanje znotraj sistema, vsebovati pa mora tudi pogoj za končanje simulacijskega teka. Na tak način lahko obravnavamo kompleksne sisteme, ki jih analitično ni možno rešiti.

Nekateri avtorji ne ločijo med simulacijo diskretnih dogodkov in t.i. simulacijo po metodi Monte Carlo, čeprav je med njima precejšnja razlika. Ustrezno poimenovanje je izvedel Von Neumann v 60. letih prejšnjega stoletja v okviru vojaškega projekta v Los Alamosu. V obeh primerih je simulacijski model vzbujan z naključnimi signali, vendar je sistem, ki ga modeliramo, v primeru simulacije

diskretnih dogodkov stohastičen, v primeru simulacije po metodi Monte Carlo pa ima deterministični značaj.

2.3 Simulacija po metodi Monte Carlo

Metodo Monte Carlo (Howell, 1993, Howell, 1998) lahko poenostavljeno opišemo kot statistično simulacijsko metodo, kjer pod statistično simulacijsko metodo smatramo vsako metodo, kjer uporabljamo naključna števila da izvedemo simulacijo. Metoda se ponavadi uporablja za obravnavo deterministčnih procesov.

Ime Monte Carlo je skoval Nicolas Metropolis, ko je skupaj s Stanislavom Ulamom delal na projektu Manhattan med drugo svetovno vojno zaradi podobnosti statistične simulacije z igro na srečo. Harris in Kahn sta l. 1948 sistematična in matematično začela obravnavati metodo. Istega leta so Fermi, Metropolis in Ulam z njo določili lastne vrednosti Schrodingejeve enačbe. L. 1964 je Howell J.R. metodo uporabil za obravnavo toplotnega sevanja. L. 1970 je nastala teorija računske kompleksnosti, ki je omogočila bolj racionalno odločanje za uporabo metode. S hitrim razvojem računalniške strojne opreme v 80 in 90 letih dvajsetega stoletja je metoda doživela pravo renesanso.

Metoda Monte Carlo se dandanes uporablja na mnogih področjih, kjer konvencionalne metode zaradi kompleksnosti problema odpovedo. Nekatera tipična področja so:

- Fizika (načrtovanje nuklearnih reaktorjev)
- Medicina (načrtovanje obsevanj)
- Energetika in metalurgija (obravnava toplotnega sevanja)
- Promet (obravnava prometnih tokov)
- Ekonomija (napovedovanje dogodkov na borzi)
- Računalniška grafika (generiranje slik)
- Telekomunikacije (EM valovanje)

Eden najpomembnejših delov simulacijskih okolij je generiranje naključnih signalov oz dogodkov. Pravo naključno dogajanje v naravi izkazuje na primer čas med posameznimi zaznavami Geigerjevega števca, ko je izpostavljen radioaktivnemu elementu. Enostaven generator naključnih števil lahko predstavlja igralna kocka, ruleta ali vlečenje oštevilčenih kroglic iz klobuka. Pri računalniški simulaciji naključna števila generira računalnik. Vsi naključni generatorji, ki so izvedeni na digitalnih računalnikih, temeljijo na matematičnih algoritmih, ki so ponovljivi in sekvenčni. Zato so naključna števila, ki jih pri tem dobimo, le psevdonaključna.

Bistvene zahteve za dobre naključne generatorje so naslednje:

- Nekorelirano zaporedje. Katerokoli podzaporedje naključnih števil ne sme biti korelirano z nobenim drugim podzaporedjem naključnih števil.
- Dolga perioda. Idealno bi bilo, da se zaporedje ne bi nikoli ponovilo, v praksi pa želimo čim daljšo periodo.
- Enakomernost. Zaporedje naključnih števil naj bo enakomerno porazdeljeno. Če npr. interval [0,1] razdelimo na podintervale, po dovolj velikem številu generiranih števil pričakujemo približno enako število naključnih števil v vsakem podintervalu.
- Učinkovitost. Generatorji, uporabljeni v višjih programskih jezikih, tipično porabijo manj kot 1% procesorskega časa celotne Monte Carlo simulacje.

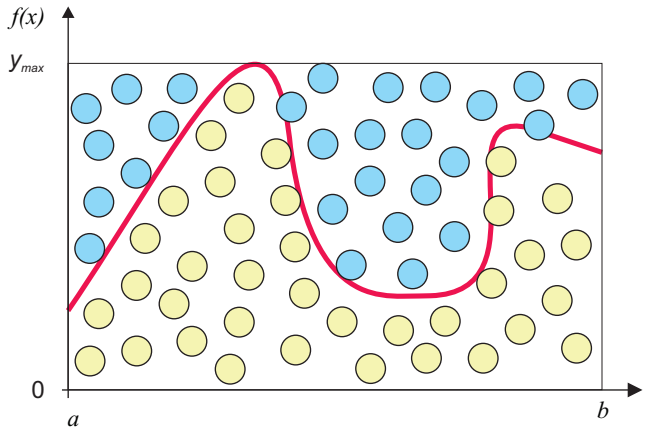
Primer: Izračun določenega integrala s pomočjo simulacije Monte Carlo

S simulacijo po metodi Monte Carlo lahko izračunamo vrednost določenega integrala

$$I = \int_a^b f(x)dx, \tag{2.1}$$

t.j. povsem deterministčnega problema tako, da z dvema naključnima generatorjema ($R_1, R_2 \in [0, 1]$) generiramo naključne vrednosti odvisne in neodvisne spremenljivke x in $f(x)$ z ustrezno statistiko - t.j. z enakomerno naključno porazdelitvijo) in pri dovolj velikem številu vzorcev izračunamo vrednost integrala. Postopek prikazuje slika 2.2.

Generirati je potrebno veliko število preizkusnih točk in določiti število točk, ki se nahajajo pod krivuljo. Če je N število vseh točk in M število točk pod krivuljo,



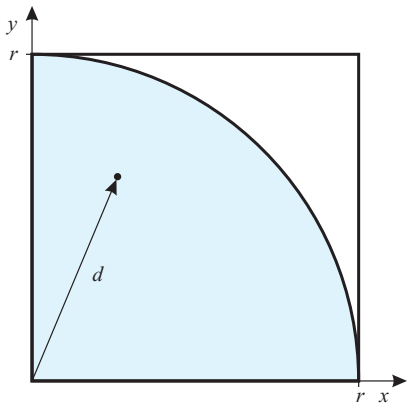
Slika 2.2: Računanje določenega integrala z metodo simulacije Monte Carlo

je vrednost določenega integrala

$$I = \int_a^b f(x)dx = (b-a)y_{max}\frac{M}{N} \tag{2.2}$$

Primer: Določitev vrednosti števila π

S pomočjo izračuna določenega integrala pod krivuljo, ki je četrtinka krožnice, določimo število π. Postopek ilustrira slika 2.3.



Slika 2.3: Računanje števila π

Ker je število točk pod krivuljo M sorazmerno ploščini $S_1 = \frac{\pi r^2}{4}$ in je število vseh

točk N sorazmerno s ploščino kvadrata $S_2 = r^2$, določimo π iz enačbe

$$\pi = 4\frac{M}{N} \tag{2.3}$$

Program v jeziku Matlab pa je naslednji

```
clear all
format long
N=4000;
M=1;
%vris cetrtine kroznice
r=1;fi=[0:0.001:pi/2]
x_kroznica=r*sin(fi);
y_kroznica=r*cos(fi);
plot(x_kroznica,y_kroznica,'r','LineWidth',4)
hold on

%Simulacija Monte Carlo
for i=1:N
    x=rand*r;
    y=rand*r;
    d=sqrt(x*x+y*y);
    if d<=r
    M=M+1;
    else
    end
    plot(x,y,':')
end

%Izracun pi
PI=4*M/N
  text(.4,.4,['PI=' num2str(PI)])
```

`rand` je funkcija za generacijo naključnega števila z enakomerno verjetnostno porazdelitvijo ($R1, R2 \in [0, 1]$

Simulacija po metodi Monte Carlo se uporablja v primerih slabo razvitih numeričnih metod ali pa, če le-te sploh ne obstajajo. Nekaj pomembnih prednosti je naslednjih:

- Omogoča obravnavo najzahtevnejših problemov, ki vsebujejo nelinearno odvisne parametre od večih spremenljivk, zapleteno geometrijo.
- Dodana dimenzija v primeru izračuna integrala ali dodana nelinearnost parametra v modelu običajno zahteva zelo majhne popravke simulacijskega modela.
- S statističnimi metodami je možno oceniti negotovost rezultatov simulacije, kar pri konvencionalnih postopkih ni slučaj.

Seveda pa ima metoda tudi slabosti:

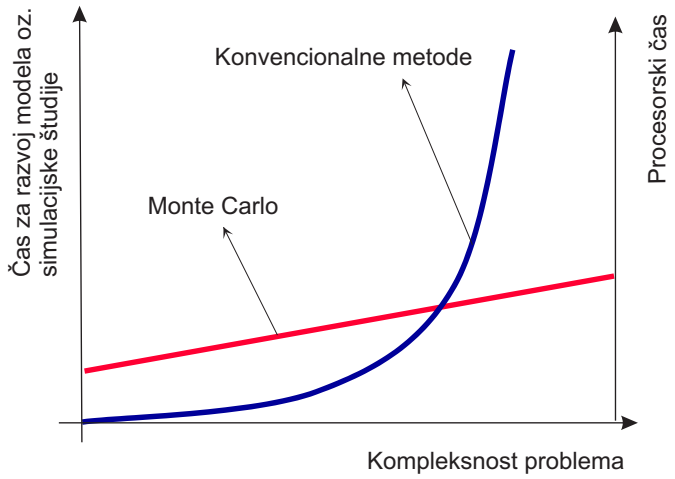
- Zaradi statistične narave rezultati izkazujejo statistično negotovost.
- Negotovost je odvisna od primera do primera, zato je potrebno število vzorčenj določiti za vsak obravnavani primer posebej.
- Konvergenca metode ni zagotovljena sama po sebi.
- Potrebna je večja stopnja prilagoditve problemu, zato pristop zahteva več izkušenj in intuicije programerja.

Slika 2.4 prikazuje odvisnost časa, ki ga potrebujemo za razvoj modela oz simulacijske študije in porabe računalniškega (procesorskega) časa v odvisnosti od kompleksnosti modela pri uporabi konvencionalne metode in metode Monte Carlo.

Vidimo, da ima metoda Monte Carlo lahko prednost pri simulaciji kompleksnih problemov

2.4 Kombinirana ali hibridna simulacija

Cellier (Cellier, 1979) je označil kombinirano ali hibridno simulacijo kot simulacijo sistemov, ki jih lahko opišemo na celotnem intervalu opazovanja ali na delu tega intervala z diferencialnimi enačbami, pri čemer pa vsaj ena spremenljivka stanj ali njen odvod ni zvezna veličina. Po tej definiciji vidimo, da je praktično vsaka simulacija realnega problema v bistvu kombinirana simulacija. Vendar je tako stroga definicija kombinirane simulacije smiselna, ker smo na isti način definirali



Slika 2.4: Odvisnost časa, ki ga potrebujemo za razvoj modela oz simulacisjke študije in porabe računalniškega (procesorskega) časa v odvisnosti od kompleksnosti modela

tudi zvezno simulacijo. To pa ne pomeni, da orodja za zvezno simulacijo sistemov niso primerna za obravnavo realnih sistemov. Ta orodja imajo običajno tudi sicer zelo omejene možnosti kombinirane simulacije. Niso pa primerna za simulacijo sistemov, kjer so diskretni dogodki ali nezveznosti zelo pogoste. Imajo namreč slabo razvite mehanizme za prehod iz zveznega v diskretni del modela in obratno. Medtem ko orodja za kombinirano simulacijo vedno numerično pravilno obdelujejo nezveznosti, imajo pri zveznih digitalnih simulacijskih orodjih to lastnost samo redki obstoječi simulacijski jeziki. Šele v zadnjem času posvečajo proizvajalci tej problematiki večjo pozornost.

2.5 Simulacija v realnem času

Simulacija v realnem času predstavlja posebno zahtevno obliko simulacije, ko je neodvisna spremenljivka sinhronizirana z realnim časom. Ponavadi je čas opazovanja (simulacije) neomejen (precej dolg), tako da redkeje govorimo o simulacijskih tekih oz. o njihovih dolžinah.

Prav področje vodenja sistemov je izredno vplivalo na razvoj materialne in programske opreme za simulacijo v realnem času. Omejene zmožnosti preizkušanja sistemov vodenja ter vadbe operaterjev na realnih industrijskih sistemih zahtevajo uporabo kompleksnih simulatorjev za delo v realnem času. Na ta način

je možno v realnem času preizkušati tudi postopke, ki lahko proces pripeljejo v napačno delovanje ali celo v katastrofo.

Razen sinhronizacije z realnim časom je pri tovrstni simulaciji običajno značilno tudi:

- zajemanje za računalniški sistem primerno pripravljenih podatkov realnih signalov,
- posredovanje rezultatov v obliki realnih signalov in
- neomejena dolžina simulacijskega teka.

Večina sodobnih konvencionalnih računalnikov ni sposobnih učinkovito izvajati simulacije v realnem času zaradi (Lincoln, 1988):

- preskromnih sposobnosti materialne opreme,
- zaradi preveč kompleksne in za potrebe simulacije v realnem času redundantne programske opreme,
- zaradi premalo učinkovitih možnosti za programiranje,
- zaradi preskromnih vhodno-izhodnih zmožnosti.

S stališča uporabe simulacije v realnem času ločimo dve glavni področji:

- razvoj in preizkušanje novih sistemov,
- izobraževanje in vadba.

V zvezi z razvojem in preizkušanjem novih sistemov predstavlja simulacija inženirsko načrtovalno orodje. Tovrstna simulacija lahko zahteva velike procesne zmožnosti. Kot primer naj omenimo simulacijo leta letala, ki se lahko izredno uspešno uporablja pri razvoju in preizkušanju sistema za avtomatsko vodenje leta. Upoštevati mora natančen in kompleksen model z visokofrekvenčnimi in nizkofrekvenčnimi komponentami. Zaradi hitrih pojavov je potreben majhen računski korak (frame time cca. $1ms$). Na drugi strani pa tak sistem ne potrebuje

posebnih vhodno-izhodnih zmožnosti, saj se lahko učinkovitost sistema vodenja preveri ob spremljanju manjšega števila signalov.

Simulatorji za izobraževanje in vadbo pa morajo zagotoviti čimbolj realno okolje. Ker ima človek omejene možnosti reagiranja, zajemajo modeli samo ustrezno frekvenčno področje. Zato take simulacije niso časovno kritične (tipični računski korak simulatorjev za vadbo pilotov je v razredu desetinke sekunde) in ne zahtevajo ekstremnih procesnih sposobnosti. Zato pa taki simulatorji zahtevajo zelo sposobne vhodno-izhodne zmožnosti za realno predstavitev problema (npr. prikaz vseh instrumentov na zaslonu v primeru simulatorja za vadbo pilota).

Simulacija v realnem času zahteva običajno velike procesne zmožnosti računalnikov. Le-te so povezane s:

- potrebnim računskim korakom (odvisnost od časovnih konstant sistema),
- kompleksnostjo simuliranega objekta (odvisnost od kompleksnosti realnega objekta, zahtevane natančnosti, ...) in
- potrebo po vhodno-izhodnih operacijah (odvisnost od namena simulatorja).

Zato si v povezavi s simulacijo v realnem času v glavnem predstavljamo uporabo namenskih digitalnih računalnikov, seveda pa tudi analogno-hibridnih računalnikov.

V zadnjem času se simulacija v realnem času vse več uporablja na področju izobraževanja. V laboratorijih se vaje na realnih napravah zamenjujejo z ustrezno simuliranimi napravami, ki delujejo v realnem času. Realne pilotne naprave so zaradi omejenih zmožnosti praviloma preveč enostranske, da bi omogočale učinkovit in kreativen pedagoški proces. S simulacijo v realnem času je možno pripraviti več problemov, tako da lahko vsak študent rešuje svoj problem. Pri vajah na pilotni napravi morajo praktično vsi študenti delati isto. Tak pristop k laboratorijskim vajam predstavlja najboljšo uskladitev med realnostjo, kompleksnostjo, stroški in varnostjo.

3.

Osnovne metode pri reševanju problemov s simulacijo

V tem delu bomo pokazali osnovne metode pri reševanju problemov s simulacijo. *Indirektni postopek* pri reševanju diferencialnih enačb prikazuje bistvo zvezne simulacije. Razen tega postopka bomo opisali direktno in implicitno metodo ter več načinov simulacije prenosnih funkcij. Prikazali bomo tudi metodo za simulacijo sistemov z mrtvim časom in način simulacije kompleksnih sistemov. Postopki nas bodo pripeljali do simulacijskih shem, ki so neodvisne od uporabljenega simulacijskega orodja in predstavljajo osnovo pri računalniški simulaciji. Končno bomo razčlenili tudi koncept digitalne simulacije.

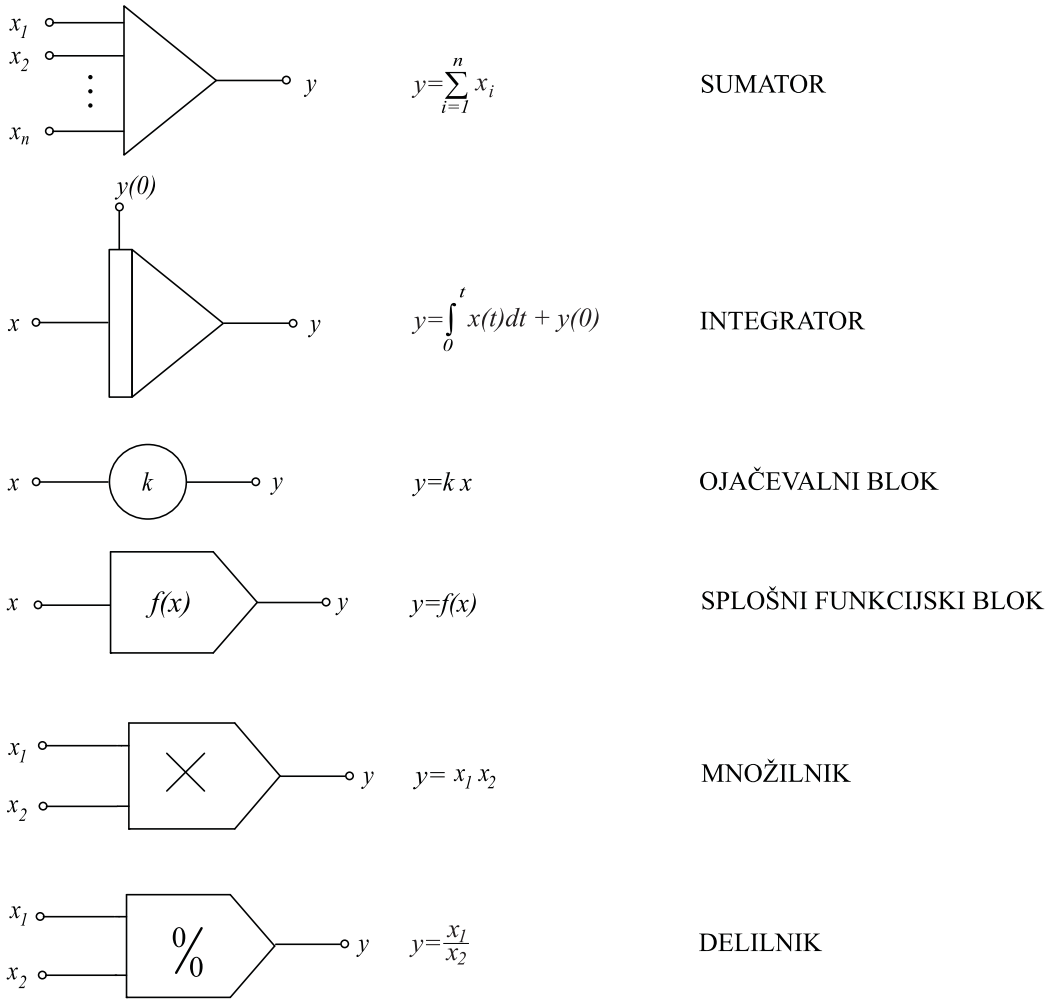
3.1 Simulacijska shema

Osnova za zvezno simulacijo nekega modela je posebna grafična predstavitev, ki jo bomo imenovali *simulacijska shema*. Le-ta ima veliko skupnega z bločnimi diagrami, ki jih zlasti uporabljamo pri zapisu sistemov vodenja. Zato bomo tudi osnovne gradnike simulacijskih shem imenovali *bloke*. Blok predstavlja funkcijo določenega gradnika simulacijske sheme. Grafično predstavitev bloka pa imenujemo *ikona*.

Ker osnove simulacije izvirajo iz konceptov analogne simulacije, izhajajo tudi simulacijske sheme iz analognih simulacijskih shem. Lord Kelvin je imenoval

ustrezno shemo “shemo diferencialne analize". Danes se v praksi uporabljajo zelo različne oblike simulacijskih shem tako, da praktično vsak izdelovalec simulacijskega orodja uvede kakšne svoje bloke oz. ikone. S čim manjšim naborom blokov oz. ikon bomo skušali uvesti shemo, ki bo uporabna za kakršno koli simulacijsko orodje. Tako bodo take sheme uporabne za simulacijo z digitalnim simulacijskim jezikom kot tudi za simulacijo z analognim računalnikom, kjer pa je potrebno dodatno upoštevati, da elementi analognega računalnika obračajo signalom predznak.

Osnovne bloke, ki jih bomo uporabljali v simulacijski shemi, prikazuje slika 3.1. Po potrebi pa bomo kasneje definirali še kakšen nov blok oz. ikono.



Slika 3.1: Pogosto uporabljeni bloki oz. ikone v simulacijski shemi

Predznake, ki učinkujejo v posameznih blokih, lahko vpeljemo preko ustreznih parametrov (npr. predznak konstante k pri ojačevalnem bloku, ki signal x pomnoži s konstanto k), lahko pa ga definiramo v bližini, kjer vhodni signal vstopa v blok (če predznaka ni, privzamemo pozitivni predznak). Sumator ima poljubno število vhodov, integrator pa le enega. Splošni funkcijski blok je najsplošnejši. Predstavlja lahko vir signalov (takrat običajno ne rišemo vhodnega priključka) ali pa poljubne nelinearne zakonitosti med vhodom in izhodom. V konkretnih shemah namesto $f(x)$ vpišemo v blok ustrezno matematično relacijo (npr. SIN za funkcijo sinus) ali pa vrišemo grafični simbol, ki nazorno pove, za kakšen tip bloka gre. Vlogo bloka lahko nadalje posplošimo, če definiramo vhod in izhod kot vektorska signala ($\boldsymbol{y} = \boldsymbol{f}(\boldsymbol{x})$).

3.2 Indirektna metoda

Ker so matematični modeli dinamičnih sistemov običajno opisani s sistemom diferencialnih enačb, predstavlja indirektna metoda za reševanje diferencialnih enačb osnovni simulacijski pristop. Po tej metodi je potrebno najvišji odvod integrirati tolikokrat, kolikor je njegov red. S tem indirektno generiramo vse nižje odvode in samo spremenljivko. V tem načinu se skriva bistvo simulacije. Analitična rešitev diferencialne enačbe in tabeliranje rešitve v določenih točkah neodvisne spremenljivke nima nobene zveze s simulacijo. Včasih sicer na ta način lahko pridemo hitreje do bolj točnih rezultatov, vendar je v praksi uporabnost takega analitičnega pristopa zelo omejena (npr. le za linearne sisteme).

Indirektna metoda je uporabna, če je možno iz diferencialne enačbe izraziti najvišji odvod in če ne nastopajo višji odvodi vhodnega signala. Indirektno metodo opišimo za sistem

$$y^{(n)} + f(y^{(n-1)}, y^{(n-2)}, \ldots, y', y, u; t) = 0 \tag{3.1}$$

y je izhodni signal, u je vhodni signal, t pa je neodvisna spremenljivka simulacije (čas). Postopek opišemo v treh točkah:

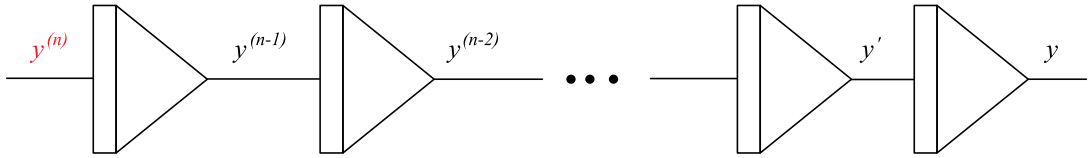
1. korak

Preuredimo diferencialno enačbo tako, da ostane na levi strani najvišji odvod, vse ostalo pa prenesemo na desno stran. Če je sistem zapisan v prostoru stanj (s sistemom diferencialnih enačb 1. reda), je zapis že ustrezen in prvi korak odpade

$$y^{(n)} = -f(y^{(n-1)}, y^{(n-2)}, \ldots, y', y, u; t) \tag{3.2}$$

2. korak

Narišemo kaskado n integratorjev, če je n red najvišjega odvoda. 2. korak prikazuje slika 3.2.

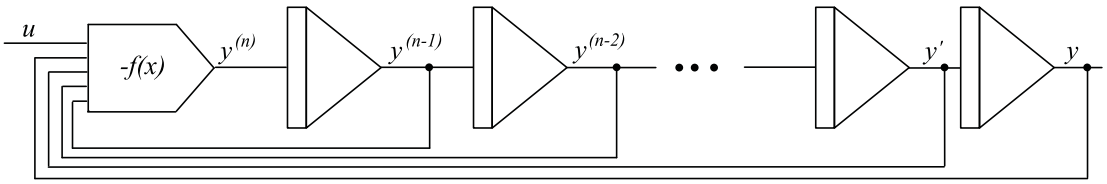


Slika 3.2: 2. korak indirektnega postopka

Predpostavimo, da je vhod v prvi integrator (najvišji odvod) znan, preostali integratorji pa generirajo nižje odvode in samo spremenljivko oz. rešitev diferencialne enačbe.

3. korak

Z upoštevanjem (virtualnih) nižjih odvodov in rešitve diferencialne enačbe generiramo negativno funkcijsko odvisnost, ki realizira desno stran enačbe (3.2). Izhod bloka, ki generira negativno funkcijsko odvisnost, je enak najvišjemu odvodu, zato ga moramo povezati na vhod prvega integratorja. Pri generaciji funkcije -f uporabljamo različne bloke (razen integratorja), kar zavisi od oblike diferencialne enačbe. Dobljeni obliki, ki jo prikazuje slika 3.3, pravimo tudi kanonična oblika.



Slika 3.3: 3. korak indirektnega postopka

Postopek, ki smo ga opisali, je neposredno uporaben, če diferencialna enačba ne vsebuje odvodov vhodnega signala. Če pa le-ti nastopajo, je v primeru lin-

earnih sistemov bolj smiselno sistem simulirati po konceptu prenosnih funkcij (podpoglavje 3.5).

Uporabnost postopka bomo prikazali na primerih.

Primer 3.1 Temperaturni proces

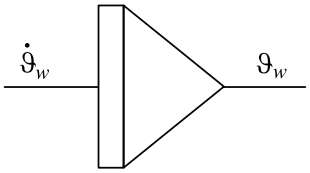
Matematični model temperaturnega procesa [Zupančič, 2012] opisuje diferencialna enačba

$$\dot{\vartheta}_w + \frac{1}{T}\vartheta_w = \frac{k}{T}p \tag{3.3}$$

Z upoštevanjem 1. koraka indirektnega postopka moramo enačbo (3.3) preurediti v obliko

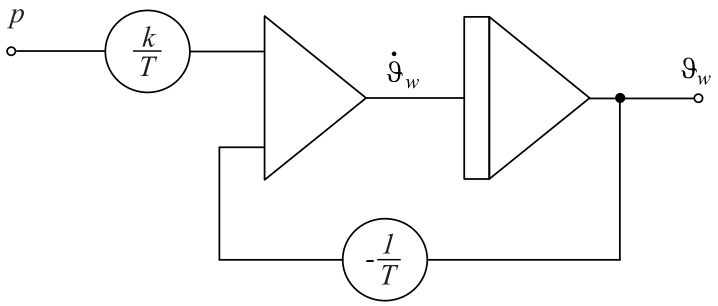
$$\dot{\vartheta}_w = -\frac{1}{T}\vartheta_w + \frac{k}{T}p \tag{3.4}$$

V drugem koraku narišemo le en integrator, ker je diferencialna enačba 1. reda. Ustrezen korak prikazuje slika 3.4.



Slika 3.4: Simulacijska shema za 2. korak

V 3. koraku generiramo desno stran enačbe (3.4). Končno simulacijsko shemo prikazuje slika 3.5.



Slika 3.5: Simulacijska shema temperaturnega procesa

Kot lahko vidimo na sliki 3.5, smo desno stran enačbe (3.4) realizirali s sumatorjem in dvema ojačevalnima blokoma. Potrebna predznaka sta vključena v ojačevalnih blokih. □

Primer 3.2 Avtomobilsko vzmetenje

Matematični model avtomobilskega vzmetenja [Zupančič, 2012] opisuje diferencialna enačba

$$\dddot{y}_1 + \frac{k_1 + k_2}{f}\ddot{y}_1 + \frac{k_2}{M}\dot{y}_1 + \frac{k_1 k_2}{Mf}y_1 = 0 \qquad y_1(0) = -y_{10} \tag{3.5}$$

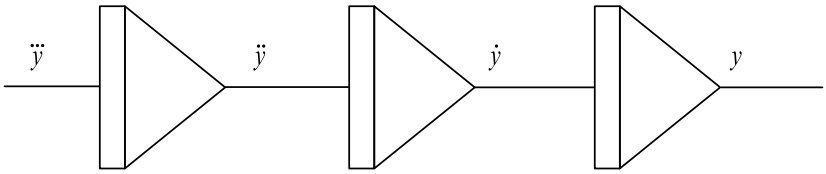
ki jo lahko poenostavimo v obliko

$$\dddot{y} + a\ddot{y} + b\dot{y} + cy = 0 \qquad y(0) = -d \tag{3.6}$$

1. korak: Preuredimo enačbo (3.6)

$$\dddot{y} = -a\ddot{y} - b\dot{y} - cy \qquad y(0) = -d \tag{3.7}$$

2. korak: Narišemo kaskado treh integratorjev, kar prikazuje slika 3.6



Slika 3.6: Simulacijska shema za 2. korak

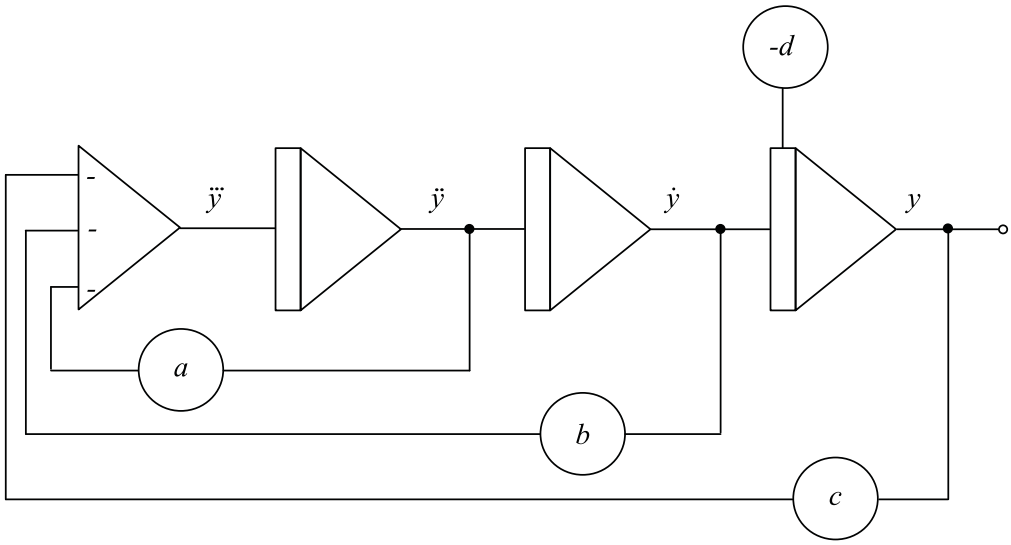
3. korak: Zaključimo simulacijsko shemo, kot prikazuje slika 3.7. V tem primeru smo nekatere predznake podali v sumacijskem bloku.

□

Primer 3.3 Ekološki sistem žrtev in roparjev

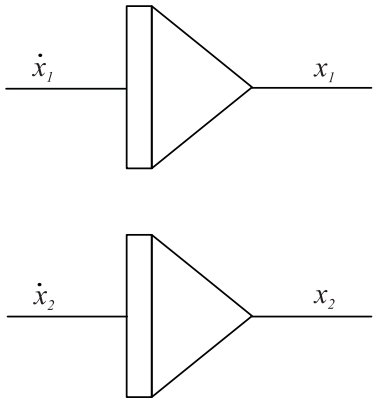
Model ekološkega sistema, v katerem nastopajo roparji in žrtve [Zupančič, 2012] prikazuje uporabnost indirektne metode pri reševanju sistema nelinearnih diferencialnih enačb. Matematični model ima obliko

$$\begin{aligned} \dot{x}_1 &= a_{11}x_1 - a_{12}x_1x_2 \qquad x_1(0) = x_{10} \\ \dot{x}_2 &= a_{21}x_1x_2 - a_{22}x_2 \qquad x_2(0) = x_{20} \end{aligned} \tag{3.8}$$



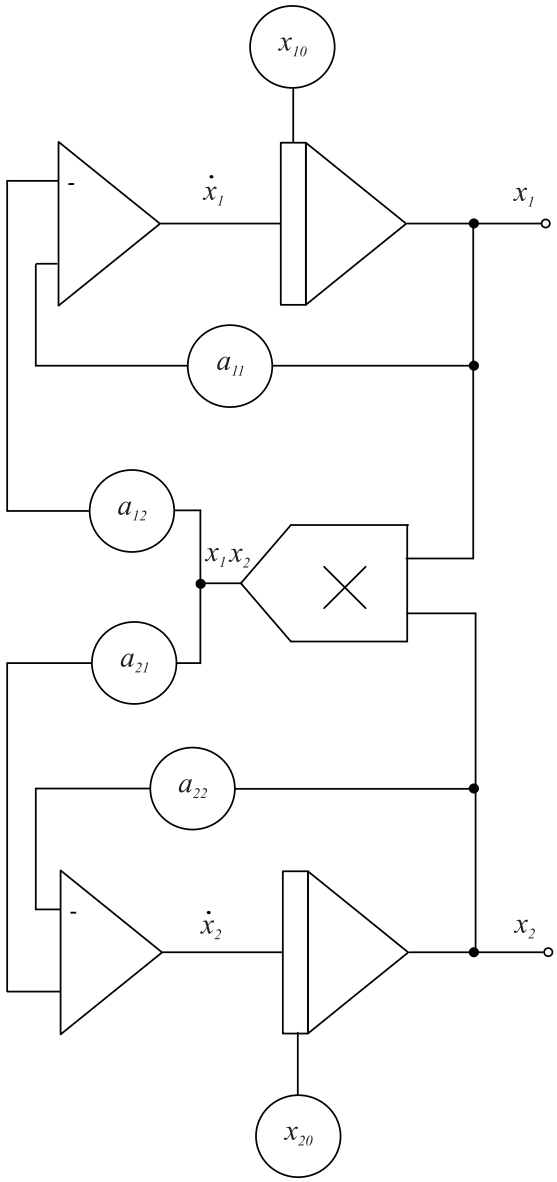
Slika 3.7: Simulacijska shema sistema avtomobilskega vzmetenja

1. korak: Ni potreben, ker imata enačbi že predpisano obliko.
2. korak: Za vsako enačbo narišemo en integrator, kar prikazuje slika 3.8.
3. korak: Zaključimo simulacijsko shemo, kot to prikazuje slika 3.9.



Slika 3.8: Simulacijska shema za 2. korak

Kot vidimo iz slik 3.8 in 3.9, je postopek simulacije nelinearnega sistema enako enostaven kot pri linearnih sistemih. Razen zank okoli posameznih integratorjev dobimo v tem primeru tudi ustrezne križne povezave. Nelinearnost vnaša v shemo množilnik. □



Slika 3.9: Simulacijska shema ekološkega sistema

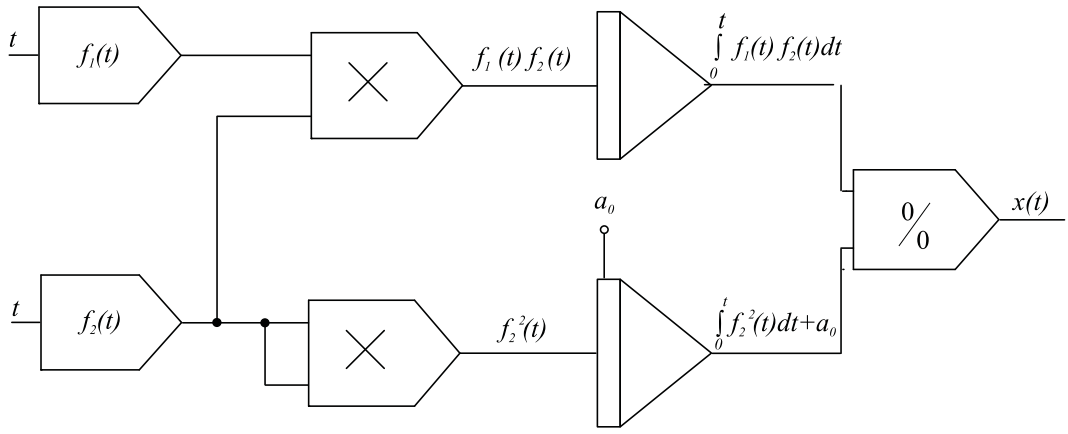
3.3 Direktna metoda

S pomočjo direktne metode realiziramo izraze v obliki formul, ki so ponavadi algebraične narave. Tako realiziramo zlasti razne izraze, ki jih potrebujemo v indirektnem načinu. Tudi v tem primeru je smiselno uporabiti simulacijsko shemo.

Primer 3.4 Potrebno je izračunati izraz

$$x(t) = \frac{\int_0^t f_1(t) f_2(t) dt}{\int_0^t f_2^2(t) dt + a_0} \tag{3.9}$$

Simulacijsko shemo prikazuje slika 3.10.



Slika 3.10: Simulacijska shema pri uporabi direktne metode

□

3.4 Implicitna metoda

S pomočjo implicitne metode uporabljamo simulacijo za generiranje funkcij. Poiskati poizkušamo neko diferencialno enačbo, katere rešitev je analitična funkcija, ki jo želimo generirati. Zaradi enostavnosti indirektnega postopka, ki ga uporabljamo pri reševanju diferencialne enačbe, je tak postopek upravičen.

Pri implicitni metodi izraz oz. funkcijo, ki jo želimo generirati, enkrat ali večkrat odvajamo na neodvisno spremenljivko. Po vsakem odvajanju moramo izraziti čim več členov s funkcijo ali njenimi odvodi. S postopkom oz. odvajanjem končamo, ko dobimo obliko, v kateri nastopa le funkcija in njeni odvodi. Seveda vedno ne moremo priti do take oblike. Na koncu je potrebno določiti tudi začetne pogoje.

Primer 3.5 Najenostavnejši problem predstavlja implicitna generacija eksponentne funkcije

$$y = e^{-at} \tag{3.10}$$

Z enkratnim odvajanjem dobimo enačbo

$$\dot{y} = -ae^{-at} \tag{3.11}$$

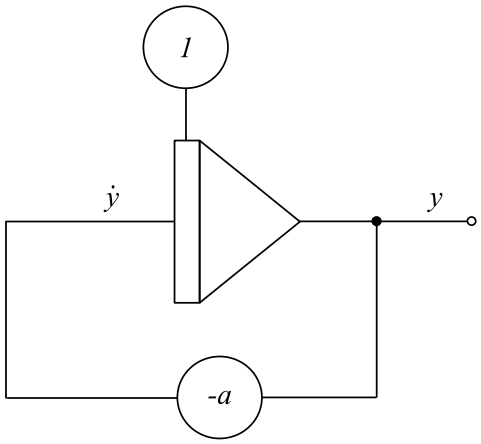
in če člen e^{-at} zamenjamo z y (enačba 3.10), dobimo diferencialno enačbo

$$\dot{y} = -ay \tag{3.12}$$

Začetni pogoj izračunamo s pomočjo enačbe (3.10) za $t = 0$

$$y(0) = 1 \tag{3.13}$$

Enačbi (7.38) in (3.13) rešimo (simuliramo) z indirektno metodo. Shemo prikazuje slika 3.11.



Slika 3.11: Implicitna generacija funkcije $y = e^{-at}$

□

3.5 Simulacija prenosnih funkcij

Prenosne funkcije imajo pomembno vlogo pri analizi in načrtovanju sistemov. Zlasti je učinkovita nazornost bločnih diagramov, v katerih nastopajo med ostalimi funkcionalnimi bloki tudi prenosne funkcije.

Od številnih metod za simulacijo prenosnih funkcij bomo prikazali dve, ki sta najbolj nazorni in se največ uporabljata. To sta *vgnezdena* in *delitvena metoda*. Razen teh dveh osnovnih metod pa bomo obravnavali še dve razčlenitveni metodi. Le-ti sta zlasti primerni pri simulaciji bolj kompleksnih prenosnih funkcij.

3.5.1 Vgnezdena metoda

Postopek bomo prikazali na primeru 3.6.

Primer 3.6 Prenosno funkcijo

$$G(s) = \frac{Y(s)}{U(s)} = \frac{as^3 + bs^2 + cs + d}{s^3 + es^2 + fs + g} \tag{3.14}$$

simuliramo z vgnezdeno metodo s pomočjo naslednjih korakov:

- Enačbo (3.14) preuredimo v obliko

$$(s^3 + es^2 + fs + g)Y = (as^3 + bs^2 + cs + d)U \tag{3.15}$$

- Če koeficient pri členu z najvišjo potenco v imenovalcu ni enak 1, je potrebno vse člene polinomov v številcu in imenovalcu deliti s tem koeficientom.
- Združimo vse člene, ki imajo enako potenco spremenljivke s

$$s^3(Y - aU) + s^2(eY - bU) + s(fY - cU) + (gY - dU) = 0 \tag{3.16}$$

- Preuredimo enačbo (3.16) tako, da imamo na levi najvišji "odvod"(potenco operatorja s) izhodne veličine

$$s^3Y = s^3aU - s^2(eY - bU) - s(fY - cU) - (gY - dU) \tag{3.17}$$

- Delimo enačbo (3.17) z najvišjo potenco spremenljivke s

$$Y = aU - \frac{1}{s}(eY - bU) - \frac{1}{s^2}(fY - cU) - \frac{1}{s^3}(gY - dU) \tag{3.18}$$

- Preuredimo enačbo (3.18) v vgnezdeno obliko

$$Y = aU + \frac{1}{s}\{(bU - eY) + \frac{1}{s}[(cU - fY) + \frac{1}{s}(dU - gY)]\} \tag{3.19}$$

- Ob vpeljavi pomožnih spremenljivk

$$\begin{aligned} U_0 &= \frac{1}{s}(dU - gY) \\ U_1 &= \frac{1}{s}(cU - fY + U_0) \\ U_2 &= \frac{1}{s}(bU - eY + U_1) \end{aligned} \tag{3.20}$$

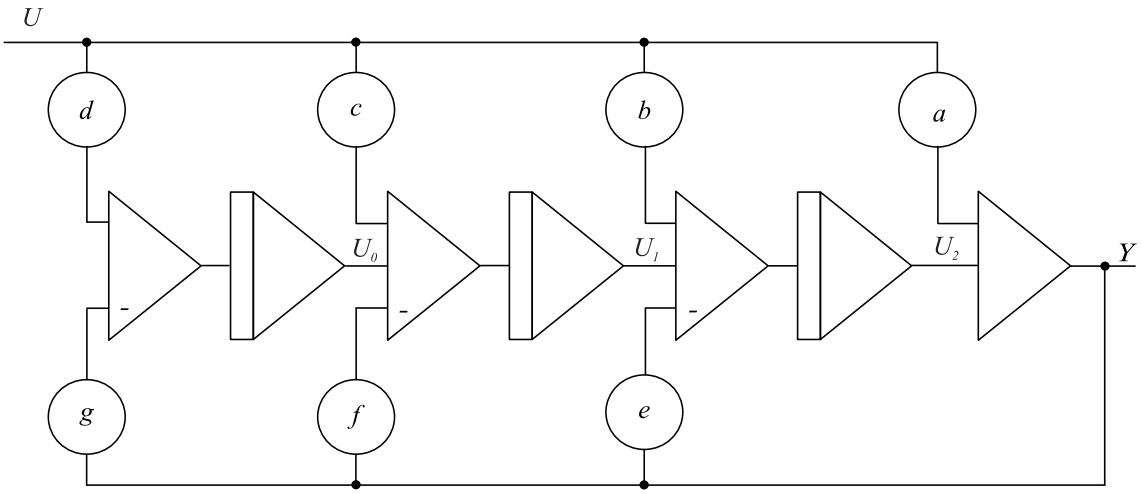
  dobi enačba (3.19) obliko

$$Y = aU + U_2 \tag{3.21}$$

- Z upoštevanjem enačb (3.20) in (3.21) dobimo simulacijsko shemo, ki jo prikazuje slika 3.12.

Če koeficient pri členu s^3 ne bi bil 1, bi bile vrednosti ojačevalnih blokov v sliki 3.12 kvocienti koeficientov a, b, c, d, e, f in g s tem koeficientom.

Shema na sliki 3.12 predstavlja realizacijo prenosne funkcije, ki je v teoriji vodenja znana kot *spoznavnostna kanonična oblika* (Ogata, 2010). Koeficienti prenosne



Slika 3.12: Simulacijska shema ob uporabi vgnezdene oblike

funkcije nastopajo kot ojačevalni bloki. Vpeljava pomožnih spremenljivk poenostavi postopek risanja simulacijske sheme kot tudi programiranje pri uporabi enačbno orientiranih simulacijskih jezikov. □

3.5.2 Delitvena metoda

Postopek bomo prikazali na primeru 3.7.

Primer 3.7 Prenosno funkcijo

$$G(s) = \frac{Y(s)}{U(s)} = \frac{as^3 + bs^2 + cs + d}{s^3 + es^2 + fs + g} \tag{3.22}$$

simuliramo z delitveno metodo s pomočjo naslednjih korakov:

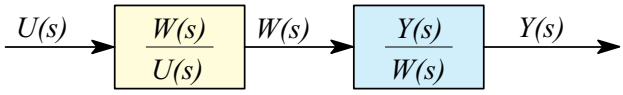
- Če koeficient pri členu z najvišjo potenco v imenovalcu ni enak 1, je potrebno vse člene polinomov v številcu in imenovalcu deliti s tem koeficientom.
- Prenosno funkcijo $\frac{Y(s)}{U(s)}$ razdelimo s pomočjo pomožne spremenljivke $W(s)$ v dve prenosni funkciji; prva predstavlja realizacijo imenovalca

$$\frac{W(s)}{U(s)} = \frac{1}{s^3 + es^2 + fs + g} \tag{3.23}$$

druga pa realizacijo števca

$$\frac{Y(s)}{W(s)} = as^3 + bs^2 + cs + d \tag{3.24}$$

Delitveni postopek nazorno prikazuje slika 3.13.



Slika 3.13: Bločna shema, ki predstavlja delitveni postopek

- Preuredimo enačbo (3.23) v obliko

$$s^3W = U - es^2W - fsW - gW \tag{3.25}$$

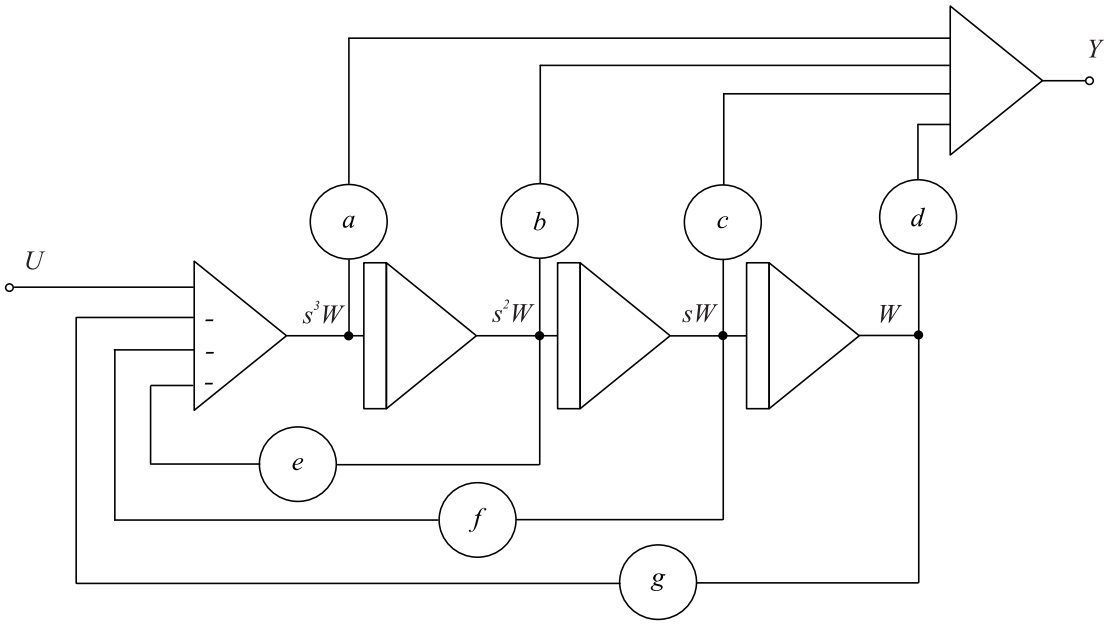
- Preuredimo enačbo (3.24) v obliko

$$Y = as^3W + bs^2W + csW + dW \tag{3.26}$$

- S pomočjo enačb (3.25) in (3.26) realiziramo simulacijsko shemo, ki jo prikazuje slika 3.14.

Če koeficient pri členu s^3 ne bil 1, bi bile vrednosti ojačevalnih blokov v sliki 3.14 kvocienti koeficientov a, b, c, d, e, f in g s tem koeficientom. Kot vidimo iz slike 3.14, sestoji struktura iz dveh delov. Eden realizira imenovalec z indirektno metodo (le ta namreč omogoča, da so razen izhodne spremenljivke dostopni tudi odvodi), drugi del pa generira števec s pomočjo direktne metode. Metoda je zelo primerna za simulacijo več prenosnih funkcij z enakim imenovalcem, saj le enkrat simuliramo prenosno funkcijo $\frac{W}{U}$.

Shema na sliki 3.14 predstavlja realizacijo prenosne funkcije, ki je v teoriji vodenja znana kot *vodljivostna kanonična oblika* (Ogata, 2010). □



Slika 3.14: Simulacijska shema ob uporabi delitvene metode

Obe metodi potrebujeta toliko integratorjev, kolikor je red prenosne funkcije. Uporabljamo jih tudi pri simulaciji sistemov, ki so opisani z diferencialnimi enačbami, pri čemer nastopajo tudi višji odvodi vhodnega signala (npr. $\dddot{y} + e\ddot{y} + f\dot{y} + gy = a\ddot{u} + b\ddot{u} + c\dot{u} + du$).

3.5.3 Vzporedna razčlenitev

Po tej metodi razčlenimo prenosno funkcijo v vsoto več prenosnih funkcij. Postopek lahko opišemo z naslednjimi koraki:

- Poiščemo realne pole prenosne funkcije.
- Po metodi nedoločenih koeficientov razčlenimo prenosno funkcijo na vsoto delnih ulomkov. Imenovalci delnih ulomkov so določeni z realnimi poli, v primeru konjugirano kompleksnih polov pa z ustreznimi kvadratnimi členi. Če je stopnja števca enaka stopnji imenovalca, delimo polinom v števcu s polinomom v imenovalcu. S tem dobimo v vzporedni razčlenitvi tudi konstantni člen.

- Narišemo simulacijsko shemo vsakega člena. Vhode vseh členov povežemo skupaj, izhode vseh členov pa seštejemo na sumatorju.

Primer 3.8 Simulirajmo prenosno funkcijo

$$G(s) = \frac{Y(s)}{U(s)} = \frac{4s + 10}{s^2 + 6s + 8} \tag{3.27}$$

Če izračunamo pole, lahko enačbo (3.27) zapišemo v obliki

$$G(s) = \frac{4(s + 2.5)}{(s + 4)(s + 2)} \tag{3.28}$$

Z razčlenitvijo na delne ulomke dobimo iz enačbe (3.28) izraz

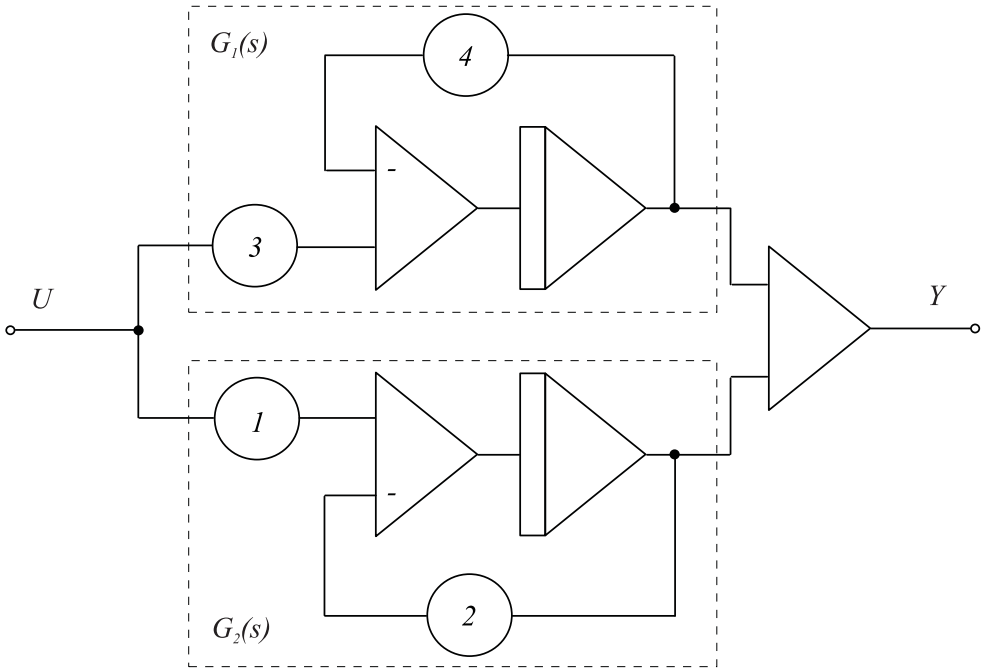
$$G(s) = G_1(s) + G_2(s) = \frac{3}{s + 4} + \frac{1}{s + 2} \tag{3.29}$$

Gre torej za enak postopek razčlenitve, kot pri računanju inverzne Laplace-ove transformacije. Ustrezno simulacijsko shemo, kjer smo $G_1(s)$ in $G_2(s)$ realizirali z vgnezdeno metodo, prikazuje slika 3.15.

□

3.5.4 Zaporedna razčlenitev

Po tej metodi poiščemo pole in ničle prenosne funkcije in zapišemo ustrezno faktorizirano obliko. V primeru konjugirano kompleksnih korenov ohranimo kvadratne člene. Nato iz prenosne funkcije generiramo produkt več prenosnih funkcij, pri čemer ena prenosna funkcija vsebuje elementarne gradnike kot npr.: ojačenje, en pol, kvocient ničle in pola, kvadratni člen v imenovalcu, kvocient kvadratnih členov, ... Te osnovne gradnike zaporedno povežemo v simulacijsko shemo.



Slika 3.15: Realizacija z metodo vzporedne razčlenitve

Primer 3.9 Simulirajmo enako funkcijo, kot v primeru 3.8

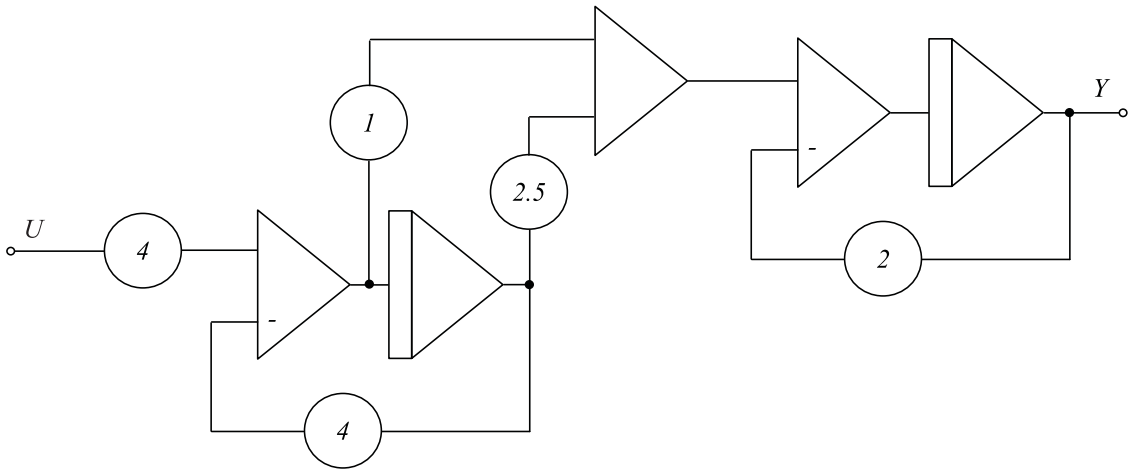
$$G(s) = \frac{Y(s)}{U(s)} = \frac{4s + 10}{s^2 + 6s + 8} \tag{3.30}$$

Po izračunu polov in ničel lahko enačbo (3.30) zapišemo v obliki

$$\begin{aligned} G(s) &= 4G_1(s)G_2(s) \\ G_1(s) &= \frac{s + 2.5}{s + 4} \\ G_2(s) &= \frac{1}{s + 2} \end{aligned} \tag{3.31}$$

$G_1(s)$ in $G_2(s)$ realizirajmo po delitveni metodi. Po ustrezni zaporedni povezavi dobimo simulacijsko shemo, ki jo prikazuje slika 3.16.

□



Slika 3.16: Realizacija z metodo zaporedne razčlenitve

3.6 Kanonične oblike

Pri simulaciji prenosnih funkcij smo že omenili vodljivostno in spoznavnostno kanonično obliko. To sta za načrtovanje vodenja dve zelo pomembni obliki, primerni za načrtovanje regulatorja stanj in observatorja. Pretvorba v kanonične oblike je možna iz poljubne oblike v prostoru stanj s pomočjo transformacijskih matrik (matrike $\boldsymbol{T}$). Simulacijska znanja pa nam omogočajo bolj inženirski pristop - transformacije iz prenosne funkcije v vodljivostno, spoznavnostno ali diagonalno kanonično obliko. Sistem, ki je zapisan v poljubni obliki v prostoru stanj, se namreč zelo enostavno prevede v prenosno funkcijo s pomočjo enačbe

$$G(s) = \boldsymbol{C}(s\boldsymbol{I} - \boldsymbol{A})^{-1}\boldsymbol{B} + D \tag{3.32}$$

Pretvorba iz prenosne funkcije v prostor stanj pa ni enolična. V tem poglavju si bomo torej ogledali, kako pretvoriti prenosno funkcijo (oz. ustrezno diferencialno enačbo) v različne *standardne* oz. *kanonične zapise*

3.6.1 Vodljivostna kanonična oblika

Sistem, ki ga opisuje prenosna funkcija

$$G(s) = \frac{Y(s)}{U(s)} = \frac{b_0 s^n + b_1 s^{n-1} + \ldots + b_{n-1} s + b_n}{s^n + a_1 s^{n-1} + \ldots + a_{n-1} s + a_n} \tag{3.33}$$

prevedemo v *vodljivostno kanonično obliko*, če uporabimo t.i. *delitveni postopek* (partitioned form method) za simulacijo prenosne funkcije. Prenosno funkcijo razdelimo v dve prenosni funkciji

$$\frac{Y(s)}{U(s)} = \frac{W(s)}{U(s)} \frac{Y(s)}{W(s)} \tag{3.34}$$

$$\frac{W(s)}{U(s)} = \frac{1}{s^n + a_1 s^{n-1} + \ldots + a_{n-1} s + a_n} \tag{3.35}$$

$$\frac{Y(s)}{W(s)} = b_0 s^n + b_1 s^{n-1} + \ldots + b_{n-1} s + b_n \tag{3.36}$$

Prvo prenosno funkcijo realiziramo (simuliramo) s t.i. indirektnim načinom, kjer enačbo (3.35) zapišemo v obliko

$$s^n W = -a_1 s^{n-1} W - \ldots - a_{n-1} s W - a_n W + U \tag{3.37}$$

Enačbo (3.36) pa zapišemo v obliko

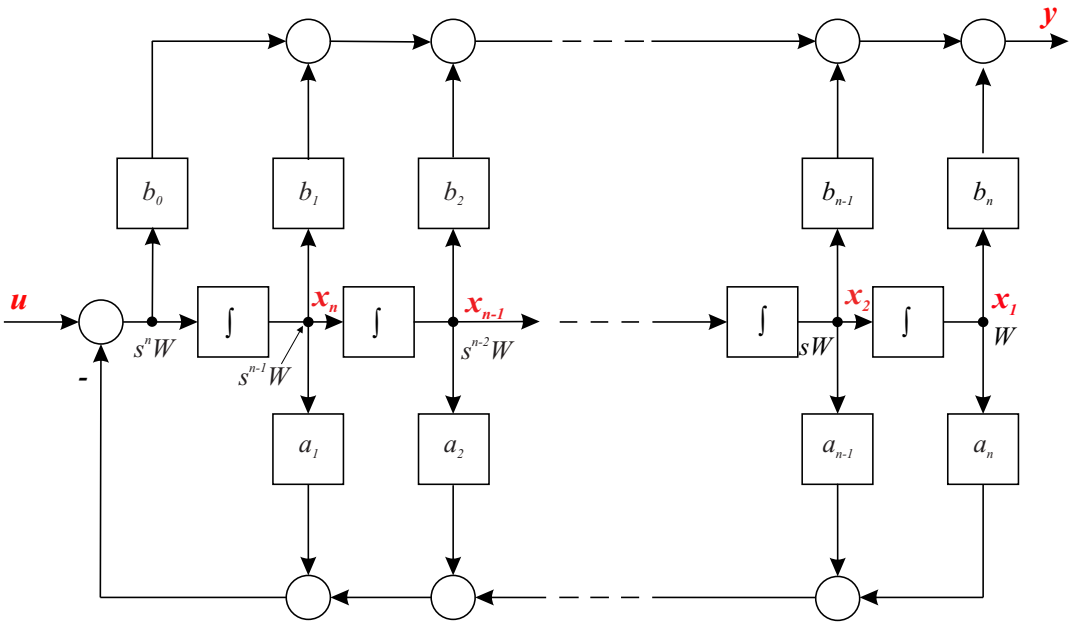
$$Y(s) = b_0 s^n W + b_1 s^{n-1} W + \ldots + b_{n-1} s W + b_n W \tag{3.38}$$

Simulacijsko (realizacijsko) shemo prikazuje slika 3.17.

Spremenljivke stanj izberemo na izhodih integratorjev in jih označimo z desne proti levi. Slika 3.17 nazorno prikazuje odvisnost odvodov spremenljivk stanj od spremenljivk stanj in vhodnega signala. Za izhodno enačbo pa enačbo (3.38) s pomočjo slike 3.17 preoblikujemo v

$$y(t) = (b_n - a_n b_0) x_1(t) + (b_{n-1} - a_{n-1} b_0) x_2(t) + \ldots + (b_1 - a_1 b_0) x_n + b_0 u \tag{3.39}$$

Torej je zapis v prostoru stanj v vodljivostni kanonični obliki



Slika 3.17: Vodljivostna kanonična oblika

$$\dot{\boldsymbol{x}}(t) = \begin{bmatrix} 0 & 1 & 0 & \dots & 0 & 0 \\ 0 & 0 & 1 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 0 & 1 \\ -a_n & -a_{n-1} & -a_{n-2} & \dots & -a_2 & -a_1 \end{bmatrix} \boldsymbol{x}(t) + \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix} u(t) \qquad (3.40)$$

$$y(t) = [\, b_n - a_n b_0 \quad b_{n-1} - a_{n-1} b_0 \quad \dots \quad b_2 - a_2 b_0 \quad b_1 - a_1 b_0 \,] \boldsymbol{x}(t) + b_0 u \qquad (3.41)$$

Matrika $\boldsymbol{A}$ ima t.i. *Frobeniusovo obliko* (companion oblika), matrika $\boldsymbol{B}$ ima obliko stolpnega vektorja s samimi ničlami razen enke na koncu, za matriko $\boldsymbol{C}$ pa ni posebnih zahtev.

Vodljivostna oblika je zlasti primerna za načrtovanje regulatorja stanj.

3.6.2 Spoznavnostna kanonična oblika

Sistem, ki ga opisuje prenosna funkcija

$$G(s) = \frac{Y(s)}{U(s)} = \frac{b_0 s^n + b_1 s^{n-1} + \ldots + b_{n-1} s + b_n}{s^n + a_1 s^{n-1} + \ldots + a_{n-1} s + a_n} \tag{3.42}$$

prevedemo v *spoznavnostno kanonično obliko*, če uporabimo t.i. *vgnezdeno metodo* (nested form method) za realizacijo (simulacijo) prenosne funkcije. Z navzkrižnim množenjem enačbe (3.42) dobimo

$$(s^n + a_1 s^{n-1} + \ldots + a_{n-1} s + a_n)Y = (b_0 s^n + b_1 s^{n-1} + \ldots + b_{n-1} s + b_n)U \tag{3.43}$$

Združimo člene z enako potenco

$$s^n(Y - b_0 U) + s^{n-1}(a_1 Y - b_1 U) + \ldots + s(a_{n-1} Y - b_{n-1} U) + (a_n Y - b_n U) = 0 \tag{3.44}$$

Enačbo (3.44) delimo z s^n, da na levi ostane le Y

$$Y = b_0 U - \frac{1}{s}(a_1 Y - b_1 U) - \ldots - \frac{1}{s^{n-1}}(a_{n-1} Y - b_{n-1} U) - \frac{1}{s^n}(a_n Y - b_n U) \tag{3.45}$$

in jo napišemo v vgnezdeni obliki

$$Y = b_0 U + \frac{1}{s}\{(b_1 U - a_1 Y) + \ldots + \frac{1}{s}[(b_{n-1} U - a_{n-1} Y) + \frac{1}{s}(b_n U - a_n Y)]\} \tag{3.46}$$

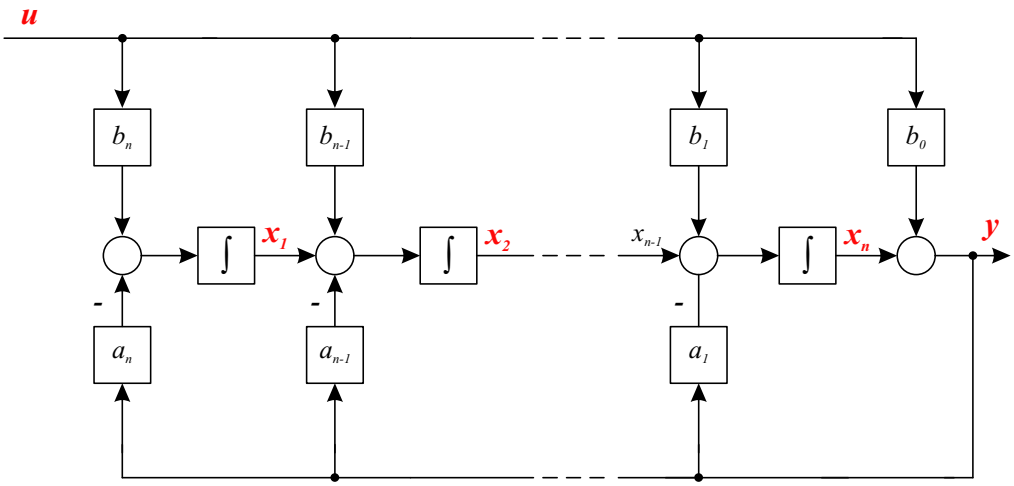
Izberemo spremenljivke stanj

$$\begin{aligned} X_1 &= \frac{1}{s}[b_n U - a_n Y] \\ X_2 &= \frac{1}{s}[(b_{n-1} U - a_{n-1} Y) + X_1] \\ &\vdots \\ X_n &= \frac{1}{s}[(b_1 U - a_1 Y) + X_{n-1}] \end{aligned} \tag{3.47}$$

Izhod določa enačba

$$Y = b_0 U + X_n \tag{3.48}$$

Shemo za realizacijo (simulacijo) prikazuje slika 3.18.



Slika 3.18: Spoznavnostna kanonična oblika

Spremenljivke stanj so ponovno izhodi integratorjev, vendar tokrat v smeri od leve proti desni.

Za zapis v prostoru stanj vstavimo enačbo (3.48) v enačbo (3.47) ter pomnožimo vse enačbe z s.

$$\begin{aligned} sX_1 &= -a_n X_n + (b_n - a_n b_0)U \\ sX_2 &= X_1 - a_{n-1} X_n + (b_{n-1} - a_{n-1} b_0)U \\ &\vdots \\ sX_n &= X_{n-1} - a_1 X_n + (b_1 - a_1 b_0)U \end{aligned} \tag{3.49}$$

Torej je zapis v prostoru stanj v spoznavnostni kanonični obliki

$$\dot{\boldsymbol{x}}(t) = \begin{bmatrix} 0 & 0 & \dots & 0 & -a_n \\ 1 & 0 & \dots & 0 & -a_{n-1} \\ 0 & 1 & \dots & 0 & -a_{n-2} \\ \vdots & \vdots & & \vdots & \vdots \\ 0 & 0 & \dots & 1 & -a_1 \end{bmatrix} \boldsymbol{x}(t) + \begin{bmatrix} b_n - a_n b_0 \\ b_{n-1} - a_{n-1} b_0 \\ b_{n-2} - a_{n-2} b_0 \\ \vdots \\ b_1 - a_1 b_0 \end{bmatrix} u(t) \quad (3.50)$$

$$y(t) = [\,0 \quad 0 \quad \dots \quad 1\,]\,\boldsymbol{x}(t) + b_0 u(t) \quad (3.51)$$

Matrika $\boldsymbol{A}$ je *transponirana Frobeniusova matrika*. Za matriko $\boldsymbol{B}$ ni posebnih zahtev, matrika $\boldsymbol{C}$ pa je v obliki vrstičnega vektorja s samimi ničlami razen enice na zadnjem mestu. Spoznavnostna oblika je zlasti primerna pri načrtovanju observatorjev.

3.6.3 Diagonalna kanonična oblika

Do te kanonične oblike pridemo, če prenosno funkcijo razvrstimo v parcialne ulomke

$$\begin{aligned} \frac{Y(s)}{U(s)} &= \frac{b_0 s^n + b_1 s^{n-1} + \ldots + b_{n-1} s + b_n}{(s-\lambda_1)(s-\lambda_2)\ldots(s-\lambda_n)} = \\ &= b_0 + \frac{c_1}{s-\lambda_1} + \frac{c_2}{s-\lambda_2} + \ldots + \frac{c_n}{s-\lambda_n} \end{aligned} \quad (3.52)$$

λ_i so poli sistema oz. lastne vrednosti matrike $\boldsymbol{A}$. Predpostavljamo, da so različni. Enačbo (3.52) lahko zapišemo v obliko

$$Y = b_0 U + \frac{c_1}{s-\lambda_1} U + \frac{c_2}{s-\lambda_2} U + \ldots + \frac{c_n}{s-\lambda_n} U \quad (3.53)$$

Definiramo spremenljivke stanj

$$X_1(s) = \frac{1}{s-\lambda_1} U(s)$$

$$\begin{aligned} X_2(s) &= \frac{1}{s-\lambda_2}U(s) \\ &\vdots \\ X_n(s) &= \frac{1}{s-\lambda_n}U(s) \end{aligned} \tag{3.54}$$

Enačbe (3.54) preuredimo v obliko

$$\begin{aligned} sX_1(s) &= \lambda_1 X_1(s) + U(s) \\ sX_2(s) &= \lambda_2 X_2(s) + U(s) \\ &\vdots \\ sX_n(s) &= \lambda_n X_n(s) + U(s) \end{aligned} \tag{3.55}$$

Izhodni signal pa je

$$Y = c_1X_1 + c_2X_2 + \ldots + c_nX_n + b_0U \tag{3.56}$$

Enačbe (3.55) in (3.56) določajo zapis v *diagonalni kanonični obliki*

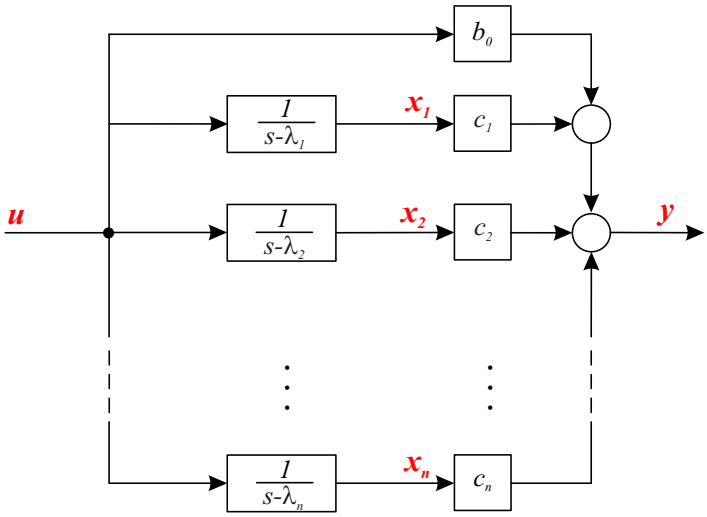
$$\dot{\boldsymbol{x}}(t) = \begin{bmatrix} \lambda_1 & 0 & \ldots & 0 \\ 0 & \lambda_2 & & 0 \\ \ldots & \ldots & & \\ 0 & 0 & \ldots & \lambda_n \end{bmatrix} \boldsymbol{x}(t) + \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix} u(t) \tag{3.57}$$

$$y(t) = [\, c_1 \quad c_2 \quad \ldots \quad c_n \,]\, \boldsymbol{x}(t) + b_0 u(t) \tag{3.58}$$

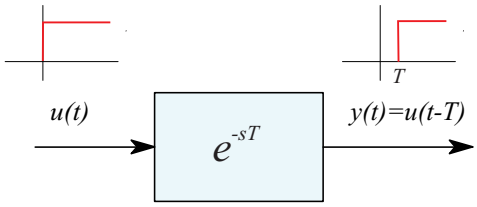
Bločno shemo prikazuje slika 3.19.

3.7 Simulacija sistemov z mrtvim časom

Slika 3.20 prikazuje sistem z idealnim mrtvim časom T.



Slika 3.19: Bločna shema za realizacijo (simulacijo) v diagonalni kanonični obliki



Slika 3.20: Sistem z idealnim mrtvim časom

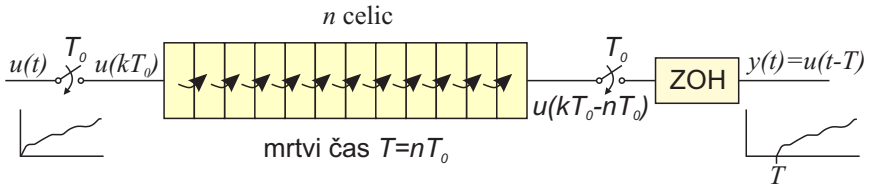
$u(t)$ je vhod, $y(t)$ pa izhod iz sistema. Zaradi teorema zakasnitve pri Laplace-ovi transformaciji je prenosna funkcija torej transcendentna funkcija.

$$\frac{Y(s)}{U(s)} = \frac{\mathcal{L}[y(t)]}{\mathcal{L}[u(t)]} = \frac{\mathcal{L}[u(t-T)]}{\mathcal{L}[u(t)]} = G(s) = e^{-sT} \tag{3.59}$$

Mrtvi čas vnaša tako na digitalnih kot na analognih računalnikih v simulacijskem postopku precej problemov, ki jih je mogoče rešiti le z aproksimacijskimi postopki.

Pri digitalni simulaciji temeljijo postopki običajno na čimbolj pogostem 'vzorčenju' vhodnega signala $u(t)$ in na zakasnjevanju s pomočjo 'premikalnega registra', kakor prikazuje slika 3.21.

Na vhodu 'premikalnega registra' je vzorčevalnik, ki vzorči signal $u(t)$ s časom vzorčenja T_0. Vzorec se zapiše v prvo celico (oz. v prvi element polja) in premakne za eno mesto v desno ob vsakem naslednjem času vzorčenja. Ker ima polje n



Slika 3.21: Izvedba mrtvega časa pri digitalni simulaciji

celic, je na koncu signal zakasnjen za čas $T = nT_0$. Ker uporabljamo mrtvi čas v zveznem sistemu, moramo na izhodu premikalnega polja uporabiti ekstrapolator ničtega reda (zero order hold ZOH), ki iz vzorčenega signala rekonstruira zvezni signal. Tako je signal $y(t) = u(t - T)$ v resnici nezvezni signal s stopničastimi spremembami, vendar se to pri dovolj kratkem času vzorčenja niti ne opazi.

Znani pa so tudi postopki, ki so bili razviti za analogne računalnike in upoštevajo metode razvrstitve transcedentne funkcije e^{-sT} v vrsto. Če se uporabi Taylorjeva vrsta, se izkaže, da potrebujemo za običajno natančnost kar precej členov. Zato so se bolj uveljavile nekatere druge razvrstitve. Najbolj so znane t.i. Padé-jeve aproksimacije 1., 2. in 4. reda.

Aproksimacija 1. reda

$$e^{-sT} \doteq \frac{1 - \frac{T}{2}s}{1 + \frac{T}{2}s} = \frac{-sT + 2}{sT + 2} \tag{3.60}$$

Aproksimacija 2. reda

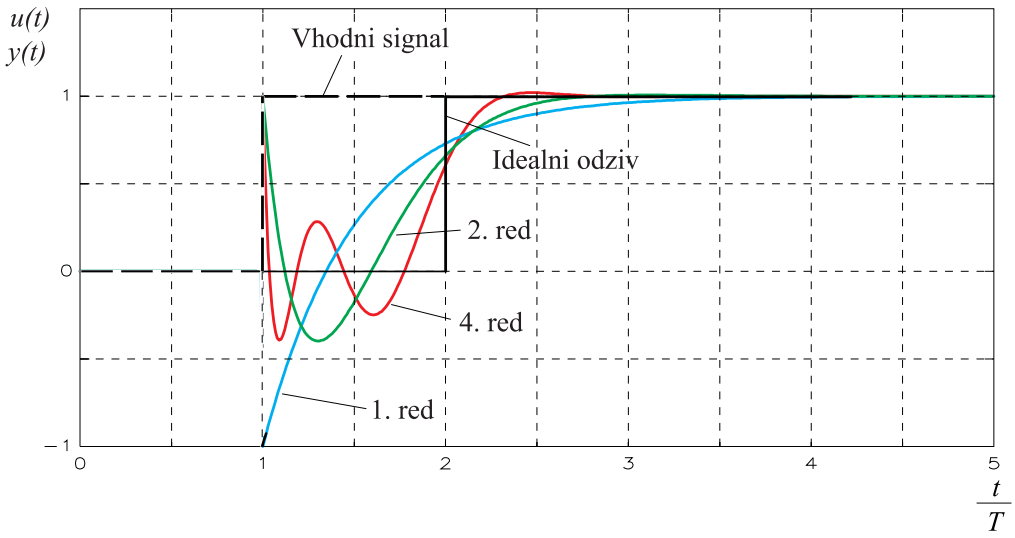
$$e^{-sT} \doteq \frac{(sT)^2 - 6sT + 12}{(sT)^2 + 6sT + 12} \tag{3.61}$$

Aproksimacija 4. reda

$$e^{-sT} \doteq \frac{(sT)^4 - 20(sT)^3 + 180(sT)^2 - 840sT + 1680}{(sT)^4 + 20(sT)^3 + 180(sT)^2 + 840sT + 1680} \tag{3.62}$$

Čim višji je red, boljšo aproksimacijo dobimo. Vse prenosne funkcije imajo lastnost sistemov z neminimalno fazo. Slika 3.22 prikazuje idealni odziv sistema z

mrtvim časom in odzive Padé-jevih aproksimacij 1., 2. in 4. reda. Vhodni signal je enotina stopnica in nastopi v trenutku $\frac{t}{T} = 1$.



Slika 3.22: Odzivi sistema z mrtvim časom pri uporabi Padé-jevih aproksimacij 1., 2. in 4. reda

Padéjeve aproksimacije se uporabljajo tudi v nekaterih postopkih analize modelov v okolju Simulink, npr. pri linearizaciji nelinearnih modelov z mrtvim časom.

Primer 3.10 Mrtvi čas e^{-sT} aproksimirajmo s Padé-jevo aproksimacijo 1. reda

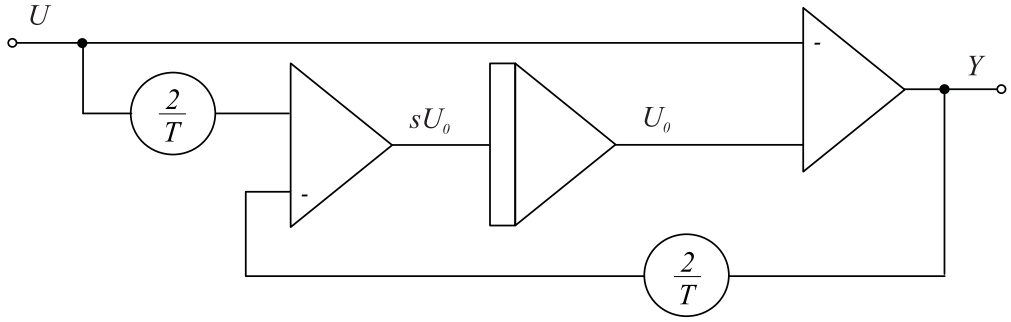
$$G(s) = \frac{Y(s)}{U(s)} = \frac{1 - \frac{T}{2}s}{1 + \frac{T}{2}s} \tag{3.63}$$

Prenosno funkcijo 3.63 simulirajmo s pomočjo vgnezdene metode

$$\begin{aligned} Y(1 + \frac{T}{2}s) &= U(1 - \frac{T}{2}s) \\ s\frac{T}{2}Y &= -s\frac{T}{2}U + U - Y \\ Y &= -U + \frac{1}{s}(\frac{2}{T}U - \frac{2}{T}Y) \\ U_0 &= \frac{1}{s}(\frac{2}{T}U - \frac{2}{T}Y) \end{aligned} \tag{3.64}$$

$$Y = -U + U_0$$

Simulacijsko shemo prikazuje slika 3.23, rezultate simulacije pri stopničastem vhodnem signalu pa ena od krivulj na sliki 3.22.



Slika 3.23: Simulacijska shema sistema, ki aproksimira mrtvi čas po Padé-jevi metodi 1. reda

□

3.8 Simulacija kompleksnih sistemov

Pri simulaciji kompleksnih sistemov je treba že v fazi modeliranja upoštevati principe modularnosti, kar pomeni, da večje sklope realiziramo z manjšimi zaključenimi sklopi (podmodeli, prenosne funkcije, ...). Medtem ko je pri analitičnih metodah analize in načrtovanja potrebno take podsklope združevati oz. poenostavljati (npr. poenostavljanje bločnih diagramov s pomočjo algebre bločnih diagramov), pa je pomembna prednost pri simulaciji v tem, da vsak podsklop ločeno simuliramo po eni izmed obravnavanih metod in nato posamezne podsklope ustrezno povežemo. Na ta način so dosegljive tudi vse spremenljivke podsklopov, simulacijske sheme so pregledne in uspešno služijo tudi za dokumentacijo.

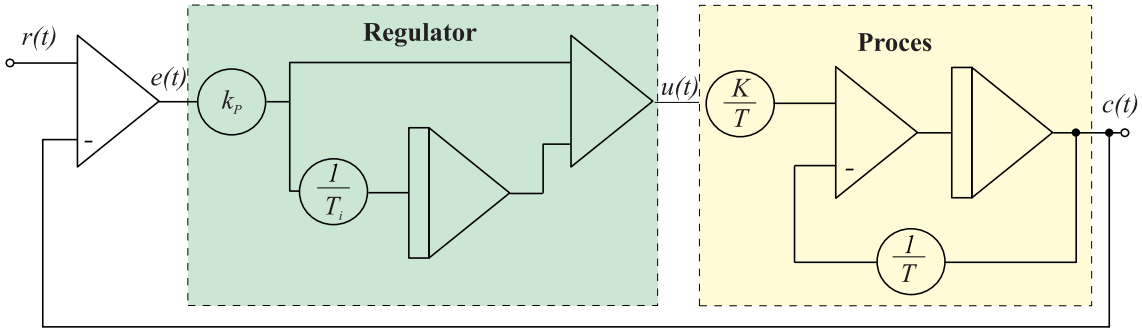
Primer 3.11 Regulacijski sistem vsebuje regulator PI

$$G_R(s) = \frac{U(s)}{E(s)} = K_P(1 + \frac{1}{T_I s}) \tag{3.65}$$

in proces 1. reda

$$G_P(s) = \frac{C(s)}{U(s)} = \frac{K}{Ts+1} \tag{3.66}$$

Ob upoštevanju nazornosti in modularnosti pri simulaciji kompleksnejših sistemov dobimo simulacijsko shemo, ki jo prikazuje slika 3.24. Slika pregledno kaže regulacijsko strukturo. Če bi npr. simulirali regulacijski sistem s pomočjo ene zaprtozančne prenosne funkcije $\frac{C(s)}{R(s)}$, ne bi bila dostopna regulirna veličina $u(t)$.



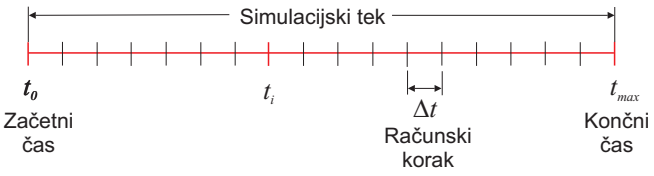
Slika 3.24: Simulacijska shema regulacijskega sistema

□

3.9 Koncept digitalne simulacije

Metode, ki smo jih spoznali v prejšnjih podpoglavjih, omogočajo programiranje problema na analognem računalniku (ob predhodnem normiranju in skaliranju). Prav tako te metode omogočajo, da s pomočjo simulacijske sheme direktno napišemo program v simulacijskem jeziku. Če pa želimo sami programirati simulacijo v nekem splošnonamenskem programskem jeziku, (npr. Visual Basic, C++, Fortran, Java, Matlab ...) je potrebno poznati koncept digitalne simulacije zveznih dinamičnih sistemov, kajti integrator je potrebno realizirati z numeričnim postopkom, paralelni sistem pa računati zaporedno.

Da lahko simuliramo zvezni sistem na digitalnem računalniku, moramo neodvisno spremenljivko simulacije (običajno čas) diskretizirati, tako da diferencialne enačbe postanejo diferenčne enačbe. Ker so vse spremenljivke modela odvisne od neodvisne spremenljivke t, so torej tudi definirane samo v diskretnih vrednostih (trenutkih) neodvisne spremenljivke (slika 3.25).



Slika 3.25: Diskretizacija po času

Če simulacija poteka s konstantnim prirastkom neodvisne spremenljivke Δt (*računski korak*), lahko trenutno vrednost neodvisne spremenljivke izrazimo kot

$$t_i = t_0 + i\Delta t \qquad\qquad i = 0, 1, 2, \ldots, i_{max} \tag{3.67}$$

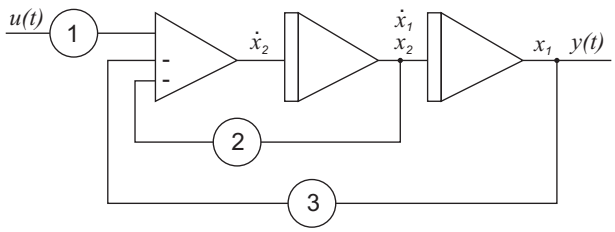
Torej simulacija prične v trenutku $t = t_0$ (*začetni čas simulacije*) in se konča v trenutku $t = t_{max} = t_0 + i_{max}\Delta t$ (*končni čas simulacije*). Dogajanju med časoma t_0 in t_{max} pa pravimo *simulacijski tek*.

Vsi sistemi (orodja) za digitalno simulacijo so osnovani na zapisu sistema v prostoru stanj, t.j. s sistemom diferencialnih enačb 1. reda. Uporabniku pa se tega običajno niti ni potrebno zavedati, saj se ustrezni zapis avtomatično vzpostavi. Izhodišče je torej vektorska diferencialna enačba

$$\dot{\boldsymbol{x}}(t) = \boldsymbol{f}(t, \boldsymbol{x}(t)) \tag{3.68}$$

Vsa stanja je potrebno hraniti v vektorju $\boldsymbol{x}(t)$, vse odvode pa v vektorju $\dot{\boldsymbol{x}}(t)$.

Simulacijska okolja sicer ne zahtevajo takega vnosa, ampak avtomatsko zgenerirajo zapis v prostoru stanj v začetku procesiranja običajno iz grafičnega ali tekstovnega zapisa. Postopek prikazuje slika 3.26.



Slika 3.26: Sistem 2. reda

Izhodi integratorjev sta stanji x_1 in x_2. Iz sheme napišeno enačbi

$$\begin{aligned} \dot{x}_1 &= x_2 \\ \dot{x}_2 &= u - 3x_1 - 2x_2 \end{aligned} \tag{3.69}$$

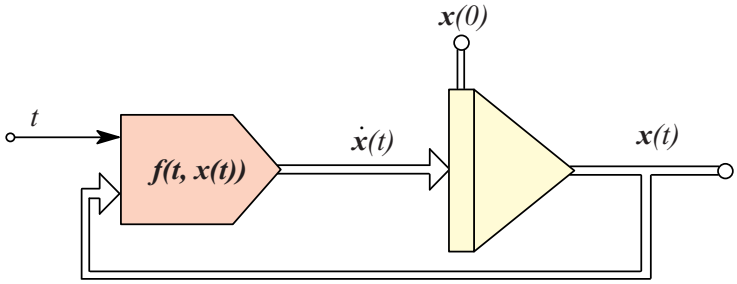
oz. matrično obliko

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -3 & -2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u$$

Pri uporabi indirektnega načina je torej prvi korak že opravljen. Drugi korak pa se realizira z enim vektorskim integratorjem

$$\boldsymbol{x}(t) = \int \dot{\boldsymbol{x}}(t)dt = \int \boldsymbol{f}(t, \boldsymbol{x}(t))dt \tag{3.70}$$

Z upoštevanjem tretjega koraka indirektnega postopka dobimo simulacijsko shemo, ki jo prikazuje slika 3.27.



Slika 3.27: Osnovna simulacijska shema

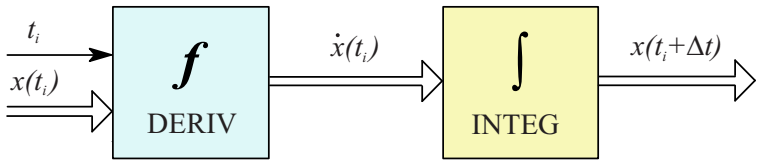
Da lahko shemo na sliki 3.27 simuliramo na digitalnem računalniku, mora imeti vsak simulacijski sistem tri pomembne značilnosti, ki jih bomo opisali v naslednjih podpoglavjih.

3.9.1 Pretvorba paralelne strukture v zaporedno

Ker splošnonamenski sekvenčni digitalni računalniki ne zmorejo paralelnega računanja kot npr. analogni računalniki, *je potrebno vse zanke, ki jih prikazuje*

slika 3.27 *na nek način prekiniti, tako da je možno spremenljivke računati zaporedno*. Zanke vedno prekinemo na izhodih t.i. spominskih blokov (včasih jim pravimo tudi bloki z zakasnitvenim atributom), katerih izhodi so običajno stanja sistema. V splošnem je to lahko vsak blok, ki ima lastnost, da trenutna vrednost njegovega izhoda ni odvisna od vrednosti vhoda v istem trenutku (npr. integrator, blok z mrtvim časom, diskretna zakasnitev, zadrževalnik ničtega reda (ZOH), zakasnilni blok, prenosna funkcija, ki ima red števca manjši od reda imenovalca,...). Najpogosteje pa so taki bloki integratorji (glej enačbo (3.71) v primeru Euler-jevega algoritma).

Če zanke sistema, ki ga prikazuje slika 3.27, prekinemo na izhodu vektorskega integratorja in če zamenjamo zvezno integracijo z numeričnim postopkom, dobimo shemo, ki jo prikazuje slika 3.28. Ta shema jasno prikazuje koncept digitalne simulacije zveznih dinamičnih sistemov.



Slika 3.28: Osnovni koncept digitalne simulacije

Na začetku simulacijskega teka se vedno izvede inicializacija, ki vključuje tudi ovrednotenje začetnih vrednosti stanj ($\boldsymbol{x}(t_0)$).

Simulacijski algoritem lahko opišemo z *dvokoračno iteracijo* v vsakem računskem koraku. V prvem koraku se izračunajo vsi odvodi spremenljivk stanja ($\dot{\boldsymbol{x}}(t_i)$) iz trenutnih vrednosti spremenljivk stanja ($\boldsymbol{x}(t_i)$) in iz vrednosti neodvisne spremenljivke simulacije (t_i). Odvodi se izračunajo v podprogramu, ki ga bomo imenovali DERIV in vključuje izračun funkcije $\boldsymbol{f}(\boldsymbol{x}, t)$ oz. ustrezno razvrščene enačbe modela. V drugem koraku se v podprogramu, ki ga bomo imenovali INTEG, izvrši integracijski postopek, ki izračuna stanja za naslednji računski korak ($\boldsymbol{x}(t_i + \Delta t)$).

Tako poenostavljena dvokoračna iteracija velja samo pri uporabi Euler-jeve integracijske metode.

