UNIVERSIDADE TÉCNICA DE LISBOA
INSTITUTO SUPERIOR TÉCNICO

Ninteger v. 2.3 Fractional control toolbox for MatLab

Duarte Pedro Mata de Oliveira Valério

User and programmer manual

Beta release — comments and bug reports are welcome

2005 / 08 / 17

Abstract

Ninteger is a toolbox for MatLab intended to help to develop fractional order controllers and assess their performance. This manual contains a thorough description of all files in version 2.3 of this toolbox.

Keywords

Fractional order calculus, fractional order control, MatLab.

Table of contents

Table of figures

1. Introduction

Ninteger is a toolbox for MatLab intended to help developing fractional order controllers and assess their performance. Its code may be freely distributed and altered provided that the source of the code is acknowledged and this warning is kept in all further copies and/or alterations. This manual contains a thorough description of all files in version 2.3 of this toolbox[1].

This version is, like its predecessors, a **Beta release**. Reports of eventual errors (the absence of which the author cannot ensure) and suggestions of improvements will be gladly accepted. Please the author's e-mail given in the Internet site given below.

1.1. Requirements

The requirements for using this toolbox are:

❖ MatLab version 7.0 (release 14) or above. Actually most functions run under version 5, but the graphical interface and the Simulink library do not.

❖ Control toolbox.

❖ Optimisation toolbox.

❖ Map toolbox. Actually this toolbox is only needed because of functions *rad2deg* and *deg2rad*. If these are not available, see the next section.

1.2. Installation procedure

The several files of the toolbox are available through the Internet, compressed into a file with the Zip format, in Matlab File Exchange site (Control Design category)

```
http://www.mathworks.com/matlabcentral/fileexchange
```

and in[2]

```
http://www.gcar.dem.ist.utl.pt/pessoal/dvalerio/ninteger/ninteger.htm
```

To install the toolbox, decompress the file into the location you want, and add folder *ninteger* to the MatLab path together with its subfolders[3].

If functions *rad2deg* and *deg2rad* are not available[4], download them from the same address given above and put them in some folder found in MatLab's path. Alternatively, create them yourself and type their code as found in the Appendix.

[1] A version history is available at the Internet site given below. See (Valério, 2001a, 2001b) about version 1.0, (Valério *et al.*, 2004b) about version 2.1.

[2] This manual is available in this last site only, together with the version history and other data.

[3] In versions 6.5 and 7.0 of MatLab this means choosing *Set path…* from the *File* menu, clicking in button *Add with subfolders…*, selecting the path for folder *ninteger* and pressing *OK*.

[4] You may check if they are by typing *exist('rad2deg')* and *exist('deg2rad')* at the command prompt. The result should be 2 for both. If is it something else, they are not.

1.3. How files are organised

The files of the toolbox were divided into three categories:

❖ Those that are to be accessible to the user are placed in the *ninteger* folder.

❖ However, a separate folder, *cfractions*, was provided for those functions that deal with continued fractions, so that they may be easily distinguished from the other functions. These functions may be useful for purposes other than this toolbox, and thus ought to be accessible to the user.

❖ Another separate folder, called *gui*, was provided for those files necessary to the graphical user interface only.

❖ A last separate folder, *obsolescent*, contains functions that are no longer necessary, but are still included to ensure compatibility with older versions of the toolbox.

1.4. About this manual

This document is organised as follows:

❖ In chapter 2 functions are described that provide integer order transfer functions approximating fractional order transfer functions.

❖ In chapter 3 functions are described fit for implementing controllers that ensure an open-loop dynamic given by

$$G(s)C(s) = ks^{v}, \quad v \in \mathbb{R} \qquad (1.1)$$

where G is the plant to control.

❖ In chapter 4 functions are described that ensure the open-loop's Nichols plot to fall outside an area corresponding to the closed-loop behaviour to avoid. So as to ensure a simple design methodology the transfer functions are normally restricted to expressions like

$$C(s) = k\,\mathrm{Re}\left[s^{z}\right], \quad z \in \mathbb{C} \qquad (1.2)$$

❖ In chapter 5 functions for finding fractional models of plants from its experimental data are found.

❖ In chapter 6 function are described for analysing the frequency behaviour of fractional plants and finding important norms.

❖ In chapter 0 a graphical user interface is described that makes use of the toolbox's functions relieving the user from having to call them from the command prompt.

❖ In chapter 8 a Simulink library is described containing blocks to implement approximations of fractional plants.

❖ Chapter 9 describes functions for handling continued fractions, mathematical entities useful when working with fractional controllers.

❖ Chapter 0 concludes with some final remarks.

This manual no longer has separate chapters for user-oriented and programmer-oriented information. Programmer-oriented material is now found in sections titled

Algorithm that exist for every function of the toolbox. Hence, these may be skipped by users with little interest in programming details.

2. Approximations of fractional order derivatives and integrals

Functions dealt with in this chapter provide integer order frequency domain or discrete-time domain approximations of transfer functions involving fractional (fractional or irrational) powers of *s*.

2.1. Function *nid*

This function is fit for implementing a controller given by the frequency domain transfer function

$$C(s) = ks^{\nu}, \quad \nu \in \mathbb{R} \tag{2.1}$$

by providing both continuous and digital approximations thereof.

Five continuous approximations are available:

❖ The (first generation) Crone approximation. Crone is the (French) acronym of *Commande robuste d'ordre non-entier* (robust fractional order control). This approximation uses a recursive distribution of *N* poles and *N* zeros leading to a transfer function as follows:

$$C(s) = k' \prod_{n=1}^{N} \frac{1 + \frac{s}{\omega_{zn}}}{1 + \frac{s}{\omega_{pn}}} \tag{2.2}$$

k' is an adjusted gain so that if *k* is 1 then the gain is 0 dB for a 1 rad/s frequency. Zeros and poles are to be found within a frequency range $[\omega_l ; \omega_h]$ (inside of which the approximation is valid, being somewhat poor near the limits of the interval) and are given, for a positive *v*, by

$$\alpha = \left(\frac{\omega_h}{\omega_l} \right)^{\frac{\nu}{N}} \tag{2.3}$$

$$\eta = \left(\frac{\omega_h}{\omega_l} \right)^{\frac{1-\nu}{N}} \tag{2.4}$$

$$\omega_{z1} = \omega_l \sqrt{\eta} \tag{2.5}$$

$$\omega_{pn} = \omega_{z,n-1} \alpha, \quad n = 1 \ldots N \tag{2.6}$$

$$\omega_{zn} = \omega_{p,n-1} \eta, \quad n = 2 \ldots N \tag{2.7}$$

For negative values of ν the role of zeros and poles is interchanged ($\omega_{p1} = \omega_l \sqrt{\eta}$ and so on). From (2.3) and (2.4) we see that there are N poles and N zeros in $[\omega_l;\omega_h]$, and that the logarithms of frequencies of zeros and poles are equidistant.