3.9.2 Vrstni algoritem

Nanaša se na algebrajske izraze v modulu `DERIV` oz. na izračun odvodov $\boldsymbol{f}(t, \boldsymbol{x}(t))$. To so enačbe modela, ki se med simulacijo izvajajo enkrat ali celo večkrat v vsakem računskem koraku. Ko neka enačba pride na vrsto z namenom, da se izračuna spremenljivka na levi strani enačaja, morajo bit predhodno izračunani vsi členi na desni strani enačbe. Torej mora desna stran vsebovati:

- konstante,
- stanja,
- vhodne signale,
- spremenljivke, izračunane v predhodnih stavkih.

Primer:

```
a=3
u=step(t)
v=sin(t)
z=u+v
x1dot=z+x2+a
X2dot=v+2*x1-x2
```

Če taka razvrstitev ni možna, govorimo o **algebrajski zanki**, ki onemogoči ali zelo upočasni simulacijo (pogosto pa nastane zaradi slabega modeliranja). Primer:

```
x1=g(x2, ...)
x2=h(x1, ...)
```

3.9.3 Numerična integracija

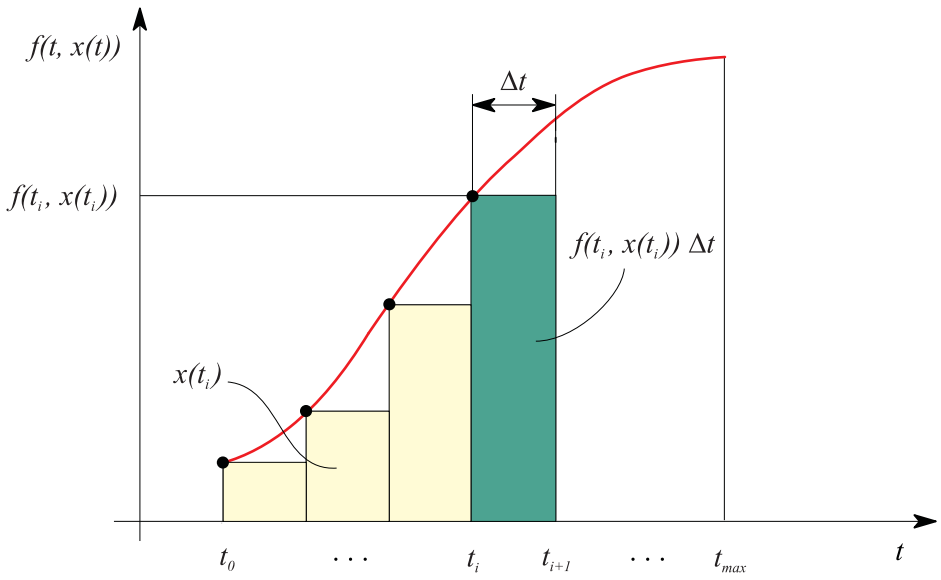
Enačbo (3.70) *je potrebno rešiti z numeričnim integracijskim postopkom.* Integracija je osrednja operacija vsakega simulacijskega orodja. Obstajajo številne

metode, ki so različno primerne za različne vrste modelov. Tako poznamo enokoračne in večkoračne metode, implicitne in eksplicitne metode, metode za toge sisteme (z zelo različnimi časovnimi konstantami), itd. Ti postopki omogočajo veliko natančnost in numerično zanesljivost in so zato zelo kompleksni.

Princip numerične integracije pa najlepše prikazuje najenostavnejši postopek, t.j. Euler-jev algoritem, ki opisuje reševanje enačbe (3.70) v obliki

$$\boldsymbol{x}(t_{i+1}) = \boldsymbol{x}(t_i + \Delta t) = \boldsymbol{x}(t_i) + \boldsymbol{f}(t_i, \boldsymbol{x}(t_i))\Delta t \tag{3.71}$$

Metoda predpostavlja stopničasto aproksimacijo funkcije $\boldsymbol{f}(t, \boldsymbol{x}(t))$, ustrezen integral pa je vsota pravokotnikov pod krivuljo $\boldsymbol{f}(t, \boldsymbol{x}(t))$. Postopek prikazuje slika 3.29.



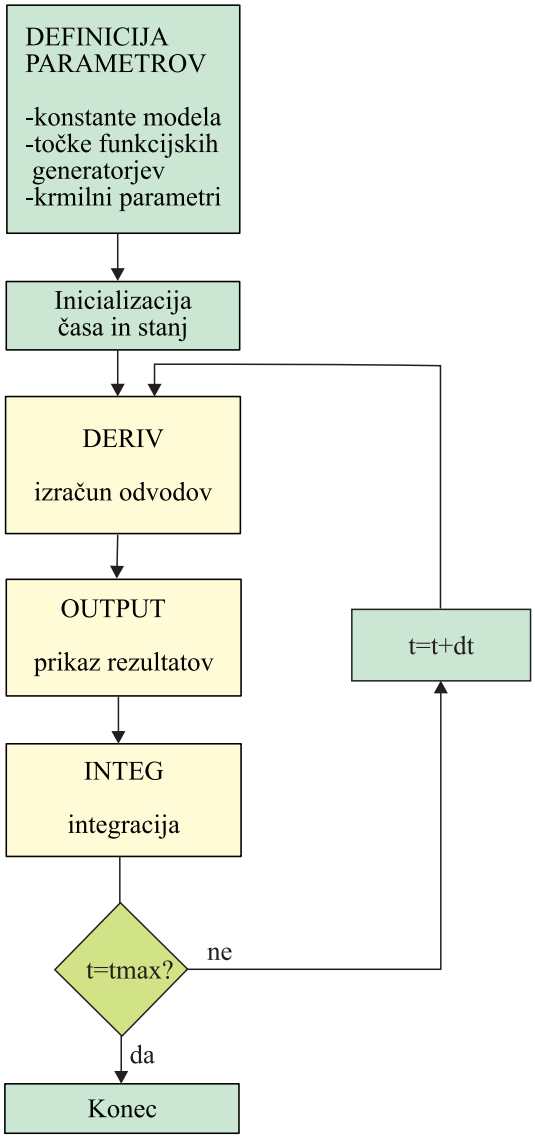
Slika 3.29: Integracija z Euler-jevo metodo

Integracijska metoda torej uporablja trenutne vrednosti stanj $\boldsymbol{x}(t_i)$ in odvodov $\boldsymbol{f}(t_i, \boldsymbol{x}(t_i))$ za ovrednotenje stanj za naslednji računski korak $\boldsymbol{x}(t_i + \Delta t)$.

Eulerjeva metoda je torej zelo nazorna za izobraževanje. Ker pa je tudi zelo učinkovita glede na porabo računalniškega časa, pa se včasih uporablja za simulacijo hitrih procesov v realnem času. Seveda pa je neprimerna, če je model numerično zahteven.

3.9.4 Izvedba programa

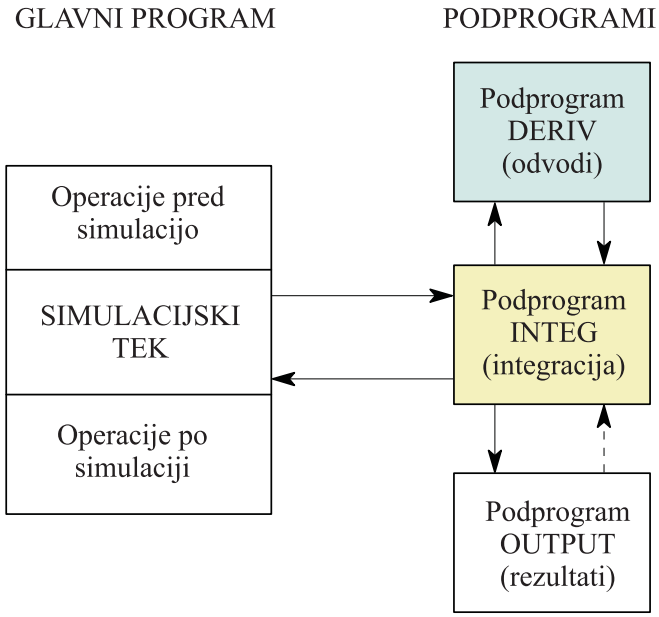
Kot smo opisali, je v primeru uporabe Eulerjeve integracijske metode simulacija dvokoračna iteracija dveh programskih modulov - `DERIV` in `INTEG`. Ustrezno programsko izvedbo prikazuje slika 3.30.



Slika 3.30: Diagram poteka simulacijskega programa pri Eulerjevi integracijski metodi

Pri uporabi numerično bolj izpopolnjenih postopkov potrebuje integracijska

metoda na enem računskem koraku več ovrednotenj odvodov, torej je potrebnih več klicev podprograma DERIV. Če ustrezno razširimo osnovno strukturo programa, dobimo običajno izvedbo (v orodjih, simulacijskih paketih), kot prikazuje slika 3.31. Pred simulacijskim tekom se izvedejo določene operacije (npr. inicializacija stanj). Simulacijski tek pa se izvrši s klicem podprograma za integracijo (INTEG), ki po potrebi kliče podprogram za izračun odvodov spremenljivk stanja (DERIV). V trenutkih, ki jih definira uporabnik, pa integracijski podprogram kliče podprogram (OUTPUT), ki skrbi za ustrezno posredovanje rezultatov simulacije. Po izpolnjenem pogoju za končanje simulacijskega teka se izvršijo še morebitne operacije po koncu simulacijskega teka.

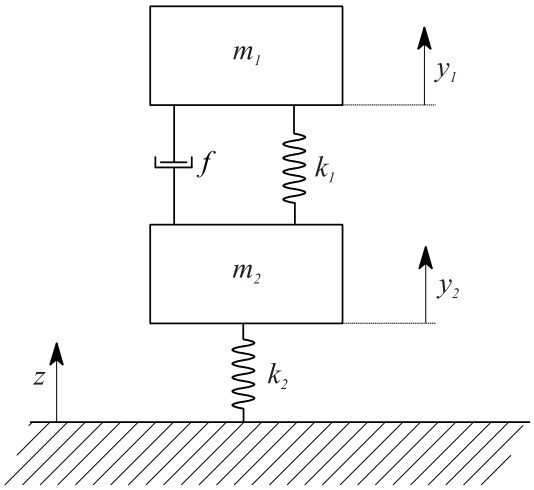


Slika 3.31: Običajna struktura simulacijskega programa

3.10 Rešene naloge

Naloga 3.1 Spremenjen model avtomobilskega vzmetenja

Kot prvi primer vzemimo nekoliko spremenjen model avtomobilskega vzmetenja glede na primer 3.2. V tem primeru ne zanemarimo mase kolesa in polovice osi. Sistem vzbujamo z oviro na podlagi (npr. če kolo zapelje na pločnik). Ustrezen mehanski model prikazuje slika 3.32.



Slika 3.32: Mehanski model avtomobilskega vzmetenja

Z y_1 smo označili gibanje karoserije, z y_2 gibanje kolesa, z z pa oviro, ki je v našem primeru stopničasta sprememba, pri čemer velikost stopnice določa višina pločnika. Matematični model opišemo z enačbama

$$\begin{aligned} m_1\ddot{y}_1 &= -f(\dot{y}_1 - \dot{y}_2) - k_1(y_1 - y_2) \\ m_2\ddot{y}_2 &= -f(\dot{y}_2 - \dot{y}_1) - k_1(y_2 - y_1) - k_2(y_2 - z) \end{aligned} \tag{3.72}$$

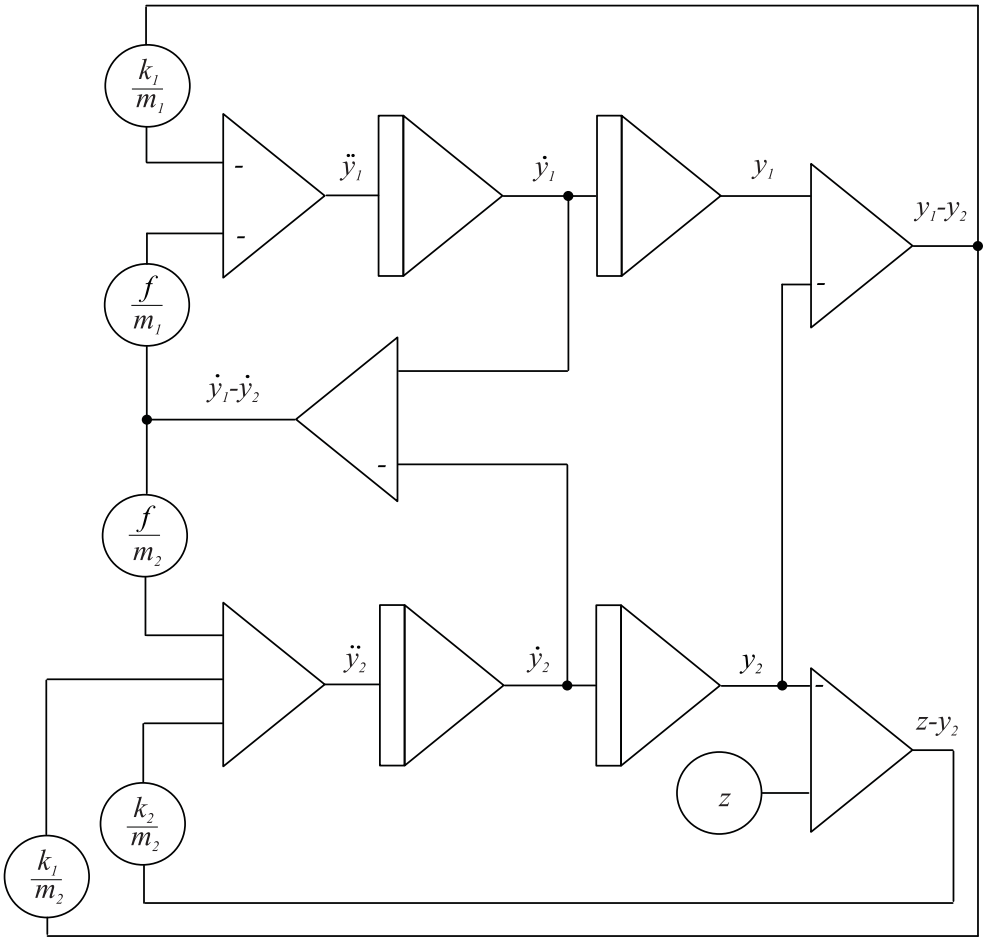
Z uporabo indirektne metode simuliramo model v naslednjih korakih:

1. korak: preureditev enačb

$$\ddot{y}_1 = -\frac{f}{m_1}(\dot{y}_1 - \dot{y}_2) - \frac{k_1}{m_1}(y_1 - y_2) \tag{3.73}$$

$$\ddot{y}_2 = \frac{f}{m_2}(\dot{y}_1 - \dot{y}_2) + \frac{k_1}{m_2}(y_1 - y_2) + \frac{k_2}{m_2}(z - y_2)$$

2. in 3. korak: generacija sheme, ki jo prikazuje slika 3.33.



Slika 3.33: Simulacijska shema modela avtomobilskega vzmetenja

Naloga 3.2 Uporaba implicitne metode

V tem primeru bomo na implicitni način realizirali signal

$$y = At^2e^{-bt} \tag{3.74}$$

Z odvajanjem enačbe (3.74) dobimo

$$\dot{y} = 2Ate^{-bt} - At^2be^{-bt} = 2Ate^{-bt} - by \tag{3.75}$$

Drugi člen v enačbi (3.75) smo dobili z upoštevanjem enačbe (3.74).

Enačbo (3.75) ponovno odvajamo

$$\ddot{y} = 2Ae^{-bt} - 2Abte^{-bt} - b\dot{y} \tag{3.76}$$

Drugi člen na desni strani enačbe (3.76) izrazimo s pomočjo enačbe (3.75)

$$2Ate^{-bt} = \dot{y} + by \tag{3.77}$$

tako da dobimo izraz

$$\ddot{y} = 2Ae^{-bt} - b\dot{y} - b^2y - b\dot{y} \tag{3.78}$$

Na tem mestu bi lahko končali s postopkom, saj bi lahko člen $2Ae^{-bt}$ realizirali z implicitnim postopkom, celotno enačbo (3.78) pa z indirektnim načinom. Če pa nadaljujemo z odvajanjem

$$\dddot{y} = -2Abe^{-bt} - 2b\ddot{y} - b^2\dot{y} \tag{3.79}$$

in nadomestimo prvi člen na desni strani enačbe (3.79) iz enačbe (3.78)

$$2Ae^{-bt} = \ddot{y} + 2b\dot{y} + b^2y \tag{3.80}$$

dobimo končno obliko

$$\dddot{y} = -3b\ddot{y} - 3b^2\dot{y} - b^3y \tag{3.81}$$

z začetnimi pogoji

$$y(0) = 0 \qquad \dot{y}(0) = 0 \qquad \ddot{y}(0) = 2A \tag{3.82}$$

Enačbo (3.81) simuliramo z indirektno metodo.

Naloga 3.3 Mathieu-jeva diferencialna enačba

V tem primeru bomo pokazali, da je indirektna metoda primerna tudi za reševanje diferencialnih enačb s spremenljivimi koeficienti. Te koeficiente, ki so običajno funkcije neodvisne spremenljivke, je potrebno dodatno generirati.

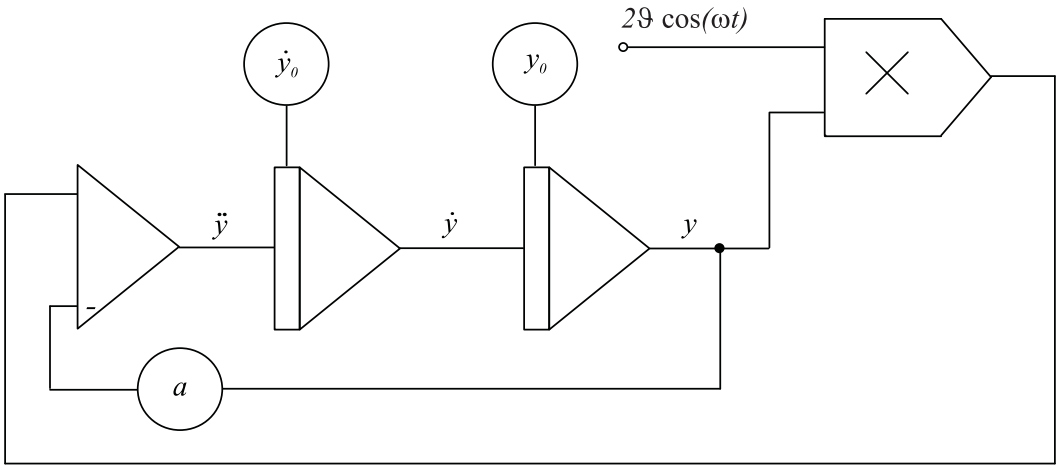
Mathieu-jeva enačba ima obliko

$$\ddot{y} + [a - 2\vartheta\cos(\omega t)]y = 0 \qquad y(0) = y_0 \qquad \dot{y}(0) = \dot{y}_0 \tag{3.83}$$

in se uporablja pri študiju frekvenčno moduliranih nihanj. Enačbo (3.83) preuredimo za indirektno metodo

$$\ddot{y} = -ay + 2\vartheta\cos(\omega t)y \qquad y(0) = y_0 \qquad \dot{y}(0) = \dot{y}_0 \tag{3.84}$$

Simulacijsko shemo prikazuje slika 3.34.



Slika 3.34: Simulacijska shema za Mathieu-jevo diferencialno enačbo

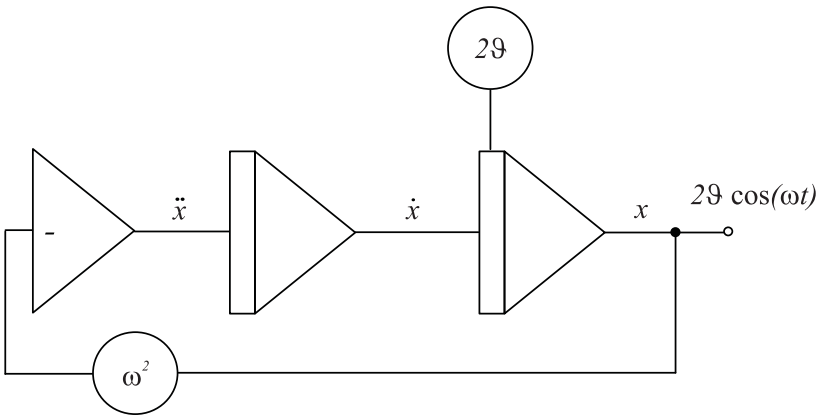
Časovno spremenljivi koeficient $2\vartheta\cos(\omega t)$ je zlasti pri simulaciji z analognim računalnikom smiselno realizirati na implicitni način

$$\begin{aligned} x &= 2\vartheta\cos(\omega t) \\ \dot{x} &= -2\vartheta\omega\sin(\omega t) \\ \ddot{x} &= -2\vartheta\omega^2\cos(\omega t) = -\omega^2 x \end{aligned} \tag{3.85}$$

Koeficient $2\vartheta\cos(\omega t)$ se torej lahko realizira s simulacijo sistema

$$\ddot{x} = -\omega^2 x \qquad x(0) = 2\vartheta \qquad \dot{x}(0) = 0 \tag{3.86}$$

z indirektnim načinom. Simulacijsko shemo prikazuje slika 3.35.



Slika 3.35: Generacija izraza $2\vartheta\cos(\omega t)$ z implicitno metodo

Naloga 3.4 Simulacija prenosne funkcije

Če je model predstavljen s prenosno funkcijo v faktorizirani obliki ali s produktom več prenosnih funkcij, lahko pomeni, da je le ta sestavljen iz več podmodelov, ki opisujejo zaključene funkcionalne sklope. Take podmodele ločeno simuliramo in jih na koncu ustrezno povežemo.[1]

Realni proces opisuje model

[1]Metoda zaporedne razčlenitve

$$G_P(s) = \frac{Y(s)}{U(s)} = \frac{1-4s}{(1+4s)(1+10s)} \tag{3.87}$$

ki ga lahko predstavimo kot zaporedno vezavo dveh prenosnih funkcij

$$G_P(s) = G_1(s)G_2(s) \tag{3.88}$$

$$\begin{aligned} G_1(s) &= \frac{1-4s}{1+4s} \\ G_2(s) &= \frac{1}{1+10s} \end{aligned} \tag{3.89}$$

Delitev prenosne funkcije (3.87) na funkciji $G_1(s)$ in $G_2(s)$ smo naredili na osnovi fizikalnega poznavanja realnega sistema. Prenosna funkcija $G_1(s)$ je namreč Padé-jeva aproksimacija zakasnitve (mrtvega časa) $2s$. Prenosno funkcijo $G_1(s)$ realizirajmo po delitveni metodi

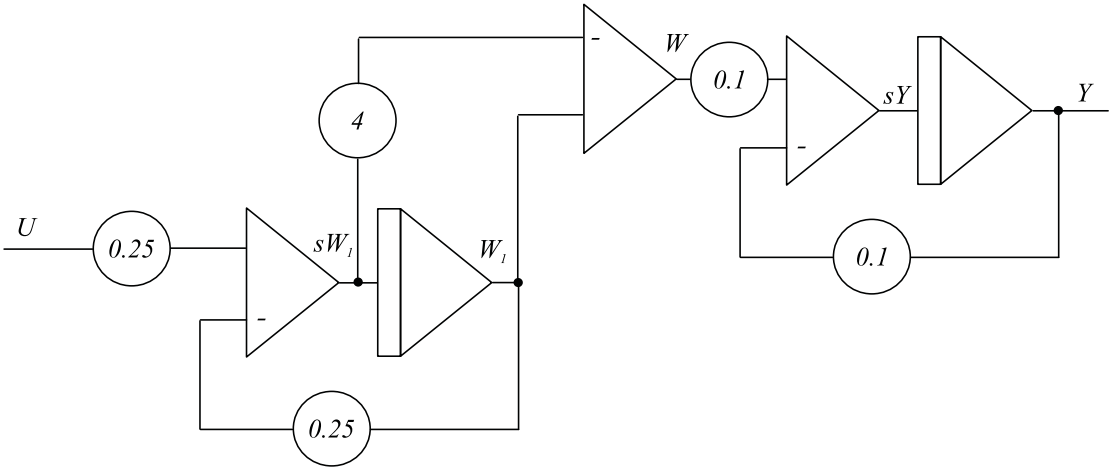
$$\begin{aligned} G_1(s) = \frac{W}{U} &= \frac{1-4s}{1+4s} \\ \frac{W_1}{U} &= \frac{1}{1+4s} \\ sW_1 &= 0.25U - 0.25W_1 \\ \frac{W}{W_1} &= 1-4s \\ W &= W_1 - 4sW_1 \end{aligned} \tag{3.90}$$

Prenosno funkcijo $G_2(s)$ pa realiziramo po vgnezdeni metodi[2]

$$\begin{aligned} G_2(s) = \frac{Y}{W} &= \frac{1}{1+10s} \\ Y &= \frac{1}{s}(0.1W - 0.1Y) \end{aligned} \tag{3.91}$$

[2]Ker prenosna funkcija nima končne ničle, v shemi ni razlike med vgnezdeno in delitveno metodo

Slika 3.36 prikazuje simulacijsko shemo.



Slika 3.36: Shema za simulacijo prenosne funkcije v faktorizirani obliki

Naloga 3.5 Simulacija diferencialne enačbe s kompleksnimi koeficienti

Diferencialna enačba ima obliko

$$\ddot{x} + (a + jb)\dot{x} + (c + jd)x = 0 \tag{3.92}$$
$$x(0) = e + jf \qquad \dot{x}(0) = g + jh$$

Ker so koeficienti kompleksni, moramo tudi rešitev diferencialne enačbe predpostaviti v kompleksni obliki

$$x(t) = x_r(t) + jx_i(t) \tag{3.93}$$

V tem primeru je sistem vzbujan s kompleksnimi začetnimi pogoji, lahko pa bi bil seveda vzbujan z realno ali kompleksno vhodno funkcijo, ki bi jo podajal izraz na desni strani enačbe (3.92). Enačbo (3.92) preuredimo v obliko

$$\ddot{x}_r + j\ddot{x}_i + (a + jb)(\dot{x}_r + j\dot{x}_i) + (c + jd)(x_r + jx_i) = 0 \tag{3.94}$$

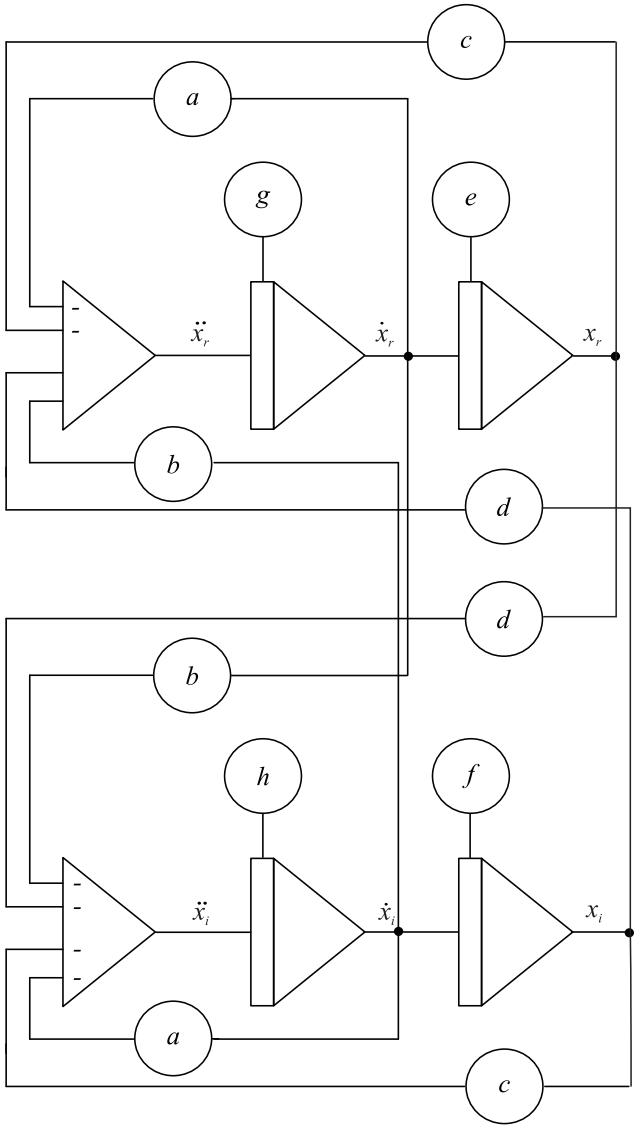
Ker je enačba (3.94) kompleksna, mora veljati ločeno za realni in imaginarni del, torej dobimo sistem dveh realnih diferencialnih enačb

$$\begin{aligned} \ddot{x}_r + a\dot{x}_r - b\dot{x}_i + cx_r - dx_i &= 0 \\ x_r(0) = e \qquad\qquad \dot{x}_r(0) &= g \\ \ddot{x}_i + b\dot{x}_r + a\dot{x}_i + dx_r + cx_i &= 0 \\ x_i(0) = f \qquad\qquad \dot{x}_i &= h \end{aligned} \tag{3.95}$$

Obe enačbi rešimo na običajni način z uporabo indirektne metode. Zato preuredimo enačbi (3.95) v obliko

$$\begin{aligned} \ddot{x}_r &= -a\dot{x}_r + b\dot{x}_i - cx_r + dx_i \\ \ddot{x}_i &= -b\dot{x}_r - a\dot{x}_i - dx_r - cx_i \end{aligned} \tag{3.96}$$

Simulacijsko shemo prikazuje slika 3.37.



Slika 3.37: Simulacijska shema za reševanje diferencialne enačbe s kompleksnimi koeficienti

4.

Orodja za simulacijo dinamičnih sistemov

Metode za simulacijo zveznih dinamičnih sistemov lahko učinkovito uporabljamo le v povezavi z modernimi računalniškimi orodji. V tem poglavju bomo opisali osnovne lastnosti in razvrstitve simulacijskih sistemov in simulacijskih jezikov. V zadnjem delu bomo z dvema primeroma prikazali, kako pridemo do realnega modela (programa), če uporabimo simulacijski jezik tipa CSSL.

4.1 Osnovne lastnosti in razvrstitev simulacijskih sistemov

Moderna simulacijska orodja omogočajo simulacijo dinamičnih modelov na tak način, da se uporabnik lahko osredotoči na samo problematiko (modeliranje, simulacija), ne pa na programiranje problema. Posamezne vrste orodij se z uporabniškega vidika med seboj razlikujejo glede na:

- potreben čas, da postane uporabnik vešč simulacijskega orodja,
- uporabniško prijaznost,
- potreben čas za razvoj simulacijskega modela,
- možnosti pri spreminjanju simulacijskega modela,

- sposobnosti pri izvajanju simulacije (hitrost, interaktivnost, numerična robustnost, možnost simulacije v realnem času, ...),
- možnosti dokumentiranja modelov, rezultatov simulacije in dr.

Glede na uporabljeno programsko in materialno opremo lahko razdelimo moderna simulacijska orodja v tri skupine, t.j. v

- simulacijske sisteme na splošnonamenskih digitalnih računalnikih,
- simulacijske sisteme na namenskih digitalnih računalnikih in
- analogno - hibridne sisteme.

V prvo skupino prištevamo simulacijske sisteme, ki delujejo na tistih digitalnih računalnikih, ki delujejo po Von Neumann-ovem principu (SISD - single instruction, single data). V drugo skupino uvrščamo sposobne kompleksne računalniške sisteme, ki temeljijo na vektorskem (SIMD - single instruction, multiple data) ali na paralelnem izvrševanju operacij (MIMD - multiple instruction, multiple data) in v zvezi s tem potrebujejo ustrezno materialno in programsko opremo. Uporabljajo se za simulacijo velikih in zahtevnih problemov, predvsem pa so primerni za simulacijo v realnem času. V tretjo skupino pa sodijo analogno - hibridni sistemi, ki so z multiprocesorskim sistemom SIMSTAR ter z nekaterimi manjšimi in predvsem cenejšimi sistemi z mikroračunalniškim upravljanjem v osemdesetih letih še enkrat oživeli, danes pa se uporabljajo le še izjemoma v zelo specifičnih simulacijah (simulacija v realnem času s HIL, simulatorji v vojski, ...).

Simulacijski sistemi na splošnonamenskih računalnikih predstavljajo težišče nadaljnje obravnave.

4.1.1 Simulacijski sistemi na splošnonamenskih digitalnih računalnikih

Digitalni simulacijski sistemi na splošnonamenskih računalnikih so povzročili velik razmah simulacije kot moderne metodologije v osemdesetih letih. Namestitev ustreznega simulacijskega sistema na enem ali več računalnikih istega tipa ali celo na računalnikih različnih tipov je daleč najcenejša možnost, ki je zadovoljiva tako za pedagoške namene kot tudi za reševanje raznih znanstveno raziskovalnih

in razvojnih problemov. Na ta način navadno ni možno simulirati v realnem času zaradi omejene hitrosti splošnonamenskih računalnikov.

Schmidt (Schmidt, 1986) je priporočil razdelitev digitalnih simulacijskih sistemov na

- simulacijske pakete in
- simulacijske jezike.

Simulacijski paketi predstavljajo starejšo obliko digitalnih simulacijskih sistemov. Sestavljeni so iz glavnega programa in knjižnice podprogramov. Uporabnik mora v jeziku simulacijskega paketa (Visual Basic, C++, Fortran, Java, Matlab) opisati strukturo modela, pri tem pa lahko kliče vnaprej programirane podprograme iz knjižnice ali dodaja svoje programe. Prednost takega sistema je v tem, da ima uporabnik na voljo izvirni program za opis modela, ki ga dobro obvlada, kar daje določeno fleksibilnost. Slabost takega simulacijskega sistema pa je v tem, da mora uporabnik poznati vsaj bistvene značilnosti numeričnih postopkov (integracijskih metod), obvladati pa mora tudi programiranje, pri čemer obstaja velika verjetnost napak.

Zelo znan je paket ODEPACK (Ordinary Differential Equation PACKage) (Hindmarsh, 1983), ki je sestavljen iz modulov za reševanje navadnih diferencialnih enačb v jeziku FORTRAN. Enačbe modela je možno podati v eksplicitni ($\dot{x} = f(x,t)$) ali implicitni obliki (npr. $F(\dot{x}, x, t) = 0$). Paket vsebuje tudi podprograme za reševanje togih sistemov. Znan je tudi paket DASSL (Differential / Algebraic System SoLver), ki omogoča numerično reševanje implicitnih sistemov diferencialnih in algebrajskih enačb.

Simulacijski jeziki pa predstavljajo sodobne in uporabnejše simulacijske sisteme. Uporabnik mora podati model in določene pogoje za izvrševanje simulacije na način, ki ga predpisuje sintaksa uporabljenega simulacijskega jezika.

Pri delu s simulacijskim jezikom je uporabnik sicer nekoliko manj fleksibilen kot s simulacijskim paketom, vendar pa v prevajalniško orientiranih jezikih tudi lahko dodaja svoje operacije. Prednost simulacijskega jezika je v bistveno večji enostavnosti pri uporabi. Uporabniku ni potrebno poznati višjega programskega jezika temveč le enostavnejši simulacijski jezik. Za neizkušenega uporabnika je simulacijski jezik bistveno primernejši.

Primerjava paketov in jezikov vodi do naslednjih zaključkov:

- Neizkušen uporabnik, ki bi rad hitro in enostavno rešil standardne simulacijske probleme, naj uporabi simulacijski jezik.

- Izjemoma je zelo komplicirane modele, ki vključujejo tudi nestandardne pristope, smiselno simulirati s simulacijskimi paketi.

Sodobni simulacijski sistemi omogočajo podajanje modela z grafičnim urejevalnikom. Čeprav izraz simulacijski jezik ob takem podajanju modela ni več najprimernejši, ker uporabnik ne piše programa v sintaksi simulacijskega jezika, pa večina literature tudi tak sistem imenuje simulacijski jezik. Nekateri sistemi iz grafično podanega modela generirajo program v simulacijskem jeziku (npr. okolja Modelica), nekateri pa ga prevajajo v druge oblike ali pa interpretersko izvajajo simulacijo (npr. Matlab-Simulink).

V nadaljevanju bomo od digitalnih simulacijskih sistemov obravnavali le simulacijske jezike.

4.1.2 Simulacijski sistemi na namenskih digitalnih računalnikih

Simulacija dinamičnih sistemov spada med operacije, ki zahtevajo največ izračunavanj in predvsem za delo v realnem času najsposobnejše računalnike. Zato je simulacija skupaj s področjem obdelave signalov najbolj vplivala na razvoj ustrezne namenske materialne in programske opreme.

Glavna omejitev konvencionalnih digitalnih računalnikov je Von Neumann-ov princip, ki temelji na istočasnem procesiranju enega podatka z eno instrukcijo (SISD - single instruction, single data). Sodobnejši supermini in superračunalniki so razširili svoje zmožnosti s t.i. vektorskim procesiranjem, kjer lahko ena operacija deluje nad nizom podatkov (SIMD - single instruction, multiple data). Superračunalniki so danes najsposobnejši računalniki (CRAY 1, CRAY X-MP, CRAY 2, CYBER-205, HITACHI, FUJITSU, NAS,...). Ekstremne sposobnosti pa nudijo za ekstremno visoko ceno. Zato je bilo ob koncu leta 1987 instaliranih vsega okrog 300 takih sistemov (Hallin, 1988). Imajo dobro dodelano programsko opremo (FORTRAN, PASCAL), za simulacijo je že možno uporabljati simulacijske jezike. Znano je, da simulacijska jezika CSSL in ACSL delujeta na nekaterih superračunalnikih (CRAY, CYBER 205 - Colijn, 1986). Toda v splošnem obstaja na superračunalnikih le simulacijska programska oprema za

posebne namene. Zato se superračunalniki uporabljajo skoraj izključno za reševanje takšnih problemov, za katere druge vrste računalnikov ne zadoščajo (meteorologija, dinamika tekočin, fizika plazme idr.).

Sodobni razvoj materialne in programske opreme pa nudi dve cenejši možnosti:

- vrstične procesorje in
- sisteme paralelnih procesorjev.

Oba načina se po zmožnostih približujeta superračunalnikom, žal pa še ni na voljo ustrezno izpopolnjene programske opreme.

Vrstični procesorji

Vrstični procesorji (array processor) so periferne računalniške enote, ki zelo povečajo sposobnosti računalnika, na katerega se priključijo. Omogočajo paralelno izvajanje določenih operacij s pomočjo več aritmetičnih enot. Lahko so v obliki modulov, ki se dodajo na vodilo glavnemu računalniku. Glavni računalnik služi za pripravo, prevajanje in nalaganje programa v periferni vrstični procesor. Ker programska oprema še ni dovolj dodelana, je programiranje zahtevno. Za nekatere vrste procesorjev obstojajo osiromašeni prevajalniki v jeziku FORTRAN, povezovalniki in ustrezne knjižnice.

V začetku (sredi sedemdesetih let) so vrstične procesorje največ uporabljali v procesiranju signalov, konec sedemdesetih let pa so se pojavile že prve simulacijske aplikacije. Leta 1984 je predstavljala simulacija že 30% vseh aplikacij z vrstičnimi procesorji. Najprej so se uporabljali predvsem za simulacijo v realnem času, saj so bili sposobni dovolj hitro simulirati kompleksne dinamične sisteme, v osemdesetih letih pa so se začeli uporabljati tudi za reševanje parcialnih diferencialnih enačb. Slednja problematika sicer navadno ne zahteva računanja v realnem času, zato pa so problemi računsko tako kompleksni, da je računanje s konvencionalnimi računalniki preveč dolgotrajno.

Med komercialnimi vrstičnimi procesorji obstaja tudi nekaj simulacijsko specializiranih procesorjev. Zanje je značilno, da imajo tako procesno enoto kot ustrezno programsko opremo prilagojeno specifiki simulacijskih operacij. Sicer je večina vrstičnih procesorjev bolj prirejena za obdelavo signalov (večje dimenzije vektorjev).

Med najbolj znanimi vrstičnimi simulacijskimi procesorji naj omenimo procesorja AD-10 in AD-100 (Applied Dynamic International), ki sta bila projektirana kot nadomestilo za hibridne sisteme (Grierson, 1986). Simulacijske probleme je na sistemih AD-10 in AD-100 bilo možno programirati zelo udobno z uporabo simulacijskega jezika ADSIM (ADvanced SIMulation language) na glavnem računalniku. Ta jezik sicer ni bil tako sposoben, kot so sorodni simulacijski jeziki vrste CSSL na splošnonamenskih računalnikih, omogočal pa je strukturno programiranje (sekcije INITIAL, DYNAMIC, TERMINAL), vseboval je več integracijskih metod, omogočal pa je tudi enostavnejše eksperimentiranje.

Enega najsodobnejših simulacijskih sistemov, ki uporablja vrstični procesor, je predstavljala simulacijska delovna postaja XANALOG-1000 (Schrage, Mc Ardle, 1986, Schmidt, Scheider, 1988). Proizvajalci so uporabili ceneni vrstični procesor in razvili popolnoma novo programsko opremo, ki je omogočala programiranje s pomočjo grafičnega bločnega urejevalnika. Ta je izvajal kompletno diagnostiko, prevajalnik pa je nato iz bločne sheme generiral program za vrstični procesor. Ob ustreznih perifernih enotah je bila postaja predvsem namenjena simulaciji v realnem času. Obstajala pa je tudi verzija programske opreme za osebni računalnik.

Paralelni procesorski sistemi

Čas za simulacijo na enoprocesorskem računalniku se zelo povečuje s kompleksnostjo problema. Če se hočemo temu vsaj delno izogniti, moramo uporabiti računalniške sisteme s paralelnimi procesorji (MIMD - multiple instruction, multiple data).

Pri simulaciji na paralelnih procesorskih sistemih si morajo procesorji med integracijo izmenjavati podatke. Učinkovito računanje zahteva ustrezno razdelitev programskih (modulnih) segmentov med procesorje, pri čemer morata biti izpolnjena predvsem naslednja dva pogoja:

- posamezni procesorji naj potrebujejo približno enake čase izračunavanja v fazi, ko ni potrebna medsebojna komunikacija in
- pri komunikaciji naj se prenaša čim manj podatkov.

Komunikacija je zelo kritična. Ker je za zvezno simulacijo značilno, da je čas izračunavanja relativno velik v primerjavi s časom komuniciranja, je uporaba paralelnih procesorskih sistemov pri simulaciji zelo učinkovita.

Pri programiranju je posebno pomembno, da dosežemo učinkovito izvajanje paralelnih operacij. Že pripravljeni program za enoprocesorski sistem je sicer možno reorganizirati tako, da se izvaja na več procesorjih. Vendar je ta metoda za simulacijo neučinkovita. Za učinkovito simulacijo je potrebno izhajati iz paralelne strukture, ki je vsebovana v problemu. Sheme modelov, ki so pripravljene za simulacijo na analognem računalniku ali za simulacijo z bločno-orientiranim digitalnim simulacijskim jezikom, so zelo primerne za razgradnjo problema za obravnavo na paralelnem procesorskem sistemu. Znani so nekateri algoritmični postopki (Boullard, Coen, 1985), ki razgradijo model na ustrezne paralelne poti. Osnovni elementi takega algoritmičnega postopka so bloki (sumator, integrator, prenosna funkcija, ...), za katere je potrebno poznati čas za izračunavanje. Algoritem zahteva, da prekinemo zanke v modelu na izhodih blokov z zakasnitvenimi atributi (integrator, zakasnitev, zadrževalnik). S pomočjo ustreznih postopkov pridemo do paralelnih poti, ki se zaključijo v blokih (vozliščih) z zakasnitvenimi atributi. Vsak simulacijski cikel se začne z znanimi vrednostmi izhodov teh blokov in se konča z izračunom izhodov teh blokov za naslednji cikel. Število paralelnih poti definira maksimalno potrebno število procesorjev. Vsaki paralelni poti je pridružen tudi čas izvajanja in najdaljši čas vpliva na hitrost izvrševanja simulacije.

Na podlagi takšnih algoritmičnih postopkov bo možno avtomatizirati celoten postopek. Uporabnik bo definiral model na običajen način z uporabo simulacijskega jezika.

Eden bolj znanih paralelnih procesorskih sistemov, ki se je uporabljal v simulaciji, je bil paralelni procesor DELFT (DPP) (Bruijn, Soppers, 1986). Programiranje je bilo možno s pomočjo glavnega računalniku tipa VAX v nekakšnem makro zbirniku in v simulacijskem jeziku PARSIM. Jezik je vseboval le osnovne bloke (kot analogni računalnik) ter dvokoračno Adams-Bashforth-ovo integracijsko metodo. Uporabnik je moral v simulacijskem modelu definirati, v kateri procesni enoti naj se izvršuje določen del modela.

Na področju digitalne simulacije zveznih sistemov pa največ obetajo transputerji (Hamblen, 1987, Eckelmann, 1987). S pomočjo teh integriranih vezij je možno sestaviti cenen paralelni procesorski računalnik izjemnih sposobnosti.

Transputer je računalnik v obliki integriranega vezja. Eden prvih transputerjev T800 je imel 32 bitni procesor (10 MIPS (milijon instrukcij v sekundi)) in 32 bitno aritmetično enoto z plavajočo vejico (1.5 MFLOPS), 4K zlogov pomnilnika RAM, vhodno izhodne vmesnike ter uro. S pomočjo serijskih povezav se lahko zelo enostavno poveže poljubno število transputerjev. Vsak ima štiri seri-

jske linije, zato komunikacija ne predstavlja posebno ozkega grla. Komunikacija med transputerji poteka asinhrono. Pretvornike A/D in D/A je možno direktno priključiti na transputer.

Programska oprema za tovrstne računalnike še ni dodelana. Podjetje INMOS je prvo razvilo visok programski jezik OCCAM, možno pa je programirati tudi v jeziku C. OCCAM je strukturiran jezik, ki omogoča zaporedne in vzporedne operacije na enem ali več transputerjih (SEQ blok - operacije se izvajajo zaporedno, PAR blok - operacije se izvajajo vzporedno z drugimi operacijami na drugih transputerjih). V vsakem bloku je potrebno definirati operacije čitanja podatka s serijske linije in operacije pošiljanja podatkov na serijske linije. Te operacije omogočajo komuniciranje med transputerji (primer 5.22).

Do paralelne razgradnje pridemo na enak način kot pri drugih paralelnih procesorskih sistemih. Prednost pa je v tem, da imajo transputerski sistemi veliko število paralelnih procesorjev, tako da je možno enostavne modele razgraditi na posamezne integratorje (sumacijske z vhodnimi ojačenji). Ob inicializaciji vsi transputerji pošljejo na izhodne serijske linije začetne vrednosti blokov z zakasnitvenim atributom, nakar ločeno računajo, dokler v ustreznem trenutku ni potrebna komunikacija.

Delovanje transputerskega sistema je tem bolj učinkovito, čim več je računanja v transputerjih v primerjavi s prenašanjem podatkov preko serijskih linij. Zato je delovanje transputerjev učinkovitejše v primeru kompleksnejših integracijskih metod.

4.1.3 Analogno-hibridni sistemi

Analogno - hibridni sistem je simulacijski sistem, ki ga dobimo s povezavo analognega in digitalnega računalnika. Analogni računalnik omogoča izredno hitro simulacijo zveznih dinamičnih sistemov s pomočjo povezave elektronskih komponent (integratorji, sumatorji, potenciometri, množilniki,...). Digitalni računalnik pa ima predvsem razne krmilne funkcije, v modernih izvedbah pa tudi funkcije za opis modela, za dokumentacijo rezultatov, včasih pa omogoča tudi pravo hibridno simulacijo, t.j. da se nekateri deli modela simulirajo na analognem, nekateri pa na digitalnem delu. Običajno je možno analogni del uporabiti kot samostojni simulacijski sistem.

Osemdeseta leta predstavljajo zadnji preporod analogno-hibridnih sistemov. Po-

javili so se cenejši mikroprocesorsko vodeni sistemi s cenenimi in zanesljivimi integriranimi komponentami. Leta 1983 pa je prišel na trg novi večprocesorski sistem SIMSTAR podjetja EAI, ki združuje prednosti pravega paralelnega računanja (predvsem analogna integracija) in večprocesorskega izvajanja drugih operacij. Ima izredno sposobno programsko opremo, ki omogoča, da uporabnik programira računalnik s pomočjo simulacijskega jezika.

Nove vrste računalnikov so bolj ali manj odpravile večino slabosti analognega računanja. Te slabosti so bile v glavnem naslednje (Havranek, 1983):

- zahtevno programiranje in težko odpravljanje napak,
- zahtevno vzdrževanje in
- visoka cena.

Digitalni simulacijski sistemi so ob današnjem stanju primernejši za reševanje večine problemov. Toda še vedno obstojajo primeri, kjer prinaša hibridna simulacija predvsem zaradi velike hitrosti določeno prednost (npr. simulacija v realnem času).Učinkovita je tudi uporaba v izobraževanju, saj je z njimi možno bolj ilustrativno prikazovati določene lastnosti dinamičnih sistemov. Ne glede na to, da so hibridni sistemi vsaj v smislu materialne opreme v zatonu, pa koncepti analogno-hibridne simulacije ostajajo pomembni tudi v prihodnje (npr. paralelizacija kode za paralelno - procesorske sisteme, skaliranje, ...).

4.2 Osnovne lastnosti in razvrstitev simulacijskih jezikov

Simulacijski jeziki predstavljajo moderno simulacijsko orodje, ki se zaradi cenenosti največ uporabljajo. V glavnem so precej neodvisni od materialne in sistemske programske opreme. Edina njihova slabost je relativna počasnost izvajanja, kar zlasti omejuje njihovo uporabnost za simulacijo v realnem času.

Simulacijske jezike glede na splošnost pri opisu modelov lahko razvrstimo v:

- splošnonamenske jezike in

- namenske oz. problemsko orientirane jezike.

Pri *splošnonamenskih simulacijskih jezikih* je način opisa modela precej neodvisen od vrste problema. Znani splošnonamenski jeziki so npr. ACSL, TUTSIM, CSMP. Z njimi opisujemo probleme, ki so opisani z algebrajskimi in diferencialnimi enačbami. Lahko rečemo, da so toliko splošni, kolikor so v teoretičnem modeliranju splošne algebrajsko - diferencialne enačbe. Zato so taki jeziki uporabni na številnih področjih, vendar pa morajo imeti uporabniki osnovna znanja iz modeliranja in simulacije. Zavedati pa se moramo, da je simulacijski jezik splošen le zato, ker z njim lahko obravnavamo splošne matematične in fizikalne zakonitosti, ne pa zato, ker bi bil sposoben reševati vse probleme. Splošnonamenski jeziki so sposobni na specialnih področjih reševati ne preveč zahtevne, predvsem pa ne preveč kompleksne probleme.

Problemsko orientirani simulacijski jeziki pa so namenjeni za učinkovito modeliranje in simulacijo na posebnih področjih. Znani so npr. simulacijski jeziki za simulacijo električnih vezij SPICE, PSPICE (Micro Sim Corp.), ECAP. Te jezike je možno rutinsko uporabljati tudi brez poglobljenega znanja o modeliranju in simulaciji. Z razumevanjem analogije fizikalnih in matematičnih zakonitosti med posameznimi področji (npr. analogija električnih in mehanskih veličin) jih je možno uporabljati tudi na drugih področjih. Vendar se ta možnost redko uporablja, saj tako delo ponovno zahteva poglobljeno znanje iz modeliranja in simulacije, vprašljiva pa je tudi učinkovitost numeričnih postopkov.

Ena od značilnosti modernih simulacijskih jezikov je tudi v tem, da lahko uporabnik na enostavne načine vključi submodele, ki jih potrebuje za reševanje svojih problemov (npr. z makro jezikom) in na ta način dejansko generira nov jezik, ki je bolj primeren za posebne namene. Vprašljiva pa ostane učinkovitost matematičnih postopkov in programov, do katerih uporabnik nima dostopa. Takšnim jezikom pravimo višji splošnonamenski jeziki, ker vsebujejo kompleksne operatorje za opise modelov. Nižji simulacijski jeziki vsebujejo le osnovne gradnike za opis modela, elemente programskega krmiljenja, ustrezne numerične postopke ter možnosti za predstavitev rezultatov.

Objektno orientiran in več domenski jezik Modelica pa združuje dobre lastnosti splošnonamenskih in namenskih jezikov. Osnovni koncept je povsem splošnonamenski, ob uporabi knjižnic pa postane zelo sposobno modelersko orodje za neko konkretno domeno.

V tem delu pretežno obravnavamo splošnonamenske simulacijske jezike. Uporabl-

jamo jih lahko za reševanje relativno zahtevnih problemov na vseh področjih znanosti in tehnike, učinkoviti pa so tudi v izobraževanju.

Glede na vrsto modela, ki ga opisujejo, delimo simulacijske jezike na:

- zvezne,
- diskretne in
- kombinirane.

Slednja delitev je pomembna tako v smislu modeliranja kot tudi razvoja simulacijskih jezikov, saj posamezne vrste simulacij (zvezna, diskretna in kombinirana) zahtevajo zelo različne koncepte pri zasnovi jezikov. Zato bomo v naslednjih podpoglavjih te tri vrste simulacijskih jezikov nekoliko podrobneje obravnavali.

Posebno problematiko predstavljajo jeziki, ki omogočajo simulacijo v realnem času. Ti so sicer lahko zvezni, diskretni ali kombinirani, vendar si bomo v tem delu ogledali le problematiko, ki v glavnem zadeva zvezno simulacijo.

4.2.1 Zvezni simulacijski jeziki

Zvezni simulacijski jeziki se uporabljajo za simulacijo zveznih dinamičnih sistemov. Običajno imajo tudi nekatere sicer omejene sposobnosti kombinirane simulacije, vendar v glavnem le v smislu realizacije nezveznosti, ne pa tudi v smislu numerično pravilne obdelave nezveznosti.

Glede na to, ali so modeli opisani z navadnimi (ODE) ali parcialnimi diferencialnimi enačbami (PDE), ločimo simulacijske jezike:

- za simulacijo problemov, opisanih z ODE in
- za simulacijo problemov, opisanih s PDE.

Prvo imenovani simulacijski jeziki so danes izredno popolni, medtem ko dosedaj še ni na voljo splošnonamenskih jezikov za reševanje PDE, čeprav so se prvi taki programski sistemi pojavili že okoli leta 1970. Numerična problematika je v

sistemih, ki jih opisujejo PDE mnogo zahtevnejša, kar lahko ocenimo z definicijo učinkovitosti integracijskega algoritma za reševanje določenega problema:

$$\eta(a,p) = \frac{t_{cpu}(a^*,p)}{t_{cpu}(a,p)} \tag{4.1}$$

kjer pomenijo

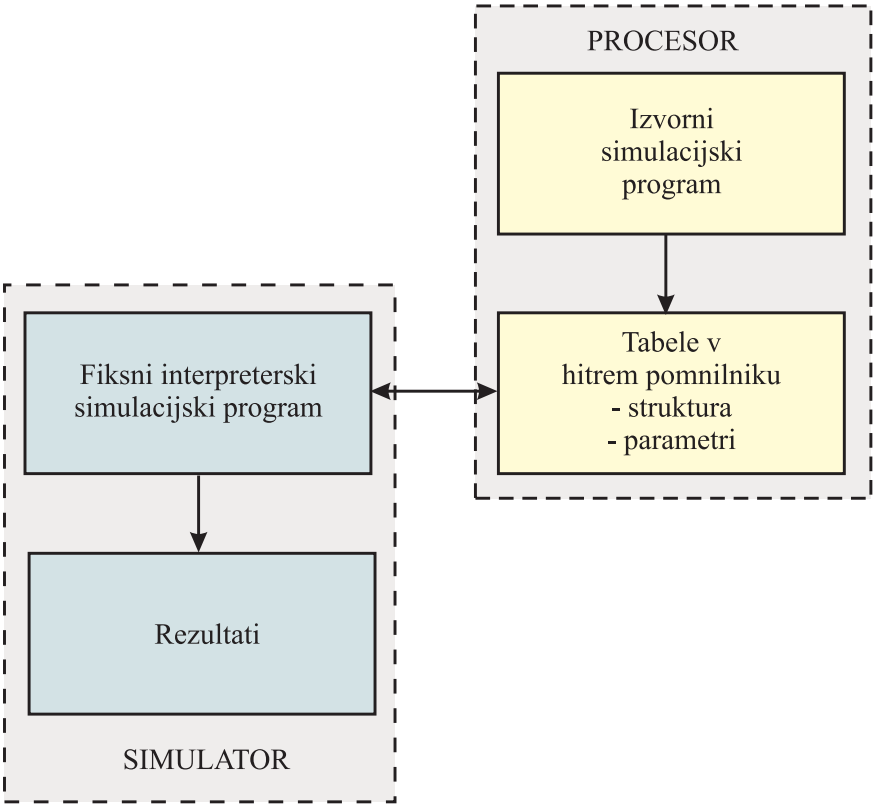
t_{cpu}	...	potreben čas CPU za rešitev problema,
η	...	učinkovitost integracijskega algoritma,
a	...	oznaka uporabljenega algoritma,
a^*	...	oznaka optimalnega algoritma za dani problem in
p	...	simulacijski problem.

Pri reševanju problemov ODE učinkovitost integracijskega algoritma η zelo redko doseže vrednost, ki je manjša od 0.01 za kakršenkoli problem in algoritem razen v primerih zelo togih ali oscilatornih sistemov. Pri reševanju problemov PDE pa je η mnogo manjši. Če upoštevamo samo metode, ki se često uporabljajo pri reševanju eliptičnih problemov, analiza pokaže, da je vrednost učinkovitosti integracijske metode η reda 10^{-3} do 10^{-6}. Torej je v tem primeru še bolj pomembna izbira ustreznega integracijskega algoritma. Metode za reševanje problemov PDE se tako razlikujejo, da je vprašljivo, če je sploh možno narediti povsem splošnonamenski simulacijski jezik. Danes znani simulacijski jeziki se lahko uporabljajo le za specializirane probleme (DSS, LEANS, FORSIM,...). V nadaljnjem delu bomo obravnavali le simulacijske jezike za reševanje ODE problemov.

Iz načina realizacije sledi najpomembnejša razdelitev tovrstnih simulacijskih jezikov na:

- interpreterske in
- prevajalniške.

Pri interpreterskih simulacijskih jezikih (slika 4.1) se iz programa, ki ga napiše uporabnik, zgradijo v hitrem pomnilniku tabele, ki opisujejo strukturo in parametre modela. Fiksni interpreterski simulacijski program v fazi simulacije izvaja bloke, ki opisujejo model, po ustreznem vrstnem redu. Uporabnik lahko interaktivno spreminja strukturo in parametre modela brez kakršnegakoli prevajanja. Zato je interpreterski način zelo primeren pri razvoju modela, ko pogosto spreminjamo tudi strukturo. Slabost interpreterskega načina pa je počasnejša simulacija,

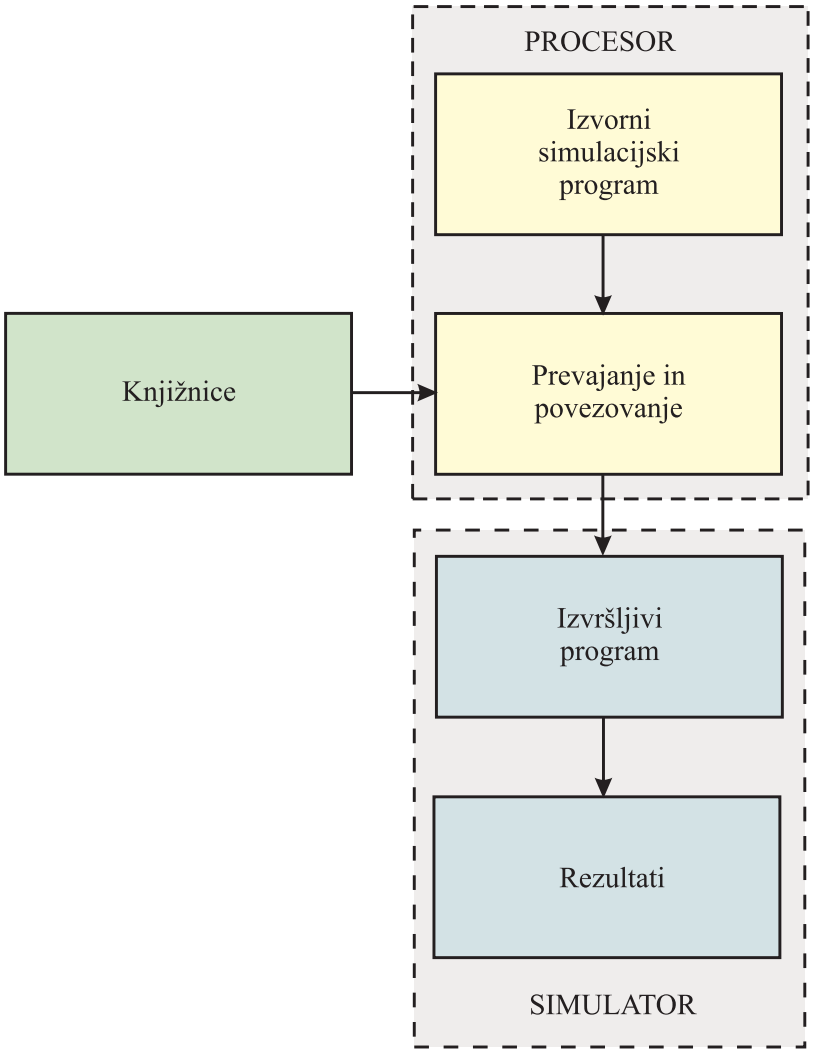


Slika 4.1: Zgradba interpreterskega simulacijskega jezika

kar ni primerno za simulacijo že preizkušenih kompleksnih modelov. Interpreterski jeziki so zlasti neprimerni, če simulacijski tek predstavlja samo del kompleksnega eksperimenta (npr. optimizacija, analiza robustnosti, ...). Slabost je tudi, da uporabnik ne more dodajati svojih blokov, ker interpreterski način zahteva glavni simulacijski program, ki je predhodno povezan s knjižnico, kar povzroča nefleksibilnost interpreterskih jezikov. Tretja slabost interpreterskih jezikov pa je ta, da so skoraj vedno bločno orientirani, kar pomeni, da je možno model opisati le z relativno enostavnimi vnaprej pripravljenimi bloki. Uporabnik mora definirati parametre blokov in načine povezave. Zato so simulacijski programi ponavadi dolgi, nepregledni in nemodularni. Omenjene slabosti rešujejo nekateri simulacijski jeziki z velikim številom blokov. To daje jeziku določeno kompaktnost, kar pomeni, da uporabnik redkokdaj pride do problema, ko bi potreboval vključitev novega lastnega bloka. Množica blokov pa predstavlja problem pri učenju jezika.

Prevajalniški simulacijski jeziki (splošno strukturo prikazuje slika 4.2) imajo manj

slabih lastnosti. Nekateri jeziki prevajajo izvorni program direktno na strojni



Slika 4.2: Zgradba prevajalniškega simulacijskega jezika

nivo. Ker so taki prevajalniki zelo hitri, omogočajo visoko stopnjo interaktivnosti tudi pri spreminjanju strukture modela. V glavnem pa imajo omejene zmožnosti za reševanje kompleksnih problemov. Sodobni simulacijski jeziki ponavadi prevajajo izvorni program v nek višji splošnonamenski programski jezik (običajno FORTRAN). Zato lahko uporabnik sam v simulacijski jezik na enostaven način vključuje nove operacije. Prevajalniški jeziki omogočajo lažjo prenosljivost na druge sisteme, hitrejšo simulacijo, veliko fleksibilnost in reševanje obsežnejših sistemov. Ker se generirajo neodvisni simulacijski programi za izvajanje simulacije, so primerni tudi za gradnjo specifičnih aplikativnih simulacijskih programov. Nji-

hova slabost pa je v tem, da je prevajanje, ki je potrebno ob vsaki spremembi strukture simulacijskega programa, lahko precej dolgotrajno. Zato prevajalniški simulacijski jeziki niso najprimernejši v fazi razvoja modelov.

Medtem ko so interpreterski jeziki skoraj vedno *bločno orientirani*, pa razdelimo prevajalniške jezike glede na način podajanja programa na *bločne* (eksplicitna bločna struktura) in *enačbne* (implicitna bločna struktura). Pri enačbnih jezikih je možno bloke modela opisati s poljubnimi matematičnimi izrazi, ki jih dovoljuje ciljni jezik prevajalnika. Razvoj simulacijskih programov je hitrejši, le-ti so bistveno krajši, razumljivejši in bolj modularni. Nekateri sodobni simulacijski jeziki (ESL, PSI) omogočajo interpreterski in prevajalniški način delovanja.

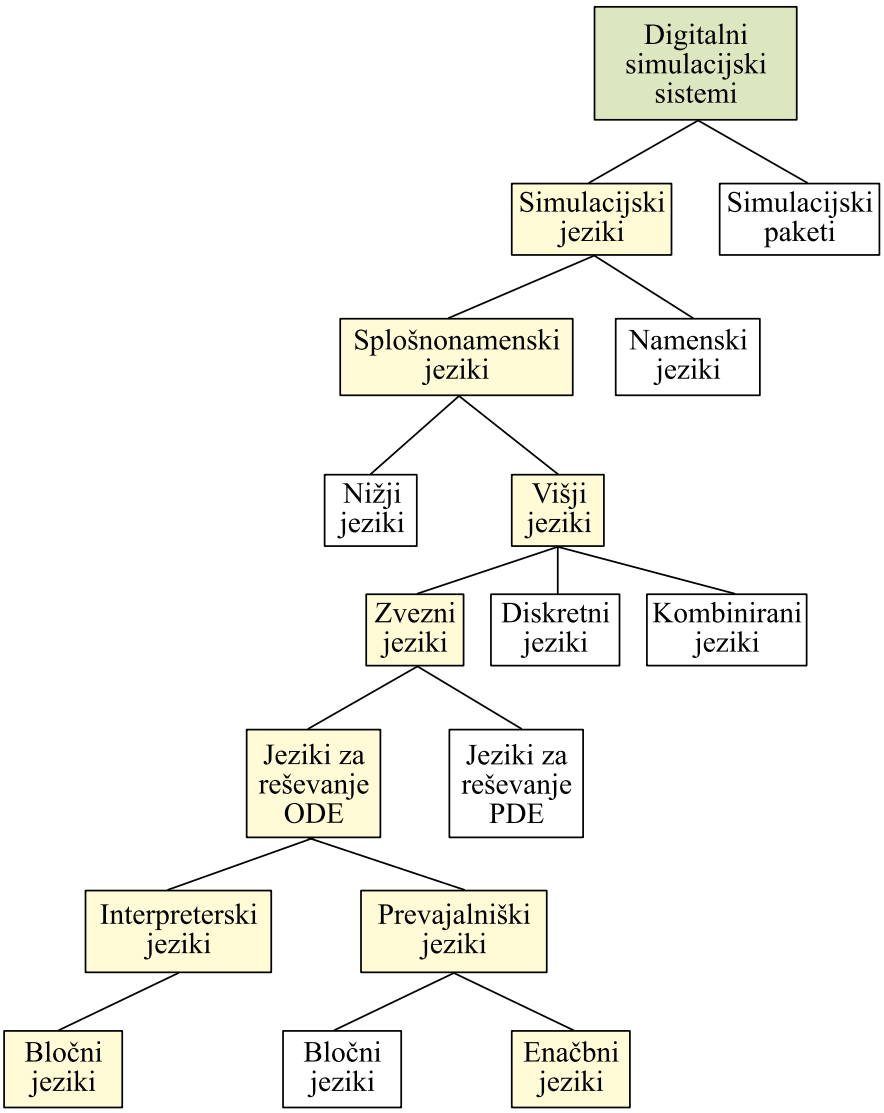
Celotno razdelitev digitalnih simulacijskih sistemov na splošnonamenskih računalnikih prikazuje slika 4.3. Razdelitev upošteva le jezike, ki jih bomo obravnavali v nadaljevanju.

4.2.2 Diskretni simulacijski jeziki

Diskretni simulacijski jeziki služijo za simulacijo diskretnih sistemov oz. dogodkov. Le-te delimo v dve skupini glede na način, kako se med simulacijo povečuje neodvisna spremenljivka, t.j. običajno čas. V večini tovrstnih simulacijskih jezikov se čas ob nastopu nekega dogodka avtomatično inkrementira na čas nastopa tega diskretnega dogodka. Ti jeziki se zato imenujejo dogodkovno orientirani simulacijski jeziki. V nekaterih, predvsem starejših diskretnih simulacijskih jezikih (npr. ena od verzij GPSS) pa se čas sinhrono povečanje z enakomernim prirastkom. Uporabnik mora v tem primeru paziti, da je prirastek zadosti majhen, da ne pride do večjih napak. Tem jezikom pravimo intervalno orientirani simulacijski jeziki.

Sodobni diskretni simulacijski jeziki so dogodkovno orientirani. Glede na vgrajene značilnosti se med seboj zelo razlikujejo. Najpomembnejša značilnost je strategija izbire naslednjega dogodka (podobno kot integracija pri zveznih simulacijskih jezikih). V glavnem uporabljajo simulacijski jeziki tri strategije (Hooper, 1987 , Neelamkavil, 1987):

- strategija razporeda dogodkov (event scheduling),
- strategija odbiranja aktivnosti (activity scanning) in



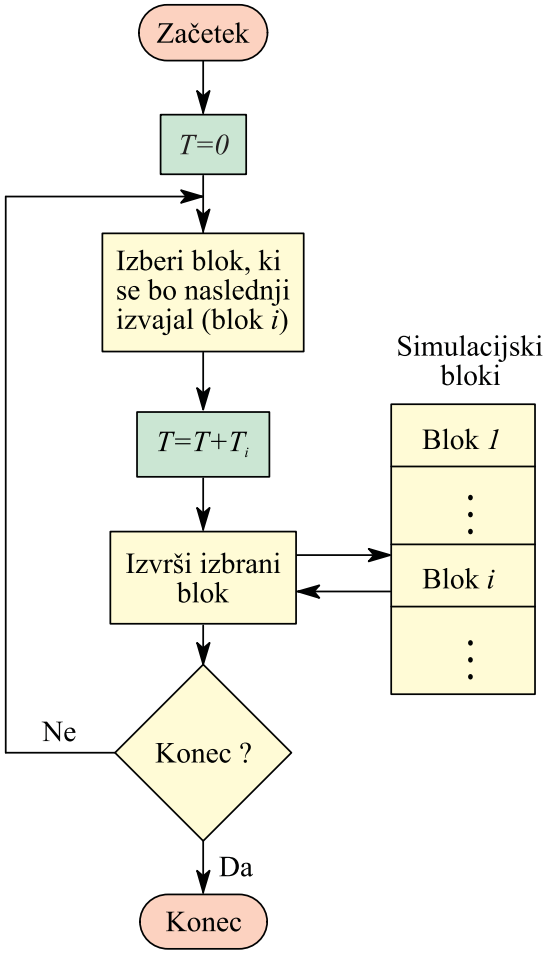
Slika 4.3: Vrste digitalnih simulacijskih sistemov na splošnonamenskih računalnikih

- strategija procesne interakcije (process interaction).

Moderni simulacijski jeziki lahko vsebujejo tudi kombinacije omenjenih strategij.

Za vse strategije velja, da se ob naslednjem dogodku izvrši ustrezni modelni blok (podprogram), ki spremeni stanje sistema. Vse strategije vsebujejo tudi koncept pogojnih in brezpogojnih dogodkov. Brezpogojni dogodki se izvršijo ob točno

določenih trenutkih, pogojni dogodki pa so odvisni tudi od stanj sistema. Splošno strukturo diskretne simulacije prikazuje slika 4.4.



Slika 4.4: Splošna struktura diskretne simulacije

Strategija razporeda dogodkov zahteva, da se brezpogojni dogodki izvajajo drug za drugim, kakor so programirani. Ob nastopu dogodka program iz liste dogodkov izbere tistega, ki ima najzgodnejši čas nastopa, čas pa se avtomatsko poveča na čas nastopa tega dogodka. Vsi pogoji, razen časovnih, morajo biti znotraj blokov, ki opisujejo dogodke. Simulacija se zaključi, ko se izvršijo vsi dogodki.

Aktivnost je stanje sistema med dvema zaporednima dogodkoma in definira prehod med njima. Strategija odbiranja aktivnosti ne vsebuje liste dogodkov, temveč poteka simulacija od dogodka do dogodka z odbiranjem aktivnosti. Vsaka aktivnost vsebuje logični pogoj, ki zavisi od simulacijskega časa in stanja sistema.

V vsakem časovnem koraku se odbira status vseh aktivnosti (testirajo se stanja vseh objektov v sistemu). Na ta način se vrstni red dogodkov vzpostavlja avtomatično glede na izbrane aktivnosti. Večina modernih jezikov ne uporablja te metode.

Objekte diskretnega modela (npr. stranka v poštnem uradu) in dogodke lahko povežemo v procese (npr. proces je čakanje stranke in streženje stranke) in uporabimo metodo procesne interakcije. Pri tem opišemo vedenje sistema z nizom procesov, ki sestojijo iz nizov medsebojno izključujočih se aktivnosti, kar pomeni, da se v določenem trenutku lahko začne le ena aktivnost. Procesi se lahko prekrivajo, interakcije med procesi pa se uporabljajo za opis modela. Poudarek je na razvrščanju procesov, dogodki pa se izvršujejo indirektno z aktiviranjem procesa (razporejanje in izvrševanje blokov, ki opisujejo akcije procesov), ki se v danem trenutku nahaja na vrhu liste. Procesi se lahko prekinjajo, obstajajo pa tudi t.i. reaktivacijske točke. Konflikt prekrivanja procesov se vrši s čakanjem v vrsti ali z zakasnitvijo.

Modeliranje z metodo procesne interakcije je enostavnejše, ker je programska struktura zelo podobna modelni strukturi, nudi pa manj programskih možnosti in fleksibilnosti v primerjavi z metodo razporeda dogodkov.

Tabela 4.1 prikazuje razvrstitev diskretnih simulacijskih jezikov glede na različne strategije. Vidimo, da imajo nekateri simulacijski jeziki različne možne strategije (SIMSCRIPT, SLAM).

Tabela 4.1: Razvrstitev diskretnih simulacijskih jezikov glede na strategije

Strategija razporeda dogodkov	Strategija odbiranja aktivnosti	Strategija procesne interakcije
GASP (II,IV)	AS	GPSS(/360,V,/H)
SIMPSCRIPT (I.5,II,II.5)	CSL	Q-GERT
SLAM, SLAM II	ECSL	SIMSCRIPT II.5
SIMAN	ESP	SLAM,SLAM II
	SIMON	SIMAN
		SIMULA
		ARENA

4.2.3 Kombinirani simulacijski jeziki

Kombinirani simulacijski jeziki vsebujejo zmožnosti zveznih in diskretnih simulacijskih jezikov (integracija, simulacija diskretnih dogodkov). Dolgo so menili, da so tovrstni simulacijski jeziki nepotrebni, ker so jeziki za simulacijo zveznih sistemov z minimalnimi zmožnostmi simulacije diskretnih dogodkov zadostovali za simulacijo realnih tovrstnih problemov, diskretni jeziki pa so zadostovali za simulacijo diskretnih dogodkov. Večina jezikov in paketov za kombinirano simulacijo je nastala z razširitvijo čistih diskretnih simulacijskih jezikov (npr. GASP - IV oz. GASP - II). Razlog je v tem, ker so zvezni simulacijski jeziki zahtevnejši glede uporabe numeričnih postopkov, pri diskretnih simulacijskih jezikih pa so mnogo bolj kompleksni strukturni koncepti. Diskretne simulacijske jezike je v glavnem potrebno nadgraditi le z ustreznimi integracijskimi postopki. V praksi pa v glavnem večina potreb po kombinirani simulaciji izhaja iz problemov, ki se rešujejo s simulacijskimi jeziki za zvezne sisteme, medtem ko problemi, ki se rešujejo z diskretnimi simulacijskimi jeziki, le redko potrebujejo integracijske postopke. Nekateri uporabniki zveznih simulacijskih jezikov želijo, da bi imeli na voljo določene zmožnosti kombinirane simulacije. Zato razvijalci širijo tudi zvezne simulacijske jezike v smeri, za katero smo sicer omenili, da je dosti zahtevnejša.

Bistvo sposobnih kombiniranih simulacijskih jezikov je v tem, da imajo razen mehanizmov zveznih in diskretnih jezikov tudi mehanizme za prehod iz zveznega dela modela v diskretnega in obratno. Zaradi pogostih nastopov nezveznosti morajo imeti vgrajene ustrezne numerične postopke za pravilno obdelavo nezveznosti. To lastnost imajo le nekateri obstoječi zvezni simulacijski jeziki ter nekateri jeziki nove generacije.

Za razliko od drugih simulacijskih jezikov, je na trgu zelo malo jezikov, ki zadovoljivo rešujejo probleme kombinirane simulacije (GASP V, COSY, SYSMOD, COSMOS). Zelo izčrpen prispevek, ki obravnava probleme kombiniranih simulacijskih jezikov, je objavil Cellier, 1979.

4.2.4 Jeziki za simulacijo v realnem času

Nekateri splošnonamenski simulacijski jeziki se že relativno dolgo uporabljajo tudi za simulacijo v realnem času. Zaradi težav pri tovrstni simulaciji pa je njih uporaba zelo omejena. Zato v literaturi zasledimo sorazmerno malo podatkov (ACSL - Gauthier, 1987, MUSIC - Cser in ostali, 1986).

Osnovna zahteva, da simulacijski jezik lahko izvaja simulacijo v realnem času, je sinhronizacija z realnim časom. Za uspešno uporabo pa mora jezik imeti še druge lastnosti:

- čim hitrejše izvajanje simulacije in s tem v zvezi učinkovite numerične postopke; to je zlasti pomembno pri hitrih mehanskih in električnih sistemih, medtem ko pri procesnih sistemih ponavadi hitrost ni tako pomembna,
- večje interaktivne zmožnosti med simulacijo ter
- programsko opremo za komuniciranje s priključenimi napravami in/ali procesi.

Za hitro izvajanje simulacije so v preteklosti zlasti uporabljali bločne jezike, ki so s prevajanjem na strojni nivo omogočili učinkovito simulacijo. Prav tako so v preteklosti veliko uporabljali poenostavljeno aritmetiko (s fiksno decimalno vejico ali celo celoštevilčno), zato je bilo potrebno normirati enačbe. Danes, ko imajo že ceneni osebni računalniki aritmetične koprocesorje, se tak način ne izplača več, saj so prihranki minimalni in ne odtehtajo vloženega truda. Moderna materialna oprema zato tudi enačbnim jezikom omogoča, da se uspešno uporabljajo za simulacijo v realnem času.

Učinkoviti in poenostavljeni numerični postopki lahko prihranijo precej časa v obeh fazah simulacije in sicer pri izvajanju

- podprograma za integracijo in
- podprograma za izračun odvodov.

Po nekaterih ocenah zahteva prva faza za izvajanje metode Runge-Kutta višjega reda ob ne preveč kompleksnih modelih cca. 70% potrebnega računalniškega časa, druga faza pa 30%. Zato je zelo pomembno, da izločimo pri simulaciji v realnem času iz integracijskega podprograma vse, kar ni nujno potrebno. Metode s prilagodljivim korakom se ne uporabljajo. Veliko se uporabljajo algoritmi, ki jih pri simulaciji v nerealnem času le redko uporabljamo, npr. Eulerjeve metode (metode prvih in zadnjih diferenc), trapezno pravilo (bilinearna transformacija) in druge. Če pa je možno simulacijski model pretvoriti v diskretno obliko, pa je seveda najbolj učinkovita diskretna simulacija. V specializiranih orodjih za simulacijo

zveznih dinamičnih sistemov je včasih možno precej pridobiti na hitrosti tudi s kodiranjem integracijskega algoritma s pomočjo jezikov, ki so bližji strojnemu nivoju.

Pri računanju odvodov pa je pomembno, da že enačbe modela podamo na tak način, da zahtevajo čim manj izračunavanja. Vnaprej programirani bloki ne smejo vsebovati redundantnosti. Veliko časa lahko pridobimo z uporabo učinkovitih aproksimacijskih postopkov za korenjenje, logaritmiranje, eksponenciranje in za računanje trigonometričnih funkcij ter drugih funkcijskih generatorjev.

Interaktivne zmožnosti med simulacijo morajo omogočati sprotno spremljanje rezultatov ter direktne posege v simulacijo. Zlasti je pomembno, da lahko uporabnik med simulacijo spreminja parametre modela in da lahko realne signale iz okolice zamenja v vsakem trenutku s simuliranimi signali. Učinkovito spremljanje rezultatov pa naj omogoča prikaz spremenljivk na načine, ki so prilagojeni obravnavanim fizikalnim veličinam (ne le funkcijska odvisnost (npr. y(t)) ampak tudi v obliki "voltmetrov", "termometrov", ...).

Pri simulaciji v realnem času imamo navadno na računalnik priključeno zunanjo materialno opremo. Zato mora imeti sistem ustrezne periferne enote: module za zajemanje in oddajanje podatkov (pretvorniki A/D in D/A, digitalni vhodno-izhodni moduli), časovni modul, po potrebi modul za sprejem prekinitev in druge.

4.2.5 Primeri uporabe zveznih enačbno orientiranih simulacijskih jezikov

Uporabnost enačbno orientiranih simulacijskih jezikov bomo prikazali na razvoju dveh simulacijskih programov. Model ekološkega sistema roparjev in žrtev in regulacijski sistem ogrevanja prostora bomo simulirali v jeziku, ki upošteva standard CSSL'67 (Strauss, 1967).

Primer 4.1 Model ekološkega sistema roparjev in žrtev

Namen tega primera je, da seznani bralca z osnovno uporabnostjo digitalnih simulacijskih jezikov. Ekološki sistem roparjev in žrtev, ki ga opisujejo enačbe

$$\begin{aligned} \dot{x}_1 &= a_{11}x_1 - a_{12}x_1x_2 \\ \dot{x}_2 &= a_{21}x_1x_2 - a_{22}x_2 \end{aligned} \tag{4.2}$$

$$\begin{aligned} x_1(0) &= x_{10} \\ x_2(0) &= x_{20} \end{aligned}$$

želimo simulirati pri konstantah $a_{11} = 5, a_{12} = 0.05, a_{21} = 0.0004, a_{22} = 0.2$ in pri začetnem številu zajcev in lisic $x_{10} = 520, x_{20} = 85$. V pomoč nam bo simulacijska shema, ki jo prikazuje slika 3.9 in enačbe (4.2).

Preden se lotimo pisanja programa, določimo vsem spremenljivkam in konstantam modela imena, ki jih bomo uporabljali v računalniškem programu. Smiselno je izbrati taka imena, ki so povezana z realnim problemom, oz. s fizikalnimi zakonitostmi. Z ozirom na oznake, ki jih uporabljamo v shemi na sliki 3.9 izberemo naslednja imena

x_1	`RAB`	$\dot{x}_1$	`RABDOT`
x_2	`FOX`	$\dot{x}_2$	`FOXDOT`
x_{10}	`RAB0`	x_{20}	`FOX0`
a_{11}	`A11`	a_{12}	`A12`
a_{21}	`A21`	a_{22}	`A22`

Simulacijski program sestoji iz stavkov. Vrstni red stavkov je v jezikih tipa CSSL običajno sicer poljuben, toda že zaradi boljše preglednosti programa je priporočljiv naslednji vrstni red:

1. stavki, ki opisujejo parametre modela,
2. stavki, ki opisujejo strukturo,
3. stavki, ki definirajo krmilne parametre simulacije.

Prvi stavek v simulacijskem programu je lahko stavek `PROGRAM`, s katerim damo programu želeno ime.

```
PROGRAM PREY_AND_PREDATOR
```

Nato definiramo konstante simulacijskega modela:

```
"konstante modela
   CONSTANT A11=5,A12=0.05,A21=0.0004,A22=0.2
   CONSTANT RAB0=520,FOX0=85
```

Vrstico s komentarjem uvedemo z znakom ". Komentarji povečajo dokumentacijsko uporabnost simulacijskega programa.

Nato opišemo strukturo modela s stavki, ki imajo poljuben vrstni red. Pri tem niti ni potrebno uporabiti simulacijske sheme na sliki 3.9, ampak lahko zaradi enačbne orientiranosti jezika tipa CSSL direktno uporabimo enačbi (3.8), ki specificirata oba odvoda modela.

```
RABDOT=A11*RAB-A12*RAB*FOX
FOXDOT=A21*RAB*FOX-A22*FOX
```

Da generiramo rešitvi obravnavanega modela, je potrebno oba odvoda integrirati

```
RAB=INTEG(RABDOT,RAB0)
FOX=INTEG(FOXDOT,FOX0)
```

Prvi argument stavkov `INTEG` je vhod v integrator, t.j. odvod, drugi parameter pa je začetni pogoj (začetno število zajcev in lisic).

Potem, ko smo opisali strukturo modela, je potrebno definirati še krmilne parametre simulacije. Potrebno je izbrati ustrezen čas opazovanja, t.j. dolžino simulacijskega teka. Ker so v konstante kot časovne enote vključena leta ($a_{11} = 5$ je število potomcev enega zajca v enem letu), je enota neodvisne spremenljivke eno leto. Ker želimo opazovati časovne poteke vsaj eno ekološko periodo, ki je v tem primeru približno sedem let, izberemo dolžino simulacijskega teka deset let. Pogoj za končanje simulacije podamo v stavku `TERMT`

```
TERMT T.GE.10
```

kar pomeni, da je pogoj za končanje simulacije izpolnjen, če je čas simulacije enak, ali pa presega deset let. Če pa imamo namen, da bomo med simulacijo interaktivno spreminjali pogoj za končanje simulacijskega teka, pa je smiselno, da v stavek `TERMT` uvedemo spremenljivko, katere vrednost inicializiramo s stavkom `CONSTANT`

```
CONSTANT TFIN = 10
TERMT T.GE.TFIN
```

Trenutke, v katerih uporabnik dobi rezultate simulacije, pa definiramo s stavkom

```
CINTERVAL CI=0.01
```

kar pomeni en rezultat na 0.01 leta oz. na 3.65 dni. Končno izberemo še spremenljivke, ki naj se med simulacijo izpisujejo na zaslon (stavek `OUTPUT`) in spremenljivke, ki naj se vpisujejo v datoteko (stavek `PREPAR`) za nadaljnjo obdelavo

```
OUTPUT 100,RAB,FOX
PREPAR 2,RAB,FOX
```

Številka 100 v prvem stavku pove, da naj se izpiše le vsak stoti rezultat (en rezultat v enem letu), številka dva v drugem stavku pa pomeni, da se v datoteko rezultatov vpiše vsak drugi rezultat (t.j. vsakih 7.3 dni). Simulacijski program zaključimo s stavkom `END`

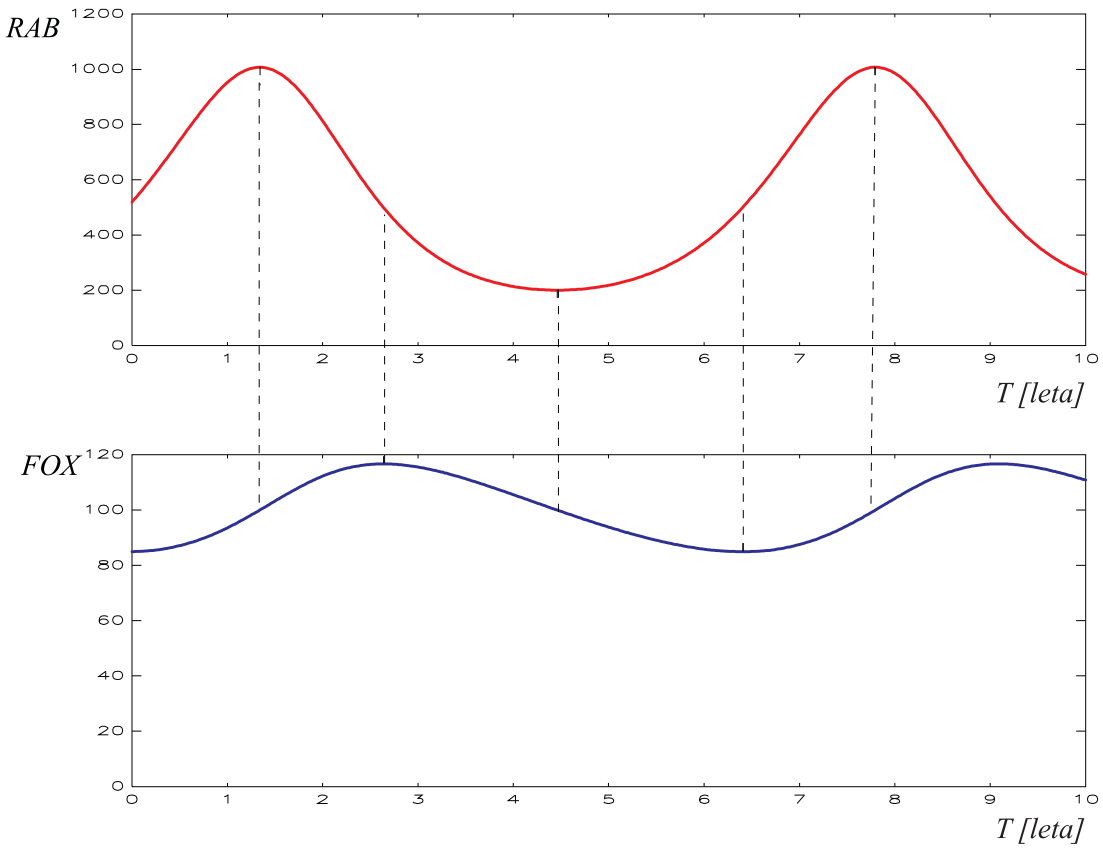
```
END
```

CSSL jeziki so prevajalniško orientirani. Simulacijski program se najprej prevede v module v splošnonamenskem jeziku (npr. FORTRAN) in v ustrezne podatkovne datoteke. Po prevajanju in povezovanju v splošnonamenskem jeziku dobimo program za simulacijo. Ko izvedemo program, se na zaslonu izpisujejo naslednje vrednosti: vrednost časa, število zajcev in število lisic. Rezultate si lahko ogledamo tudi v grafični obliki po simulacijskem teku. Celotni simulacijski program je naslednji:

```
PROGRAM PREY_AND_PREDATOR
"-------------------------------------------------
 " konstante modela
   CONSTANT A11=5,A12=0.05,A21=0.0004,A22=0.2
   CONSTANT RAB0=520,FOX0=85
"-------------------------------------------------
 " struktura modela
       RABDOT=A11*RAB-A12*RAB*FOX
       FOXDOT=A21*RAB*FOX-A22*FOX
       RAB=INTEG(RABDOT,RAB0)
       FOX=INTEG(FOXDOT,FOX0)
```

```
"-------------------------------------------------
 "dolzina simulacijskega teka
   CONSTANT TFIN = 10
   TERMT(T.GE.TFIN)
"-------------------------------------------------
 "dolzina komunikacijskega intervala
   CINTERVAL CI=0.01
"-------------------------------------------------
 "izhodne zahteve
   OUTPUT 100, RAB,FOX
   PREPAR 2,RAB,FOX
"-------------------------------------------------
END
```

Slika 4.5 prikazuje populaciji zajcev in lisic.



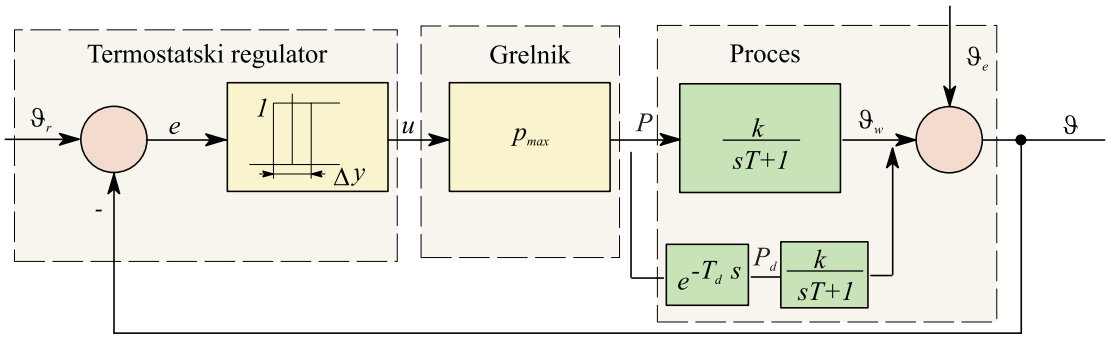
Slika 4.5: Časovna poteka števila zajcev in lisic

Opazimo, da je ekološka perioda približno sedem let, da število lisic najhitreje narašča ob največjem številu zajcev in najhitreje upada ob najmanjšem številu zajcev. □

Primer 4.2 Model regulacije ogrevanja prostora

Primer omogoča nekoliko globlji vpogled v jezike tipa CSSL in nakazuje tudi značilne korake, ki so potrebni pri analizi nekega problema s simulacijo.

Temperaturni regulacijski sistem smo uvedli v primeru 1.2. Ustrezen bločni diagram prikazuje slika 4.6.



Slika 4.6: Bločni diagram temperaturnega regulacijskega sistema

Slika 4.6 nakazuje dve možni obravnavi. Najprej bomo upoštevali model procesa brez mrtvega časa, nato pa še z mrtvim časom. Enačba

$$\dot{\vartheta}_w + \frac{1}{T}\vartheta_w = \frac{k}{T}p \tag{4.3}$$

opisuje model procesa v delovni točki, če ne upoštevamo mrtvega časa. Podatki za simulacijo so naslednji: širina histereze je $\Delta y = 1^oC$, moč grela je $p = p_{max} = 5kW$, temperatura okolice je $\vartheta_e = 15^oC$, ojačenje procesa je $k = 2^oC/KW$, časovna konstanta procesa je $T = 1h$, začetna temperatura v prostoru je $\vartheta(0) = 16^oC$ oz. $\vartheta_w(0) = 1^oC$. Referenčna temperatura je časovno spremenljiva funkcija in jo opisuje tabela 4.2.

Na začetku ponovno izberemo ustrezna imena, ki jih bomo uporabljali v programu:

Tabela 4.2: Želena sobna temperatura

t[h]	0	5.99	6	8.99	9	14.99	15	20.99	21
$\vartheta_r[^oC]$	15	15	20	20	18	18	20	20	15

ϑ_r	THR	ϑ_e	THE
ϑ	TH	$\vartheta_w(0)$	THWO
$\dot{\vartheta}_w$	THWD	ϑ_w	THW
e	E	p	P
p_{max}	PMAX	Δy	DELTAY
u	U	T	TIMCON
k	GAIN		

Na začetku programa običajno definiramo parametre modela, t.j. konstante in morebitne funkcijske generatorje

```
"konstante modela
   CONSTANT DELTAY=1,PMAX=5,GAIN=2,TIMCON=1
   CONSTANT THE=15,THWO=1
"tocke funkcijskega generatorja
   TABLE REF,1,9,...
     0.,  5.99, 6.,  8.99, 9., 14.99, 15., 20.99, 21.,...
    15., 15.,  20., 20.,  18., 18.,   20., 20.,   15.
```

Funkcijski generator definiramo z imenom (REF), s številom neodvisnih spremenljivk (1), s številom točk, v katerih je podana funkcija (9) ter z vrednostmi neodvisne (naslednjih 9 podatkov) in odvisne spremenljivke (zadnjih 9 podatkov).

Nato je potrebno definirati strukturo modela. Vrstni red stavkov je povsem poljuben. Referenčni signal v regulacijskem sistemu realiziramo s klicem funkcije za realizacijo funkcijskega generatorja

```
THR=REF(T)
```

kjer je T neodvisna spremenljivka (čas) in REF ime funkcijskega generatorja, ki smo ga definirali s stavkom TABLE.

Pogrešek v regulacijskem sistemu definiramo s stavkom

```
E=THR-TH
```

Vodimo ga v blok, ki modelira stopenjski (ON-OFF) termostatski regulator. Ustrezni model podaja histerezna funkcija s histerezno širino Δy, spodnjim nivojem nič in zgornjim nivojem ena.

```
CONSTANT STATE =0.
U=HSTRSS(E,-DELTAY/2.,DELTAY/2.,0.,1.,STATE)
```

Zadnji parameter v funkciji `HSTRSS` je začetni pogoj, ki določa izhodno vrednost v primeru, če ob prvem klicu pogrešek e leži v področju $-\frac{\Delta y}{2} \leq e \leq \frac{\Delta y}{2}$.

Grelo modeliramo s pomočjo ojačevalnega bloka

```
P=PMAX*U
```

Proces pa simuliramo z indirektno metodo s pomočjo simulacijske sheme, ki jo prikazuje slika 3.5 ali pa s pomočjo enačbe (4.3 oz. enačbe 3.4)

```
THWD=-1./TIMCON*THW+GAIN/TIMCON*P
THW=INTEG(THWD,THW0)
```

Drugi parameter stavka `INTEG` je začetni pogoj modela v delovni točki. Absolutno temperaturo pa dobimo tako, da temperaturi, ki jo daje model v delovni točki, prištejemo temperaturo okolice.

```
TH=THW+THE
```

Po opisu strukture moramo definirati še krmilne parametre simulacije. Ker želimo opazovati časovne poteke v teku enega dneva, je pogoj za končanje simulacije

```
CONSTANT TFIN=24
TERMT T.GT.TFIN
```

Integracija je osrednji postopek vsakega simulacijskega sistema. Dobri simulacijski sistemi imajo več integracijskih postopkov. Če ga posebej ne navedemo, se uporabi privzeta metoda (običajno Runge-Kutta s prilagodljivim računskim korakom). V modelih, ki vsebujejo histerezno funkcijo, pa je priporočljivo uporabiti metodo s konstantnim računskim korakom. To dosežemo s stavkom

```
ALGORITHM IALGOR=1,JALGOR=5
```

Prvi parameter `(IALGOR=1)` se uporablja za izbiro metode za inicializacijo integracijskega postopka, drugi parameter `(JALGOR=5)` pa za izbiro integracijskega postopka.

Komunikacijski interval izberemo s stavkom

```
CINTERVAL CI=0.02
```

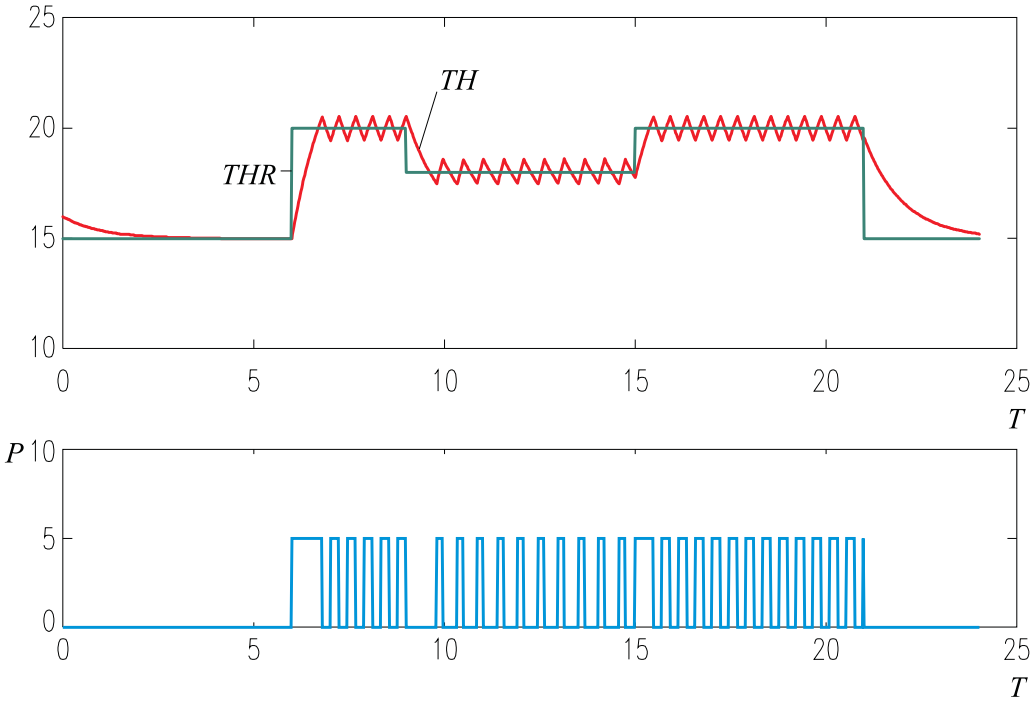
kar pomeni, da ima uporabnik na voljo rezultate simulacije vsake 0.02 h (1.2 min). Ponovno naj poudarimo, da morajo biti enote konsistentne. Čas mora biti povsod izražen v enaki enoti (npr. konstante modela, dolžina simulacijskega teka, komunikacijski interval). Če stavka `CINTERVAL` ne uporabljamo, se uporabi privzeta vrednost (1) komunikacijskega intervala. Končno definiramo še spremenljivke, ki se med simulacijo izpisujejo na zaslon in v datoteko

```
OUTPUT 10,THR,P,TH
PREPAR THR,P,TH
```

Simulacijski program zaključimo s stavkom

```
END
```

Po procesiranju lahko uporabnik izvede simulacijski tek. Slika 4.7 prikazuje rezultate simulacije (temperaturo v prostoru ϑ in želeno temperaturo ϑ_r v zgornjem in moč grela p v spodnjem diagramu). Grelo se vključuje približno na 30 min, temperatura v prostoru pa niha v pasu 1^oC. Če podvojimo velikost histereze (2^oC), dobimo rezultate, ki jih prikazuje slika 4.8. Nihanja temperature postanejo v tem



Slika 4.7: Rezultati temperaturnega regulacijskega problema

primeru prevelika za ugodno počutje v prostoru. Perioda preklapljanja grela pa se poveča na približno 50 min.

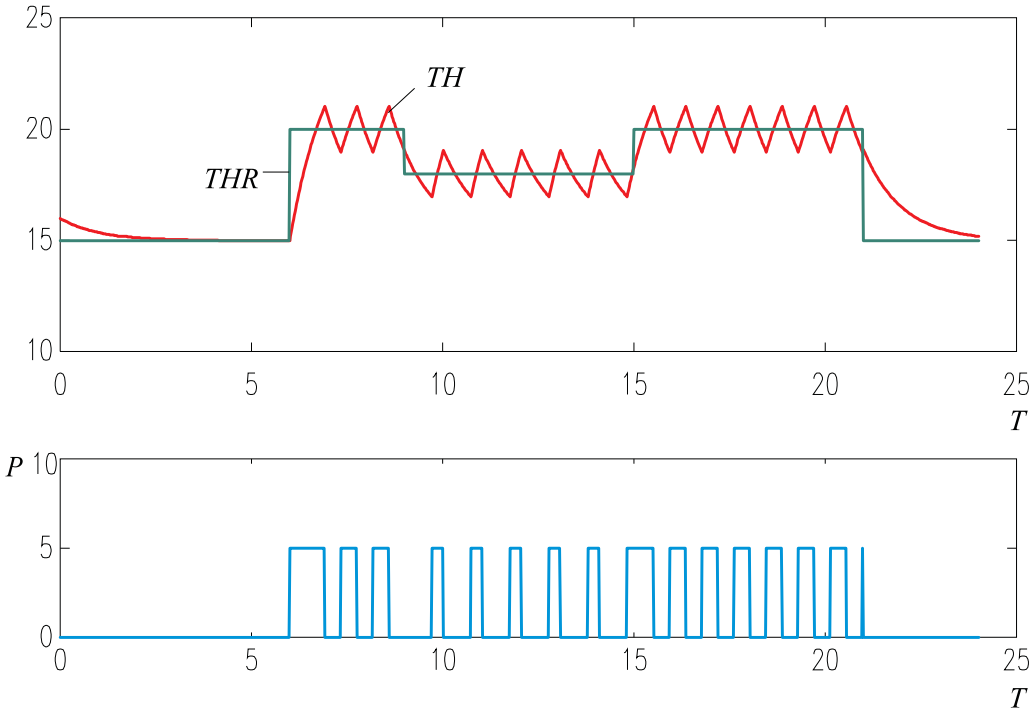
Če je termostatsko tipalo precej oddaljeno od grela, je smiselno v model ogrevanja vključiti mrtvi čas. Ustrezen bločni diagram je razviden iz slike 4.6. V ta namen vpeljemo dve novi spremenljivki

p_d	PD
T_d	TDELAY

Mrtvi čas realiziramo s klicem vgrajene funkcije DELAY

```
CONSTANT TDELAY=0.1,WORK=50*0
ARRAY WORK(50)
    PD = DELAY(P,TDELAY,WORK,CI)
```

Prvi parameter v klicnem stavku je spremenljivka, ki jo želimo zakasniti. Drugi parameter je mrtvi čas, tretji parameter pa je delovno polje, ki se uporablja



Slika 4.8: Rezultati simulacije pri podvojeni širini histereze

za realizacijo mrtvega časa. Za delovno polje je potrebno rezervirati dimenzije (s stavkom `ARRAY`) in določiti njegove začetne vrednosti. Četrti parameter je čas vzorčenja, ki določa trenutke, v katerih se v delovnem polju (premikalnem registru) izvedejo pomiki. Čim manjši je čas vzorčenja, boljši približek idealnemu zveznemu mrtvemu času dosežemo. Zato smo izbrali minimalni možni čas vzorčenja, ki je enak komunikacijskemu intervalu `CI`.

Ker sedaj spremenljivka p_d predstavlja vhod v proces, moramo modelno enačbo

```
THWD=-1./TIMCON*THW+GAIN/TIMCON*P
```

zamenjati z enačbo

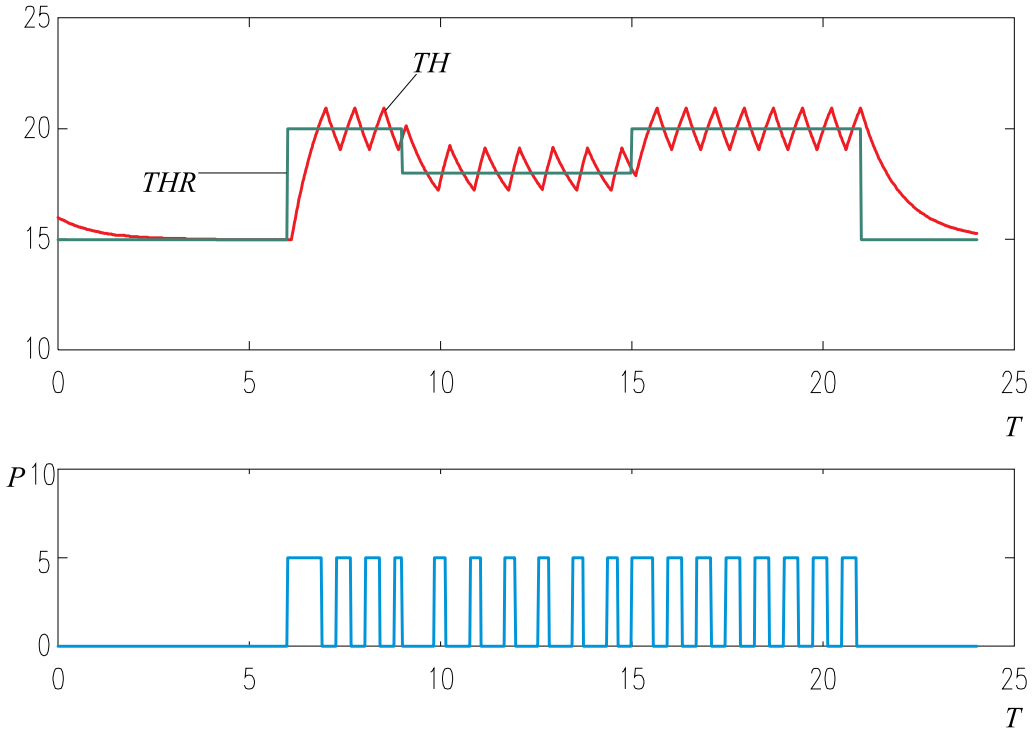
```
THWD=-1./TIMCON*THW+GAIN/TIMCON*PD
```

Celoten simulacijski program je naslednji:

```
"------------------------------------------------
"TEMPERATURNI REGULACIJSKI SISTEM
"------------------------------------------------
"konstante modela
   CONSTANT DELTAY=1,PMAX=5,GAIN=2,TIMCON=1
   CONSTANT THE=15,TDELAY=0.1,THW0=1
   CONSTANT WORK=50*0,STATE=0
"tocke funkcijskega generatorja
   TABLE REF,1,9,...
     0.,  5.99, 6.,  8.99, 9., 14.99, 15., 20.99, 21.,...
    15., 15.,  20., 20.,  18., 18.,   20., 20.,   15.
"------------------------------------------------
"delovno polje za realizacijo mrtvega casa
   ARRAY WORK(50)
"------------------------------------------------

"struktura modela
       THR=REF(T)
       E=THR-TH
       U=HSTRSS(E,-DELTAY/2.,DELTAY/2.,0.,1.,STATE)
       P=PMAX*U
       PD=DELAY(P,TDELAY,WORK,CI)
       THWD=-1./TIMCON*THW+GAIN/TIMCON*PD
       THW=INTEG(THWD,THW0)
       TH=THW+THE
"------------------------------------------------

"trajanje simulacijskega teka
   CONSTANT TFIN=24
   TERMT T.GT.TFIN
"------------------------------------------------
"izbira integracijske metode
   ALGORITHM IALGOR=1,JALGOR=5
"dolocitev komunikacijskega intervala
   CINTERVAL CI=0.02
"------------------------------------------------
"zahteve za spremljanje rezultatov
   HDR HEATING CONTROL SYSTEM
   OUTPUT 10,THR,P,TH
   PREPAR THR,P,TH
"------------------------------------------------
```

```
END
```

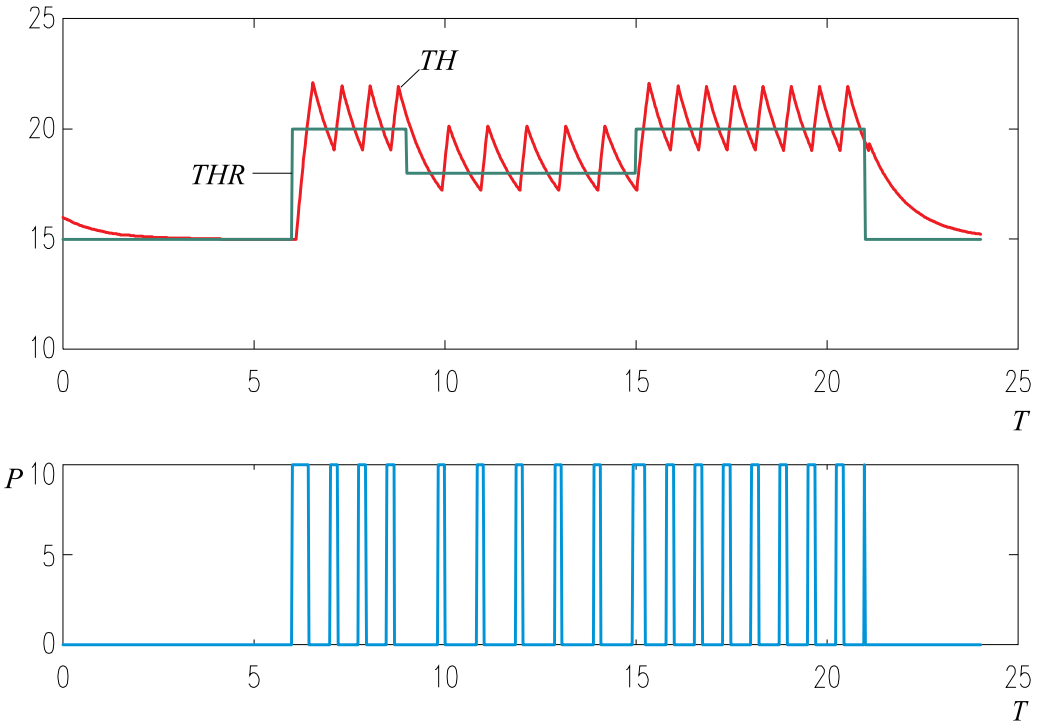
Slika 4.9 prikazuje rezultate simulacije pri mrtvem času $T_d = 0.1\ h$. Če primer-



Slika 4.9: Rezultati pri dodatnem mrtvem času

jamo rezultate simulacije z rezultati na sliki 4.7, vidimo, da temperatura niha v širšem pasu (približno 2^oC). Nihanje temperature je torej odvisno od širine histereze in od zakasnitev procesa. Očitno je, da je termostatsko tipalo predaleč od grela, da bi lahko dosegli ugodno temperaturo.

Ko dobimo preveden simulacijski program, lahko eksperimentiramo pri različnih vrednostih konstant, ne da bi bilo potrebno spreminjati in ponovno procesirati izvorni program. Če želimo npr. spremeniti moč grela, interaktivno spremenimo konstanto `PMAX`. Slika 4.10 prikazuje rezultate simulacije pri 10 kW grelu. Opazimo, da temperatura narašča hitreje, vendar so nihanja temperature večja kot v primeru na sliki 4.9 (približno 3^oC). Vpliv zunanje temperature proučujemo s spreminjanjem konstante `THE`. Slika 4.11 prikazuje rezultate, če je zunanja temperatura 17^oC. Razumljivo je, da je nemogoče doseči referenčno temperaturo, ki je nižja od temperature okolice, v kolikor ne uporabljamo hlajenja.



Slika 4.10: Rezultati simulacije pri 10 kW grelu

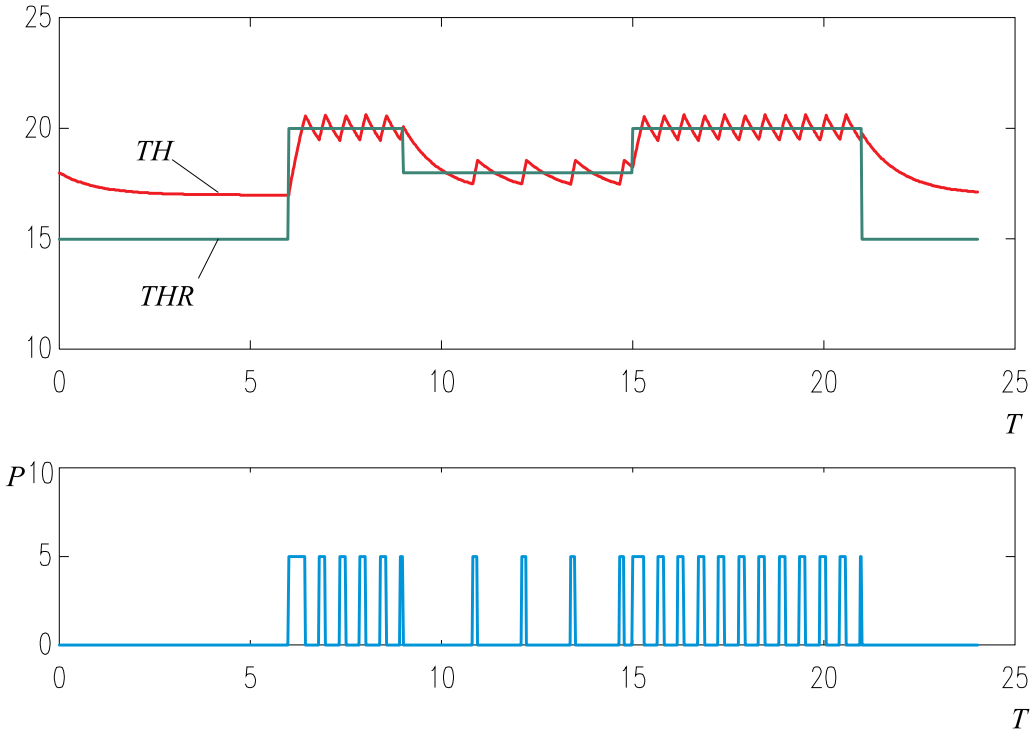
Še bolj realno sliko dogajanja običajno dosežemo z modelom, ki upošteva različni časovni konstanti v fazah ogrevanja in ohlajanja. Blok `PROCEDURAL` predstavlja učinkovito možnost za vključitev nelinearnosti (preklapljanje med dvema časovnima konstantama). V našem primeru fazo ogrevanja in fazo ohlajanja natančno določa odvod temperature v prostoru. Konstanto T=1 med ogrevanjem ($\dot{\vartheta}_w \geq 0$) in T = 4 med ohlajanjem ($\dot{\vartheta}_w < 0$) realiziramo tako, da definicijo konstante

```
CONSTANT TIMCON=1
```

zamenjamo z blokom, katerega vhod je spremenljivka $\dot{\vartheta}_w$, izhod pa časovna konstanta T. Blok prikazuje slika 4.12.

Blok `PROCEDURAL` opišemo z naslednjim delom programa:

```
CONSTANT T1=1, T2=4
PROCEDURAL (TIMCON = THWD)
       TIMCON = T1
```



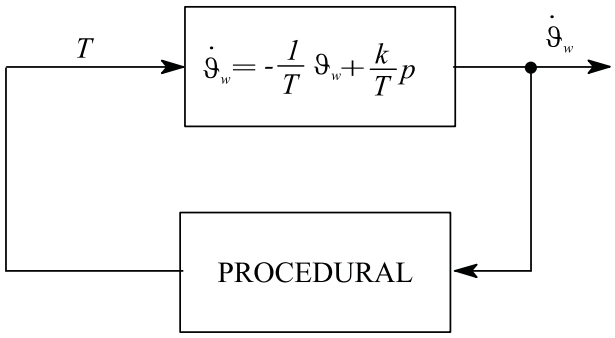
Slika 4.11: Rezultati simulacije pri višji temperaturi okolice

Slika 4.12: Blok `PROCEDURAL` za določitev konstante T

```
        IF (THWD.LT.0) TIMCON = T2
END
```

Blok `PROCEDURAL` vpeljemo s stavkom `PROCEDURAL`, ki vsebuje v oklepaju spisek izhodnih spremenljivk (levo od enačaja) in spisek vhodnih spremenljivk (desno od enačaja). Na žalost pa taka realizacija povzroči t.i. algebrajsko zanko, kajti za izračun spremenljivke $\dot{\vartheta}_w$ mora biti dana časovna konstanta T. Da pa bi lahko izračunali T, mora biti dan temperaturni odvod $\dot{\vartheta}_w$. Algebrajsko zanko prikazuje slika 4.13.

Nekateri simulacijski jeziki vsebujejo algoritme za numerično reševanje algebrajske zanke, vendar je taka simulacija zelo počasna.



Slika 4.13: Algebrajska zanka

V našem primeru pa lahko enoumno določimo fazo ogrevanja in ohlajanja s pomočjo spremenljivke p_d, saj pozitivna vrednost spremenljivke p_d vedno pomeni naraščanje temperature. Pri problemu brez dodatnega mrtvega časa pa lahko uporabimo kar regulirno veličino u.

```
CONSTANT T1=1, T2=4
PROCEDURAL (TIMCON = U)
       TIMCON = T1
       IF (U.LE.0) TIMCON = T2
END
```

□

Ekološki sistem roparjev in žrtev in temperaturni regulacijski sistem smo simulirali s simulacijskim jezikom SIMCOS (Zupančič, 1992). Vendar bi bili programi v katerem koli drugem jeziku, ki spoštuje standard CSSL (npr. CSSL IV, ACSL) skoraj identični.

4.2.6 Primeri uporabe bločno orientiranega orodja z grafičnim vnosom modela - Matlab-Simulink

Uporabnost bločno orientiranih simulacijskih jezikov bomo prikazali na razvoju treh simulacijskih programov -model ekološkega sistema roparjev in žrtev, regulacijski sistem ogrevanja prostora in model avtomobilskega vzmetenja bomo simulirali v okolju Matlab-Simulink.

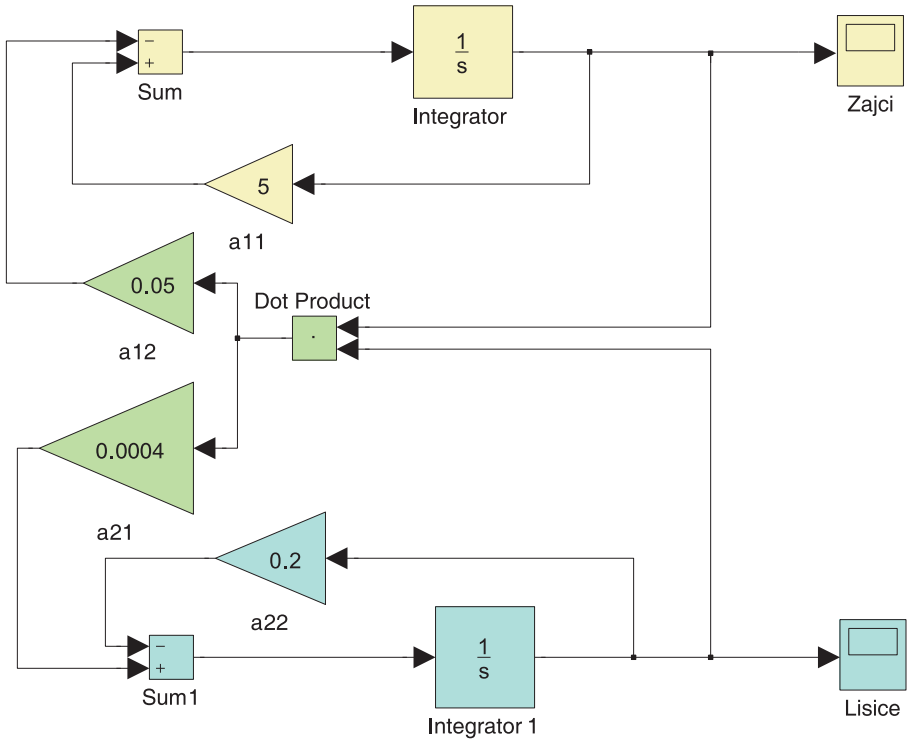
Primer 4.3 Ekološki sistem roparjev in žrtev

Primer 3.3 obravnava razvoj simulacijske sheme modela ekološkega sistema roparjev in žrtev, ki je predhodno opisan v Primeru 1.3. Model želimo simulirati pri konstantah $a_{11} = 5, a_{12} = 0.05, a_{21} = 0.0004, a_{22} = 0.2$ in pri začetnem številu zajcev in lisic $x_{10} = 520, x_{20} = 85$. V pomoč nam bo simulacijska shema, ki jo prikazuje slika 3.9 in enačbe (3.8). Ker okolje Matlab-Simulink zahteva vnos modela na grafični, bločno orientirani način, prenesemo na zaslon vse potrebne bloke, ki jih vsebuje slika 3.9. Potrebujemo dva integratorja (na zaslon jih povlečemo iz knjižničnega okna Continuous, privzeti imeni sta `Integrator` in `Integrator1`), štiri ojačevalne bloke (privzeta imena so `Gain`, `Gain1`, `Gain2` in `Gain3`, a smo ta imena v shemi nadomestili z imeni konstant `a11`, `a12`, `a21` in `a22`), dva sumatorja (privzeti imeni `Sum` in `Sum1`) in en množilnik (`DotProduct`). Vse te bloke prenesemo iz knjižničnega okna Math Operations. Na koncu iz knjižničnega okna Sinks dodamo dva prikazovalna bloka (privzeti imeni `Scope` in `Scope1` smo zamenjali z besedama `Zajci` in `Lisice`). Bloke razporedimo tako, kot kaže slika 3.9 in jih nato ustrezno povežemo (z miško potegnemo od izhoda bloka proti ustreznemu vhodu bloka). Ko je shema povezana, nastavimo parametre blokov z dvojnim klikom, ki odpre uporabniški vmesnik ustreznega bloka. Ojačevalnim blokom podamo vrednost ojačenja, integratorjema pa začetna pogoja. Ker je problem elementaren, ni potrebno nastaviti nobenih simulacijskih parametrov razen trajanja simulacijskega teka - vrednost 10 nastavimo v posebnem okencu v opravilni vrstici desno zgoraj. V kakšnih enotah je neodvisna spremenljivka, lahko ve le uporabnik, ki je problem tudi modeliral. Spomnimo se, da je enota neodvisne spremenljivke eno leto, torej traja simulacija deset let. Rezultati simulacije so enaki, kot na sliki 4.5. Opazimo, da je ekološka perioda približno sedem let, da število lisic najhitreje narašča ob največjem številu zajcev in najhitreje upada ob najmanjšem številu zajcev. Simulink shemo prikazuje slika 4.14.

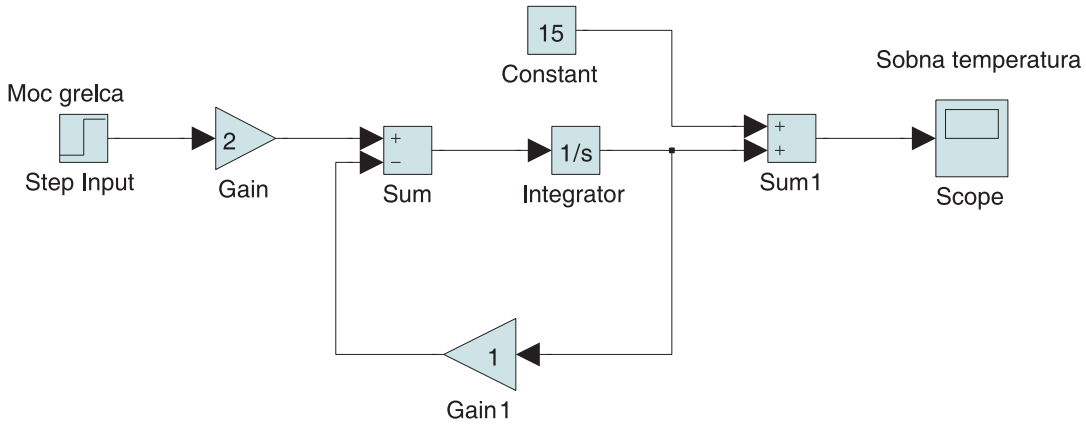
□

Primer 4.4 Regulacijski sistem ogrevanja prostora

Primer 3.1 obravnava razvoj simulacijske sheme temperaturnega procesa, ki je predhodno opisan v Primeru 1.2. Simulacijsko shemo razvijemo s pomočjo slike 3.5. Potrebujemo sumator, integrator in dva ojačevalna bloka. Zaradi veljavnosti enačbe (1.14) smo dodali še en sumator, s pomočjo katerega smo iz temperature v delovni točki in temperature okolice izračunali absolutno temperaturo. Simulink shemo prikazuje slika 4.15.



Slika 4.14: Simulacijska shema v Simulinku za ekološki primer zajcev in lisic



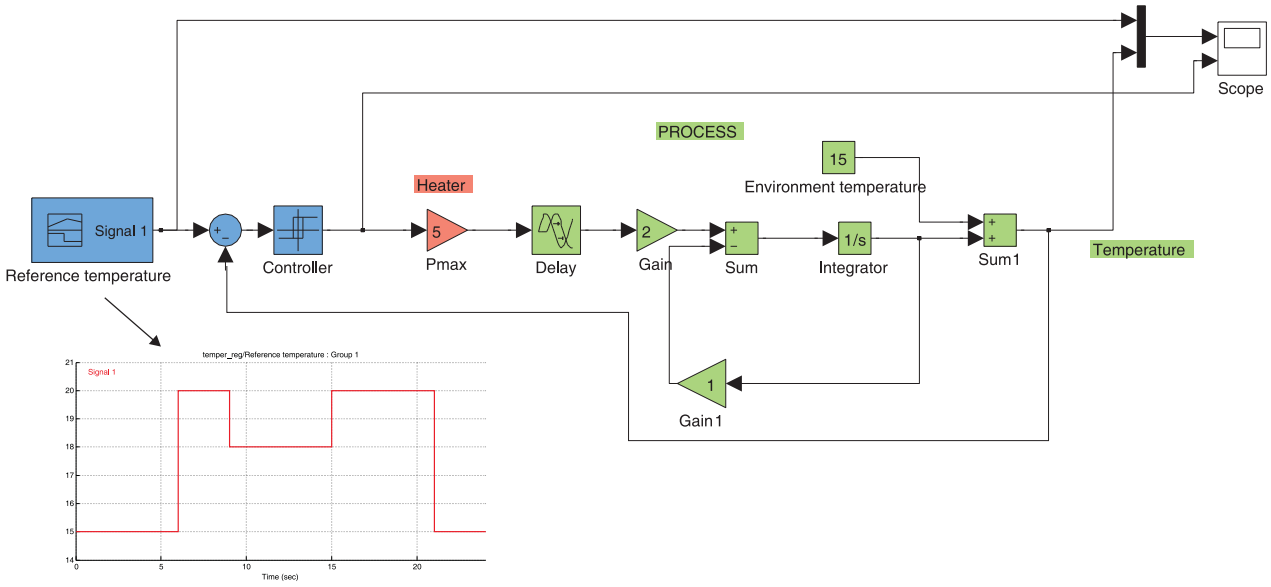
Slika 4.15: Simulacijska shema v Simulinku za model ogrevanja prostora

Simulacijski model procesa nadgradimo v shemo za regulacijo po sliki 1.6. Oporabimo blok `Signal builder` iz knjižnice Sources za generacijo spremenljivega referenčnega signala, ki ga opisuje tabela 4.3. Z blokom `Signal builder` opišemo referenčni signal s pomočjo grafičnega uporabniškega vmesnika. Iz knjižnice

Tabela 4.3: Želena sobna temperatura

t[h]	0	6	6	9	9	15	15	21	21
$\vartheta_r[^oC]$	15	15	20	20	18	18	20	20	15

Discontinuous uporabimo `Relay` blok (privzeto ime smo nadomestili z imenom `Controller`). Bloku podamo vrednosti, pri katerih vklopi oz. izklopi ter vklopno in izklopno vrednost. Grelo simuliramo z ojačevalnim blokom `pmax`, dodatno zakasnitev pa z blokom `Transport Delay` iz knjižnice Continuous. Simulink shemo prikazuje slika 4.16.



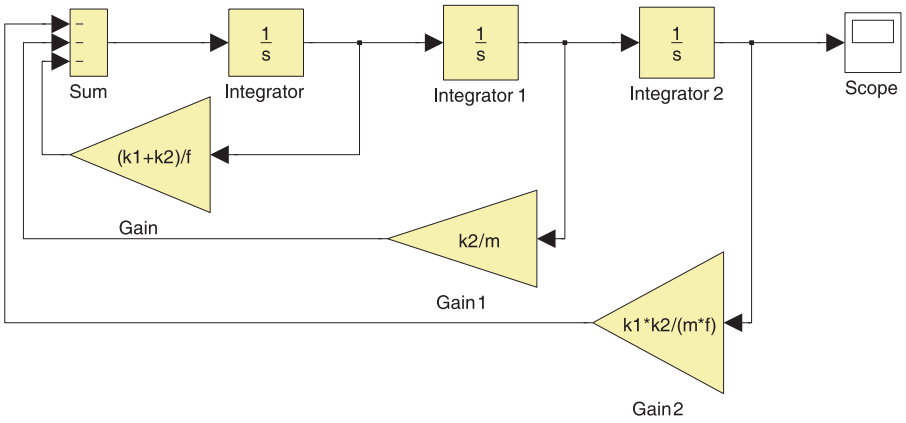
Slika 4.16: Simulacijaska shema regulacije temperature v prostoru in prikaz referenčnega signala

Temperatura v prostoru ter vklapljanje in izklapljanje grela sta prikazana na sliki 4.7. □

Primer 4.5 Avtomobilsko vzmetenje

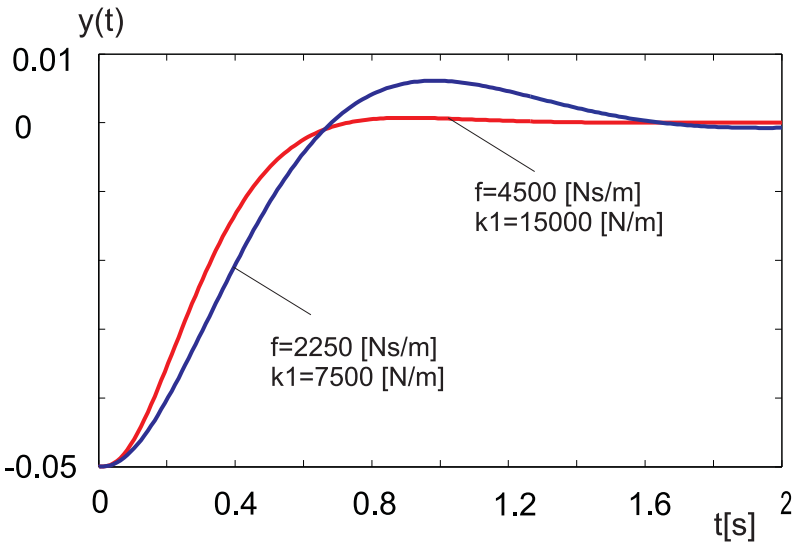
Primer 3.2 obravnava razvoj simulacijske sheme avtomobilskega vzmetenja, ki je predhodno opisan v Primeru 1.1. Simulacijsko shemo razvijemo s pomočjo

slike 3.7. Potrebujemo tri integratorje, tri ojačevalne bloke in sumator. Tokrat v ojačevalne bloke ne vpišemo številskih vrednosti konstant ampak kar aritmetične izraze, ki se prikažejo tudi znotraj blokovih ikon, če le te dovolj povečamo. Vrednosti za `k1`, `k2`, `m` in `f` pa podamo v komandnem oknu Matlaba. Simulacijsko shemo prikazuje slika 4.17.



Slika 4.17: Simulink shema modela avtomobilskega vzmetenja

Opazujemo pomik karoserije pri negativnem začetnem pogoju `y(0)=-0.05 m`. Praktično to lahko pomeni, da se je voznik vsedel v avtomobil in v trenutku t=0 izstopil iz vozila. Opazujemo, kako se karoserija vrne v mirovno lego. Rezultate simulacije prikazuje slika 4.18.



Slika 4.18: Pomik karoserije

Izhodiščni podatki za študijo so bili: m=500 kg, k1=7500 N/m, k2=150000 N/m, f=2250 Ns/m. Ker je prišlo do premalo dušenega odziva, smo povečali togost vzmeti na k1=15000 N/m (2x povečanje), dušenje pa smo tudi 2x povečali (f=4500 Ns/m). Tako smo dobili optimalnejše delovanje avtomobilskega vzmetenja.

□

5.

Jeziki za simulacijo zveznih dinamičnih sistemov

V poglavju bomo podali pregled razvoja simulacijskih jezikov, programsko zgradbo simulacijskih jezikov ter opisali možnosti simulacije z enačbno in bločno orientiranimi simulacijskimi jeziki ter s splošnonamenskimi programskimi jeziki.

5.1 Pregled razvoja simulacijskih jezikov

V razvoju jezikov za simulacijo dinamičnih sistemov je pomembno leto 1967, ko je bil sprejet standard za t.i. simulacijske jezike CSSL (Continuous System Simulation Language). Zato bomo pregled razdelili v obdobji pred in po letu 1967.

5.1.1 Razvoj simulacijskih jezikov pred sprejetjem standarda

Prve korake v digitalno simulacijo so naredili uporabniki analognih računalnikov. Na digitalnih računalnikih so želeli reševati probleme na podoben način kot na analognih. Zato so bili prvi jeziki podobni konceptom analognega računanja.

Za začetek razvoja digitalnih simulacijskih jezikov jemljemo leto 1955, ko je Selfridge prvi podal idejo o bločnem simulacijskem jeziku. Leta 1959 je tako prišel v uporabo prvi simulacijski jezik SELFRIDGE. Uporabnik je moral spremenljivke podajati s številkami, jezik ni vseboval vrstnega algoritma, za integracijo pa je bila uporabljena neprimerna Simpsonova metoda. Istega leta so Stein, Rose in Parker prvi podali idejo prevajalniškega simulacijskega jezika. Jezik MIDAS, ki so ga razvili pri IBM leta 1963 pa predstavlja prvi zares uporabni simulacijski sistem. Za spremenljivke in parametre je uporabljal alfanumerične oznake, vseboval je tudi vrstni algoritem. Za integracijo je uporabljal Milnejevo metodo 4. reda vrste prediktor-korektor s spremenljivim računskim korakom.

Od tedaj dalje lahko zasledujemo izredno hiter razvoj simulacijskih jezikov. V letu 1964 se pojavi jezik COBLOC (univerza Wisconsin) za računalnik CDC 1604. Vseboval je že tri integracijske algoritme in v nasprotju z dotedanjimi jeziki, ki so zahtevali točno določen format pisanja programa, omogoča že prosti format. Istega leta je prišel na trg tudi PACTOLUS za računalnik IBM 1620, ki je bil predhodnik enega kasneje najuspešnejših simulacijskih jezikov CSMP (Continuous System Modelling Program). Prva verzija jezika CSMP je bila prirejena za računalnik IBM 1130 (CSMP 1130). Za večje tipe računalnikov IBM 360/370 pa so leta 1965 predelali sicer že v letu 1959 razviti simulacijski jezik DYNAMO. Jezik je imel že dokaj sposobno diagnostiko, enostaven makro jezik, prikaz rezultatov v obliki tabel in grafov, odkril je celo algebrajske zanke. Vendar pa je imel le predpisani format podajanja modela. Za integracijo je uporabljal samo Eulerjev integracijski algoritem.

Zaradi precej omejenih sposobnosti takratnih računalnikov, pomanjkljivih programerskih izkušenj in zlasti neustreznih numeričnih metod so imeli omenjeni jeziki veliko slabosti:

- Bili so povsem bločno orientirani, zato je moral uporabnik izdelati program na osnovi bločne sheme podobno kot na analognem računalniku, odpadlo je le normiranje. Zato je bilo veliko možnosti za napake.
- Jeziki niso imeli možnosti za razširitve.
- Programi so bili nepregledni in zato manj primerni za dokumentiranje modela.
- Simulacija je bila numerično precej nezanesljiva.

Leta 1965 je IBM razvil jezik DSL 90 (Dynamic Simulation Language) za raču-

nalnik IBM 7090. Le-ta predstavlja pomemben mejnik v razvoju simulacijskih jezikov, ker je prvi primer prevajalniškega, enačbno orientiranega jezika. Vseboval je tudi že t.i. blok PROCEDURE, s katerim je bilo možno blok modela opisati s stavki v jeziku FORTRAN. Iz jezika DSL 90 so leta 1967 razvili prvi sposobni simulacijski jezik za hitre in velike računalnike - CSMP 360. Le-ta je omogočal simulacijo kompleksnih sistemov. Istega leta se je pojavila še verzija CSMP III, ki je omogočala tudi uporabo grafičnega terminala IBM 2250. S to za takrat zelo moderno verzijo je IBM brez večjih modifikacij obvladal trg kar do leta 1978. Za oba jezika je značilno, da sta prevajalniškega tipa, ciljni jezik pa je FORTRAN. Uporabnik je na ta način pri simulaciji lahko uporabljal tudi vse zmožnosti jezika FORTRAN. Oba jezika sta vsebovala že funkcijske generatorje, makro jezik, možnost reševanja algebrajske zanke, sedem integracijskih metod in približno 60 operatorjev za opis modela. Verzije za manjše računalniške sisteme (IBM 1130, IBM 1180) pa so se imenovale DSL 1130/1180 in predstavljajo prve zametke interaktivnosti v simulaciji. S pomočjo funkcijskih stikal je bilo možno prekiniti simulacijski tek, spremeniti parametre in ponovno začeti s simulacijo. Na računalnik je bilo možno priključiti digitalni risalnik in ustrezni osciloskop.

Zaradi razvoja številnih simulacijskih jezikov so se sredi šestdesetih let pojavile težnje po standardizaciji jezikov za simulacijo zveznih dinamičnih sistemov. Jeziki so bili precej nezanesljivi, neprenosljivi in interaktivnost je bila prisotna le v zametkih. Njihova glavna slabost pa je bila, da so bili simulacijski modeli zaradi zelo različnih opisov povsem neprenosljivi. Zato je že leta 1965 Simulation Council (SCi), ki je bil predhodnik današnje Society for Computer Simulation (SCS) ustanovil standardizacijski komite, ki je leta 1967 izdelal predlog standarda (Strauss ,1967), ki je dobil popularno ime standard CSSL 67 (Continuous System Simulation Language). Na koncept tega standarda so precej vplivali do takrat razviti simulacijski jeziki, predvsem MIDAS in DSL 90.

Zaradi izredno dobre zasnove standarda, je le ta v uporabi kar dve desetletji in še danes so komercialno najuspešnejši jeziki osnovani na standardu CSSL 67.

5.1.2 Standard za zvezne simulacijske jezike

Standard CSSL 67 je imel tri glavne cilje:

- Zagotovil naj bi enostavno simulacijsko programsko opremo tudi za uporabnike brez večjih izkušenj. Zato so predvideli razumljivo sintakso za opis

diferencialnih enačb, blokov, diagnostiko napak, sortiranje blokov ter komande za interaktivno simulacijo.

- Izkušenemu uporabniku naj bi jezik predstavljal fleksibilno orodje za modeliranje in simulacijo večjih in kompleksnih sistemov. Zato so se odločili za prevajalniški tip jezika, kar daje uporabniku možnost, da lahko dodaja kompleksne operacije v ciljnem jeziku prevajalnika. To enostavno dodajanje novih operacij je omogočalo odprtost jezikov.
- Zaradi predvidenega tehnološkega razvoja (grafika, interaktivni računalniški sistemi) naj bi standard omogočal fleksibilne razširitve. To je kasneje omogočalo vključiti v jezike več novih lastnosti, kar je povzročilo, da je standard CSSL 67 ostal tako dolgo v veljavi. Zato pa so nastali tudi številni CSSL dialekti, kar je v osemdesetih letih pripeljalo do teženj po novem standardu.

Standard CSSL 67 vsebuje strukturne in funkcionalne elemente.

Strukturni elementi

Še preden so bili sploh znani pojmi strukturnega programiranja in podatkovnih struktur, je standard CSSL 67 že vseboval take elemente, kot so:

- Elementi, ki omogočajo razvrščanje (zaradi te lastnosti je program navidezno paralelen).
- Makro jezik, ki omogoča, da enake dele programa ne vnašamo večkrat.
- Modelne sekcije, ki lahko uporabljajo tudi različne integracijske metode.
- Strukturni opis modela s sekcijami INITIAL (proceduralni program, ki se izvrši pred simulacijskim tekom), DYNAMIC (neproceduralni del za opis modela) in TERMINAL (proceduralni program, ki se izvrši po simulaciji).
- Možnost za krmiljenje izvajanja simulacije. Standard je že predvidel dve možnosti: s programom v ciljnem jeziku prevajalnika, ki kliče simulacijski tek kot podprogram ali pa z uporabo interaktivnega komandnega jezika.

Te strukturne lastnosti so bile vsaj dve desetletji ustrezne, pozneje pa so začele danes utesnjevati jezike.

Funkcionalni elementi

Standard CSSL 67 je definiral samo osnovne funkcionalne elemente. Za opis modela so predvideli integrirni operator, diferencirni operator, implicitno reševanje algebrajske zanke, operator zakasnitve in nelinearnosti. Predvidene so bile konstante, navadne in indeksirane spremenljivke ter stavki za krmiljenje integracije (vrste metode, dolžina računskega koraka, dopustni pogreški) in simulacije (dolžina komunikacijskega intervala, dolžina simulacijskega teka, izbira načina prikaza rezultatov).

5.1.3 Razvoj simulacijskih jezikov po sprejetju standarda

Standard CSSL 67 je močno vplival na nadaljnji razvoj prevajalniško orientiranih simulacijskih jezikov. Jeziki, ki so se držali omenjenega standarda, so bili vse do danes tudi komercialno najuspešnejši.

Leta 1968 so pri podjetju CDC iz MIDASA razvili MIMIC, ki je predstavljal prvi simulacijski jezik po vzoru standarda CSSL 67. Vendar je jezik precej zaostajal za obstoječimi jeziki družine CSMP. Ni imel prostega formata za opis modela in diagnostika je bila zelo slaba. Jezik je vseboval t.i. logične krmilne spremenljivke, ki so omogočale izvajanje določenih delov modela ob izpolnitvi zahtevanih pogojev.

Leta 1969 pa je prišel v uporabo CSSL III, prvi jezik, ki je strogo upošteval navodila standarda. Ta jezik je skupno z nadaljnjimi verzijami (leta 1972 CSSL IV) naslednjih petnajst let močno vplival na razvoj digitalnih simulacijskih jezikov. Imel je prosti format za podajanje modela, bil je prevajalniški enačbni jezik in je vseboval veliko funkcij in operatorjev. Model je bilo možno podajati s strukturnimi sekcijami INITIAL, DYNAMIC in TERMINAL, možno pa je bilo uporabljati sposoben a uporabniško neprijazen makro jezik. Uporabnik je lahko izbiral med sedmimi numerično že dokaj robustnimi integracijskimi metodami, lahko pa je vključil tudi lastno metodo. Kasnejše verzije so dobile tudi komandni interaktivni jezik za krmiljenje simulacijskih tekov. Jezik je v začetku dobavljal CDC za svoje računalnike CDC 6000/7000, kasneje pa je razvoj prevzel Simulation Service. Zaradi velike sposobnosti in relativne enostavnosti za uporabo je CSSL IV kmalu dobil veliko privržencev. Jezik se je sicer nenehno razvijal, a le funkcionalno. Strukturno pa se še današnja verzija, ki predstavlja enega najsposobnejših simulacijskih jezikov, ne loči mnogo od prvih verzij (Nilsen, 1984).

Standard CSSL 67 je upošteval tudi simulacijski jezik SL-1 (l. 1970) za računalnik Sigma XDS. Zaradi vezanosti na eno vrsto računalnikov se ni dolgo uporabljal. Imel pa je nekaj posebnih lastnosti. Deloval je kot dvojni prevajalnik (v FORTRAN in nato v zbirnik). V različnih delih modela je bilo možno uporabiti različne integracijske postopke. Možno je bilo uporabiti tudi več neodvisnih spremenljivk, kar je omogočalo reševanje parcialnih diferencialnih enačb. Najbolj pa je zanimivo, da je jezik SL-1 omogočal sinhronizacijo z realnim časom, prekinitve ter uporabo pretvornikov A/D in D/A, torej simulacijo v realnem času. Bolj kot širša uporabnost tega jezika so bile pomembne nove ideje.

Podjetje IBM, ki je imelo sredi šestdesetih let vodilno vlogo na področju digitalne simulacije zveznih dinamičnih sistemov, je postalo po nastanku sposobnih jezikov CSSL precej neaktivno na področju simulacije. Leta 1972 so sicer izdali novo verzijo jezika CSMP III, ki pa je ostala pri starih konceptih in ni strožje upoštevala standarda. Verzija je bila primerna samo za računalnike IBM in je omogočala tudi uporabo grafičnega terminala IBM 2250. Glavna prednost jezika CSMP III in tudi kasnejših dialektov je bila v izredno dobrih zmožnostih za grafično predstavitev rezultatov. Jezik CSMP je precej pred drugimi jeziki omogočal grafično podajanje časovnih odzivov, faznih trajektorij, združevanje rezultatov različnih simulacijskih tekov in celo trodimenzionalno prikazovanje ter senčenje.

Simulacijski jeziki so imeli v začetku sedemdesetih let še vedno izredno slabe interaktivne zmožnosti in so v glavnem delovali v t.i. paketnem načinu obdelave na velikih računalnikih, čeprav so nekatere vrste računalnikov že omogočale določeno interaktivnosti. Eden prvih simulacijskih jezikov, kjer so načrtovalci dali večji poudarek interaktivnosti, je simulacijski jezik SIMNON (SIMulation progam for NONlinear systems, Lund University of Technology). Prva verzija tega jezika je bila razvita l. 1972, prva uporabna verzija pa je prišla na trg leta 1975. Jezik ni upošteval standarda CSSL 67. Interaktivno zmožnost so jeziku povečali tudi tako, da so izvedli prevajanje direktno na nivo strojnega jezika. Zaradi lažje prenosljivosti so to prevajanje realizirali v dveh delih. V prvem delu se izvorni program prevede v kodo virtualnega procesorja, v drugem delu, ki je specifičen za določeno implementacijo, pa v strojno kodo določenega procesorja. Ker je prevajalnik relativno hiter, je možno na interaktivni način spreminjati tudi strukturo modela. SIMNON predstavlja prvi simulacijski jezik, ki je imel moderno zasnovo interaktivnega komandnega jezika za krmiljenje simulacijskih tekov. Jezik je bil relativno enostaven za uporabo, ni poznal indeksiranih spremenljivk, imel pa je novost v opisu modela z diferencialnimi enačbami. Odvode v diferencialnih enačbah je bilo možno označiti povsem matematično (npr. y', y'') in jih v opisu modela ni bilo potrebno integrirati. Ta princip sedaj uporabljajo nekateri najsodobnejši simulacijski jeziki. Slabost tega načina pa je v tem, da mod-

ela ni možno normirati, kar je pri numerično zahtevnih problemih lahko zelo pomembno. Kasnejše verzije jezika SIMNON so omogočale vključevanje modulov v jeziku FORTRAN, zapise z diferenčnimi enačbami, povezovanje več modelov ter optimizacijo. SIMNON še danes predstavlja uspešen simulacijski jezik (Åström, 1985a).

Leta 1972 smo na Fakulteti za elektrotehniko v Ljubljani razvili simulacijski jezik HYSIM (HYbrid SIMulation, Divjak, 1975) za računalnik IBM 1130. Jezik je bil v tistem času izredno napreden, saj je vseboval tudi elemente kombinirane simulacije.

Od leta 1972 naprej se je začela razvijati tudi družina zelo naprednih simulacijskih jezikov DARE (Korn, Wait, 1978). Jeziki se niso strogo držali standarda CSSL 67, zato pa so prinašali precej novosti. Leta 1975 je prišla v uporabo verzija DARE P, ki je delovala sicer še na paketni način, a je predstavljala prvi relativno dobro prenosljiv simulacijski jezik, ki je deloval tudi na miniračunalnikih. Namesto sekcij INITIAL, DYNAMIC in TERMINAL je imel DARE P t.i. blok LOGIC, v katerem je uporabnik s pomočjo proceduralnega programa opisal krmiljenje simulacijskih tekov. Ta način je predstavljal prvi poizkus ločitve modela in eksperimenta. V nekaterih drugih dialektih jezika DARE so bile vgrajene določene rešitve, ki so povečale učinkovitost simulacije v realnem času (DARE/ELEVEN). Uporabljali so bločni princip, saj je bločno strukturo možno relativno enostavno prevajati v zbirnik. To je bilo zlasti učinkovito, ko so se pojavili sposobni makro prevajalniki za zbirne jezike. Pokazalo se je, da je na ta način možno simulirati probleme štiri do desetkrat hitreje kot z prevajanjem v FORTRAN. Še večji časovni prihranek pa je prinesla uporaba aritmetike s fiksno decimalno vejico (cca. 4krat), kar seveda v jeziku FORTRAN ni možno. Uporabnik je v DARE/ELEVEN lahko kombiniral simulacijske segmente z enačbnim načinom ter aritmetiko s pomično vejico in bločne segmente z aritmetiko z fiksno decimalno vejico za simulacijo časovno kritičnih operacij (nekakšna simulacija hibridnega sistema). Nekateri naslednji dialekti (MICRODARE, DESIRE, DESKTOP), ki so se pojavili okoli leta 1980, pa so imeli t.i. ultra hitre prevajalnike, ki so prevajali samo del programa, ki opisuje model, na strojni nivo (Korn, 1983b). Taki jeziki so znani kot jeziki z direktnim izvajanjem, saj uporabnik pri delu praktično ne opazi prevajanja. Ti jeziki so vsebovali tudi zelo dodelane t.i. grafične postprocesorje, ki so ločeni od simulacije omogočali visoko stopnjo interaktivnosti. V nekaterih ozirih so prekašali celo grafiko na sistemih CSMP.

Hitrost izvajanja simulacijskih programov je predstavljala v sedemdesetih letih zelo aktualno problematiko. Zato so nastale prve ideje, da bi se simulacijski jeziki uporabljali v povezavi s hibridnimi računalniki. Prvi jezik, ki je kombiniral

programske učinkovitosti jezikov CSSL in računske zmožnosti hibridnih računalnikov je bil jezik HL-1 (leta 1973). To so bili prvi zametki konceptov, ki so postali resnično uporabni šele čez deset let.

Leta 1975 je prišel na trg danes komercialno najuspešnejši simulacijski jezik ACSL (Advanced Continuous Simulation Language - Mitchel & Gauthier Ass.). Jezik temelji na standardu CSSL 67, vsebuje pa tudi veliko izkušenj iz razvoja jezika CSSL IV in ima podobne karakteristike. Z dodatno modelno sekcijo DISCRETE so zelo povečali uporabnost jezika za simulacijo kombiniranih sistemov (Mitchel & Gauthier, 1981). V primerjavi z drugimi simulacijskimi jeziki se je ACSL najhitreje prilagodil na različne računalnike. V sredini osemdesetih let so bile dosegljive verzije na številnih tipih računalnikov (od osebnega računalnika do superračunalnika CRAY).

Pojavile so se tudi potrebe po večji računalniški podpori v fazi modeliranja. Jezik DYMOLA (Elmqvist, 1978) je omogočal opis modela v obliki fizikalnih zakonov (npr. Kirchoffove enačbe)), zapis v prostoru stanj pa se je izračunal avtomatično. Zaradi načina povezovanja podmodelov, ki je bilo bolj splošno, kot povezovanje preko vhodov in izhodov, je DYMOLA eno od prvih objektno orientiranih orodij. Računalniške sposobnosti pa so bile takrat še preslabe, da bi koncepti resnično zaživeli.

Sedemdeseta leta so torej prinesla velik napredek na področju digitalne simulacije. Jeziki so že vključevali interaktivne zmožnosti računalnikov in njihove grafične sposobnosti. Velik napredek so omogočile tudi nove numerične metode (knjižnice EISPACK, LINPACK), kar je omogočilo večjo numerično robustnost simulacijskih jezikov.

Osemdeseta leta so prinesla pomemben nadaljnji razvoj omenjenih značilnosti. V zvezi z interaktivnimi zmožnostmi so se razvili vsi znani načini dialoga: vprašanje-odgovor, menujski način in komandni način. Kot zelo uporaben eksperiment se je začela uporabljati optimizacija. Ker so bili obstoječi jeziki strukturno omejeni, so jih v glavnem širili z novimi funkcijami. Tako so začeli v nekatere simulacijske jezike (ACSL, CSSL) vgrajevati postopke iz sistemske teorije kot npr. Bodejev in Nyquistov diagram, diagram lege korenov, hitro Fourier-jevo transformacijo, analizo ustaljenega stanja, linearizacijo itd. Na ta način so jeziki začeli preraščati v pakete CACSD. V začetku osemdesetih let so se pokazale tudi potrebe po prenosu simulacijskih jezikov na mini in predvsem mikroračunalnike. Le-ti so postali toliko sposobni (velik pomnilnik, prevajalniki za FORTRAN, PASCAL, C), da prenos posameznih sicer osiromašenih verzij jezikov z večjih sistemov ni delal resnejših težav. Prav simulacijski jeziki na mikroračunalnikih so šele omogočili,

da je simulacija postala dostopna vsakomur.

Zelo sposoben jezik za mikroračunalnike predstavlja ISIM, ki so ga razvili iz jezika ISIS (Crosbie, 1984, Hay, Crosbie, 1984). Jezik se strogo drži standarda CSSL 67, deluje pa na osem in šestnajst bitnih računalnikih pod operacijskima sistemoma CP/M in MS-DOS. Prevajalnik prevede izvorni program v vmesno obliko, ki se potem izvaja na interpreterski način. Zato ima jezik zelo dobre interaktivne zmožnosti. Ker pa ga ni mogoče kombinirati z jezikom FORTRAN, so v sintakso vnesli dodatne stavke kot npr. DO, IF in GOTO. Diferencialne enačbe se direktno vnašajo, odvodov pa v izvornem programu ni potrebno integrirati. Jezik omogoča tudi enostavno eksperimentiranje s proceduralnim programom. Nekatere nadaljnje verzije vsebujejo tudi prve poizkuse modularnega programiranja.

Iz tega obdobja je tudi jezik TUTSIM (Twente University of Technology (1980), TUTSIM, 1983), ki deluje na operacijskih sistemih CP/M in MS-DOS. Predstavlja interpreterski bločno orientirani jezik, napisan v zbirniku. Je enostaven za uporabo, a neprimeren za reševanje kompleksnih problemov. Model je možno podati tudi z bond grafi. Za današnje razmere pa vsebuje relativno slabe integracijske metode.

Podobne lastnosti ima tudi interpreterski bločno orientirani jezik PSI (Delft University of Technology (1983), Van den Bosch, 1987). Slabe lastnosti interpreterskih jezikov skuša odpraviti z velikim številom blokov za opis modela (cca 60), ima pa tudi dodatni uporabniški blok, tako da lahko uporabnik po določenih pravilih v jeziku FORTRAN napiše svoj blok, ki po prevajanju in povezovanju postane standardni PSI blok. Ima tudi sposoben komandni jezik za interaktivno krmiljenje simulacijskih tekov (100 komand). Omogoča pa tudi optimizacijo.

Od simulacijskih jezikov na večjih računalnikih prinaša novost jezik DSL/VS podjetja IBM iz leta 1984 (Syn, Dost, 1985). Tako se je IBM po skoraj 15 letih spet bolj intenzivno vključil v razvoj simulacijskih jezikov, čeprav je vseskozi za svoje lastne potrebe veliko uporabljal in tudi sproti posodabljal simulacijske jezike CSMP in DSL. Tudi novo verzijo, ki v glavnem temelji na predhodnih jezikih, so v glavnem razvili za lastne potrebe. Jezik deluje na računalnikih 370 in 4300, je vsestransko sposoben in omogoča reševanje kompleksnih problemov. Poleg običajnih sekcij za opis modela INITIAL, DYNAMIC in TERMINAL so dodali tudi sekcijo SAMPLE, ki je namenjena predvsem za opis diskretnih regulatorjev. Jezik vsebuje več sposobnih integracijskih postopkov, vključena je tudi optimizacija. Kot je značilno za vse predhodne jezike CSMP, ima zelo sposobno grafiko. Za delo na osebnih računalnikih so razvili jezik PCESP (Personal Com-

puter Engineering Simulation Program, Shah, 1988) iz predhodnega jezika DSL 1130. Le-ta je kompatibilen z DSL/VS, tako da je možno probleme razvijati na PC računalniku (kot na delovni postaji), po potrebi pa je možno simulacijo izvajati na velikem računalniku. Podoben je tudi jezik SYSL/M (90% kompatibilen s CSMP).

V osemdesetih letih so nastali tudi simulacijski jeziki za delo na sodobnih hibridnih sistemih. Jezika HYBSIS (Kleinert in ostali, 1983) in STARTRAN (Landauer, 1988) smo že omenili pri pregledu hibridnih sistemov. Znani pa so tudi simulacijski jeziki na namenskih simulacijskih digitalnih računalnikih kot npr. ADSIM (Grierson, 1986) in PARSIM (Bruijn, Soppers, 1986), ki smo jih omenili pri opisu simulacijskih sistemov na namenskih računalnikih.

V začetku osemdesetih let so se začele kazati potrebe po novem standardu CSSL, ki ne bi več strukturno utesnjeval jezikov ampak bi upošteval sodobne koncepte programskega inženirstva. Toda zaradi zelo različnih interesov sta dve delovni skupini (pri IMACS in SCS) uspeli izdelati le priporočila. Na osnovi teh priporočil so se sredi osemdesetih let začeli razvijati simulacijski jeziki nove generacije (ESL, SYSMOD, COSMOS). Ti jeziki so se razvijali pod močnim vplivom programskega inženirstva in imajo predvsem naslednje pomembne lastnosti:

- modularnost pri opisu modela (hierarhična gradnja modela iz podmodelov),
- vgradnja bolj kompleksnih eksperimentov (razen simulacijskega teka omogočajo jeziki še optimizacijo, linearizacijo, parametrizacijo, izračun ustaljenega stanja, analizo občutljivosti, analize v frekvenčnem prostoru, ...),
- fleksibilne programske in podatkovne strukture,
- značilnosti kombinirane simulacije,
- numerična robustnost (algoritmi za integracijo, optimizacijo, obdelava nezveznosti,...).

Konec osemdesetih let smo tudi na Fakulteti za elektrotehniko in računalništvo v Ljubljani (v sodelovanju z Institutom Jožef Stefan) razvili in dali v uporabo simulacijski jezik tipa CSSL z imenom SIMCOS (SIMulation of COntinuous Systems) (Zupančič, 1989, Zupančič, 1992). Jezik omogoča simulacijo, optimizacijo, linearizacijo in parametrizacijo zveznih in diskretnih dinamičnih sistemov. Omogoča tudi simulacijo v realnem času.

Devetdeseta leta so prinesla velik napredek predvsem pri razvoju uporabniških vmesnikov simulacijskih orodij. Tako na osebnih računalnikih kot na sposobnih delovnih postajah prevladujejo koncepti okenskih vmesnikov. Nesluten razvoj je naredilo programsko okolje MATLAB tudi na področju simulacij zlasti s paketom SIMULINK (Simulink, 2009). MATLAB-SIMULINK je postalo standardno okolje na vseh akademskih institucijah, kasneje pa se je izdatno začelo uporabljati tudi v industriji. Kasneje so za SIMULINK razvili razne namensko uporabne dodatke, eno je npr. STATE FLOW, ki omogoča vključevanje dogodkov. Velik je tudi poudarek na objektni orientiranosti (npr.Xmath). Nekatera orodja uvajajo tudi večjo podporo v smislu modeliranja (npr. DYMOLA z orodji DYMODRAW, DYMOVIEW, DYMOSIM (Elmqvist, 1994, Cellier, 1991).

V novem tisočletju je simulacija najbolj zaznamovana z objektno orientiranim, fizikalnim in več domenskim jezikom Modelica (Fritzson, 2004, Modelica, 2010, Sodja, Zupančič, 2009). Podobno idejo fizikalnega modeliranja vsebujejo tudi Bond grafi - grafična modelerska tehnika, ki opisuje pretok energije med komponentami (pristop podpira npr. simulacijski paket 20-sim). Modelica postaja priznani standard za modeliranje zveznih pa tudi diskretnih in hibridnih dinamičnih sistemov. Omogoča zlasti veliko podporo za modeliranje, saj ni potrebno izraziti odvodov stanj, kot v primeru večine konvencionalnih simulacijskih orodij. Zlasti je pomemben način povezovanja komponent, ki omogoči gradnjo knjižnic ponovno uporabljivih komponent. Povezujemo pa lahko komponente različnih področij, kar je zlasti pomembno v mehatroniki, robotiki, v avtomobilski industriji, v vodenju sistemov ipd. Zlasti sposobni okolji, ki podpirata jezik Modelica, sta Dymola (Dymola, 2011) in MathModelica. Podoben način uvaja tudi podjetje MathWorks - l. 2008 je prišlo na trg okolje Simscape v sklopu programskega paketa Matlab-Simulink. Žal pa ta način upošteva jezik Modelica le kot idejni koncept.

V tem pregledu smo se omejili predvsem na simulacijske jezike, ki so največ prispevali k napredku in uporabi simulacijskih postopkov. Izpustili smo simulacijske zmožnosti paketov za računalniško podprto načrtovanje vodenja sistemov. Le-ti imajo v mnogočem podobne lastnosti, kot obravnavani jeziki in so se tudi razvijali pod njihovimi vplivi. Pregled teh orodij smo podali v podpoglavju 1.8.

Tabela 5.1 predstavlja v skrčeni obliki glavne karaketeristike razvoja simulacijskih jezikov za simulacijo dinamičnih sistemov.

Tabela 5.1: Razvoj in glavne karakteristike simulacijskih jezikov

1955	SELFRIDGE	prvi simulacijski jezik
1956		prevajalniški koncept (Stein,Rose, Parker)
1963	MIDAS	prvi sposobnejši jezik, vrstni algoritem, integracijska metoda spremenljivega koraka
1964	COBLOC	več integracijskih metod, prosti format pisanja programa
1965	DYNAMO	diagnostika, makro jezik, prikaz rezultatov v obliki grafa, odkrije algebrajsko zanko
1965	DSL 90	prvi prevajalniški enačbno orientirani jezik
1967	CSMP 360, CSMP III	prvi sposobni simulacijski jeziki, funkcijski generatorji, 60 operatorjev, reševanje algebrajske zanke
1967	DSL 1130/1180	prve interaktivne zmožnosti, možna uporaba digitalnega risalnika, spominskega osciloskopa
1967		**standard CSSL 67**
1968	MIMIC	prvi jezik po vzoru CSSL 67
1969	CSSL III	prvi sposobni jezik CSSL
1970	SL-1	prvi jezik za simulacijo v realnem času
1972	CSSL IV	vse do danes eden najsposobnejših simulacijskih jezikov
1972	CSMP III	učinkovit grafični prikaz rezultatov
1972	HYSIM	kombinirana simulacija
1973	HL-1	prvi jezik za programiranje hibridnega sistema
1975	SIMNON	sposoben interaktivni jezik, prevajanje direktno na strojni nivo, uporabniku ni potrebno integrirati odvodov
1975	ACSL	vsestransko dober, danes komercialno najuspešnejši simulacijski jezik
1975	DARE P	prvi jezik za miniračunalnik, prenosljiv, poskus ločitve modela in eksperimenta
1976	DARE/ELEVEN	kombinacija enačbnega in bločnega jezika za hitro simulacijo v realnem času
1978	DYMOLA	nadgradnja za modeliranje, objektna orientiranost
1980	MICRODARE, DESIRE DESKTOP	ultra hitri prevajalnik
1981		nova priporočila CSSL 81 (SCS, IMACS)
1983	ISIM	na osem in šestnajst bitnih mikroračunalnikih, modularno programiranje
1983	HYBSIS, STARTRAN	simulacijska jezika za hibridne sisteme, simulacija v realnem času
1984	ADSIM, PARSIM	simulacijska jezika za namenske simulacijske računalnike, simulacija v realnem času
1984	ESL, SYSMOD, COSMOS	simulacijski jeziki nove generacije
1989	SIMCOS	zvezna in diskretna simulacija, eksperimentiranje, delovanje v realnem času
1990	SIMULINK	grafični uporabniški vmesnik, delovanje v okolju MATLAB
1992	okolje DYMOLA	podpora za modeliranje, objektna orientiranost
1995	20-sim	fizikalno modeliranje, uporaba Bond grafov
1996	jezik MODELICA	poizkus standardizacije jezika za objektno orientirano več domensko modeliranje - nov standard po CSSL'67
2008	Simscape	rešitev podjetja Mathworks za več domensko objektno orientirano modeliranje

5.2 Programska zgradba zveznih simulacijskih jezikov

Pod izrazom simulacijski jezik si običajno ne predstavljamo le jezika za podajanje modela, ampak celotni programski paket, ki omogoča simulacijo. Celotno programsko zgradbo sodobnih jezikov običajno razdelimo v:

- del za opis eksperimenta t.j. modela in metode,
- procesor,
- sistem za izvajanje simulacije oz. eksperimenta (simulator),
- postprocesor za grafično predstavitev rezultatov,
- nadzorni program.

Celotno programsko zgradbo prikazuje slika 5.1.

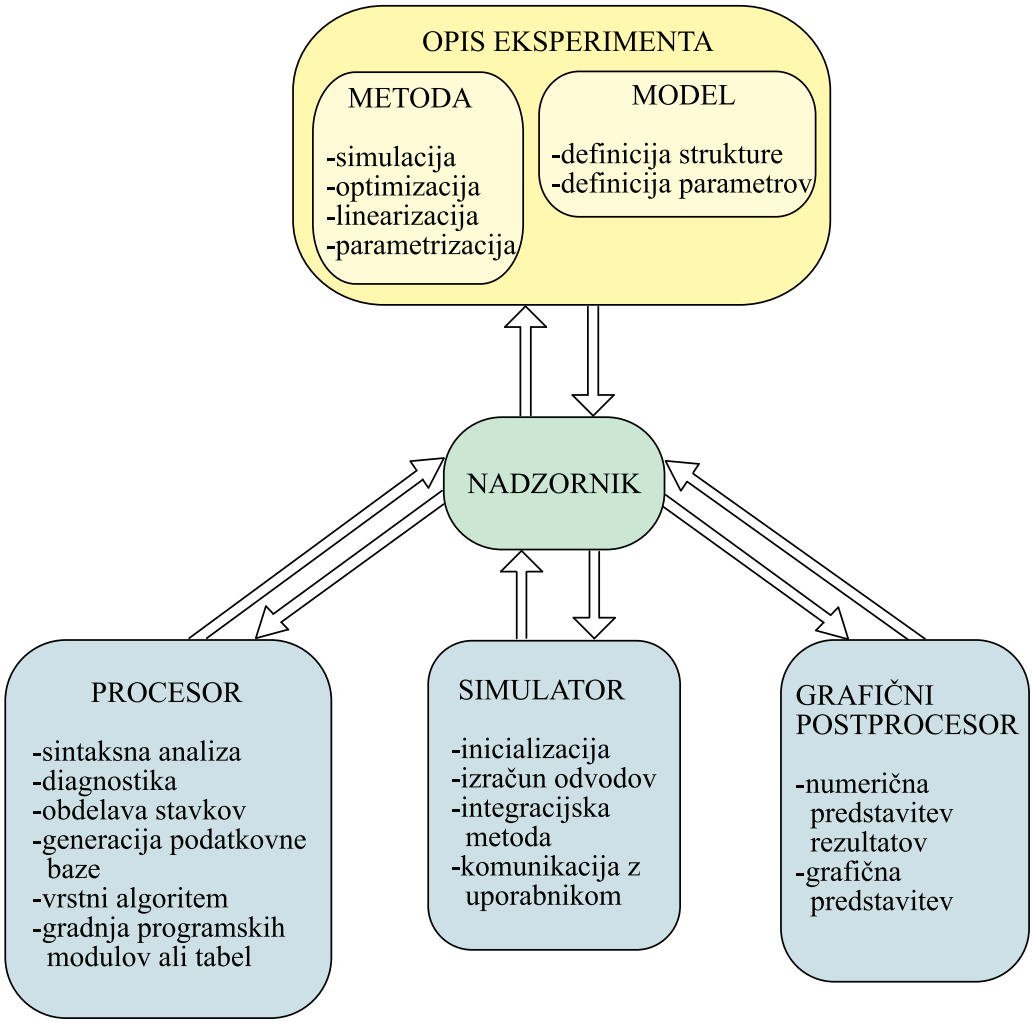
5.2.1 Opis eksperimenta

Do nedavnega so imeli simulacijski jeziki le en eksperiment, t.j. simulacijski tek. Opis eksperimenta je zahteval le opis modela in nekaterih parametrov za krmiljenje simulacijskega teka. Programski modul za opis eksperimenta (modela) je torej omogočal definirati

- strukturo modela,
- parametre modela,
- krmilne parametre simulacije,
- parametre oz. zahteve za prikaz rezultatov.

Dejstvo, da je bilo v istem programskem modulu potrebno opisati model in nekatere krmilne parametre, imamo za precejšnjo slabost tovrstnih sistemov.

Sodobna simulacijska orodja striktno ločijo med opisom modela in metode. Model in metodo je potrebno opisati v različnih programskih modulih (včasih v različnih



Slika 5.1: Programska zgradba zveznih simulacijskih jezikov

datotekah). Poleg elementarne metode - simulacijski tek ima uporabnik na voljo tudi optimizacijo, linearizacijo, parametrizacijo, analizo ustaljenega stanja, analizo občutljivosti itd.

Struktura modela

Glede na zmožnosti, kako definirati strukturo modela, se simulacijski jeziki zelo razlikujejo. Nekateri so bločno orientirani in vsebujejo le osnovne bloke na nivoju analognega računalnika. Moderni simulacijski jeziki pa imajo izredno razvite

zmožnosti za opis strukture. Možno je direktno vnašanje kompliciranih diferencialnih enačb, vgnezdenih izrazov itd. Vedno več orodij omogoča opis modela z grafično simulacijsko shemo.

Parametri modela

V večini simulacijskih jezikov razlikujemo med dvema vrstama parametrov:

- konstante modela,
- podatki, ki opisujejo generacijo funkcij (signalov).

Krmilni parametri simulacije

To so podatki za metodo simulacija. S temi parametri definiramo:

- Simulacijski (integracijski) algoritem.
- Komunikacijski interval, ki definira točke neodvisne spremenljivke, v kateri želi uporabnik dobiti rezultate simulacije.
- Računski korak, t.j. korak integracijskega postopka. Velikost koraka je odvisna od časovnih konstant sistema, ki ga simuliramo, od zahtev po relativni oz. absolutni natančnosti ter od izbranega integracijskega postopka.
- Začetno in končno vrednost neodvisne spremenljivke. Pogoj za končanje simulacijskega teka lahko pogosto opišemo tudi z aritmetičnim ali logičnim izrazom.
- Dopustno relativno oz. absolutno napako, če uporabljamo metodo s prilagodljivim računskim korakom.

Parametri oz. zahteve za prikaz rezultatov

S temi parametri uporabnik pove, katere spremenljivke bi rad spremljal, na katerem mediju in v kakšni obliki naj se prikazujejo. V splošnem lahko rezultate spremljamo med simulacijo ali pa jih prikažemo po simulaciji.

Najmodernejši simulacijski jeziki pa dopuščajo tudi bolj kompleksna eksperimentiranja. V takih jezikih lahko uporabnik izbere že vključene metode (npr. optimizacijo, linearizacijo, parametrizacijo, ...) ali pa sam programira svojo metodo. Za take jezike pravimo, da imajo popolno ločitev metode in modela, eksperiment pa je možno izvršiti z uporabo katerekoli metode nad katerimkoli modelom (hierarhični koncept model, metoda eksperiment - Breitenecker, Solar, 1986).

Klasični simulacijski jeziki, ki so komercialno še vedno najuspešnejši in v glavnem še spoštujejo standard CSSL'67 (npr. ACSL, CSSL IV) tudi omogočajo bolj kompleksna eksperimentiranja, vendar je ločitev modela in metode bolj navidezna. Eksperimenetiranje omogočajo t.i. sekcije INITIAL, DYNAMIC in TERMINAL. Z njimi uporabnik definira operacije, ki se izvajajo pred simulacijo, med simulacijo paralelno z modelom in po simulaciji.

5.2.2 Procesor

Procesor simulacijskega jezika sestoji iz enega ali več programskih modulov, ki prevedejo simulacijski model (oz. eksperiment) v programske module in ustrezno podatkovno bazo v primeru prevajalniških jezikov oz. v ustrezne tabele (datoteke) v primeru interpreterskih jezikov. Procesor torej zgradi najbolj vitalni del, ki ga potrebuje simulator.

Procesor najprej izvrši sintaksno analizo nad podanim modelom (simulacijskim programom). Nato se specifično obdelujejo različni stavki izvornega simulacijskega programa. Predvsem se specifično obdelajo stavki (bloki), ki definirajo stanje sistema (bloki, ki vsebujejo spomin, oz. zakasnitveni atribut). Realizirati je namreč potrebno prekinitev vseh zank na izhodih omenjenih blokov (podpoglavje 3.9). Obdelavi stavkov sledi razvrščanje stavkov (blokov), kar omogoča pravilno simulacijo fizikalno gledano paralelno delujočega sistema. Vrstni algoritem lahko razvrsti stavke le v primeru, če je v vsaki zanki vsaj en blok z zakasnitvenim atributom (običajno integrator), saj se v tem primeru že pri predhodni obdelavi stavkov vse zanke prekinejo. Zanke, v katerih ni blokov z zakasnitvenim atributom, jemlje vrstni algoritem za algebrajske zanke. Pri nekaterih simulacijskih jezikih se v tem primeru le sporoči ustrezna diagnostika, procesiranje pa se prekine. Nekateri jeziki pa imajo vgrajene posebne numerične postopke, ki omogočajo reševanje algebrajskih zank iterativno. Taki postopki pa izredno upočasnijo simulacijo.

Med obdelavo stavkov se zgradi tudi podatkovna baza modela. Nato se zgradijo

ustrezne tabele, ki omogočajo interpretersko simulacijo, v primeru prevajalniških jezikov pa se zgradi simulacijski program (iz enega ali več modulov), ki je lahko direktno v strojni obliki, ali pa v nekem splošnonamenskem programskem jeziku (npr. FORTRAN, PASCAL, C, ...). V slednjem primeru zahteva procesiranje tudi nadaljnje prevajanje in povezovanje (linkanje) v splošnonamenskem jeziku.

Dobri procesorji imajo učinkovito diagnostiko v vseh fazah procesiranja.

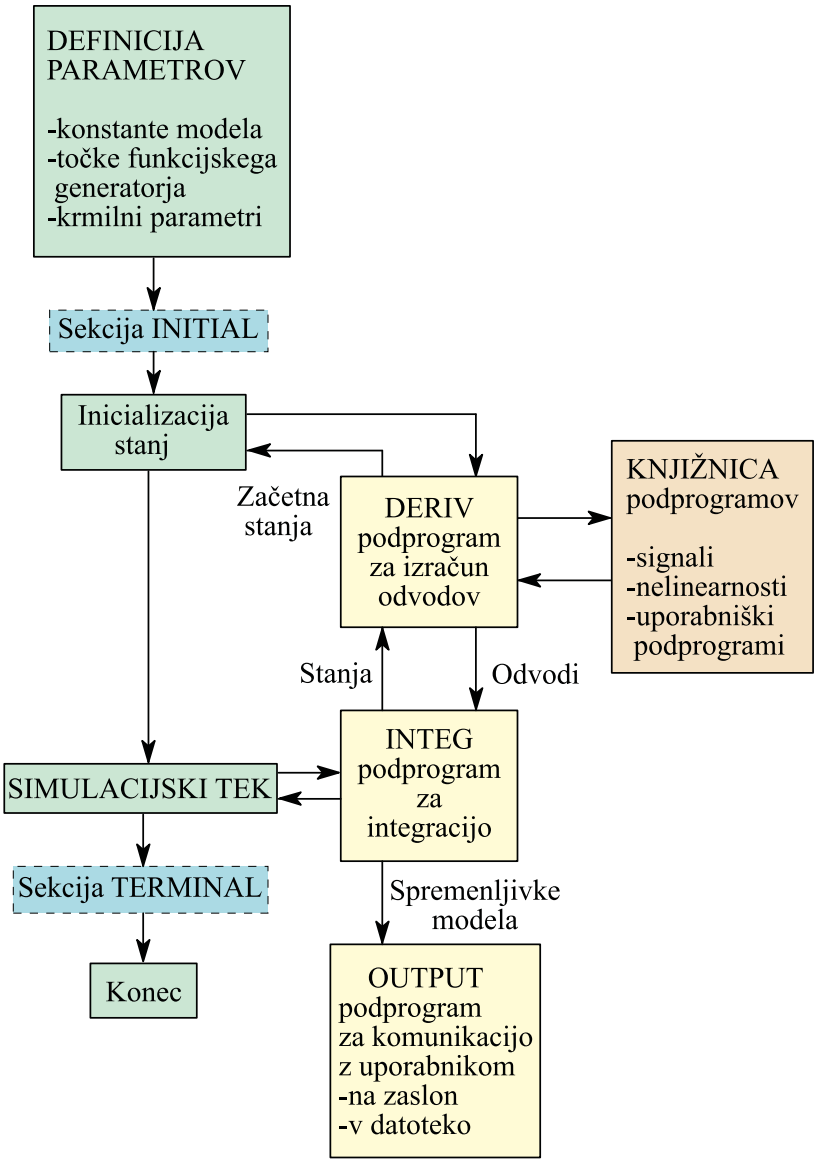
5.2.3 Simulator

Simulator ali sistem za izvajanje simulacije je fiksni program (v primeru interpreterskih jezikov) ali pa je program, ki ga zgradi simulacijski procesor (v primeru prevajalniških jezikov). Torej je simulator najbolj vitalni oz. osrednji del simulacijskega jezika. Za kvaliteto simulatorja je najpomembnejša natančnost, numerična stabilnost in računska učinkovitost (hitrost) simulacijskega (integracijskega) algoritma. Slika 5.2 prikazuje programsko zgradbo simulatorja.

Pred začetkom simulacijskega teka mora simulator dobiti določene parametre, t.j. konstante modela, podatke za funkcijske generatorje, krmilne parametre simulacije in zahteve za prikaz rezultatov. Ti podatki so shranjeni na eni ali več datotekah, ki jih je predhodno zgradil simulacijski procesor. Nekateri simulacijski jeziki vsebujejo sekcijo INITIAL, ki opisuje operacije pred simulacijskim tekom. V tem primeru se po čitanju podatkov torej izvede sekcija INITIAL, nato pa se s pomočjo klica podprograma DERIV (le-ta vsebuje enačbe modela) ovrednotijo začetne vrednosti stanj. Simulacijski tek pa se začne s klicem podprograma INTEG, ki med simulacijo zahteva številne klice podprograma za izračun odvodov DERIV (npr. en klic podprograma DERIV na računski korak pri Eulerjevi metodi, štirje klici pri metodi Runge-Kutta, ...). V t.i. komunikacijskih intervalih pa se kliče podprogram OUTPUT, ki skrbi za ustrezno posredovanje rezultatov simulacije uporabniku. Pri jezikih, ki vsebujejo sekcijo TERMINAL se po zaključku simulacije izvedejo še operacije, ki so opisane v tej sekciji.

5.2.4 Postprocesor za grafično predstavitev rezultatov

Uporabnik simulacijskega jezika lahko spremlja rezultate med simulacijskim tekom v grafični ali numerični obliki, bolj kompleksen pregled rezultatov pa se običajno izvede po simulaciji s pomočjo grafičnega postprocesorja. Le-ta je zlasti



Slika 5.2: Programska zgradba simulatorja

učinkovit, če omogoča vključitev rezultatov različnih simulacij oz. eksperimentov. Grafični predprocesor običajno omogoča:

- risanje več krivulj na zaslon,
- avtomatsko skaliranje,
- spreminjanje območij,

- povečevanje (zoom),
- odbiranje vrednosti iz krivulj (trace),
- različne vrste interpolacij med sosednjima točkama (ničti red, linearna interpolacija, točkovna predstavitev, ...),
- linearna ali logaritmična skala na oseh, z mrežo ali brez.

Vse funkcije morajo biti dosegljive na interaktivni in uporabniško prijazni način.

5.2.5 Nadzornik

Nadzornik omogoča povezavo vseh na sliki 5.1 prikazanih delov simulacijskega jezika. Najpomembnejše funkcije nadzornika so naslednje:

- Izbira simulacijskega modela. Pri prevajalniških jezikih lahko izberemo model v izvirni ali prevedeni obliki.
- Izbira metode za eksperiment (npr. simulacije, optimizacije, ...).
- Možnost za opis modela oz. metode ali za editiranje obstoječega modela oz. metode.
- Procesiranje modela (pri prevajalniških jezikih vključuje tudi prevajanje in povezovanje v vmesnem splošnonamenskem jeziku).
- Izvršitev simulacijskega teka ali bolj kompleksnega eksperimenta.
- Uporabniško prijazno editiranje konstant modela, podatkov za generacijo funkcij, krmilnih parametrov simulacije, zahtev za komunikacijo z uporabnikom ne da bi bilo potrebno editiranje izvornega programa in ponovno prevajanje v primeru prevajalniških jezikov.

Tako kot pri grafičnem postprocesorju morajo biti tudi tu vse funkcije dosegljive na interaktivni in uporabniško prijazni način.

5.2.6 Uporabniški vmesnik

Uporabniški vmesnik predstavljajo tisti programski moduli, ki omogočajo komunikacijo med simulacijskim jezikom in uporabnikom. Na sliki 5.1 sicer ni omenjen, vendar se nahaja v več delih opisane programske strukture, kot npr. pri opisu eksperimenta (npr. z grafično simulacijsko shemo), v nadzorniku in v grafičnem postprocesorju. Glavna odlika uporabniškega vmesnika je visoka interaktivnost in uporabniška prijaznost.

5.3 Enačbno orientirani simulacijski jeziki

Kot smo omenili v poglavju 4.1, so prevajalniški jeziki običajno enačbno orientirani, kar pomeni, da lahko uporabnik diferencialne enačbe direktno vključuje v izvorni program, ne da bi moral pred tem razviti simulacijsko shemo. Razvoj simulacijskega modela je tako enostavnejši in hitrejši, simulacijski programi pa so krajši, bolj modularni in bolj razumljivi in s tem zelo primerni za dokumentacijo modela.

Zaradi izjemne vloge standarda CSSL'67 na razvoj digitalnih simulacijskih jezikov, bomo kot prvi jezik predstavili jezik SIMCOS (Zupančič, 1992), ki je bil razvit pod vplivom tega standarda. Sicer sta v svetovnem merilu najuspešnejša tovrstna simulacijska jezika ACSL in CSSL IV . Izkušnje, ki jih pridobimo ob uporabi enega jezika tipa CSSL, omogočajo rabo kateregakoli drugega CSSL jezika brez večjih problemov.

5.3.1 Simulacijski jezik SIMCOS

Simulacijski jezik SIMCOS (Zupančič, 1989, Zupančič, 1992) smo razvili v Laboratoriju za modeliranje, simulacijo in vodenje na Fakulteti za elektrotehniko v Ljubljani in spada med prevajalniške jezike tipa CSSL. Izvorni program v sintaksi CSSL se prevaja v module v jeziku FORTRAN, ob tem pa se generira tudi potrebna podatkovna baza. Moduli v jeziku FORTRAN se nato prevajajo s prevajalnikom FORTRAN, tako dobljeni objektni moduli pa se nato povežejo (linkajo) s knjižnicami jezikov FORTRAN in SIMCOS v izvršljivi simulacijski program.

Opis modela

Simulacijski model je možno opisati s tekstovnim načinom ali pa s pomočjo grafičnega predprocesorja BLOCK v obliki simulacijske sheme (Zupančič, 1992).

Pri opisu modela s tekstovnim načinom moramo poznati sintakso jezika SIMCOS. Program, ki ga napišemo v datoteko, sestavljajo naslednji stavki:

- osnovni stavki,
- krmilni stavki,
- predstavitveni stavki in
- izhodni stavki.

Ker je jezik SIMCOS natančno opisan v priročniku (Zupančič, 1992), bomo na tem mestu podali le nekoliko posplošen in skrajšan pregled stavkov.

Osnovni stavki

Osnovni stavki omogočajo definicijo nekaterih osnovnih lastnosti simulacijskega modela (npr. deklaracija spremenljivk, definicije konstant,...). Prvi stavek vsakega programa je običajno stavek `PROGRAM`. Stavek določa ime modela. Zadnji stavek mora biti stavek `END`. Komentar uvedemo s stavkom `COMMENT` ali z znakom "v prvi koloni.

Stavek `ARRAY` določa imena in dimenzije indeksiranih spremenljivk in ima obliko

```
ARRAY sprem(dim[,dim][,dim]) [,sprem(dim[,dim][,dim])]...
```

`sprem`	...	ime indeksirane spremenljivke
`dim`	...	dimenzija indeksirane spremenljivke

Oglati oklepaji vključujejo izbirne parametre, spremenljivke ali izraze.

Stavek `CONSTANT` uporabljamo za definicijo konstant oz. za inicializacijo spremenljivk (navadnih in enodimenzionalnih indeksiranih). Stavek ima obliko

```
CONSTANT sprem = konst [,sprem = konst]...
```

`sprem`	...	ime spremenljivke
`konst`	...	vrednost(i) spremenljivke (vrednosti indeksirane spremenljivke so ločene z vejicami)

Stavek `TABLE` določa funkcijo ene ali dveh neodvisnih spremenljivk. Ima obliko

```
TABLE ime,n,dim,pod
```

`ime`	...	ime funkcijskega generatorja
`n`	...	število neodvisnih spremenljivk (celoštevilčna konstanta 1 ali 2)
`dim`	...	število lomnih točk neodvisnih spremenljivk (dim1, če je ena neodvisna spremenljivka oz. dim1, dim2, če sta dve neodvisni spremenljivki)
`pod`	...	vrednosti neodvisnih in odvisne spremenljivke v lomnih točkah (realne ali celoštevilčne konstante)

Stavek `TERMT` določa pogoj za končanje simulacijskega teka. Ima obliko

```
TERMT izraz
```

Ko vrednost logičnega izraza `izraz` postane `.TRUE.`, se simulacijski tek konča.