❖ The Carlson approximation, that is the application to (2.1) of the method described below, in subsection 2.4, for function *newton*. This is done by letting $g = s$.

❖ The Matsuda approximation, that is the application to (2.1) of the method described below, in subsection 2.5, for function *matsudaCFE*. Gains of transfer function (2.1) are found at logarithmically spaced frequencies and these are the values entered in expressions (2.45) and (2.46).

❖ The high-frequency continued fraction approximation, that consists in a truncated continued fraction expansion of approximation

$$s^\nu \approx \lambda^\nu \left(1+\frac{s}{\lambda}\right)^\nu, \quad \text{if } \omega \gg \lambda \tag{2.8}$$

It may be shown that the continued fraction is

$$C(s)=\lambda^\nu \left[0;\frac{1}{1},\frac{-\nu\frac{s}{\lambda}}{1},\left\{\frac{\frac{i(i+\nu)}{(2i-1)2i}\frac{s}{\lambda}}{1},\frac{\frac{i(i-\nu)}{2i(2i+1)}\frac{s}{\lambda}}{1}\right\}_{i=1}^{N}\right], \quad \text{for } \omega > \lambda \tag{2.9}$$

❖ The low-frequency continued fraction approximation, that consists in a truncated continued fraction expansion of approximation

$$s^\nu \approx \lambda^\nu \left(1+\frac{\lambda}{s}\right)^{-\nu}, \quad \text{if } \omega \ll \lambda \tag{2.10}$$

It may be shown that the continued fraction is

$$C(s)=\lambda^\nu \left[0;\frac{1}{1},\frac{\nu\frac{\lambda}{s}}{1},\left\{\frac{\frac{i(i-\nu)}{(2i-1)2i}\frac{\lambda}{s}}{1},\frac{\frac{i(i+\nu)}{2i(2i+1)}\frac{\lambda}{s}}{1}\right\}_{i=1}^{N}\right], \quad \text{for } \omega < \lambda \tag{2.11}$$

Thirty-two digital approximations are available, resulting from choosing a formula (out of eight) and a method of expansion (out of four). Available expansion methods are:

❖ truncated MacLaurin series expansions;
❖ truncated time developments;
❖ truncated continuous fraction expansions;
❖ inverted and truncated MacLaurin series expansions;
❖ inverted and truncated time developments;
❖ inverted and truncated continuous fraction expansions.

The eight possible formulas are:

- first order backwards finite difference: $s \approx \frac{1-z^{-1}}{T}$;
- second order backwards finite difference: $s \approx \frac{3-4z^{-1}+z^{-2}}{2T}$;
- third order backwards finite difference: $s \approx \frac{11-18z^{-1}+9z^{-2}-2z^{-3}}{6T}$;
- Tustin formula: $s \approx \frac{2}{T}\frac{1-z^{-1}}{1+z^{-1}}$;
- Simpson formula: $s \approx \frac{3}{T}\frac{1-z^{-2}}{1+4z^{-1}+z^{-2}}$;
- delta transform formula;
- impulse response formula;
- time response formula.

Continued fraction expansions (inverted or not) may be applied to all these formulas; MacLaurin series expansions (inverted or not) may be applied to first six only; the last two may be expanded into time developments (inverted or not). Since formulas that can be the object of a MacLaurin series expansion cannot be the object of a time development and vice-versa, these two expansion methods are, for simplicity, returned by this function for a single value of variable *expansion*, as seen below.

In the following expressions, *T* is the sampling time.

First order backwards finite difference formula, raised to a fractional exponent *v*, expanded into a truncated MacLaurin series, gives

$$C\left(z^{-1}\right)=\frac{1}{T_s^{\nu}}\sum_{k=0}^{n}(-1)^k\binom{\nu}{k}z^{-k} \tag{2.12}$$

Second order backwards finite difference formula, raised to a fractional exponent *v*, expanded into a truncated MacLaurin series, gives

$$\begin{aligned}C\left(z^{-1}\right)=&\frac{\left[\Gamma(\nu+1)\right]^2}{(2T)^{\nu}}\times\\ &\times\sum_{k=0}^{N}z^{-k}\sum_{j=0}^{k}\frac{3^{\nu-j}(-1)^k}{\Gamma(j+1)\Gamma(\nu-j+1)\Gamma(k-j+1)\Gamma(\nu-k+j+1)}\end{aligned} \tag{2.13}$$

Third order backwards finite difference formula, raised to a fractional exponent *v*, expanded into a truncated MacLaurin series, gives

$$C\left(z^{-1}\right)=\frac{\left[\Gamma\left(\nu+1\right)\right]^{3}}{\left(3T\right)^{\nu}}\sum_{k=0}^{N}z^{-k}\left[\sum_{t=0}^{k}\sum_{p=0}^{t}\frac{\left(-1\right)^{p}}{\Gamma\left(p+1\right)\Gamma\left(\nu-p+1\right)}\right.$$
$$\left.\frac{\left(-\frac{7}{4}-\frac{\sqrt{39}}{4}j\right)^{\nu-t+p}\left(-\frac{7}{4}+\frac{\sqrt{39}}{4}j\right)^{\nu-k+t}}{\Gamma\left(t-p+1\right)\Gamma\left(\nu-t+p+1\right)\Gamma\left(k-t+1\right)\Gamma\left(\nu-k+t+1\right)}\right] \tag{2.14}$$

(In the above, *j* is the imaginary unit; in the end, all coefficients turn up to be real.)

Tustin formula, raised to a fractional exponent *v*, expanded into a truncated MacLaurin series, gives

$$C\left(z^{-1}\right)=\left(\frac{2}{T}\right)^{\nu}\Gamma\left(\nu+1\right)\Gamma\left(-\nu+1\right)\times$$
$$\times\sum_{k=0}^{N}z^{-k}\left[\sum_{j=0}^{k}\frac{\left(-1\right)^{j}}{\Gamma\left(\nu-j+1\right)\Gamma\left(j+1\right)\Gamma\left(k-j+1\right)\Gamma\left(-\nu+j-k+1\right)}\right] \tag{2.15}$$

Simpson formula, raised to a fractional exponent *v*, expanded into a truncated MacLaurin series, gives

$$C\left(s\right)=\left(\frac{3}{T}\right)^{\nu}\Gamma\left(\nu+1\right)\left[\Gamma\left(-\nu+1\right)\right]^{2}\sum_{q=0}^{N}z^{-q}\left[\sum_{n=0}^{\mathrm{C}\left(\frac{q}{2}\right)}\sum_{p=2n}^{q}\frac{\left(-1\right)^{n}}{\Gamma\left(n+1\right)\Gamma\left(\nu-n+1\right)}\right.$$
$$\left.\frac{\left(2-\sqrt{3}\right)^{-\nu-p+2n}\left(2+\sqrt{3}\right)^{-\nu-q+p}}{\Gamma\left(p-2n+1\right)\Gamma\left(-\nu-p+2n+1\right)\Gamma\left(q-p+1\right)\Gamma\left(-\nu-q+p+1\right)}\right] \tag{2.16}$$

Delta transform formula, raised to a fractional exponent *v*, expanded into a truncated MacLaurin series, gives

$$C\left(z^{-1}\right)=\sum_{i=0}^{N}{}_{-\nu}g_{i}z^{-i} \tag{2.17}$$

where

$$\sum_{i=0}^{N}{}_{\nu}g_{i}z^{i}=T^{-\nu-1}\left\{\left(\log 2\right)^{\nu}+\ldots\right.$$
$$\left.\ldots+\sum_{i=1}^{N}\left[\left(z-2\right)^{i}\sum_{j=1}^{i}\left(-1\right)^{j+1}\left(\log 2\right)^{\nu-\left(i-j+1\right)}\frac{\Gamma\left(\nu+1\right)}{\Gamma\left(\nu-i+j\right)\Gamma\left(i+1\right)2^{i}}D_{ij}\right]\right\} \tag{2.18}$$

A truncated time development based on impulse response gives

$$C\left(z^{-1}\right)=\frac{T_s^{-\nu}}{\Gamma(-\nu+1)}-\frac{T_s^{-\nu-1}}{\Gamma(-\nu)}+\sum_{i=1}^{N}\frac{\left(iT_s\right)^{-\nu-1}}{\Gamma(-\nu)}z^{-i} \tag{2.19}$$

A truncated time development based on step response gives

$$C\left(z^{-1}\right)=\sum_{i=0}^{N}a_i z^{-i} \tag{2.20}$$

where

$$\begin{aligned} a_0 &= \frac{T_s^{-\nu}}{\Gamma(-\nu+1)}-\frac{T_s^{-\nu-1}}{\Gamma(-\nu)} \\ a_k &= -\sum_{i=0}^{k-1}a_i+\frac{\left(kT_s\right)^{-\nu}}{\Gamma(-\nu+1)},\quad k=1,2,\ldots,n \end{aligned} \tag{2.21}$$

Continued fraction expansions of all eight formulas above may be reckoned as follows. Let $c_{10}+c_{11}z^{-1}+c_{12}z^{-2}+c_{13}z^{-3}+\ldots$ be the formula to expand. Then

$$\begin{aligned} &c_{00}=1;\, c_{0i}=0,\quad \forall_{i\in\mathbb{N}} \\ &c_{j,i}=c_{j-1,0}c_{j-2,i+1}-c_{j-2,0}c_{j-1,i+1}=-\begin{vmatrix} c_{j-2,0} & c_{j-2,i+1} \\ c_{j-1,0} & c_{j-1,i+1} \end{vmatrix} \end{aligned} \tag{2.22}$$

$$C\left(z^{-1}\right)=\left[0;\frac{c_{10}}{c_{00}};\frac{c_{j+1,0}z^{-1}}{c_{j,0}}\right]_{j=1}^{N} \tag{2.23}$$

If an inverted formula is wanted for a controller of fractional order ν, then one of the possibilities above is used for obtaining an approximation for an order $-\nu$, which we will call $\hat{C}\left(z^{-1}\right)$; and finally

$$C\left(z^{-1}\right)=\frac{1}{\hat{C}\left(z^{-1}\right)} \tag{2.24}$$

is returned.

Syntax

```
C = nid(k, v, bandwidth, n, formula, expansion, decomposition)
```

Arguments

❖ *k* — gain, as seen in (2.1).
❖ *v* — fractional order ν, as seen in (2.1).
❖ *bandwidth* — a bandwidth specification, depending on variable *formula* as follows:

♦ ‘crone’ and ‘matsuda’ — *bandwidth* must be a two number vector, containing the limits, given in rad/s, of $\left[\omega_l;\omega_h\right]$, the frequency range where the approximation is valid.

♦ ‘carlson’ — *bandwidth* can be the frequency, given in rad/s, around which the approximation will be valid. It can also be a two number vector, containing the limits, given in rad/s, of a frequency range. In that case the approximation will be valid around the geometric mean of the interval's limits. Depending on the number of poles and zeros, the approximation may cover the whole frequency range, cover a larger range, or cover only its central region.

♦ ‘cfehigh’ — *bandwidth* is λ in (2.9). (If *bandwidth* is a vector, the first value is chosen.)

♦ ‘cfelow’ — *bandwidth* is λ in (2.11). (If *bandwidth* is a vector, the last value is chosen.)

♦ ‘1ofd’, ‘2ofd’, ‘3ofd’, ‘tustin’, ‘simpson’, ‘delta’, ‘impulse’ and ‘step’ — *bandwidth* is the sampling time, in seconds.

❖ *n* — specification about the number of zeros and poles of the approximation. If *decomposition* is ‘all’, what *n* means depends on variable *formula* as follows:

♦ ‘crone’ — the approximation has *n* zeros and *n* poles;

♦ ‘carlson’ — the approximation has at least *n* zeros and *n* poles, but may have more;

♦ ‘matsuda’ — the number of zeros plus the number of poles is *n*; if *n* is odd, the resulting transfer function will be improper (zeros being more than poles by a difference of 1);

♦ ‘cfehigh’ — the approximation has $n - 1$ zeros and *n* poles;

♦ ‘cfelow’ — the approximation has *n* zeros and *n* poles;

♦ ‘1ofd’, ‘2ofd’, ‘3ofd’, ‘tustin’, ‘simpson’, ‘delta’, ‘impulse’ and ‘step’ — *n* is the number of delays in the transfer function; if there are poles and zeros, it is the sum of the number of delays in the numerator and denominator (Tustin formula expanded into a continued fraction—inverted or not—does not accept odd values. If one is given, the even number immediately above is assumed).

If *decomposition* is ‘frac’, *n* affects the fractional part only. When the integer part is added, the number of poles or zeros will be increased accordingly.