Krmilni stavki

Krmilne stavke uporabljamo za definicijo imen in vrednosti krmilnih spremenljivk. Z njimi podajamo zahteve v zvezi z neodvisno spremenljivko simulacije, zahteve za simulacijski algoritem ter izhodne zahteve. Običajno ni potrebno navesti vseh krmilnih stavkov. V takem primeru se uporabijo privzeta imena in vrednosti. Stavki imajo obliko

```
VARIABLE  sprem  =  konst
CINTERVAL sprem  =  konst
NSTEPS    sprem  =  konst
MERROR    sprem  =  konst
XERROR    sprem  =  konst
ALGORITHM sprem1 =  konst1, sprem2 = konst2
```

`sprem, sprem1, sprem2 ...`	poljubna imena spremenljivk
`konst, konst1, konst2 ...`	vrednosti spremenljivk (realna ali celoštevilčna)

Stavek `VARIABLE` določa ime neodvisne spremenljivke in njeno začetno vrednost. Stavek `CINTERVAL` določa velikost komunikacijskega intervala v enotah neodvisne spremenljivke (običajno čas). Stavek `NSTEPS` določa pri integracijski metodi s prilagodljivim računskim korakom začetno število računskih korakov znotraj komunikacijskega intervala, pri metodah s stalnim korakom pa število računskih korakov znotraj komunikacijskega intervala. Stavka `MERROR` in `XERROR` določata dopustno relativno in absolutno napako pri integracijskih postopkih s prilagodljivim računskim korakom. Stavek `ALGORITHM` določa uporabljeni simulacijski algoritem. Uporabnik lahko z vrednostjo `konst2` izbira med naslednjimi metodami: diskretna simulacija (`konst2=1`), Eulerjeva metoda (`konst2=3`), metoda Runge-Kutta-Gill s stalnim korakom (`konst2=1`), metoda Runge-Kutta-Gill s prilagodljivim korakom (`konst2=8`), metoda Runge-Kutta-Merson (`konst2=11`), Rosenbrock-ova polimplicitna metoda (`konst2=6`), ekstrapolacijska metoda z linearno implicitnim sredinskim pravilom (`konst2=7`), Gear-ova metoda za toge sisteme (`konst2=9`) in Adams-Moulton-ova prediktor korektor metoda (`konst2=10`). Prve tri metode uporabljajo stalni računski korak, preostale pa prilagodljiv računski korak, kar pomeni, da se med simulacijo ocenjuje dejanska napaka, glede na velikost predpisane napake pa se sproti avtomatsko prilagaja velikost računskega koraka. Izbira optimalnega integracijskega algoritma zahteva poglobljeno znanje o numerični problematiki integracijskih algoritmov. Na tem mestu naj povemo le, da so najbolj splošno uporabne metode Runge-Kutta, v primeru togih sistemov (zelo različne časovne konstante) pa je smiselno uporabiti Gearovo metodo. Nekateri od zgoraj omenjenih algoritmov omogočajo tudi simulacijo v realnem v času.

Predstavitveni stavki

Predstavitveni stavki (bloki) podajajo sistem, ki ga simuliramo. Vrstni red predstavitvenih stavkov ni važen, ker simuliramo sistem, ki deluje paralelno. SIMCOS prevajalnik z vgrajenim vrstnim algoritmom uredi stavke tako, da so vse vhodne spremenljivke definirane prej, preden se stavek izvrši.

Predstavitveni stavek določi vrednost ene ali več izhodnih spremenljivk kot rezultat določenih operacij na naboru vhodnih spremenljivk. Uporabljamo lahko simulacijsko orientirane operatorje (kot na primer integratorje in proceduralne bloke), običajne aritmetične prireditvene stavke (v njih lahko kličemo tudi vse sistemske in uporabniške funkcijske podprograme), stavke za realizacijo signalov, nelinearnosti in stavke za realizacijo zveznih in diskretnih dinamičnih podmodelov.

Simulacijsko orientirani stavki

`INTEG` operator predstavlja temeljni stavek simulacijskega jezika in realizira simulacijski algoritem (integriranje ali zakasnitev). Stavek ima obliko

```
sprem = INTEG(izraz1,izraz2)
```

`sprem` ... ime spremenljivke, ki predstavlja izhod iz operatorja (stanje)

`izraz1` ... vhod v operator (odvod)

`izraz2` ... začetni pogoj

Stavek `PROCEDURAL` uvaja skupino stavkov v jeziku FORTRAN, ki jih napiše uporabnik za realizacijo določene strukture (bloka) oz. za opis relacij med vhodnimi in izhodnimi spremenljivkami. Zapis ima obliko

```
PROCEDURAL (sprem1 = sprem2)
```

`sprem1` ... seznam izhodnih spremenljivk

`sprem2` ... seznam vhodnih spremenljivk

Vsak blok `PROCEDURAL` se mora začenjati s stavkom `PROCEDURAL` in končati s stavkom `END`. Vrstni algoritem jezika SIMCOS obravnava blok `PROCEDURAL` kot en predstavitveni stavek in ga uvrsti glede na seznama vhodnih in izhodnih spremenljivk. Vrstni red stavkov znotraj bloka ostane popolnoma nespremenjen.

Stavki za realizacijo signalov in nelinearnosti

Slika 5.3 prikazuje v jezik SIMCOS vgrajene signale in nelinearnosti.

Signali in nelinearnosti so kot funkcijski podprogrami vključeni v knjižnici jezika SIMCOS. Klicni stavki imajo naslednje oblike:

Stopnica
```
Y=STEP(X,P)
```

Linearno naraščajoči signal
```
Y=RAMP(X,P)
```

Vlak impulzov
```
Y=PULSE(X,TZ,P,W)
```

Harmonska funkcija
```
Y=HARM(X,TZ,W,P)
```

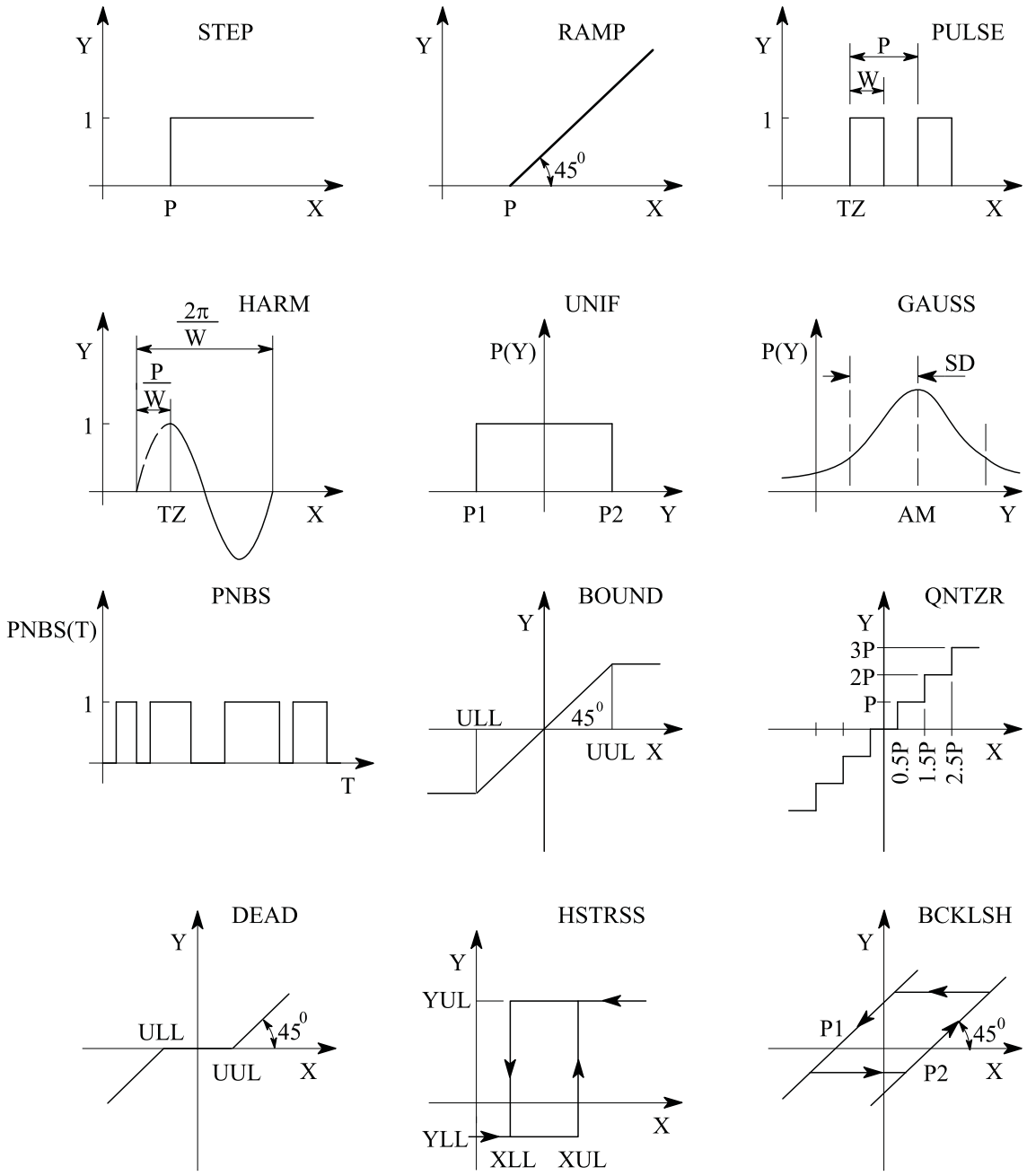
Uniformni šum
```
Y=UNIF(P1,P2,R0,R1,TS)
```

Gaussov šum
```
Y=GAUSS(AM,SD,R0,R1,TS)
```

Pseudonaključni binarni šum
```
Y=PNBS(R1,R2,R3,TS)
```
`R1`, `R2`, `R3` so začetni pogoji,
`TS` je čas vzorčenja

Omejevalnik
```
Y=BOUND(X,ULL,UUL)
```

Kvantizator



Slika 5.3: Signali in nelinearnosti

```
Y=QNTZR(X,P)
```

Mrtva cona

```
Y=DEAD(X,ULL,UUL)
```

Histereza

```
Y=HSTRSS(X,XLL,XUL,YLL,YUL,STANJE)
```

Mrtvi hod

```
Y=BCKLSH(X,P1,P2,STANJE)
```

Komparator

```
Y=COMPAR(X1,X2)
```

$$\begin{aligned} \texttt{Y} &= \texttt{0.} \quad \textit{če} \quad \texttt{X1} < \texttt{X2} \\ \texttt{Y} &= \texttt{1.} \quad \textit{če} \quad \texttt{X1} \geq \texttt{X2} \end{aligned}$$

Funkcijsko stikalo

```
Y=FCNSW(X,P1,P2,P3)
```

$$\begin{aligned} \texttt{Y} &= \texttt{P1} \quad \textit{če} \quad \texttt{X} < \texttt{0.} \\ \texttt{Y} &= \texttt{P2} \quad \textit{če} \quad \texttt{X} = \texttt{0.} \\ \texttt{Y} &= \texttt{P3} \quad \textit{če} \quad \texttt{X} > \texttt{0.} \end{aligned}$$

Vhodno stikalo

```
Y=SWIN(X,P1,P2)
```

$$\begin{aligned} \texttt{Y} &= \texttt{P1} \quad \textit{če} \quad \texttt{X} < \texttt{0.} \\ \texttt{Y} &= \texttt{P2} \quad \textit{če} \quad \texttt{X} \geq \texttt{0.} \end{aligned}$$

Pri vseh funkcijah velja, da je `X` vhodni signal, `Y` izhodni signal, vloga večine parametrov pa je razvidna iz slike 5.3.

Dinamični podmodeli

Dinamični podmodeli so vnaprej prevedeni moduli, uporabnik pa jih kliče kot funkcije (podobno kot pri signalih, nelinearnostih). V vsaki zanki simulacijske sheme se mora nahajati vsaj en podmodel z zakasnitvenim atributom (trenutna vrednost vhoda ne vpliva na trenutno vrednost izhoda). Podmodeli, ki opisujejo splošne dinamične strukture (npr. prenosna funkcija, zapis v prostoru stanj), imajo zato dva primerka: brez zakasnitve in z zakasnitvijo. Matrike je potebno podajati kot enodimenzionalna polja tako, da jih prepisujemo po vrsticah.

Zvezni dinamični podmodeli

Integrator ($I0$)

```
Y=FIN(U,YO)
```

$G(s) = \frac{Y(s)}{U(s)} = \frac{1}{s}$ $\quad y(0) = y_0$

Vektorski integrator

```
Y=VIN(U,N,YO)
```

$N \cdots$število integratorjev

Krmiljeni integrator ($I0$)

```
Y=FKIN(U,YO,IC,OP)
```

$G(s) = \frac{Y(s)}{U(s)} = \frac{1}{s}$ $\quad y(0) = y_0$

IC	OP	Stanje integratorja
0	0	drži (HD)
0	1	integrira (OP)
1	0	začetni pogoji (IC)
1	1	začetni pogoji (IC)

Sistem 1.reda ($P1$)

```
Y=FLAG(U,K,TAU)
```

$G(s) = \frac{Y(s)}{U(s)} = \frac{K}{\tau s+1}$

Integrirni sistem ($I1$)

```
Y=FINTLG(U,K,TAU)
```

$G(s) = \frac{Y(s)}{U(s)} = \frac{K}{s(\tau s+1)}$

Diferencirni sistem ($D1$)

```
Y=DIFF(U,KD,TF)
```

$G(s) = \frac{Y(s)}{U(s)} = \frac{K_D s}{T_f s+1}$

Zakasnilno - prehitevalni člen

```
Y=FLEDLG(U,K,Z,P)
```

$G(s) = \frac{Y(s)}{U(s)} = K\frac{s-z}{s-p}$

Sistem 2.reda ($P2$)

```
Y=SECOR(U,ZETA,OMEGAN)
```

$G(s) = \frac{Y(s)}{U(s)} = \frac{\omega_n^2}{s^2+2\zeta\omega_n s+\omega_n^2}$

Sistem z mrtvim časom

```
Y=DELAY(U,TD,STATES,TS)
```

$G(s) = \frac{Y(s)}{U(s)} = e^{-T_d s}$
$T_s \cdots$ perioda vzorčenja

Prenosna funkcija v polinomski obliki (brez zakasnitve)

```
Y=CTF(U,N,B,A)
```

$G(s) = \frac{Y(s)}{U(s)} = \frac{b_0 s^n+b_1 s^{n-1}+\cdots+b_{n-1}s+b_n}{s^n+a_1 s^{n-1}+\cdots+a_{n-1}s+a_n} = \frac{B(1)s^N+B(2)s^{N-1}+\cdots+B(N)s+B(N+1)}{s^N+A(1)s^{N-1}+\cdots+A(N-1)s+A(N)}$

Prenosna funkcija v polinomski obliki (z zakasnitvijo)

```
Y=CTFD(U,M,N,B,A)
```

$G(s) = \frac{Y(s)}{U(s)} = \frac{b_0 s^m + b_1 s^{m-1} + \cdots + b_{m-1} s + b_m}{s^n + a_1 s^{n-1} + \cdots + a_{n-1} s + a_n} = \frac{B(1)s^M + B(2)s^{M-1} + \cdots + B(M)s + B(M+1)}{s^N + A(1)s^{N-1} + \cdots + A(N-1)s + A(N)}$
$m < n$

Prenosna funkcija v faktorizirani obliki (brez zakasnitve)

```
Y=CZP(U,N,K,Z,P)
```

$G(s) = \frac{Y(s)}{U(s)} = K\frac{(s-z_1)\cdots(s-z_n)}{(s-p_1)\cdots(s-p_n)} = K\frac{(s-Z(1))\cdots(s-Z(N))}{(s-P(1))\cdots(s-P(N))}$

Prenosna funkcija v faktorizirani obliki (z zakasnitvijo)

```
Y=CZPD(U,M,N,K,Z,P)
```

$G(s) = \frac{Y(s)}{U(s)} = K\frac{(s-z_1)\cdots(s-z_m)}{(s-p_1)\cdots(s-p_n)} = K\frac{(s-Z(1))\cdots(s-Z(M))}{(s-P(1))\cdots(s-P(N))}$
$m < n$

Sistem v prostoru stanj (brez zakasnitve)

```
Y=CSS(U,N,A,B,C,D,X)
```

$\dot{\boldsymbol{x}} = \boldsymbol{A}\boldsymbol{x} + \boldsymbol{B}u$
$y = \boldsymbol{C}\boldsymbol{x} + \boldsymbol{D}u$
$n \cdots$red sistema

Sistem v prostoru stanj (z zakasnitvijo)

```
Y=CSSD(U,N,A,B,C,X)
```

$\dot{\boldsymbol{x}} = \boldsymbol{A}\boldsymbol{x} + \boldsymbol{B}u$
$y = \boldsymbol{C}\boldsymbol{x}$
$n \cdots$red sistema

Multivariabilni sistem v prostoru stanj (brez zakasnitve)

```
Y=MCSS(U,N,M,L,A,B,C,D,X)
```

$\dot{\boldsymbol{x}} = \boldsymbol{A}\boldsymbol{x} + \boldsymbol{B}\boldsymbol{u}$
$\boldsymbol{y} = \boldsymbol{C}\boldsymbol{x} + \boldsymbol{D}\boldsymbol{u}$

n	$\cdots$	red sistema
m	$\cdots$	število vhodov
l	$\cdots$	število izhodov

Multivariabilni sistem v prostoru stanj (z zakasnitvijo)

```
Y=MCSSD(U,N,M,L,A,B,C,X)
```

$\dot{\boldsymbol{x}} = \boldsymbol{A}\boldsymbol{x} + \boldsymbol{B}\boldsymbol{u}$
$\boldsymbol{y} = \boldsymbol{C}\boldsymbol{x}$

n	$\cdots$	red sistema
m	$\cdots$	število vhodov
l	$\cdots$	število izhodov

PI regulator

```
U=PI(E,KP,KI)
```

$G(s) = \frac{U(s)}{E(s)} = K_P + \frac{K_I}{s}$

PID regulator

```
U=PID(E,KP,KI,KD,TF)
```

$G(s) = \frac{U(s)}{E(s)} = K_P + \frac{K_I}{s} + \frac{K_D s}{T_f s+1}$

Industrijski PID regulator

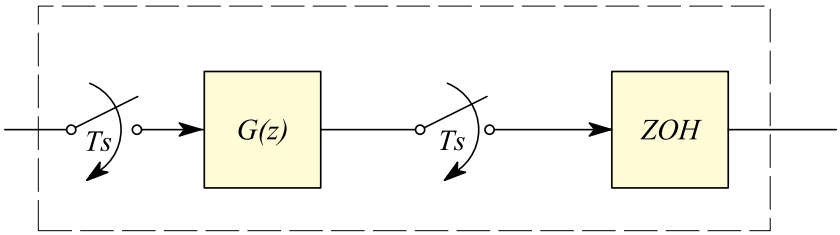
```
U=PIDAB(Y,R,UOO,MA,KP,TI,TD,TF,KA,GAMA,UMIN,UMAX)
```

$E = R - Y$
$G(s) = \frac{U(s)}{E(s)} = K_P(1 + \frac{1}{T_i s} + \frac{T_D s}{T_f s+1})$

U_{00}	$\cdots$	signal za vodenje v režimu ROČNO
`MA`	$\cdots$	preklop ROČNO/AVTOMATSKO (1/0)
K_a	$\cdots$	konstanta zaščite pred integralskim pobegom (priporočljivo $k_a = k_p$)
γ	$\cdots$	faktor, ki določa način uporabe D-člena $\gamma = 0 \cdots$na D-člen je pripeljan pogrešek $\gamma = 1 \cdots$na D-člen je pripeljana regulirana veličina
U_{min}, U_{max}	$\cdots$	minimalna in maksimalna vrednost regulirne veličine

Diskretni dinamični podmodeli

Vsak diskretni podmodel ima na vhodu vzorčevalnik, na izhodu pa zadrževalnik ničtega reda (zero - order hold ZOH ali D/A pretvornik). T_s je perioda vzorčenja. Izvedbo v primeru, če ga opisuje prenosna funkcija, prikazuje slika 5.4.



Slika 5.4: Izvedba diskretnega podmodela

Diskretna zakasnitev

```
Y=DDLY(U,D,STATES,TS)
```

$G(z) = \frac{Y(z)}{U(z)} = z^{-d}$

d	$\cdots$	vrednost diskretne zakasnitve
`STATES`	$\cdots$	začetne vrednosti zakasnitev

Prenosna funkcija v polinomski obliki (brez zakasnitve)

```
Y=DTF(U,N,B,A,TS)
```

$G(z) = \frac{Y(z)}{U(z)} = \frac{b_0+b_1z^{-1}+\cdots+b_{n-1}z^{-(n-1)}+b_nz^{-n}}{1+a_1z^{-1}+\cdots+a_{n-1}z^{-(n-1)}+a_nz^{-n}} = \frac{B(1)+B(2)z^{-1}+\cdots+B(N)z^{-(N-1)}+B(N+1)z^{-N}}{1+A(1)z^{-1}+\cdots+A(N-1)z^{-(N-1)}+A(N)z^{-N}}$

Prenosna funkcija v polinomski obliki (z zakasnitvijo)

```
Y=DTFD(U,M,N,B,A,TS)
```

$G(z) = \frac{Y(z)}{U(z)} = \frac{b_1 z^{-1} + \cdots + b_{m-1} z^{-(m-1)} + b_m z^{-m}}{1 + a_1 z^{-1} + \cdots + a_{n-1} z^{-(n-1)} + a_n z^{-n}} = \frac{B(1)z^{-1} + B(2)z^{-2} + \cdots + B(M-1)z^{-(M-1)} + B(M)z^{-M}}{1 + A(1)z^{-1} + \cdots + A(N-1)z^{-(N-1)} + A(N)z^{-N}}$

Prenosna funkcija v faktorizirani obliki (brez zakasnitve)

```
Y=DZP(U,M,N,K,Z,P,TS)
```

$G(z) = \frac{Y(z)}{U(z)} = K\frac{(1-z_1 z^{-1})\cdots(1-z_m z^{-1})}{(1-p_1 z^{-1})\cdots(1-p_N z^{-1})} = K\frac{(1-Z(1)z^{-1})\cdots(1-z(M)z^{-1})}{(1-P(1)z^{-1})\cdots(1-P(N)z^{-1})}$

Prenosna funkcija v faktorizirani obliki (z zakasnitvijo)

```
Y=DZPD(U,D,M,N,K,Z,P,TS)
```

$G(z) = \frac{Y(z)}{U(z)} = K\frac{(1-z_1 z^{-1})\cdots(1-z_m z^{-1})}{(1-p_1 z^{-1})\cdots(1-p_N z^{-1})} z^{-d} = K\frac{(1-Z(1)z^{-1})\cdots(1-z(M)z^{-1})}{(1-P(1)z^{-1})\cdots(1-P(N)z^{-1})} z^{-d}$
$d > 0$

Sistem v prostoru stanj (brez zakasnitve)

```
Y=DSS(U,N,A,B,C,D,X,TS)
```

$\boldsymbol{x}(k+1) = \boldsymbol{A}\boldsymbol{x}(k) + \boldsymbol{B}u(k)$
$y(k) = \boldsymbol{C}\boldsymbol{x}(k) + \boldsymbol{D}u(k)$
$n \cdots$red sistema

Sistem v prostoru stanj (z zakasnitvijo)

```
Y=DSSD(U,N,A,B,C,X,TS)
```

$\boldsymbol{x}(k+1) = \boldsymbol{A}\boldsymbol{x}(k) + \boldsymbol{B}u(k)$
$y(k) = \boldsymbol{C}\boldsymbol{x}(k)$
$n \cdots$red sistema

Multivariabilni sistem v prostoru stanj (brez zakasnitve)

```
Y=MDSS(U,N,M,L,A,B,C,D,X,TS)
```

$\boldsymbol{x}(k+1) = \boldsymbol{A}\boldsymbol{x}(k) + \boldsymbol{B}\boldsymbol{u}(k)$
$\boldsymbol{y}(k) = \boldsymbol{C}\boldsymbol{x}(k) + \boldsymbol{D}\boldsymbol{u}(k)$
n $\cdots$ red sistema
m $\cdots$ število vhodov
l $\cdots$ število izhodov

Multivariabilni sistem v prostoru stanj (z zakasnitvijo)

```
Y=MDSSD(U,N,M,L,A,B,C,X,TS)
```

$\boldsymbol{x}(k+1) = \boldsymbol{A}\boldsymbol{x}(k) + \boldsymbol{B}\boldsymbol{u}(k)$
$\boldsymbol{y}(k) = \boldsymbol{C}\boldsymbol{x}(k)$
n $\cdots$ red sistema
m $\cdots$ število vhodov
l $\cdots$ število izhodov

Sistem vzorči, drži (sample&hold)

```
Y=SH(U,STATE,TS)
```

PID regulator

```
U=DPID(E,Q,STATES,TS)
```

$G(z) = \frac{Y(z)}{U(z)} = \frac{q_0+q_1z^{-1}+q_2z^{-2}}{1-z^{-1}}$

Industrijski PID regulator

```
U=DPIDAB(Y,R,UOO,MA,KP,TI,TD,TF,KA,GAMA,UMIN,UMAX,TS)
```

$E = R - Y$
$G(z) = \frac{U(z)}{E(z)} = K_P[1 + (\frac{1}{T_i s})_{s=\frac{z-1}{T_s}} + (\frac{T_D s}{T_f s+1})_{s=\frac{2}{T_s}\frac{z-1}{z+1}}]$

U_{00}	$\cdots$	signal za vodenje v režimu ROČNO
`MA`	$\cdots$	preklop ROČNO/AVTOMATSKO (1/0)
K_a	$\cdots$	konstanta zaščite pred integralskim pobegom (priporočljivo $k_a = k_p$)
γ	$\cdots$	faktor, ki določa način uporabe D-člena $\gamma = 0 \cdots$na D-člen je pripeljan pogrešek $\gamma = 1 \cdots$na D-člen je pripeljana regulirana veličina
U_{min}, U_{max}	$\cdots$	minimalna in maksimalna vrednost regulirne veličine

Izhodni stavki

Izhodni stavki omogočajo prikaz rezultatov na zaslon in shranjevanje v datoteko.

Stavek `OUTPUT` povzroči izpis spremenljivk med simulacijskim tekom na vsakih n komunikacijskih intervalov na zaslon. Ima obliko

```
OUTPUT [n,] sprem [,sprem,]...
```

Stavek `PREPAR` povzroči vpis spremenljivk med simulacijskim tekom na vsakih n komunikacijskih intervalov v datoteko. Ima obliko

```
PREPAR [n,] sprem [,sprem,]...
```

Zmožnosti eksperimentiranja

Simulacijski jezik SIMCOS omogoča programiranje kompleksnih eksperimentov s simulacijskim modelom. Običajno so v take eksperimente vključeni iterativni simulacijski teki (analiza odvisnosti od parametrov, optimizacija ipd.). V ta namen mora uporabnik definirati razen modela v sintaksi jezika SIMCOS še tri sekcije (datoteke):

INITIAL	(datoteka model.INI)
DYNAMIC	(datoteka model.DYN)
TERMINAL	(datoteka model.TER)

INITIAL

V tej sekciji uporabnik s stavki v jeziku FORTRAN opiše operacije, ki naj se izvršijo pred simulacijskim tekom.

TERMINAL

V tej sekciji uporabnik s stavki v jeziku FORTRAN opiše operacije, ki naj se izvršijo po simulacijskem teku.

DYNAMIC

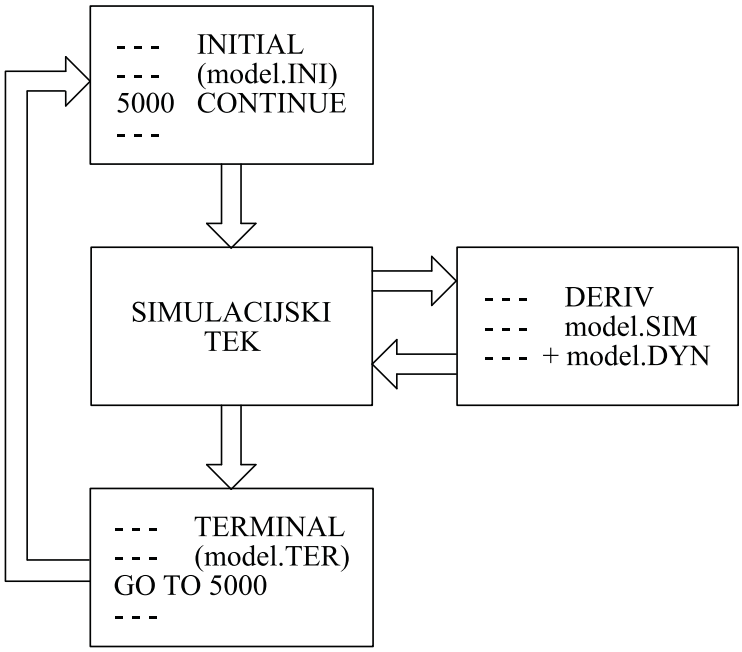
Sekcijo (datoteka model.DYN) se opiše s programom v jeziku SIMCOS. Uporablja se za opis tistih funkcij eksperimenta, ki se morajo izvajati med simulacijo (paralelno z modelom). V tej sekciji navadno definiramo vhodne signale in cenilko za optimizacijo. Zaradi sistematičnosti lahko vključimo tudi stavke za določitev konstant eksperimenta, stavke za krmiljenje simulacije in stavke za izhodne operacije. Tako lahko v izvornem simulacijskem programu (model.SIM) ostanejo le enačbe modela.

Način eksperimentiranja v simulacijskem jeziku SIMCOS prikazuje slika 5.5. S pomočjo take strukture lahko dosežemo ponavljanje simulacijskih tekov z ustreznimi skočnimi stavki v sekcijah INITIAL oziroma TERMINAL.

Za učinkovito eksperimentiranje s simulacijskim jezikom SIMCOS so vgrajeni nekateri eksperimenti s pomočjo vnaprej pripravljenih eksperimentalnih sekcij. Ti eksperimenti so optimizacija, parametrizacija in linearizacija.

Optimizacija

Metoda omogoča določiti optimalne vrednosti parametrov glede na predpisano cenilko, ki se izračuna s pomočjo simulacijskega teka. Optimizacija se izvaja v iteracijah, vsaka iteracija pa sestoji iz več simulacijskih tekov. Število tekov v eni iteraciji je odvisno od uspešnosti poizkusov in od števila optimiranih parametrov.



Slika 5.5: Koncept eksperimentiranja v jeziku SIMCOS

Rezultat vsake iteracije so nove vrednosti optimiranih parametrov, ki običajno predstavljajo izboljšano vrednost cenilke.

Pred optimizacijo mora uporabnik izbrati parametre (konstante), katerih optimalne vrednosti želi poiskati, definirati mora cenilko in omejitve. Razen tega mora podati še začetne iskalne korake, ter pogoje za končanje optimizacije.

Parametrizacija

Eksperiment parametrizacija omogoča avtomatsko ponavljanje simulacijskih tekov ob spreminjajoči se vrednosti ene izmed konstant simulacijskega modela. Ime ustrezne konstante, njeno začetno in končno vrednost ter korak spreminjanja podamo v nadzornem programu.

Linearizacija

S pomočjo linearizacije lahko uporabnik dobi linearizirani zapis modela v prostoru stanj (matrike $\boldsymbol{A}$, $\boldsymbol{B}$, $\boldsymbol{C}$ in $\boldsymbol{D}$) v končni točki simulacije. V nadzornem programu je potrebno izbrati vhode in izhode modela.

Interaktivne zmožnosti

Nadzorni program skupaj z uporabniškim vmesnikom omogoča avtomatsko procesiranje od izvornega do izvršljivega programa ter uporabniško prijazno opisovanje in editiranje izvornega modela, prevedenega modela ali eksperimenta (npr. editiranje konstant modela, editiranje točk funkcijskih generatorjev, editiranje izhodnih zahtev, editiranje krmilnih parametrov simulacije, editiranje parametrov eksperimenta,...). Delo je menujsko orientirano in zato primerno tudi za manj izkušene uporabnike.

5.4 Simulacijsko okolje Matlab-Simulink

Simulink je orodje, ki ga lahko uporabljamo v okviru programskega paketa Matlab in omogoča simulacijo dinamičnih sistemov. Nastal je kot dopolnilo Matlab-a, saj je za uporabnika zelo priročno, če lahko v enem programskem okolju izvaja različne operacije in pri tem ni potreben prenos podatkov iz oz. v druge programe (filtriranje signalov, opazovanje meritev in primerjava meritev z odzivi modela, načrtovanje vodenja, simulacijsko opazovanje delovanja načrtanih regulacijskih sistemov, itd.) in tako ponuja določene prednosti pri simulaciji, hkrati pa ohranja funkcionalnost Matlaba.

5.4.1 Osnovna uporaba Simulinka

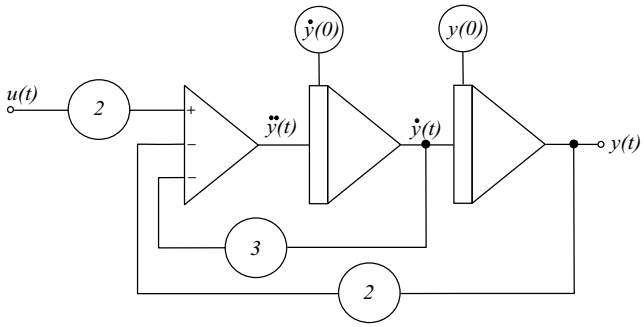
Opišimo uporabo osnovnih funkcij orodja Simulink na primeru reševanja modela drugega reda naslednje oblike:

$$\ddot{y}(t) + 3\dot{y}(t) + 2y(t) = 2u(t) \tag{5.1}$$

Enačbo (5.1) preoblikujmo za realizacijo simulacijske sheme po indirektni metodi:

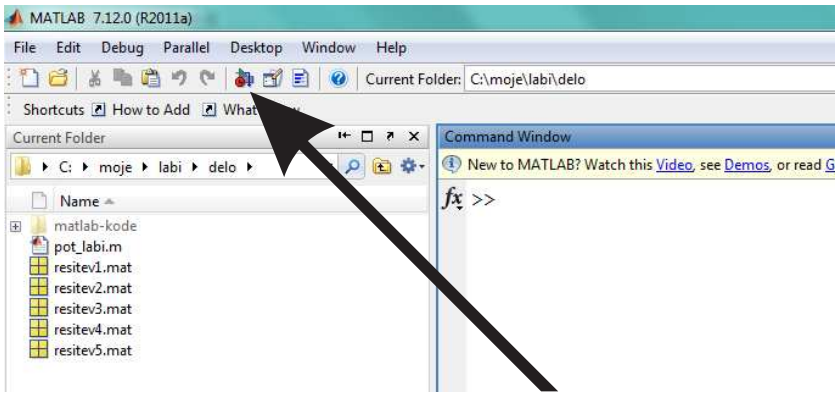
$$\ddot{y}(t) = -3\dot{y}(t) - 2y(t) + 2u(t) \tag{5.2}$$

in narišimo pripadajočo splošno simulacijsko shemo, kot jo prikazuje slika 5.6.



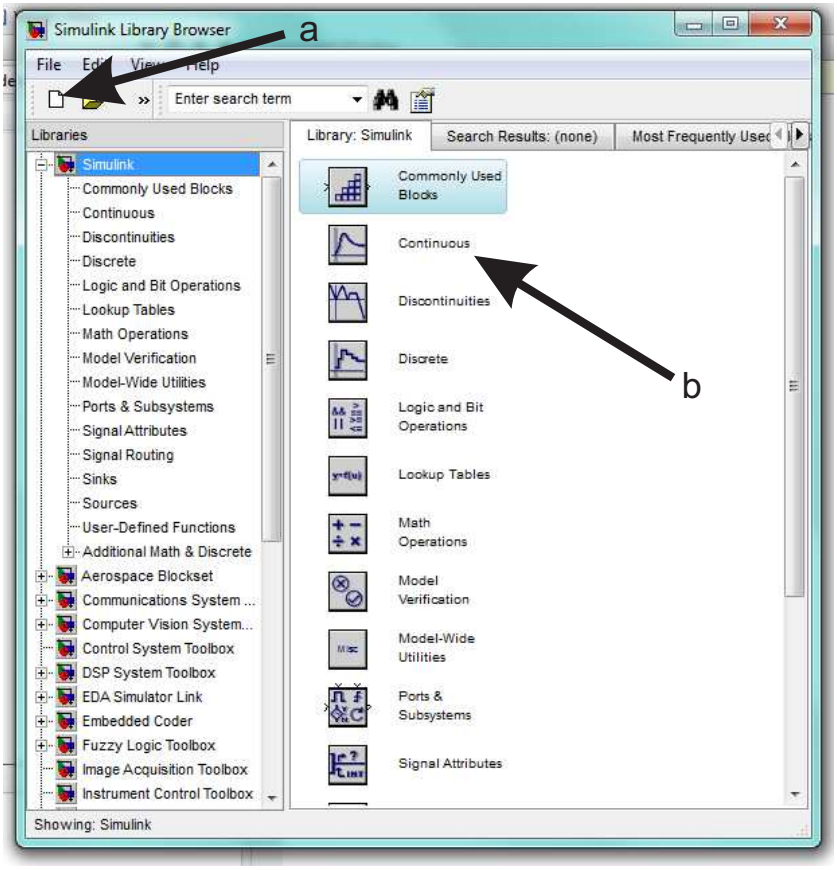
Slika 5.6: Simulacijska shema sistema 2. reda

Sedaj se lotimo izvedbe simulacijske sheme v Simulinku. Orodje zaženemo tako, da v komandnem oknu odtipkamo ukaz **simulink** ali pa kliknemo na ikono v orodni vrstici, kot prikazuje slika 5.7.



Slika 5.7: Zagon Simulinka iz orodne vrstice

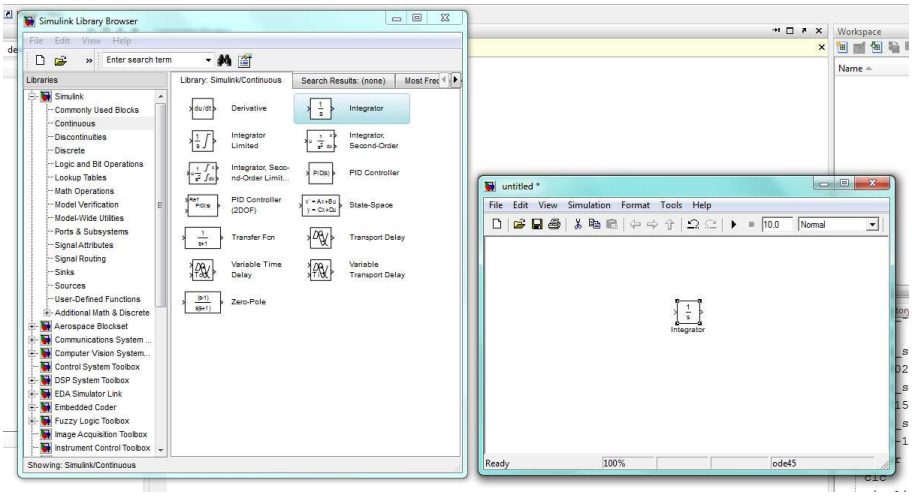
Ta ukaz odpre brskalnik Simulinkove knjižnice, kot prikazuje slika 5.8. Najprej s klikom na ikono iz orodne vrstice, kot kaže puščica na sliki 5.8, odpremo novo okno (nov model). Ko ta model shranimo, dobimo datoteko *.mdl.



Slika 5.8: Brskalnik Simulinkove knjižnice

Simulinkova knjižnica vsebuje veliko število blokov, ki jih lahko uporabljamo za gradnjo simulacijskih modelov. Zaradi preglednosti so urejeni po skupinah (podknjižnicah). Oglejmo si najprej skupino zveznih modelov. To odpremo z dvojnim klikom na oznako **Continuous**, kot kaže puščica b na sliki 5.8. Pri tem se odpre okno, kot je prikazano na levi strani slike 5.9.

Med bloki najdemo tudi takega, z oznako integrator, ki ga prenesemo v odprti simulacijski model. To storimo z ukazoma **ctrl**/**c** in **ctrl**/**v**, ali pa tako, da kliknemo na blok, držimo levi gumb na miški in blok vlečemo na ustrezno mesto v okno modela. Ker potrebujemo dva integratorja, bi lahko postopek ponovili. Lahko pa naredimo kopijo bloka kar znotraj modela z ukazoma **ctrl**/**c** in **ctrl**/**v**, ali pa tako, da kliknemo integrator z desnim gumbom in vlečemo na mesto, kjer želimo imeti kopijo.



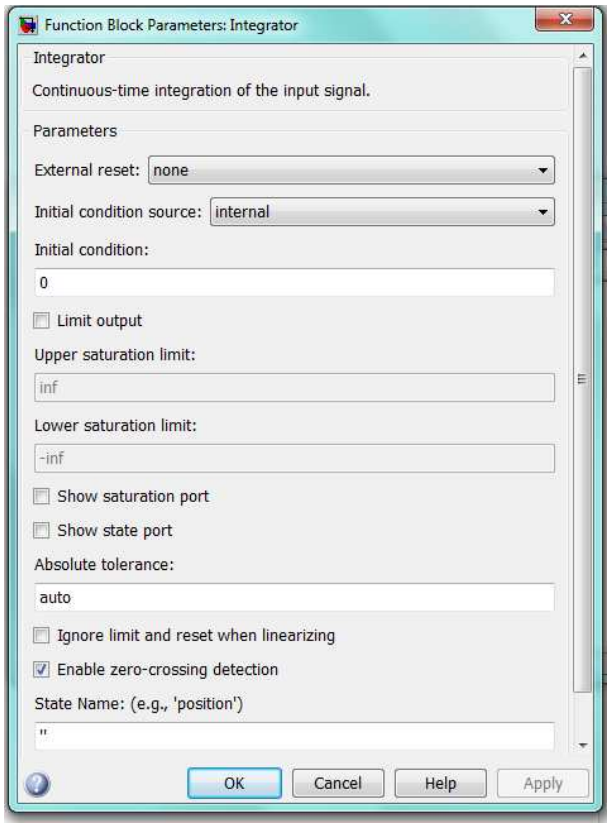
Slika 5.9: Skupina zveznih modelov (Continuous) Simulinkove knjižnice

Blokom, ki smo jih prenesli v model, nastavljamo lastnosti oz. parametre. V ta namen dvakrat kliknemo na prvi integrator in opazujmo vsebino okna, ki se odpre (slika 5.10).

Vidimo, da je pri integratorju mogoče nastaviti zunanji reset, izvor začetnega stanja, (v našem primeru izberimo notranji) in vrednost začetnega stanja (Initial condition). Predlagana začetna vrednost je nič in v našem primeru na obeh integratorjih ohranimo to vrednost. Lahko tudi omejimo izhod na določeno želeno območje in nastavimo t.i. absolutno toleranco ter omogočimo detekcijo prehoda ničle. Slednji dve lastnosti nastavljamo le pri zahtevnejših simulacijah. V našem primeru pustimo vse privzete vrednosti in to potrdimo s klikom na gumb OK v spodnjem delu okna.

Sedaj potrebujemo v simulacijski shemi še sumator, ki ga najdemo v skupini blokov z oznako **Math Operations**. Na enak način kot integrator, sedaj v simulacijsko shemo prenesemo blok z oznako **Add**, to je sumator. V tej skupini blokov se nahaja še ojačevalni blok **Gain**. Ko prenesemo v model tudi ojačevalni blok, dobimo prikaz na zaslonu na sliki 5.11).

Kliknimo dvakrat na sumator, da se odpre pripadajoče komunikacijsko okno, kot je prikazano na sliki 5.12 in v vrstico, ki nosi oznako *List of signs*, natipkajmo en plus (+) in dva minusa (-) ter potrdimo s pritiskom na gumb OK. To povzroči, da se na ikoni sumatorja v naši datoteki pojavijo trije vhodi, ki so ustrezno predz-

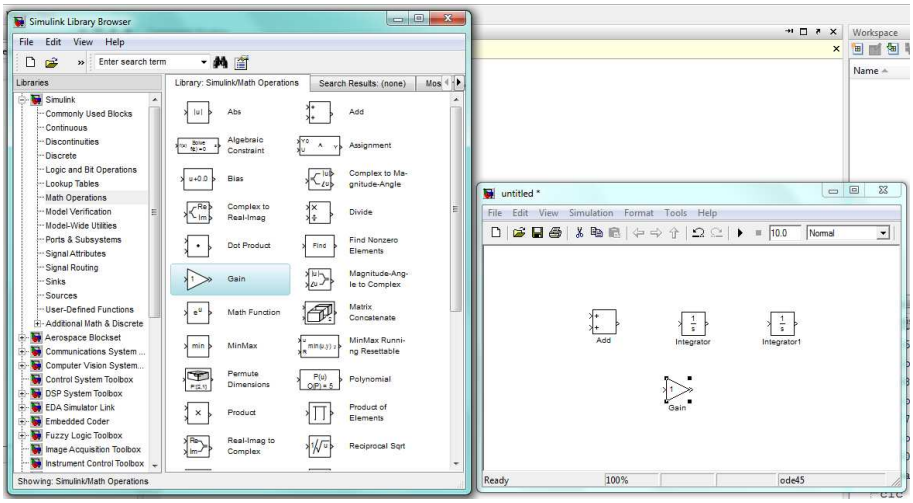


Slika 5.10: Komunikacijsko okno integratorja

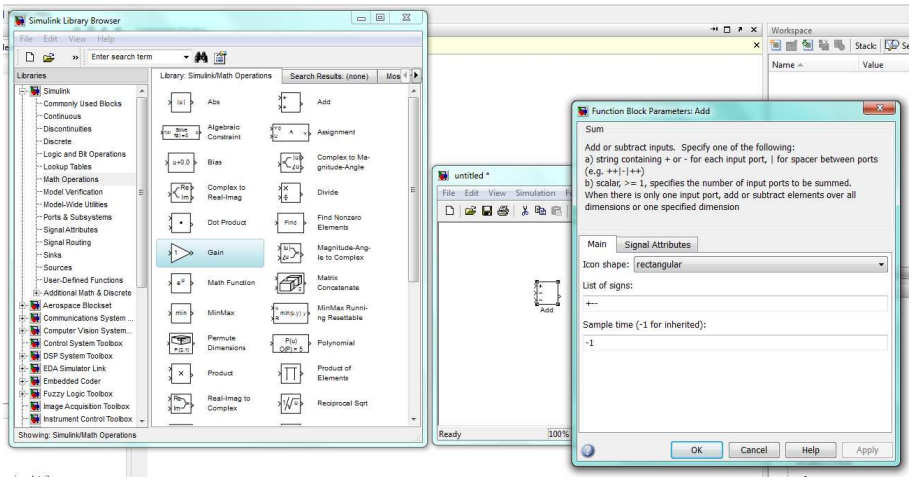
načeni.

Glede na razmere na sliki 5.6 bomo potrebovali tri ojačevalne bloke. Zato ojačevalni blok, ki ga že imamo v shemi, še dvakrat skopirajmo. S klikom na vsakega od ojačevalnih blokov odprimo pripadajoče komunikacijsko okno ter v vrstici z oznako *Gain* nastavimo pri dveh ojačevalnikih vradnost 2, pri enem pa 3.

Zaradi preglednosti včasih priročno, da katerega od elementov v simulacijski shemi zasukamo ali spremenimo njegovo velikost. Element lahko zavrtimo za 180 stopinj tako, da ga najprej izberemo (kliknemo nanj), nato pa v orodni vrstici simulacijske sheme kliknemo na gumb z oznako *Format* in izberemo možnost *Flip block*. Blok lahko povečamo ali pomanjšamo, oz. preoblikujemo tako, da najprej nanj kliknemo (ga izberemo). Pri tem se na robovih prikažejo majhni črni kvadratki. Če postavimo kurzor na takšen črn kvadratek in držimo levi gumb na



Slika 5.11: Skupina blokov z oznako Matematične operacije (**Math Operations**) (levo okno) in gradnja lastnega simulacijskega modela (desno okno)

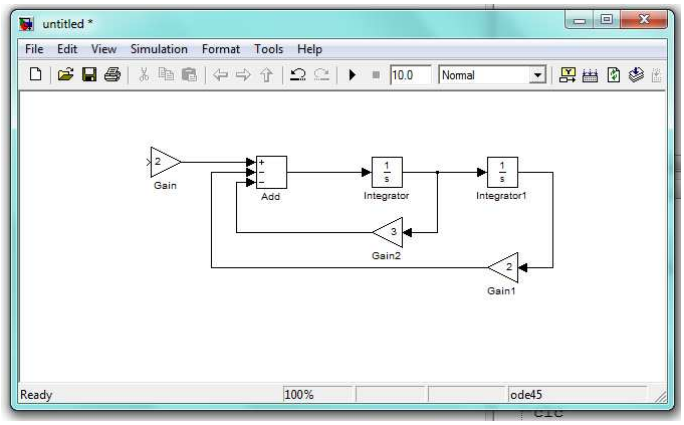


Slika 5.12: Komunikacijsko okno sumatorja (desno okno na sliki)

miški, bomo oblikovali velikost ikone tega bloka. To seveda vpliva le na prikaz bloka, ostale lastnosti pa ostajajo nespremenjene.

Povezovanje blokov je zelo preprosto. S kurzorjem se postavimo na izhod izbranega bloka in pritisnemo levi gumb na miški. Gumb držimo in pomikamo kurzor do vhoda bloka, ki ga želimo povezati.

Slika 5.13 prikazuje simulacijsko shemo v Simulinku za model, ki ga podaja enačba 5.1.



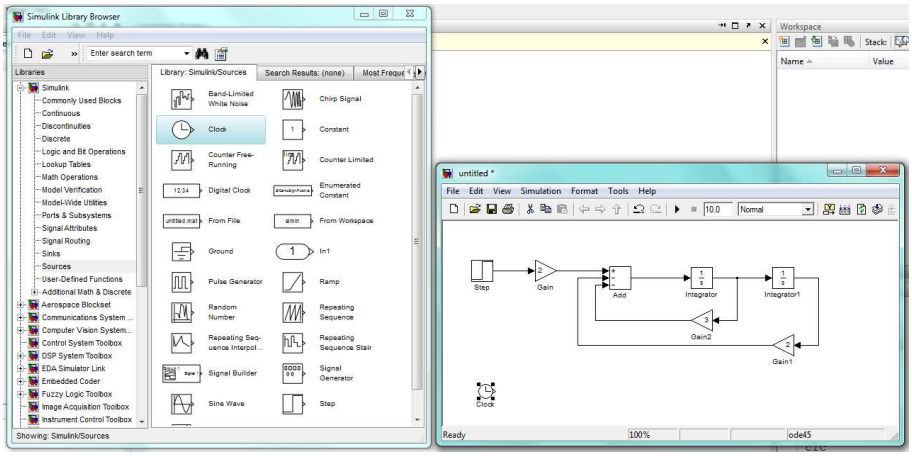
Slika 5.13: Simulink shema za sistem 2. reda

Opazimo, da je Simulinkov simulacijski model zelo podoben splošni simulacijski shemi na sliki 5.6.

Predno bomo lahko pričeli s simulacijo, pa moramo narediti še tri pomembne korake. Najprej moramo dodati vzbujalni signal. Predpostavimo, da želimo uporabiti enotino stopnico. V ta namen odprimo skupino z oznako **Sources**, kjer najdemo številne bloke, s pomočjo katerih lahko generiramo potrebne signale (glej levo okno na sliki 5.14). Iz te skupine prenesemo blok z oznako **Step** in **Clock**. Stopničasti signal (blok **Step**) povežemo z vhodom v model (glej desno okno na sliki 5.14)

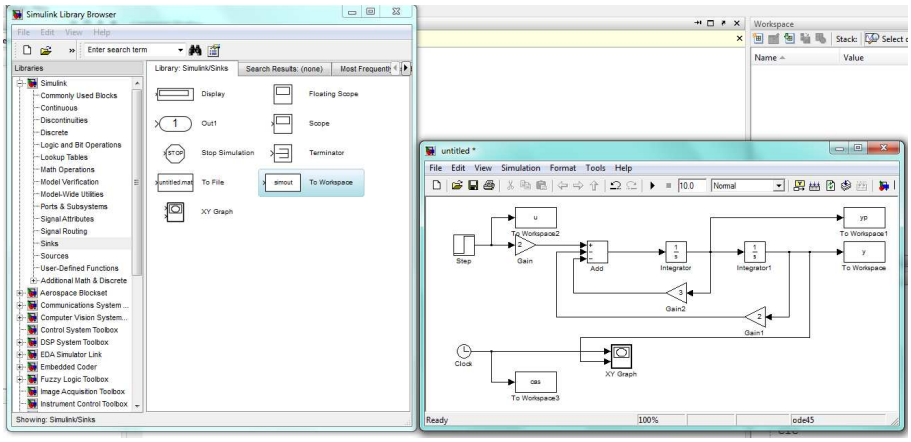
Z dvojnim klikom na blok **Step** odpremo pripadajoče komunikacijsko okno, kjer nastavimo parametre tega bloka na naslednje vrednosti: čas nastopa stopničastega signala (*Step time*) je 0, začetna vrednost (*Initial value*) je 0 in končna vrednost (*Final value*) je 1. Parameter *Sample time* oz. čas vzorčenja pustimo na vrednosti 0, saj imamo opravka z zveznim modelom.

Rezultate simulacije lahko opazujemo na dva osnovna načina in sicer sprotno, med samo simulacijo in po končanem simulacijskem teku. V ta namen uporabljamo bloke za iz skupine **Sinks** (levo okno na sliki 5.15). V model prenesemo bloka z oznako **XY Graph** in **To Workspace**. Blok **XY Graph** služi za sproten prikaz signalov. Na prvi vhod tega bloka priključimo izhod bloka **Clock**, to



Slika 5.14: Vzbujalni signali (levo okno) in dopolnjen simulacijski model (desno okno)

je bloka, ki poskrbi za možnost uporabe vektorja časa, ki ga program generira med simulacijskem tekom. Na drugi vhod pa priključimo signal, ki bi ga želeli opazovati. V našem primeru se odločimo za opazovanje izhodnega signala $y(t)$ (desno okno na sliki 5.15). Z dvojnim klikom na ta blok ugotovimo, da je mogoče izbrati območje opazovanja signala po x in y osi.

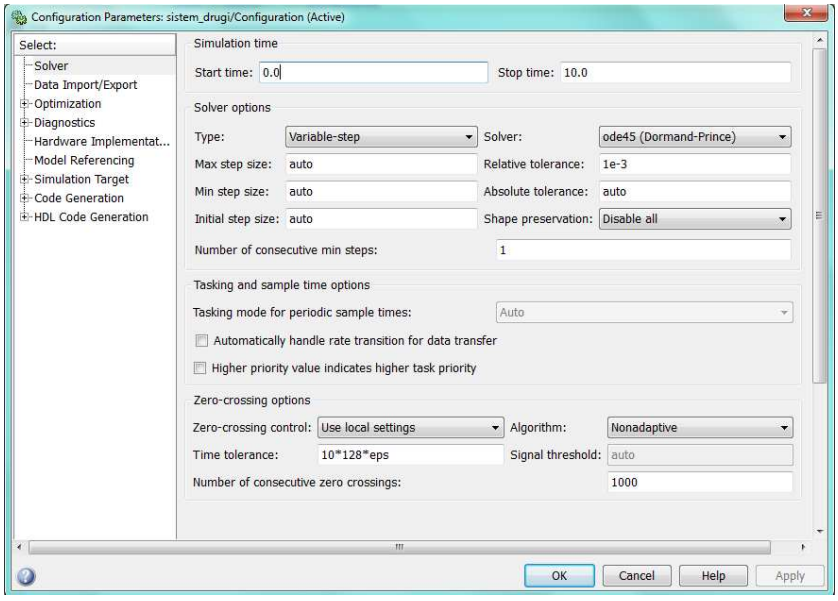


Slika 5.15: Bloki za opazovanje (levo okno) in dopolnjen simulacijski model (desno okno)

Zelo uporaben je tudi blok **To Workspace**, s pomočjo katerega lahko poskrbimo, da se bodo po končanem simulacijskem teku v delovnem prostoru Matlaba

nahajali signali, ki bi jih želeli uporabiti. Če odpremo komunikacijsko okno tega bloka, ugotovimo, da lahko izberemo ime spremenljivke in tudi obliko oz. *format* podatkovne strukture. Običajno uporabimo format na *Array*, kar pomeni, da bomo po končanem simulacijskem teku imeli v delovnem prostoru celoten vektor določenega signala z izbranim imenom in ga bomo lahko direktno uporabili v okolju Matlab. V našem primeru smo se odločili za uporabo štirih tovrstnih blokov, ki bodo poskrbeli, da se bo po končanem simulacijskem teku v delovnem prostoru nahajala neodvisna spremenljivka čas pod imenom *cas*, vhodni signal u, izhod y in odvod izhodnega signala pod imenom *yp* (glej desno okno na sliki 5.15).

Sedaj pa moramo ustrezno nastaviti samo še parametre simulacije. To najlaže storimo tako, da kliknemo v orodni vrstici simulacijske sheme na gumb *Simulation* in izberemo možnost *Configuration Parameters*. Pri tem se odpre okno uporabniškega vmesnika, kot je prikazano na sliki 5.16.

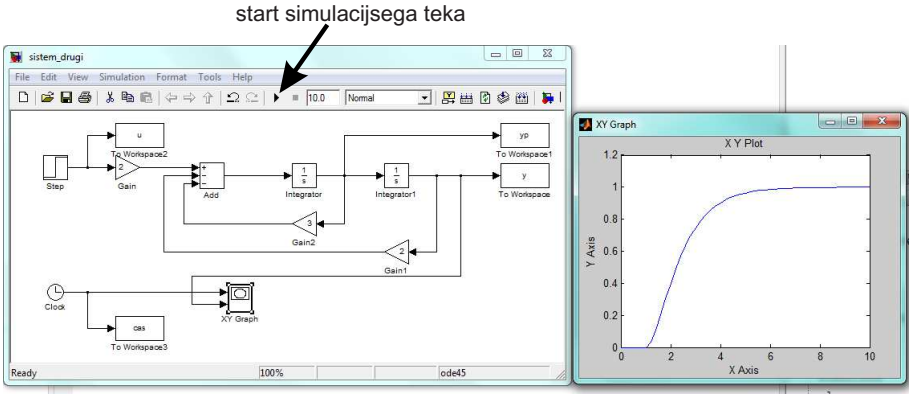


Slika 5.16: Okno uporabniškega vmesnika za nastavitev parametrov simulacije

Najprej se odločimo za primerno dolžino simulacijskega teka. Ker je daljša od obeh časovnih konstant enaka 1 sekundo, naj simulacija traja desetkrat tako (nastavitev v okencu *Stop time*). Nastavitev ostalih parametrov običajno pri enostavnejših simulacijah ni potrebno spreminjati. Omenimo pa, da v tem oknu lahko izbiramo med različnimi numeričnimi (integracijskimi) algoritmi, nastavljamo zahteve za natančnost izračunov in podobno.

Na koncu shranimo model pod želenim imenom in sprožimo izvajanje simulacijskega teka. To lahko naredimo na več načinov. Najpreprostejša možnost je ta, da kliknemo na črno puščico v orodni vrstici simulacijske sheme (glej levo okno na sliki 5.17). Blok za sprotni prikaz (XY Graph) poskrbi, da se ob proženju simulacijskega teka odpre okence, kamor se med simulacijskim tekom izrisuje izbrani signal (desno okno na sliki 5.17).

Po končanem simulacijskem teku se v delovnem prostoru nahajajo spremenljivke cas, u, y in yp, ki jih lahko uporabimo za grafični prikaz simulacijskih rezultatov.

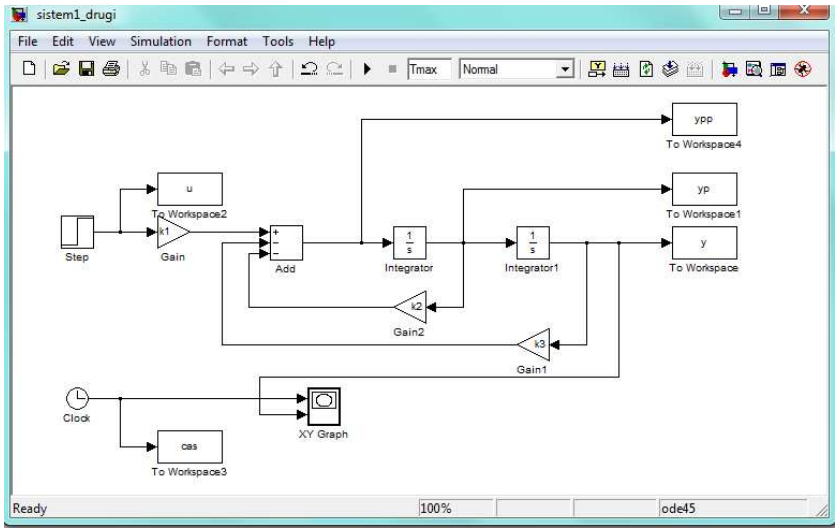


Slika 5.17: Sprotni grafični prikaz rezultatov (desno okno), ki ga je povzročil blok **XY Graph** v simulacijski shemi (levo okno)

Oglejmo si še enkrat simulacijsko shemo, ki smo jo pravkar zgradili v Simulinku. Parametre modela smo v tej shemi določili kar z ustreznimi vrednostmi, ki smo jih vnesli preko komunikacijskih oken uporabljenih blokov. Namesto konkretnih vrednosti pa lahko uporabimo tudi imena spremenljivk ali celo matematične izraze. Pri tem je pomembno, da v delovnem prostoru Matlaba priredimo uporabljenim spremenljivkam želene vrednosti, preden zaženemo simulacijo. V nasprotnem primeru izračun seveda ne bo mogoč in Simulink bo javil napako.

Da bi ilustrirali nakazano možnost, preuredimo simulacijsko datoteko, kot je prikazano na sliki 5.18 in jo shranimo pod imenom sistemdrugi.mdl. Parametre smo preimenovali v spremenljivke $k1$, $k2$ in $k3$, v delovni prostor pa smo napeljali tudi drugi odvod $\ddot{y}(t)$. Poleg tega smo označili dolžino simulacijskega teka s $Tmax$.

Namesto da v komandnem oknu Matlaba odtipkamo več komand, lahko komande



Slika 5.18: Simulacijska shema sistemdrugi.mdl

vpišemo v Matlabovo tekstovno datoteko *.m -t.j. program v jeziku Matlab.

```
% Simulacija sistema 2.  reda
%
% Nastavitev parametrov modela:
k1=2;
k2=3;
k3=2;
%
% Dolžina simulacijskega teka:
Tmax=10;
%
%Proženje simulacijskega teka:
sim('sistemdrugi.mdl',Tmax);
%
subplot(2,2,1)
plot(cas,u,'r','Linewidth',3);
xlabel('čas [s]','Fontsize',16)
ylabel('u(t) - vhodni signal','Fontsize',16)
grid
subplot(2,2,2)
plot(cas,y,'b','Linewidth',3);
xlabel('čas [s]','Fontsize',16)
```

```
ylabel('y(t) - izhodni signal','Fontsize',16)
grid
subplot(2,2,3)
plot(cas,yp,'c','Linewidth',3);
xlabel('čas [s]','Fontsize',16)
ylabel('yp(t) - prvi odvod izhodnega signala','Fontsize',16)
grid
subplot(2,2,4)
plot(cas,ypp,'g','Linewidth',3);
xlabel('čas [s]','Fontsize',16)
ylabel('ypp(t) -drugi odvod izhodnega signala','Fontsize',16)
grid
```

Spomnimo se, da je desno od oznake % v vrstici komentar, ki smo ga dodali zaradi preglednosti. Opozorimo posebej samo na vrstico z naslednjo vsebino:

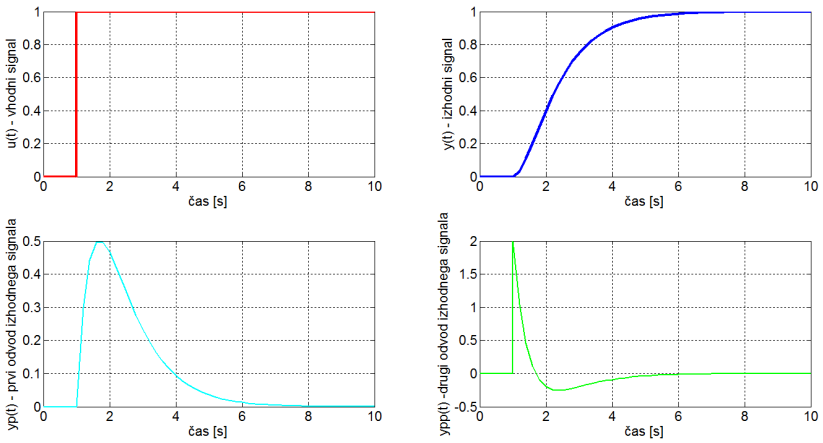
sim('sistemdrugi.mdl',Tmax);

Uporaba funkcije **sim** omogoča proženje simulacije iz okolja Matlab. Pri tem smo kot vhodni parameter najprej navedli ime simulacijskega modela, nato pa še spremenljivko, ki definira dolžino simulacijskega teka.

Če sedaj v komandnem oknu natipkamo ime komandne datoteke oz. Matlabovega programa, se izvrši simulacijski tek in narišejo prikaz rezultati simulacije, kot prikazuje slika 5.19.

5.4.2 Analiza modelov v Simulinku

Analiziranje modelov zgrajenih v Simulinku je možno na več različnih načinov: z uporabo orodja **Linear Analysis**, če v orodni vrstici izberemo (**Tools Control design Linear Analysis**) ali iz Matlaba z uporabo funkcij `linmod` in `trim`.



Slika 5.19: Rezultati simulacijskega izračuna

Linear Analysis

Orodje omogoča linearno analizo dela modela, ki ga označimo z vhodno in izhodno točko (določimo z desnim klikom ustreznega signala in izberemo Linearization points/Input ali Output Point). Linearni model med vhodno in izhodno točko dobimo v obliki prenosne funkcije ali prostora stanj. Nadalje lahko izberemo nekaj osnovnih eksperimentov v časovnem in frekvenčnem prostoru s pomočjo lineariziranega modela: odziv na stopnico, impulzni odziv, Bodejev diagram, Nyquistov diagram, izris polov in ničel v s-ravnini, ...

Linmod

`Linmod` je funkcija s pomočnjo katere lahko linearizirамo splošni nelinearni sistem diferencialnih enačb okrog delovne točke. Rezultat linearizacije je zapis lineariziranega modela v prostoru stanj.

Če linearizirамo Simulink model, potem moramo definirati vhode in izhode iz dela, ki ga želimo linearizirati, s pomočjo blokov **In1** (v skupini Sources) in **Out1**(v skupini Sinks).

Trim

Funkcija `trim` omogoča izračun ustaljenega stanja obravnavanega modela. Ustaljeno stanje lahko izračunamo pri določeni vrednosti vhoda, lahko pa izračunavamo vhodni signal in stanja pri znanem izhodu v ustaljenem stanju. Vgrajen je optimizacijski postopek, ki minimizira vrednosti odvodov.

5.4.3 Poglobljena uporaba Simulinka

V tem delu bo natančneje predstavljeno delovanje Simulinka ter zgradba S-funkcij.

Delovanje Simulinka

Modeli v Simulinku so predvsem enostavni za gradnjo, s stališča učinkovitost izvajanja simulacije pa niso optimalni. Prvi korak procesiranja modela je generiranje podatkovne strukture, ki je optimalna za simulacijo. Vključijo se ustrezni bloki, ki se tudi razvrstijo. Simulacija poteka s pomočjo numerične integracije diferencialnih enačb. Simulacijo lahko zaganjamo iz Simulink okna ali pa iz Matlabovega ukaznega okna z uporabo ukaza `sim`.

S-funkcije

Ko zgradimo Simulink model in ga shranimo, sistem zgradi t.i. S-funkcijo, ki je dostopna iz Matlaba. V tej funkciji je definirana dinamika modela. Tako definiran model lahko simuliramo (integriramo), lineariziramo in iščemo ustaljena stanja (ravnotežne točke) modela. S-funkcija deluje enako kot katerakoli Matlab funkcija, ima pa določeno sintakso:

sys=proces(t,x,u,flag)

kjer je proces ime modela in parameter flag določa informacijo, ki jo funkcija vrne v izhodno spremenljivko sys. Na primer ob vrednosti flag=1 funkcija vrne odvode stanj modela pri vektorju stanj x in vhodni spremenljivki u. S-funkcijo

lahko napišemo tudi v obliki navadne M-datoteke ali MEX-datoteke, ki je C-jevska ali Fortranska funkcija. Tako lahko ločimo tri različne načine pisanja S-funkcij:

- grafični GUI Simulink,
- M-datoteka Matlab,
- MEX-datoteka C ali Fortran funkcija.

Način podajanja S-funkcij je odvisen od zahtevnosti podajanja, uporabe S-funkcije in hitrosti, ki je zahtevana za izvajanje.

Uporabniško definirano S-funkcijo lahko uvedemo v Simulink shemo tako, da jo pridružimo posebnemu bloku (S-function block). Z maskiranjem lahko takemu bloku dodamo še običajni vmesnik, ki ga imajo ostali Simulink bloki. Na ta način je uporabniku omogočeno lastno generiranje dinamičnih modelov.

Kako delujejo S-funkcije? Funkcije so zgrajene tako, da omogočajo izgradnjo modelov za reševanje časovno zveznih, diskretni in hibridnih problemov. Zato ima S-funkcija točno določeno strukturo. Ključni element S-funkcije je parameter **flag**, ki definira informacijo, ki jo daje funkcija ob klicu na izhodu:

- flag=0 S-funkcija vrne velikost parametrov in začetnih pogojev,
- flag=1 S-funkcija vrne odvode stanj (dx/dt),
- flag=2 S-funkcija vrne predikcijo diskretnih stanj (x(k+1)),
- flag=3 S-funkcija vrne izhodno spremenljivko,
- flag=4 S-funkcija vrne čas naslednjega časovnega trenutka.

Vsaka situacija posreduje del informacije, ki je potreben za izvajanje simulacije.

V primeru flag=0 kličemo funkcijo takole:

```
>>flag=0
>>[sizes,x0]=sfun([],[],[],flag)
```

Vektor x0 je vektor začetnih stanj, v vektorju **sizes** pa je naslednja informacija:

- sizes(1) število zveznih stanj,
- sizes(2) število diskretnih stanj,
- sizes(3) število izhodnih spremenljivk,
- sizes(4) število vhodnih spremenljivk,
- sizes(5) število nezveznosti,
- sizes(6) število algebrajskih zank.

V primeru zveznega sistema sta za simulacijo potrebni le informaciji o odvodih stanj v vsakem trenutku simulacije (flag=1) in izhodi sistema (flag=3). Opozoriti velja, da se za razliko od klica funkcije z zastavico flag=0, ostali klici pojavljajo v vsakem računskem koraku simulacije.

Pretvorba S-funkcije v Simulink blok. Katerokoli obliko S-funkcije lahko pretvorimo v Simulink blok. V ta namen je potrebno v funkcijski blok **S-function block** oz. v njen uporabniški vmesnik vpisati ime funkcije in parametre. Blok namreč omogoča prenos določenih parametrov, ki sledijo formalni strukturi parametrov t, x, u in flag. Medsebojno jih ločimo z vejico. Na koncu postopka običajno tak blok tudi maskiramo.

Izgradnja S-funkcije v obliki M-datoteke je najlažja, če si pomagamo s predlogo **sfuntmpl**, ki se nahaja na poddirektoriju toolbox \simulink sistemskega Matlabovega direktorija.

5.5 Analiza modelov v okolju Matlab-Simulink

Matlab vsebuje nekaj ukazov, ki omogočajo analizo Simulink modelov. Ogledali si bomo naslednje ukaze:

- `model` - funkcija vrne osnovne podatki modela v Simulinku,
- `sim` - zagon simulacije iz okolja Matlab,

- `linmod` - omogoča linearizacijo modelov,
- `trim` - izračuna ustaljene vrednosti spremenljivk modela,
- `fminsearch` - omogoča optimiranje modelov.

5.5.1 Osnovni podatki o modelu

Pred simulacijo, linearizacijo, računanjem ustaljenih (ravnotežnih) stanj ali pred optimizacijo je pogosto koristno pridobiti nekaj podatkov o modelu, ki je sicer izveden v okolju Simulink. Temu je namenjena funkcija `model`.

Funkcija `model`

Ukaz `model` vrne nekaj osnovnih podatkov modela v Simulinku. Funkcijo kličemo na sledeče načine:

```
sizes=model
[sizes,x0]=model
[sizes,x0,states]=model
```

`model`	Ime modela v Simulinku (brez navednic in brez podaljška .mdl)
`sizes(1)`	Število zveznih stanj
`sizes(2)`	Število diskretnih stanj
`sizes(3)`	Število izhodov
`sizes(4)`	Število vhodov
`sizes(5)`	Število nezveznih korenov
`sizes(6)`	Zakasnitveni atribut (0...zakasnitev 1...takojšna reakcija na izhodu)
`sizes(7)`	Število različnih časov vzorčenja
`x0`	Začetna stanja
`states`	Informacija, kako je Simulink procesiranje zgradilo vektor stanj (na katero mest

Primer 5.1 Karakteristike modela ekološkega sistema roparjev in žrtev (primer 4.3, slika 4.14)

```
[sizes,x0,states]=lisice

sizes =
     2
     0
     2
     0
     0
     0
     1

x0 =
   520
    85

states =
    'lisice/Integrator'
    'lisice/Integrator1'
```

Sistem ima 2 stanji, 2 izhoda, izpišeta se začetni stanji (520 zajcev in 85 lisic) ter informacija, kako je Simulink procesiranje izbralo stanja: prvo stanje (populacija zajcev) - `'lisice/Integrator'` in drugo stanje (populacija lisic) - `'lisice/Integrator1'` (prvi del teksta je ime modela, drugi del je ime bloka, katerega izhodna veličina je ustrezno stanje). □

5.5.2 Izvajanje simulacije Simulink modela iz okolja Matlab

Uporaba ukaza `sim` omogoča izvajanje simulacije iz komandnega okna ali iz Matlabove m-datoteke. Krmilne parametre simulacije lahko nastavljamo v Simulinku ali v Matlabu. Z ukazom `simset` nastavljamo krmilne parametre simulacije, z ukazom `simget` pa prečitamo trenutne nastavitve.

Simulacija s funkcijo `sim`

S funkcijo `sim` zaženemo model, ki je opisan s Simulink shemo. Tako lahko v okolju Matlab programiramo kompleksne eksperimente, ki vključujejo simu-

lacijske teke modela v Simulinku. Ponavadi je koristno, da modelu v Simulinku z out bloki iz knjižnice Sinks označimo izhode oz. signale, ki jih opazujemo.

`sim('model')`
ukaz simulira Simulink model `model.mdl`. Veljajo krmilni parametri, kot so nastavljeni v Simulink shemi.

`[t,x,y]=sim('model', timespan)`
`t` je vektor, ki vsebuje trenutke, v katerih se izračunajo rezultati simulacije. `x` je matrika stanj (vsaka kolona opisuje eno stanje)- stanja so izhodi integratorjev. `y` je matrika izhodov (vsaka kolona opisuje en izhod, izhodi so določeni z out bloki v Simulink shemi). `timespan` je v splošnem vektor. Če ima vektor en element, je to končni čas simulacije `tf`, če ima dva elementa, sta to začetni in končni čas simulacije `[t0 tf]`, če pa je več elementov, so to trenutki, v katerih želimo dobiti rezultate simulacije `[t0 t1 t2 ... tf]`.

`[t,x,y]=sim('model', timespan, options)`
options vsebuje vse mogoče nastavitve, ki se sicer lahko nastavijo v Simulink shemi, npr. tolerance, integracijsko metodo, minimalni in maksimalni dopustni računski korak,... Za nastavitev parametra `options` se uporablja posebna fukcija `simset`. Če želimo izbrati integracijsko metodo `ode45`, uporabimo ukaz

`options=simset('solver','ode45')`.

Nastavitev krmilnih parametrov simulacije s funkcijo `simset`

Matlabov ukaz `simset` ima naslednjo obliko:

`options=simset('ime_1',vrednost_1,'ime_2',vrednost_2,...)`

`ime` je vedno tekstovna spremenljivka, `vrednost` pa je numerična ali tekstovna. Nekaj značilnih parov (`ime, vrednost`) je navedenih v tabeli

`solver`	Izbere integracijsko metodo:`'ode45'`, `'ode23'`, `'ode113'`, `'ode15s'`, `'ode23s'`, `'ode5'`, `'ode4'`, `'ode3'`, `'ode2'`, `'ode1'`, `'FixedStepDiscrete'`, `'VariableStepDiscrete'`
`RelTol`	Prenastavi vrednost nastavitve relativne napake integracijske metode (**Relative tolerance**) v Simulink modelu
`AbsTol`	Prenastavi vrednost nastavitve absolutne napake integracijske metode (**Absolute tolerance**) v Simulink modelu
`MaxStep`	Prenastavi vrednost za največjo dopustno vrednost računalniškega koraka (**Max step size**) v Simulink modelu
`InitialStep`	Prenastavi začetno vrednost računalniškega koraka (**Initial step size**) v Simulink modelu
`InitialState`	Prenastavi vrednost začetnih pogojev v Simulink modelu (**Initial condition**) ali v Workspace I/O pri konfiguraciji simulacije

Primer:

```
options=simset('RelTol', 1.0E-4,'Solver','ode4')
```

Izbrali smo relativno napako 10^{-4} in integracijsko metodo Runge-Kutta 4 reda.

Čitanje krmilnih parametrov simulacije s funkcijo `simget`

Matlabov ukaz `simget` uporabimo za čitanje vseh krmilnih nastavitev simulacije ali za čitanje posameznih nastavitev.

Čitanje vseh nastavitev:

```
opts =simget(model)
```

Čitanje posamezne nastavitve

```
vrednost=(model,ime)
```

`model` je ime modela v Simulinku (z navednicami in brez podaljška .mdl), `ime` je ime krmilne spremenljivke, `vrednost` pa njena vrednost.

Primer 5.2 Risanje faznega portreta

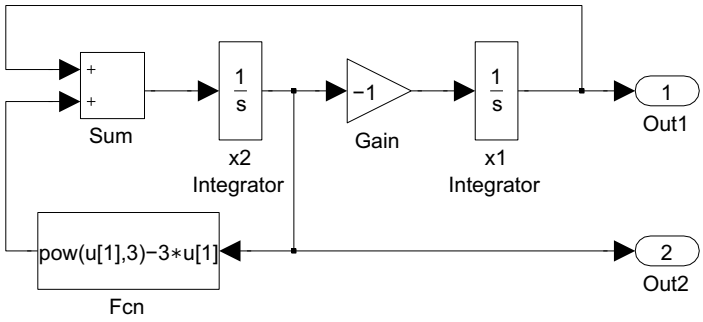
Fazni portret je običajni pripomoček zlasti pri analizi nelinearnih dinamičnih

sistemov. Analizirajmo sistem

$$\begin{aligned} \dot{x}_1 &= -x_2 \\ \dot{x}_2 &= x_1 + x_2^3 - 3x_2 \end{aligned} \tag{5.3}$$

Fazni portret pregledno opisuje dinamiko sistema v fazni ravnini (x_1,x_2) zlasti v bližini t.i. **ravnotežnih točk**, za katere velja $\dot{x}_1 = 0$ in $\dot{x}_2 = 0$. Če v nekem omejenem območju okoli ravnotežne točke ugotovimo, da trajektorije iz začetnih pogojev znotraj omenjenega območja konvergirajo v ravnotežno točko, pravimo, da je sistem v tem območju stabilen oz. da je ravnotežna točka stabilna. Za sistem 5.3 lahko ugotovimo, da ima eno stabilno ravnotežno točko pri $x_1 = 0$ in $x_2 = 0$.

Slika 5.20 prikazuje nelinearni model v okolju Simulink (datoteka `sysmdl_e.mdl`).



Slika 5.20: Simulink shema nelinearnega sistema 2. reda

Analizo modela začnemo z ukazom `model`

```
[sizes,x0,states]=sysmdl_e

sizes =
     2
     0
     2
     0
     0
     0
     1

x0 =
```

```
     0
     0

states =
    'sysmdl_e/x1 Integrator'
    'sysmdl_e/x2 Integrator'
```

Ugotovimo, da ima model dve stanji, prvo je povezano s spremenljivko x_1, drugo pa s spremenljivko x_2.

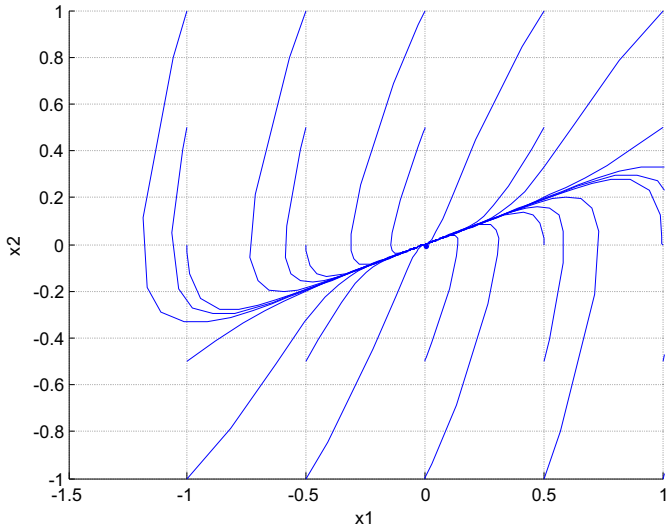
Fazni portret pa narišemo s programom (`ch8_a.m`), ki zaporedoma simulira sistem iz različnih začetnih pogojev okoli ravnotežne točke. Začetne pogoje spreminjamo z dvema `for` zankama, nastavitev začetnega pogoja pa izvedemo s funkcijo `simset`.

```
% Fazni portret simulink modela sysmdl_e
%
for x1 = -1:0.5:1
  for x2 = -1:0.5:1
    opts = simset('InitialState',[x1,x2]) ;
    [t,x,y] = sim('sysmdl_e',15,opts) ;
    hold on ;
    plot(x(:,1),x(:,2)) ;
  end
end
axis([-1.5,1,-1,1]);
xlabel('x1') ;
ylabel('x2') ;
grid ;
```

Slika 5.21 prikazuje fazni portret sistema 2. reda. Opazimo, da gredo vse trajektorije proti koordinatnemu izhodišču, ki torej predstavlja stabilno ravnotežno točko.

□

Primer 5.3 Quiverjev portret



Slika 5.21: Fazni portret nelinearnega sistema 2. reda

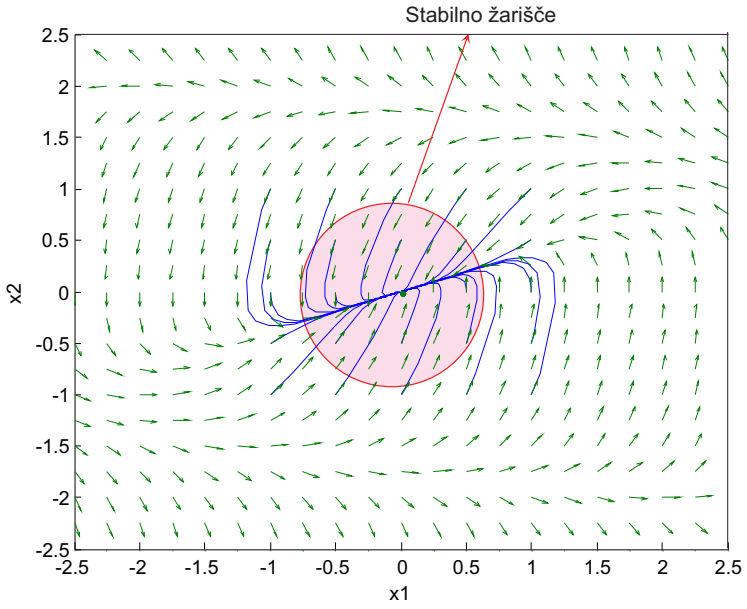
Vendar pa je s takim pristopom težko določiti lastnosti dinamičnega sistema v širšem področju. Za to je dosti primernejši Quiverjev portret, ki je sestavljen iz velikega števila simulacij, ki pa trajajo le en računski korak. Za vsako simulacijo narišemo trajektorijo s puščico na koncu omenjenega računskega koraka.

Program v Matlabu je naslednji (datoteka `ch8_b.m`):

```
%Quiverjev portret sistema sysmdl_e
h = 0.01 ; % Fiksni računski korak
opts = simset('Solver','ode5','FixedStep',h) ;
x1 = -2.5:0.25:2.5 ;   % Dolocitev obmocja
x2 = -2.5:0.25:2.5 ;
[nr,nc] = size(x1) ;
x1m = zeros(nc,nc) ;
x2m = x1m ;
for nx1 = 1:nc
  for nx2 = 1:nc
    opts = simset(opts,'InitialState',[x1(nx1),x2(nx2)]) ;
    [t,x,y] = sim('sysmdl_e',h,opts) ;
    dx1 = x(2,1)-x1(nx1) ;
    dx2 = x(2,2)-x2(nx2) ;
    l = sqrt(dx1^2 + dx2^2)*7.5 ;  % skaliranje puscic
    if l > 1.e-10
```

```
      x1m(nx2,nx1)=dx1/l ;
      x2m(nx2,nx1)=dx2/l ;
    end
  end
end
quiver(x1,x2,x1m,x2m,0) ;
axis([-2.5,2.5,-2.5,2.5]) ;
xlabel('x1') ;
ylabel('x2') ;
grid ;
```

Slika 5.22 prikazuje Quiverjev portret nelinearnega sistema 2. reda. Tudi v tem primeru opazimo, da je delovanje okoli koordinatnega izhodišča stabilno, da pa sistem postane nestabilen za bolj oddaljene začetne pogoje. V sredini je vrisan tudi fazni portret iz slike 5.21.



Slika 5.22: Quiverjev portret nelinearnega sistema 2. reda

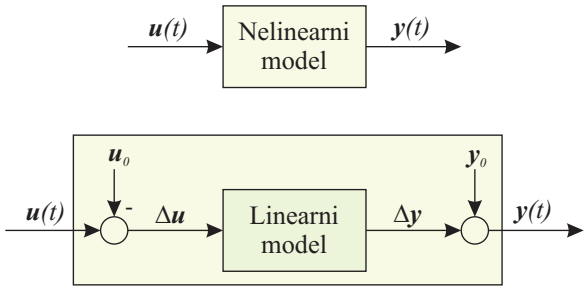
□

5.5.3 Linearizacija modelov

Modeli realnih procesov so običajno nelinearni. Linearizacija pa je zelo pogost pristop pri analizi in načrtovanju, ki predpostavi približno linearno delovanje dinamičnega sistema v nekem manjšem operativnem področju. Na področju vodenja sistemov lahko v zvezi s tem omenimo dve pomembni značilnosti:

- Velikokrat želimo proučevati vedenje sistema okoli nekih operativnih, delovnih vrednosti spremenljivk. Govorimo tudi o vedenju sistema v bližnji okolici delovne točke. Delovanje v bližini delovne točke je zlasti običajno za regulacijske sisteme, ki jih regulacijski algoritem drži čim bliže delovnih oz. želenih vrednosti.
- Obstaja zelo veliko metod za analizo in načrtovanje linearnih sistemov (frekvenčne metode, diagram lege korenov, analiza stabilnosti, ...) in zelo malo metod za nelinearne sisteme.

Slika 5.23 prikazuje, na kakšen način uporabljamo nelinearni in linearni model.



Slika 5.23: Nelinearni in linearni model

Predvsem se moramo zavedati, da so procesne spremenljivke sestavljene iz delovnih vrednosti in majhnih sprememb. Tako je vhodni signal sestavljen iz

$$\boldsymbol{u}(t) = \boldsymbol{u}_0 + \boldsymbol{\Delta u}$$

izhodni signal pa

$$\boldsymbol{y}(t) = \boldsymbol{y}_0 + \boldsymbol{\Delta y}$$

$\boldsymbol{u}_0$ in $\boldsymbol{y}_0$ sta delovni (ustaljeni) vrednosti vhodnega in izhodnega signala, $\boldsymbol{\Delta u}$ in $\boldsymbol{\Delta y}$ pa sta majhni spremembi. Pri uporabi linearnega modela moramo zato

od operativnega vhodnega signala $\boldsymbol{u}(t)$ odšteti delovno vrednost $\boldsymbol{u}_0$, da dobimo signal $\boldsymbol{\Delta u}$, ki je vhodni signal lineariziranega modela. Izhodu lineariziranega modela $\boldsymbol{\Delta y}$ pa moramo nato prišteti delovno vrednost izhodnega signala $\boldsymbol{y}_0$ in tako dobimo operativni izhodni signal $\boldsymbol{y}(t)$.

Ker je postopek linearizacije najbolj sistematičen v prostoru stanj, imamo še stanje sisteme $\boldsymbol{x}(t)$, ki je prav tako sestavljeno iz delovne vrednosti in majhne spremembe

$$\boldsymbol{x}(t) = \boldsymbol{x}_0 + \boldsymbol{\Delta x}$$

Izpeljava formule za linearizacijo

Nelinearni dinamični in časovno spremenljivi model podaja enačba stanj

$$\dot{\boldsymbol{x}} = \boldsymbol{f}(t, \boldsymbol{x}, \boldsymbol{u}) \tag{5.4}$$

Delovna točka $\mathbf{0}$ je določena s spremenljivkami t_0, $\boldsymbol{x}_0$, $\boldsymbol{u}_0$ in $\boldsymbol{y}_0$ in velja enačba

$$\dot{\boldsymbol{x}}_0 = \boldsymbol{f}(t_0, \boldsymbol{x}_0, \boldsymbol{u}_0) \tag{5.5}$$

Če spremenimo vhodno spremenljivko $\boldsymbol{u}$ iz delovne vrednosti $\boldsymbol{u}_0$ za $\boldsymbol{\Delta}u$, potem se stanje $\boldsymbol{x}$ spremeni iz vrednosti $\boldsymbol{x}_0$ za $\boldsymbol{\Delta}x$. Zato lahko enačbo 5.5 spremenimo v

$$\frac{d(\boldsymbol{x}_0 + \boldsymbol{\Delta x})}{dt} = \boldsymbol{f}(t_0, \boldsymbol{x}_0 + \boldsymbol{\Delta x}, \boldsymbol{u_0} + \boldsymbol{\Delta u}) \tag{5.6}$$

Na levi strani enačbe 5.6 uporabimo pravilo za odvod vsote, desno stran pa razvijemo v Taylorjevo vrsto in upoštevamo le prve člene.

$$\dot{\boldsymbol{x}}_0 + \boldsymbol{\Delta \dot{x}} \approx \boldsymbol{f}(t_0, \boldsymbol{x}_0, \boldsymbol{u}_0) + \left.\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}}\right|_{\mathbf{0}} \boldsymbol{\Delta}x + \left.\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{u}}\right|_{\mathbf{0}} \boldsymbol{\Delta}u \tag{5.7}$$

Izraz $\left.\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}}\right|_{\mathbf{0}}$ je parcialni odvod funkcije $\boldsymbol{f}$ na spremenljivko $\boldsymbol{x}$ v delovni točki $\mathbf{0}$, torej je treba v izraz vstaviti vrednosti za t_0, $\boldsymbol{x}_0$ in $\boldsymbol{u}_0$. Podobno velja za izraz

$\left.\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{u}}\right|_{\mathbf{0}}$. Ker sta prva člena na obeh straneh enačbe 5.7 enaka, dobimo enačbo

$$\boldsymbol{\Delta \dot{x}} \approx \left.\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}}\right|_{\mathbf{0}} \boldsymbol{\Delta} x + \left.\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{u}}\right|_{\mathbf{0}} \boldsymbol{\Delta} u = \boldsymbol{A \Delta x} + \boldsymbol{B \Delta u} \tag{5.8}$$

oz.

$$\begin{aligned}
\begin{bmatrix} \Delta \dot{x}_1 \\ \Delta \dot{x}_2 \\ \vdots \\ \Delta \dot{x}_{n-1} \\ \Delta \dot{x}_n \end{bmatrix} &= \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \frac{\partial f_1}{\partial x_3} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \frac{\partial f_2}{\partial x_3} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_{n-1}}{\partial x_1} & \frac{\partial f_{n-1}}{\partial x_2} & \frac{\partial f_{n-1}}{\partial x_3} & \cdots & \frac{\partial f_{n-1}}{\partial x_n} \\ \frac{\partial f_n}{\partial x_1} & \frac{\partial f_n}{\partial x_2} & \frac{\partial f_n}{\partial x_3} & \cdots & \frac{\partial f_n}{\partial x_n} \end{bmatrix}_0 \begin{bmatrix} \Delta x_1 \\ \Delta x_2 \\ \vdots \\ \Delta x_{n-1} \\ \Delta x_n \end{bmatrix} \\
&+ \begin{bmatrix} \frac{\partial f_1}{\partial u_1} & \frac{\partial f_1}{\partial u_2} & \frac{\partial f_1}{\partial u_3} & \cdots & \frac{\partial f_1}{\partial u_l} \\ \frac{\partial f_2}{\partial u_1} & \frac{\partial f_2}{\partial u_2} & \frac{\partial f_2}{\partial u_3} & \cdots & \frac{\partial f_2}{\partial u_l} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_{n-1}}{\partial u_1} & \frac{\partial f_{n-1}}{\partial u_2} & \frac{\partial f_{n-1}}{\partial u_3} & \cdots & \frac{\partial f_{n-1}}{\partial u_l} \\ \frac{\partial f_n}{\partial u_1} & \frac{\partial f_n}{\partial u_2} & \frac{\partial f_n}{\partial u_3} & \cdots & \frac{\partial f_n}{\partial u_l} \end{bmatrix}_0 \begin{bmatrix} \Delta u_1 \\ \Delta u_2 \\ \vdots \\ \Delta u_{l-1} \\ \Delta u_l \end{bmatrix}
\end{aligned} \tag{5.9}$$

Enačbi 5.8 in 5.9 torej opisujeta linearni model v delovni točki. $\boldsymbol{A}$ in $\boldsymbol{B}$ sta Jakobijevi matriki in sta odvisni od delovne točke. V kolikor se delovna točka ne spreminja, je možno uporabiti enkrat izračunane vrednosti.