❖ *formula* — the approximating formula used. It should be one of the following strings:

♦ ‘crone’ — (continuous) Crone approximation;

♦ ‘carlson’ — (continuous) Carlson approximation;

♦ ‘matsuda’ — (continuous) Matsuda approximation;

♦ ‘cfehigh’ — (continuous) high-frequency continued fraction approximation;

♦ ‘cfelow’ — (continuous) low-frequency continued fraction approximation;

♦ ‘1ofd’ — (digital) first order backward finite differences approximation;

♦ ’2ofd’ — (digital) second order backward finite differences approximation;

♦ ‘3ofd’ — (digital) third order backward finite differences approximation;

♦ ‘tustin’ — (digital) Tustin continuous-discrete transform;

♦ ‘simpson’ — (digital) Simpson continuous-discrete transform;

♦ ‘delta’ — (digital) Delta transform based approximation;

♦ ‘impulse’ — (digital) approximation based on the impulse response;

♦ ‘step’ — (digital) approximation based on the step response.

❖ *expansion* (optional) — method used, when a digital approximation is built, for expanding a power of the formula chosen with variable *formula*. It should be one of the following strings:

- ♦ 'mcltime' (default) — truncated MacLaurin expansion or truncated time development;
- ♦ 'cfe' — continued fraction expansion;
- ♦ 'mcltimeINV' — truncated MacLaurin expansion or truncated time development, reckoned for a symmetrical value of *v*, and inverted;
- ♦ 'cfeINV' — continued fraction expansion, reckoned for a symmetrical value of *v*, and inverted.

This parameter is neglected when *formula* is 'crone', 'carlson', 'matsuda', 'cfehigh' or 'cfelow'.

❖ *decomposition* (optional) — determines how the approximation is performed. It may have two values:

- ♦ 'frac' (default) — Parameter *v* is split into an integer and a fractional part. The integer part is the largest integer smaller than *v*. The fractional part is approximated using the method chosen with *formula*. The integer part is approximated, depending on variable *formula*, as follows:
  - 'crone', 'carlson' and 'matsuda' — no approximation is needed: zeros and poles are added as necessary;
  - '1ofd', '2ofd', '3ofd', 'tustin' and 'simpson' — each zero or pole is approximated with the chosen formula;
  - 'delta', 'impulse' and 'step' — each zero and pole is approximated with Tustin formula.

  The fractional and the integer parts are then joined together.
- ♦ 'all' — An approximation corresponding to order *v* is found. This option is often inferior to the former. Also, it cannot be used together with Carlson method when *v* is larger than 1 or less than –1.

Return values

❖ *C* — tf object given according to the applicable formulas referred to above.

Error messages

❖ *The method has an invalid format.* — See above the available options.
❖ *The expansion has an invalid format.* — See above the available options.
❖ *The decomposition has an invalid format.* — See above the available options.
❖ *Variable bandwidth should consist of two frequencies.* — This appears when *bandwith* must consist of two frequencies but contains only one number.
❖ *Carlson method with |v|>1 demands that decomposition be 'frac'.*

Warnings

❖ *Order v was rounded to* (followed by a numerical value) — This may happen when Carlson approximation is used, since not all values of *v* are possible, as explained below, in subsection 2.2, for function *newton*.

Limitations

With digital approximations and with continuous continued fraction approximations, there is no way to specify the range of frequencies in which the approximation will be valid.

Some formulas only perform acceptably for positive or negative values of *v*, and some only perform acceptably in one of the time or frequency domains. Literature should be checked to choose in advance the formulas that are expected to perform according to what is desired given the particular requirements of the problem under consideration.

Algorithm

The first thing this function does is providing default values for input variables that may not exist (this is checked by means of *nargin*). According to the value of *formula*, *bandwidth* may be required to have two elements, or *decomposition* may be required to be 'frac'. That is checked by means of *if* structures. If *bandwidth* consists of two values, these will be the limits of a frequency range, and are stored in two variables called *wl* and *wh*. If a digital approximation is to be found, *bandwidth* is renamed *Ts* to stress that it is a sampling time.

By means of an *if* structure the two possible cases of variable *expansion* are handled. If it is 'frac', the approximation of the fractional part is reckoned by calling *nid* again, and stored in variable *C*. Then, according to *formula*, zeros or poles are added. If *expansion* is 'all', one of six functions is called to handle the approximation. This is to simplify the code and render it easier to read. Five of these functions are internal to *nid* and are described below. Function *crone1*, used for the Crone approximation, is described in section 2.3. Function *newton*, used for the Carlson approximation, is described in section 2.4, and is called here with function *g* set to a zero.

Function *matsuda*

This function handles the Matsuda approximation case. Frequencies are obtained in the range from *wl* to *wh* by means of function *logspace*, that provides a logarithmically spaced set. The number of frequencies is equal to *n* plus 1, since a single frequency results in no zeros and poles, and each additional frequency adds a zero or a pole (a zero if the number of zeros and poles is equal, a pole if there are more zeros than poles). Gains of (2.1) at frequencies *w* are found and fed to function *matsudaCFE*, described below in section 2.5, that takes care of the rest.

Function *cfehigh*

This function handles the high-frequency continued fraction approximation. It builds vectors *a* and *b* with the numerators and the denominators of continued fraction (2.9) and then assembles it using function *contfraceval*, described below in section 0.

Function *cfelow*

This function handles the low-frequency continued fraction approximation. It is similar to *cfehigh*. Since several unnecessary zeros and poles in the origin appear, they are done away with by converting the transfer function into a state-space representation and then back into a transfer function again.

Function *digital*

This function handles all digital approximations, since they all have common characteristics.

In this function *switch* structures are used to select the appropriate expression to apply. The outermost one copes with the four possibilities of parameter *expansion*. Corresponding *case* tags are marked with whole lines of %, so as to render them more visible.

If variable *expansion* is equal to 'mcltime', another *switch* structure is used to cope with the several possible values of variable *method*. Most methods deserve little comment: coefficients are reckoned with one or two *for* cycles and stored into vector *dc*, according to the formulas given above. Method ‘delta’ requires an auxiliary matrix, and *dc* is first of all reckoned as a vector with coefficients for a δ-domain discrete transfer function, corresponding to a differentiation order of $-\nu$. A *for* cycle is then required for converting each monomial into a z-domain transfer function. The sum of all these is stored into variable *temp*, and then the order of coefficients is inverted so as to cancel the effect of the changed signal of ν.

Last of all the transfer function C is obtained from vector *dc* with function *tf*, property *Variable* being set to *z^-1*.

If a continued fraction expansion is chosen, the following formulas, reckoned using function *contfraceval*, are used for methods ‘tustin’ and ‘1ofd’:

$$\left(\frac{1-z^{-1}}{1+z^{-1}}\right)^{\nu} = \left[1; \frac{2\nu}{-\frac{1}{z^{-1}}-\nu}, \left\{\frac{\nu^2-i^2}{-\frac{2i+1}{z^{-1}}}\right\}\right]_{i=1}^{+\infty} \tag{2.25}$$

$$\left(1-z^{-1}\right)^{\nu} = \left[0; \frac{1}{1}, \frac{\nu z^{-1}}{1}, \left\{\frac{-\frac{i(i+\nu)}{(2i-1)2i}z^{-1}}{1}, \frac{-\frac{i(i-\nu)}{2i(2i+1)}z^{-1}}{1}\right\}\right]_{i=1}^{+\infty} \tag{2.26}$$

These formulas provide the same results that would be obtained using (2.22) and (2.23). For all other methods, a MacLaurin series expansion is performed first. The resulting polynomial is expanded into a continued fraction with function *contfracf*. Then the fraction is expanded with *contfraceval*.

All continued fractions are expanded up to n+1 terms, so that the sum of the number of delays in the numerator and denominator will be n. The sole exception is Tustin formula. Formula (2.25) adds one pole and one zero per iteration, and so the expansion is carried out to the smaller integer right above n/2 (above and not below, so that there will not be too few delays). That is why no odd values of n are possible.

If an inversion is asked for, the function calls itself for a symmetrical value of ν. Then the result is inverted.

References

On the Crone approximation, see (Outaloup, p. 154-174, 1991).

On the Carlson approximation, see (Carlson *et al.*, 1964), (Vinagre, p. 153, 2001).

On the Matsuda approximation, see (Matsuda *et al.*, 1993), (Vinagre, p. 153-154, 2001).

On continuous continued fraction approximations, see (Vinagre, p. 153, 2001).

On digital approximations, see (Machado, 1999, 2001), (Lubich, 1986), (Vinagre *et al.*, 2000), (Valério *et al.*, 2002, 2005a).

A revision of all this material is found in (Valério, p. 59-85, 2005c).

2.2. Function *nipid*

This function is fit for implementing a controller given by the frequency domain transfer function

$$C(s) = k_P + k_D s^{\nu_D} + \frac{k_I}{s^{\nu_I}}, \quad \nu_D, \nu_I \in \mathbb{R}^+ \tag{2.27}$$

by providing both continuous and digital approximations thereof. Controllers with this structure are known as fractional PIDs. In this function, however, no conditions are imposed on the orders, that may be both positive or both negative (even though strictly speaking *C* will be a fractional PID only if one order is negative and the other positive as seen in (2.27)).

The available approximations are those listed in the previous subsection for function *nid*. But function *newton*, described below in section 2.4, may also be used for particular cases of fractional PIDs.

Syntax

```
function C = nipid(kp, kd, vd, ki, vi, bandwidth, n, formula,...
    expansion, decomposition)
```

Arguments

❖ *kp* — proportional gain, as seen in (2.27).
❖ *kd* — derivative gain, as seen in (2.27).
❖ *vd* — fractional derivative order, as seen in (2.27).
❖ *ki* — integral gain, as seen in (2.27).
❖ *vi* — fractional integral order, as seen in (2.27).
❖ *bandwidth* — a bandwidth specification, depending on variable *formula* as follows:

♦ ‘crone’ and ‘matsuda’ — *bandwidth* must be a two number vector, containing the limits, given in rad/s, of $[\omega_l ; \omega_h]$, the frequency range where the approximation is valid.

♦ ‘carlson’ — *bandwidth* can be the frequency, given in rad/s, around which the approximation will be valid. It can also be a two number vector, containing the limits, given in rad/s, of a frequency range. In that case the approximation will be valid around the geometric mean of the interval’s limits. Depending on the number of poles and zeros, the approximation may cover the whole frequency range, cover a larger range, or cover only its central region.

♦ ‘cfehigh’ — *bandwidth* is λ in (2.9). (If *bandwidth* is a vector, the first value is chosen.)

♦ ‘cfelow’ — *bandwidth* is λ in (2.11). (If *bandwidth* is a vector, the last value is chosen.)

♦ ‘1ofd’, ‘2ofd’, ‘3ofd’, ‘tustin’, ‘simpson’, ‘delta’, ‘impulse’ and ‘step’ — *bandwidth* is the sampling time, in seconds.

❖ *n* — specification about the number of zeros and poles of the approximation. If *decomposition* is ‘all’, what *n* means depends on variable *formula* as follows:

♦ ‘crone’ — the approximation has *n* zeros and *n* poles;

♦ ‘carlson’ — the approximation has at least *n* zeros and *n* poles, but may have more;

♦ ‘matsuda’ — the number of zeros plus the number of poles is *n*; if *n* is odd, the resulting transfer function will be improper (zeros being more than poles by a difference of 1);

♦ ‘cfehigh’ — the approximation has *n* – 1 zeros and *n* poles;

♦ ‘cfelow’ — the approximation has *n* zeros and *n* poles;

♦ ‘1ofd’, ‘2ofd’, ‘3ofd’, ‘tustin’, ‘simpson’, ‘delta’, ‘impulse’ and ‘step’ — *n* is the number of delays in the transfer function; if there are poles and zeros, it is the sum of the number of delays in the numerator and denominator (Tustin formula expanded into a continued fraction—inverted or not—does not accept odd values. If one is given, the even number immediately above is assumed).

If *decomposition* is ‘frac’, *n* affects the fractional part only. When the integer part is added, the number of poles or zeros will be increased accordingly.

❖ *formula* — the approximating formula used. It should be one of the following strings:

♦ ‘crone’ — (continuous) Crone approximation;
♦ ‘carlson’ — (continuous) Carlson approximation;
♦ ‘matsuda’ — (continuous) Matsuda approximation;
♦ ‘cfehigh’ — (continuous) high-frequency continued fraction approximation;
♦ ‘cfelow’ — (continuous) low-frequency continued fraction approximation;
♦ ‘1ofd’ — (digital) first order backward finite differences approximation;
♦ ‘2ofd’ — (digital) second order backward finite differences approximation;
♦ ‘3ofd’ — (digital) third order backward finite differences approximation;
♦ ‘tustin’ — (digital) Tustin continuous-discrete transform;
♦ ‘simpson’ — (digital) Simpson continuous-discrete transform;
♦ ‘delta’ — (digital) Delta transform based approximation;
♦ ‘impulse’ — (digital) approximation based on the impulse response;
♦ ‘step’ — (digital) approximation based on the step response.

❖ *expansion* (optional) — method used, when a digital approximation is built, for expanding a power of the formula chosen with variable *formula*. It should be one of the following strings:

♦ ‘mcltime’ (default) — truncated MacLaurin expansion or truncated time development;

♦ ‘cfe’ — continued fraction expansion;

♦ ‘mcltimeINV’ — truncated MacLaurin expansion or truncated time development, reckoned for a symmetrical value of *v*, and inverted;

♦ ‘cfeINV’ — continued fraction expansion, reckoned for a symmetrical value of *v*, and inverted.