Podobno lineariziramo izhodno enačbo

$$\boldsymbol{y} = \boldsymbol{g}(t, \boldsymbol{x}, \boldsymbol{u}) \tag{5.10}$$

$$\boldsymbol{\Delta y} \approx \left.\frac{\partial \boldsymbol{g}}{\partial \boldsymbol{x}}\right|_{\mathbf{0}} \boldsymbol{\Delta x} + \left.\frac{\partial \boldsymbol{g}}{\partial \boldsymbol{u}}\right|_{\mathbf{0}} \boldsymbol{\Delta u} = \boldsymbol{C \Delta x} + \boldsymbol{D \Delta u} \tag{5.11}$$

oz.

$$\begin{bmatrix} \Delta y_1 \\ \Delta y_2 \\ \vdots \\ \Delta y_{n-1} \\ \Delta y_n \end{bmatrix} = \begin{bmatrix} \frac{\partial g_1}{\partial x_1} & \frac{\partial g_1}{\partial x_2} & \frac{\partial g_1}{\partial x_3} & \cdots & \frac{\partial g_1}{\partial x_n} \\ \frac{\partial g_2}{\partial x_1} & \frac{\partial g_2}{\partial x_2} & \frac{\partial g_2}{\partial x_3} & \cdots & \frac{\partial g_2}{\partial x_n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \frac{\partial g_{n-1}}{\partial x_1} & \frac{\partial g_{n-1}}{\partial x_2} & \frac{\partial g_{n-1}}{\partial x_3} & \cdots & \frac{\partial g_{n-1}}{\partial x_n} \\ \frac{\partial g_n}{\partial x_1} & \frac{\partial g_n}{\partial x_2} & \frac{\partial g_n}{\partial x_3} & \cdots & \frac{\partial g_n}{\partial x_n} \end{bmatrix}_0 \begin{bmatrix} \Delta x_1 \\ \Delta x_2 \\ \vdots \\ \Delta x_{n-1} \\ \Delta x_n \end{bmatrix}$$

$$+ \begin{bmatrix} \frac{\partial g_1}{\partial u_1} & \frac{\partial g_1}{\partial u_2} & \frac{\partial g_1}{\partial u_3} & \cdots & \frac{\partial g_1}{\partial u_l} \\ \frac{\partial g_2}{\partial u_1} & \frac{\partial g_2}{\partial u_2} & \frac{\partial g_2}{\partial u_3} & \cdots & \frac{\partial g_2}{\partial u_l} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \frac{\partial g_{m-1}}{\partial u_1} & \frac{\partial g_{m-1}}{\partial u_2} & \frac{\partial g_{m-1}}{\partial u_3} & \cdots & \frac{\partial g_{m-1}}{\partial u_l} \\ \frac{\partial g_m}{\partial u_1} & \frac{\partial g_m}{\partial u_2} & \frac{\partial g_m}{\partial u_3} & \cdots & \frac{\partial g_m}{\partial u_l} \end{bmatrix}_0 \begin{bmatrix} \Delta u_1 \\ \Delta u_2 \\ \vdots \\ \Delta u_{l-1} \\ \Delta u_l \end{bmatrix} \tag{5.12}$$

Običajno **delovna točka** pomeni ustaljeno stanje, ko so spremenljivke stanja konstantne, kar pomeni, da so odvodi v delovni točki enaki nič. Pri analizi in načrtovanju v prostoru stanj govorimo tudi o **ravnotežni točki**. Torej velja

$$\dot{\boldsymbol{x}}_0 = \mathbf{0} = \boldsymbol{f}(t_0, \boldsymbol{x}_0, \boldsymbol{u}_0) \tag{5.13}$$

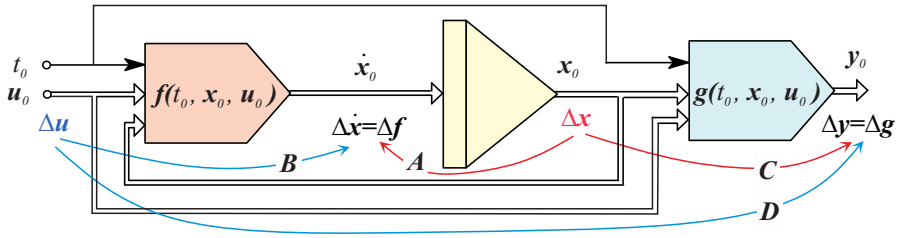
V tem primeru je možno izračunati delovno oz. ravnotežno točko z rešitvijo algebrajske enačbe 5.13. Pogosto kombiniramo algebrajski izračun delovne točke in nato linearizacijo v tej delovni točki. V okolju Matlab-Simulink je to kombiniranje funkcije **trim** in **linmod**.

Kako deluje linearizacija v simulacijskih okoljih?

Enačbe 5.8, 5.9, 5.11, 5.12 jasno nakazujejo, kako analitično pristopimo k linearizaciji. V okoljih, kot je Matlab-Simulink, pa se enačbe izvedejo numerično. Glede na standardne simulacije moramo sheme le dopolniti z informacijo o vhodnih in izhodnih signalih, saj od tega zavisijo matrike $\boldsymbol{B}$, $\boldsymbol{C}$ in $\boldsymbol{D}$. Pri običajnih simulacijah so sicer vhodi določeni z raznimi signali, izhodi pogosto z opazovalnimi bloki, vendar to ne zadošča. Stanja pa simulacijski sistem sam izbere tako, da je izhod vsakega integratorja eno stanje. Na zaporedje posameznih spremenljivk stanj v vektorju stanj ne moremo neposredno vplivati. Matrika $\boldsymbol{A}$ je

tudi edina, ki ni odvisna od izbire vhodnih in izhodnih signalov in se tudi brez izbranih vhodov in izhodov lahko določi.

Numerično poteka linearizacija s perturbacijskim postopkom, ki ga prikazuje slika 5.24.



Slika 5.24: Perturbacijski postopek pri linearizaciji

Linearizacijski algoritem mora najprej v shemi zagotoviti delovno točko $\mathbf{0}$ z nastavitvijo časa t_0, stanj $\boldsymbol{x}_0$, vhodov $\boldsymbol{u}_0$ in izhodov $\boldsymbol{y}_0$ na ustrezne delovne vrednosti. Če je sistem časovno nespremenljiv, vrednost t_0 nima vpliva na rezultate.

Nato se izvedejo majhne perturbacije oz. majhne spremembe stanj $\boldsymbol{\Delta x}$. S pomočjo modelne sheme se določi spremembe, ki jih $\boldsymbol{\Delta x}$ povzroči na odvodih stanj $\boldsymbol{\Delta \dot{x}} = \boldsymbol{\Delta f}$. Kvocienti med spremembo odvodov in spremembo stanj določajo matriko $\boldsymbol{A}$. Hkrati določimo tudi vpliv $\boldsymbol{\Delta x}$ na spremembo izhodov $\boldsymbol{\Delta y} = \boldsymbol{\Delta g}$. Kvocient spremembe izhodov in spremembe stanj določa matriko $\boldsymbol{C}$.

V drugem delu postopka se vsilijo perturbacije oz. majhne spremembe vhodnega signala $\boldsymbol{\Delta u}$. S pomočjo modelne sheme se določi spremembe, ki jih $\boldsymbol{\Delta u}$ povzroči na odvodih stanj $\boldsymbol{\Delta \dot{x}} = \boldsymbol{\Delta f}$. Kvocienti med spremembo odvodov in spremembo vhodov določajo matriko $\boldsymbol{B}$. Hkrati določimo tudi vpliv $\boldsymbol{\Delta u}$ na spremembo izhodov $\boldsymbol{\Delta y} = \boldsymbol{\Delta g}$. Kvocient spremembe izhodov in spremembe vhodov določa matriko $\boldsymbol{D}$.

Kot vsaka numerična metoda je tudi linearizacija podvržena nekaterim problemom. Eden od teh je izbira perturbacijskih sprememb $\boldsymbol{\Delta u}$ in $\boldsymbol{\Delta x}$. Orodja to izberejo sama glede na neke privzete nastavitve, kar pa je možno v primeru težav spremeniti.

Funkcija `linmod`

Funkcija `linmod` je osnovna Matlabova funkcija za linearizacijo zveznih dinamičnih sistemov podanih v Simulinku. Ukaz ima naslednjo obliko:

```
[A,B,C,D]= linmod(model,x0,u0,par)
```

Navajamo le bistvene parametre, pa še ti niso vedno potrebni. Edini obvezni parameter je model. Pomen je naslednji:

`model`	Ime Simulink modela v narekovajih brez podaljška `.mdl`, npr.. 'lisice'
`x0`	Stanje v delovni točki, okoli katere velja linearizirani model. Simulink lahko naredi linearizacijo za kakršno koli delovno stanje. Včasih je koristno predhodno uporabiti funkcijo `model`, da ugotovimo, kako so se izbrala stanja. Če parametra ni ali če je prazna matrika `[]`, potem je privzeta vrednost nič
`u0`	Vhodni signali v delovni točki, okoli katere velja linearizirani model. Če parametra ni ali če je prazna matrika `[]`, potem je privzeta vrednost nič. Zaporedje vhodnih signalov v vektorju ustreza oštevilčenosti `Inport` blokov v Simulink modelu.
`par`	Vektor s tremi komponentami. Prva vrednost določa perturbacijski korak pri linarizaciji oz. numeričnem izračunavanju parcialnih odvodov. Privzeta vrednost je 10^{-5}. Druga vrednost določa čas za linearizacijo t_0 (pomembno pri časovno spremenljivih sistemih). Privzeta vrednost je nič. Tretjo vrednost postavimo na 1, če žalimo izločiti stanja, ki niso na povezavah med vhodi in izhodi.

Postopek linearizacije v okolju Matlab-Simulink je naslednji:

1. Pripravimo Simulink model, dodamo `Inport` bloke na vhodne signale in `Outport` bloke na izhodne signale. Viri in ponori signalov iz knjižnic `Sources` in `Sinks` ne štejejo kot vhodi ali izhodi. Model je lahko brez `Inport` blokov, mora pa imeti vsaj en `Outport` blok.
2. Predhodno je priporočljivo uporabiti ukaz model, da ugotovimo osnovne lastnosti: število stanj in vrstni red v vektorju stanj.
3. V Matlabu uporabimo funkcijo `linmod`.

Primer 5.4 Linearizacija modela

Linearizirajmo nelinearni sistem iz primera 5.2, ki ga opisujeta enačbi

$$\begin{aligned} \dot{x}_1 &= f_1 = -x_2 \\ \dot{x}_2 &= f_2 = x_1 + x_2^3 - 3x_2 \end{aligned} \tag{5.14}$$

Z uporabo enačb 5.9 izračunamo

$$\frac{\partial f_1}{\partial x_1} = 0 \qquad \frac{\partial f_1}{\partial x_2} = -1 \tag{5.15}$$

$$\frac{\partial f_2}{\partial x_1} = 1 \qquad \frac{\partial f_2}{\partial x_2} = 3x_2 - 3\Big|_{x_2=0} = -3 \tag{5.16}$$

in s tem elemente sistemske matrike linearnega modela:

$$\boldsymbol{A} = \begin{bmatrix} 0 & -1 \\ 1 & -3 \end{bmatrix} \tag{5.17}$$

S pomočjo Matlaba uporabimo Simulink model `sysmdl_e.mdl` na sliki 5.20 in program za linearizacijo okoli delovne točke nič

```
[A,B,C,D]=linmod('sysmdl_e');
disp(A);
disp(eig(A));
```

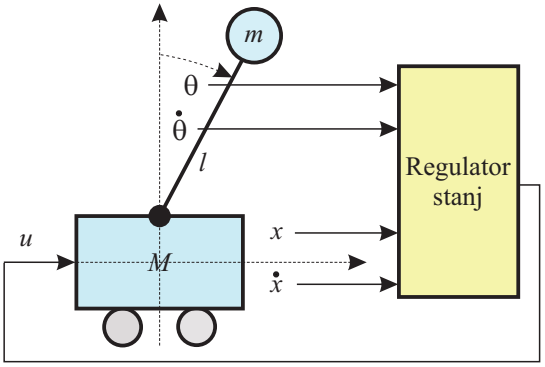
```
     0         -1.0000
     1.0000    -3.0000

   -0.3820
   -2.6180
```

Program izpiše matriko $\boldsymbol{A}$ in obe lastni vrednosti. Prepričali smo se, da je koordinatno izhodišče stabilna ravnotežna točka, saj imata obe lastni vrednosti negativna realna dela (glej tudi fazne portrete na slikah 5.21 in 5.22).

□

Primer 5.5 Načrtovanje linearnega regulatorja s pomočjo linearizacije

V tem primeru bomo prikazali načrtovanje linearnega regulatorja za nelinearni proces - za voziček z invertiranim nihalom. Regulator stanj mora držati pal-



Slika 5.25: Voziček z invertiranim nihalom

ico v navpični legi s pomočjo sile u, ki deluje na voziček. Praktične rešitve tega problema temeljijo na algoritmu, ki dvigne palico v približno navpično lego. Nato pa prevzame funkcijo regulator stanj, ki zagotavlja kot $\theta \approx 0$ tudi ob morebitnih motnjah. Ker je vloga primera prikaz linearizacije, se ne bomo spuščali v modeliranje in načrtovanje vodenja.

Vhod v sistem je torej horizontalna sila u ki deluje na voziček z maso M. Nihalo se prosto vrti na ležaju brez trenja. Dolžina nihala je l, masa nihala m pa je zbrana v eni točki na konci nihala. Parametri sistema so naslednji:

$$M = 2kg \tag{5.18}$$
$$m = 0.5kg \tag{5.19}$$
$$l = 0.5m \tag{5.20}$$
$$g = 9.8ms^{-2} \tag{5.21}$$

Z modeliranjem pridemo do diferencialnih enačb

$$(M+m)\ddot{x} - ml\dot{\theta}^2 + ml\ddot{\theta}cos\theta = u \tag{5.22}$$
$$m\ddot{x}cos\theta + ml\ddot{\theta} = mgsin\theta \tag{5.23}$$

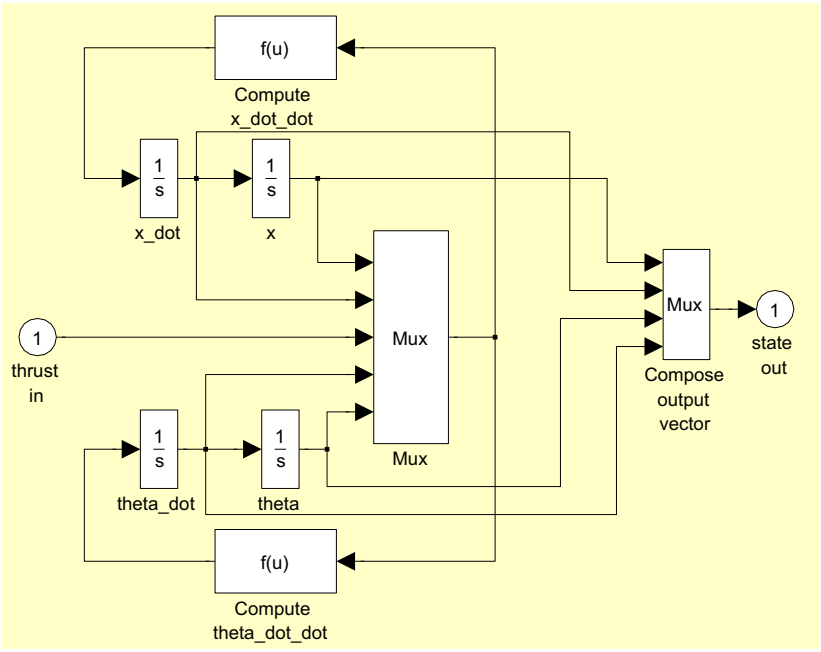
Problem je tudi simulacijsko zahtevnejši, ker $\ddot{x}$ in $\ddot{\theta}$ nastopata v obeh enačbah. Običajni postopek z indirektno metodo je sicer možno uporabiti, vendar se pokaže, da ima model algebrajsko zanko. Z nekaj algebre lahko algebrajsko zanko

odpravimo in pridemo do modela

$$\ddot{x} = \frac{ml\dot{\theta}^2 sin\theta - mgsin\theta cos\theta + u}{M + msin^2\theta} \tag{5.24}$$

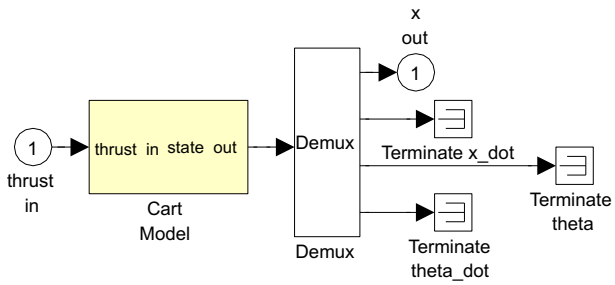
$$\ddot{\theta} = \frac{-(ml\dot{\theta}^2 sin\theta cos\theta - mgsin\theta + ucos\theta - Mgsin\theta)}{l(M + msin^2\theta)} \tag{5.25}$$

Naslednja naloga je zgraditi model v okolju Simulink z vhodnim signalom u in stanji $\boldsymbol{x}$ na izhodu. Izberemo spremenljivke stanja x, $\dot{x}$, θ, $\dot{\theta}$. Regulator stanj uporabe vse štiri spremenljivke za izračun vhodnega signala u. Celoten model je izveden s podmodelom s skalarnim vhodnim signalom (`Inport` blok za u) in vektorskim izhodnim signalom (`Outport` blok z vsemi stanji). Podmodel prikazuje slika 5.26, glavni podmodel, ki se nahaja na datoteki `sysmdl_h.mdl` pa slika 5.27.



Slika 5.26: Podsistem invertiranega nihala

Nelinearne enačbe 5.24 in 5.25 so realizirane z dvema `Fcn` blokoma (glej sliko 5.26). Glavni model pa je na izhodu pripravljen za priključitev na regulator stanj. Zato vodimo stanje na demultiplekser (`Demux`), izhod pa so 4 skalarne veličine, ki jih zaključimo z enim `Outport` blokom (za pomik vozička x, ki je izhod sistema) in tremi `Terminator` bloki (za ostale tri spremenljivke - s tem le preprečimo opozorila, da signali niso nikamor povezani, dokler na njih ne vključimo regulatorja).



Slika 5.27: Celoten model za linearizacijo

Nato izvedemo naslednji Matlabov program (`ch8_c.m`):

```
% Initialize the system parameters and obtain the
% model characteristics
%
% System parameters
l = 0.5 ;   % Pendulum length
g = 9.8 ;   % Gravity
m = 0.1 ;   % Pendulum bob mass
M = 2 ;     % Cart mass

% Get the model characteristics
[sizes,x0,states] = sysmdl_h
```

Matlab vrne naslednje podatke o modelu, ki je pripravljen za linearizacijo.

```
sizes =
     4
     0
     1
     1
     0
     1
     1

x0 =
```

```
     0
     0
     0
     0

states =
    'sysmdl_h/Cart
Model/x'
    'sysmdl_h/Cart
Model/x_dot'
    'sysmdl_h/Cart
Model/theta_dot'
    'sysmdl_h/Cart
Model/theta'
```

Pomembna informacija, ki bi jo kaj lahko spregledali, je naslednja. Simulink je določil vektor stanj

$$\boldsymbol{x} = \begin{bmatrix} x \\ \dot{x} \\ \dot{\theta} \\ \theta \end{bmatrix}$$

torej v drugačnem zaporedju, kot smo ga zgradili na multiplekserju.

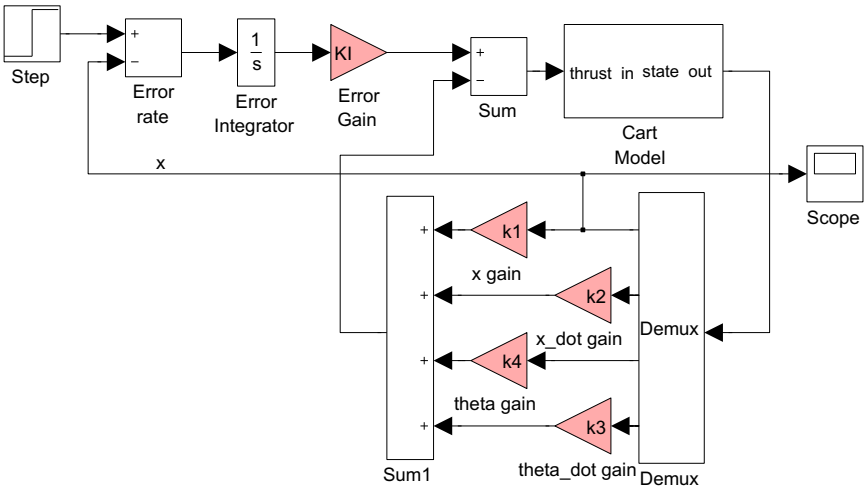
Nato zgradimo celoten model regulacijskega sistema, kjer regulator stanj s štirimi ojačenji množi štiri spremenljivke stanja in produkte sešteje. Na tem mestu je pomembno, da ojačenje k_1 pomnoži signal x, ojačenje k_2 signal $\dot{x}$, ojačenje k_3 signal $\dot{\theta}$ in ojačenje k_4 signal θ.

Celotno regulacijsko shemo prikazuje slika 5.28.

Glavna želena vrednost je pomik vozička, ki se izvede s `step` funkcijo (ki nastopi v trenutku $t = 1s$). Voziček se mora hitro postaviti v novo lego in medtem obdržati palico v pokončnem položaju.

Matlabov program za linearizacijo in izračun regulatorja stanj je naslednji (datoteka `ch8_d.m`)

```
% Design the feedback gain matrix for the cart with inverted
% pendulum example.
```



Slika 5.28: Regulacija invertiranega nihala

```
%
% System parameters
l = 0.5 ;   % Pendulum length
g = 9.8 ;   % Gravity
m = 0.1 ;   % Pendulum bob mass
M = 2 ;     % Cart mass

% Get the system matrices
[A,B,C,D] = linmod('sysmdl_h',[0,0,0,0],[0])

A1 = [A zeros(4,1) ; -C 0] ;
B1 = [B ; 0] ;

% Define the controllability matrix M
MM = [B1 A1*B1 A1^2*B1 A1^3*B1 A1^4*B1] ;

% Check the rank of MM, must be 5 if system in completely controllable
rank(MM)

% Obtain the characteristic polynomial
J = [-1+sqrt(3)*i,0,0,0,0;0,-1-sqrt(3)*i,0,0,0;...
     0,0,-5,0,0;0,0,0,-5,0;0,0,0,0,-5] ;
Phi = polyvalm(poly(J),A1) ;

% Get the feedback matrix using Ackermann's formula
```

```
KK = [0,0,0,0,1]*(inv(MM))*Phi
k1 = KK(1)  ; % x gain
k2 = KK(2)  ; % x_dot gain
k3 = KK(3)  ; % theta_dot gain
k4 = KK(4)  ; % theta gain
KI = -KK(5) ; % error integrator gain
```

Program uporabi funkcijo linmod za določitev lineariziranega modela, nato pa se uporabi Ackermanova formula za izračun regulatorja stanj (Zupančič, 2010c). Izpišejo se naslednji rezultati:

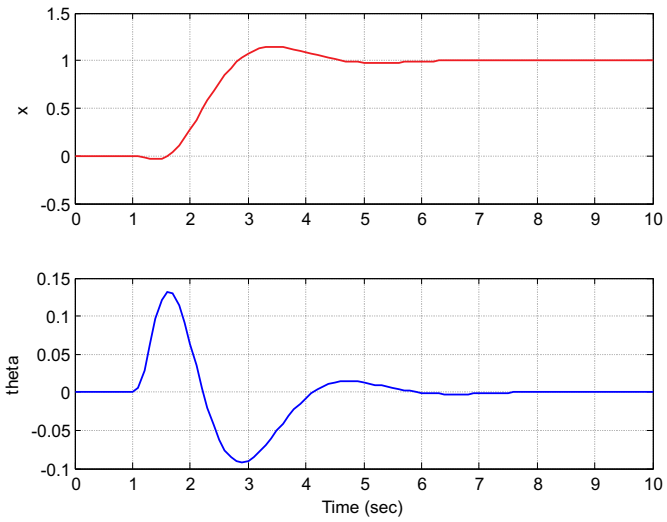
```
A =
         0    1.0000         0         0
         0         0         0   -0.4900
         0         0         0   20.5800
         0         0    1.0000         0

B =
         0
    0.5000
   -1.0000
         0

C =
     1     0     0     0

D =
     0

ans =
     5

KK =
  -56.1224  -36.7868  -35.3934 -157.6412   51.0204
```

Z naslednjim programom pa preizkusimo delovanje regulacijskega sistema

```
% Run the inverted pendulum simulation
```

```
opts = simset('InitialState',[0,0,0,0,0],'Solver','ode45') ;
[t,x] = sim('sysmdl_i',[0:0.1:10],opts) ;

% Plot the position and pendulum angle
subplot(2,1,1);
plot(t,x(:,1));
grid;
ylabel('x');
subplot(2,1,2);
plot(t,x(:,4));
grid;
xlabel('Time (sec)') ;
ylabel('theta');
```

Slika 5.29 prikazuje pomik vozička x proti želeni legi in potek kota θ, ki v prehodnem pojavu zaniha okoli ničelne lege, nato pa se umiri.



Slika 5.29: Rezultati regulacije

□

5.5.4 Določevanje ravnotežnih točk (ustaljenih vrednosti)

Pri linearizaciji smo že vpeljali pojem delovne oz. ravnotežne točke. Le to lahko določimo tako, da sisteme simuliramo pri konstantnih vhodnih signalih do ustal-

jenega stanja in odčitamo ustrezne vrednosti stanj in izhodov. Lahko pa določimo ravnotežno stanje tudi z reševanjem algebrajske enačbe 5.13

$$\dot{\boldsymbol{x}}_0 = \mathbf{0} = \boldsymbol{f}(t_0, \boldsymbol{x}_0, \boldsymbol{u}_0) \tag{5.26}$$

Dve pomembni uporabnosti določitve ravnotežnih točk sta naslednji:

- Pri analizi nelinearnih sistemov določimo ravnotežne točke in nato linearizirамo sistem okoli vsake točke posebej. Z določitvijo lastnih vrednosti lineariziranih modelov lahko sklepamo na stabilnostni karakter ravnotežnih točk.
- Pri regulacijskih sistemih nas pogosto zanima delovanje v ustaljenem stanju (npr. ustaljene vrednosti pogreška, regulirne in regulirane veličine). Na nekaterih področjih (zlasti v kemičnih procesih) se pod besedo simulacija predvsem računajo ustaljena stanja in se tudi simulacija v našem pomenu v takih orodjih ne izvaja.

Funkcija `trim`

Matlabov ukaz `trim` deluje nad Simulink modelom in določi ravnotežno točko z reševanjem enačbe 5.13. Možno je določiti tudi 'kvazi ravnotežne točke', ko se zahteva, da so le nekateri odvodi stanj enaki nič.

`trim` funkcija poišče ravnotežno točko z iskanjem takih vrednosti x, u in y, da algoritem doseže minimum vektorja odvodov (minimizira se največja absolutna vrednost v odvodnem vektorju). Vektorje x, u in y ali samo posamezne komponente lahko tudi fiksiramo - to pomeni, da mora optimizacijska metoda upoštevati omejitve. V tem primeru optimizacijska metoda minimizira odvodni vektor in omejitveno napako (da je razlika med fiksiranimi in dejanskimi končnimi vrednostmi čim manjša). Pri izbiri omejitev moramo biti pazljivi, tako da navedemo le najnujnejše. Če je npr. neka spremenljivka hkrati stanje in izhod, potem deklariramo omejitev le na enem mestu.

Sintaksa ukaza, ki ne vključuje vseh parametrov, je naslednja:

```
[x,u,y,dyx]=trim(model,x0,u0,y0,ix,iu,iy)
```

Vsi parametri razen imena modela so opcijski. Pomen je naslednji:

`x`	Stanje v ravnotežni točki. Tudi v primeru, če ne navedemo izhodnih parametrov, funkcija vrne `x`.
`u`	Vrednost vektorja vhodov v ravnotežni točki. Če sistem nima `inport` blokov, funkcija vrne prazen vektor.
`y`	Vrednost vektorja izhodov v ravnotežni točki. Če sistem nima `outport` blokov, funkcija vrne prazen vektor.
`dx`	Odvod stanj v ravnotežni točki.
`model`	Ime Simulink modela z navednicami in brez podaljška, npr. `'lisice'`.
`x0`	Začetna ocena vrednosti stanj v ravnotežni točki. Algoritem prične z optimiranjem s to začetno vrednostjo. Posamezni elementi so lahko fiksirani z vrednostjo v `x0` z uporabo parametra `ix`.
`u0`	Začetna ocena vrednosti vhodov v ravnotežni točki. Posamezni elementi so lahko fiksirani z vrednostjo v `u0` z uporabo parametra `iu`.
`y0`	Začetna ocena vrednosti izhodov v ravnotežni točki. Posamezni elementi so lahko fiksirani z vrednostjo v `y0` z uporabo parametra `iy`.
`ix`	S tem vektorjem določimo fiksirane vrednosti stanj: če npr. želimo fiksirati drugo in četrto stanje z vrednostjo, ki je določena z `x0`, navedemo `[2,4]`.
`iu`	S tem vektorjem določimo fiksirane vrednosti vhodov: če npr. želimo fiksirati drugi in četrti izhod z vrednostjo, ki je določena z `u0`, navedemo `[2,4]`.
`iy`	S tem vektorjem določimo fiksirane vrednosti izhodov: če npr. želimo fiksirati drugi in četrti izhod z vrednostjo, ki je določena z `y0`, navedemo `[2,4]`.

Navedli smo sedem vhodnih parametrov od enajst možnih. V primeru, če sta funkciji $\boldsymbol{f}$ in $\boldsymbol{g}$ časovno odvisni, moramo z enajstim parametrom podati še čas, v katerem računamo ravnotežno točko.

Ni nikakršnega zagotovila, da funkcija trim najde ravnotežno točko okoli predpisane začetne vrednosti. Tudi ni zagotovila, da iskalni postopek konvergira, čeprav ravnotežna točka obstaja. Pogosto pomaga, če spremenimo začetno točko.

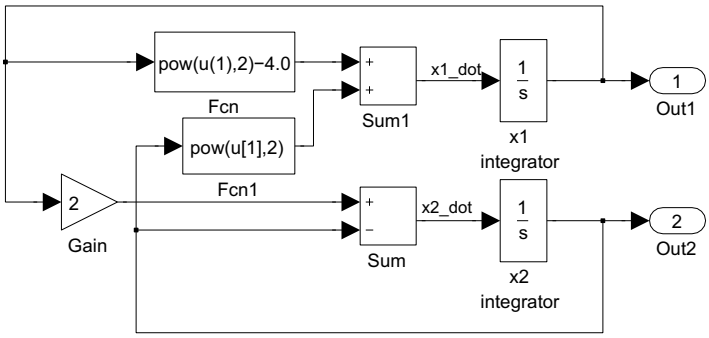
Primer 5.6 Določitev ravnotežnih točk nelinearnega sistema

Nelinearni dinamični sistem 2. reda opisujeta diferencialni enačbi

$$\dot{x}_1 = x_1^2 + x_2^2 - 4 \tag{5.27}$$

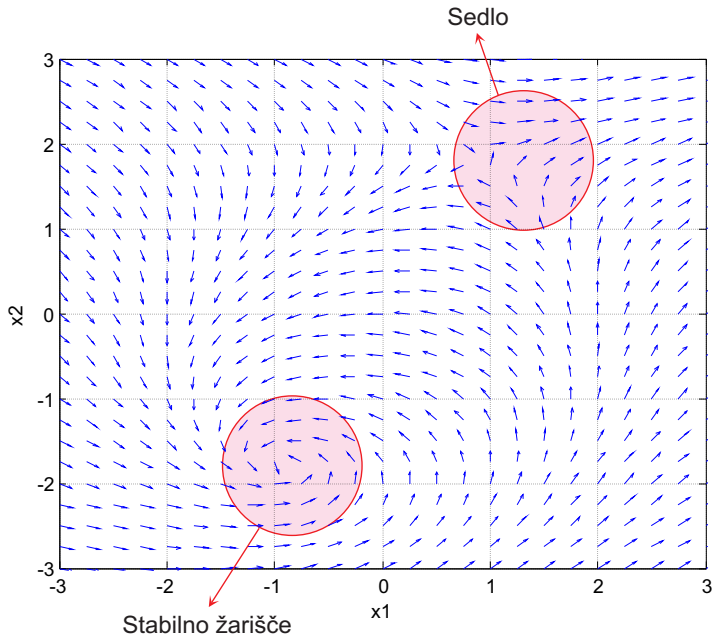
$$\dot{x}_2 \quad = \quad 2x_1 - x_2 \tag{5.28}$$

Želimo določiti vse ravnotežne točke in stabilnostni karakter vsake točke. Ustrezni Simulink model z imenom `sysmdl_1.mdl` prikazuje slika 5.30.



Slika 5.30: Simulink shema nelinearnega sistema

Slika 5.31 prikazuje Quiverjev fazni portret, ki smo ga narisali s podobnim programom, kot je prikazan v primeru 5.3 (datoteka `ch8_f.m`).



Slika 5.31: Quiverjev fazni portret nelinearnega sistema

Iz slike ugotovimo, da obstajata dve ravnotežni točki, ki se nahajata v bližini točk [-1, -2] in [1, 2]. Slika tudi kaže, da je delovanje v bližini prve točke stabilno, v bližini druge točke pa nestabilno.

S pomočjo Matlaba in Simulinka določimo ravnotežni točki s funkcijo `trim`. Nato pa s funkcijo `linmod` linearizirамо sistem v vsaki točki. Končno določimo še lastne vrednosti obeh linearizirанih modelov s funkcijo `eig`. Program v Matlabu je naslednji (`ch8_g.m`):

```
% Locate the two equilibrium points for the system
% sysmdl_1. Linearize the system about each equilibrium,
% then compute the eigenvalues at each equilibrium.
xa = trim('sysmdl_1',[-1;-2]) ;  % Locate the equilibrium
[A,B,C,D] = linmod('sysmdl_1',xa) ;
eigv_a = eig(A) ;
fprintf('Eigenvalues at (%f,%f):\n',xa) ;
eigv_a
xb = trim('sysmdl_1',[1;2]) ;  % Locate the other equilibrium
[A,B,C,D] = linmod('sysmdl_1',xb) ;
eigv_b = eig(A) ;
fprintf('Eigenvalues at (%f,%f):\n',xb) ;
eigv_b
```

Zgornji program izpiše naslednje rezultate:

```
Eigenvalues at (-0.894427,-1.788854):

eigv_a =
  -1.3944 + 2.6457i
  -1.3944 - 2.6457i

Eigenvalues at (0.894427,1.788854):

eigv_b =
    3.4110
   -2.6222
```

Vidimo, da je ravnotežna točka v bližini začetno ocenjene točke pri [-1, -2] natančno pri [-0.89, -1.79], ravnotežna točka v bližini začetno ocenjene točke pri [1, 2] pa natančno pri [0.89, 1.79]. V prvem primeru sta obe lastni vrednosti z negativnima realnima deloma, torej je sistem v okolici prve ravnotežne točke stabilen. Taki ravnotežni točki rečemo tudi stabilno žarišče. V drugem primeru

pa sta obe lastni vrednosti realni, vendar ima ena pozitivno vrednost, torej je sistem v okolici druge ravnotežne točke nestabilen. Ravnotežno točko imenujemo sedlo.

□

5.5.5 Optimiranje sistemov

Optimiranje je ključnega pomena pri načrtovanju najrazličnejših sistemov. Na tem mestu si bomo ogledali, kako uspešno povežemo optimizacijska in simulacijska okolja. Koncept, ki ga bomo prikazali, je enako uporaben za optimiranje pri modeliranju sistemov kot tudi pri načrtovanju vodenja.

Parametrska optimizacija nekega sistema je postopek, s pomočjo katerega določimo njegove parametre tako, da ob tem minimiziramo ali maksimiziramo (odvisno od situacije) določeno cenilko (kriterijsko funkcijo, performančni indeks) v nekem prehodnem pojavu (npr. v regulacijskem sistemu spremenjena referenčna veličina ali motnja sproži prehodni pojav). Cenilka je odvisna od signala pogreška med želenimi in dejanskimi signali in je pogosto integralskega tipa. Razen cenilke lahko v optimizacijske postopke vključimo tudi omejitve. S tem upoštevamo nekatere realne lastnosti dinamičnih sistemov, ki jih sicer pri teoretični obravnavi radi zanemarimo (na primer fizične omejitve izvršnih sistemov, merilnih sistemov in regulatorjev 4-20ma).

Samo najenostavnejše probleme, ki so ponavadi zgolj teoretičnega pomena, je možno rešiti analitično. V tem primeru cenilko, ki jo določimo v nekem prehodnem pojavu (npr. $C = \int e^2(t)dt$), zapišemo kot funkcijo parametrov

$$C = f(p_1, p_2, p_3, ...) \tag{5.29}$$

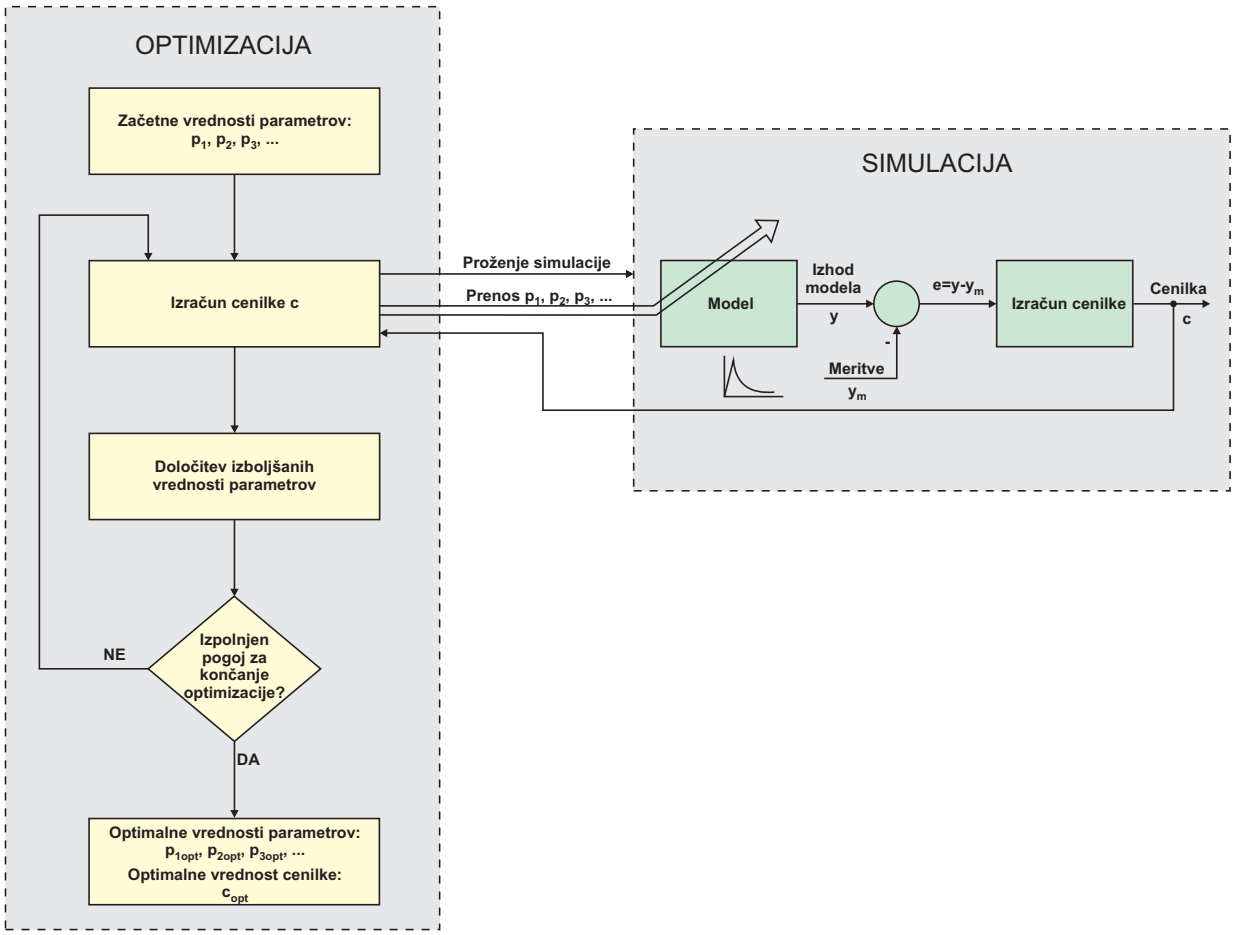
Če želimo določiti optimalno vrednost parametra p_1, je potrebno cenilko odvajati po tem parametru in odvod izenačiti z nič

$$\frac{\partial C}{\partial p_1} = 0 \quad \Longrightarrow \quad p_{1opt} \tag{5.30}$$

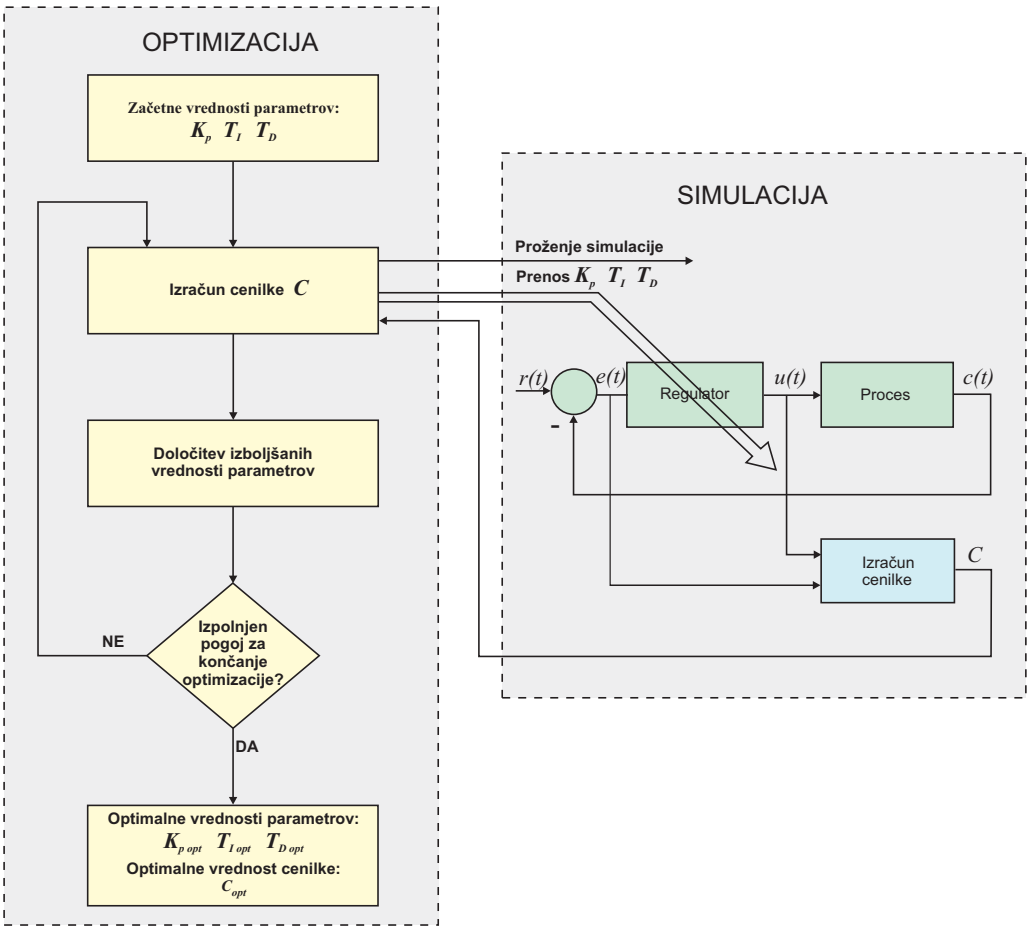
Za kompleksnejše realne probleme pa uporabljamo računalniško parametrsko optimizacijo. To je poseben algoritem, ki pri ponavljajočih prehodnih pojavih toliko časa spreminja parametre modela (regulatorja), da na koncu zagotovi optimum

cenilke. Glavna slabost postopka je velika potratnost računalniškega časa, saj vsaka nova kombinacija parametrov regulatorja zahteva nov simulacijski tek oz. nov prehodni pojav, v katerem se ovrednoti vrednost cenilke. Ker je pri reševanju kompleksnejših problemov potrebno izvesti tudi nekaj sto poizkusov - simulacijskih tekov, je razumljivo, kako neobhoden je sposoben računalnik. Glavna prednost takega pristopa pa je, da je uporaben tudi za kompleksne nelinearne sisteme.

Koncept računalniške optimizacije, ki uporablja simulacijo, prikazujeta sliki 5.32 in 5.33. Prva slika kaže postopek pri modeliranju, druga pa pri načrtovanju vodenja. Tak koncept je zlasti možno učinkovito uporabiti v okolju Matlab-Simulink. Optimizacijo izvaja okolje Optimization toolbox (npr. funkcija `fminsearch`), simulacijo pa okolje Simulink.



Slika 5.32: Optimizacija pri modeliranju

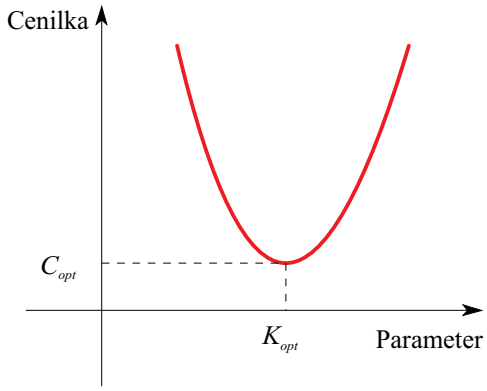


Slika 5.33: Optimizacija pri načrtovanju vodenja

Cenilke

Cenilka je število, ki pove, kako dobro deluje model oz. regulacijski sistem v nekem prehodnem pojavu. V splošnem lahko tudi že znani pokazatelji (maksimalni prevzpon, čas vzpona in čas umiritve) nastopajo v cenilki, vendar v zvezi z računalniško optimizacijo največkrat uporabljamo integralske cenilke. Za cenilko je izjemno pomembno, da kaže določeno selektivnost, kar pomeni, da ima pri optimalnih vrednostih parametrov jasno izražen optimum. Ustrezno obliko v odvisnosti od enega parametra prikazuje slika 5.34.

Literatura priporoča naslednje integralske cenilke, slednja je namenjena predvsem



Slika 5.34: Vrednost cenilke v odvisnosti od parametra

optimizaciji regulacijskih sistemov:

$$\begin{aligned}
C_1 = ISE &= \int_0^\infty e^2(t)dt && \text{integral square error} \\
C_2 = ITSE &= \int_0^\infty t\, e^2(t)dt && \text{integral time square error} \\
C_3 = IAE &= \int_0^\infty |e(t)|dt && \text{integral absolute error} \\
C_4 = ITAE &= \int_0^\infty t|e(t)|dt && \text{integral time absolute error} \\
C_5 = S_{eu} &= \int_0^\infty [e^2(t) + R(u(t) - u(\infty))^2]dt && (5.31)
\end{aligned}$$

V regulacijskih sistemih je pogrešek $e(t)$ razlika med referenčno in regulirano veličino, $u(t)$ je regulirna veličina v primeru načrtovanja sistemov vodenja. V primeru modeliranja pa je pogrešek običajno povezan z razliko med merjenimi signali in pripadajočimi spremenljivkami modela. Izbira ustrezne cenilke je ključna pri optimiranju regulacijskega sistema. Neskončni integracijski interval pa zamenjamo s tako končno dolžino, da prehodni pojav izzveni in da ustaljeni pogrešek postane nič ali zelo majhen. Ker je težko ugotoviti, katera cenilka je primernejša, lahko preizkusimo več cenilk in izberemo najprimernejšo. Kvadriranje v cenilki ali računanje absolutne vrednosti se uporablja zlasti pri podkritično dušenih prehodnih pojavih, ko so pogreški pozitivni in negativni. Množenje s časom se uporablja, v kolikor želimo bolj utežiti del po prehodnem pojavu, t.j. ustaljeno stanje. V enačbi 5.31 je R utežni faktor regulirne veličine, $u(\infty)$ vrednost regulirne veličine v ustaljenem stanju. Ker v cenilki upoštevamo tudi regulirno veličino, s tem zmanjšamo porabo energije, ki jo je potrebno v določenem prehodnem pojavu dovajati procesu. Zato postane regulacijski sistem nekoliko počasnejši oz. bolj

dušen v prehodnem pojavu. Na regulirni veličini se to odraža tako, da so njene vrednosti v trenutkih hitrih sprememb reference ali nastopa motnje tem manjše, čim večji je utežni faktor R.

Omejitve

Razen cenilke lahko v postopkih računalniške optimizacije vpeljemo tudi omejitve. Tipična omejitev je omejitev regulirne veličine zaradi končnega območja izvršnega sistema. Tudi maksimalni dopustni prevzpon lahko vključimo kot omejitev. Optimizacijski algoritem mora izračunati take optimalne vrednosti parametrov, da niso kršeni omejitveni pogoji.

Pogoj za končanje optimizacije

Računalniški optimizacijski algoritmi omogočajo več različnih pogojev za končanje optimizacije. Ker optimizacija poteka v iteracijah (ena iteracija je del algoritma, ki določi izboljšane vrednosti parametrov), je možno podati število iteracij. Nadalje je možno optimizacijo zaključiti, če je med dvema iteracijama sprememba cenilke manjša od predpisanega praga ali če je normirana vrednost spremembe parametrov pod predpisanim pragom.

Optimiranje v okolju Matlab-Simulink

Okolje Matlab je z nekaterimi drugimi dodatki - predvsem Optimization Toolbox zelo učinkovito pri optimiranju dinamičnih sistemov s pomočjo modelov v okolju Simulink. Optimization Toolbox vsebuje številne optimizacijske postopke, tudi take, kjer je možno vključiti omejitve. Uporabili bomo koncept, ki ga prikazujeta sliki 5.32 in 5.33.

Za optimizacijo bomo uporabili funkcijo `fminsearch`, ki omogoča večdimenzionalno nelinearno optimizacijo (Nelder-Mead). Sintaksa ukaza v Matlabu je naslednja:

```
x=fminsearch(fun,x0,options)
```

Pomen parametrov je naslednji:

`fun`	Ime Matlabove funkcije, ki vsebuje cenilko. V našem konceptu mora torej ta funkcija sprožiti simulacijo Simulink modela (s funkcijo `sim`), saj se cenilka izračuna s pomočjo simulacije.
`x0`	Začetne vrednosti parametrov, ki jih optimiramo.
`options`	Vektor za krmiljenje optimizacije. Nastavi se ga s funkcijo `optimstep`.
`x`	Rezultat optimizacije - vektor optimalnih parametrov.

Optimizacijska funkcija mora torej poiskati vektor $\boldsymbol{x}$, ki minimizira cenilko (glej sliko 5.34). Cenilka mora biti napisana kot M funkcijska datoteka v obliki

```
function f=fun(x)
```

V opisanem optimizacijskem konceptu ima funkcija `fun` naslednjo zgradbo:

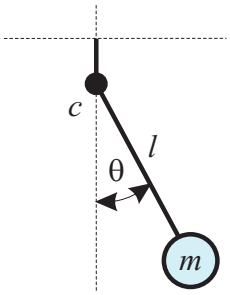
1. Inicializira parametre Simulink modela s trenutnimi optimalnimi vrednostmi `x`.
2. Proži simulacijo s funkcijo `sim`. Po potrebi shranjuje stanja, izhode, ... za kasnejši izračun cenilke.
3. Izračuna cenilko - skalarno vrednost `f`. Cenilko se dostikrat računa kot integral nekih spremenljivk, lahko pa se izračuna na osnovi kakšnih vrednosti med simulacijo ali na koncu simulacije.
4. Znotraj funkcije je treba pred proženjem simulacije izvršiti ukaz `opts=simset('SrcWorkspace','current','DstWorkspace','current')` in s tem ustrezno nastaviti krmilni vektor za simulacijo.

Nekaj napotkov za optimizacijo:

- Koristno je podati čim boljše začetne vrednosti parametrov, ki že omogočajo neko spodobno, vsaj stabilno delovanje.
- Nikoli ni garancije, da nas je optimizacija pripeljala v globalni optimum. Lahko je dosegla lokalni optimum. Praktična rešitev tega problema je lahko optimizacija iz različnih začetnih vrednosti.
- Nekaj pozornosti je treba posvetiti pogoju za končanje optimizacije. V kolikor optimizacija ne potrebuje preveč računalniškega časa, je najenostavneje predpisati število iteracij.

Primer 5.7 Optimizacija dušenja nihala

Dušeno nihalo prikazuje slika 5.35.



Slika 5.35: Dušeno nihalo

Z optimizacijo določimo dušenje c, ki minimizira cenilko

$$f = \int_0^{10} (\theta^2 + \dot{\theta}^2) dt$$

Torej optimiramo proces umirjanja nihala. Če je dušenje preveliko, se nihalo prepočasi vrača v mirovno lego. Če je dušenje premajhno, pa lahko dolgo niha okoli ravnovesne lege. V tem primeru je torej pogrešek povezan kar s kotom in kotno hitrostjo, ki na začetku odstopata od končnih ravnovesnih vrednosti nič. Začetni odmik nihala je $\pi/3$, $m = 1kg$, $l = 1m$. Matematično modeliranje nas pripelje do diferencialne enačbe

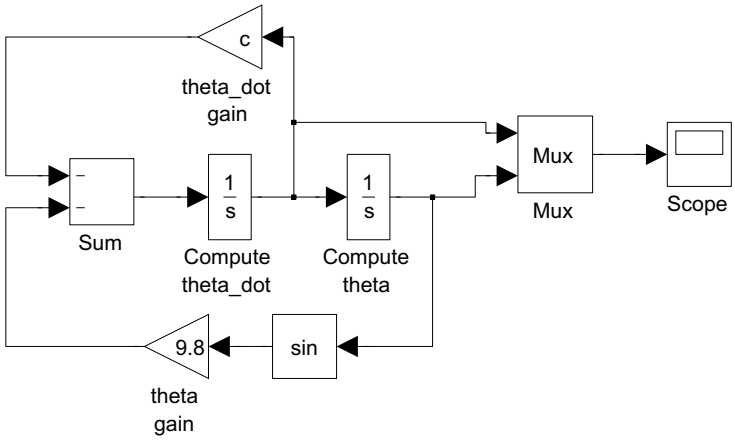
$$\ddot{\theta} + \frac{c}{ml^2}\dot{\theta} + \frac{g}{l} sin\theta = 0 \tag{5.32}$$

Simulink shemo prikazuje slika 5.36 (`model sysmdl_m.mdl`).

Glavni program v Matlabu za optimizacijo (datoteka `optimise.m`):

```
options=optimset('MaxIter',10,'display','iter');
k=5;
c=fminsearch('pdn_prf',k,options)
c
```

`k` je začetna vrednost dušenja, `c` pa končna optimalna. `pdnprf` je ime funkcije, ki vsebuje izračun cenilke, program pa nastavi še krmilne parametre z vektorjem `options`, ki ga nastavi funkcija `optimset`: nastavili smo optimizacijo z 10 iteracijami in zahtevali izpis po vsaki iteraciji.



Slika 5.36: Model dušenega nihala v Simulinku

Cenilka (datoteka `pdnprf.m`):

```
function f = pdn_prf(c)
% System parameters used by SIMULINK model
t0 = 0 ;    % Start time
tf = 10 ;   % End time
h = .1 ;    % Time step size
opts=simset('SrcWorkspace','current') ;   % Set to use current workspace
opts=simset(opts,'DstWorkspace','current') ;
[t,x] = sim('sysmdl_m',[t0:h:tf],opts) ;  % Get the trajectory
plot(t,x);
hold on;
[nt,nx] = size(x) ;
jp = zeros(1,nt) ; % Preallocate performance integrand
for i = 1:nt
  jp(i) = x(i,:)*x(i,:)' ; % Integrand of objective function
end
f = h*(sum(jp)-(jp(1)+jp(nt))/2) ;  % Trapezoidal integration
```

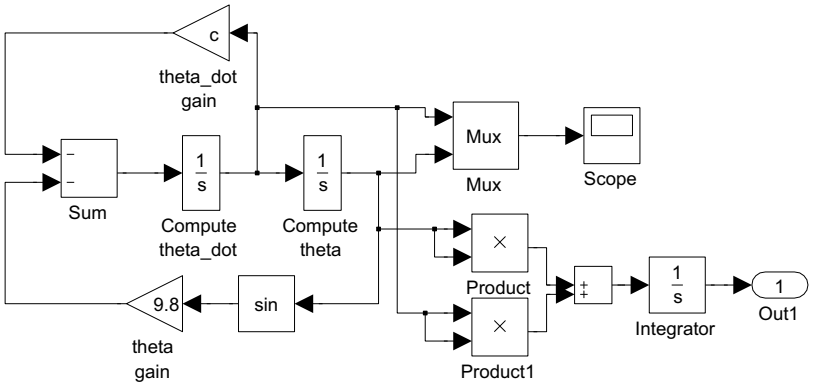
Drugi del programa je namenjen izračunu cenilke s pomočjo trapezne integracije.

Optimizacija izračuna optimalno dušenje $c = 9.5$. □

Primer 5.8 Enak problem, cenilka pa je pretežno določena v Simulink shemi

Enak problem smo optimirali tudi tako, da smo integralsko cenilko določili kar v Simulinku, kjer je integriranje enostavnejše. Cenilko sproti računamo med simulacijo, na koncu simulacijskega teka ima ustrezno vrednost, ki se prenese v funkcijo cenilke `pdn_prf_new`.

Simulink model (datoteka `sysmdl_m_new.mdl`) prikazuje slika 5.37.



Slika 5.37: Model dušenega nihala v Simulinku, ki vključuje tudi izračun cenilke

Glavni program v Matlabu za optimizacijo (datoteka `optimise_new.m`):

```
options=optimset('MaxIter',10,'display','iter');
k=5;
c=fminsearch('pdn_prf_new',k,options);
c
```

Cenilka (datoteka `pdnprf_new.m`):

```
function f = pdn_prf_new(c)
% System parameters used by SIMULINK model
t0 = 0 ;    % Start time
tf = 10;   % End time
opts=simset('SrcWorkspace','current') ;   % Set to use current workspace
opts=simset(opts,'DstWorkspace','current') ;
[t,x,y] = sim('sysmdl_m_new',[t0,tf],opts) ;  % Get the trajectory
plot(t,x,t,y);
hold on;
[nt,nx]=size(x);
f=y(nt);
```

Matlabov glavni optimizacijski program izpiše naslednje vrstice:

```
Iteration   Func-count     min f(x)         Procedure
     0            1          1.41002
     1            2          1.37337         initial simplex
     2            4          1.31355         expand
     3            6          1.23389         expand
     4            8          1.17081         expand
     5           10          1.16777         contract outside
     6           12          1.16756         contract inside
     7           14          1.16726         contract inside
     8           16          1.16726         contract inside
     9           18          1.16726         contract inside
    10           20          1.16726         contract inside

Exiting: Maximum number of iterations has been exceeded
         - increase MaxIter option.
         Current function value: 1.167256

c =
    9.4688
```

Optimizacija je naredila 10 iteracij in skupaj 20 simulacijskih tekov. Cenilka se je iz začetne vrednosti 1.41 znižala na optimalno vrednost 1.167. Dušenje pa se je iz iz začetne vrednosti 5 povečalo na 9.47.

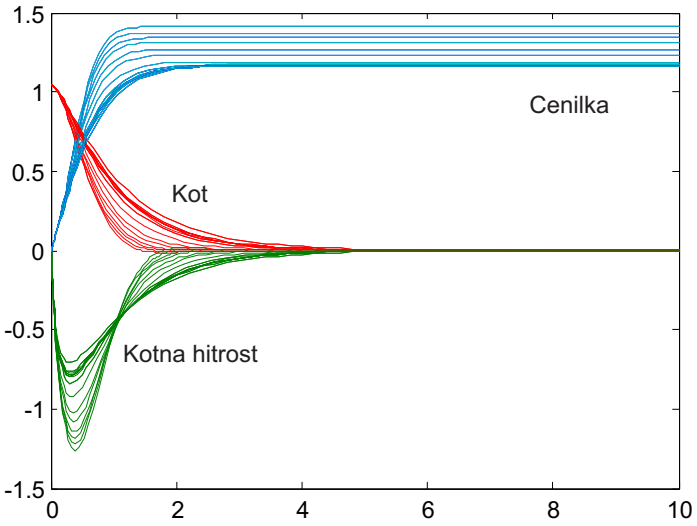
Slika 5.38 prikazuje rezultate v grafični obliki - cenilko f, kot θ in kotno hitrost $\dot{\theta}$. Rezultati so vrisani za vseh 20 simulacijskih tekov. Ker je po desetih simulacijah oz. po petih iteracijah postopek optimizacije že praktično zaključen, so vidne odebeljene krivulje, ki predstavljajo optimalne poteke.

□

Primer 5.9 Optimizacija regulacijskega sistema

Slika 5.39 prikazuje regulacijski sistem. Potrebno je načrtati P regulator

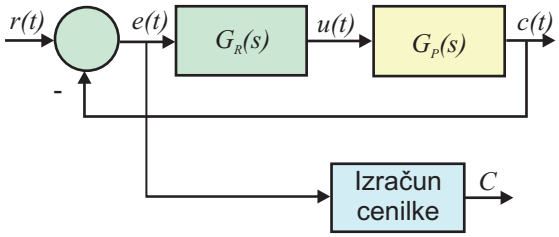
$$G_R(s) = k_P \tag{5.33}$$



Slika 5.38: Rezultati simulacije

za integrirni proces

$$G_P(s) = \frac{\omega_n^2}{s(s^2 + 2\zeta\omega_n s + \omega_n^2)} \qquad \omega_n = 1,\ \zeta = 0.7 \tag{5.34}$$



Slika 5.39: Regulacijski sistem

Ker je kritično ojačenje regulacijskega sistema $K_{KR} = 1.4$, je po nastavitvenem Ziegler-Nichols-ovem pravilu ojačenje proporcionalnega regulatorja $K_P = \frac{K_{KR}}{2} = 0.7$. Vendar je potek regulirane veličine pri stopničasti spremembi reference zelo nihajoč, prevzpon pa je kar 40% (slika 5.42, krivulja a). Zato smo uporabili računalniško optimizacijo.

Prva izbrana cenilka je bila

$$C_1 = \text{ITAE} = \int_0^{30} t\,|e(t)|dt \tag{5.35}$$

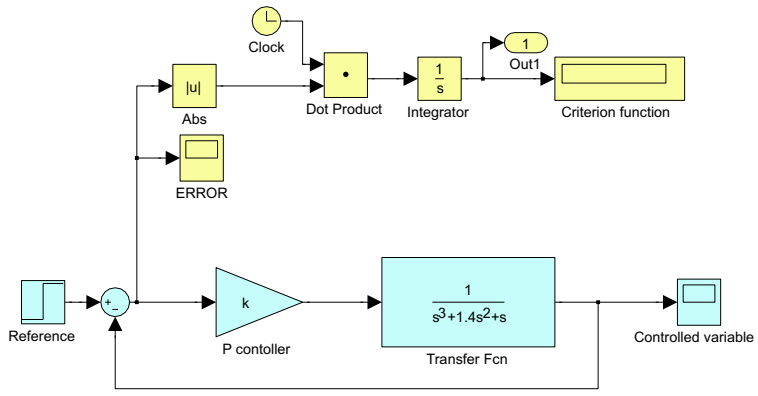
Optimizacijski program v Matlabu (datoteka `ZRS1_Primer4_11.m`) je naslednji:

```
k=0.7;
options=optimset('MaxIter',50,'TolFun',0.1, 'display','iter');
[par,val_cf]=fminsearch('cf_primer4_11',k,options)
```

Matlabova funkcija, ki definira cenilko (`cf_primer4_11.m`) pa je

```
function f = cf_primer4_11(k)
% System parameters used by SIMULINK model
t0 = 0  ;   % Start time
tf = 30 ;   % End time
opts=simset('SrcWorkspace','current') ;   % Set to use current workspace
opts=simset(opts,'DstWorkspace','current') ;
[t,x,y] = sim('ZRS1_Primer_4_11',[t0,tf],opts) ;  % Get the trajectory
[nt,nx]=size(x);
f=y(nt);
```

Simulink model (datoteka `ZRS1_Primer_4_11.mdl`) za cenilko ITAE prikazuje slika 5.40.



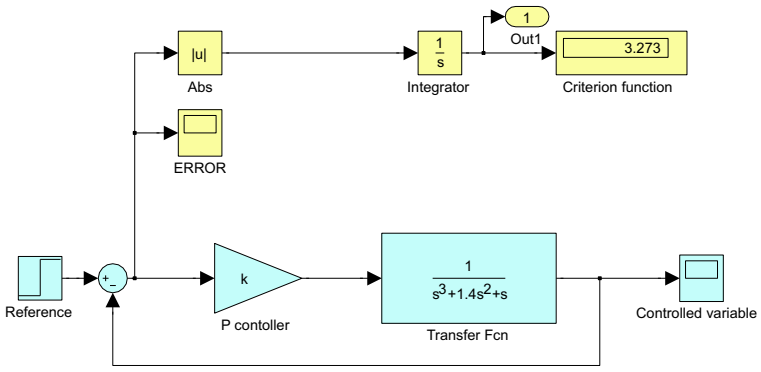
Slika 5.40: Model regulacijskega sistema v Simulinku s cenilko ITAE

Ta cenilka je povečala dušenje v regulacijskem sistemu in zmanjšala ojačenje na $K_{P_{opt}} = 0.317$. Njena vrednost se je pri tem od 21.37 pri $K_{P_{zač}} = 0.7$ spremenila na 7.39 pri $K_{P_{opt}} = 0.317$. Prevzpon pa se je zmanjšal na 4.1% (slika 5.42, krivulja b).

Druga izbrana cenilka je bila

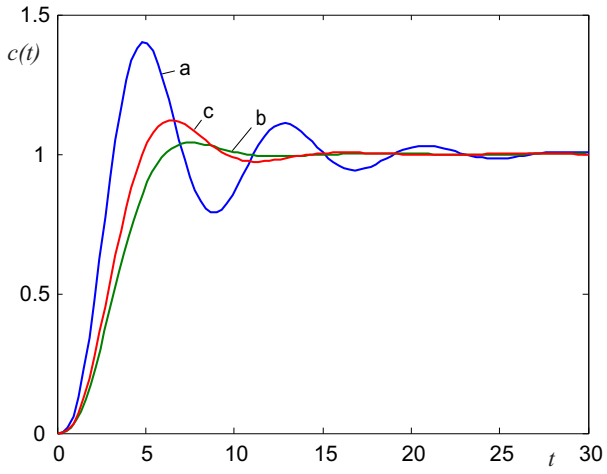
$$C_2 = \text{IAE} = \int_0^{30} |e(t)|dt \tag{5.36}$$

Ustrezni Simulink model prikazuje slika 5.41.



Slika 5.41: Model regulacijskega sistema v Simulinku s cenilko IAE

Začetno vrednost ojačenja regulatorja smo izbrali kot v prvem primeru, t.j. $K_{P_{zač}} = 0.7$. Optimizacijski algoritem je modificiral ojačenje $K_{P_{zač}} = 0.7$ v ojačenje $K_{P_{opt}} = 0.392$, cenilka IAE pa se je pri tem spremenila od 4.162 na 3.273. Prevzpon se je zmanjšal na 11.9% (slika 5.42, krivulja c).



Slika 5.42: Poteki regulirane veličine: a) $K_P = 0.7$ b) $K_P = 0.317$ c) $K_P = 0.392$

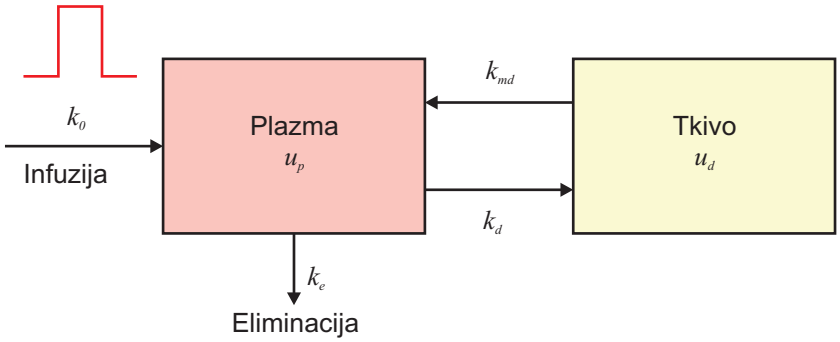
Obe cenilki imata dobro selektivnost, prva pa v splošnem daje bolj dušen potek regulirane veličine.

□

Primer 5.10 Uporaba simulacije in optimizacije v farmakokinetiki

Simulacija in optimizacija se uporabljata tudi pri študiju in modeliranju učinkovanja zdravil v človeškem telesu. Vedo imenujemo farmakokinetika. Z modeliranjem in simulacijo bomo pokazali učinkovanje sodobnega antibiotika ciprofloksacina, ki se uporablja za zdravljenje določenih bakterijskih okužb (okužbe dihal, sečil, hudih sistemskih okužb, ...). V času učinkovanja zdravila bo v naši študiji bolnik priključen na hemodializo, ki po odpovedi ledvic omogoči človeku preživetje. Hemodializa poleg 'čiščenja krvi' zmanjšuje koncentracijo učinkovine v krvi.

Prikazali bomo kombinacijo teoretičnega in eksperimentalnega modeliranja. Priljubljen način modeliranja v farmacije in farmakokinetiki je t.i. oddelčno (prostorno) modeliranje (angl. compartmental modelling). Model je v grafični obliki prikazan kot povezava več oddelkov (prostorov), npr. plazma, tkivo, ... Naš primer prikazuje slika 5.43.



Slika 5.43: Oddelčni model za študij učinkovanja ciprofloksacina

Bolnik dobi učinkovino ciprofloksacin preko pol urne infuzije (k_0) z masnim pretokom $400mg/h$. Model prikazuje koncentracijo v plazmi (u_p) in v tkivu (u_d). Učinkovanje koncentracije v plazmi na koncentracijo v tkivu je modelirano preko konstante k_d, učinkovanje koncentracije tkiva na plazmo pa preko konstante k_{md}. Eliminacija učinkovine iz plazme (preko ledvic v urin) se modelira s konstanto eliminacije k_e. Le ta ima vrednost 0.5, ko pa je bolnik priključen na hemodializo (med 4. in 8. uro 12 urnega opazovanja), pa vrednost 1.

Matematični model opišemo z naslednjimi enačbami:

$$\dot{u}_p = k_0 + k_{md}u_d - k_d u_p - k_e u_p \quad (5.37)$$
$$\dot{u}_d = -k_{md}u_d + k_d u_p \quad (5.38)$$

Infuzijo opisuje enačba

$$k_0 = \begin{cases} 400mg/h & ; \quad 0.5 > t \geq 0 \\ 0 & ; \quad 0 > t \geq 0.5 \end{cases} \quad (5.39)$$

Približno ocenjeni konstanti medsebojnega vplivanja oddelkov pa sta

$$k_d = 2 \quad (5.40)$$
$$k_{md} = 1 \quad (5.41)$$

Konstanta eliminacije je ocenjena z dvema vrednostima, za čas brez hemodialize (k_{ee}) in za čas med hemodializo (k_{ed})

$$k_e = \begin{cases} k_{ee} = 0.5 & ; \quad 4 > t \geq 8 \\ k_{ed} = 1 & ; \quad 4 \leq t < 8 \end{cases} \quad (5.42)$$

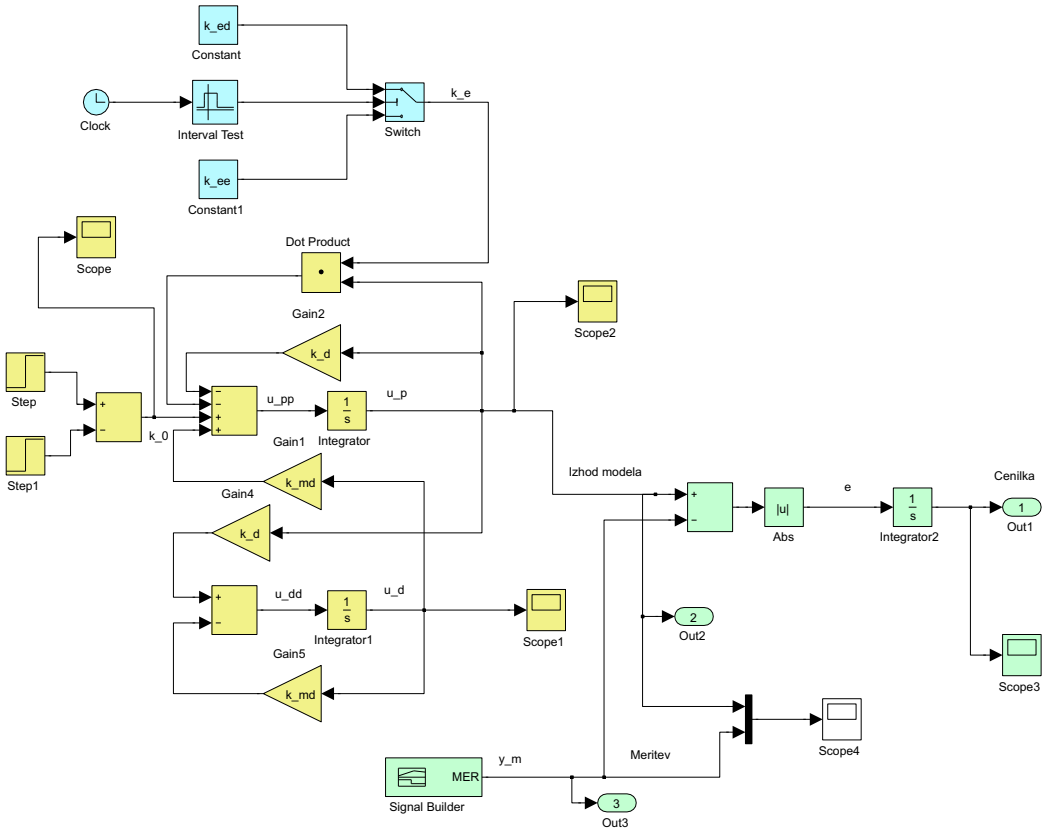
Za modeliranje smo imeli na voljo meritev koncentracije učinkovine v plazmi (y_d) v času 12 ur.

t	0	0.5	1.5	3	4	4.5	6	7.5	8	10	12
y_m	0	124.8	35.7	26.2	22.8	18.7	15	12.1	11.4	9.6	8.4

Simulink model (`cipro_sim.mdl`) prikazuje slika 5.44, rezultate simulacije pa sliki 5.45 (koncentracija v plazmi in meritev) in 5.46 (koncentracija v tkivu). Program `cipro_main_sim.m` definira parametre, požene simulacijo in izriše rezultate.

Ker so bile konstante k_{ee}, k_{ed}, k_d in k_{md} ocenjene zelo približno, smo uporabili optimizacijo z namenom, da bi dosegli boljše ujemanje odziva modela in meritev. Uporabili smo koncept, ki ga prikazuje slika 5.32 in cenilko

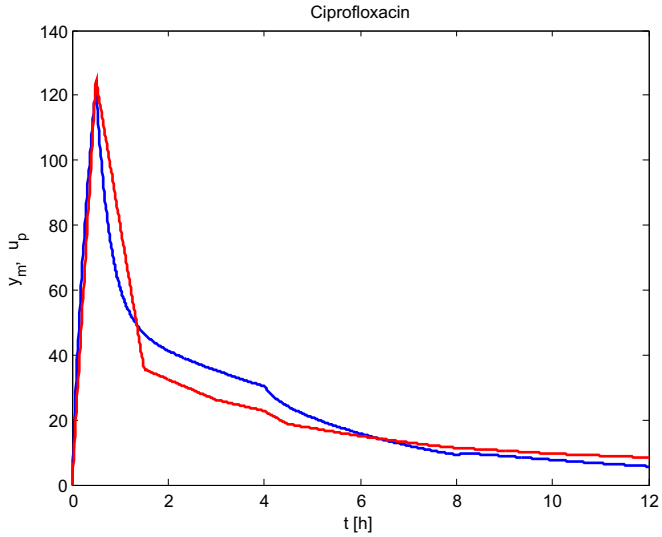
$$e(t) = u_p(t) - y_m(t)$$
$$c = \int_0^{12} |e(t)|dt \quad (5.43)$$



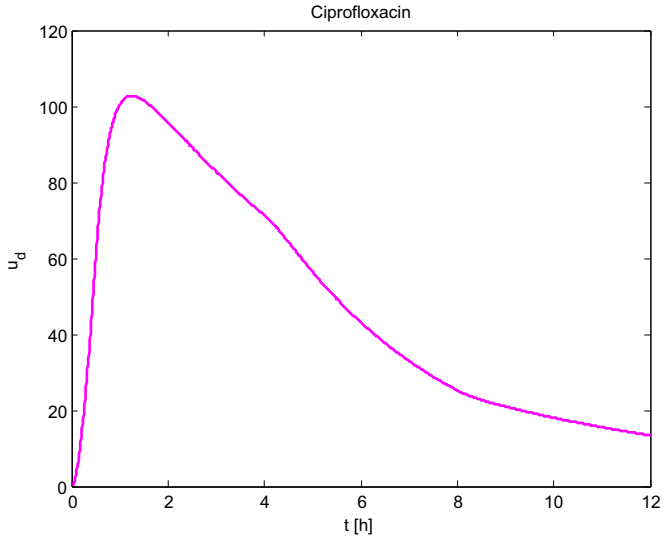
Slika 5.44: Simulink model učinkovanja ciprofloksacina

Glavni program za optimizacijo (`cipro_main_opt.m`) je naslednji:

```
clear all;
global tf k_0;
tf=12;
k_ee=0.5;
k_ed=1;
k_d=2;
k_md=1;
k_0=400;

[t,x,y] = sim('cipro_sim',tf) ;  % Get the trajectory
plot(t,y(:,2),t,y(:,3),'r','LineWidth',2)
hold on;
```



Slika 5.45: Potek koncentracije učinkovine v plazmi: model (modro) in meritev (rdeče)



Slika 5.46: Potek koncentracije učinkovine v tkivu

```
k=[k_ee k_ed k_d k_md];
options=optimset('MaxIter',100,'TolFun',0.1, 'display','iter');
[par,val_cf]=fminsearch('cipro_f',k,options)

k_ee=par(1);
k_ed=par(2);
```

```
k_d=par(3);
k_md=par(4);

[t,x,y] = sim('cipro_sim',tf) ;  % Get the trajectory
plot(t,y(:,2),'k','LineWidth',2)
title('Ciprofloxacin')
xlabel('t [h]')
ylabel('y_m,u_p')
```

Funkcija, ki ovrednoti cenilko s pomočjo simulacije (`cipro_f.m`), pa je

```
function f = cipro_f(k)
global tf k_0;
k_ee=k(1);
k_ed=k(2);
k_d=k(3);
k_md=k(4);
opts=simset('SrcWorkspace','current') ;   % Set to use current workspace
opts=simset(opts,'DstWorkspace','current') ;

[t,x,y] = sim('cipro_sim',tf,opts) ;  % Get the trajectory
[nt,nx]=size(x);
f=y(nt);
```

Rezultati, ki jih izpiše Matlabov program, so naslednji:

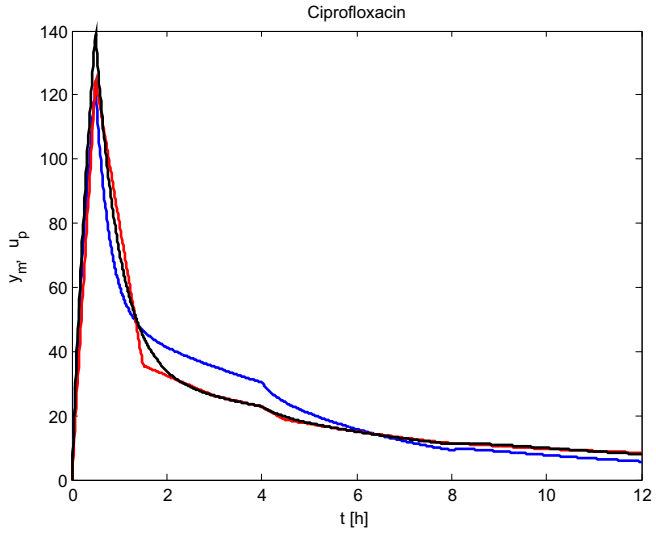
```
 Iteration   Func-count     min f(x)         Procedure
     0           1            57.3
     1           5          55.6441        initial simplex
     2           7           49.027        expand
     3           8           49.027        reflect
     4          10           45.016        expand
     5          12          38.5186        expand
             .
             .
             .
    98         159          18.3471        reflect
    99         161          18.3393        reflect
```

```
   100          163          18.3393         contract inside

Exiting: Maximum number of iterations has been exceeded
         - increase MaxIter option.
         Current function value: 18.339265

par =
    0.5054    0.7682    1.1358    0.4305

val_cf =
   18.3393
```

Optimalne vrednosti parametrov so torej

$k_{ee} = 0.5054$, $k_{ed} = 0.7682$, $k_d = 1.1358$ in $k_{md} = 0.4305$. Cenilka se je iz začetne vrednosti $c = 57.3$ zmanjšala na minimalno vrednost $c_{min} = 18.34$.

Slika 5.47 prikazuje poteke učinkovine v plazmi: meritev, potek modela pred optimizacijo in potek modela po optimizaciji.



Slika 5.47: Poteki učinkovine v plazmi: meritev (rdeče), potek modela pred optimizacijo (modro) in potek modela po optimizaciji (črno)

□

5.6 Simulacija s pomočjo Matlabovih funkcij

Razen simulacije v Simulinku nudi Matlab in dodatek Control Toolbox še nekaj drugih možnosti:

- odziv linearnih sistemov na enotino stopnico s pomočjo funkcije `step` ali odziv na delta impulz s pomočjo funkcije `impulse`,
- simulacijo linearnih časovno nespremenljivih sistemov s pomočjo funkcije `lsim`,
- simulacijo linearnih in nelinearnih, časovno nespremenljivih in časovno spremenljivih sistemov s pomočjo vgrajenih funkcij za numerično integracijo,
- simulacijo Simulink modela iz okolja Matlab s pomočjo funkcije `sim`.

5.6.1 Določitev odziva linearnega sistema s funkcijama `step` in `impulse`

Funkcija `step` izračuna odziv na enotino stopnico linearnega sistema (LTI), ki je opisan s prenosno funkcijo v polinomski (`tf`) ali faktorizirani obliki (`zpk`) ali v prostoru stanj (`ss`).

`step(sys)`
izračuna in nariše odziv sistema `sys` na enotino stopnico. Časovno območje in časovni korak se izbereta avtomatsko.

`step(sys,tf)`
izračuna in nariše odziv na časovnem intervalu `[0 tf]`.

`step(sys,t)`
izračuna in nariše odziv v trenutkih, ki jih določa vektor `t`, npr. `t=0:dt:tf`

`y=step(sys,t)`
izračuna se odziv `y` v trenutkih, ki jih določa vektor `t`, npr. `t=0:dt:tf`. Funkcija ne nariše odziva, pač pa ga lahko narišemo z ukazom `plot(t,y)`.

`[y,t]=step(sys)`
izračuna odziv `y` v trenutkih, ki jih določa `t`. Funkcija ne nariše odziva, pač pa

ga lahko narišemo z ukazom `plot(t,y)`. Časovno območje in časovni korak se izbereta avtomatsko.

`[y,t]=step(sys,tf)`
izračuna odziv `y` v trenutkih, ki jih določa `t` na časovnem intervalu `[0 tf]`. Funkcija ne nariše odziva, pač pa ga lahko narišemo z ukazom `plot(t,y)`.

`[y,t,x]=step(sys)`
funkcija vrne še stanja `x`, kar je predvsem uporabno pri sistemih v prostoru stanj.

Na podoben način se uporablja funkcija `impulse` za določitev odziva na delta impulz.

Primer 5.11 Primer prikazuje izračun in izris odziva prenosne funkcije sistema 2. reda z ojačenjem 2, lastno frekvenco 1 in dušilnim koeficientom 1 na enotino stopnico. Čas opazovanja je 10s. Program v Matlabu

```
G=tf([2],[1 2 1]);
tf=10;
[y,t]=step(G,tf);
plot(t,y)
```

nariše sliko 5.48.

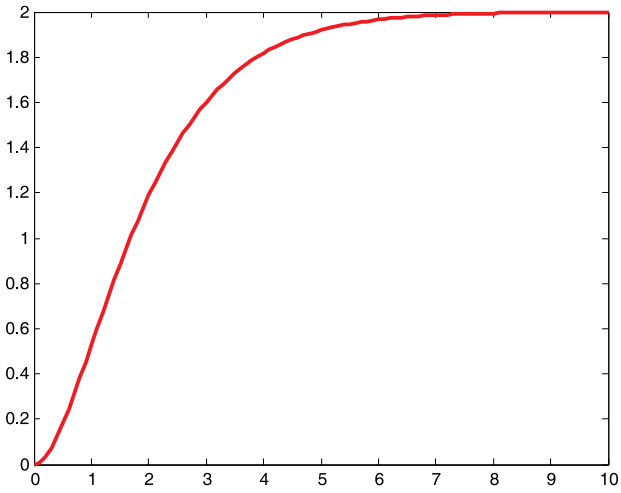
□

5.6.2 Simulacija s funkcijo `lsim`

S funkcijo `lsim` simuliramo zvezne (ali diskretne) dinamične sisteme, ki jih v Matlab vnesemo s funkcijami `ss` (prostor stanj), `tf` (prenosna funkcija v polinomski obliki) in `zpk` (prenosna funkcija v faktorizirani obliki). To so t.i. LTI modeli (linear time invariant model). Vhodni signal definiramo s pomočjo numerično podane funkcijske odvisnosti (s točkami podana funkcija).

Brez argumentov na levi strani enačaja lahko uporabljamo naslednji obliki:

```
lsim(sys,u,t)
lsim(sys,u,t,x0)
```



Slika 5.48: Odziv s pomočjo funkcije `step`

Funkcija `lsim` na zaslonu nariše odziv LTI modela. Pri tem je `sys` sistem, ki ga podamo s fukcijami `ss`, `tf` ali `zpk`. Par `u` in `t` je sestavljen iz dveh vektorjev, ki določata vhodni signel *u(t)*. `t` torej določa trenutke, v katerih je določen vhodni signal, `u` pa so vrednosti signala v ustreznih trenutkih. Trenutke lahko definiramo z naslednjo Matlabovo definicijo vektorja:

```
t = 0:dt:t_koncni
```

sinusni signal pa z

```
u=sin(t)
```

Vektorja `u` in `t` morata imeti enako število elementov (`length(t)=length(u)`).

Med dvema zaporednima točkama vhodnega signala se upošteva linearna interpolacija.

`x0` je matrika začetnih stanj (vsaka kolona opisuje časovni potek enega stanja). Predvsem se začetna stanja uporabljajo v zvezi z modelom v prostoru stanj.

Če kličemo funkcijo `lsim` le z enim argumentom - imenom modela

```
lsim(sys)
```

se odpre uporabniški vmesnik - Linear Simulation Tool. Uporabniški vmesnik za-

hteva vnos vhodnega signala in začetnih stanj, nakar lahko poženemo simulacijo.

Lahko pa navedemo tudi spremenljivke na levi strani enačaja:

```
y = lsim(sys,u,t)
[y,t] = lsim(sys,u,t)
[y,t,x] = lsim(sys,u,t)
[y,t,x] = lsim(sys,u,t,x0)
```

`y` je odziv LTI sistema `sys`. Če na levi strani enačaja navedemo `t`, potem se v določenih primerih lahko dogodi, da se vektor `t` po simulaciji spremeni (npr. v primeru, če je opisan z malo elementi). `x` je matrika stanj (vsaka kolona opisuje časovni potek enega stanja), `x0` pa matrika začetnih stanj. Slednja argumanta imata pomen predvsem pri zapisu modela v prostoru stanj.

Primer 5.12 Primer prikazuje določitev odziva sistema 2. reda z ojačenjem 2, lastno frekvenco 1 in dušilnim koeficientom 1 na sinusni vhodni signal. Čas opazovanja je 10s. Z uporabo programa v jeziku Matlab

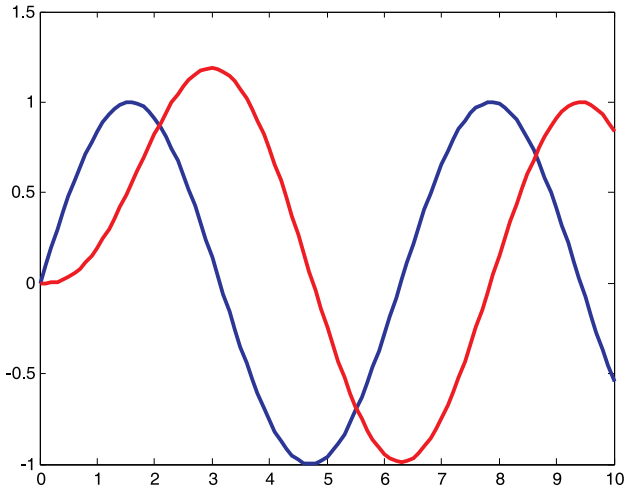
```
G=tf([2],[1 2 1]);
t=0:0.1:10;
u=sin(t);
y=lsim(G,u,t);
plot(t,u,t,y)
```

□

dobimo sliko 5.49.

5.6.3 Simulacija s pomočjo vgrajenih funkcij za numerično integracijo

Splošnejši način simulacije, ki se ne omejuje na linearne in časovno nespremenljive sisteme, poteka preko vgrajenih funkcij za numerično integriranje. Bolj matematično imenujemo postopek začetnovrednostni problem. Koncept je najlaže razumljiv, če si predstavljamo, da je nelinearni dinamični sistem zapisan v



Slika 5.49: Odziv s pomočjo funkcije `lsim`

prostoru stanj, torej z diferencialnimi enačbami 1. reda, oz. da so vsi odvodi stanj sistema odvisni od stanj, vhodnega signala in neodvisne spremenljivke simulacije.

$$\dot{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{u}; t) \tag{5.44}$$

Včasih je zelo ilustrativen način, da za nelinearni sistem razvijemo simulacijsko (bločno) shemo, v njej označimo stanja in odvode stanj in iz sheme napišemo enačbi 5.44 (slika 3.26).

Sintakso opisujejo naslednje vrstice:

```
[t,x] = solver(@odefun,tspan,x0)
[t,x] = solver(@odefun,tspan,x0,options)
```

kjer je `solver` ena izmed integracijskih funkcij `ode45`, `ode23`, `ode113`, `ode15s`, `ode23s`, `ode23t` ali `ode23tb`.

Vhodni argumenti imajo naslednji pomen:

`odefun`
Funkcija ovrednoti trenutne vrednosti odvodov (`dx`), to je levo stran enačb stanj, iz trenutnih stanj (`x`) in trenutne vrednosti neodvisne spremenljivke (`t`). Navedena mora biti z znakom @ spredaj, kar pomeni klic funkcije preko kazalca.

```
function dx=odefun(t,x)
dx(1)=...
dx(2)=...
dx(3)=...
dx=[dx(1) dx(2) dx(3) ...]'
...
end
```

Funkcijo zapišemo kot m-datoteko `odefun.m`

`tspan`
Če ima vektor `tspan` 2 elementa, potem opisuje simulacijski (integracijski) interval `[t0,tf]`. Integracijska metoda vsili začetne pogoje v trenutku `t0` in nato izvrši integracijo do trenutka `tf`. Vmesne trenutke določa integracijska metoda. Če pa želimo dobiti rezultate simulacije v predpisanih trenutkih, navedemo te trenutke v vektorju `tspan`: `tspan = [t0,t1,...,tf]`. V tem primeru mora torej vektor `tspan` imeti več kot dva elementa. Vmesni trenutki pa so le trenutki, v katerih dobimo rezultate. Integracijske metode sicer določijo svoj korak, ki omogoča, da napaka pri simulaciji ni večja od dopustne.

`x0`
Vektor začetnih pogojev.

`options`
Integracijska metoda omogoča številne nastavitve, ki imajo sicer privzete vrednosti. Nastavitve definiramo s funcijo `odeset`. Glavne nastavitve zadevajo dopustno relativno `RelTol` (`1e-3` je privzeta vrednost) in absolutno napako `AbsTol` (`1e-6` je privzeta vrednost) ter razne omejitve v zvezi z računskim korakom.

Izhodni argumenti imajo naslednji pomen:

`t`
Časovni trenutki, v katerih dobimo rezultate.

`x`
Matrika rezultatov. V kolikor je rezultat en signal (eno stanje), je to vektor, sicer vsaka kolona pomeni en signal (eno stanje).

Integracijski algoritmi

`ode45`
Eksplicitna enokoračna metoda Runge-Kutta, ki za sprotno vrednotenje napake kombinira metodo 4 in 5 reda (Dormand Prince). Je najbolj univerzalna metoda in zato s to metodo najprej poskusimo rešiti problem.

`ode23`
Eksplicitna enokoračna metoda Runge-Kutta, ki za sprotno vrednotenje napake kombinira metodo 2 in 3 reda (Bogacki Shampine). Je bolj učinkovita, če se ne zahteva velika natančnost. Uporabna tudi za srednje toge sisteme.

`ode113`
je večkoračna metoda Adams-Bashforth-Moulton spremenljivega reda. Je učinkovitejša od ode45 pri strogih zahtevah za točnost, pa tudi v primerih kompleksnih izrazov za izračun odvodov. Ker je večkoračna metoda, uporaba ni priporočljiva, če nastopajo nezveznosti v signalih. V tem primeru so primernejše enokoračne metode (npr. `ode45, ode23`).