This parameter is neglected when *formula* is ‘crone’, ‘carlson’, ‘matsuda’, ‘cfehigh’ or ‘cfelow’.

❖ *decomposition* (optional) — determines how the approximation is performed. It may have two values:

♦ ‘frac’ (default) — Parameter *v* is split into an integer and a fractional part. The integer part is the largest integer smaller than *v*. The fractional part is approximated using the method chosen with *formula*. The integer part is approximated, depending on variable *formula*, as follows:

- ‘crone’, ‘carlson’ and ‘matsuda’ — no approximation is needed: zeros and poles are added as necessary;

- ‘1ofd’, ‘2ofd’, ‘3ofd’, ‘tustin’ and ‘simpson’ — each zero or pole is approximated with the chosen formula;
- ‘delta’, ‘impulse’ and ‘step’ — each zero and pole is approximated with Tustin formula.

The fractional and the integer parts are then joined together.

♦ ‘all’ — An approximation corresponding to order *v* is found. This option is often inferior to the former. Also, it cannot be used together with Carlson method when *v* is larger than 1 or less than –1.

Return values

❖ *C* — tf object given according to the applicable formulas referred to above.

Error messages and warnings

Errors and warnings are those described above in subsection 2.1 for function *nid*.

Limitations

Comments above in subsection 2.1 about digital approximations are valid for this function also.

Algorithm

The first thing this function does is providing for inexistent input variables, just like function *nid* does. Then a *switch* structure handles different possible values of variable *formula*.

When *formula* is ‘crone’, ‘carlson’, ‘cfehigh’ or ‘cfelow’, function *nid* is called twice to handle the derivative and the integral parts, which are then summed up with the proportional part. These calls have parameter *n* set to *ceil(n/2)* so that, when adding the integral part to the derivative part, no more than *n* (*n*+1 if *n* is odd) zeros or poles appear (if *decomposition* is set to ‘all’—check the warnings above).

Case ‘matsuda’, after an initial check on variable *bandwidth*, is handled with function *newton* just as *nid* does. The only difference lies in the expression that reckons the gain to approximate.

All other (digital) cases are handled together, with another *switch* structure, required to deal with the several possible cases of variable *expansion*, since the number of terms that must be used for approximating the derivative and integral parts is not always the same.

References

On this subject, see (Podlubny, p. 249-250, 2001), (Valério, p. 121-127, 2005c).

2.3. Function *crone1*

This function is fit for implementing a controller given by the frequency domain transfer function

$$C(s) = ks^{v}, \quad v \in \mathbb{C} \tag{2.28}$$

by providing a continuous Crone approximation thereof. The form the approximation takes for real differentiation orders was given above in section 2.1, equation (2.2). For complex differentiation orders $v = a + b\sqrt{-1}$, a recursive distribution of poles and zeros is used:

$$C(s) = k' \frac{\prod_{n=1}^{M} 1 + \frac{s}{\omega_{zn}}}{\prod_{n=1}^{N} 1 + \frac{s}{\omega_{pn}}} \tag{2.29}$$

Notice that the number of zeros and poles is no longer the same. *k*' is an adjusted gain so that if *k* is 1 then the gain is 0 dB for a 1 rad/s frequency. Zeros and poles are to be found from frequency ω_l on and are given by

$$\alpha = \left(\frac{\omega_h}{\omega_l}\right)^{a\frac{N+M}{NM}} \tag{2.30}$$

$$\eta = 10^{\frac{\pi \log_{10}(\alpha)}{\pi - 2b\log(10)\log_{10}(\alpha)\operatorname{tgh}\frac{b\pi}{2}}} \tag{2.31}$$

$$\omega_{z1} = \omega_l \sqrt{\eta} \tag{2.32}$$

$$\omega_{p1} = \omega_l \sqrt{\alpha} \tag{2.33}$$

$$\omega_{pn} = \omega_{p,n-1}\alpha, \quad n = 2 \ldots N \tag{2.34}$$

$$\omega_{zn} = \omega_{z,n-1}\eta, \quad n = 2 \ldots M \tag{2.35}$$

Syntax

```
C = crone1(k, v, wl, wh, n)
```

Arguments

❖ *k* — gain, as seen in (2.28).
❖ *v* — non-integer order, as seen in (2.28).
❖ *wl* — lower limit, given in rad/s, of $[\omega_l;\omega_h]$, the frequency range where the approximation is valid.
❖ *wh* — upper limit, given in rad/s, of $[\omega_l;\omega_h]$, the frequency range where the approximation is valid. Notice that, according to formulas (2.32) to (2.35), this limit is not taken into account for complex differentiation orders.
❖ *n* — order of the approximation. If *v* is real, the number of zeros and poles is the same; so only the first element of *n* is taken into account. If *v* is complex, *n* is to contain two values: these are $[N \quad M]$ as seen in equation (2.29).

Return values

❖ *C* — tf object given according to the applicable formulas referred to above.

Error messages

❖ *Complex differentiation orders require a different number of zeros and poles.* — If argument *v* is complex but *n* contains only one value, or contains two equal values, no approximation may be built.

Algorithm

If *v* is real, the value of product $\alpha\eta$ is reckoned and stored in variable *alphaXeta*. From (2.3) and (2.4) it is given by

$$\alpha\eta = \left(\frac{\omega_h}{\omega_l}\right)^{\frac{1}{N}} \tag{2.36}$$

Values for *alpha* and *eta* are reckoned from this variable.

In what follows vectors with the frequencies of poles and zeros (called *poles* and *zeros*) are obtained using formulas (2.3) to (2.7). If *v* is negative the roles of zeros and poles are interchanged.

A tf object is built using these frequencies. Minus signs are needed since zeros and poles are stable. This tf object, called *C*, does not have a unit gain at frequency 1 rad/s. For that reason it is divided by *bode(C, 1)*. Function *bode* returns gains and phases, but the first value it returns it that of gains. Frequency 1 is specified and so this ensures the desired gain. Multiplication by *k* ends the function.

If *v* is complex, *n* is split into *n* and *m*, and *v* split into *a* and *b*, to simplify the following commands. Variables *alpha* and *eta* are found and vectors with zeros and poles built, much like the real case.

References

On this subject, see (Oustaloup, p. 154-174 and 336-353, 1991), (Valério, p. 59-61 and 120-121, 2005c).

2.4. Function *newton*

This function provides an approximation of a fractional order power of a transfer function using Newton's iterative method. The power must be the inverse of an integer number, since the equation solved is

$$C^a(s) = g(s) \tag{2.37}$$

In the end we will have an approximation of

$$C(s) = g^{\frac{1}{a}}(s) \tag{2.38}$$

The recursive expression

$$C_n(s) = C_{n-1}(s)\frac{(a-1)C_{n-1}^a(s)+(a+1)g(s)}{(a+1)C_{n-1}^a(s)+(a-1)g(s)} \tag{2.39}$$

is used, C_0 being set to 1.

This function uses the method outlined above for approximating

$$C(s) = kg^\nu(s) \tag{2.40}$$

When ν is larger than 1 or less than –1, it is split into an integer part and a fractional part. The integer part is the largest integer smaller than ν. The fractional part is approximated using Newton's method. The integer part is found without any approximation and joined together with the fractional one.

When ν is between –1 and 1, no such splitting is needed. But in both cases the fraction has to be rounded off to the nearest inverse of an integer.

Syntax

```
C = newton (k, v, bandwidth, n, g, stop)
```

Arguments

❖ *k* — gain, as seen in (2.40).

❖ *v* — fractional order, as seen in (2.40). This may have to be rounded off as explained above.

❖ *bandwidth* — a bandwidth specification. It can be the frequency, given in rad/s, around which the approximation will be valid. It can also be a two number vector, containing the limits, given in rad/s, of a frequency range. In that case the approximation will be valid around the geometric mean of the interval's limits. Depending on the number of poles and zeros, the approximation may cover the whole frequency range, cover a larger range, or cover only its central region.

❖ *n* — number of iterations (given by (2.39)) performed, or number of zeros and poles of the approximation, depending on the value of *stop*.

❖ *g* — function raised to a fractional power, as seen in (2.40).

❖ *stop* (optional) — criterion for stopping the iterations. It should be one of the following strings:

- ♦ 'iter' (default) — *n* iterations are performed;
- ♦ 'zp' — iterations are performed until the approximation has at least *n* zeros and *n* poles (but it may have more).

Return values

❖ *C* — tf object given according to the applicable formulas referred to above.

Limitations

Two things should be noticed. Firstly, that the number of zeros and poles grows very quickly with the number of iterations. When these are performed until *n* is reached, this usually means that *n* was largely exceeded.

Secondly, that the expressions (2.37) to (2.40) give no choice as to the frequency range where the approximation will be valid. So in the end poles and zeros must be conveniently augmented or diminished according to what is desired. This requires correcting the gain accordingly. A consequence is that if *bandwidth* is set so that the range of frequencies where the approximation is good does not include 1 rad/s there will be an error in the gain. This is because this function sets the gain to its proper value at 1 rad/s, which will be a cause of error if the gain is no longer properly approximated at that frequency.

How to use function *newton* with fractional PIDs

Function *newton* may be used to implement a fractional PID in one of two cases:

❖ the fractional derivative order is the inverse of an integer and the fractional integral order is a multiple of the fractional derivative order;

❖ the fractional integral order is the inverse of an integer and the fractional derivative order is a multiple of the fractional integer order.

In the first case, (2.27) becomes

$$\begin{aligned} C(s) &= k_P + k_D s^{1/a_D} + k_I s^{-n/a_D} = k_P + \left(k_D^{a_D} s\right)^{1/a_D} + \left(k_I^{a_D}\frac{1}{s^n}\right)^{1/a_D} = \\ &= k_P + \left(\frac{k_D^{a_D} s^{1+n} + k_I^{a_D}}{s^n}\right)^{1/a_D} \end{aligned} \tag{2.41}$$

So function $\frac{k_D^{a_D} s^{1+n} + k_I^{a_D}}{s^n}$ may be given to function *newton*, together with order $1/a_D$. Proportional gain k_P must be added in the end.

In the second case, (2.27) becomes

$$\begin{aligned} C(s) &= k_P + k_D s^{n/a_I} + k_I s^{-1/a_I} = k_P + \left(k_D^{a_I} s^n\right)^{1/a_I} + \left(k_I^{a_I}\frac{1}{s}\right)^{1/a_I} = \\ &= k_P + \left(\frac{k_D^{a_I} s^{1+n} + k_I^{a_I}}{s}\right)^{1/a_I} \end{aligned} \tag{2.42}$$

So function $\frac{k_D^{a_I} s^{1+n} + k_I^{a_I}}{s}$ may be given to function *newton*, together with order $1/a_I$. Proportional gain k_P must be added in the end.

Algorithm

The first thing this function does is providing the default values for input variable *stop*, that may not exist (this is checked by means of *nargin*). The frequency around which the approximation is valid may be given in *bandwidth*, or may have to be reckoned from the two variables therein. Then *v* is decomposed into an integer part *m* and a fractional part, which is the inverse of *a*, rounded so as to be an integer. Values of 1 or –1 would not do, and in those cases *a* is set to 2 or –2. It should be noticed that if *m* is not zero, *a* will always be positive. A warning is given if the decomposition of *v* changed its value. It may happen that this warning appears solely because of slight round-off errors, but this is hard to avoid.

An *if* structure is used to deal with both possible cases of *stop*, which are then handled with a *while* or a *for*. In the implementation of formula (2.39), the only detail worth of notice is that the intermediate results are stored in tf objects. Since this usually introduces several zeros and poles in the origin, a final double conversion to state-space and back to transfer function is included to eliminate them.

At the end of the function zeros, poles and gain are extracted using function *zpkdata* and stored in the variables with the corresponding names. The tf object is rebuilt with poles and zeros multiplied by *wc* and the gain of *g* at 1 rad/s, so that the frequency range where the approximation is valid is recentred. The gain is then corrected with input parameter *k* and the ratio between its correct value at 1 rad/s and the value of the former approximation at the same frequency. That the correction be done at 1 rad/s is the reason why, if at that frequency the approximation is poor, the gain will be incorrect everywhere. At the same time, *m* poles (or zeros if *m* is negative) are added.

It should also be noticed that although a provision is included in this function for a gain *k*, this is strictly not necessary since it could always be made a part of *g*. It was included to ensure similarity with function *nid*.

References

On this subject, see (Carlson *et al.*, 1964), (Vinagre *et al.*, 2000), (Vinagre, p. 153, 2001), (Valério, p. 61, 2005c).