Zgornji algoritmi se predvsem uporabljajo za netoge sisteme. Če se čas simulacije zelo poveča, je verjetno, da gre za togi sistem (zelo različni velikostni razred časovnih konstant) in v tem primeru se izplača poizkusiti z eno od naslednjih metod: `ode15s, ode23s, ode23t, ode23tb`. S temi metodami dosežemo primerno hitrost simulacije, ne pa tudi velike natančnosti. Običajno med temi metodami največjo natančnost zagotavlja metoda `ode15s`.

Več o integracijskih metodah opisuje poglavje 6.1.

Primer 5.13 Model ekološkega sistema, v katerem nastopajo roparji in žrtve, smo predstavili v primeru 1.3. Matematični model ima obliko

$$\begin{aligned} \dot{x}_1 &= a_{11}x_1 - a_{12}x_1x_2 \qquad x_1(0) = x_{10} \\ \dot{x}_2 &= a_{21}x_1x_2 - a_{22}x_2 \qquad x_2(0) = x_{20} \end{aligned} \tag{5.45}$$

Sedaj izvedimo simulacijo v programskem okolju Matlab. Glavni program ima obliko:

```
rab0=520; % initial no. of rabbits
fox0=85;  % initial no. of foxes
tfin=10;
[t,x]=ode45(@lhotka,[0 tfin], [rab0 fox0]);
subplot(2,1,1)
```

```
plot(t,x(:,1))
title('Rabbits')
subplot(2,1,2)
plot(t,x(:,2))
title('Foxes')
```

Funkcija `lhotka` pa je naslednja:

```
function dx=lhotka(t,x)
  a11=5;
  a12=0.05;
  a21=0.0004;
  a22=0.2;
  % calculation of derivatives
  dx(1)=a11.*x(1)-a12.*x(1).*x(2);
  dx(2)=a21.*x(1).*x(2)-a22.*x(2);
  dx=[dx(1) dx(2)]'
end
```

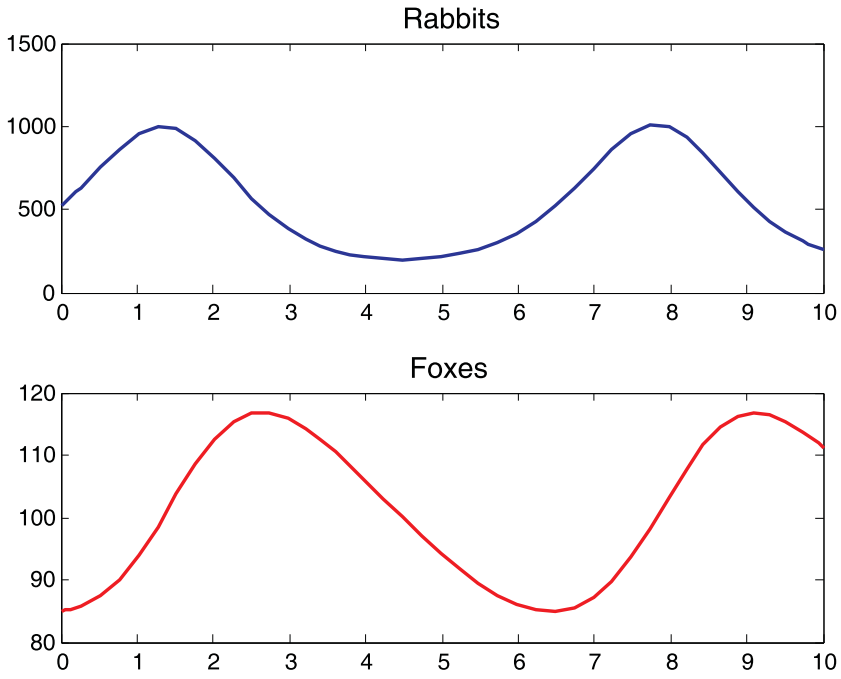
Rezultate simulacije prikazuje slika 5.50. □

5.6.4 Simulacija s funkcijo `sim`

S funkcijo `sim` zaženemo model, ki je opisan s Simulink shemo. Tako lahko v okolju Matlab programiramo kompleksne eksperimente, ki vključujejo simulacijske teke modela v Simulinku. Ponavadi je koristno, da modelu v Simulinku z out bloki iz knjižnice Sinks označimo izhode oz. signale, ki jih opazujemo.

`sim('model')`
ukaz simulira Simulink model `model.mdl`. Veljajo krmilni parametri, kot so nastavljeni v Simulink shemi.

`[t,x,y]=sim('model', timespan)`
`t` je vektor, ki vsebuje trenutke, v katerih se izračunajo rezultati simulacije. `x` je matrika stanj (vsaka kolona opisuje eno stanje)- stanja so izhodi integratorjev. `y` je matrika izhodov (vsaka kolona opisuje en izhod, izhodi so določeni z out bloki v Simulink shemi). `timespan` je v splošnem vektor. Če ima vektor en element,



Slika 5.50: Rezultati simulacije

je to končni čas simulacije `tf`, če ima dva elementa, sta to začetni in končni čas simulacije `[t0 tf]`, če pa je več elementov, so to trenutki, v katerih želimo dobiti rezultate simulacije `[t0 t1 t2 ... tf]`.

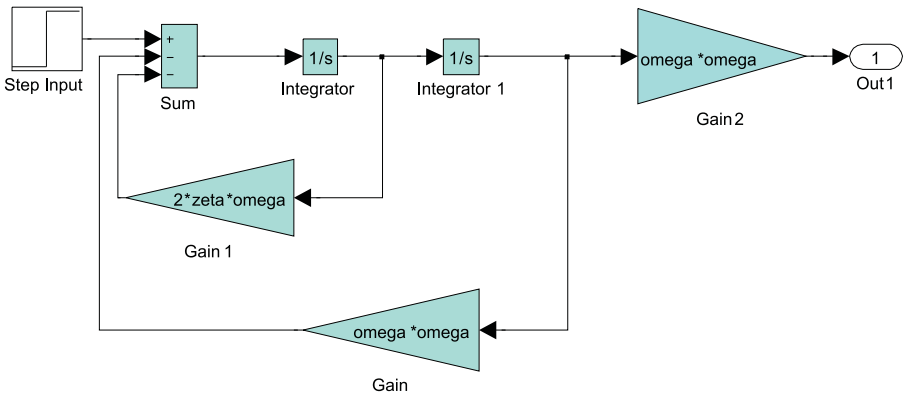
`[t,x,y]=sim('model', timespan, options)`
options vsebuje vse mogoče nastavitve, ki se sicer lahko nastavijo v Simulink shemi, npr. tolerance, integracijsko metodo, minimalni in maksimalni dopustni računski korak,... Za nastavitev parametra `options` se uporablja posebna fukcija `simset`. Če želimo izbrati integracijsko metodo `ode45`, uporabimo ukaz

`options=simset('solver','ode45')`.

Primer 5.14 Primer prikazuje parametrizacijo sistema 2. reda pri različnih vrednostih parametra ζ. Parameter ζ spreminjamo od 0 do 2 s korakom 0.1. Odziv sistema določimo s pomočjo funkcije `step` in s pomočjo funkcije `sim`. Slednja požene model `secor1.mdl` v Simulinku. Prepričamo se, da dobimo po obeh metodah enake rezultate. Program v jeziku Matlab

```
omega=1;
tf=20;
for zeta=0:0.1:2;
    [t,x,y]=sim('secor1',tf);
    Gs=tf(1,[1 2*zeta*omega omega*omega]);
    [y1,t1]=step(Gs,tf);
    subplot(2,1,1);
    hold on
    plot(t,y,'r');
    subplot(2,1,2);
    plot(t1,y1,'b');
    hold on;
end
```

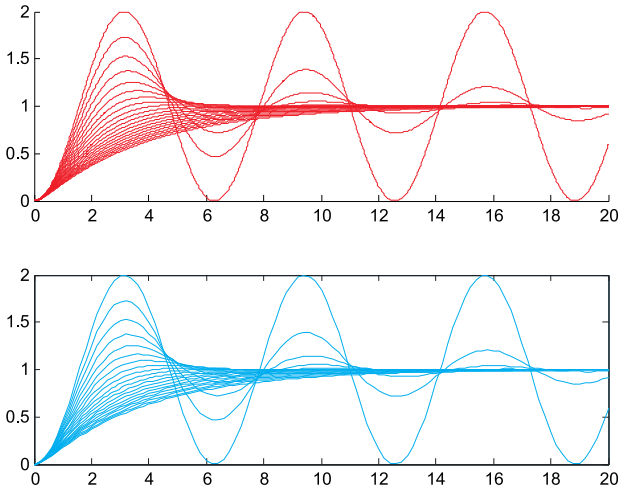
ki uporabi Simulink shemo `secor1.mdl`, ki jo prikazuje slika 5.51.



Slika 5.51: Simulink shema sistema 2.reda

nariše sliko 5.52.

□



Slika 5.52: Parametrizacija sistema 2. reda

5.7 Simulacija s splošnonamenskimi programskimi jeziki

Sliki 3.30 in 3.31 prikazujeta strukturo simulacijskega programa, ki temelji na obravnavanem konceptu digitalnih simulacijskih sistemov (glej podpoglavje 3.9). Za izvedbo lahko uporabimo katerikoli splošnonamenski programski jezik oz. okolje: Matlab, Visual Basic, C++, Fortran, Java script, Java, ASP, Phyton, Visual Studio, itd.

V nadaljevanju bomo prikazali enostavnejši način izvedbe simulacijskega programa z enim programskim modulom in zahtevnejšo bolj modularno izvedbo s pomočjo večih programskih modulov.

5.7.1 Enostavnejša izvedba v enem programskem modulu

Uporabimo splošno strukturo simulacijskega programa (slika 3.30).

Primer 5.15 Simulacija ekološkega sistema žrtev in roparjev v jeziku Matlab

Ekološki problem žrtev in roparjev smo uvedli v primeru 1.3. Opisujeta ga difer-

encialni enačbi

$$\begin{aligned} \dot{x}_1 &= a_{11}x_1 - a_{12}x_1x_2 \qquad x_1(0) = x_{10} \\ \dot{x}_2 &= a_{21}x_1x_2 - a_{22}x_2 \qquad x_2(0) = x_{20} \end{aligned} \tag{5.46}$$

Podatkovna baza modela sestoji iz štirih konstant $(a_{11}, a_{12}, a_{21}, a_{22})$ in dveh začetnih pogojev $(x_1(0), x_2(0))$. Krmilni parametri simulacije pa so: začetna vrednost neodvisne spremenljivke, t.j. začetni čas, računski korak (t.j. prirastek neodvisne spremenljivke med simulacijo) in končni čas simulacije.

Upoštevali bomo naslednjo relacijo med računalniškimi in problemskimi spremenljivkami:

x_1	`rab`	$\dot{x}_1$	`rabdot`
x_2	`fox`	$\dot{x}_2$	`foxdot`
a_{11}	`a11`	a_{12}	`a12`
a_{21}	`a21`	a_{22}	`a22`

Konstante modela lahko uvedemo direktno v enačbe, ki opisujejo matematični model. Toda če želimo konstante med simulacijskimi teki spreminjati, jim je potrebno dati ime, oz. uvesti ustrezne spremenljivke. Spremenljivke inicializiramo s prireditvenimi stavki

```
a11=5;
a12=0.05;
a21=0.0004;
a22=0.2;
```

Na enak način definiramo začetni čas simulacije `t0`, računski korak `dt` in trajanje simulacijskega teka `tfin`

```
t0=0;
dt=0.01;
tfin=10;
```

`dt` moramo izbrati glede na karakteristične dinamične lastnosti modela tako, da se izognemo numerični problematiki in zagotovimo predpisano točnost.

Nato definiramo začetne vrednosti stanj

```
rab=520;
fox=85;
```

Simulacija poteka v zanki, v kateri se neodvisna spremenljivka spreminja od začetne do končne vrednosti z zahtevanim prirastkom

```
%časovna zanka
  for t=0:dt:tfin;
  ...
  end;
```

Glede na osnovni koncept digitalne simulacije, ki ga prikazuje slika 3.28, sta pri vsakem obhodu zanke dve temeljni operaciji: ovrednotenje odvodov spremenljivk stanja v nekem trenutku t in integracija, ki določi izhode vseh integratorjev v trenutku $t + \Delta t$. Razen teh temeljnih operacij pa moramo dodati še programske vrstice za prikaz rezultatov in za končanje simulacijskega teka. Glede na kompleksnost integracijske metode so omenjene operacije precej prepletene, v primeru najenostavnejše Eulerjeve metode pa velja naslednji vrstni red:

1. **Ovrednotenje odvodov spremenljivk stanja:** Odvoda oz. izhoda integratorjev izračunamo z dvema prireditvenima stavkoma

   ```
   %DERIV
     rabdot=a11*rab-a12*rab*fox;
     foxdot=a21*rab*fox-a22*fox;
   ```

   Stavka sta sicer povsem enaka, kot v jeziku SIMCOS (primer 4.1), v splošnem pa je bistvena razlika v tem, da je pri uporabi simulacijskega jezika vrstni red teh stavkov poljuben, pri simulaciji s splošnonamenskim programskim jezikom pa mora uporabnik paziti na ustrezni vrstni red. Torej če se spremenljivka v nekem stavku n1 na desni strani enačaja izračuna v nekem drugem n2 (se nahaja levo od enačaja), mora biti stavek n2 pred stavkom n1. V našem primeru pa je vrstni red stavkov poljuben, saj so na desni strani obeh stavkov le konstante modela in izhoda integratorjev (stanji sistema).

2. **Prikaz rezultatov:** Rezultate simulacije shranimo v ustrezne indeksirane spremenljivke, ki jih bo možno po simulaciji uporabiti za poljubne prikaze ali obdelave

```
% OUTPUT
  zajci(i)=rab;
  lisice(i)=fox;
  cas(i)=t;
```

3. **Integracija:** Uporabimo najenostavnejšo in najbolj ilustrativno Eulerjevo integracijsko metodo, ki jo določa enačba 3.71. Podrobneje pa so metode opisane v poglavju 6.1. Integracijo realiziramo s pomočjo naslednjih stavkov

```
% INTEG
  rab=rab+rabdot*dt;
  fox=fox+foxdot*dt;
```

Integracijski postopek izračuna vrednosti `rab` in `fox` že za naslednji računski korak, zato je nujno, da je shranjevanje trenutnih rezultatov opravljeno pred integriranjem. Po izvršitvi integracije je potrebno celotni postopek ponoviti, t.j. narediti je potrebno naslednji prehod po zanki. Če je izpolnjen pogoj za končanje simulacijskega teka, pa se izvrši Matlabov ukaz za izris rezultatov

```
% prikaz rezultatov po simulaciji
  plot(cas,lisice,cas,zajci);
```

Celotni program za simulacijo ekološkega sistema žrtev in roparjev v jeziku Matlab je naslednji:

```
% inicializacija
  a11=5;
  a12=0.05;
  a21=0.0004;
  a22=0.2;
  dt=0.01;
  tfin=10;
  rab=520;
  fox=85;
  i=1;

% časovna zanka
  for t=0:dt:tfin;
```

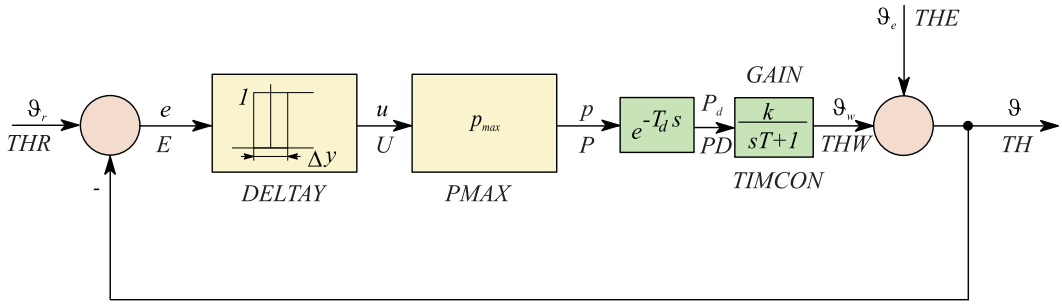
```
      % DERIV
        rabdot=a11*rab-a12*rab*fox;
        foxdot=a21*rab*fox-a22*fox;

      % OUTPUT
        zajci(i)=rab;
        lisice(i)=fox;
        cas(i)=t;

      % INTEG
        rab=rab+rabdot*dt;
        fox=fox+foxdot*dt;
      i=i+1;
    end;

% prikaz rezultatov po simulaciji
  plot(cas,lisice);
  plot(cas,zajci);
```

Rezultati simulacije so identični rezultatom, ki jih prikazuje slika 4.5 pri primeru 4.1. □

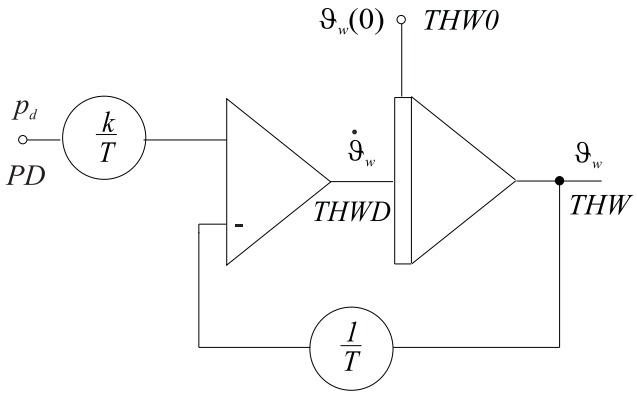
Pri uporabi splošnonamenskih jezikov mora torej uporabnik sam določiti vrstni red stavkov, ki opisujejo model (odvode spremenljivk stanja). To sicer v tem primeru ni bilo pomembno, pomembno pa bo v naslednjem primeru, ki prikazuje simulacijo regulacije ogrevanja.

Primer 5.16 Simulacija regulacije ogrevanja v jeziku Matlab

Model regulacije ogrevanja opisuje primer 1.2, ustrezno simulacijo v jeziku SIMCOS pa prikazuje primer 4.2. V modelu uporabljamo le en integrator, katerega izhod je spremenljivka `THW` (slika 5.54). To spremenljivko obravnavamo kot znano oz. izhodiščno pri razvrščanju blokov (stavkov). Najprej je potrebno izračunati referenčno temperaturo (`THR`) in absolutno temperaturo (`TH`). Vrstni red izračuna teh dveh spremenljivk ni pomemben, saj je povezava med njima ločena z integratorjem. Nato izračunamo pogrešek, izhod dvopoložajnega regulatorja, izhodni signal grela, njegov zakasnjeni signal in končno še odvod `THWD`. Določitev tega



Slika 5.53: Bločni diagram regulacijskega sistema



Slika 5.54: Simulacijska shema temperaturnega procesa

vrstnega reda zelo olajša uporaba bločnega diagrama 5.53 in simulacijske sheme procesa na sliki 5.54.

Simulacijski program v jeziku Matlab je naslednji:

```
%Temperaturni regulacijski sistem
%Simulacija z osnovnim programiranjem v Matlabu

%Konstante modela
DELTAY=1; PMAX=5; GAIN=2; TIMCON=1; T1=1; T2=1
THE=15; TDELAY =0; THW0=1;

%Opis referenčnega signala
REFX=[0 6  9 15 21];
REFY=[15 20 18 20 15];
```

```
%Krmilni parametri simulacije
DT=0.02; TFIN=24;

%Inicializacija polja za realizacijo mrtvega časa
INDEX=TDELAY/DT+1;
WORK= zeros(1,INDEX);

%Začetni pogoji
THW=THW0; U=0; K=1;

%Začetek simulacijske zanke
for T=0:DT:TFIN;
    %Generacija referenčne temperature
    for I=1:5;
        if T>=REFX(I);
        THR=REFY(I);
        end
    end
    TH=THW+THE;

    %Izračun pogreška
    E=THR-TH;

    %Histereza
    if E>DELTAY/2
        U=1;
    elseif E<-DELTAY/2
        U=0;
    end

    %Preračun v moč grelca
    P=PMAX*U;

    %Realizacija mrtvega časa
    for I=INDEX:-1:2
        WORK(I)=WORK(I-1);
    end
    WORK(1)=P;
    PD=WORK(INDEX);
```

```
    %Izračun odvoda (DERIV)
    THWD=(-1/TIMCON)*THW+(GAIN/TIMCON)*PD;

    %Lahko upoštevamo različni časovni kostanti pri ogrevanju in ohlajanju
    if THWD<0
        TIMCON=T2;
    elseif THWD>=0
        TIMCON=T1;
    end

    %Shranjevanje rezultatov (OUTPUT)
    T_1(K)=T;THR_1(K)=THR;P_1(K)=P;TH_1(K)=TH;

    K=K+1;

    % Eulerjeva integrac. metoda (INTEG)
    THW=THW+THWD*DT;

end

%Prikaz rezultatov
plot(T_1,THR_1,T_1,P_1,T_1,TH_1)
```

Struktura programa je podobna, kot v prejšnjem primeru. Vsi podatki in oznake spremenljivk pa so enake kot pri simulaciji z jezikom SIMCOS v primeru 4.2. Dodatno pojasnilo zahtevajo le naslednji sklopi: funkcijski generator (za izvedbo referenčne temperature), ki ga opisuje tabela 4.2 v primeru 4.2, vrstice za realizacijo histereze in vrstice za realizacijo mrtvega časa.

Funkcijski generator v jeziku Matlab definiramo z vektorjema `REFX` in `REFY`, ki hranita vrednosti lomnih točk neodvisne in odvisne spremenljivke. Vektorja inicializiramo med operacijami pred simulacijskim tekom:

```
    REFX=[0 6  9 15 21];
    REFY=[15 20 18 20 15];
```

Ti programski stavki torej ustrezajo stavku `TABLE` v jeziku SIMCOS. Referenčno temperaturo pa generiramo z naslednjimi programskimi vrsticami:

```
for I=1:5;
     if T>=REFX(I);
     THR=REFY(I);
     end
 end
```

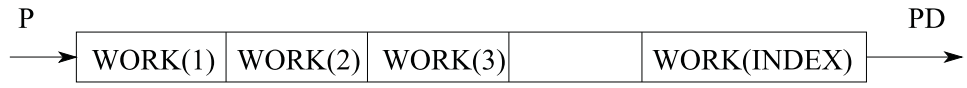
Te programske vrstice sicer niso ekvivalent funkcijskega generatorja v jeziku SIMCOS, saj ne vsebujejo linearne interpolacije. Toda za odsekovno konstantni referenčni signal dobimo ustrezno delovanje.

Histerezo realiziramo z naslednjimi vrsticami programa:

```
%Histereza
    if E>DELTAY/2
        U=1;
    elseif E<-DELTAY/2
        U=0;
    end
```

Zunaj področja histereze je torej izhodni signal `U=0` ali `U=1`. Znotraj histereze pa vrstice ne spremenijo izhodnega signala, torej obdrži `U` prejšnjo vrednost. S tem modeliramo 'spomin' histereze.

Mrtvi čas (časovno zakasnitev) realiziramo z vektorjem `WORK`, ki mora imeti dimenzijo `TDELAY/DT+1`, kar je v programu označeno s spremenljivko `INDEX`. Ustrezen sistem za realizacijo mrtvega časa prikazuje slika 5.55.



Slika 5.55: Sistem za realizacijo mrtvega časa

Vektor `WORK` je inicializiran s stavki

```
INDEX=TDELAY/DT+1;
WORK= zeros(1,INDEX);
```

Pri vsakem obhodu po zanki moramo elemente vektorja `WORK` premakniti za eno mesto proti višjemu indeksu. Vhod v blok za zakasnitev je povezan s prvim elementom polja `WORK`, izhod pa je element z indeksom `INDEX`:

```
for I=INDEX:-1:2
     WORK(I)=WORK(I-1);
 end
 WORK(1)=P;
 PD=WORK(INDEX);
```

Z opisano realizacijo lahko proučujemo tudi model brez mrtvega časa (`TDELAY=0`).

Tako kot v simulacijskem programu v jeziku SIMCOS, lahko tudi v tem primeru modeliramo nelinearno lastnost modela, t.j. različni časovni konstanti med ogrevanjem `T1` in ohlajanjem `T2`.

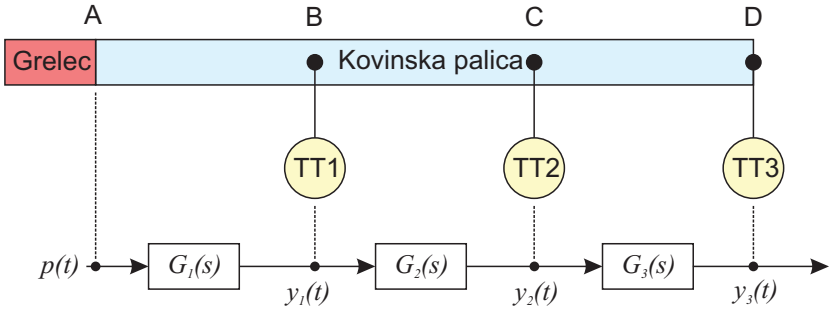
```
if THWD<0
    TIMCON=T2;
elseif THWD>=0
    TIMCON=T1;
end
```

Rezultati simulacije so identični rezultatom v primeru 4.2, čeprav smo uporabili nenatančni Eulerjev integracijski postopek, a zadosti majhen računski korak. Slika 4.7 prikazuje rezultate simulacije (temperaturo v prostoru ϑ in želeno temperaturo ϑ_r v zgornjem in moč grela p v spodnjem diagramu). Grelo se vključuje približno na 30 min, temperatura v prostoru pa niha v pasu 1^oC. Če podvojimo velikost histereze (2^oC), dobimo rezultate, ki jih prikazuje slika 4.8. Nihanja temperature postanejo v tem primeru prevelika za ugodno počutje v prostoru. Perioda preklapljanja grela pa se poveča na približno 50 min. Slika 4.9 prikazuje rezultate simulacije pri mrtvem času $T_d = 0.1\ h$. Če primerjamo rezultate simulacije z rezultati na sliki 4.7, vidimo, da temperatura niha v širšem pasu (približno 2^oC). Nihanje temperature je torej odvisno od širine histereze in od zakasnitev procesa. Očitno je, da je termostatsko tipalo predaleč od grela, da bi lahko dosegli ugodno temperaturo.

Slika 4.10 prikazuje rezultate simulacije pri 10 kW grelu. Opazimo, da temperatura narašča hitreje, vendar so nihanja temperature večja kot v primeru na sliki 4.9 (približno 3^oC). □

Primer 5.17 Simulacija segrevanja kovinske palice

Proučujemo segrevanje kovinske palice v štirih točkah (slika 5.56). Postopek modeliranja pokaže, da je možno v nekem ožjem področju (okoli $0°C$) uporabiti linearni model. V tem področju velja, da je časovna konstanta ogrevanja približno enaka časovni konstanti ohlajanja. Lahko si predstavljamo, da eksperimente dejansko delamo okoli temperature okoli $0°C$ ali pa, da smo pri modeliranju izvedli linearizacijo okoli neke delovne točke in izvajamo eksperimente okoli te delovne točke.



Slika 5.56: Ogrevanje kovinske palice

S pomočjo eksperimentalnega modeliranja smo prišli do naslednjih prenosnih funkcij:

$$G_1(s) = \frac{Y_1(s)}{P(s)} = \frac{k_1}{T_1 s + 1} = \frac{10}{10s + 1} \tag{5.47}$$

$$G_2(s) = \frac{Y_2(s)}{Y_1(s)} = \frac{k_2}{T_2 s + 1} = \frac{0.5}{20s + 1} \tag{5.48}$$

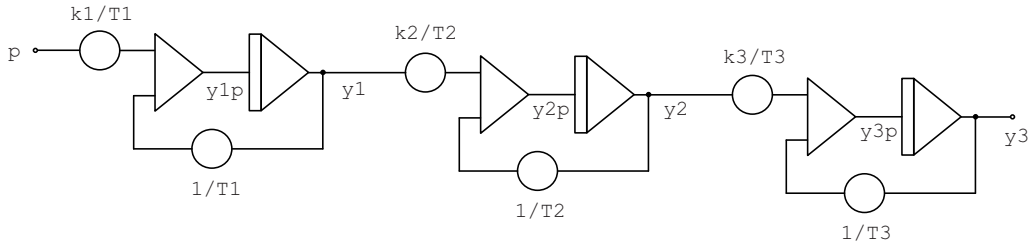
$$G_3(s) = \frac{Y_3(s)}{Y_2(s)} = \frac{k_3}{T_3 s + 1} = \frac{0.5}{20s + 1} \tag{5.49}$$

$p(t)$ je moč grelca $[kW]$, $y_1(t)$, $y_2(t)$ in $y_3(t)$ pa so temperature $[°C]$. Časovne konstante so v sekundah $[s]$. Predpostavimo, da je v začetku temperatura palice enaka temperaturi okolice $0°C$. Nato pa v točki A izvedemo vzbujanje s toplotnim virom konstantne vrednosti $p(t) = 1kW$.

Simulacijsko shemo, ki nam je v pomoč pri pisanju programa, prikazuje slika 5.57.

Program za simulacijo v jeziku Matlab je naslednji:

```
clear;
```



Slika 5.57: Simulacijska shema za simulacijo ogrevanja kovinske palice

```
dt=1;           % rač. korak
Tmax=200;       %čas sim. teka

%začetna stanja
y1=0;
y2=0;
y3=0;

%parametri modela
k1=10;
k2=0.5;
k3=0.5;
T1=10;
T2=20;
T3=20;
i=1;
%simulacijska zanka
for t=0:dt:Tmax

   % 1. izračun odvodov - DERIV
    p=1;        %vhodni signal
    y1p=k1/T1*p-1/T1*y1;
    y2p=k2/T2*y1-1/T2*y2;
    y3p=k3/T3*y2-1/T3*y3;

   % 2.Prikaz rezultatov
   %   shranjevanje v vektorje za kasnejši prikaz
    cas(i)=t;
    yy(i,1)=y1;
    yy(i,2)=y2;
    yy(i,3)=y3;
```

```
    % 3. Integracija po enokoračni EULER-jevi metodi
    y3=y3+y3p*dt;
    y2=y2+y2p*dt;
    y1=y1+y1p*dt;

    i=i+1;
end

%Prikaz rezultatov po simulacijskem teku
figure(1);
plot(cas,yy);

 %Izračun rezultatov še s pomočjo funkcij Control Toolbox
hold on
t=0:dt:Tmax
G1=tf(10,[10 1])
G2=tf(0.5,[20 1])
G3=tf(0.5,[20 1])
G1G2=series(G1,G2)
Gs=series(G1G2,G3)
y=step(Gs,t);
plot(t,y,':');
```

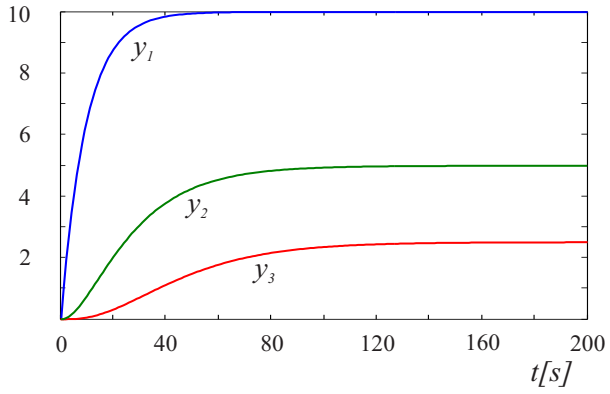
Ker je model linearen, smo lahko temperaturo y_3 določili tudi z izračunom prenosne funkcije $G(s) = G_1(s)G_2(s)G_3(s)$ in uporabo funkcije `step`. Rezultate simulacije prikazuje slika 5.58. Temperatura y_3 je enaka po obeh metodah, kar pomeni, da smo za Eulerjevo integracijsko metodo izbrali dovolj majhen računski korak.

□

Primer 5.18 Simulacija filtra, podanega s prenosno funkcijo, v realnem času

Chebyshev filter 2. reda z mejno frekvenco $f_m = 12.5Hz$ in dušenjem vsaj 30db v zapornem delu ($f > 700Hz$) podaja prenosna funkcija

$$G(s) = \frac{as^2 + bs + c}{s^2 + ds + e} \tag{5.50}$$



Slika 5.58: Rezultati simulacije pri ogrevanju kovinske palice

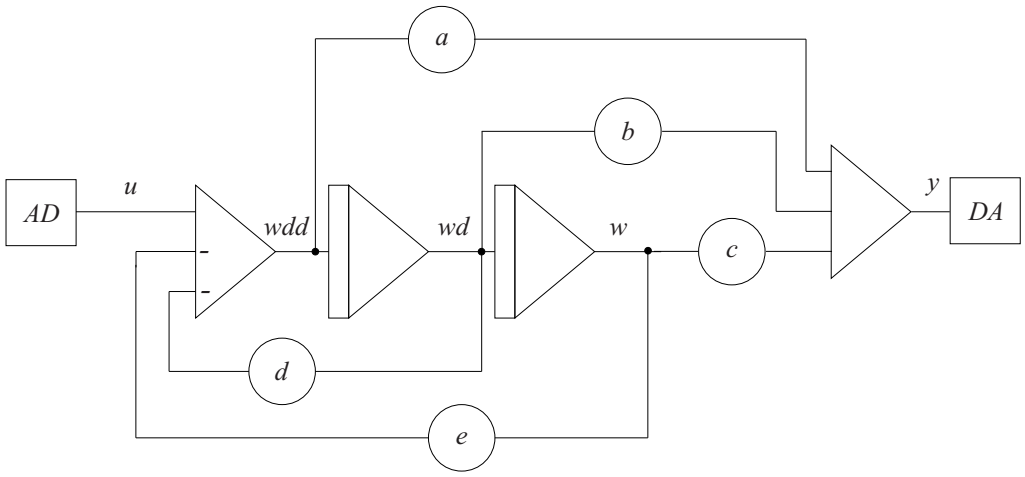
s konstantami

$$\begin{aligned} a &= 0.03162 \\ b &= 0 \\ c &= 6242 \\ d &= 110 \\ e &= 6242 \end{aligned}$$

Filter želimo realizirati na digitalnem računalniku, zato signal, ki ga je potrebno filtrirati, vzorčimo z A/D pretvornikom, filtrirani signal pa vračamo iz računalnika v obliki analognega signala na D/A pretvorniku. Ustrezno simulacijsko shemo prikazuje slika 5.59.

Program za realizacijo filtra v jeziku FORTRAN je naslednji:

```
10  a=0.03162
11  c=6242
12  d=110
13  e=6242
14  w=0
15  wd=0
16  t=0
17  dt=0.008
20  % sinhronizacija z realnim časom (dt)
30  %   u ... iz A/D pretvornika
40  wdd=u-d*wd-e*w
50  y=a*wdd+c*w
```



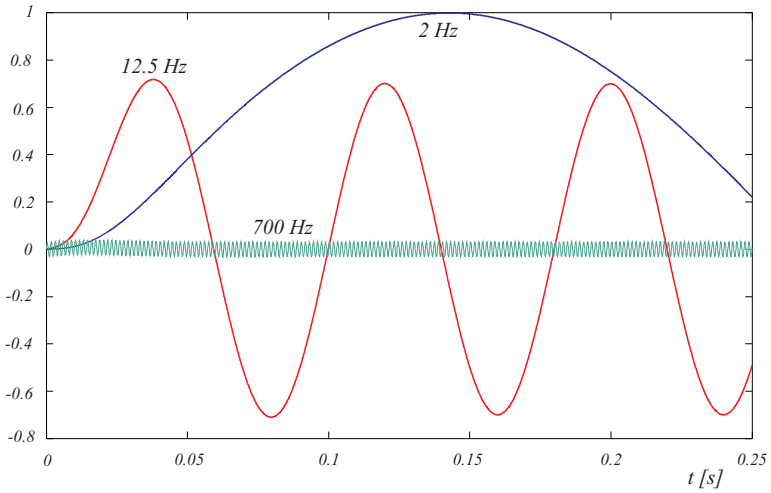
Slika 5.59: Shema za simulacijo (realizacijo) Chebyshev -ega filtra

```
 60  %   y ... na D/A pretvornik
 70  t=t+dt
 80  w=w+wd*dt
 90  wd=wd+wdd*dt
100  goto 20
110  end
```

V vrsticah 10 do 17 so definirane vse konstante prenosne funkcije, začetni vrednosti integratorjev, začetni čas in velikost računskega koraka. Izbrali smo $8ms$, saj ima filter mejno frekvenco $12.5Hz$, torej je perioda najvišjih frekvenc, ki jih prepušča, približno $80ms$. Torej smo dosegli približno 10 računskih korakov na eno periodo. Vse vrstice med stavkoma 20 in 100 so v neskončni zanki, saj mora filter delovati poljubno dolgo v realnem času. Zato je potrebno vsak nov prehod po zanki natančno sinhronizirati z realnim časom, kar naj bi zagotovil programski modul v vrstici 20 (prehodi na 8ms). V vrstici 30 se izvrši A/D pretvorba vhodnega signala, vrstice od 40 do 90 pa simulirajo (realizirajo) filter z uporabo Eulerjeve integracijske metode. Pri enačbah je zelo pomemben vrstni red. Najprej se mora izračunati `wdd` in šele nato `y`. Odvoda `wd` ni potrebno izračunavati, saj je to hkrati stanje, ki ga izračunava integracijska metoda. Zaradi zaporedno povezanih integratorjev je tudi važen vrstni red obeh enačb integratorjev. Najprej je potrebno iz trenutne vrednosti `wd` izračunati `w` za naslednji računski korak in šele nato `wd` za naslednji računski korak iz trenutne vrednosti spremenljivke `wdd`. Programski moduli za A/D in D/A pretvorbo ter za sinhronizacijo z realnim časom so odvisni od specifične materialne opreme, zato so v zgornjem programu le ustrezni komentarski stavki. Procesor računalnika mora biti zadosti hiter, da

v času 8ms izvrši vse operacije v zanki. Vprašljiva pa je numerična stabilnost in natančnost Eulerjeve metode.

Slika 5.60 prikazuje, kako filter prepušča frekvenco $2Hz$ (nizka frekvenca v prevajalnem delu), mejno frekvenco $12.5Hz$ (dušenje $3db$ oz. upad amplitude iz 1 na 0.71) in frekvenco $700Hz$ (zaporni del, dušenje približno $30db$, oz. upad amplitude iz 1 na 0.032). Amplitude vhodnih signalov so bile 1.



Slika 5.60: Izhodni signali Chebyshevega filtra

□

5.7.2 Modularna izvedba simulacijskega programa

V dosedanjih primerih smo celoten program realizirali v enem programskem modulu. Splošnonamenski programski jeziki (Matlab, Visual Basic, C++, Fortran, Java script, Java, ASP, Phyton, Visual Studio, itd.) pa so dobro strukturirani jeziki in omogočajo bolj pregledno in modularno realizacijo koncepta digitalnih simulacijskih jezikov, ki ga prikazujeta sliki 3.30 in 3.31.

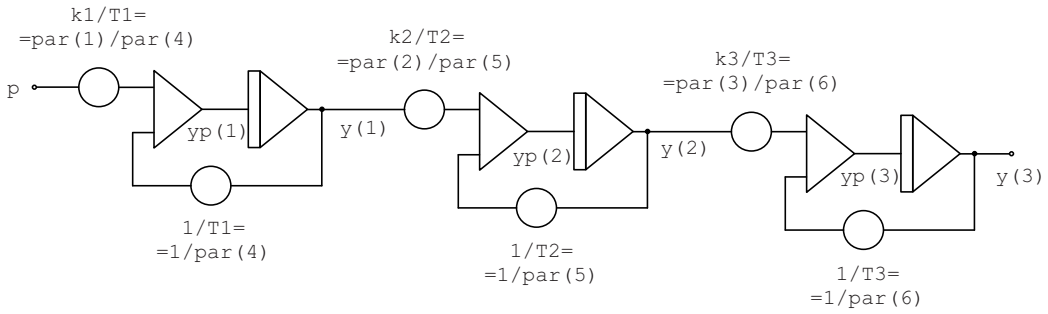
V primeru Eulerjeve integracijske metode je simulacija nekakšen dvokoračni postopek, ko v vsakem računskem koraku enkrat izvedemo enačbe za izračun odvodov spremenljivk stanj (`DERIV`) in integracijo (`INTEG`) - slika 3.28. Do sedaj smo vse izvedli v enem programskem modulu. Vendar je program dosti bolj pregleden, če za omenjene operacije uporabimo podprogramske strukture.

Pri razumevanju koncepta digitalne simulacije prikaz rezultatov sicer ni tako pomemben, mora pa biti rešen v vsakem simulacijskem programu. Zahteve za prikaz programiramo v modulu `OUTPUT`, ki je uvrščen za modulom `DERIV` in pred modulom `INTEG` (ki izračuna stanja za naslednji računski korak).

V primeru zahtevnejših integracijskih metod simulacija ni več dvokoračna iteracija modulov `DERIV` in `INTEG`, ampak se morajo enačbe modela večkrat izvajati v enem računskem koraku. V tem primeru uporabimo koncept po sliki 3.31. Program sestoji iz glavnega programa, ki poskrbi za vse potrebne inicializacije (npr. čitanje konstant modela) in za klic osrednje operacije, t.j. integracijskega podprograma (`INTEG`). V tem podprogramu je realizirana simulacijska zanka, ki poteka od začetne vrednosti neodvisne spremenljivke simulacije do izpolnitve pogoja za končanje simulacije. Vmes pa podprogram `INTEG` kliče podprogram za izračun odvodov spremenljivk stanja (`DERIV` - t.j. podprogram, ki vsebuje enačbe modela) in v zahtevanih trenutkih podprogram za komunikacijo z uporabnikom (`OUTPUT` - podprogram za prikaz rezultatov). Čeprav bomo v zvezi s podprogramom `DERIV` običajno vedno govorili kot o podprogramu za izračun odvodov (v povezavi z zapisom v prostoru stanj so to enačbe stanj $\dot{\boldsymbol{x}} = \boldsymbol{Ax}+\boldsymbol{Bu}$), pa se v tem podprogramu ovrednotijo tudi razne druge spremenljivke kot funkcije spremenljivk stanja. Gre za enačbe, ki za sam integracijski postopek niso pomembne (v povezavi z zapisom sistema v prostoru stanj so to izhodne enačbe ($\boldsymbol{y} = \boldsymbol{Cx} + \boldsymbol{Du}$). Slednje bi bilo možno vključiti tudi v podprogram `OUTPUT`, kar bi povečalo hitrost simulacije zlasti pri kompleksnejših integracijskih postopkih.

Primer 5.19 Modularna simulacija segrevanja kovinske palice

Model ogrevanja kovinske palice (slika 5.56) je uveden v primeru 5.17. Simulacijsko shemo, ki nam je v pomoč pri pisanju modularnega programa, prikazuje slika 5.61.



Slika 5.61: Simulacijska shema za simulacijo ogrevanja kovinske palice

Glede na opisani koncept uporabimo glavni program in programske module `INTEG`, `DERIV` in `OUTPUT`. Uporabili smo sicer ponovno Eulerjevo integracijsko metodo, ki pa jo je v tako strukturiranem programu možno enostavno zamenjati z zahtevnejšo metodo. Krmilne parametre simulacije smo združili v vektor `krm`, parametre modela v vektor `par`, stanja v vektor `y`, odvode stanj pa v vektor `yp`. Vhodni parametri v integracijsko metodo `INTEG` so začetna stanja, krmilni parametri in parametri modela. Integracijska metoda vsebuje simulacijsko zanko in po izstopu vrne rezultate simulacije (časovne trenutke in stanja). Funkcija `DERIV` ob klicu iz trenutnih stanj izračuna odvode, funkcija `OUTPUT` pa na zaslon izpiše trenutno vrednost neodvisne spremenljivke in ustrezna stanja.

Glavni simulacijski program

```
%glavni simulacijski program
clear;
%krmilni parametri simulacije
dt=1;       % računski korak
Tmax=200;   %dolžina sim. teka
krm=[Tmax,dt];

%začetna stanja
y=[0 0 0];

%parametri modela
k1=10;
k2=0.5;
k3=0.5;
T1=10;
T2=20;
T3=20;
par=[k1 k2 k3 T1 T2 T3];

%klic integracijske metode
[cas,yy]=INTEG(y,par,krm);

%Prikaz rezultatov po simulacijskem teku
plot(cas,yy)
```

Podprogram za integracijo (Eulerjeva metoda)

```
%Eulerjeva integracijska metoda - INTEG.M
function[cas,yy]=INTEG(y,par,krm)

%čas    časovni vektor
%yy     rezultati
%par    parametri modela
%krm    krmilni parametri simulacije
%krm(1) dolžina sim. teka
%krm(2) računski korak
i=1;

%simulacijska zanka
for t=0:krm(2):krm(1);

    %  izračun odvodov
        yp=DERIV(t,y,par);

    %  prikaz rezultatov med sim. tekom
        [izh]= OUTPUT(t,y);

    %  shranjevanje rezultatov
        yy(i,:)=y;
        cas(i)=t;

    %  integracija po enokoracni EULER-jevi metodi
    %  (izracun stanj za naslednji rač. korak)

        y=y+yp*krm(2);
        i=i+1;

end
```

Podprogram z enačbami modela - izračun odvodov spremenljivk stanja

```
%izračun odvodov - enačbe modela - DERIV.M
function [yp]=DERIV(t,y,par)

p=1;    %vhodni signal, v splošnem je to časovna funkcija
yp(1)=par(1)/par(4)*p-1/par(4)*y(1);
```

```
yp(2)=par(2)/par(5)*y(1)-1/par(5)*y(2);
yp(3)=par(3)/par(6)*y(2)-1/par(6)*y(3);
```

Podprogram za prikaz rezultatov

```
% prikaz rezultatov med simulacijo - OUTPUT.M
function[izh]=OUTPUT(t,y)
izh=[];
%izpis časa in temperatur
t
y
```

□

Primer 5.20 Zamenjava podprograma z Eulerjevo integracijo s podprogramom za integracijo po metodi Runge-Kutta 4. reda

Ob poznavanju zgoraj navedenih konceptov je enostavno uporabiti bolj uveljavljene integracijske metode (glej pogl. 6). Metoda Runge-Kutta 4. reda je opisana v pogl. 6.1.3. Ob upoštevanju enačb 6.15 in 6.16 razvijemo naslednji integracijski podprogram v jeziku Matlab:

```
% Integracijska metoda Runge-Kutta 2. reda - INTEG.M
function[cas,yy]=INTEG(y,par,krm)

%cas    časovni vektor
%yy     rezultati
%par    parametri modela
%krm    krmilni parametri simulacije
%krm(1) dolžina sim. teka
%krm(2) računski korak
i=1;

%simulacijska zanka
for t=0:krm(2):krm(1);
```

```
    %   izračun koeficientov k1,k2,k3 in k4
        k1=krm(2)*DERIV(t,y,par);
        k2=krm(2)*DERIV(t+0.5*krm(2),y+0.5*k1,par);
        k3=krm(2)*DERIV(t+0.5*krm(2),y+0.5*k2,par);
        k4=krm(2)*DERIV(t+krm(2),y+k3,par);

    %  prikaz rezultatov med sim. tekom
        [izh]= OUTPUT(t,y);

    %  shranjevanje rezultatov
        yy(i,:)=y;
        cas(i)=t;

    %  integracija po enokoračni RK4 metodi
    %  (izračun stanj za naslednji rač. korak)

        y=y+1/6*k1+1/3*k2+1/3*k3+1/6*k4;
        i=i+1;
end
```

□

Simulacija da natančnejše rezultate, ki pa se skoraj ne razlikujejo od rezultatov, ki smo jih dobili z Eulerjevo integracijsko metodo (glej sliko 5.58).

5.7.3 Nekateri problemi pri opisanih izvedbah

Algebrajska zanka

Razvrstitev stavkov (blokov), ki opisujejo model pa ni vedno možna. To se dogodi, kadar imamo v simulacijski shemi zanke brez t.i. spominskih blokov (blokov z zakasnitvenim atributom, npr. integratorjev). Vrstic v obliki

```
X2=X1+X2+...
```

ali

```
X1=X2+ ...
X2=X1+ ...
```

ni možno razvrstiti. Takim strukturam pravimo “algebrajske zanke", modele, ki jih vsebujejo, pa je možno simulirati le s posebnimi metodami (poglavje 6.3). Vendar so algebrajske zanke mnogokrat posledica slabega modeliranja.

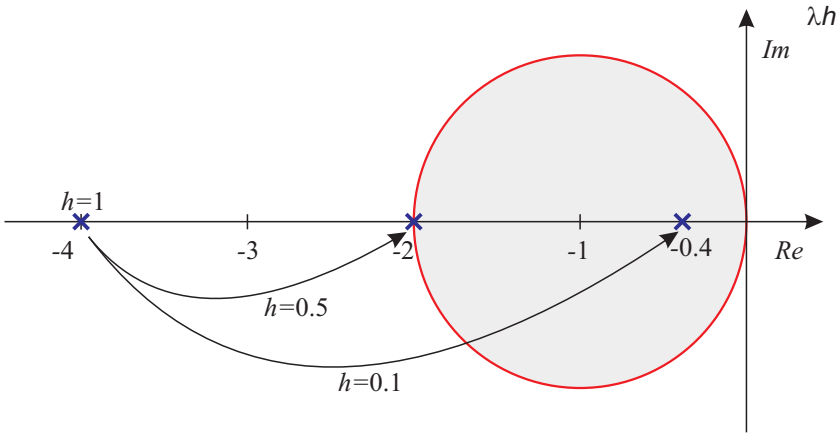
Učinkovito eksperimentiranje

Pri eksperimentiranju z nekim simulacijskim modelom lahko pričakujemo, da bo potrebno večkrat izvesti simulacijski tek z različnimi parametri (npr. različni začetni pogoji, konstante modela,...). V jeziku Matlab, ki je običajno interpreterskega tipa, to spreminjanje ni problematično, saj le spremenimo ustrezne programske vrstice in takoj spet izvedemo simulacijo. V primeru prevajalniških jezikov (npr. FORTRAN) pa je potrebno po vsaki spremembi programa le tega ponovno prevajati in povezati s knjižnicami, kar je lahko (predvsem na manjših računalnikih) kar zamudno. Zato v tem primeru običajno vse simulacijske parametre zberemo v datoteki, ki jo mora simulacijski program prebrati na začetku med operacijami pred simulacijskim tekom. Med simulacijskimi teki lahko to datoteko enostavno editiramo.

Numerična stabilnost Eulerjeve metode

Digitalna simulacija se izvaja z diskretizirano integracijo, kar pomeni, da naravno zvezni sistem diskretiziramo. Zaradi tega se zgodi, da zvezni stabilni sistem postane zaradi neustreznosti numerične integracije nestabilen. Poli oz. lastne vrednosti stabilnih sistemov ležijo v levi polravnini ravnine s oz. ravnine λ. Zaradi diskretizacije se področje stabilnosti zmanjša. Boljše integracijske metode imajo širša področja stabilnosti.

Slika 5.62 prikazuje področje numerične stabilnosti za Eulerjevo integracijsko metodo v kompleksni ravnini λh. λ je lastna vrednost oz. pol, h pa je velikost računskega koraka.



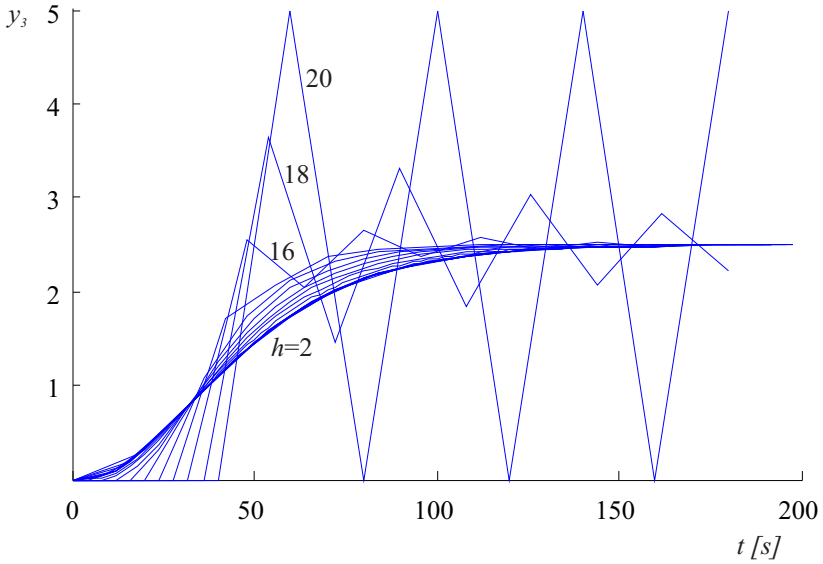
Slika 5.62: Področje numerične stabilnosti Eulerjeve metode

Metoda je stabilna znotraj kroga. V kolikor lastna vrednost oz. pol pomnožen z računskim korakom leži zunaj kroga, lahko problem rešimo z zmanjšanjem računskega koraka. Če simuliramo prenosno funkcijo

$$G(s) = \frac{4}{s+4} \tag{5.51}$$

potem je pri računskem koraku $h = 1$ vrednost $\lambda h = 4$ in simulacija je nestabilna. Če zmanjšamo računski korak na $h = 0.5$, potem smo dosegli mejo stabilnosti. S $h = 0.1$ pa dosežemo $\lambda h = 0.4$ in s tem stabilno pa tudi zadovoljivo natančno delovanje.

Model kovinske palice (primer 5.17) ima časovne kontante $T_1 = 10s$, $T_2 = 20s$ in $T_3 = 20s$ oz. pole pri $s_1 = 0.1$, $s_2 = 0.05$ in $s_3 = 0.05$ oz. lastne vrednosti pri $\lambda_1 = 0.1$, $\lambda_2 = 0.05$ in $\lambda_3 = 0.05$. Glede numerične stabilnosti so kritične krajše časovne konstante oz. večje vrednosti polov ali lastnih vrednosti. Če bi pri Eulerjevi integracijski metodi izbrali računski korak $h = 20$, bi pri lastni vrednosti $\lambda = 0.1$ veljalo $\lambda h = 2$ in simulacija bi bila na meji numerične stabilnosti. Računski korak mora biti torej $h < 20$, da je sistem stabilen, za ustrezno natančnost pa še še precej manjši, npr. $h = 1$. Slika 5.63 prikazuje vpliv računskega koraka na natančnost rezultatov. Računski korak smo spreminjali od $h = 2$ do $h = 20$ s korakom 2. Iz slike tudi ugotovimo, da manjšanje računskega koraka pod vrednost $h = 2$ ni več smiselna, saj so rezultati približno enaki.



Slika 5.63: Vpliv računskega koraka na numerično stabilnost pri simulaciji ogrevanja kovinske palice

5.7.4 Problematika pri vključevanju naprednih integracijskih metod iz knjižnic

Priporočljivo je, da za integracijo uporabljamo profesionalne in komercialno dostopne podprograme. Le-ti uporabljajo numerično bolj stabilne in natančnejše integracijske metode (npr. Runge-Kutta, Adams-Bashforth, Gear-stiff,...,glej pogl. **??**), ki običajno vključujejo tudi postopke, ki s prilagajanjem velikosti računskega koraka ali s prilagajanjem same metode sproti nadzirajo absolutni ali relativni pogrešek med simulacijo. Pri uporabi takih podprogramov, ki jih običajno nimamo v izvorni obliki, pa se srečamo z naslednjimi problemi:

1. Podprogram za integracijo mora dobiti podatek o številu stanj (integratorjev) v našem problemu.
2. Podprogram za integracijo mora imeti povezavo z imeni spremenljivk, ki nastopajo v modelu.
3. Podprogram za integracijo mora dobiti podatek o izpolnjenem pogoju za končanje simulacijskega teka.
4. Podprogram za integracijo mora vedeti za imeni podprogramov za ovrednotenje odvodov spremenljivk stanj in za izpis rezultatov.

Prva dva problema rešimo tako, da vse vhode integratorjev združimo v polje odvodov, vse izhode integratorjev pa v polje stanj. Ti polji in število integratorjev (število elementov polj) so parametri za prenos v integracijski podprogram. S tem, da integracijski podprogram uporablja polji poljubnih dimenzij, postane univerzalno uporaben. Zato pa je treba tudi vse enačbe modela zapisati z elementi polj.

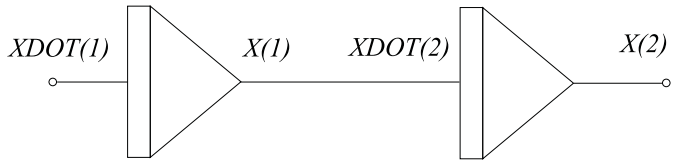
Tretji problem rešimo tako, da pogoj za končanje simulacije ni v integracijskem podprogramu ampak v enem od podprogramov, ki ju napiše uporabnik, t.j. ali v podprogramu za ovrednotenje odvodov ali pa v podprogramu za izpis rezultatov. Eden od teh programov torej določi konec simulacijskega teka in to s posebno zastavico (flag) sporoči integracijskemu podprogramu, da le ta dejansko konča s simulacijsko zanko. Ker se pri zahtevnejših integracijskih metodah podprogram za izpis rezultatov kliče precej redkeje kot podprogram za ovrednotenje odvodov, je zaradi hitrosti simulacije smiselno pogoj za končanje simulacije vgraditi v podprogram za izpis rezultatov.

Četrti problem je možno elegantno rešiti v jeziku FORTRAN. Jezik omogoča, da navedemo imena podprogramov kot parametre v klicnem stavku. Imena pa moramo navesti tudi v stavku EXTERNAL, ki se nahaja med deklarativnimi stavki glavnega programa. Pri programiranju v PASCAL-u pa moramo v izvorno integracijsko proceduro vpisati ustrezna imena, če pa nimamo izvorne kode, pa moramo za imeni procedur za ovrednotenje odvodov in izpis rezultatov uporabiti predpisani oz. v integracijski proceduri uporabljeni imeni.

Ker moramo v podprogramu za izračun odvodov po omenjenem konceptu uporabljati elemente polj namesto imen problemskih spremenljivk, tak program izgubi preglednost in ne predstavlja osnove za dobro dokumentacijo modela (npr. v PASCAL-u uporabljamo spremenljivki `x[1]` in `xdot[1]` namesto `RAB` in `RABDOT` pri simulaciji modela žrtev in roparjev). V tem primeru programski jezik FORTRAN nudi elegantno rešitev s pomočjo stavka `EQUIVALENCE`. Ta stavek (npr. `EQUIVALENCE (X(1), RAB))` zagotovi, da imajo navedene spremenljivke isto lokacijo, torej je vseeno, katero uporabljamo (ali `X(1))` ali `RAB)`. Ustrezne `EQUIVALENCE` stavke definiramo v vseh programskih modulih, ki jih napiše uporabnik (v glavnem programu, v podprogramu za izračun odvodov, v podprogramu za izpis rezultatov). V teh modulih nato operiramo s problemskimi imeni, medtem ko integracijski podprogram, do katerega nimamo dostopa (predpostavljamo, da je to profesionalni podprogram, za katerega običajno niti nimamo izvorne kode ali pa ga ne želimo spreminjati) operira z ustreznimi elementi polj oz. z vektorjema stanj in odvodov. Pri prenosu spremenljivk v podprograme pa je potrebno upoštevati, da spremenljivke v stavkih `EQUIVALENCE` ne smejo biti

med klicnimi parametri podprograma.

Pri takem modularnem programiranju je treba paziti, da v primeru, če je nek signal v shemi hkrati stanje in odvod, le tega označimo z dvema spremenljivkama, kar prikazuje slika 5.64.



Slika 5.64: Označevanje spremenljivk pri zaporedni povezavi integratorjev

Primer 5.21 Simulacija ekološkega sistema žrtev in roparjev v jeziku FORTRAN

Postopek simulacije bomo začeli z integracijskim podprogramom in čeprav bomo nakazali koncept pri uporabi profesionalnih programov, bomo zaradi kontinuitete in večje ilustrativnosti ostali pri Eulerjevi integracijski metodi. Programski jezik FORTRAN ne pozna globalnih spremenljivk, zato poteka prenos običajno preko klicnih parametrov podprogramov (subroutine ali function) včasih pa preko spremenljivk, ki jih navedemo v stavku `COMMON`. Minimalni nabor podatkov, ki ga običajno zahteva nek profesionalni integracijski podprogram, je: začetni čas simulacije, velikost računskega koraka, spremenljivka kot zastavica (flag) za končanje simulacije (vse tri spremenljivke so običajno združene v vektorju (polju)), vektor stanj (izhodov integratorjev), ki jih integracijski podprogram izračunava med simulacijo, vektor odvodov spremenljivk stanja, dolžina obeh vektorjev oz. število stanj oz. integratorjev ter imeni podprogramov za izračun odvodov in izpis rezultatov. Enostavni integracijski podprogram je naslednji:

```
      SUBROUTINE INTEG(PRMT,X,XDOT,NDIM,MODEL,RESULTS)
      DIMENSION PRMT(1),X(1),XDOT(1)
      EXTERNAL MODEL, RESULTS
      T=PRMT(1)
10    CALL MODEL(T)
      CALL RESULTS(T,PRMT)
      IF(PRMT(3).NE.0) RETURN
      T=T+PRMT(2)
      DO 20 I=1,NDIM
```

```
20    X(I)=X(I)+XDOT(I)*PRMT(2)
      GO TO 10
      END
```

Pomen klicnih parametrov podprograma je naslednji:

`PRMT` - polje z dimenzijo tri ali več z naslednjimi parametri:
`PRMT(1)` - začetna vrednost neodvisne spremenljivke
`PRMT(2)` - velikost računskega koraka
`PRMT(3)` - zastavica za končanje simulacijskega teka
Vsi trije parametri morajo biti definirani ob klicu integracijskega podprograma. `PRMT(3)` je na začetku enak nič, ko pa je izpolnjen pogoj za končanje simulacijskega teka, ga podprogram za prikaz rezultatov postavi na vrednost različno od nič

`X` - polje stanj (izhodov integratorjev). Ob klicu podprograma `INTEG` mora vsebovati začetne vrednosti stanj

`XDOT` - polje odvodov spremenljivk stanja (vhodov v integratorje)

`NDIM` - število stanj (integratorjev)

`MODEL` - podprogram za izračun odvodov spremenljivk stanja

`RESULTS` - podprogram za prikaz rezultatov

Podprogram `INTEG` je v praksi neki profesionalni podprogram in ga zato ne želimo spreminjati. Običajno tudi nimamo izvorne kode in poznamo le pomen parametrov, ki jih je potrebno prenašati. Uporabnik pa mora seveda napisati glavni program, podprogram za izračun odvodov (predpostavimo ime `DERIV`) in podprogram za izpis rezultatov (predpostavimo ime `OUTPUT`).

Povsem ekvivalenten glavni program programu v jeziku PASCAL v primeru **??** je naslednji:

```
PROGRAM PREY_PREDATOR
DIMENSION X(2),XDOT(2),PRMT(3)
COMMON X,XDOT,A11,A12,A21,A22,TFIN,N,NCOUNT
EXTERNAL DERIV,OUTPUT
EQUIVALENCE (X(1),RAB),(X(2),FOX)
```

```
EQUIVALENCE (PRMT(2),DT)
OPEN(1,FILE='DATABASE',FORM='FORMATTED',STATUS='OLD')
READ(1,*)A11,A12,A21,A22
READ(1,*)DT,TFIN,N
READ(1,*)T,RAB,FOX
CLOSE(1)
NCOUNT=0
PRMT(1)=T
PRMT(3)=0.
CALL INTEG(PRMT,X,XDOT,2,DERIV,OUTPUT)
STOP
END
```

Ustrezni podprogram za izračun odvodov spremenljivk stanja `DERIV` pa je

```
SUBROUTINE DERIV(T)
DIMENSION X(2),XDOT(2)
COMMON X,XDOT,A11,A12,A21,A22,TFIN,N,NCOUNT
EQUIVALENCE (X(1),RAB),(X(2),FOX)
EQUIVALENCE (XDOT(1),RABDOT),(XDOT(2),FOXDOT)
RABDOT=A11*RAB-A12*RAB*FOX
FOXDOT=A21*RAB*FOX-A22*FOX
RETURN
END
```

Podprogram za izpis rezultatov `OUTPUT` pa je naslednji:

```
SUBROUTINE OUTPUT(T,PRMT)
DIMENSION PRMT(1)
DIMENSION X(2),XDOT(2)
COMMON X,XDOT,A11,A12,A21,A22,TFIN,N,NCOUNT
EQUIVALENCE (X(1),RAB),(X(2),FOX)
IF (NCOUNT.LE.0) THEN
                   WRITE(*,*)T,RAB,FOX
                   NCOUNT=N
                   ENDIF
NCOUNT=NCOUNT-1
IF(T.GE.TFIN) PRMT(3)=1.
RETURN
```

```
      END
```

Stavki `EQUIVALENCE` v podprogramih `DERIV` in `OUTPUT` omogočajo, da v teh dveh uporabniških modulih uporabljamo problemska imena (npr. `RAB, RABDOT`) hkrati pa se v integracijskem podprogramu vrši integracija z uporabo polj `X` in `XDOT` (`RAB` in `RABDOT` imata isto lokacijo kot `X(1)` in `XDOT(1)`). Konstante modela (`A11`, `A12`, `A21`, `A22`) ter nekateri krmilni parametri (čas simulacije `(TFIN)`, podatki za izpis rezultatov `(N, NCOUNT)` pa se prenašajo v podprograma `DERIV` in `OUTPUT` preko t.i. `COMMON` bloka. Vektor stanj `X` in vektor odvodov `XDOT` pa se prenašata med podprogramom `INTEG` in podprogramoma `DERIV` oz. `OUTPUT` indirektno preko glavnega programa. Med podprogramom `INTEG` in glavnim programom namreč obstoja povezava preko parametrov klicnega stavka, med glavnim programom in podprograma `DERIV` in `OUTPUT` pa preko `COMMON` bloka. Neodvisno spremenljivko `T` in vektor krmilnih parametrov `PRMT` pa prenašamo preko parametrov v klicnih stavkih. Struktura bi bila nekoliko enostavnejša, če bi se vektorja `X` in `XDOT` med vsemi moduli prenašala preko parametrov, vendar FORTRAN ne dovoljuje, da je ista spremenljivka, ki se prenaša v podprogram (`DERIV` oz. `OUTPUT`) preko parametrov hkrati tudi v `EQUIVALENCE` stavku znotraj tega podprograma. Lahko pa bi med vsemi moduli stanja in odvode prenašali preko `COMMON` bloka.

Opisana struktura je torej bolj komplicirana, kot pri primerih v jeziku Matlab. Vendar smo z njo dosegli, da uporabnik lahko uporablja nek profesionalni integracijski podprogram. Pri pisanju podprogramov `DERIV` in `OUTPUT` uporablja problemske spremenljivke, tema dvema podprogramoma pa da lahko tudi poljubno ime ne glede na ime, ki ga uporablja podprogram `INTEG`.

Pri prikazu rezultatov smo vseskozi omenjali izpis na zaslon, čeprav tudi izpis v datoteko ali grafična predstavitev med simulacijo ne spremeni koncepta. V podprogramu `OUTPUT` je potrebno le uporabiti ustrezne izhodne stavke ali klicati ustrezne podprograme. □

5.7.5 Osnovne ideje simulacije na paralelnih računalnikih

V naslednjem primeru bomo nakazali uporabnost tranputerskih paralelnih procesorskih sistemov v simulaciji. Osnovne lastnosti smo opisali v podpoglavju 4.1.2.

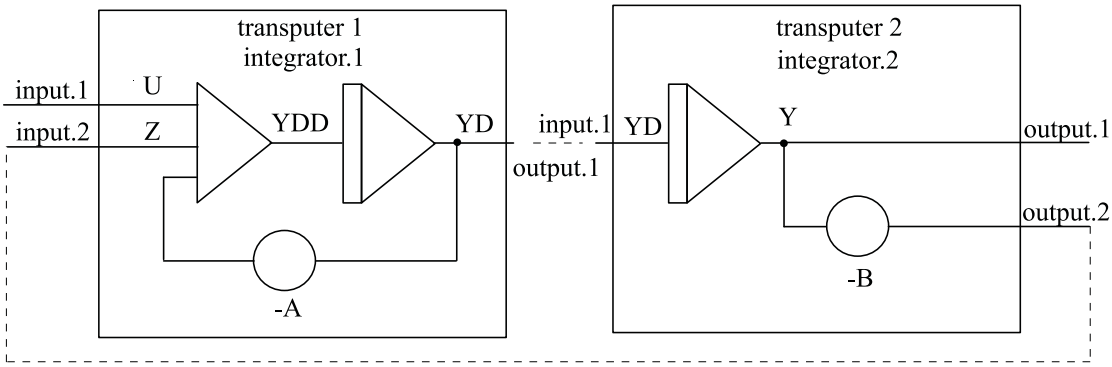
Primer 5.22 Simulacija sistema drugega reda v OCCAM-u

Simulirajmo sistem, ki ga opisuje diferencialna enačba

$$\ddot{y} + a\dot{y} + by = u \tag{5.52}$$

z ničnimi začetnimi pogoji in s konstantama $a = 1.4, b = 1$ na paralelnem transputerskem sistemu. Za integracijo uporabimo Eulerjevo metodo z računskim korakom $\Delta t = 0.01$ s.

Predpostavljamo, da ima računalnik dva transputerja. Najprej moramo določiti, kaj bo računal prvi in kaj drugi transputer. Smiselno je, da vsak transputer računa eno spremenljivko stanj. Simulacijsko shemo in razdelitev operacij prikazuje slika 5.65.



Slika 5.65: Simulacijska shema sistema 2. reda

Računalniške spremenljivke `U, Z, YDD, YD, Y, A, B, DT` smo izbrali za označitev problemskih spremenljivk $u, z, \ddot{y}, \dot{y}, y, a, b, \Delta t$. Črtkane črte predstavljajo fizične povezave med transputerjema. Prvi transputer potrebuje tri serijske priključke: dva vhoda `input.1, input.2` in en izhod `output.1`. Drugi transputer potrebuje enako število priključkov: en vhod `input.1` in dva izhoda `output.1` in `output.2`. Spremenljivki stanj `Y,YD` sta znani na začetku vsakega računskega koraka. Zato transputerja najprej postavita od stanj odvisne izhodne signale, nato sprejmeta vhodne signale, izračunata odvoda spremenljivk stanj (`YD, YDD`) in nato izračunata z Eulerjevo integracijsko metodo (enačba (3.71)) stanji (`Y, YD`) za naslednji računski korak. Postopek izmenjavanja podatkov in izračunavanja se nato ponavlja v vsakem računskem koraku. Vsak transputer razen podatkov, ki jih sprejme preko serijske povezave, potrebuje tudi lokalne spremenljivke, ki jih hrani v lastnem pomnilniku.

Slika 5.66 prikazuje ustrezni program v jeziku OCCAM. Program sestoji iz dveh t.i. procesov (`PROC integrator.1` in `PROC integrator.2`), ki se izvajata par-

```
PROC integrator.1 (CHAN input.1, input.2, output.1)
  VAL A IS      1.4 (REAL32):
  VAL DT IS    0.01 (REAL32):

  REAL32 U,YDD,YD,Z,DT:

  SEQ
   YD:=0.0(REAL32)
   WHILE TRUE
     SEQ
       PAR
         output.1!YD
         input.1?U
         input.2?Z
       YDD:=(U+Z)-(A*YD)
       YD:=YD+(DT*YDD)

PROC integrator.2 (CHAN input.1, output.1, output.2)
  VAL B IS   1.0(REAL32):
  VAL DT IS  0.01(REAL32):

  REAL32 YD, Y, Z, DT

  SEQ
   Y:=0.0(REAL32)
   WHILE TRUE
     SEQ
       Z:-B*Y
       output.2!Z
       PAR
          output.1!Y
          input.1?YD
        Y:=Y+(DT*YD)
```

Slika 5.66: Program v jeziku OCCAM

alelno. Znotraj vsakega procesa nastopajo sekvenčni bloki (`SEQ` - izvajanje operacij druga za drugo) in paralelni bloki (`PAR` - poljubni vrstni red operacij znotraj takih blokov)). Stavek `PROC` določa fizične povezave. Nato so definirane konstante in deklaracije spremenljivk. V prvem sekvenčnem bloku je podan začetni pogoj in

izvedena simulacijska zanka (s pomočjo `WHILE` stavka). Z naslednjim sekvenčnim blokom, ki je vgnezden v prvega (torej v simulacijsko zanko), pa izvajamo operacije v vsakem računskem koraku. Izračunavanja, ki se lahko izvajajo paralelno (v poljubnem vrstnem redu), smo navedli v `PAR` bloku. Vrstice s klicajem pošiljajo podatke preko serijske povezave, vrstice z vprašajem pa sprejemajo podatke, pri čemer čakajo na veljavnost podatkov na povezavah (s tem je zagotovljeno tudi usklajeno delovanje obeh transputerjev). Ko se v paralelnih blokih pridobijo vsi potrebni podatki, se v obeh procesih nadaljuje izvajanje sekvenčnih blokov, v katerih se po Eulerjevi integracijski metodi izračunajo stanja za naslednji računski korak. Konce blokov ne določajo stavki `end`, ampak ustrezni zamiki pri pisanju vrstic.

V tem primeru nismo nič govorili o vhodni spremenljivki `u`. Dobili bi jo lahko tako, da bi vzorčili nek realni signal z A/D pretvornikom, ki bi ga povezali na transputer. □

6.

Numerični postopki v simulaciji

Numerične integracijske metode predstavljajo srce vsakega digitalnega simulacijskega sistema. Za reševanje numerično nezahtevnih problemov ne potrebujejo uporabniki praktično nobenega znanja o integracijskih postopkih. Pri numerično zahtevnih problemih pa je zelo pomembno poznavanje osnovnih lastnosti integracijskih metod in še predvsem, kaj so prednosti in slabosti posameznih postopkov. Pregled integracijskih postopkov je namenjen tistim, ki uporabljajo simulacijo za praktično reševanje problemov in ne strokovnjakom s področja numeričnih metod. Opisujemo pa tudi problem integracije preko nezveznosti v signalih.

Poseben numerični problem predstavlja algebrajska zanka. V simulaciji jo redko srečamo, ponavadi pa nastane zaradi neustreznega modeliranja. Pogosto se pojavi zlasti neizkušenim modelerjem in programerjem. Zato podajamo tudi osnove numeričnega reševanja algebrajske zanke.

Z osnovnim znanjem o integracijskih metodah ter o reševanju algebrajske zanke bo dobil uporabnik bolj zanesljive rezultate, prihranil pa bo tudi veliko računalniškega časa.

6.1 Numerične integracijske metode

Simulacijska orodja, ki so nam na voljo, nudijo bogat izbor različnih integracijskih metod. Običajno pa uporabnik ne ve, katera metoda je za njegov problem najprimernejša. Včasih pa sploh ne ve, da obstajajo različne metode, saj uporablja kar privzeto (default) metodo (običajno metoda Runge - Kutta 4. reda). Le-ta dobro deluje za ne preveč zahtevne probleme. Razlog je seveda jasen: uporabnik ne pozna osnovnih lastnosti, ki vplivajo na izbiro metode. Zato ignorira možnost izbire in čez čas celo pozabi na to možnost. Idealna bi bila seveda metoda, ki bi reševala vse numerične probleme z enako učinkovitostjo. Toda pričakovanja, da bi nekdo razvil tako metodo, so nestvarna. Veliko obetajo le t.i. ekspertni sistemi, ki bi uporabniku pomagali izbrati optimalno metodo za simulacijo njegovega problema.

Uporabnik, ki pa si želi pridobiti bolj poglobljeno znanje, naj uporabi kakšno od naslednjih referenc: (Hairer, Wanner, 1990), (Gustaffson, 1990),
(Hairer, Lubich, 1988), (Gustaffson, 1988), (Press in ostali, 1986), (Korn, Wait, 1978), (Hall, Watt, 1976), (Lapidus, Seinfeld, 1971), (Gear, 1971).

6.1.1 Splošna oblika numeričnega integracijskega algoritma

Začenši od poznega osemnajstega stoletja, ko je Euler razvil svoj algoritem, je numerična integracija za reševanje navadnih diferencialnih enačb predstavljala pomembno področje numerične matematike. Mi se bomo osredotočili predvsem na metode, ki jih je možno uporabiti za širši spekter problemov in pa na metode, ki imajo sicer manjšo praktično vrednost, a so zaradi enostavnosti razumljivejše in tako pedagoško učinkovitejše.

Ker je model vsakega realnega sistema skoraj vedno zapisan z diferencialnimi enačbami višjih redov, je potrebno te enačbe transformirati (ročno ali avtomatsko) v sistem enačb prvega reda z ustreznim definiranjem novih spremenljivk. Tako lahko reševanje predstavimo kot začetnovrednostni problem z vektorsko diferencialno enačbo

$$\dot{\boldsymbol{x}} = \boldsymbol{f}(t, \boldsymbol{x}) \tag{6.1}$$

in začetnim pogojem $\boldsymbol{x}(t_0) = \boldsymbol{x}_0$, kjer je $\boldsymbol{x}$ vektor stanj in $\boldsymbol{f}$ odvodna funkcija. Pri

simulaciji je potrebno enačbo (6.1) integrirati od začetnega časa t_0 do končnega časa t_{max}. Ob uporabi numerične integracijske metode moramo celotni čas simulacije razdeliti na ustrezno število računskih korakov. Predpostavimo, da je računski korak h konstanten, tako je potrebno izračunati rešitev v trenutkih $t_k = t_0 + k \cdot h$, $k = 0, 1, 2, \ldots, k_{max}$. Točna rešitev v trenutku t_{k+1} je

$$\boldsymbol{x}(t_{k+1}) = \boldsymbol{x}(t_0) + \int_{t_0}^{t_{k+1}} \boldsymbol{f}(t, \boldsymbol{x})dt = \boldsymbol{x}(t_k) + \int_{t_k}^{t_{k+1}} \boldsymbol{f}(t, \boldsymbol{x})dt \tag{6.2}$$

Z diskretizacijo enačbe (6.2) dobimo izraz

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \boldsymbol{I}_k \tag{6.3}$$

kjer je $\boldsymbol{I}_k$ približna vrednost integrala

$$\boldsymbol{I}_k \approx \int_{t_k}^{t_{k+1}} \boldsymbol{f}(t, \boldsymbol{x})dt \tag{6.4}$$

Zaradi te aproksimacije je $\boldsymbol{x}_{k+1}$ samo približek prave vrednosti $\boldsymbol{x}(t_{k+1})$. V vsakem računskem koraku torej nastane lokalna napaka, ki se lahko med simulacijo tudi akumulira. Akumulirano napako imenujemo globalna napaka.

Znani so številni numerični algoritmi, ki rešijo enačbo (6.4). Delimo jih v

- enokoračne metode (implicitne in eksplicitne),
- večkoračne metode (implicitne in eksplicitne),
- ekstrapolacijske metode in
- metode za simulacijo togih sistemov.

6.1.2 Vrste numeričnih integracijskih napak

Če postopek integracije izvedemo z numeričnim algoritmom na digitalnem računalniku, se pojavita dve vrsti napak:

- *napaka zaradi končnega reda numerične metode* in
- *napaka zaradi končne dolžine besede* (oz. omejene natančnosti), s katero deluje aritmetika določene programske opreme na računalniku.

Napaka numerične metode

Napaka numerične metode (numerical approximation or truncation error) je torej pogojena z omejeno natančnostjo integracijskega postopka in ni odvisna od natančnosti računalnika oz. njegove aritmetike. Če bi bila aritmetika računalnika povsem točna, bi določena integracijska metoda povzročila v enem računskem koraku *lokalno napako*

$$\begin{aligned} \boldsymbol{e}_{k+1} &= \boldsymbol{x}_{k+1} - \boldsymbol{x}(t_{k+1}) = \\ &= \boldsymbol{x}_{k+1} - \left(\boldsymbol{x}_k + \int_{t_k}^{t_{k+1}} \boldsymbol{f}(t, \boldsymbol{x}) dt \right) = \\ &= \boldsymbol{I}_k - \int_{t_k}^{t_{k+1}} \boldsymbol{f}(t, \boldsymbol{x}) dt \end{aligned} \tag{6.5}$$

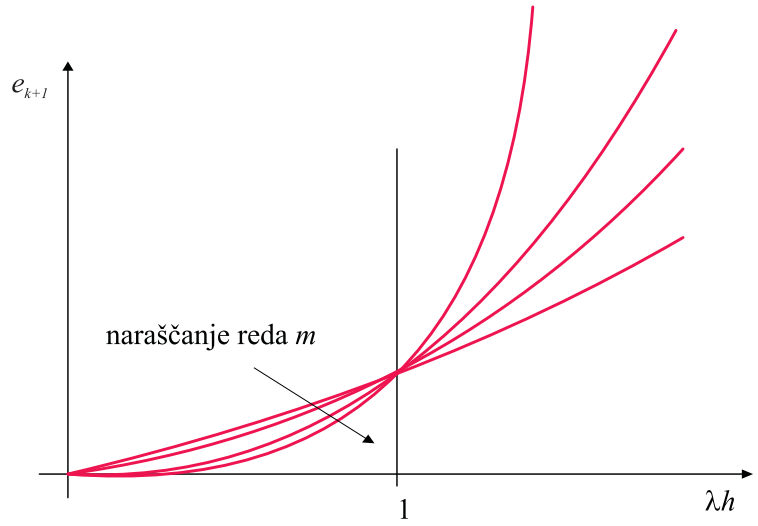
rezultirajoče vrednosti $\boldsymbol{x}_{k+1}$, pri čemer je t_k začetni trenutek opazovanja. Predpostavljamo, da imamo v tem trenutku točen rezultat. Često lahko $\boldsymbol{e}_{k+1}$ ocenjujemo med potekom simulacije. Vendar pa je za vrednotenje rezultata pomembnejša *globalna napaka*

$$\boldsymbol{e}_{T,k+1} = \boldsymbol{x}_{k+1} - \left(\boldsymbol{x}_0 + \int_{t_0}^{t_{k+1}} \boldsymbol{f}(t, \boldsymbol{x}) dt \right) \tag{6.6}$$

Pri tem je $t = t_0$ začetni čas simulacije. Običajno je globalna napaka večja od lokalne napake. V določenih primerih lahko postane neskončna. Toda globalne napake ni možno oceniti med simulacijo. Omejena lokalna napaka tudi ne zagotavlja omejene globalne napake. Vendar predstavlja lokalna napaka edino možnost za nadzor točnosti med simulacijo. Za večino integracijskih metod velja, da je lokalna napaka zaradi numerične metode sorazmerna $m + 1$ potenci računskega koraka

$$\boldsymbol{e}_{k+1} \propto h^{m+1} \tag{6.7}$$

kjer je h računski korak in m red integracijske metode. Torej je potrebno za natančnejše rezultate zmanjšati računski korak ali pa povečati red integracijske metode (glej sliko 6.1).



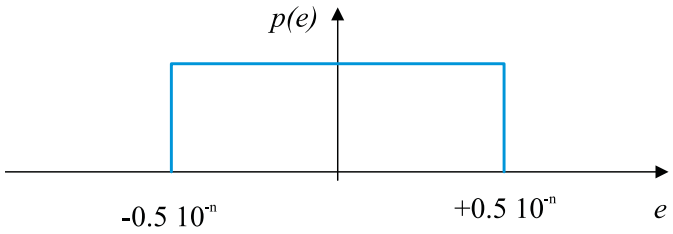
Slika 6.1: Vpliv reda metode na napako numerične metode

Napaka zaradi končne dolžine besede

Numerični integracijski postopki se seveda realizirajo s pomočjo digitalnega računalnika, ki ima zaradi omejene dolžine besede aritmetike, ki jo uporablja simulacijsko orodje, tudi omejeno natančnost. To vodi do napake, ki jo bomo imenovali napaka zaradi končne dolžine besede (roundoff error). Ta vrsta napake je zelo podvržena akumuliranju. Odvisna je

- od časa integriranja (oz. od dolžine simulacijskega teka) $t_{max} - t_0$,
- od reda integracijske metode m (zaradi zahtevnejšega izračunavanja),
- narašča pa tudi pri manjšanju računskega koraka h, saj manjši h pomeni, da imamo več računskih korakov v času integracije $t_{max} - t_0$.

Napako zaradi končne dolžine besede je težko natančno izračunati. Če računalniška aritmetika zaokrožuje rezultate, lahko predpostavimo, da so napake enakomerno (uniformno) porazdeljene med plus in minus $\frac{1}{2}10^{-n}$, kjer je n število

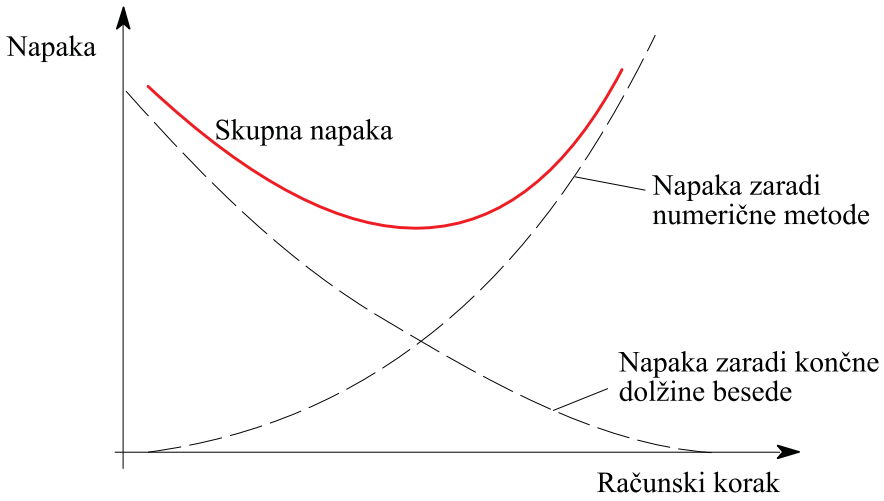


Slika 6.2: Enakomerna porazdelitev napake zaradi končne dolžine besede

decimalnih digitov (glej sliko 6.2). Ob tej predpostavki ocenimo napako zaradi končne dolžine besede z izrazom (Korn, Wait, 1978)

$$\boldsymbol{e}_{T,k+1} \approx \frac{10^{-n}}{2} \sqrt[2]{\frac{m \cdot (t_{max} - t_0)}{12 \cdot h}} \tag{6.8}$$

Slika 6.3 prikazuje vpliv računskega koraka h na obe vrsti obravnavanih napak ter na skupno napako. Opazimo, da obstaja optimalna dolžina računskega koraka h_{opt}, pri katerem ima skupna napaka minimalno vrednost. Vrednost h_{opt} je seveda težko določiti, saj je odvisna od sistema diferencialnih enačb ($\boldsymbol{f}(t, \boldsymbol{x})$), od integracijske metode in od uporabljenega računalnika oz. njegove aritmetike.



Slika 6.3: Vpliv računskega koraka na celotno napako

6.1.3 Enokoračne integracijske metode

Enokoračne metode so numerični postopki, v katerih se rešitev v trenutku t_{k+1} oceni iz vrednosti le enega predhodnega trenutka t_k. V splošnem dobimo algoritem z razvojem enačbe (6.2) v Taylorjevo vrsto v okolici točke $\boldsymbol{x}(t_k)$

$$\boldsymbol{x}(t_k + h) = \boldsymbol{x}(t_k) + h\dot{\boldsymbol{x}}(t_k) + \frac{h^2}{2!}\ddot{\boldsymbol{x}}(t_k) + \frac{h^3}{3!}\dddot{\boldsymbol{x}}(t_k) + \cdots \tag{6.9}$$

Enokoračna metoda je m - tega reda, če uporabimo $m + 1$ členov v Taylorjevi vrsti. Ker člene višjih redov zanemarimo, lahko ocenimo lokalno napako z izrazom

$$\boldsymbol{e}(t_k) \approx \frac{h^{m+1}}{(m+1)!}\boldsymbol{x}^{(m+1)}(t_k) \tag{6.10}$$

Če uporabimo le dva člena Taylorjeve vrste, dobimo najpreprostejšo enokoračno metodo - Eulerjevo metodo

$$\boldsymbol{x}(t_{k+1}) \approx \boldsymbol{x}(t_k) + h\dot{\boldsymbol{x}}(t_k) \tag{6.11}$$

oziroma

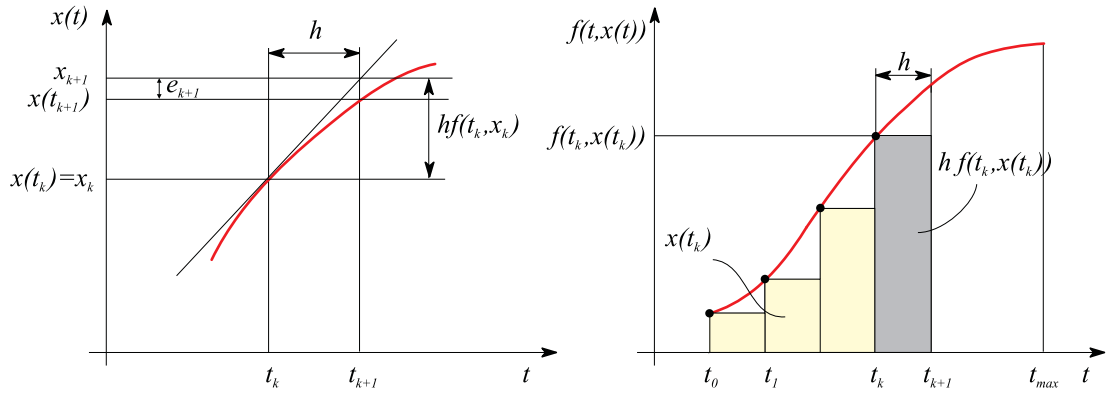
$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + h\boldsymbol{f}(t_k, \boldsymbol{x}_k) \tag{6.12}$$

Enačba (6.12) predstavlja eksplicitno Eulerjevo metodo. Le-to lahko zelo jasno predstavimo s sliko 6.4.

Ker je $\boldsymbol{x}(t_k) = \boldsymbol{x}_k$, predpostavimo, da je rezultat točen v trenutku t_k, oz. da trenutek t_k predstavlja začetni čas integracije. Zaradi lokalne napake numerične metode pa je vrednost $\boldsymbol{x}_{k+1}$ le ocena, ki bolj ali manj odstopa od prave vrednosti $\boldsymbol{x}(t_{k+1})$.

Eulerjev integracijski postopek je intuitivno zelo razumljiv in zahteva izračun le enega odvoda v računskem koraku. Vendar daje sprejemljivo točne rezultate običajno le pri dovolj majhnem računskem koraku h. Manjši računski korak pa seveda pomeni

- večjo porabo računalniškega časa



Slika 6.4: Grafična predstavitev Eulerjeve integracijske metode

- pa tudi večjo napako zaradi končne dolžine besede.

Zato resnejše simulacijske študije zahtevajo numerično bolj izpopolnjene postopke.

Boljšo natančnost seveda dosežemo z uporabo več členov Taylorjeve vrste. Toda to nas vodi do potrebe po ovrednotenju višjih odvodov, kar pa ni enostavno. Runge je prvi pokazal, kako se je v Taylorjevi vrsti možno znebiti višjih odvodov, ne da bi pri tem poslabšali natančnost. Del , ki v Taylorjevi vrsti vsebuje višje odvode, je nadomestil z delom, ki vsebuje nedoločene koeficiente in prve odvode, t.j. funkcijo $\boldsymbol{f}(t, \boldsymbol{x})$ v več točkah intervala med $(t_k, \boldsymbol{x}_k)$ in $(t_{k+1}, \boldsymbol{x}_{k+1})$. Na ta način lahko zapišemo splošno obliko enokoračne metode

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \sum_{i=1}^{v} w_i \boldsymbol{k}_i \tag{6.13}$$

kjer so w_i utežnostni koeficienti, ki jih je potrebno izračunati, v je število izračunov odvodne funkcije v enem računskem koraku, $\boldsymbol{k}_i$ pa določimo iz eksplicitne oblike

$$\boldsymbol{k}_i = h\boldsymbol{f}(t_k + c_i h, \boldsymbol{x}_k + \sum_{j=1}^{i-1} a_{ij} \boldsymbol{k}_j) \qquad i = 1, 2, \cdots, v \tag{6.14}$$

pri čemer je $\boldsymbol{k}_i$ potrebno računati rekurzivno in so c_i in a_{ij} ustrezni koeficienti. Takim algoritmom pravimo tudi eksplicitne enokoračne metode, znane tudi pod

imenom metode Runge - Kutta. Red metode m ni direktno razviden iz enačb (6.13) in (6.14) vendar je enak redu prototipne Taylorjeve vrste.

Če izpeljemo enačbi (6.13) in (6.14) za določen red m, je možno dobiti večje število algoritmov z ustrezno izbiro nekaterih prostih parametrov. Za

- rede $1 \leq m \leq 4$ je število potrebnih izračunanih odvodov na računski korak v kar enako redu metode m,
- za $m > 4$ pa je število potrebnih izračunov vedno večje kot je red.

Postopki za izpeljavo koeficientov c_i, a_{ij} in w_i so opisani v literaturi (Gear, 1971 ali Press in ostali, 1986). Na tem mestu bomo navedli predstavitev enokoračnih metod v skrčeni ali t.i. Butcherjevi obliki (Butcher, 1964). To je nekakšna matrična oblika, ki vsebuje koeficiente enačb (6.13) in (6.14):

$$\begin{array}{c|cccc|c} 0 & & & & & w_1 \\ c_2 & a_{21} & & & & w_2 \\ c_3 & a_{31} & a_{32} & & & w_3 \\ \vdots & \vdots & \vdots & \ddots & & \vdots \\ c_v & a_{v1} & a_{v2} & \cdots & a_{vv-1} & w_v \end{array} \qquad \boldsymbol{c} \mid \boldsymbol{A} \mid \boldsymbol{w}$$

Tipične eksplicitne enokoračne metode (Runge-Kutta) so: Euler (1,1), izboljšana Eulerjeva metoda (2,2), Heun (2,2), Nystrom (3,3), Heun (3,3), klasična metoda Runge-Kutta (4,4), England (4,4), Runge-Kutta-Fehlberg (4,5), Runge-Kutta-Fehlberg (5,6). Številke v oklepajih predstavljajo dvojico (m, v), kjer je (m) red in (v) število izračunov odvodne funkcije v enem računskem koraku. Koeficienti klasične metode Runge - Kutta so prikazani v tabeli 6.1.

Tabela 6.1: Koeficienti klasične metode Runge - Kutta

$$\begin{array}{c|cccc|c} 0 & & & & & \frac{1}{6} \\ \frac{1}{2} & \frac{1}{2} & & & & \frac{1}{3} \\ \frac{1}{2} & 0 & \frac{1}{2} & & & \frac{1}{3} \\ 1 & 0 & 0 & 1 & & \frac{1}{6} \end{array}$$

Z uporabo tabele 6.1 lahko napišemo algoritem v obliki

$$\begin{aligned} \boldsymbol{x}_{k+1} &= \boldsymbol{x}_k + \frac{1}{6}\boldsymbol{k}_1 + \frac{1}{3}\boldsymbol{k}_2 + \frac{1}{3}\boldsymbol{k}_3 + \frac{1}{6}\boldsymbol{k}_4 && (6.15)\\ \boldsymbol{k}_1 &= h\boldsymbol{f}(t_k, \boldsymbol{x}_k) \\ \boldsymbol{k}_2 &= h\boldsymbol{f}(t_k + \frac{1}{2}h, \boldsymbol{x}_k + \frac{1}{2}\boldsymbol{k}_1) \\ \boldsymbol{k}_3 &= h\boldsymbol{f}(t_k + \frac{1}{2}h, \boldsymbol{x}_k + \frac{1}{2}\boldsymbol{k}_2) && (6.16)\\ \boldsymbol{k}_4 &= h\boldsymbol{f}(t_k + h, \boldsymbol{x}_k + \boldsymbol{k}_3) \end{aligned}$$

Razen eksplicitnih enokoračnih metod pa so zelo znane tudi implicitne enokoračne metode. Ustrezne izraze dobimo iz enačb (6.13) in (6.14), če zgornjo mejo vsote v enačbi (6.14) spremenimo iz $i-1$ v v

$$\begin{aligned} \boldsymbol{x}_{k+1} &= \boldsymbol{x}_k + \sum_{i=1}^{v} w_i \boldsymbol{k}_i && (6.17)\\ \boldsymbol{k}_i &= h\boldsymbol{f}(t_k + c_i h, \boldsymbol{x}_k + \sum_{j=1}^{v} a_{ij}\boldsymbol{k}_j) \qquad i = 1, 2, \cdots, v && (6.18) \end{aligned}$$

Matrika $\boldsymbol{A}$ v Butcherjevi obliki ni več trikotniška, ampak ima elemente tudi nad diagonalo. Glavni problem pa je seveda v tem, ker je koeficient $\boldsymbol{k}_i$ odvisen od koeficientov $\boldsymbol{k}_j$ (za $j = 1, 2, \cdots, v$) tako, da ga ni možno računati z navadnim rekurzivnim postopkom. V tem je glavna težava implicitnih postopkov, saj je potreben poseben iterativni algoritem za izračun $\boldsymbol{k}_i$. Vendar pa v primerjavi z eksplicitnimi metodami dosežemo

- višje rede pri enakem številu izračunanih odvodov, kar pomeni tudi manjšo lokalno napako in
- boljšo numerično stabilnost.

Predstavniki teh metod so naslednji: Gauss (2,1), Gauss (4,2), Radau (5,3), Milne (4,3) in Lobatto (6,4).

Polimplicitne metode so nekaj vmesnega med eksplicitnimi in implicitnimi. S temi metodami skušamo obdržati dobre lastnosti obeh prej opisanih postopkov,

- predvsem dobro numerično stabilnost implicitnih metod in
- računsko učinkovitost eksplicitnih metod (ker ni potrebno iterativno računanje).

Vendar pa je namesto iteracij potrebno računati Jacobijevo matriko

$$\boldsymbol{J} = \frac{\partial \boldsymbol{f}(t, \boldsymbol{x})}{\partial \mathbf{x}} \tag{6.19}$$

in njeno inverzno vrednost za izračun koeficienta $\boldsymbol{k}_i$. Kljub temu pa to zahteva precej manj računalniškega časa kot implicitne iteracije, čeprav je treba v vsakem računskem koraku v krat izračunati Jacobijevo matriko in njeno inverzno vrednost. Zelo znani predstavniki polimplicitnih metod so Rosenbrock-ove metode.

Ocena lokalne napake

Vemo, da je lokalna napaka zaradi numerične metode sorazmerna izrazu h^{m+1}, kjer je h velikost računskega koraka in m red metode. Za zanesljive rezultate je potrebno to napako med simulacijo izračunavati. Najenostavneje to napako ocenimo tako, da izvršimo proces integracije na intervalu $2h$ z dvema različnima računskima korakoma: h in $2h$. Iz razlike lahko ocenimo lokalno napako numeričnega postopka. Če s to oceno izboljšamo rešitev v vsakem računskem koraku, potem seveda izboljšamo tudi celotno rešitev. Pri metodi Runge - Kutta četrtega reda je ocena lokalne napake

$$\boldsymbol{e}_{k,h} \approx \frac{1}{15}(\boldsymbol{x}_{k,h} - \boldsymbol{x}_{k,2h}) \tag{6.20}$$

kjer sta $\boldsymbol{x}_{k,h}$ in $\boldsymbol{x}_{k,2h}$ rešitvi, ki smo jih dobili z računskima korakoma h in $2h$. Izboljšana rešitev je torej

$$\boldsymbol{x}^*_{k,h} = \boldsymbol{x}_{k,h} + \boldsymbol{e}_{k,h} = \frac{1}{15}(16\boldsymbol{x}_{k,h} - \boldsymbol{x}_{k,2h}) \tag{6.21}$$

Opisani postopek pa seveda zahteva precej dodatnega izračunavanja.

Drugi način za ocenitev lokalne napake temelji na osnovi razlike integralov, ki jih izračunamo na računskem koraku z metodama reda m in $m-1$. Pri tem ni potrebno povsem ločeno izvajati oba postopka, ampak so nekateri vmesni rezultati uporabni v obeh metodah. Zelo znani metodi, ki na ta način preverjata napako, sta metodi Runge-Kutta-Merson in Runge-Kutta-Fehlberg. Pri slednji je potrebno za ovrednotenje integrala po metodi petega reda izračunati le en dodatni odvod glede na metodo četrtega reda.

Stabilnost enokoračnih metod

Znano je, da je zvezni dinamični sistem $\dot{x} = f(t, x(t))$ stabilen, če imajo lastne vrednosti njegove Jacobijeve matrike

$$\boldsymbol{J} = \frac{\partial \boldsymbol{f}(t, \boldsymbol{x})}{\partial \boldsymbol{x}} \tag{6.22}$$

ki jih dobimo z rešitvijo enačbe

$$det(\lambda_i \boldsymbol{I} - \boldsymbol{J}) = 0 \tag{6.23}$$

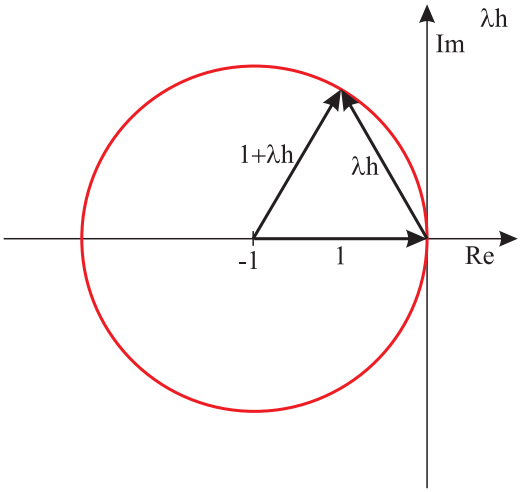
negativne realne dele. Numerični integracijski postopek pa pretvori sistem zapisan z diferencialnimi enačbami v sistem zapisan z diferenčnimi enačbami. Slednji lahko postane nestabilen tudi v primeru, če je originalni zvezni sistem stabilen. Nestabilnost pa pomeni, da se lahko majhne numerične napake med simulacijo izdatno ojačujejo. Poglejmo, kaj predstavlja uporaba Eulerjevega algoritma na enostavni diferencialni enačbi sistema prvega reda:

$$\begin{aligned} \frac{dx}{dt} &= f(t,x) = \lambda x \\ x_{k+1} &= x_k + hf(t_k, x_k) = x_k + h\lambda x_k = x_k(1+\lambda h) \end{aligned} \tag{6.24}$$

Ker je λ lastna vrednost zveznega sistema, je sistem stabilen za $\lambda < 0$, saj rešitev diferencialne enačbe $x(t) = x(0)e^{-\lambda t}$ upada. Rešitev diferenčne enačbe $x_k = x(0)(1+\lambda h)^k$ pa upada, če velja

$$|1 + \lambda h| < 1 \tag{6.25}$$

Enačba 6.25 predstavlja enotin krog s središčem v točki -1 ravnine λh (slika 6.5). Notranjost tega kroga predstavlja stabilno področje Eulerjeve integracijske metode (glej krivuljo za $m = 1$ na sliki 6.6). Vidimo, da lahko dosežemo stabilnost z dovolj majhno vrednostjo računskega koraka h.

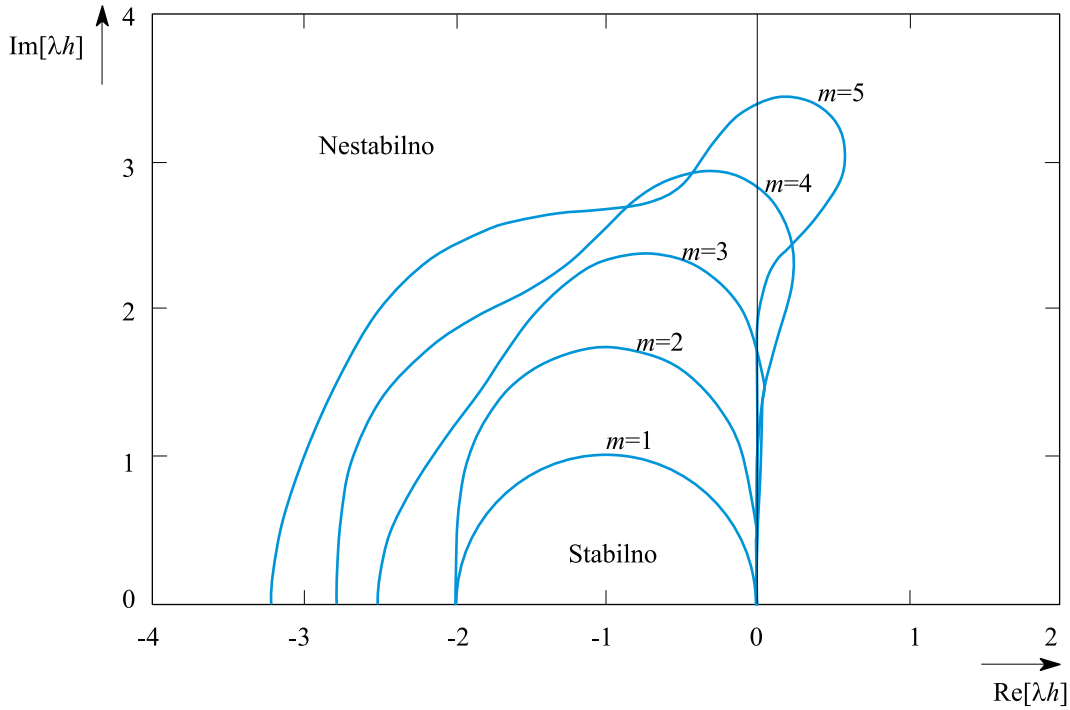


Slika 6.5: Stabilnostno področje Eulerjeve metode

Enake omejitve veljajo pri nelinearnih sistemih, če so le ti opisani s sistemom diferencialnih enačb prvega reda. Za razliko od linearnih sistemov pa so lastne vrednosti Jacobijeve matrike pri nelinearnih sistemih časovno spremenljive, tako da je lahko integracijski algoritem v določenem področju stabilen, v določenem področju pa nestabilen.

Podobno kot za Eulerjevo metodo določimo stabilnostna področja v ravnini λh tudi pri eksplicitnih metodah Runge - Kutta. Slika 6.6 prikazuje stabilnostna področja za Eulerjevo metodo ter za različne metode tipa Runge - Kutta (redi 2–5). Tako na tej kot na naslednjih slikah podajamo le zgornji del λh ravnine, saj so krivulje simetrične glede na realno os. Vse metode so stabilne znotraj zaključenih krivulj.

Čeprav metode Runge-Kutta višjih redov povečajo natančnost, pa se velikost stabilnostnih področij bistveno ne spremeni. Lahko zaključimo, da so metode Runge-Kutta stabilne, če izbrani računski korak h približno zadovoljuje neenačbo



Slika 6.6: Stabilnostna področja za Eulerjevo metodo (m=1) in za metode Runge-Kutta (m=2,3,4,5)

$$|\lambda_{max}|h < 3 \tag{6.26}$$

kjer je λ_{max} maksimalna lastna vrednost Jacobijeve matrike. Ker je λ_{max} obratno sorazmerna minimalni časovni konstanti T_{min} simuliranega sistema, velja tudi naslednja ocena stabilnosti:

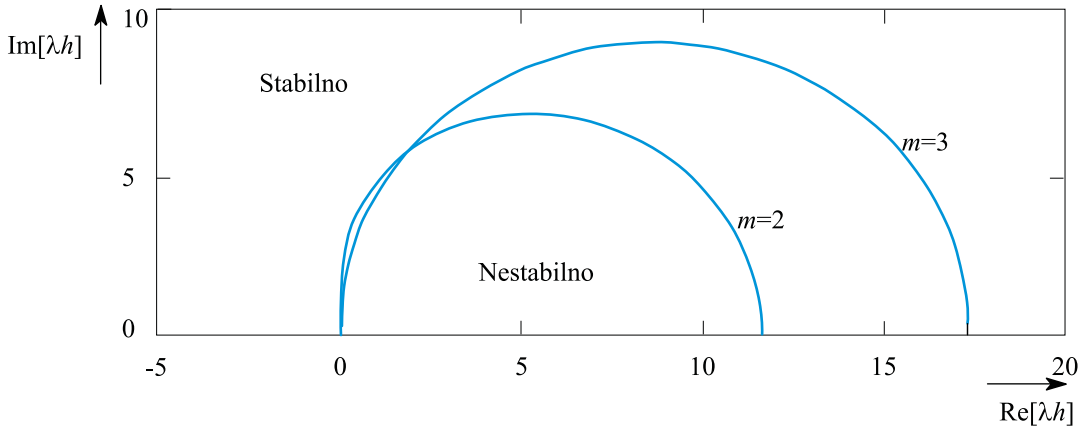
$$h < 3T_{min} \tag{6.27}$$

Toda enačba 6.27 predstavlja le stabilnostni pogoj. Sicer je dobro znano pravilo, da mora biti računski korak nekajkrat manjši od najmanjše časovne konstante, da dosežemo zadovoljivo natančnost (glej sliko 6.3).

Implicitne enokoračne metode imajo večja stabilnostna področja. Le-ta določimo podobno kot pri eksplicitnih metodah. Gaussove metode prvega, drugega (trapezoidno pravilo) in tretjega reda so stabilne v celotnem levem delu ravnine λh.

Metodi Radau in Lobatto imata manjši stabilnostni področji toda vseeno večji, kot pri eksplicitnih metodah ($|\lambda_{max}|h < 5$).

Dobre stabilnostne lastnosti odlikujejo tudi polimplicitne enokoračne metode. Slika 6.7 prikazuje stabilnostna področja Rosenbrock-ovih polimplicitnih metod drugega in tretjega reda. Metodi sta nestabilni znotraj zaključenih krivulj.



Slika 6.7: Stabilnostna področja Rosenbrock-ovih metod (drugega in tretjega reda)

Implicitne in polimplicitne metode omogočajo predvsem z vidika numerične stabilnosti relativno velike računske korake. Toda računalniški čas, ki ga prihranimo z velikim računskim korakom, delno plačamo z iterativnim postopkom za reševanje implicitnih odvisnosti ali za izračun Jacobijeve matrike in njene inverzne vrednosti v primeru polimplicitnih metod. Metode pa so predvsem učinkovite pri simulaciji sistemov z zelo različnimi časovnimi konstantami (togi (stiff) sistemi).

6.1.4 Večkoračne integracijske metode

Enokoračne metode so predvsem pri višjih redih računsko precej potratne, saj zahtevajo večje število izračunov odvodne funkcije v enem računskem koraku. Računsko manj potratne metode je možno razviti, če upoštevamo rezultate več predhodnih izračunov in ne le zadnjega, kot je to primer pri enokoračnih metodah. Tako dobimo večkoračne postopke, pri katerih potrebujemo bistveno manjše število izračunov odvodne funkcije in s tem precej pridobimo na hitrosti.

Večkoračna integracijska metoda ($p+1$ koračna, potrebuje $p+1$ preteklih vred-

nosti) je definirana z enačbo

$$\begin{aligned} \boldsymbol{x}_{k+1} &= a_0\boldsymbol{x}_k + a_1\boldsymbol{x}_{k-1} + \cdots + a_p\boldsymbol{x}_{k-p} + \\ &+ h[b_{-1}\boldsymbol{f}(t_{k+1}, \boldsymbol{x}_{k+1}) + \cdots + b_p\boldsymbol{f}(t_{k-p}, \boldsymbol{x}_{k-p})] = \\ &= \sum_{i=0}^{p} a_i\boldsymbol{x}_{k-i} + h\sum_{i=-1}^{p} b_i\boldsymbol{f}(t_{k-i}, \boldsymbol{x}_{k-i}) \end{aligned} \tag{6.28}$$

in zahteva podatke o vektorju stanj in odvodov v $p+1$ preteklih vrednostih ($p+1$ koračna metoda).

- Če je $b_{-1} = 0$ potem je metoda eksplicitna ali prediktorska, saj ne potrebuje bodoče vrednosti $\boldsymbol{f}(t_{k+1}, \boldsymbol{x}_{k+1})$.
- Za $b_{-1} \neq 0$ je metoda implicitna ali korektorska, saj je potrebno v vsakem računskem koraku izvesti iterativni postopek za rešitev enačbe (6.28).

Ker so pretekle vrednosti $\boldsymbol{x}_k$ in $\boldsymbol{f}(t_k, \boldsymbol{x}_k)$ shranjene, zahteva večkoračna metoda izračun le enega odvoda v enem računskem koraku.

Koeficienti a_i in b_i so izbrani tako, da je enačba (6.28) veljavna, če izrazimo stanje $\boldsymbol{x}$ s polinomom reda m ($m+1$ koeficientov). Na ta način je možno

- $m+1$ koeficientov od $2p+3$ v enačbi (6.28) izraziti z metodo nedoločenih koeficientov.
- Ostale koeficiente pa določimo tako, da minimiziramo napake in da dosežemo čim večja stabilnostna področja.

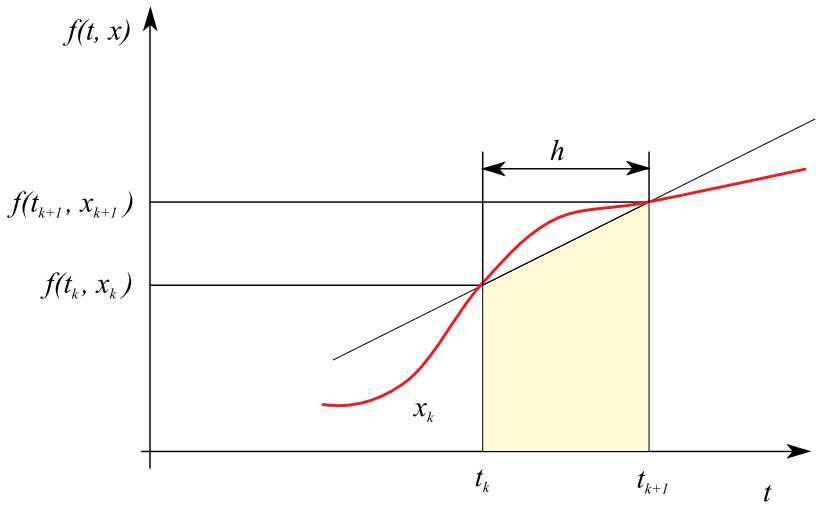
Večkoračne metode predstavljajo problem na začetku, ko pretekle vrednosti $\boldsymbol{x}_k$ in $\boldsymbol{f}(t_k, \boldsymbol{x}_k)$ še niso na voljo. Ta problem se reši tako, da se s pomočjo enokoračne metode predhodno izračunajo potrebni začetni podatki. Uporaba večkoračnih metod je problematična tudi pri nezveznostih v funkciji $\boldsymbol{x}$ ali v njenih odvodih. V takem primeru je aproksimacija (6.28) precej nenatančna, saj stanja $\boldsymbol{x}$ ni možno zadovoljivo natančno izraziti s polinomom m-tega reda. V takih primerih bi natančne rezultate dobili le tako, da bi v trenutku nastopa nezveznosti na novo sprožili večkoračno metodo.

Lokalna napaka numerične metode je odvisna od reda polinoma m, ki izraža stanje sistema in je sorazmerna h^{m+1}.

Če izberemo $p = 0$, $a_0 = 1$, $b_{-1} = \frac{1}{2}$, $b_0 = \frac{1}{2}$ dobimo zelo znano in priljubljeno trapezoidno pravilo

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \frac{h}{2}(\boldsymbol{f}(t_k, \boldsymbol{x}_k) + \boldsymbol{f}(t_{k+1}, \boldsymbol{x}_{k+1})) \tag{6.29}$$

Princip tovrstne integracije prikazuje slika 6.8.



Slika 6.8: Integracija ob uporabi trapezoidnega pravila

Integral (ploščino) odvodne funkcije na intervalu t_k, t_{k+1} aproksimiramo s ploščino trapezoida.

Ker potrebuje metoda eno predhodno vrednost odvoda $\boldsymbol{f}(t_k, \boldsymbol{x}_k)$ $(p = 0)$, je pravzaprav implicitna enokoračna metoda in kot taka poseben primer večkoračne metode. Metodo lahko učinkovito uporabljamo pri simulaciji enostavnejših problemov, saj porabi malo računskega časa in omogoča dobro numerično stabilnost.

Najbolj znane večkoračne metode so Adamsove metode. Poznamo

- eksplicitne (metode Adams-Bashforth) in

- implicitne (metode Adams-Moulton).

Adams-Bashforth-ovo metodo dobimo, če izberemo

$$p = m - 1 \qquad a_0 = 1 \quad a_1 = a_2 = a_3 = \cdots = a_{m-1} = 0 \qquad b_{-1} = 0 \tag{6.30}$$

Da dobimo metodo m-tega reda, moramo uporabiti $p + 1 = m$ predhodnih vrednosti. To je torej m koračna metoda.

Adams-Moulton-ovo metodo dobimo, če izberemo

$$p = m - 2 \qquad a_0 = 1 \quad a_1 = a_2 = a_3 = \cdots = a_{m-2} = 0 \tag{6.31}$$

Da dobimo metodo m-tega reda, moramo uporabiti le $p + 1 = m - 1$ predhodnih vrednosti. Taka metoda je $m - 1$ koračna. Vendar pa je potrebno v vsakem računskem koraku izvršiti iterativni postopek, saj je metoda implicitna. Da zagotovimo enotino ojačenje pri integraciji konstante, mora veljati

$$\sum_{i=-1}^{p} b_i = 1 \tag{6.32}$$

Tabela 6.2 prikazuje koeficiente često uporabljenih Adamsovih metod. Nekatere že znane enokoračne metode lahko obravnavamo kot posebne primere večkoračnih metod.

Metode prediktor-korektor

Pri implicitnih enokoračnih ali večkoračnih metodah predstavlja glavni problem uporaba iterativnega algoritma (npr. Newton-Raphson) pri reševanju enačb (6.17), (6.18) in (6.28). Lahko pa uporabimo eksplicitno enokoračno ali večkoračno metodo, da ocenimo $\boldsymbol{x}_{k+1}$. Oceno $\widehat{\boldsymbol{x}}_{k+1}$ nato upoštevamo pri ovrednotenju odvoda $\boldsymbol{f}(t_{k+1}, \widehat{\boldsymbol{x}}_{k+1})$, ki ga potrebuje implicitna metoda. Le-ta torej služi kot korektor za izračun vrednosti $\boldsymbol{x}_{k+1}$. Take metode so poznane pod imenom

Tabela 6.2: Koeficienti pri uporabi Adamsovih metod ($a_0 = 1$, vsi ostali $a_i = 0$)

m	b_{-1}	b_0	b_1	b_2	b_3	Ime	Št. korakov
1	0	1	0	0	0	eksplicitna Eulerjeva metoda	1
2	0	$\frac{3}{2}$	$\frac{-1}{2}$	0	0	eksplicitno trapezoidno pravilo	2
3	0	$\frac{23}{12}$	$\frac{-16}{12}$	$\frac{5}{12}$	0	Adams-Bashforth 3. reda	3
4	0	$\frac{55}{24}$	$\frac{-59}{24}$	$\frac{37}{24}$	$\frac{-9}{24}$	Adams-Bashforth 4. reda	4
1	1	0	0	0	0	implicitna Eulerjeva metoda	0
2	$\frac{1}{2}$	$\frac{1}{2}$	0	0	0	implicitno trapezoidno pravilo	1
3	$\frac{5}{12}$	$\frac{8}{12}$	$\frac{-1}{12}$	0	0	Adams-Moulton 3. reda	2
4	$\frac{9}{24}$	$\frac{19}{24}$	$\frac{-5}{24}$	$\frac{1}{24}$	0	Adams-Moulton 4. reda	3

prediktor - korektor (PC). Red korektorske metode mora biti vedno večji ali enak redu prediktorske metode.

Enostavno metodo prediktor - korektor dobimo, če uporabimo eksplicitno Nystrom-ovo metodo (pravilo srednje točke - midpoint rule) kot prediktorsko metodo in implicitno trapezoidno pravilo kot korektorsko metodo (enačba (6.29)):

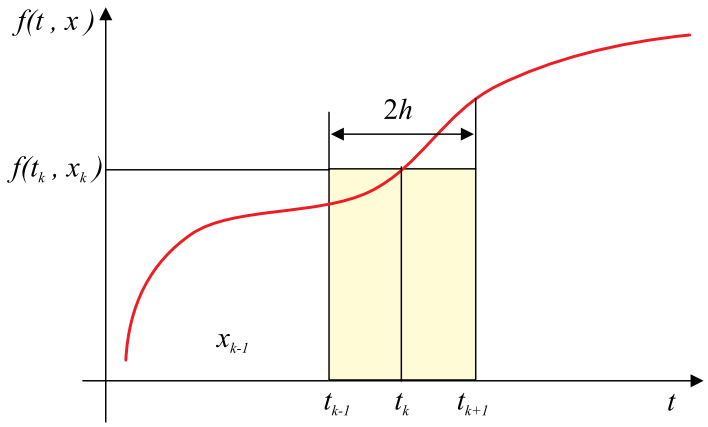
$$\widehat{\boldsymbol{x}}_{k+1} = \boldsymbol{x}_{k-1} + 2h\boldsymbol{f}(t_k, \boldsymbol{x}_k) \tag{6.33}$$

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \frac{h}{2}(\boldsymbol{f}(t_k, \boldsymbol{x}_k) + \boldsymbol{f}(t_{k+1}, \widehat{\boldsymbol{x}}_{k+1})) \tag{6.34}$$

Nystrom-ovo metodo ilustrira slika 6.9.

Vidimo, da sta potrebna dva izračuna odvodov v enem računskem koraku.

Metode prediktor - korektor imajo dobre stabilnostne lastnosti, in relativno majhno lokalno napako numerične metode. Lokalno napako lahko ocenimo iz razlike med prediktorsko vrednostjo $\widehat{\boldsymbol{x}}_{k+1}$ in korektorsko vrednostjo $\boldsymbol{x}_{k+1}$. Če ta napaka ni znotraj predpisanih toleranc, lahko korektorsko metodo iterativno uporabljamo. Vsaka iteracija zahteva en izračun odvoda in uporabo korektorske enačbe. Postopek običajno konvergira pri dovolj majhnem računskem koraku. Če ne, je



Slika 6.9: Nystrom-ova metoda, sredinsko pravilo

potrebno zmanjšati računski korak. Običajno dosežemo dovolj natančne rezultate z le nekaj iteracijami.

Pogosto v prediktor - korektor postopkih uporabljamo metode Adams-Bashforth in Adams-Moulton četrtega reda.

Ocena lokalne napake numerične metode

Postopek za oceno lokalne napake večkoračnih metod je enak kot pri enokoračnih metodah. Za metode Adams-Bashforth in Adams-Moulton četrtega reda podajata oceno napake in izboljšano vrednost rezultata naslednji enačbi:

$$\boldsymbol{e}_{k,h} \approx \frac{1}{15}(\boldsymbol{x}_{k,h} - \boldsymbol{x}_{k,2h}) \tag{6.35}$$

$$\boldsymbol{x}^*_{k,h} = \frac{1}{15}(16\boldsymbol{x}_{k,h} - \boldsymbol{x}_{k,2h}) \tag{6.36}$$

$\boldsymbol{x}_{k,h}$ je rešitev pri računskem koraku h, $\boldsymbol{x}_{k,2h}$ pa je rešitev pri uporabi računskega koraka $2h$. $\boldsymbol{x}^*_{k,h}$ je izboljšana vrednost rezultata.

Pri metodah prediktor - korektor je potrebno za ovrednotenje lokalne napake precej manj računalniškega časa, saj dobimo oceno iz razlike med rešitvijo prediktorja in korektorja. Če uporabimo Adams-Bashforth-ovo metodo četrtega reda

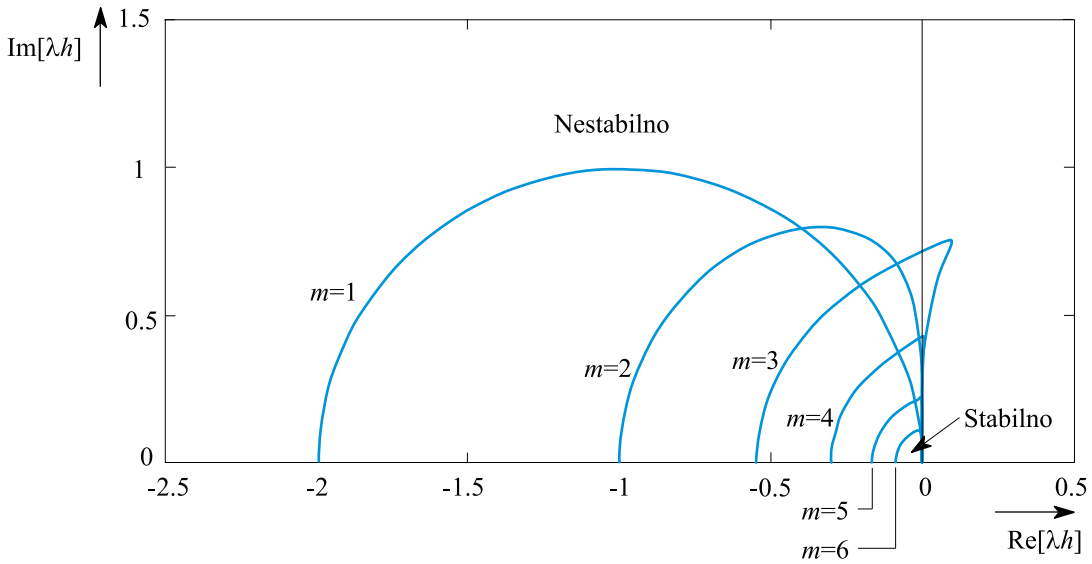
kot prediktorsko in Adams-Moulton-ovo metodo kot korektorsko, sta ocena napake in izboljšana vrednost podani z enačbama

$$\boldsymbol{e}_k \approx -\frac{1}{14}(\boldsymbol{x}_{k,C} - \boldsymbol{x}_{k,P}) \tag{6.37}$$

$$\boldsymbol{x}_k^* = \frac{1}{14}(13\boldsymbol{x}_{k,C} + \boldsymbol{x}_{k,P}) \tag{6.38}$$

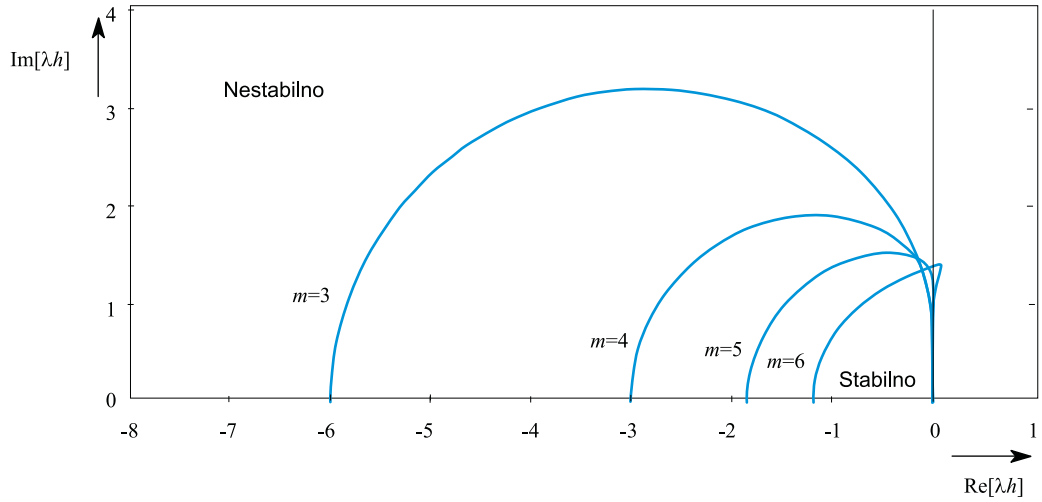
Stabilnost večkoračnih metod

Eksplicitne večkoračne metode imajo majhna stabilnostna področja v λh ravnini. Z večanjem reda metode postajajo ta področja celo manjša. Slika 6.10 prikazuje stabilnostna področja za Adams-Bashforth-ove metode. Metode so stabilne znotraj zaključenih krivulj.



Slika 6.10: Stabilnostna podpodročja metod Adams-Basforth

Implicitne večkoračne metode pa imajo precej večja stabilnostna področja, tako da omogočajo izbiro večjega računskega koraka. Implicitna Eulerjeva metoda in implicitno trapezoidno pravilo sta stabilna v celotni levi polovici ravnine λh. Slika 6.11 prikazuje stabilnostna področja Adams-Moulton-ovih metod. Le-te so stabilne znotraj zaključenih krivulj.



Slika 6.11: Stabilnostna področja metod Adams-Moulton

Stabilnostne lastnosti metod prediktor-korektor so odvisne od prediktorske in korektorske metode ter od števila korektorskih iteracij. Stabilnostna področja so večja kot pri prediktorskih metodah (eksplicitne metode) in manjša kot pri korektorskih metodah (implicitne metode).

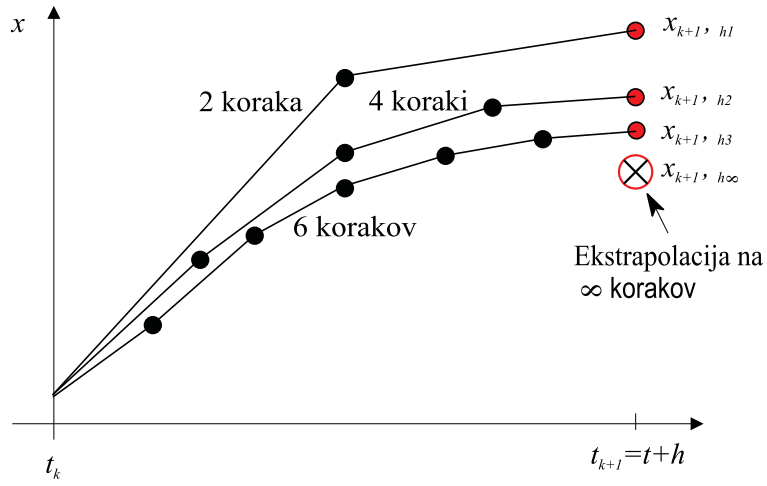
6.1.5 Ekstrapolacijske metode

V primerjavi z dosedaj obravnavanimi metodami so ekstrapolacijske metode specifične in se tudi redkeje uporabljajo. Glavna ideja tega postopka je v tem, da se integracija zaporedno vrši na vsakem računskem koraku z različno dolgimi in vedno krajšimi računskimi koraki. Označimo rezultate integracije na intervalu od t_k do t_{k+1} z $\boldsymbol{x}_{k+1,h_1}$, $\boldsymbol{x}_{k+1,h_2}$, $\boldsymbol{x}_{k+1,h_3}$, $\cdots$, $\boldsymbol{x}_{k+1,h_r}$.Postopek prikazuje slika 6.12.

Zaporedje rezultatov $\boldsymbol{x}_{k+1,h_i}$ in zaporedje računskih korakov $h_i, i = 1, 2, \cdots, r$ uporabimo za določitev analitične funkcije $\boldsymbol{x}_{k+1}(h)$. S pomočjo te odvisnosti je možno z ekstrapolacijo dobiti rešitev $\boldsymbol{x}_{k+1,h_\infty}$. Pri tem h_∞ pomeni računski korak dolžine nič, kar teoretično vodi do točne rešitve.

Običajno uporabimo integracijsko metodo nižjega reda in enega od naslednjih ekstrapolacijskih postopkov:

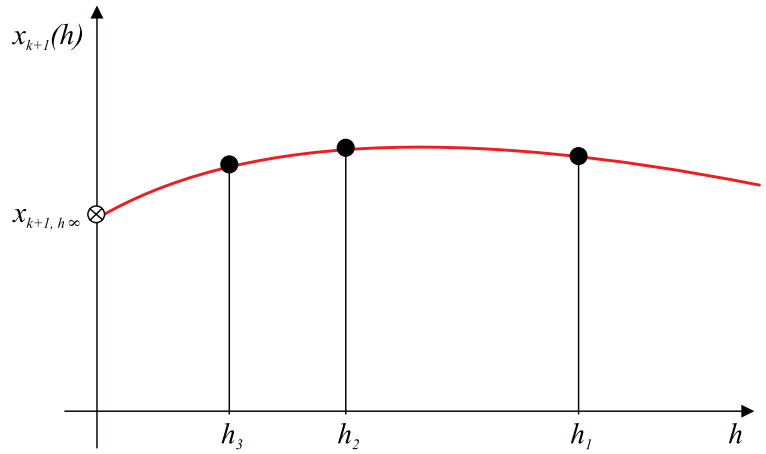
- Richardson-ovo ekstrapolacijo in



Slika 6.12: Osnovni princip ekstrapolacijske metode

- racionalno ekstrapolacijo.

Richardson-ovo ekstrapolacijo dobimo, če izrazimo analitično funkcijo $\boldsymbol{x}_{k+1}(h)$ v obliki polinoma (slika 6.13)



Slika 6.13: Richardson-ova ekstrapolacija

$$\boldsymbol{x}_{k+1}(h) = \boldsymbol{x}_{k+1,h_\infty} + \boldsymbol{\alpha}_1 h + \boldsymbol{\alpha}_2 h^2 + \cdots + \boldsymbol{\alpha}_r h^r \tag{6.39}$$

kjer so $\boldsymbol{\alpha}_i$ koeficienti. V številnih primerih pa dobimo bolj natančne rezultate, če funkcijo $\boldsymbol{x}_{k+1}(h)$ analitično izrazimo kot ulomljeno racionalno funkcijo. Taki

ekstrapolaciji pa pravimo racionalna ekstrapolacija in jo za eno komponento vektorja $\boldsymbol{x}_{k+1}(h)$ podaja enačba

$$\begin{aligned} x_{k+1}(h) &= \frac{p_0 + p_1 h + p_2 h^2 + \cdots + p_r h^r}{q_0 + q_1 h + q_2 h^2 + \cdots + q_r h^r} \\ x_{k+1,h_\infty} &= \frac{p_0}{q_0} \end{aligned} \tag{6.40}$$

in so p_i in q_i ustrezne konstante.

V ekstrapolacijskem postopku se uporabljajo različna zaporedja računskih korakov. Literatura najbolj priporoča zaporedje $\frac{h}{2}, \frac{h}{3}, \frac{h}{4}, \frac{h}{6}, \frac{h}{8}, \frac{h}{12}, \cdots$.

Celoten postopek je naslednji:

- Po vsaki integraciji na nekem računskem koraku se izvede ekstrapolacija. Postopek izračuna ekstrapolirano vrednost in oceno lokalne napake.
- Če je napaka večja od dopustne, se izvede nova integracija z manjšim računskim korakom.
- Ko napaka postane manjša od dopustne, se nadaljuje integracija na naslednjem računskem koraku.

Ena od osnovnih ekstrapolacijskih metod je Euler Romberg-ova metoda. V tej metodi se za integracijo na posameznih računskih korakih uporablja Eulerjeva metoda. Boljšo natančnost in ugodnejše stabilnostne lastnosti lahko dosežemo z uporabo trapezoidnega pravila, vendar moramo v tem primeru reševati implicitne enačbe. Obe možnosti uporabljata polinomsko ekstrapolacijo.

Zelo kvalitetne ekstrapolacijske metode so metode Bulirsch-Stoer-Gragg. Nekatere od njih uporabljajo integracijsko metodo s sredinskim pravilom in racionalno ekstrapolacijo.

Natančnost in stabilnost ekstrapolacijskih integracijskih metod je odvisna od uporabljenega integracijskega in ekstrapolacijskega algoritma. Za neko povprečno natančnost potrebujejo ekstrapolacijske metode običajno večje število izračunov odvodne funkcije kot npr. metode Runge-Kutta. Postanejo pa zelo učinkovite pri visokih zahtevah glede točnosti. Nekateri avtorji jih priporočajo tudi za simulacijo sistemov, v katerih nastopajo nezveznosti.

6.1.6 Integracijske metode za toge sisteme

Modeli mnogih fizikalnih sistemov lahko vsebujejo zelo različne lastne vrednosti oz. časovne konstante. Znani so primeri modelov

- dinamike tekočin,
- sistemov vodenja,
- električnih sistemov,
- kemičnih reakcij, itd.

Takim sistemom pravimo togi (stiff) sistemi. Dobro digitalno simulacijsko orodje mora vsebovati vsaj eno integracijsko metodo za učinkovito obravnavo tovrstnih problemov.

Za sistem pravimo, da je tog, če velja neenačba

$$\frac{\max Re[\lambda_i]}{\min Re[\lambda_i]} > 100 \tag{6.41}$$

Pri tem je $\max Re[\lambda_i]$ največja in $\min Re[\lambda_i]$ najmanjša vrednost realnih delov lastnih vrednosti Jacobijeve matrike. Togi sistemi povzročajo problematiko tako glede

- natančnosti kot
- glede numerične stabilnosti.

Pri simulaciji togih sistemov prihaja do naslednjih problemov:

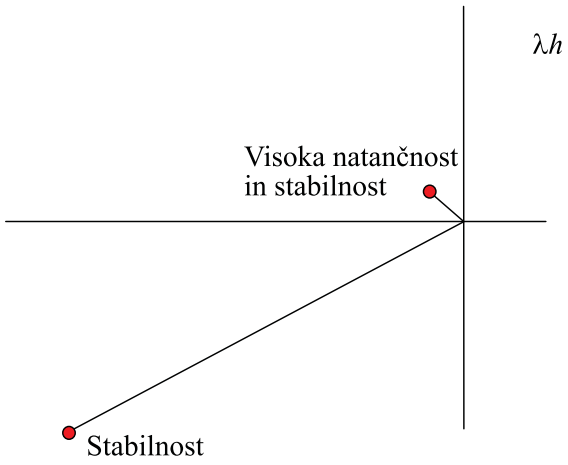
- Če uporabimo metodo z omejenim stabilnostnim področjem, (npr. metoda Runge-Kutta), bode velike vrednosti realnih delov nekaterih lastnih vrednosti povzročile, da bo potrebno za dosego numerične stabilnosti izbrati zelo majhen računski korak. Taka simulacija je s stališča potrebnega računalniškega časa zelo potratna, majhen računski korak pa pomeni tudi povečano napako zaradi končne natančnosti računalnika oz. dolžine besede njegove aritmetike.

- Če pa uporabimo metodo, ki je stabilna na celotnem levem delu ravnine λh (npr. trapezoidno pravilo), se sicer izognemo problemom stabilnosti, toda komponente, ki pripadajo velikim lastnim vrednostim, bodo pri sprejemljivi velikosti računskega koraka vnašale velike napake.
- Tudi iterativni postopki pri implicitnih integracijskih metodah so konvergentni le pri dovolj majhnih računskih korakih. Lastnosti konvergentnosti so seveda odvisne od uporabljene iterativne metode.

Zaradi ugodnih stabilnostnih lastnosti predstavljajo implicitne in polimplicitne metode osnovo integracijskih metod za toge sisteme. Čeprav so take metode računalniško potratne, pa se to poplača z možnostjo povečanja računskega koraka. Na žalost so v celotnem levem delu ravnine λh stabilne le metode nižjih redov.

Gear pa je uspel razviti povsod uveljavljene metode, ki dajejo ugodne stabilnostne lastnosti tudi za rede večje od dve. Uvedel je posebne postopke, ki zagotavljajo

- visoko natančnost in stabilnost za dominantne (majhne) lastne vrednosti in
- samo stabilnost za relativno nepomembne kratke časovne konstante oz. velike lastne vrednosti (glej sliko 6.14).



Slika 6.14: Osnovna ideja Gearovih metod

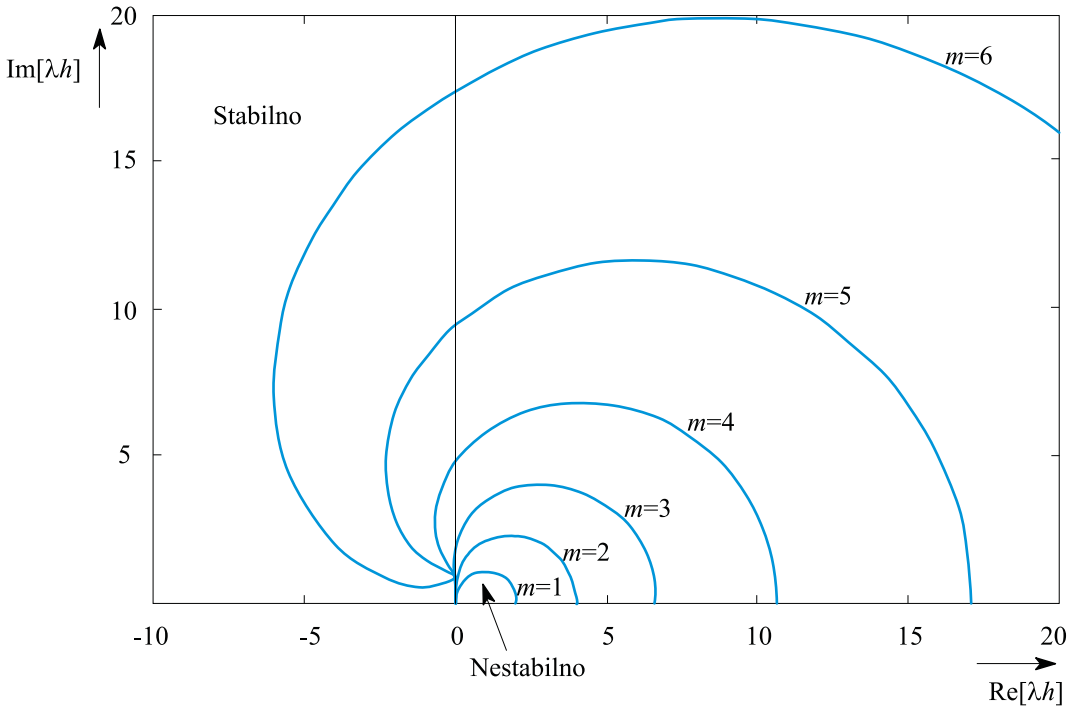
Tako je razvil metode od prvega do šestega reda z naslednjimi značilnostmi:

- Implicitne enačbe se rešujejo z Newton-Raphson-ovim algoritmom.

- Jacobijeva matrika pa se ponovno izračuna le, če poseben test pokaže, da sedanja vrednost ni več ustrezna.

- Lokalna napaka se nadzira med simulacijo z metodami spreminjanja računskega koraka in reda metode.

Slika 6.15 prikazuje stabilnostna področja Gear-ovih metod prvega do šestega reda. Metode so stabilne zunaj zaključenih krivulj.



Slika 6.15: Stabilnostna področja Gear-ovih metod

Poznamo še druge metode, ki so primerne za simulacijo togih sistemov. Ena od njih je polimplicitna Rosenbrock-Wanner-jeva metoda. Veliko pa na tem področju obetajo ekstrapolacijske metode z implicitnim sredinskim pravilom.

6.1.7 Izbira računskega koraka in postopki za njegovo avtomatsko nastavljanje

Izbira računskega koraka

V predhodni obravnavi smo ugotovili, da ima zmanjšanje računskega koraka naslednje posledice:

- zmanjša se lokalna (in s tem tudi globalna) napaka numerične metode,
- izboljšajo se lastnosti v zvezi z numerično stabilnostjo,
- izboljša se konvergenca iterativnih postopkov pri implicitnih metodah,
- poveča se napaka zaradi končne dolžine besede in
- poveča se porabljeni računalniški čas.

Na srečo imajo danes vsa sodobna simulacijska orodja vgrajene algoritme za avtomatsko nastavljanje velikosti računskega koraka. Tako je lahko le-ta relativno velik, če se odvodi spreminjajo počasi, če pa le-ti postanejo živahnejši, se temu ustrezno zmanjša tudi računski korak. Postopek zagotavlja, da je med celotno simulacijo napaka znotraj dopustnih meja. Tak način je posebej učinkovit v primerih, če se med simulacijo spreminja dinamika sistema ali dinamika vhodnih signalov (nelinearni sistemi, časovno spremenljive lastne vrednosti, nezveznosti vhodnih spremenljivk, odvodov,...).

Tako se za uporabnika problem določitve primernega računskega koraka spremeni v problem izbire smiselne tolerance za napako. Toda tudi to ni povsem enostavna naloga. Za smiselno izbiro mora imeti uporabnik nekaj znanja o modelu, ki ga simulira oz. predvsem o tem, kako natančen je model. To pa je v veliki meri povezano z natančnostjo podatkov, ki so bili uporabljeni pri razvoju modela.

Razen tolerance lahko uporabnik poda tudi začetno vrednost računskega koraka. Le-tega izbere tako, da je enak intervalu opazovanja (komunikacijski interval) rezultatov simulacije. S smiselo izbranim začetnim računskim korakom je včasih tudi možno prihraniti nekaj računalniškega časa. Zato pa mora imeti uporabnik nekaj znanja (informacij)

- o dinamiki modela (lastne vrednosti, časovne konstante,...) in

- o dinamiki vhodnih signalov (npr. frekvenčni spekter).

Priporočljivo je izbrati vsaj tako majhno začetno vrednost računskega koraka,

- da je izpolnjen pogoj za numerično stabilnost (npr. $\mid \lambda_{max} \mid h < 3$ za metode Runge-Kutta).
- Razen tega moramo zagotoviti, da na eno časovno konstanto pride vsaj nekaj računskih korakov (velja najmanjša časovna konstanta za metode, ki niso predvidene za toge sisteme in največja pri metodah za toge sisteme).
- Glede na dinamiko vhodnih signalov pa moramo upoštevati Shannon-ov teorem (teoretično naj bosta vsaj dva računska koraka na periodo maksimalne frekvence, praktično pa pet do deset).

Metode za avtomatsko nastavljanje računskega koraka

Kot smo že omenili, imajo sodobna simulacijska orodja algoritme za avtomatsko nastavljanje računskega koraka med simulacijo. Le-ta se običajno lahko giblje med maksimalno vrednostjo, ki je podana s komunikacijskim intervalom in med minimalno vrednostjo, ki jo lahko poda uporabnik. Vse metode avtomatskega nastavljanja temeljijo na sprotnem ocenjevanju lokalne napake numerične metode, ki je za metodo m-tega reda

$$\boldsymbol{e} = \boldsymbol{\Phi} \cdot h^{m+1} \tag{6.42}$$

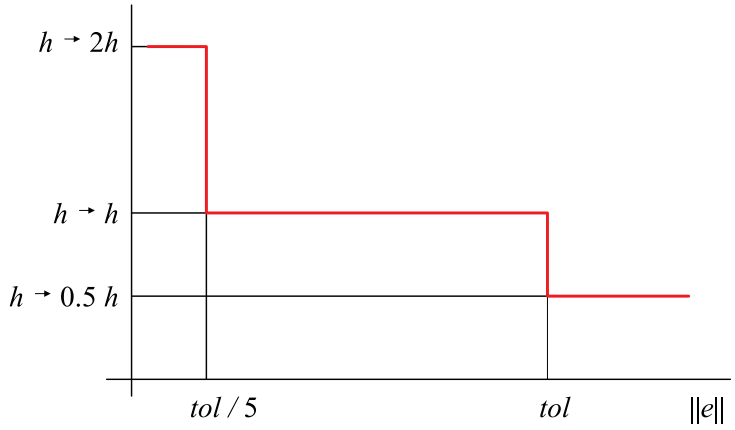
pri čemer je $\boldsymbol{\Phi}$ vektor, ki zavisi od rešitve diferencialne enačbe. Vektor ocene $\boldsymbol{e}$ se izračuna v vsakem računskem koraku na način, ki je odvisen od vrste integracijske metode. Iz vektorja ocene je potrebno izračunati skalarno veličino (normo napake), saj jo moramo v algoritmu za avtomatsko nastavljanje računskega koraka primerjati s toleranco. Uporabljajo se različne norme, ki navadno kombinirajo tudi relativno in absolutno napako. Tipično normo podaja enačba

$$\|\boldsymbol{e}\| = \max_i \left| \frac{e_i}{|x_i| + \eta_i} \right| \tag{6.43}$$

kjer je e_i i-ta komponenta $\boldsymbol{e}$ in η_i skalirni faktor za i-to komponento (običajno je ena). Za $|x_i| \gg \eta_i$ predstavlja norma relativno napako, za $|x_i| \ll \eta_i$ pa absolutno napako. Ob uporabi take norme ima potem tudi toleranca, ki jo podaja uporabnik, absolutno - relativni značaj.

Klasična metoda za avtomatsko nastavljanje računskega koraka temelji na razpolavljanju in podvajanju računskega koraka. Postopek, ki ga ilustrira tudi slika 6.16, je naslednji:

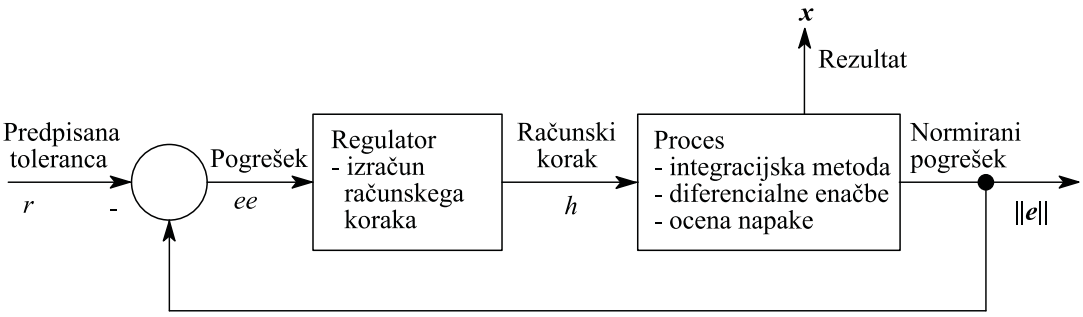
- Če norma napake v nekem trenutku presega podano toleranco, se računski korak razpolovi in integracija se ponovi.
- Če pa norma pomnožena s celoštevilčno konstanto (npr. 5) postane manjša od tolerance, se računski korak podvoji.
- Konstanta večja od ena vnese v algoritem mrtvo cono, kar prepreči morebitna nihanja računskega koraka h. Napake v področju mrtve cone namreč ne spremenijo računskega koraka.



Slika 6.16: Tropoložajni regulator za nastavitev računskega koraka

Opisan postopek je v teoriji avtomatskega vodenja poznan kot tropoložajna nelinearna regulacija. Iz te teorije pa tudi vemo, da je primernejše vedenje regulacijskega sistema možno dobiti z zveznim regulatorjem. Postopek prikazuje slika 6.17.

Norma lokalne napake $\|\boldsymbol{e}\|$, ki je izhod "procesa", se primerja s predpisano toleranco r. Razliko vodimo v regulator, ki izračuna novo vrednost računskega koraka



Slika 6.17: Regulacijski sistem za avtomatsko nastavljanje računskega koraka

h. Regulator mora zagotoviti, da je $\|\boldsymbol{e}\|$ približno enak r. Dober regulator mora zagotoviti ne preveč nihajoči potek računskega koraka.

Na žalost izsledki moderne regulacijske teorije v preteklosti niso bili uporabljeni za obravnavani problem. Mnoge sodobne integracijske metode (npr. Gear-ova metode za toge sisteme (Gear, 1971) uporabljajo za avtomatsko nastavljanje računskega koraka integrirni regulator. Vemo pa, da tak regulator predvsem v smislu stabilnosti slabo vpliva na delovanje regulacijskega sistema, zato gotovo ne predstavlja optimalne rešitve. Nihajoči potek računskega koraka je zlasti očiten, če uporabljamo metode za netoge sisteme (npr. Runge-Kutta) za simulacijo togih sistemov.

Gustaffson (Gustaffson, 1988, Gustaffson, 1990) je med prvimi poskušal uporabiti sodobne regulacijske algoritme za avtomatsko nastavljanje računskega koraka. V eksplicitni metodi Runge-Kutta je uporabil diskretni proporcionalno-integrirni (PI) regulator. Razvil je tudi diskretni model “procesa", ki ga je uporabil pri nastavitvi parametrov regulatorja. PI regulator je omogočil boljše delovanje pri le majhnem povečanju potrebnega računskega časa.

Veljajo naslednje ugotovitve:

- Avtomatsko nastavljanje računskega koraka je relativno enostavno vgraditi v enokoračne integracijske metode.
- Pri večkoračnih metodah pa je problem v tem, da po spremembi računskega koraka pretekle vrednosti, ki se uporabljajo v algoritmu, niso več direktno uporabne, ampak jih je potrebno z interpolacijskimi postopki preračunati na novo vrednost računskega koraka, kar seveda zahteva dodatni računski čas.

- Metode z avtomatskim nastavljanjem računskega koraka se ne uporabljajo za simulacijo v realnem času zaradi računalniške potratnosti in zaradi preveč spremenljive porabe računalniškega časa na posameznih intervalih.

6.1.8 Izbira integracijske metode

Primerno izbrana integracijska metoda mora

- zagotoviti rezultate v okviru predpisanih toleranc
- pri čim manjši porabi računalniškega časa.

Zavedati se moramo, da ni metode, ki bi bila enako primerna za vse probleme. Enostavni problem je običajno sicer možno simulirati s katerokoli metodo, toda brž ko imamo opraviti z realnimi in kompleksnimi problemi, lahko prihranimo veliko računalniškega časa z izbiro primerne metode. V tem podpoglavju bomo podali nekaj nasvetov v tem smislu (Cellier, 1979). Iz problema, ki ga simuliramo, moramo izluščiti nekaj informacij, ki so odločilne pri izbiri integracijske metode. To so

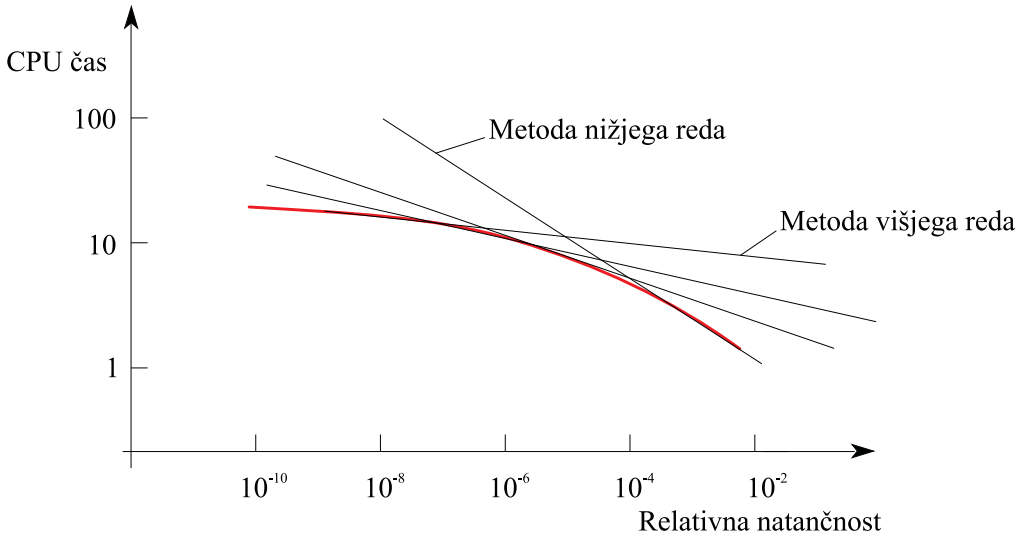
- zahteve po natančnosti,
- pogostost nezveznosti,
- togost in
- oscilatornost

Zahteve po natančnosti

Odvisnost med porabo računalniškega časa in relativno natančnostjo pri različnih redih metod prikazuje slika 6.18.

Velja:

- Uporabimo metodo nizkega reda, če ne potrebujemo velike natančnosti (npr. pri večjih zanemaritvah pri modeliranju, pri nenatančnih meritvah,...)



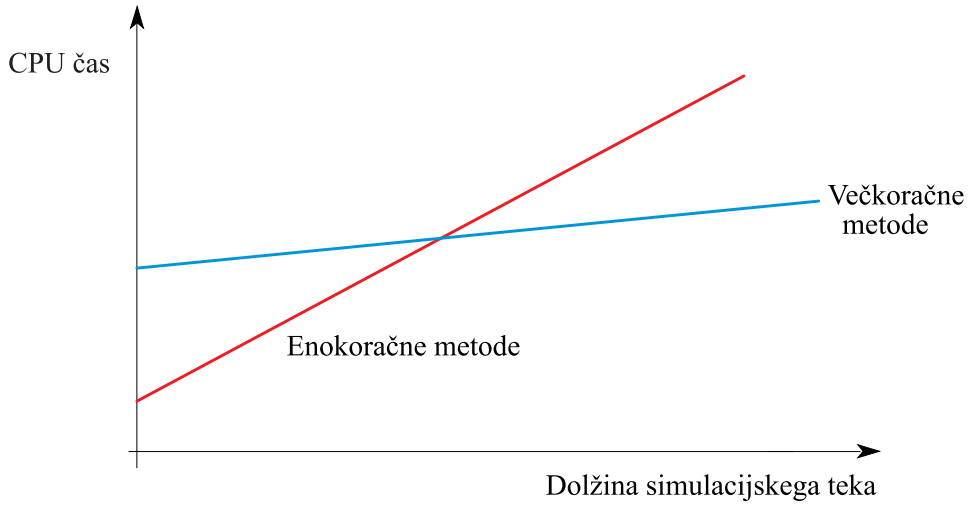
Slika 6.18: Odvisnost potrebnega računalniškega časa od zahtevane relativne natančnosti

- in metodo višjega reda pri večji zahtevani natančnosti, saj le-te zagotavljajo ob enakem računalniškem času bolj natančne rezultate.

- Poenostavljeno pravilo pravi, da je za relativno natančnost 10^{-m} potrebno uporabiti metodo m-tega reda.

Pogostost nezveznosti

Slika 6.19 prikazuje odvisnost porabe računalniškega časa od dolžine simulacijskega teka pri enokoračnih in pri večkoračnih metodah. Večkoračne metode so v primerjavi z enokoračnimi metodami bolj potratne v začetnem delu simulacije, zato pa so bolj ekonomične pri daljših simulacijah. To razložimo s tem, da enokoračne metode (npr. Runge-Kutta) ne potrebujejo nikakršne inicializacije, ker ne potrebujejo predhodnih vrednosti, večkoračne metode (npr. Adams-Moulton) pa morajo na začetku izračunati manjkajoče predhodne vrednosti. To dejstvo je zlasti pomembno, če imajo spremenljivke v modelu pogoste nezvezne prehode. Druga potrebna informacija, ki jo je treba izluščiti je torej, ali imamo opravka s *pogostimi nezveznostmi*. Za integracijo preko nezveznosti kvalitetni algoritmi detektirajo trenutek njegovega nastopa. Pri enokoračnih metodah je potrebno le spremeniti zadnjo vrednost računskega koraka, tako da se integracija zaključi natančno v trenutku nastopa nezveznosti. Pri večkoračnih metodah pa



Slika 6.19: Odvisnost potrebnega računalniškega časa od dolžine simulacijskega teka

je treba v vsaki točki nezveznosti ponoviti postopek inicializacije. Zato te metode niso primerne, če imamo med simulacijo pogosto opraviti z nezveznostmi (npr. dvopoložajna regulacija, funkcijski generatorji,...).

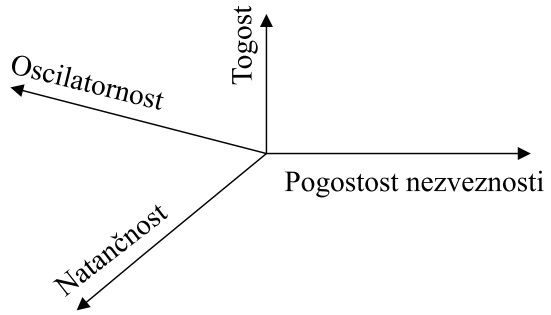
Togost

Tretja informacija, ki vpliva na izbiro integracijske metode, je *togost* modela. Le-to določimo z izračunom lastnih vrednosti Jacobijeve matrike (npr. z orodjem MATLAB, če simulacijsko orodje tega ne omogoča). Če je razmerje med maksimalno in minimalno vrednostjo realnih delov lastnih vrednosti večje kot 100, je priporočljivo izbrati integracijsko metodo za toge sisteme (npr. Gear-ovo metodo).

Oscilatornost

Končno predstavljajo poseben problem tudi *oscilatorni sistemi*, to so sistemi, ki imajo dominantne pole blizu imaginarne osi v ravnini λh. V tem primeru moramo nujno uporabiti metodo, ki ima stabilno področje vsaj v celotni levi polovici ravnine λh (implicitne ali polimplicitne metode nizkega reda).

Podatki o omenjenih štirih lastnostih (natančnost, pogostost nezveznosti, togost, oscilatornost, glej sliko 6.20) običajno zadostujejo za ustrezno izbiro integracijske metode. Zato zaključimo z zbranimi nasveti, pri tem pa poudarimo, da je uporabnik seveda omejen z zmožnostmi, ki mu jih daje uporabljeno simulacijsko orodje.



Slika 6.20: Značilnosti modela, ki vplivajo na izbiro integracijske metode

Končna priporočila

- Uporabimo metodo višjega reda za večjo natančnost in metodo nižjega reda, če se zahteva manjša natančnost.

- Uporabimo metodo Runge-Kutta-Fehlberg 4,5, če ne vemo ničesar o numerični problematiki v simulaciji, če je simulirani sistem netog ali če nastopajo pogoste nezveznosti. V okolju Matlab-Simulink sta uporabni metodi ode45 Dormand Prince in ode23 Bogacki-Shampine.

- Uporabimo Gear-ovo metodo za toge sisteme, ekstrapolacijsko metodo za toge sisteme ali kakšno drugo implicitno ali polimplicitno metodo (trapezoidno pravilo, implicitna Eulerjeva metoda,...) pri togih sistemih (v okolju Matlab-Simulink uporabljamo ode15s večkoračno metodo in ode23s enokoračno metodo).

- Uporabimo Adamsovo večkoračno metodo, če ne nastopajo nezveznosti in če sistem ni tog (v okolju Matlab-Simulink je to metoda ode113-prediktor korektor metoda iz Adams-Bashforth in Adams-Moulton).

- Uporabimo ekstrapolacijsko metodo, če se zahtevajo izredno natančni rezultati.

- Uporabimo implicitno metodo nizkega reda pri simulaciji oscilatornih sistemov.

Metode, ki morajo med simulacijo izračunavati Jacobijevo matriko, postanejo z naraščajočim redom diferencialnih enačb izredno računalniško potratne. V tem primeru postanejo nekateri od zgoraj navedenih nasvetov vprašljivi.

6.2 Numerična problematika pri simulaciji sistemov z nezveznostmi

Realni fizikalni sistemi so po naravi strogo gledano zvezni. Zaradi zanemaritev in poenostavljanj med modeliranjem pa njihovi modeli pogosto vsebujejo nezveznosti. Standard CSSL'67 vključuje uporabo gradnikov, ki vsebujejo nezveznosti. Tako imajo orodja v svojih knjižnicah številne nezvezne nelinearne funkcije (npr. mrtva cona, nasičenje, mrtvi hod, histereza, ...). Toda le najsodobnejša orodja te nezvezne funkcije tudi numerično pravilno obdelajo.

Nezveznosti delimo v dve skupini:

1. Nezveznosti, ki nastopajo v vnaprej znanih časovnih trenutkih.
2. Nezveznosti, ki jih proži stanje sistema oz. spremenljivke sistema; trenutek nastopa ni znan vnaprej.

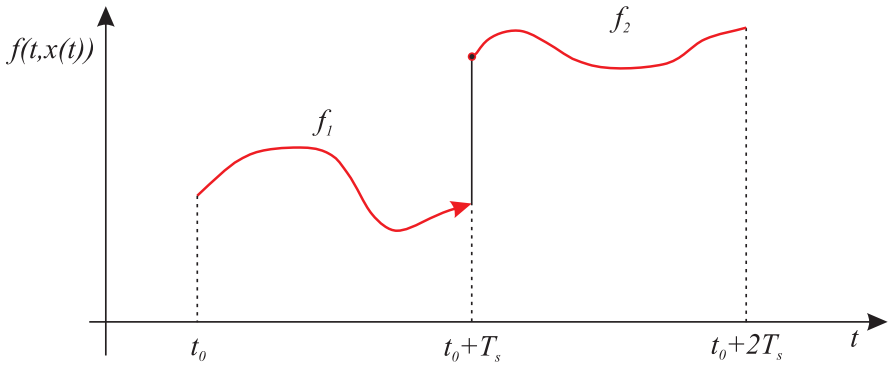
6.2.1 Nezveznosti, ki nastopajo v vnaprej znanih časovnih trenutkih

Ta problem je znan pri diskretnem (računalniškem) vodenju zveznih procesov, ko regulator enkrat na periodo vzorčenja preračuna novo vrednost regulirne veličine, vmes pa je vrednost konstantna. Numerično problematiko je v tem primeru relativno enostavno rešiti. Ker je trenutek nastopa nezveznosti znan vnaprej, mora postopek nastavljanja računskega koraka med simulacijo zadnji računski korak spremeniti tako, da bo zaključek časovno sovpadal s trenutkom vzorčenja. Pri večkoračnih metodah se mora ob vsakem vzorčnem trenutku metoda inicializirati.

Vendar je potrebno tudi v integracijski postopek uvesti nekatere spremembe. Osnovna enačba, ki opisuje enokoračne integracijske metode, je

$$\boldsymbol{x}(t_0 + T_s) = \boldsymbol{x}(t_0) + \int_{t_0}^{t_0+T_s} \boldsymbol{f}(t, \boldsymbol{x}(t))dt \tag{6.44}$$

Naj bo funkcija $\boldsymbol{f}(t, \boldsymbol{x}(t))$ funkcija z nezveznostjo v trenutku $t_0 + T_S$. Prikazuje jo slika 6.21.



Slika 6.21: Nezvezna funkcija $f(t, x(t))$

Funkcija $f(t, x(t))$ je torej

$$f(t, x(t)) = \left\{ \begin{array}{ll} f_1 & t_0 \leq \quad t < t_0 + T_s \\ f_2 & t_0 + T_s \leq \quad t < t_0 + 2T_s \end{array} \right\} \tag{6.45}$$

Modifikacijo, ki je potrebna v enačbi 6.44, opisuje naslednja enačba

$$\boldsymbol{x}(t_0 + T_s) = \boldsymbol{x}(t_0) + \int_{t_0}^{(t_0+T_s)-} \boldsymbol{f}(t, \boldsymbol{x}(t))dt \tag{6.46}$$

Iz enačb 6.44 in 6.45 ter iz slike 6.21 vidimo, da je potrebno integrirati na intervalu tako, da se zaključi integracija na tem intervalu prej, preden pride do spremembe odvodne funkcije v trenutku $t = t_0 + T_s$. Torej je pravilni vrstni red operacij

- integracija do $t = t_0 + T_s$,

- sprememba odvodne funkcije (izhodov diskretnih blokov v primeru diskretnih regulatorjev) v $t = t_0 + T_s$,
- prikaz rezultatov v $t = t_0 + T_s$.

Zato sta potrebna dva klica (izračuna) odvodne funkcije v trenutku $t = t_0 + T_s$.

6.2.2 Nezveznosti, ki jih proži stanje sistema oz. spremenljivke sistema; trenutek nastopa ni znan vnaprej

Ta vrsta nezveznosti povzroča resnejše probleme. Orodja, ki numerično pravilno obdelajo take vrste nezveznosti, morajo:

- odkriti (detektirati) nezveznost,
- ugotoviti točko nezveznosti (lokacijo), t.j. določiti vrednost neodvisne spremenljivke (običajno čas),
- numerično pravilno integrirati.

Odkrivanje nezveznosti

Gear in Osterby (Gear, Osterby, 1984) predlagata, da ima sistem nezveznost v primeru, če metoda prilagodljivega računskega koraka ugotovi, da je nova vrednost računskega koraka manjša od polovice prejšnjega računskega koraka. Tak način omogoča avtomatsko odkrivanje.

V večini simulacijski orodij pa mora uporabnik definirati t.i. preklopno funkcijo oz. funkcijo nezveznosti (switching function, discontinuity function). To je spremenljivka modela ali linearna funkcija več spremenljivk in ima eno od naslednjih oblik:

$$\begin{aligned} \phi(t) &= x(t) \\ \phi(t) &= x(t) - y(t) \\ \phi(t) &= \sum_{i=1}^{n} [k_i x_i(t) + n_i] \end{aligned} \tag{6.47}$$

Ko ta funkcija postane enaka vrednosti nekega praga (običajno nič), pomeni to indikacijo integracijskemu postopku, da je prišlo do nezveznosti. Nezveznost nastopi pri prehodu iz pozitivne v negativno vrednost ali obratno.

Do nezveznosti pride, ko je $x(t) = 0$ oz. $x(t) = y(t)$, t.j. $\phi(t) = 0$. Torej modeler pomaga simulacijskemu sistemu v postopku detekcije in lokacije nezveznosti. Zelo znan simulacijski problem je modeliranje skakajoče žoge. Sistem modeliramo tako, da v trenutku, ko višina $h(t)$ postane nič (žoga se dotakne tal)

$$h(t) = 0$$

spremenimo spremenljivko stanja modela- hitrost

$$v = kv \qquad k > -1$$

V tem primeru je torej preklopna funkcija

$$\phi(t) = h(t)$$

V vsakem računskem koraku se razen enačb modelnih spremenljivk ovrednoti tudi preklopna funkcija. Če med dvema zaporednima računskima trenutkoma (angl. mesh points) preklopna funkcija spremeni predznak, pomeni, da je nastopila nezveznost, nakar je treba natančno določiti trenutek nezveznosti.

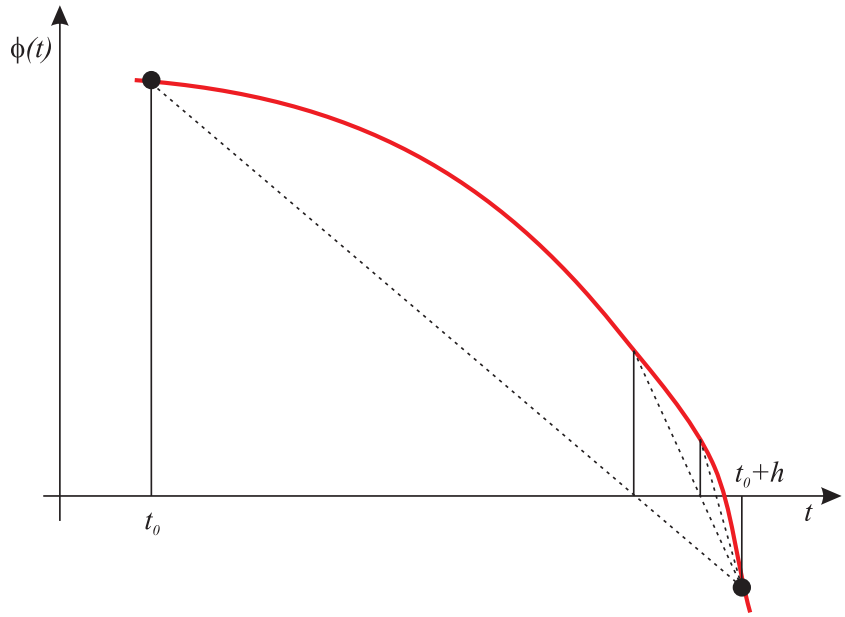
Problem pa je, če ima preklopna funkcija več prehodov skozi nič na enem računskem intervalu. Če je teh prehodov sodo število, seveda sploh ne odkrijemo nezveznosti, saj se predznak ne spremeni. Do danes še ni povsem zanesljive metode oz. dokaza za detekcijo nezveznosti. V praksi pa opisani postopek deluje zadovoljivo. Če je $\phi(t)$ linearna funkcija spremenljivk stanj, se redko dogodi, da bi imela več prehodov skozi nič.

Problem je tudi takrat, če nastopa več preklopnih funkcij, ki imajo ničlo na istem računskem koraku.

Funkcija $\phi(t)$ sicer v ničemer ne vpliva na enačbe, ki opisujejo matematični model.

Ugotavljanje trenutka nezveznosti

Po detektiranju nezveznosti je potrebno določiti trenutek nastopa nezveznosti. Uporabljamo iterativne postopke s hitro konvergenco in širokim konvergenčnim področjem. Itertivni postopek prikazuje slika 6.22.



Slika 6.22: Iterativni postopek za lokacijo nezveznosti

Pravilno numerično integriranje

Ko je algoritem odkril in lociral nezveznost, je nadaljnji postopek enak kot pri nezveznostih, pri katerih vnaprej poznamo trenutek nastopa.

Izvedba preklopne funkcije v okolju Matlab-Simulink

Omenjena preklopna funkcija ima v okolju Matlab-Simulink ime Explicit zero crossing function in daje modelerju možnost, da 'pomaga' integracijskemu algoritmu pri ugotavljanju trenutka nezveznosti. Izvedena je z blokom `Hit crossing` v menuju `Sinks`. Na vhod v ta blok pripeljemo spremenljivko-preklopno funkcijo, ki se primerja z pragom, ki je parameter bloka. Pri prehodu razlike med preklopno funkcijo in pragom skozi nič (v pozitivni, negativni ali v obeh smereh) se sproži postopek za določitev trenutka nezveznosti.

Nekateri bloki, ki sicer povzročajo nezveznosti, imajo opisane mehanizme vgrajene v sam blok, saj je znotraj bloka jasno, kaj povzroča nezveznost. Pravimo,

Tabela 6.3: Bloki v Simulinku, ki imajo vgrajen mehanizem detekcije nezveznosti

`Abs`	`Backlash`
`Compare To Constant`	`Compare To Zero`
`Dead Zone`	`Enable`
`From Workspace`	`If`
`Integrator`	`MinMax`
`Relational Operator`	`Relay`
`Saturation`	`Sign`
`Signal Builder`	`Step`
`Switch`	`Switch Case`
`Trigger`	`Enabled and Triggered Subsystem`

da imajo bloki vgrajeno preklopno funkcijo (Intrinsic zero crossing function). Tabela 6.3 vsebuje bloke, ki imajo vgrajeno preklopno funkcijo.

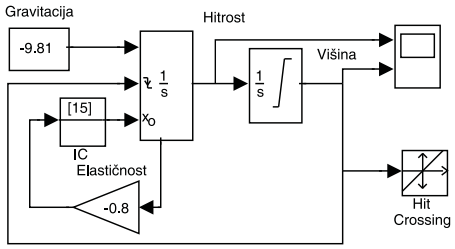
Primer 6.1 Skakajoča žoga

Primer najdemo v številnih priročnikih simulacijskih orodij. Žogo spustimo z določene začetne višine in z začetno hitrostjo. Ko doseže tla, se seveda odbije. Odboj modeliramo tako, da v trenutku spremenimo hitrost. Hitrost po odboju ima obratno in zaradi izgube energije nekoliko nižjo vrednost. Za modeliranje uporabimo Newtonove zakone

$$\begin{aligned} \sum F_i &= ma \\ -mg &= ma = mh'' \\ h'' &= -g = -9.81\ [m/s^2] \\ h(0) &= 10\ [m] \\ v(0) &= 15\ [m/s] \\ h(t) = 0 \longrightarrow v &= kv \quad k = -0.8 \end{aligned} \tag{6.48}$$

Simulacijsko shemo prikazuje slika 6.23.

Za realizacijo nezveznosti uporabljamo proženi integrator. Ko gre hitrost iz pozitivne v negativno področje, spremenimo vrednost hitrosti v skladu z enačbo



Slika 6.23: Simulacijska shema v Simulinku za model skakajoče žoge

$v = -0.8v$. Uporabimo stanje integratorja, ki je po vrednosti enako izhodu integratorja, vendar omogoča, da signal povežemo na vhod za proženje. Uporabimo tudi blok za določitev začetne vrednosti signala - hitrosti (`IC`).

Preklopna funkcija je v tem primeru kar signal $h(t)$, zato na ta signal priključimo blok `Hit Crossing` in na ta način omogočimo določitev trenutka odboja žoge. V tem primeru uporaba tega bloka niti ni potrebna, saj za ustrezni numerični postopek poskrbi tudi proženi integrator, ki spada med bloke z vgrajenim mehanizmom za določitev nezveznosti.

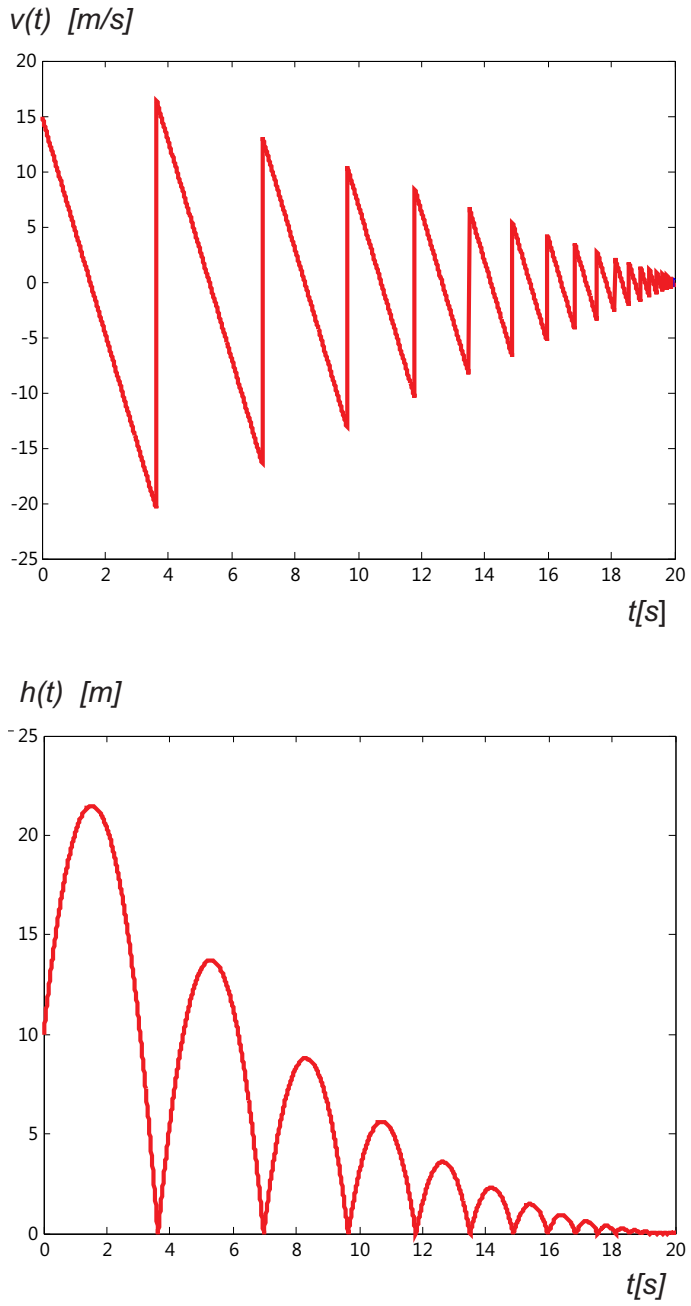
Slika 6.24 prikazuje potek višine $h(t)$ in hitrosti $v(t)$.

□

6.3 Algebrajske zanke

Pred izvajanjem simulacije je potrebno v fazi procesiranja modela razvrstiti stavke oz. bloke, ki opisujejo model. Ko v fazi izračunavanja odvodov nek stavek (blok) pride na vrsto, morajo biti vse spremenljivke desno od enačaja predhodno ovrednotene.

- Pri dobro modeliranem realnem problemu je možno vse spremenljivke modela izračunati iz spremenljivk stanja (iz izhodov integratorjev) in iz vhodnih signalov. V takem primeru vrstni algoritem lahko razvrsti stavke (bloke) iz simulacijskega programa v pravilen proceduralni vrstni red za izračun odvodov (npr. podprogram DERIV).



Slika 6.24: Potek višine $h(t)$ in hitrosti $v(t)$

- Vsak problem, v katerem je možno stavke razvrstiti na opisani način, ima lastnost, da je v vsaki zanki simulacijske sheme vsaj en blok z zakasnitvenim atributom (integrator, diskretna zakasnitev, zakasnilni člen,...).

- Če so modeli v t.i. kanoničnih oblikah, potem ni problemov v razvrščanju.
- Zaradi
  - napak v programiranju (nedefinirane konstante, vhodni signali, tipkarske napake v imenih spremenljivk,...),
  - zaradi slabega modeliranja,
  - zaradi kakšnih drugih razlogov, ki zahtevajo bolj eksotično modeliranje ali
  - pa zaradi komponent, ki ne vsebujejo zakasnitev (npr. regulator, električno vezje),

  vrstni algoritem ne uspe razvrstiti blokov. Eden od razlogov je lahko, da je vhod nekega bloka algebraično odvisen od lastnih izhodov. Taki zanki pravimo *algebrajska zanka*. Pomeni, da ima model vsaj eno zanko, v kateri nastopajo le bloki brez zakasnitvenega atributa (običajno so to statični bloki kot npr. sumatorji, množilniki,...). Take zanke lahko povzročijo zaradi neidealnih komponent probleme celo pri simulaciji na analognem računalniku, pri digitalni simulaciji pa je taka zanka še posebej neprijetna.

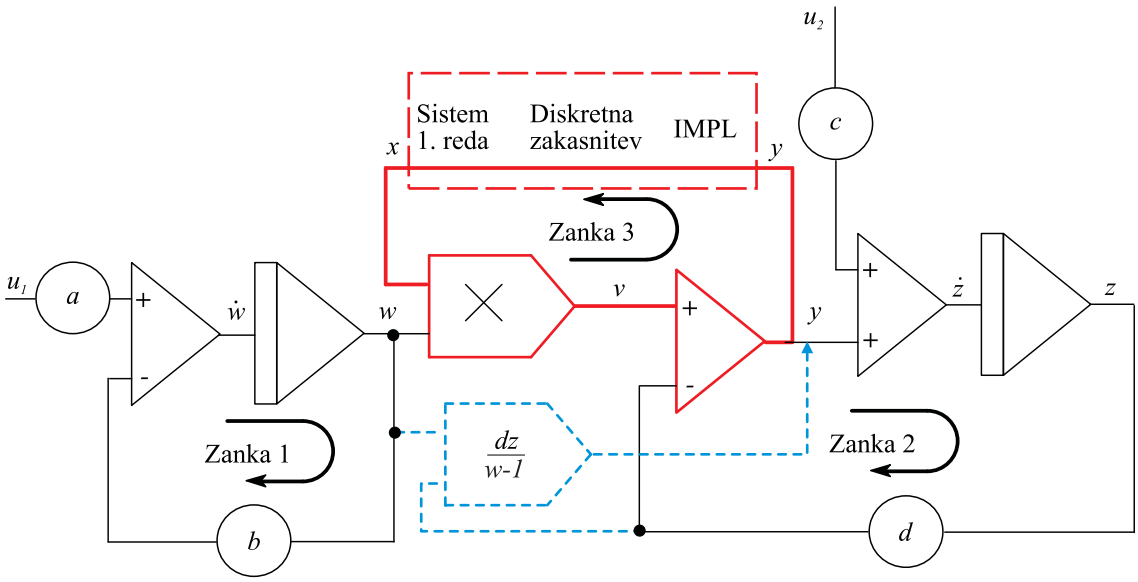
V številnih primerih lahko algebrajske zanke odpravimo z algebraično preureditvijo enačb, ki opisujejo model oz. njegove podmodele. Taka rešitev omogoča znaten prihranek potrebnega računalniškega časa, precej pa se poveča tudi natančnost rezultatov. Zato je preureditev enačb (algebraična eliminacija zanke) vedno prvo, kar je vredno poizkusiti. Nekatera orodja, ki ne rešujejo le problema simulacije, ampak nudijo tudi močno podporo pri modeliranju, izvršijo ta postopek avtomatsko (npr. DYMOLA, Dymola, 2011).

Slika 6.25 prikazuje simulacijsko shemo nekega sistema drugega reda. V tej shemi se nahajajo tri zanke. Prva in tretja zanka vsebujeta integratorja, druga zanka, ki je poudarjeno označena pa je algebrajska zanka, saj velja

$$y = f(y) \tag{6.49}$$

V tem primeru je algebrajsko zanko možno izločiti z algebraično preureditvijo. Ker je

$$y = r - dz = wy - dz \tag{6.50}$$



Slika 6.25: Simulacijska shema nekega sistema drugega reda

lahko y izračunamo iz enačbe

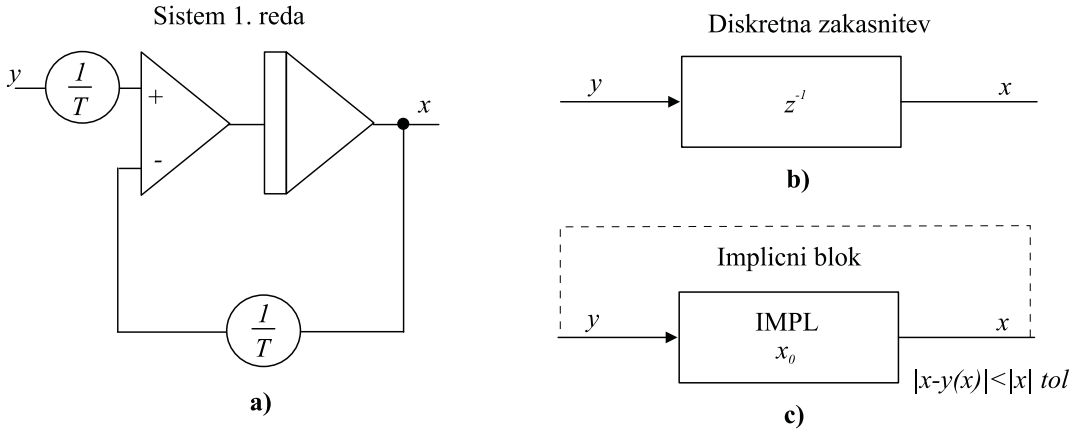
$$y = \frac{dz}{w-1} \tag{6.51}$$

Na sliki 6.25 je postopek označen s črtkano črto.

Včasih pa algebrajske zanke ni možno odpraviti na tako enostaven način. Če simulacijsko shemo na sliki 6.25 pretvorimo v simulacijski program, bo simulacijski prevajalnik verjetno sporočil, da ne more razvrstiti blokov. In če ne vsebuje algoritma za t.i. implicitno reševanje algebrajske zanke, ima uporabnik le to možnost, da zanko umetno prekine, tako da doda zakasnilni člen (sistem prvega reda s časovno konstanto T in ojačenjem ena), ali pa diskretno zakasnitev. Možnosti so prikazane na sliki 6.26.

Če uporabimo zvezni sistem, mora biti časovna konstanta T precej manjša od ostalih časovnih konstant modela. Diskretna zakasnitev pa je običajno enaka komunikacijskemu intervalu. Teoretično bi seveda morali izbrati čim manjšo časovno konstanto in čim krajšo zakasnitev, praktično pa nas to pripelje do numeričnih problemov v zvezi s togimi sistemi.

Mnogi simulacijski jeziki pa omogočajo reševanje algebrajske zanke na implicitni način. V tem primeru se zanka prekine s t.i. implicitnim blokom (operator



Slika 6.26: Možnosti za prekinitev algebrajske zanke

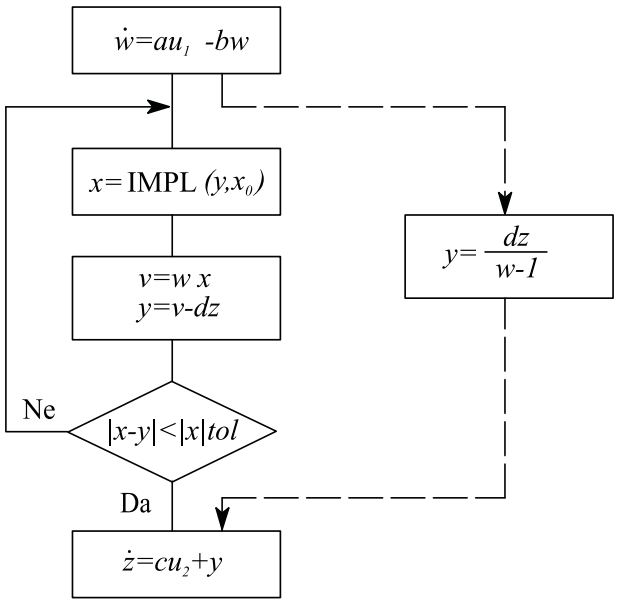
IMPL, ki ga vstavi uporabnik), kakor prikazujeta sliki 6.25 in 6.26. Ta blok jemlje prevajalnik kot blok z zakasnitvenim atributom. Zato ga vrstni algoritem postavi pred vse druge bloke v zanki. Med računanjem odvodov postavi IMPL blok na svoj izhod začetno vrednost spremenljivke x. S pomočjo te vrednosti se izračuna vrednost $y(x)$, kakor zahteva zanka. S to vrednostjo se običajno s pomočjo Newton-Raphsonove iterativne metode znotraj IMPL bloka izračuna nova ocena spremenljivke x. Postopek se nato ponavlja, dokler razlika $\mid x - y(x) \mid$ ne postane znotraj tolerance $\mid x \mid tol$ (tol je relativna toleranca).

Za učinkovito izračunavanje je potrebno v iterativnem računanju izvajati le stavke (bloke), ki se nahajajo znotraj algebrajske zanke. Vrstni red izračunavanja prikazuje slika 6.27. Na isti sliki je s črtkano povezavo označen tudi postopek algebraične preureditve enačb.

Stavki, ki jih je možno razvrstiti tudi brez IMPL operatorja, se izvršijo na začetku. Nato sledijo stavki, ki tvorijo algebrajsko zanko in se iterativno izračunavajo. Na koncu pa se izvajajo stavki, ki za izračunavanje potrebujejo rezultate implicitnega postopka, vendar se ne nahajajo v algebrajski zanki.

Kot smo že omenili, se pri implicitnem računanju običajno uporablja Newton-Raphson-ova iterativna metoda. Postopek se začne izvajati z začetnimi vrednostmi

$$x_0 \ldots \text{definira uporabnik}$$
$$x_1 = x_0 + 0.0001 x_0$$



Slika 6.27: Implicitno računanje algebrajske zanke

opisujeta pa ga enačbi

$$c_n = \frac{y(x_n) - y(x_{n-1})}{x_n - x_{n-1}} \tag{6.52}$$

$$x_{n+1} = \begin{cases} \frac{y(x_n) - c_n x_n}{1 - c_n} & c_n \neq 1 \\ y(x_n) & c_n = 1 \end{cases} \tag{6.53}$$

za $n = 1, 2, 3, \ldots$. Začetno izbrana vrednost x_0 se uporablja le na začetku simulacijskega teka. Na naslednjih računskih korakih se izbere začetna vrednost, ki je enaka rešitvi v prejšnjem računskem koraku ($x_0(t_k) = x(t_{k-1})$). Razen začetne vrednosti x_0 mora uporabnik običajno podati tudi relativno toleranco (tol) kot pogoj za končanje iterativnega postopka, maksimalno število iteracij (to je pomembno, če iterativni algoritem ne konvergira) in vrednost, ki se uporabi pri inkrementiranju spremenljivke x (0.0001 v našem primeru).

Pomembno je, da ima uporabljeni iterativni algoritem čim boljše konvergenčne lastnosti. Nobeden pa ne zagotavlja absolutne konvergentnosti.

Implicitni operator pri reševanju algebrajske zanke močno poveča porabo računalniškega časa. Tudi če algoritem naredi le nekaj iteracij, se morajo le-te izvršiti nekajkrat v vsakem računskem koraku (odvisno od uporabljene integracijske

metode). Zato je vredno že med modeliranjem vložiti več truda v to, da se po možnosti izognemo algebrajskim zankam.

7.

Inženirski pristop v eksperimentalnem modeliranju

Eksperimentalno modeliranje je pomembno področje matematičnega modeliranja, v katerem ne uporabljamo fizikalnih zakonov ampak s pomočjo merjenih vhodnih in izhodnih signalov realnega procesa ugotovimo matematične zakonitosti, ki vodijo do modela, ki pa podaja le vhodno-izhodno relacijo, nič pa ne pove o notranjih dogajanjih. Področje je zelo kompleksno in ga uvaja učbenik [Zupančič, 2012], podrobno pa obdeluje učbenik [Matko, 1998].

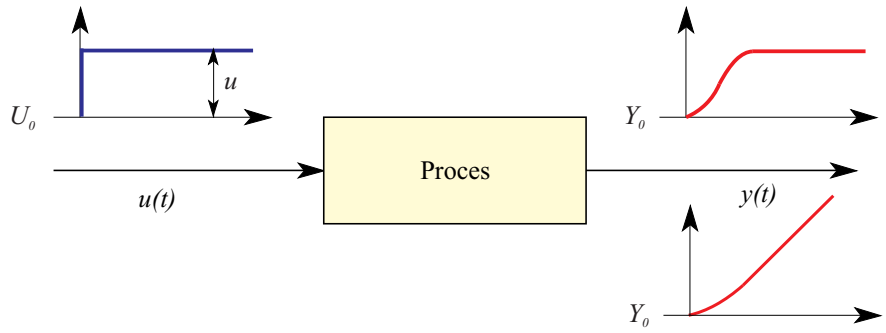
Na tem mestu bomo obdelali le nekaj preprostih, bolj inženirskih načinov za eksperimentalno modeliranje. Postopki temeljijo na izmerjenem odzivu realnega procesa na stopničasti vhodni signal. Primeren je za

- proporcionalne in integrirne procese,
- dušenje pa mora biti nadkritično (poli morajo biti realni).

Postopek ni primeren, če

- ima proces pole s pozitivnimi realnimi deli (nestabilni procesi) in
- če so merjeni signali izdatneje moteni.

Postopek ilustrira slika 7.1. Poudarimo, da so meritve vedno opravljene v neki delovni točki, ki je določena z delovno vrednostjo vhodnega (U_0) in izhodnega (Y_0) signala. Postopek identifikacije nas bo pripeljal do linearnih modelov (prenosnih funkcij), ki bodo pretežno veljavni le v okolici delovne točke.



Slika 7.1: Eksperiment za inženirsko modeliranje

Na sliki sta prikazana karakteristična odziva proporcionalnega in integrirnega procesa.

7.1 Ekperimentalno modeliranje proporcionalnih procesov

Odziv proporcionalnega sistema na konstantno (stopničasto) spremembo vhodnega signala je po prehodnem pojavu konstantna vrednost izhodnega signala (primer - vključimo grelnik konstantne moči in temperatura se začne dvigati in se čez nekaj časa ustali). Take sisteme skušamo modelirati z modeli P_0, P_1, P_2 ali P_n [Zupančič, 2012].

7.1.1 P_1 model

P_1 sistem opisuje prenosna funkcija z enim realnim polom oz. z ojačenjem K in časovno konstanto T

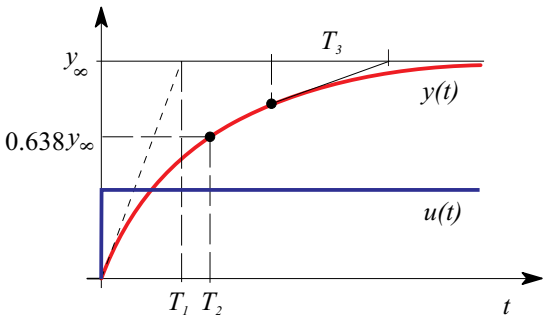
$$G(s) = \frac{K}{Ts + 1} \tag{7.1}$$

Model uporabimo, če velja

$$\dot{y}(0) > 0 \tag{7.2}$$

torej, če ima odziv v koordinatnem izhodišču pozitivni odvod.

Slika 7.2 prikazuje tipični odziv realnega procesa, ki ga lahko modeliramo s P_1 modelom.



Slika 7.2: Določanje parametrov P_1 modela

Časovno konstanto T določimo na več načinov: s pomočjo tangente iz koordinatnega izhodišča (T_1), s pomočjo časa, ko naraste odziv na 63.8 % končne vrednosti (T_2) in s pomočjo tangente v poljubni točki odziva (T_3). Če se realni proces vede kot idealni P_1 sistem, potem so vse tri vrednosti enake. Ker pa so v praksi procesi vedno tudi nelinearni, dobimo po vsej verjetnosti tri različne vrednosti, zato izračunamo srednjo vrednost

$$T = \frac{T_1 + T_2 + T_3}{3} \tag{7.3}$$

Ojačenje izračunamo kot kvocient konstantne, ustaljene vrednosti spremembe izhodnega in vhodnega signala

$$K = \frac{y_\infty}{u} \tag{7.4}$$

7.1.2 P_0 model

V kolikor je časovna konstanta T nekega sklopa zelo majhna v primerjavi z časovnimi konstantami ostalih sklopov (npr. zaporedna povezava tipala, merilnega pretvornika, filtra, ...), lahko časovno konstanto takega sklopa zanemarimo oz. izločimo ($T = 0$) in sistem (sklop) modeliramo s P_0 modelom

$$G(s) = K \tag{7.5}$$

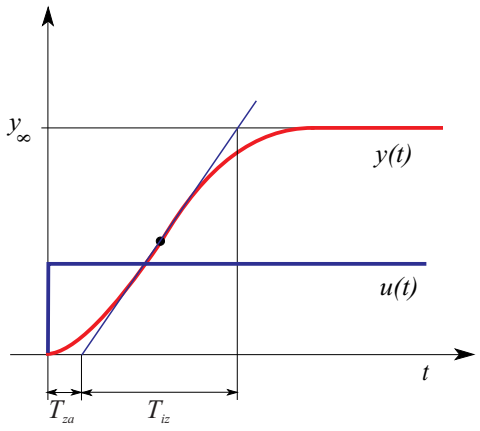
Veljati mora, da je tangenta odziva takega sklopa na konstantno vzbujanje v koordinatnem izhodišču zelo strma (teoretično npr. $\dot{y}(0) = \infty$).

7.1.3 P_2 in P_n model

Če pa je začetni odvod odziva

$$\dot{y}(0) = 0 \tag{7.6}$$

potem pri modeliranju uporabimo značilke, ki jih dobimo s pomočjo slike 7.3.



Slika 7.3: Značilke za določanje modelov P_2 in P_n

$K = \frac{y_\infty}{u}$	ojačenje
T_{za}	čas zaostajanja
T_{iz}	čas izravnave

Iz teh značilk določimo parametre modelov P_2 ali P_n v odvisnosti od razmerja $\frac{T_{za}}{T_{iz}}$.

P_2 model $\quad \frac{T_{za}}{T_{iz}} < 0.104$

Če je $\frac{T_{za}}{T_{iz}} < 0.104$, se priporoča model 2. reda P_2 z dvema različnima realnima poloma oz. z dvema različnima časovnima konstantama

$$G(s) = \frac{K}{(T_1 s + 1)(T_2 s + 1)} \tag{7.7}$$

S pomočjo diagramov na sliki 7.4 določimo iz razmerja $\frac{T_{za}}{T_{iz}}$ najprej razmerji $\frac{T_2}{T_1}$ in $\frac{T_{iz}}{T_1}$, iz tega pa lahko določimo obe časovni konstanti T_1 in T_2.

Ojačenje K določimo s pomočjo enačbe 7.4.

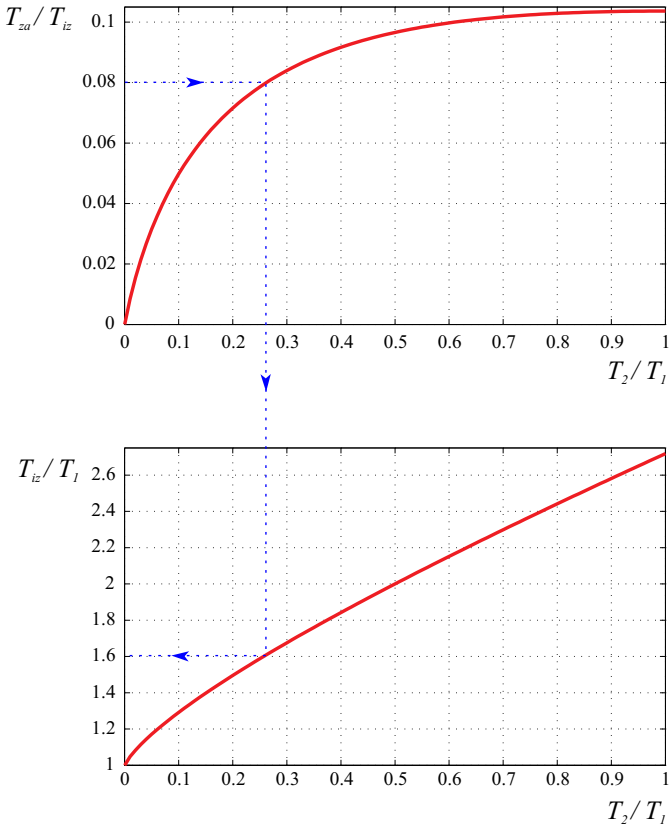
P_n model $\quad \frac{T_{za}}{T_{iz}} > 0.104$

Če je $\frac{T_{za}}{T_{iz}} > 0.104$, se priporoča P_n model z n enakimi realnimi poli oz. z n enakimi časovnimi konstantami

$$G(s) = \frac{K}{(Ts + 1)^n} \tag{7.8}$$

S pomočjo diagramov na sliki 7.5 določimo iz razmerja $\frac{T_{za}}{T_{iz}}$ najprej red n, nato razmerje $\frac{T_{za}}{T}$, iz tega pa časovno konstanto T.

Ojačenje K določimo s pomočjo enačbe 7.4.



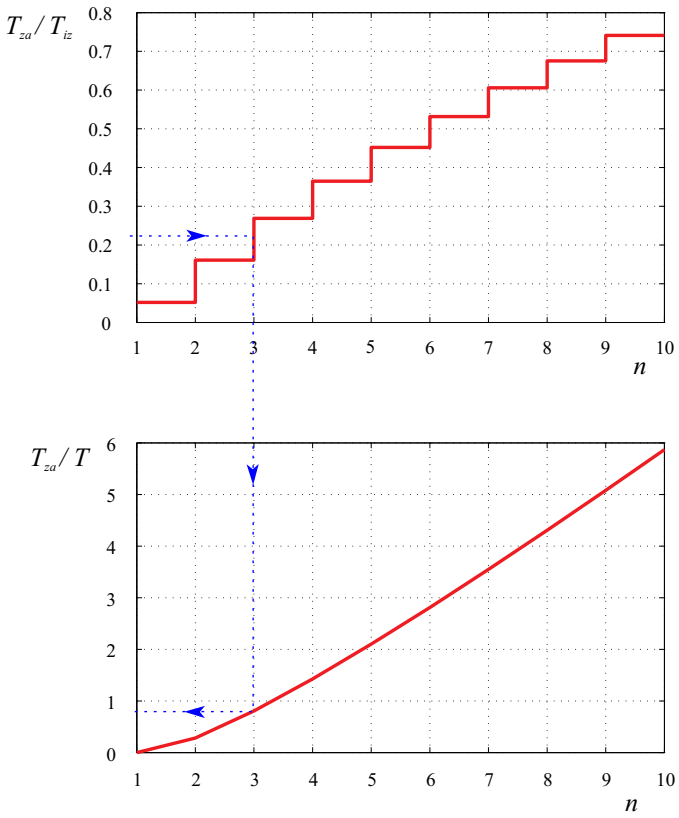
Slika 7.4: Diagrama za določitev T_1 in T_2 P_2 modela

7.2 Ekperimentalno modeliranje integrirnih procesov

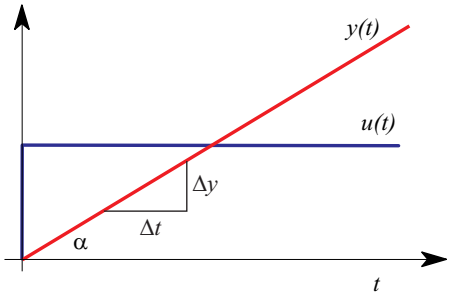
Odziv integrirnega sistema na konstantno (stopničasto) spremembo vhodnega signala se po prehodnem pojavu ne ustali na konstantno vrednost, ampak s časom narašča (primer-nivo tekočine v posodi narašča pri konstantnem dotoku). Take sisteme skušamo modelirati z modeli I_0 ali I_n.

7.2.1 I_0 model

Značilen odziv procesa, ki ga je smiselno modelirati z modelom I_0, prikazuje slika 7.6.



Slika 7.5: Diagrama za določitev n in T P_n modela



Slika 7.6: Odziv procesa, ki ga modeliramo z modelom I_0

Za take procese je značilno, da je hitrost izhodnega signala že na začetku enaka (maksimalni) hitrosti v kvazi ustaljenem stanju, ki ga označujemo z $y_{ss}(t)$ (ss označuje steady state - ustaljeno stanje)

$$\begin{aligned} \dot{y}(0) &= \dot{y}_{ss} \\ \dot{y}_{ss} &= \frac{\Delta y}{\Delta t} \end{aligned} \tag{7.9}$$

Model I_0 opisuje prenosna funkcija

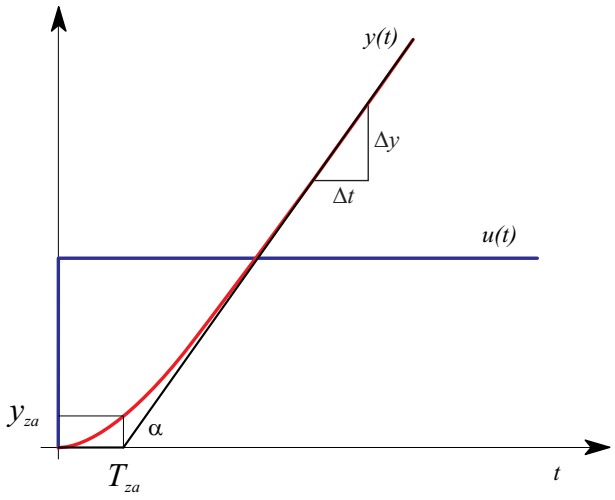
$$G(s) = \frac{K_I}{s} \tag{7.10}$$

Ojačenje K_I določimo s pomočjo enačbe

$$K_I = \frac{\dot{y}_{ss}}{u} \tag{7.11}$$

7.2.2 I_n model

Slika 7.7 pa prikazuje tipični odziv procesa, ki ga je smiselno modelirati z I_n modelom.



Slika 7.7: Odziv sistema I_n in ustrezne značilke

Začetna hitrost odziva je torej nič

$$\dot{y}(0) = 0 \tag{7.12}$$

Model opišemo s prenosno funkcijo z enim polom v koordinatnem izhodišču in še n istoležnimi realnimi poli

$$G(s) = \frac{K_I}{s(Ts+1)^n} \tag{7.13}$$

S pomočjo slike 7.7 določimo značilke

$\dot{y}_{ss} = \frac{\Delta y}{\Delta t}$	hitrost odziva v kvazi ustaljenem stanju, ko učinek časovnih konstant izzveni
T_{za}	čas zaostajanja
y_{za}	vrednost izhoda v času $t = T_{za}$

in iz teh značilk parametre modela I_n.

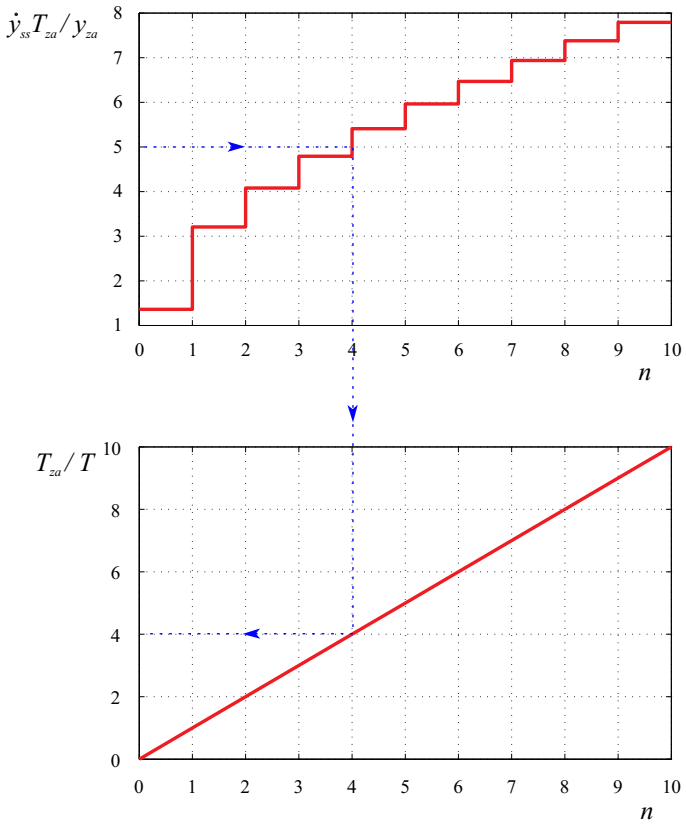
Ojačenje K_I podaja enačba

$$K_I = \frac{\dot{y}_{ss}}{u} \tag{7.14}$$

S pomočjo diagramov na sliki 7.8 določimo iz izračunanega izraza $\frac{\dot{y}_{ss}T_{za}}{y_{za}}$ najprej red n, nato razmerje $\frac{T_{za}}{T}$, iz tega pa časovno konstanto T in uporabimo model, ki ga podaja enačba 7.13.

7.3 Vključitev mrtvega časa v modeliranje

Pri opazovanju merjenih procesnih signalov pa dostikrat opazimo, da se kljub izvedeni spremembi vhodnega signala izhodni signal še nekaj časa ne spreminja. To se zlasti kaže v procesih, ki imajo t.i transportne zakasnitve (npr. odpremo nek ventil v hidravličnem sistemu, vendar tekočina šele čez nekaj časa začne polniti rezervoar zaradi cevnega sistema). Zato moramo neposredno po spremembi vzbujanja zelo natančno opazovati odziv in ugotoviti, koliko časa se le-ta ne spremeni. Ta čas imenujemo mrtvi čas T_m. Uporabimo ga lahko v vseh obravnavanih pro-



Slika 7.8: Diagrama za določitev n in T pri I_n modelu

porcionalnih in integrirnih modelih. Ker je prenosna funkcija sistema z idealnim mrtvim časom

$$G(s) = e^{-T_m s} \tag{7.15}$$

vse prej navedene modele ustrezno razširimo v oblike

$$G(s) = K\ e^{-T_m s} \tag{7.16}$$

$$G(s) = \frac{K}{Ts+1}\ e^{-T_m s} \tag{7.17}$$

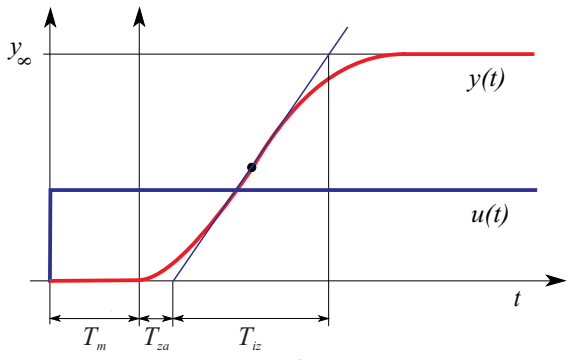
$$G(s) = \frac{K}{(T_1 s+1)(T_2 s+1)}\ e^{-T_m s} \tag{7.18}$$

$$G(s) = \frac{K}{(Ts+1)^n}\, e^{-T_m s} \tag{7.19}$$

$$G(s) = \frac{K_I}{s}\, e^{-T_m s} \tag{7.20}$$

$$G(s) = \frac{K_I}{s(Ts+1)^n}\, e^{-T_m s} \tag{7.21}$$

Mrtvi čas T_m torej določimo na začetku postopka in ga nato izločimo iz nadaljnega ugotavljanja modela tako, da ordinatno os premaknemo na mesto trenutka, ko se je odziv začenjal spreminjati. Od takrat naprej uporabimo vse opisane postopke. Primer proporcionalnega sistema z mrtvim časom prikazuje slika 7.9.



Slika 7.9: Odziv proporcionalnega sistema z dodatnim mrtvim časom

Zlasti če velja $T_{za} << T_{iz}$, je upravičeno, da T_{za} prenesemo v nadomestni mrtvi čas T_{mn}

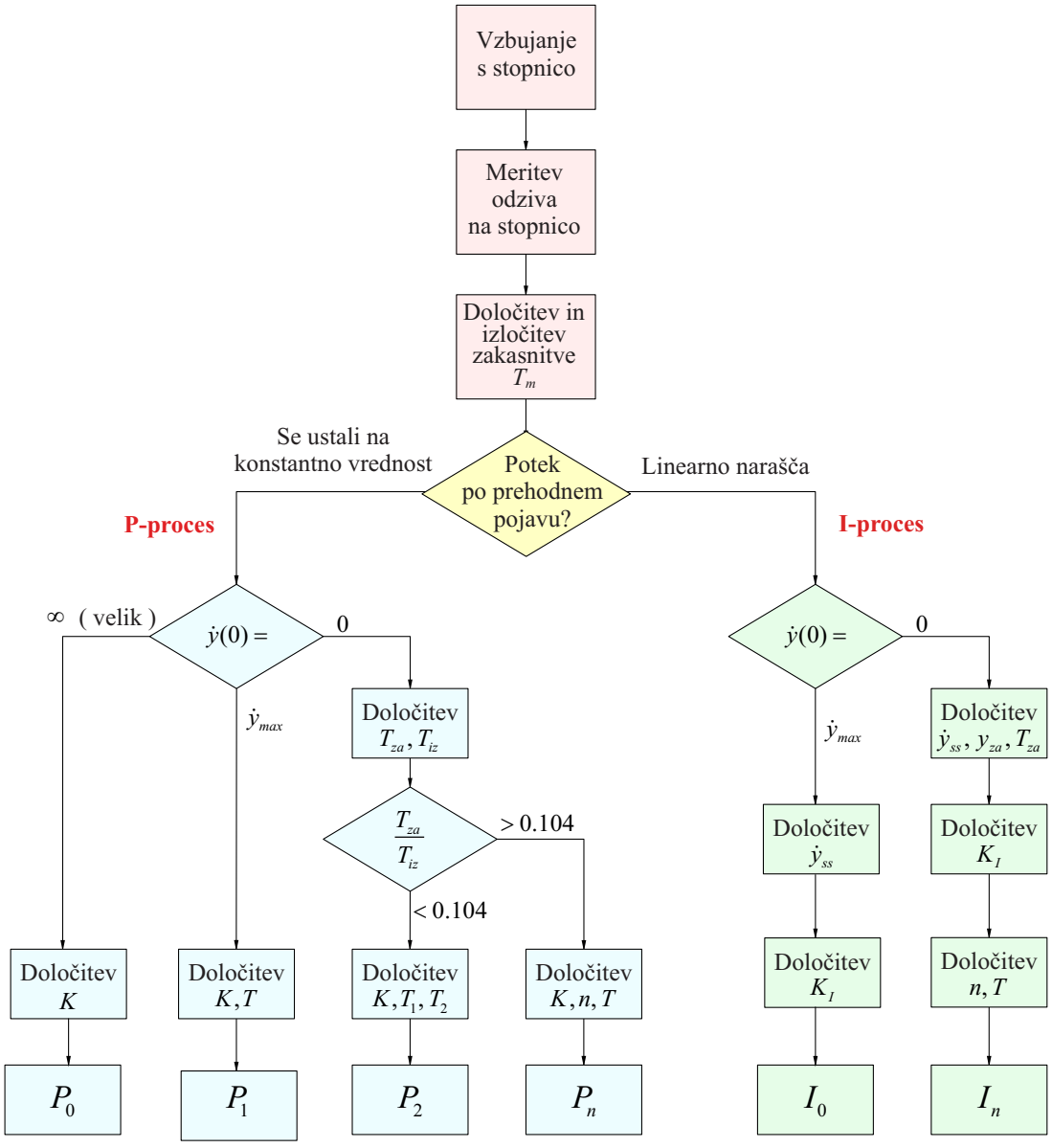
$$T_{mn} = T_m + T_{za} \tag{7.22}$$

in uporabimo model

$$G(s) = \frac{K}{T_{iz}s+1}\, e^{-T_{mn} s} \tag{7.23}$$

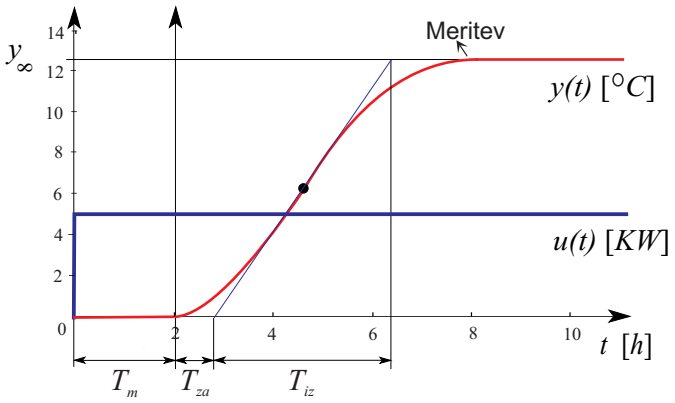
7.4 Diagram poteka inženirskega pristopa v eksperimentalnem modeliranju

Opisani postopki so v literaturi poznani kot Strejčeva metoda [Strejc, 1959] in so zbrani v diagramu poteka na sliki 7.10.



Slika 7.10: Potek določevanja parametrov modelov za P in I procese po Strejcu

Primer 7.1 Slika 7.11 prikazuje meritev temperature v procesu ogrevanja. Grelo z močjo $0.5KW$ ogreva podhlajen prostor, ki se v približno 10h ogreje za $12.5°C$. (iz $10°C$ na $22.5°C$). Obravnavane metode želimo uporabiti za razvoj modela.



Slika 7.11: Merjena temperatura toplotnega procesa in označene značilke

Upoštevamo, da je $10°C$ temperatura delovne točke in linearni model bo prikazoval razmere okoli te temperature. Iz slike določimo naslednje značilke:

$$u = 0.5$$
$$y_\infty = 12.5$$
$$T_m = 2$$
$$T_{za} = 0.74$$
$$T_{iz} = 3.52$$

Očitno moramo uporabiti proporcionalni model. Izračunamo

$$\frac{T_{za}}{T_{iz}} = 0.21$$

Ker je vrednost večja 0.104, uporabimo P_n model.

S pomočjo diagramov 7.5 izberemo iz $\frac{T_{za}}{T_{iz}} = 0.21$ $n = 3$. Določimo $\frac{T_{za}}{T} = 0.82$ in

$$T = 0.90$$

.

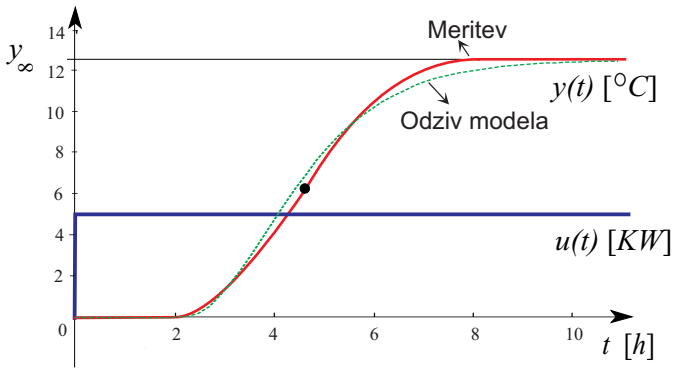
Ojačenje pa je

$$K = \frac{y_\infty}{u} = 25$$

Torej je model

$$G(s) = \frac{25}{(0.90s+1)^3}\, e^{-2s}$$

Slika 7.12 prikazuje merjeno temperaturo in odziv modela in potrjuje uspešnost eksperimentalnega modeliranja.



Slika 7.12: Merjena temperatura toplotnega procesa in odziv modela

□

7.5 Inženirsko razumevanje odzivov in poenostavljanje modelov

Za računanje odzivov in tudi za poenostavljanje modelov je na voljo zelo veliko računalniških orodij. Vendar je za bolj inženirsko razumevanje pomembno vedeti, kako vplivajo v modelu poli in ničle pri linearnih oz. lineariziranih modelih in predvsem razumeti vlogo dominantnih polov (oz. lastnih vrednosti, časovnih konstant).

7.5.1 Vpliv polov in ničel na časovni odziv

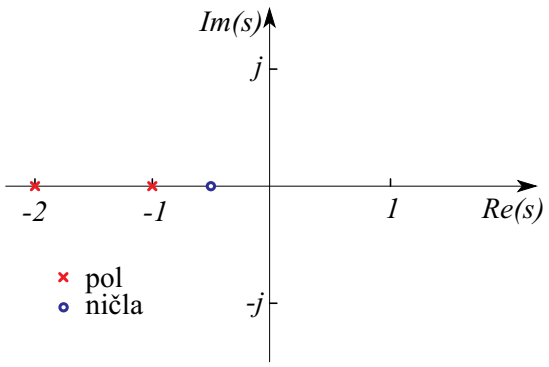
Prenosna funkcija kvocient dveh polinomov spremenljivke s. Oglejmo si sistem, ki ga opisuje diferencialna enačba

$$\ddot{y} + 3\dot{y} + 2y = 2\dot{u} + u \tag{7.24}$$

oz. prenosna funkcija

$$G(s) = \frac{Y(s)}{U(s)} = \frac{B(s)}{A(s)} = \frac{2s+1}{s^2+3s+2} = \frac{2(s+\frac{1}{2})}{(s+1)(s+2)} \tag{7.25}$$

Če predpostavljamo, da $B(s)$ in $A(s)$ nimata skupnih korenov (to je običajen primer), potem vrednosti s, za katere velja $A(s) = 0$, naredijo vrednost prenosne funkcije $G(s)$ neskončno. Te vrednosti s imenujemo pole prenosne funkcije $G(s)$. V našem primeru sta pola pri $s = -1$ in $s = -2$. Vrednosti s, za katere pa velja $B(s) = 0$, pa naredijo vrednost prenosne funkcije $G(s) = 0$, zato jih imenujemo ničle prenosne funkcije. V našem primeru je ničla pri $s = -\frac{1}{2}$. Poli in ničle do multiplikativne konstante natančno določajo prenosno funkcijo, zato le to lahko predstavimo s sliko polov in ničel v ravnini s, kar prikazuje slika 7.13. Tudi to je torej možna predstavitev sistema.



Slika 7.13: Prikaz polov in ničel v s ravnini

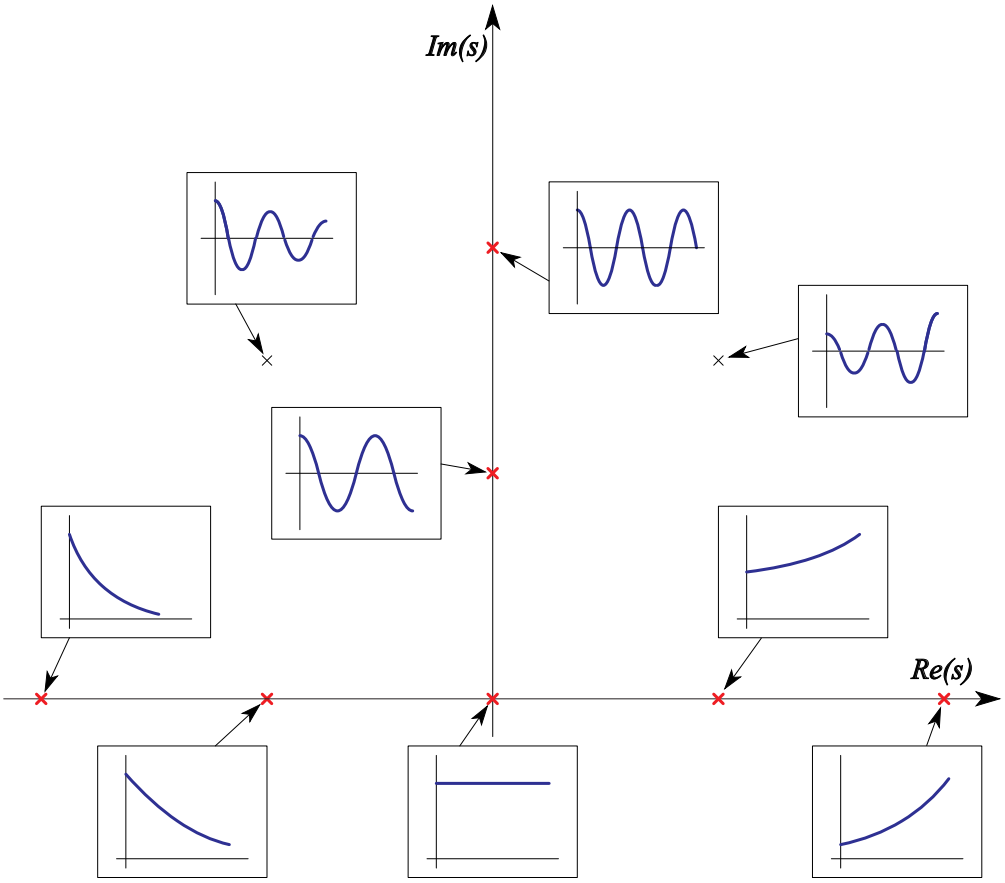
S pomočjo inverzne Laplace-ove transformacije izračunajmo iz prenosne funkcije odziv sistema na δ impulz. Slednje omogoča analizo vpliva polov in ničel na časovni odziv. Prenosno funkcijo $G(s)$ razvijemo v parcialne ulomke

$$G(s) = \frac{2s+1}{(s+1)(s+2)} = \frac{-1}{s+1} + \frac{3}{s+2} \tag{7.26}$$

Ustrezni inverzni Laplace-ov transform pa se glasi

$$g(t) = -e^{-t} + 3e^{-2t} \tag{7.27}$$

Vidimo, da je oblika eksponencialnih funkcij, ki sestavljata odziv, odvisna le od polov pri $s = -1$ in $s = -2$. To velja v splošnem tudi za bolj kompleksne sisteme. Ničla sistema, ki oblikuje števec prenosne funkcije, pa skupaj s poli vpliva le na uteži, preko katerih oba člena (e^{-t} in e^{-2t}) vplivata na končni odziv. Slika 7.14 prikazuje vpliv lege polov na naravni odziv sistema.



Slika 7.14: Vpliv polov na naravni odziv sistema

Ker člen e^{-2t} izzveni hitreje kot člen e^{-t}, pravimo, da je pol pri $s = -2$ hitrejši kot pol $s = -1$. Zato včasih govorimo o hitrih in počasnih polih. Zapomniti si velja, da poli, ki so bolj odmaknjeni od imaginarne osi, predstavljajo prehodni pojav, ki hitreje izzveni.

7.5.2 Dominantni poli

Poli v prenosni funkciji so dominantni, če pretežno vplivajo na odziv sistema. V primeru, da sistem nima ničel, je relativna dominantnost definirana kot razmerje realnih delov polov. Če je razmerje realnih delov polov večje kot 4 in če v bližini pola, ki je bliže koordinatnemu izhodišču, ni ničel, potem pol, ki je bližje koordinatnemu izhodišču, dominantno vpliva na časovni potek, kajti predstavlja prehodni pojav, ki počasneje izzveni. Take pole imenujemo dominantni poli. Često nastopajo v konjugirano kompleksnih parih. Čeprav je koncept dominantnih polov koristen pri oceni dinamičnega obnašanja sistemov, pa moramo biti previdni in se prepričati, ali veljajo vse omenjene predpostavke.

Ker so časovne konstante obratne vrednosti polov, seveda lahko govorimo tudi o dominantnih časovnih konstantah, ki morajo biti po vrednosti bistveno večje (npr. vsaj 4x) od drugih časovnih konstant. Časovne konstante pa vpeljemo le v primeru realnih polov, ko imajo ilustrativen fizikalni pomen.

Na primeru prenosne funkcije

$$G(s) = \frac{2.5}{(s+1)(5s+1)} \tag{7.28}$$

ki ima dominantno časovno konstanto 5 časovnih enot (oz dominanten pol pri $s = 0.2$), si oglejmo, kakšne so možnosti poenostavljanja.

Zanemarimo nedominantne časovne konstante

Nedominantno časovno konstanto izpustimo iz zapisa. Čim večje je razmerje med obema konstantama, bolj je taka poenostavitev upravičena. Pri tem pazimo, da ojačenje prenosne funkcije ($G(0)$)ostane po poenostavitvi enako, kot pred poenostavitvijo. Tu je potrebno zlasti paziti, če imamo običajno faktorizirano obliko po polih in ničlah, saj tam ojačenje posameznega gradnika ni 1 kot v primeru 7.28.

Rezultat poenostavljanja je prenosna funkcija

$$G(s) = \frac{2.5}{5s+1} \tag{7.29}$$

Majhne časovne konstante modeliramo z nadomestnim mrtvim časom

Majhne časovne konstante lahko seštejemo in jih zamenjamo z nadomestnim mrtvim časom. To je zlasti učinkovito, če je razmerje med majhnimi in večjimi vsaj 1:20. Večje (dominantne) časovne kontante pa ohranimo v modelu. V našem primeru torej pridemo do modela

$$T_{mn} = 1 \tag{7.30}$$

$$G(s) = \frac{2.5}{5s+1}e^{-1s} \tag{7.31}$$

Poenostavitev s pomočjo eksperimentalno dobljene tabele

Če pa sta časovni konstanti bliže skupaj

$$T_1 = (0,05 \div 1)T_2 \tag{7.32}$$

in ima model še mrtvi čas T_m, sta parametra prehodnega pojava časovna konstanta T in nadomestni mrtvi čas T_{mn}

$$T = T_1 + T_2 \tag{7.33}$$

$$T_{mn} = T_m + cT_1 \tag{7.34}$$

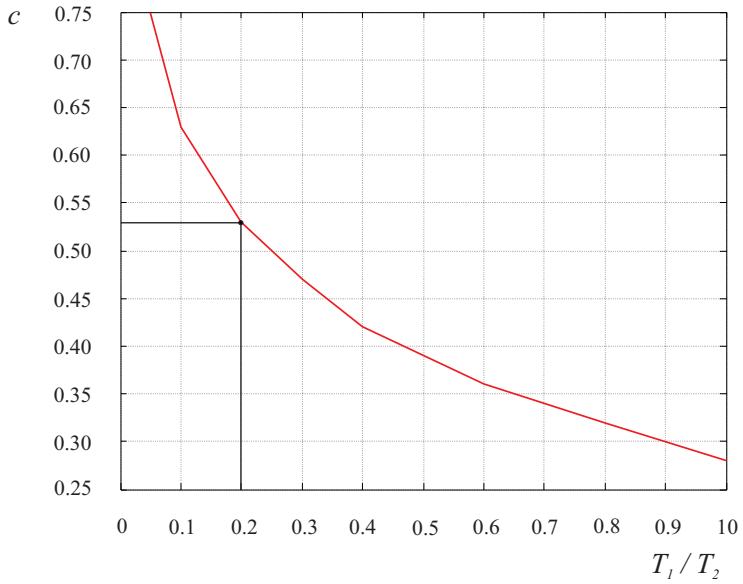
T_1 je manjša časovna konstanta, c pa določimo s pomočjo eksperimentalno dobljene tabele

ali slike 7.15.

V našem primeru je $\frac{T_1}{T_2} = 0.2$ in $c = 0.53$.

$\frac{T_1}{T_2}$	0.05	0.1	0.2	0.3	0.4	0.6	0.8	1
c	0.75	0.63	0.53	0.47	0.42	0.36	0.32	0.28

Tabela 7.1: Tabela za določitev konstante c



Slika 7.15: Diagram za določitev konstante c

Torej je predlagani model

$$T = T_1 + T_2 = 6 \tag{7.35}$$

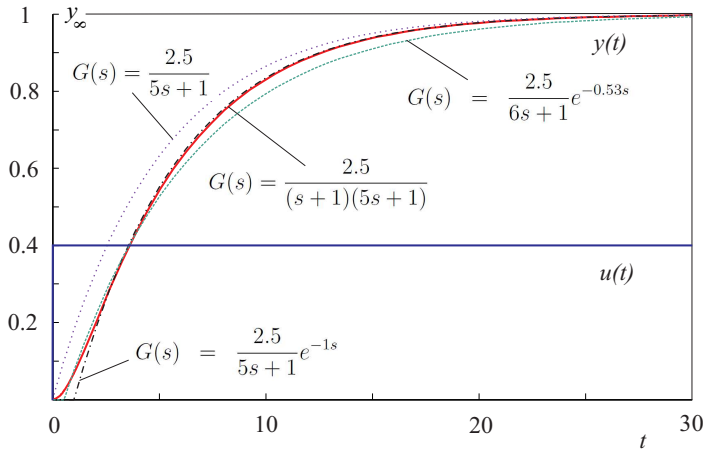
$$T_{mn} = cT_1 = 0.53 \tag{7.36}$$

$$G(s) = \frac{2.5}{6s+1} e^{-0.53s} \tag{7.37}$$

Slika 7.16 prikazuje odziv originalnega sistema in vseh treh poenostavitev.

V kolikor imamo v modelu več časovnih konstant, lahko postopek večkrat uporabimo.

Poudariti je potrebno, da v primeru dveh časovnih konstant nadomestitev z mrtvim časom ni smiselna, ker to ne predstavlja nikakršne poenostavitve. Poenostavitev je smiselna, če je v modelu že mrtvi čas in le-tega ustrezno povečamo.



Slika 7.16: Odziv origialnega in poenostavljenih sistemov

7.5.3 Vpliv ničel na 'dominantnost' polov

Na primeru dveh prenosnih funkcij si oglejmo, kako zaradi ničle postane pol, ki je sicer bolj oddaljen od koordinatnega izhodišča, vplivnejši. Prva prenosna funkcija, ki vsebuje le dva pola, se glasi

$$G_1(s) = \frac{2}{(s+1)(s+2)} = \frac{2}{s+1} - \frac{2}{s+2} \tag{7.38}$$

Druga naj ima dodatno ničlo pri $s = -1.1$ a enako ojačenje ($G_1(0) = G_2(0)$)

$$G_2(s) = \frac{2(s+1.1)}{1.1(s+1)(s+2)} = \frac{0.18}{s+1} + \frac{1.64}{s+2} \tag{7.39}$$

Ničla pri $s = -1.1$ precej zmanjša vpliv pola pri $s = -1$, kar se vidi v tem, da se ustrezna utež v enačbi (7.38) zmanjša na 0.18 v enačbi (7.39), kajti ničla delno krajša pol. Ničla pri $s = -1$ pa bi povsem izničila vpliv pola.

Opisana analiza je zelo primerna v kompleksnejših sistemih, saj dobimo člene, ki malo vplivajo na vedenje sistema. Tako lahko pridemo do enostavnejših modelov.

Če splošno prenosno funkcijo opišemo z enačbo

$$G(s) = \frac{R_1}{s+\sigma_1} + \frac{R_2}{s+\sigma_2+j\omega_2} + \frac{\bar{R}_2}{s+\sigma_2-j\omega_2} + \cdots \tag{7.40}$$

potem imajo majhen vpliv na vedenje sistema tisti členi, katerih $\frac{|R_j|}{\sigma_j}$ so mnogo manjši od drugih. Ugotovili smo namreč, da pol tem bolj vpliva na časovni odziv, čim večja je pripadajoča utež in čim bliže je koordinatnemu izhodišču.

Pri poenostavljanju imamo tri možnosti:

a) Zanemarimo člen, ki malo vpliva na časovni odziv.

b) Člen, ki malo vpliva na časovni odziv, upoštevamo z njegovim ojačenjem $\frac{R_j}{\sigma_j \pm j\omega_j}$.

c) Zanemarimo člen, ki malo vpliva na časovni odziv, ojačenje preostalega dela pa spremenimo tako, da je enako, kot je bilo pred poenostavljanjem.

Ker v primeru prenosne funkcije, ki jo podaja enačba (7.39), velja

$$\frac{R_1}{\sigma_1} = 0.18 \ll \frac{R_2}{\sigma_2} = 0.82 \tag{7.41}$$

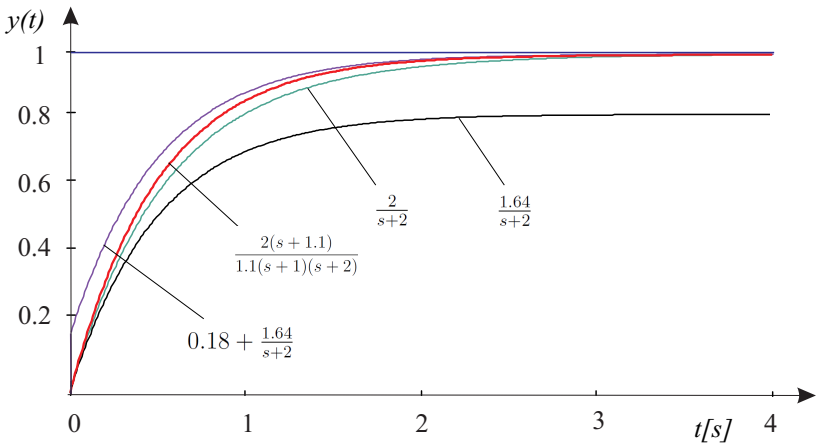
lahko prenosno funkcijo (7.39) poenostavimo v skladu s tremi navedenimi možnostmi:

a) $G_2(s) \doteq \frac{1.64}{s+2}$

b) $G_2(s) \doteq 0.18 + \frac{1.64}{s+2}$

c) $G_2(s) \doteq \frac{2}{s+2}$

Slika 7.17 prikazuje odziv nepoenostavljenega sistema ter odzive treh poenostavljenih modelov na stopničasti vhodni signal velikosti 1. Model $G_2(s) \doteq \frac{1.64}{s+2}$ se dobro prilega nepoenostavljenemu modelu v začetku prehodnega pojava, v ustaljenem stanju pa ima zaradi spremenjenega ojačenja veliko odstopanje. Model $G_2(s) \doteq 0.18 + \frac{1.64}{s+2}$ se dobro ujema v ustaljenem stanju, v začetku prehodnega pojava pa je precejšnje odstopanje. Poenostavitev $G_2(s) \doteq \frac{2}{s+2}$ je v tem primeru še najbolj upravičena, saj omogoča dobro ujemanje v celotnem področju. V splošnem pa je izbira načina poenostavitve odvisna od vrste in namena modela.



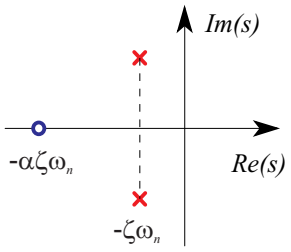
Slika 7.17: Odzivi originalnega in poenostavljenega modela na stopničasto vzbujanje

7.5.4 Učinkovanje dodatne ničle ali dodatnega pola

Ničle v prenosni funkciji vplivajo le na uteži eksponentnih členov, katerih oblika je odvisna le od polov. Prav tako smo ugotovili, da so dominantni poli tisti, ki so bliže koordinatnemu izhodišču, vendar v bližini ne sme biti ničel. Oglejmo si, kako vpliva dodatna ničla na vedenje sistema, ki ima konjugirano kompleksni par polov (obravnavani sistem drugega reda za $0 < \zeta < 1$, [Zupančič, 2012]). Uporabimo prenosno funkcijo v normirani obliki

$$G(s) = \frac{\frac{s}{\alpha\zeta\omega_n} + 1}{(\frac{s}{\omega_n})^2 + 2\zeta\frac{s}{\omega_n} + 1} \tag{7.42}$$

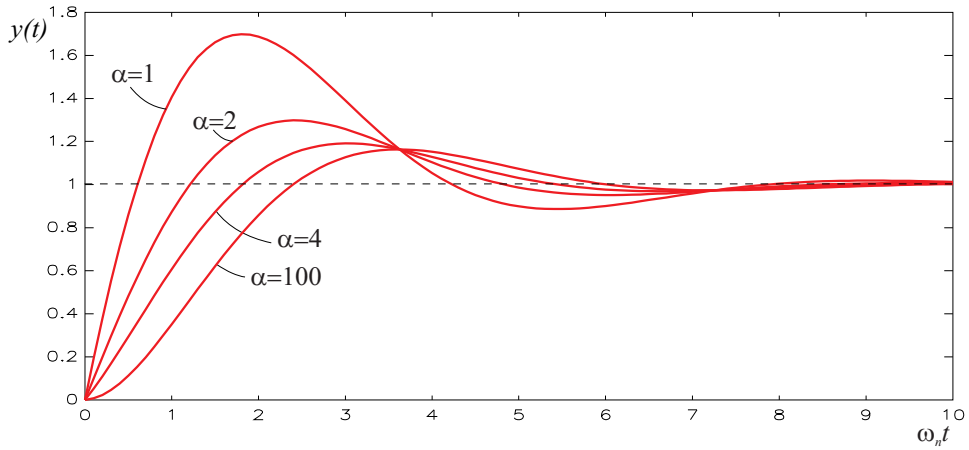
z ničlo in poloma, kot prikazuje slika 7.18. Ničla leži pri $s = -\alpha\zeta\omega_n = -\alpha\sigma$.



Slika 7.18: Lega ničle in polov v s ravnini

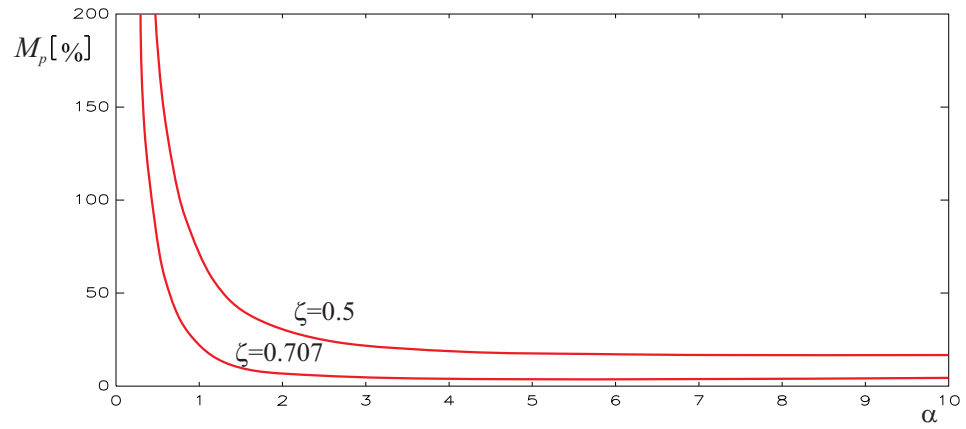
Če je α velik, je ničla daleč od polov in ima majhen vpliv na odziv. Ko pa se α

približuje 1, postaja učinek ničle velik. Slika 7.19 prikazuje vpliv faktorja α na prehodni pojav pri $\zeta = 0.5$.



Slika 7.19: Vpliv dodatne ničle v sistemu drugega reda ($\zeta = 0.5$)

Ničla močno vpliva na povečanje prevzpona, le malo pa na umiritveni čas. Vpliv na prevzpon prikazuje slika 7.20.



Slika 7.20: Vpliv ničle na prevzpon

Pri $\alpha = 0$ je ničla v koordinatnem izhodišču in sistem ima lastnost diferencirnega sistema. Pri $\alpha \geq 4$ ničla ne vpliva na spremembo prevzpona.

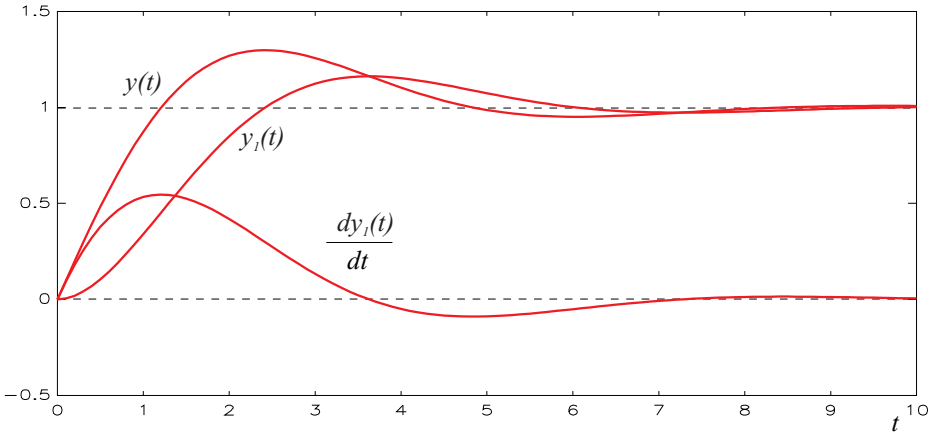
Učinek ničle na spremembo prevzpona lahko nazorno predstavimo, če normalizirano prenosno funkcijo ($\omega_n = 1$)

$$G(s) = \frac{\frac{s}{\alpha\zeta} + 1}{s^2 + 2\zeta s + 1} \tag{7.43}$$

zapišemo v obliki dveh členov

$$G(s) = \frac{1}{s^2 + 2\zeta s + 1} + \frac{1}{\alpha\zeta} \frac{s}{s^2 + 2\zeta s + 1} \tag{7.44}$$

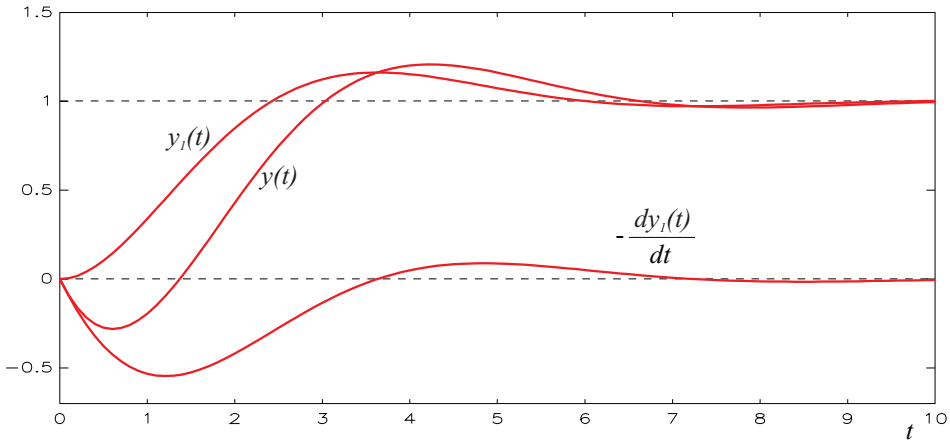
Prvi člen predstavlja obravnavani normirani sistem drugega reda. Drugi člen je odvod prvega člena, pomnožen s konstanto $\frac{1}{\alpha\zeta}$. Slika 7.21 prikazuje odziv sistema drugega reda brez ničle $y_1(t)$, njegov odvod $\frac{dy_1}{dt}$ in njuno vsoto $y(t)$ pri $\zeta = 0.5$, $\omega_n = 1$, $\alpha = 2$ in nazorno prikazuje, zakaj dodatna ničla vpliva na povečanje prevzpona in le malo na spremembo umiritvenega časa.



Slika 7.21: Odziv sistema drugega reda brez ničle, odvod odziva in vsota

Opisana analiza učinkovito prikaže tudi vpliv ničle v desni polravnini. Tak sistem imenujemo sistem z neminimalno fazo. V tem primeru je α negativen ($\zeta = 0.5$, $\omega_n = 1$, $\alpha = -2$), člen z odvodom je potrebno odšteti in nastane značilni odziv sistema z neminimalno fazo. Pri vzbujanju s pozitivnim stopničastim signalom tak sistem v začetku reagira z negativnim izhodnim signalom (z začetnim podnihajem). Ustrezne razmere prikazuje slika 7.22. Takšne sisteme je dokaj težko regulirati.

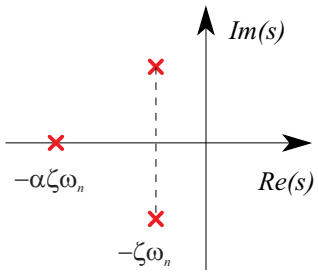
Podobno kot smo analizirali vpliv dodatne ničle, bomo analizirali tudi vpliv dodatnega pola. V tem primeru ima normirana prenosna funkcija naslednjo obliko:



Slika 7.22: Odziv sistema drugega reda z neminimalno fazo

$$G(s) = \frac{1}{(\frac{s}{\alpha\zeta\omega_n} + 1)\left[(\frac{s}{\omega_n})^2 + 2\zeta\frac{s}{\omega_n} + 1\right]} \tag{7.45}$$

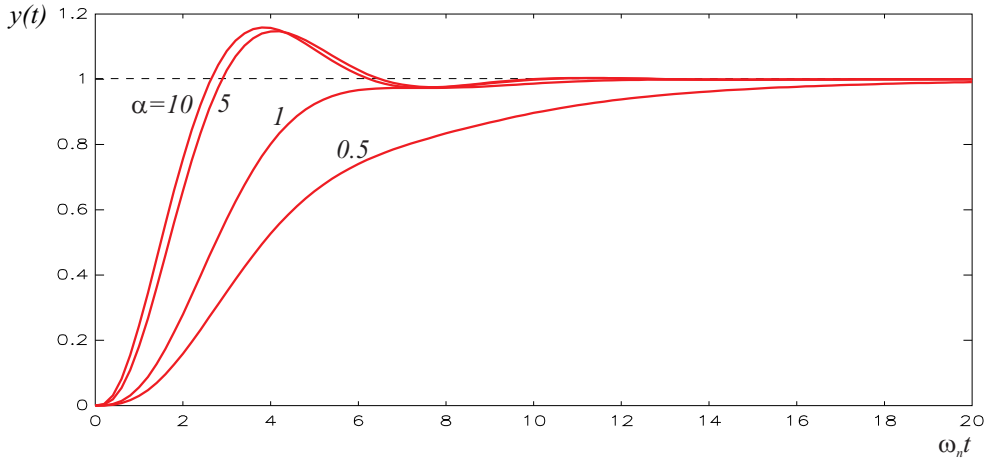
Ustrezno lego polov prikazuje slika 7.23.



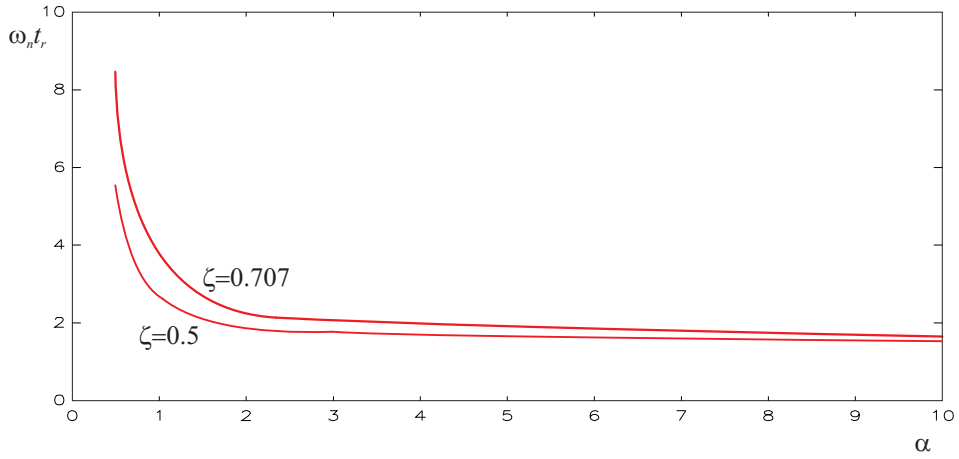
Slika 7.23: Lega polov v s ravnini

Vpliv lege pola pri $s = -\alpha\zeta\omega_n$ na odziv prikazuje slika 7.24 pri dušilnem koeficientu $\zeta = 0.5$.

V tem primeru dodatni pol posebno izrazito vpliva na spremembo prevzpona in časa vzpona. Slika 7.25 prikazuje vpliv lege pola na čas vzpona. V tem primeru smo uporabili za čas vzpona definicijo $10\% - 90\%$, saj obravnavamo tudi aperiodične odzive (pri majhnem α).



Slika 7.24: Vpliv dodatnega pola v sistemu drugega reda na časovni odziv



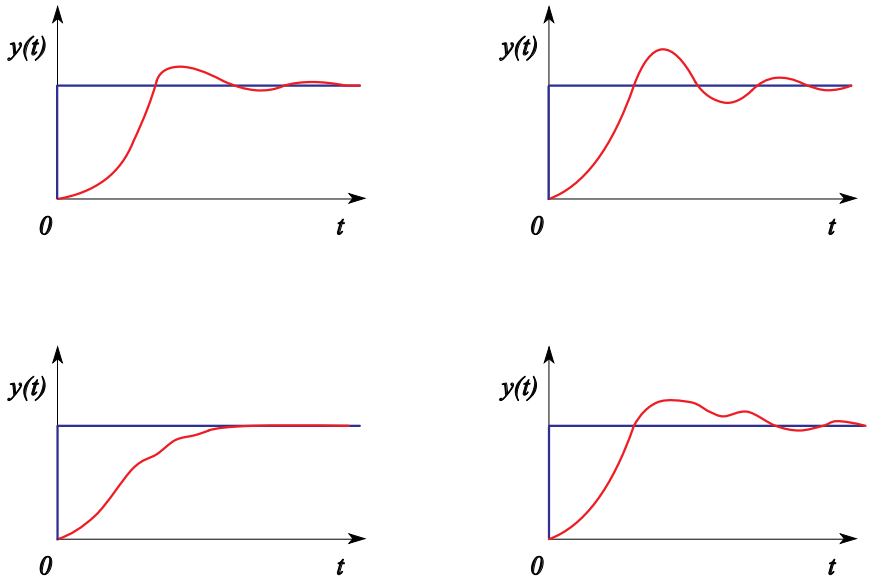
Slika 7.25: Vpliv dodatnega pola na čas vzpona

7.5.5 Sistemi višjega reda

Videli smo, kako vpliva dodatni pol v sistemu drugega reda oz., kako se vede sistem tretjega reda. V splošnem postajajo z večanjem reda analitične rešitve zapletene, zato si za natančne rezultate pomagamo z računalniško simulacijo. Zlasti se analiza zaplete, če imamo v sistemu večkratne pole. Običajen postopek analize je tak, da prenosno funkcijo zapišemo v faktorizirano obliko. Odziv sistema, ki temelji na tej obliki, razbijemo po metodi parcialnih ulomkov na osnovne že obravnavane člene. Taka oblika tudi pokaže, ali je možno odziv poenostaviti. Pri tem lahko zanemarimo pole, ki so daleč od koordinatnega izhodišča ali pa

pole, ki so "delno" krajšani z ničlami in imajo zato majhno pripadajočo utež in s tem majhen vpliv na prehodni pojav.

Odziv stabilnega sistema višjega reda je torej vsota večih eksponencialk in dušenih sinusnih krivulj. Majhna nihanja so superponirana večjim nihanjem ali eksponentnim krivuljam. Komponente, ki hitro izzvenijo, vplivajo le v začetnem delu prehodnega pojava. Slika 7.26 prikazuje tipične odzive sistemov višjih redov pri vzbujanju s stopničastimi signali.



Slika 7.26: Tipični odzivi sistemov višjih redov

Literatura

Aburdene, M.F. (1988), *Digital Continuous System Simulation*, Wm.C. Brown Publishers, Dubuque, Iowa, USA.

Cellier, F.E. (1991), *Continuous System Modeling*, Springer - Verlag, New York, USA.

Cha P. D., Molinder J.I.(2006), *Fundamentals of Signals and Systems: A Building Block Approach*, Cambridge University Press, UK.

Dabney J.B., Harman T.L. (2004), *Mastering SIMULINK* , Prentice Hall, Upper Saddle River, N.J., USA.

Dymola (2011), Multi-engineering modeling and simulation, Users manual, ver. 2012. Dessault System, Dynasim AB, Sweden, Lund.

Elmqvist, H. (1978), *A structured model language for large continuous systems*, Ph. D. diss. Rep. CODEN: LUTFD2/(TFRF-1015), Departement of Automatic Control, Lund Institute of Technology, Sweden.

Karba, R. (1994), *Modeliranje procesov*, Založba Fakultete za elektrotehniko in Fakultete za računalništvo in informatiko, Univerza v Ljubljani, SLO

Korn, G.A. and J.V. Wait (1978), *Digital Continuous System Simulation*, Prentice Hall, Englewood Clifs, N.J., USA.

Matko, D., B. Zupančič, R. Karba (1992), *Simulation and Modelling of Continuous Systems - A Case Study Approach.* Prentice Hall.

Matko, D. (1998): *Identifikacije*, Založba Fakultete za elektrotehniko in Fakultete za računalništvo in informatiko, Univerza v Ljubljani, SLO

Mihelič, F. (2006), *Signali*, Založba Fakultete za elektrotehniko in Fakultete za računalništvo in informatiko, Univerza v Ljubljani, SLO

Mlakar, J. (2002), *Linearna vezja in signali*, Založba Fakultete za elektrotehniko in Fakultete za računalništvo in informatiko, Univerza v Ljubljani, SLO

Modelica (2010), A Unified Object-Oriented Language for Physical Systems Modeling, Language Specification, Version 3.2, Modelica Association, http://www.modelica.org/documents/ModelicaSpec32.pdf

Neelamkavil, F. (1987), *Computer Simulation and Modelling*, John Willey, New York, USA.

Oblak, S., Škrjanc, I. (2008), *Matlab s Simulinkom, priročnik za laboratorijske vaje* , Univerza v Ljubljani, Založba FE in FRI, Slovenija.

Ogata K. (2010), *Modern Control Engineering*, Fifth edition, Prentice Hall, USA.

Schmidt, B. (1987), "What does simulation do? Simulation's place in the scientific method." *Syst. Anal. Model. Simul.* (Benelux journal), 4 (1987) 3, 193-211.

Simulink, Dynamic System Simulation Software (2009), Users manual, R2009b, Math Works, Inc., Natick, MA, ZDA.

Smith, D.J.M (1995), *Continuous System Simulation*, Chapman & Hall, London, UK.

Sodja, A, B. Zupančič (2009), *Modelling thermal processes in buildings using an object-oriented approach and Modelica*. Simulation Modelling Practice and Theory, vol. 17, no. 6, str. 1143-1159.

Strauss, J.C. (1967), "The SCi continuous system simulation language." *Simulation*, no.9, 281-303.

Strejc, V. (1959): *Aproximation aperiodischer Übertragungscharakteristiken*, Regelungstechnik,7:124-128.

Strmčnik, S., R.Hanus, Đ. Juričić, R. Karba, Z. Marinšek, D.Murray-Smith, H. Verbruggen, B. Zupančič (1998):*Celostni pristop k računalniškemu vodenju procesov*, Univerza v Ljubljani, Fakulteta za elektrotehniko, SLO

Zupančič B. (1989), *Sinteza simulacijskega jezika pri računalniško podprtem načrtovanju sistemov avtomatskega vodenja*, Doktorska disertacija, Fakulteta za elektrotehniko in računalništvo, Univerza v Ljubljani, Ljubljana, Slovenija.

Zupančič B. (1992), *SIMCOS- jezik za simulacijo zveznih in diskretnih dinamičnih sistemov*, Fakulteta za elektrotehniko in računalništvo, Univerza v Ljubljani, Slovenija.

Zupančič, B. (2010a): *Simulacija dinamičnih sistemov*, Založba Fakultete za elektrotehniko in Fakultete za računalništvo in informatiko, Univerza v Ljubljani, SLO

Zupančič, B. (2010b), *Zvezni regulacijski sistemi, I.del*, Založba Fakultete za elektrotehniko in Fakultete za računalništvo in informatiko, Univerza v Ljubljani, SLO

Zupančič, B. (2010c), *Zvezni regulacijski sistemi, II.del*, Založba Fakultete za elektrotehniko in Fakultete za računalništvo in informatiko, Univerza v Ljubljani, SLO

Zupančič, B. (2012):*Avtomatsko vodenje sistemov*, delovna verzija učbenika (http://msc.fe.uni-lj.si/Books.asp?book=1), Fakulteta za elektrotehniko, Univerza v Ljubljani, SLO

Howell, J.R. (1993): *Application of Monte Carlo to Heat Transfer Problems, Advances in Heat Transfer*, Academic Press, San Diego, Vol 5, 1-54.

Howell, J.R. (1998): *The Monte Carlo Method in Radiative Heat Transfer*, Journal of Heat Transfer, ASME, Vol. 120, 547-560.

Annino, J. S., E.C. Russel (1979), "The ten most frequent causes of simulation analysis failure - and how to avoid them!" *Simulation*, 137-140.

Araki, M. (1985), *Industrial applications in Japan of computer aided design packages for control systems.* Preprints of the 3 IFAC/IFIP International symposium CADCE '85, Copenhagen, Denmark, 71-76.

Åström, K.J. (1985a), *A Simnon tutorial.* Department of Automatic Control, Lund Institute of Technology, CODEN: LUTFD2/(TFRT-3168)/ (1982), Sweden.

Åström, K.J. (1985b), "Computer aided tools for control system design", in M.J.Jamshidi and C.J. Herget (eds.), *Computer-aided control system engineering*, North-Holland, Amsterdam, Netherlands, 3-40.

Atherton, D.P. (1986), *Simulation in control system design.* Preprints of the IFAC/IMACS International symposium on simulation of control systems, Wien, Austria, 53-59.

Baker, N.J.C., P.J. Smart (1983), "The SYSMOD simulation language". *Proceedings of the* 2st *European simulation congress*, Aachen, W. Germany, 281-286.

Baker, N.J.C., P.J. Smart (1985), "Elements of the SYSMOD simulation language". *Proceedings of the* 11th *IMACS world congress*, Oslo, Norwege, vol.2.

Barker, H.A., P. Townsend, M. Chen, J. Harvey (1988b), "CES - a workstation environment for computer aided design in control systems". *Preprints of the* 4th *IFAC Symposium on Computer aided design in control systems CADCS '88*, China, Beijing, 248-251.

Bausch - Gall, I. (1987), "Continuous system simulation languages". *Veh. Syst. Dyn. (Netherlands)*, vol.16, 347-366.

Bekey, G.A. and W.J. Karplus (1968). *" Hybrid Computation"*, John Wiley, New York, USA.

Blum, J.J. (1969), *Introduction to Analog Computation*, Harcourt, brace & world, inc., New York, USA.

Boullard, L., L.D. Coen (1985), "A multi - microprocessor system for parallel computing in simulation". *Proceedings of the* 11th *IMACS world congress*, Oslo, Norwege, vol.3, 163-166.

Breitenecker, F., Solar, D. (1986), "Models, methods and experiments - modern aspects of simulation languages". *Proceedings of the* 2nd *European simulation congress*, Antwerpen, Belgium, 195-199.

Bosch, P.P.J. Van den, A.J.W. Van den Boom (1985), "Industrial applications in the Netherlands of computer - aided design packages for control systems; trends and practice". *Preprints of the* 3rd *IFAC/IFIP International symposium CADCE '85*, Copenhagen, Denmark, 58-61.

Bosch, P.P.J. Van den (1987), *Simulation program PSI (manual).*Delft university of technology, Netherlands.

Breitenecker, F. (1989). "Need for Hybrid Simulation?", *Proceedings of the* 3rd *European Simulation Congress*, (D. Murray - Smith, J. Stephenson and R.N. Zobel Eds.), Edinburgh, 421-425.

Britt, J.I, J. S. Wareck, J.A. Smith (1991), "A computer - aided process synthesis and analysis environment".In J.J. Siirola, I.E. Grossmann, G. Stephanopoulos (Eds.), *Foundations of Computer - Aided Process Design*, Elsevier, Amsterdam, 381-307

Bruijn, M. A. de, F.P.J. Soppers (1986), "The Delft parallel processor and software for continuous system simulation". *Proceedings of the* 2nd *European simulation congress*, Antwerpen, Belgium, 777-783.

Butcher, J.C. (1964), "Implicit Runge-Kutta processes", *Math. Comp. 18*, 50.

Carlson A., G. Hannauer, T. Carey , P.J. Holsberg (1967), *Handbook of Analog Computation*. Electronic Associates, Inc. Princeton, N.J., USA.

Cellier, F.E. (1979), *Combined continuous / discrete system simulation by use of digital computers, Techniques and tools*. Ph.D. Swiss Federal Institute of Technology, Zürich, Switzerland.

Cellier, F.E. (1983), "Simulation software - today and tomorrow." *Proceedings of the IMACS conference*, Nantes, France.

Cellier, F.E. (1991), *Continuous System Modeling*, Springer - Verlag, New York, 1991, USA.

Chen, S. (1989), *Toward the future. In Supercomputers: directions and technology and applications*, National Academy Press, Washington D.C.,USA, 51-65

Colijn, A.W., P.D. Ariel (1986), "Continuous system simulation languages on supercomputers". *Proceedings of the 1986 Summer Computer Simulation Conference*, Reno, USA, 3-7.

Crosbie, R.E., F.E. Cellier (1982), "Progres in simulation language standards"; *An activity report for Technical Committee TC3 on simulation software. Proceedings of the* 10^{th} *IMACS world congress*, Montreal, Canada, vol.1. 411-412.

Crosbie, R.E.(1984), "Simulation on microcomputer - experiences with ISIM". *Proceedings of the 1984 Summer Computer Simulation Conference*, Boston, Massachusetts, USA, vol.1, 13-17.

Crosbie, R.E., S. Javey, J.S. Hay, J.G. Pearce (1985a), "ESL - a new continuous system simulation language". *Simulation* , vol.44, no.5, 242-246.

Crosbie, R.E. (1986), "Developments in ESL and other's for the 1990's". *Proceedings of the 1986 Summer Computer Simulation Conference*, USA, 313-314.

Cser J., P.M. Brujn, Verbruggen (1986), "Music - a tool for simulation and real - time control". 4^{th} *IFAC/IFIP symposium on the software for computer control, SOCOCO'86*, Graz, Austria, 58-62.

Delebeque, F. and S. Steer (1986), "Some remarks about the design of an interactive CACSD package: The BLAISE experience", in K.L.Lineberry (ed.),*Proc. IEEE* 3^{rd} *Symp. on computer aided control system design*, New York, USA, 48-51.

Hindmarsh, A. C. (1983), "ODEPACK, a systematized collection of ODE solvers", *IMACS Transactions on Scientific Computing*, 1, 55-64.

D'Hollander, E. H. (1985), "A multiprocessor for dynamic simulation". *Proceedings of the 1985 Summer Computer Simulation Conference*, USA, 112-117.

Divjak, S. (1975), *Synthesis of time optimal digital simulation system for continuous dynamical systems*. Ph.D. Faculty of electrical engineering, University of Ljubljana, Yugoslavia.

Duncan, R. (1990), "A survey of parallel computer architectures". *IEEE Computer*, February 1990, 5-16.

EAI Handbook of Analog Computation, (A Carlson, G. Hannauer, T. Carey and P.J. Holsburg Eds.)(1967), Electronic Associates, Inc., West Long Branch, New Jersey, USA.

EAI-580 Analog-Hybrid Computing System - Reference Handbook(1968), West Long Branch, New Jersey, USA.

EAI-2000 Analog Reference Handbook(1982), West Long Branch, New Jersey, USA.

Eckelmann, P. (1987), "The transputer - component for the 5 generation systems". *Proceedings VLSI and computers. First international conference on computer technology, systems and applications*, Hamburg, West Germany, 977.

Elmqvist, H. (1975), *SIMNON, an interactive simulation program for nonlinear systems*. Users manual. Departement of Automatic Control, Lund Institute of Technology, Sweden, Report 7502.

Elmqvist, H. (1977), "SIMNON: an interactive simulation program for nonlinear system". *Proceedings Simulation'77*, Montreux, France.

Elmqvist, H. (1978), *A structured model language for large continuous systems*, Ph. D. diss. Rep. CODEN: LUTFD2/(TFRF-1015), Departement of Automatic Control, Lund Institute of Technology, Sweden.

Elmqvist, H., S.E. Matson (1986), "A simulator for dynamical systems using graphics and equations for modeling". *Proceedings of the IEEE Control System Society. Third symposium on computer aided control system design*, Arlington, USA, 134-139.

Elmqvist, H. (1994), *Dymola - Dynamic Modeling Language*, User's Manual, Dynasim AB, Lund, Sweden.

Fadden E.J (1987), "SYSTEM 100: Time - critical simulation of continuous systems". *Multiprocessor and Array Processor Conference, publishing number 87 02mT15*, San Diego, USA.

Fischlin, A., M. Mansour, M. Rimvall, W. Schaufelberger, (1986), "Simulation and computer aided control system design in engineering education", *IFAC/IMACS International symposium on simulation of control systems*, Wien, Austria, 61-73.

Fishman, G.S. (1973), *"Concepts and methods in discrete event digital simulation"*, John Wiley, New York, USA.

Fritzson P. (2004), *Principles of Object Oriented Modeling and Simulation with Modelica 2.1*, IEEE Press, John Wiley&Sons Inc., Publication, USA.

Gauthier, J.E. (1987), "ACSL and simulators". *Proceedings of the 1987 Summer Computer Simulation Conference*, Orlando, USA, 73-77.

Gear, C.W. (1971), *Numerical Initial Value Problems in Ordinary Differential Equations*, Prentice Hall.

Gear, C.W. (1984), "Efficient step size control for output and discontinuities", *Trans. Soc. Comput. Sim. 1*, 27-31.

Gear, C.W. (1986), "The potential for parallelism in ordinary differential equations". Proc. Int. Conf. *Numerical Mathematics*. 33-48.

Gear, C.W., O. Osterby (1984), "Solving ordinary differential equations with discontinuities". *ACM Trans. Math. Software, 10*, 23-44.

Giloi, W.K. (1975), *Principles of Continuous System Simulation* B.G. Teubner, Stuttgart.

Goforth, R.R., R.M. Crisp (1988), "Simulation with microcomputers: an overview". *Acces, a journal of microcomputer applications*, feb. 1988, vol.7, no.1, 4-14.

Golden, D.G. (1985), "Software engineering considerations for the design of simulation languages". *Simulation*, vol.45, no.4, 169-178.

Grierson, W.O. (1986), "Perspectives in simulation hardware and software architecture". *Modeling, Identification and Control (norwegian research bulletin)*, vol.6 ,no. 4, 249-255.

Gustaffson, K. (1988), *Stepsize control in ODE-Solvers-Analysis and Synthesis*, Licentiate Thesis TFRT-3199, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden

Gustaffson, K. (1990), "Using control theory to improve stepsize selection in numerical integration of ODE", *Preprints of* 11^{th} *IFAC World Congress*, Tallin, Estonia, USSR, vol. 10, 139-144.

Hairer, E., C. Lubich (1988), "Extrapolation at stiff differential equations", *Numer. Math.*, Vol. 52, 377-400.

Hairer, E., G. Wanner (1990), *Solving Ordinary Differential Equations II. Stiff and Differential–Algebraic Problems* , Springer Verlag.

Hall, G., J.M. Watt (1976), *Modern numerical methods for ordinary differential equations*, Clarendon Press, Oxford.

Hallin, H.J., S.A.R. Hepner (1984), "Solving benchmark problems with PSCSP and other simulation languages". *Proceedings of the 1984 Summer Computer Simulation Conference*, Boston, Massachusetts, USA, vol.1, 3-8.

Hallin, H.J (1988), "Simulation im Zeitalter von Supercomputern und Mini-Supercomputern". *Proceedings 5. Symposium Simulationstechnik*, Aachen, W. Germany, 2-13.

Hamblen, J.O. (1987), "Parallel continuous system simulation using transputer". *Simulation, vol.49, no.6*, 249-253.

Havranek, W.A. (1983), "Simulation in the 80s". *Proceedings of the 1883 Summer Computer Simulation Conference*, Vencouver, Canada, vol.1, 523-528.

Hay, J.L., R.E. Crosbie (1981), *Outline proposal for a new standard for continuous system simulation language (CSSL 81)*. Computer Simulation Centre, Departement of Electrical Engineering, University of Salford, USA.

Hay, J.L., R.E. Crosbie (1984), "ISIM - a simulation language for microprocessors". *Simulation* , sep. 1984, USA, 133-136.

Hay, J.L., R.E. Crosbie (1986), "Parallel processor simulation with ESL". *Proceedings of the 1986 Summer Computer Simulation Conference*, Reno, USA, 959-964.

Heinrich, R. (1986), "Simulation in control engineering". *Proceedings of the* 2^{nd} *European simulation congress*, Antwerpen, Belgium, 10-13.

Herget, C.J. (1988), "Survey of existing computer-aided design in control systems packages in the United States of America". *Preprints of the* 4^{th} *IFAC Symposium on Computer aided design in control systems CADCS '88*, China, Beijing, 46-50.

Hooper, J.W. (1987), "Strategy- related characteristics of discrete event languages and models". *Simulation*, vol. 46, no. 4, 153-158.

Huber, R.M., A. Guasch (1985), "Towards a specification of a structure for continuous system simulation languages". *Proceedings of the* 11^{th} *IMACS world congress*, Oslo, Norwege, 109-113.

Jackson, A.S. (1960), *Analog Computation*, McGraw Hill, London.

Karba, R., B. Zupančič, F. Bremšak, A. Mrhar and S. Primožič (1990), "Simulation tools in pharmacokinetic modelling", *Acta Pharm. Jugosl.*, Vol. 40, 247-262.

Karplus, W.J. (1984a), "Selection criteria and performance evaluation methods for peripheral array processors". *Simulation*, vol.43, no.3, 125-131.

Karplus, W.J. (1984b), "The changing role of peripheral array processors in continuous systems simulation", *Proceedings of the 1984 Summer Computer Simulation Conference*, Boston, Massachusetts, USA, vol.1, 1-13.

Kettenis, D.L. (1986), "COSMOS: a member of a new generation simulation languages". *Proceedings of the* 2^{nd} *European simulation congress*, Antwerpen, Belgium, 263-269.

Kleinert, W., D. Solar, F. Berger (1983), "Status report on TU Vienna's hybrid time sharing system". *Proceedings of the* 2^{st} *European simulation congress*, Aachen, W. Germany, 193-200.

Kleinert, W., M. Graff, R. Karba, B. Zupančič (1988), "Simulation einer Destillationskolonne - Modellierung mit SIMCOS und Vergleich der Ergebnisse von ACSL und SIMSTAR Simulationen". *Proceedings of the* 5^{th} *symposium Simulationstechnik*, Aachen, W. Germany, 254-259.

Korn, G.A. (1983a), "A wish for simulation - language specifications". *Simulation* ,jan. 1983, USA.

Korn, G.A. (1983b), "Direct - executing languages for interactive simulation and computer - aided experiments." *Proceedings of the* 2^{nd} *European simulation congress*, Aachen, W. Germany, 225-233.

Koskossidis, D.A. and C.J. Brennan (1984), "A Review of Simulation Techniques and Modelling", *Proceedings of the 1984 Summer Computer Simulation Conference (W.D. Wade ed.)*, Vol.1, Boston, USA, 55-58.

Landauer, J, P. (1988), *EAI STARTRAN environment.* Users manual, Electronic Associates, Inc., West Long Branch, New Jersey, USA.

Landauer, J.P. (1988b), "Real-time simulation of the Space Shuttle main engine on the SIMSTAR multiprocessor". *Proceedings of the SCS Multiconference on Aerospace Simulation III*, San Diego, USA.

Lapidus, L., J.H. Seinfeld (1971), *Numerical Solution of Ordinary Differential Equations*, Academic Press New York and London.

Lincoln, A. (1988), A modular parallel computer system for high perfomance real-time simulation. *Proceedings of the* 12th *IMACS world congress*, Paris, France, vol.2., 619-621.

Marquardt, W. (1991), "Dynamic process simulation - recent progress and future challenges." *Preprints of CPC IV, Forth International Conference on Chemical Process Control*, South Padre Island, Texas, USA.

Matko, D., B. Zupančič, S. Divjak (1989), "New concepts in control system simulation". *Proceeding of the* 3rd *European Simulaton Congress*, Edinburgh, Scotland, 444-446.

$PC - MATLAB^{TM}$ for MS-DOS personal computers, The MathWorks Inc. SouthNatick, USA (1989).

Mc Leod, J.P.E. (1986a), "Computer modelling and simulation: the changing chalenge." *Simulation*, vol.46, no.3, 114-118.

Mc Leod, J.P.E. (1987b), "What is simulation?" (Simulation in the service of society). *Simulation*, 219-221.

Mitchel & Gauthier, Assoc. (1981), *ACSL: advanced continuous simulation language.* (user guide / reference manual).

Myers, W. (1990), "Parallel programming: views from the trenches." *IEEE Software, jan. 1990.*

Nilsen, N.R. (1984), *The CSSL IV simulation language (reference manual).* Simulation Service, Chatsworth, California, USA.

Nilsen, N.R. (1985), "Recent advances in CSSL IV." *Proceedings of the* 11th *IMACS world congress*, Oslo, Norwege, vol.3, 101-103.

Ören, T.I. (1974),"A bibliography of bibliographies on modelling, simulation and programing", *Simulation*, Vol.23, No.3, 90-95 and Vol.23, No.4, 115-116.

Ören, T.I., B.P. Ziegler (1979), "Concepts for advanced simulation methodologies." *Simulation*, vol.32, no.3.

Pearce, J.G., P. Holliday, J.O. Gray (1985), "Survey of parallel processing in simulation." *Proceedings of the* 2nd *European workshop on parallel processing techniques for simulation*, Manchester, United Kingdom, 183-202.

Press, W.H., B.P., Flannery, S.A.,Teukolsky, W.T. Vetterling (1986), *"Numerical recipes - The Art of Scientific Computing"*, Cambridge University Press, UK.

Pritsker, A.A.B (1979), "Comparisons of definitions of simulation." *Simulation* ,vol.33, no.2, 61-63.

Rimvall, M., F.E. Cellier (1985), "The matrix environment as enhancement to modelling and simulation." *Proceedings of the* 11th *IMACS world congress*, Oslo, Norwege, vol.3., 93-96.

Rimvall, M., F.E. Cellier (1986), "Evaluation and perspectives of simulation languages following the CSSL standard." *Modeling, Identification and Control (norwegian research bulletin), vol.6 ,no. 4*, 181-199.

Saucedo R., Schirring E. E. (1986), *Introduction to Continuous and Digital Control Systems*, Mcmillan Applied System Science, New York, USA.

Schmid, C. (1988), "Tehniques and tools of CACDS." *Preprints of the* 4^{th} *IFAC Symposium on Computer aided design in control systems CACDS '88*, China, Beijing, 67-75.

Schmidt, G. (1980), *Simulationstechnik*. R. Oldenbourg, München, W. Germany.

Schmidt, B. (1986), "Classification of simulation software." *Syst. Anal. Model. Simul. (Benelux journal), 3 (1986) 2*, 133-140.

Schmidt, A., F. Scheider (1988), "Erfahrungen mit Hardware in-the-loop Simulation an der Workstation XANALOG XA-1000". *Proceedings of the* 5th *symposium Simulationstechnik*, Aachen, W. Germany, 2-13.

Schrage, M.H., D.F. Mc Ardle (1986), " New array processor continuous simulation system uses tactile sensing icon programming." *Proceedings of the 1986 Summer Computer Simulation Conference*, Reno, USA, 138-142.

Schumann A., D. Matko and B. Zupančič (1991), "Simulation of a diesel engine using a digital simulation language," MELECON '91, Ljubljana, Slovenia.

Shah, M., J. (1988), *Engineering simulation: tools and applications using the IBM PC family.* Prentice Hall, Englewood Cliffs, New Jersey, USA.

Shneiderman, B. (1987), *Designing the user interface.* Addison - Wesley Publishing Company, USA.

Simulink, Dynamic System Simulation Software (2009), Users manual, R2009b, Math Works, Inc., Natick, MA, ZDA.

Smith, J.M. (1977), *Mathematical modeling and digital simulation for engineers and scientists.* John Wiley & Sons, Inc.

Smith, D.J.M (1995), *Continuous System Simulation*, Chapman & Hall, London, UK.

Sodja, A, B. Zupančič (2009), *Modelling thermal processes in buildings using an object-oriented approach and Modelica.* Simulation Modelling Practice and Theory, vol. 17, no. 6, str. 1143-1159.

Spriet, J.A. and G.C. Vansteenkiste (1982), *Computer Aided Modelling and Simulation*, Academic Press, London, U.K.

Steppard, S. (1983), "Applying software engineering to simulation." *Simulation*, 13-19.

Strauss, J.C. (1967), "The SCi continuous system simulation language." *Simulation*, no.9, 281-303.

Syn, W.M., H. Dost (1985), "On the dynamic simulation language (DSL/VS) and its use in the IBM corporation." *Proceedings of the* 11$^{\text{th}}$ *IMACS world congress*, Oslo, Norwege, vol.3, 115-118.

Šega, M., S. Strmčnik, R.Karba and D.Matko (1985), "Interactive program package ANA for system analysis and control design," *Prep.* 3$^{\text{rd}}$ *IFAC int. Symp. CADCE'85*, Copenhagen, 95-98.

Taylor, H., D.K. Friderick, C.M. Rimvall, H.A. Sutherland (1990), "Computer - aided control engineering environments: architecture, user interface, data - base management, and expert aiding." *Preprints of* 11$^{\text{th}}$ *IFAC World Congress*, Tallinn, USSR, vol. 10, 55-65.

Terrel, J.T. (1988), *Introduction to digital filters.* Macmillan Education, Houndmills.

Tesler, L.G. (1989), "Achieving a pioneering outlook with supercomputing." *Supercomputers: directions in technology and applications*, National Academy Press, Washington D.C., USA, 90-95.

Tomovic, R. and W.J. Karplus (1962), *High Speed Analog Computers*, John Wiley, New York, USA.

TUTSIM user's manual for IBM PC computers (1983). Twente University of Technology, Netherlands.

Tyso, A. (1985), "Simulation as a tool in operational safety, reliability and control." *Modeling, Identification and Control* (norwegian research bulletin), vol.6 ,no. 3, 127-140.

Worlton, J. (1989), *Existing conditions. In Supercomputers: directions in technology and applications*, National Academy Press, Washington D.C., 21-50

Ziegler, B.P. (1976), *Theory of modelling and simulation.* John Wiley & Sons, Inc., New York, USA.

Ziegler, B.P. (1987), *Simulation objectives: experimantal frames and validity. System & Control Encyklopedia*, Pergamon Press, 4388-4392.

Zupančič, B., D. Matko, M. Šega, P. Tramte (1986), "Simulation in the program package ANA." *Proceedings of the* 2nd *European simulation congress*, Antwerpen, Belgium, 314-318.

Zupančič, B., D. Matko, R. Karba, M. Atanasijević, Z. Šehić (1991), "Extensions of the simulation language SIMCOS towards continuous - discrete complex experimentation system." *Preprints of the IFAC Symposium CADCS '91* , University of Wales, Swansea, U.K., 351-356.

Paynter H.M. (1961), *Analysis and design of engineering systems.* The M.I.T. Press, Cambridge.

Borutsky, W. (2010), *Bond graph methodology, development and analysis of multidisciplinary dynamic system models.* Springer.