2.5. Function *matsudaCFE*

Matsuda method approximates a function by a continued fraction (the CFE in the name of the function stands for continued fraction expansion). If the value of function f is known in a set of points x_0, x_1, x_2, x_3, …, then the following set of function is recursively defined:

$$\begin{aligned} &d_0(x) = f(x) \\ &d_{k+1}(x) = \frac{x - x_k}{d_k(x) - d_k(x_k)}, \quad k = 0,1,2\ldots \end{aligned} \tag{2.43}$$

Function f is approximated by

$$f(x) = d_0(x_0) + \cfrac{x - x_0}{d_1(x_1) + \cfrac{x - x_1}{d_2(x_2) + \cfrac{x - x_2}{d_3(x_3) + \cfrac{x - x_3}{\cdots}}}} \tag{2.44}$$

This method may be applied for transfer functions using their gain as the value known at several frequencies and replacing variable x by s in (2.44). Thus, for a transfer function C,

$$
\begin{aligned}
&d_0(\omega) = \left|C(j\omega)\right| \\
&d_{k+1}(\omega) = \frac{\omega - \omega_k}{d_k(\omega) - d_k(\omega_k)}, \quad k = 0,1,2\ldots
\end{aligned}
\tag{2.45}
$$

$$
C(\omega) = d_0(\omega_0) + \cfrac{s-\omega_0}{d_1(\omega_1) + \cfrac{s-\omega_1}{d_2(\omega_2) + \cfrac{s-\omega_2}{d_3(\omega_3) + \cfrac{s-\omega_3}{\cdots}}}}
\tag{2.46}
$$

Syntax

```
C = matsudaCFE(w, gain)
```

Arguments

The function receives the following parameters:

❖ *w* — vector (row or column) with a list of frequencies, given in rad/s.

❖ *gain* — vector (row or column) with a list of gains, in dB, at frequencies given in vector *w*.

Return values

❖ *C* — tf object given according to the applicable formulas referred to above.

Algorithm

In this function, and since gains are received in dB, the first thing to do is to convert them into absolute values, which are stored in variable *f*. Since the indexes begin at 1, frequency ω_i is to be found in *w(i+1)* and the corresponding gain is to be found in *f(i+1)*.

Functions *d* are to be evaluated at frequencies *w*, and a matrix is built with such values: different lines correspond to different functions, and different columns to different frequencies. Again, since indexes of columns and lines begin at 1, the value of $d_i(\omega_j)$ is to be found at *d(I+1,j+1)*; hence the +1 operation whenever indexes appear.

But while the value of d_0 is needed for all frequencies, the value of d_1 is only needed for $\omega_1, \omega_2, \omega_3, \ldots$; the value of d_2 is only needed for $\omega_2, \omega_3, \ldots$; and, generally speaking, the value of d_i is only needed for frequencies from ω_i onwards. This means that the matrix may be upper-triangular.

As a consequence, matrix *d* is built while being swept by two different *for* cycles. The first sweeps columns; the innermost sweeps lines down from the top until the main diagonal. Function d_0 is dealt with outside this innermost loop, since it is given by a different expression. Inside the loop lines below the first are reckoned. At the end of the outermost cycle the main diagonal is copied to a variable called *alpha*; in the end, we shall have

$$
\alpha_i = d_i(\omega_i) \tag{2.47}
$$

these being the values needed for the continued fraction.

The fraction is reckoned from its innermost part out, which is the most efficient way since its length is known.

References

On this subject, see (Matsuda *et al.*, 1993), (Vinagre *et al.*, 2000), (Vinagre, p. 153-154, 2001), (Valério, p. 61-63, 2005c).

2.6. Obsolescent functions

In previous versions of this toolbox, there were several functions doing the job now committed to functions *nid* and *nipid*. So that code based upon these no longer necessary functions may still be used, compatibility is ensured by still providing them (in a separate folder called obsolescent).

It is recommended that they should *not* be used. They may be removed in future versions.

In their present forms, they merely call *nid* or *nipid* with the appropriate parameters, as follows:

Function…	… is the same as…
`C = carlson (k, a, wc, n)`	`C = nid(k, v, wc, n, 'carlson', [], 'all')`
`C = matsuda(k, v, wl, wh, n)`	`C = nid(k, v, [wl wh], n, 'matsuda', [],... 'all')`
`C = nidiz(k, v, n, Ts, formula, expansion)`	`C = nid(k, v, Ts, n, formula, expansion,... 'all')`
`C = nipidcrone(kp, kd, vd, ki, vi, wl,... wh, n)`	`C = nipid(kp, kd, vd, ki, vi, [wl, wh],... n, 'crone', [], 'all')`
`C = nipidcarlson(kp, kd, ad, ki, ai, wc, n)`	`C = nipid(kp, kd, 1/ad, ki, 1/ai, wc, n,... 'carlson', [], 'all')`
`C = nipidmatsuda(kp, kd, vd, ki, vi, wl,... wh, n)`	`C = nipid(kp, kd, vd, ki, vi, [wl, wh],... n, 'matsuda', [], 'all')`
`C = nipidz(kp, kd, vd, ki, vi, n, Ts,... formula, expansion)`	`C = nipid(kp, kd, vd, ki, vi, Ts, n,... formula, expansion, 'all')`

In version 2.2 function *crone1* was also considered obsolescent. This is no longer the case.

3. Functions ensuring a real fractional order behaviour for the open-loop

The two functions described in this section reckon controllers that ensure an open-loop dynamic described by (1.1). Function *crone2* works in the frequency domain and *crone2z* works in the discrete-time domain[5].

3.1. Function *crone2*

This function identifies a frequency domain controller that will ensure an open-loop dynamic given by (1.1). The corresponding phase ought to be constant:

$$\arg\left[G\left(j\omega\right)C\left(j\omega\right)\right]=\nu\frac{\pi}{2},\quad \forall\omega \tag{3.1}$$

Suppose you know the frequency behaviour of plant G at N frequencies ω_1, ω_2,... ω_N. Equation (3.1) implies that at those frequencies

$$\arg\left[C\left(j\omega_i\right)\right]=\nu\frac{\pi}{2}-\arg\left[G\left(j\omega_i\right)\right],\quad i=1\ldots N \tag{3.2}$$

Let us call the right hand difference ϕ. Suppose that the controller is given by

$$C\left(s\right)=k\frac{\prod_j 1+\frac{s}{\omega_{zj}}}{\prod_j 1+\frac{s}{\omega_{pj}}} \tag{3.3}$$

where the ω_{zj} are the frequencies of the poles and the ω_{pj} are the frequencies of the poles. If we replace each stable zero by an unstable pole at the same frequency, and each unstable zero by a stable pole at the same frequency, the phase of C will be just the same:

$$C\left(s\right)=k\frac{1}{\prod_{j=1}^{M} 1+\frac{s}{\omega_{zpj}}} \tag{3.4}$$

M is the number of pole or zero frequencies of the controller. Equality (3.2) becomes

[5] Users interested in the methodologies implemented with these functions might want to consider the use of the *Crone toolbox*, (commercially) available from the University of Bordeaux. See for instance (Cois *et al.*, 2002), (Oustaloup *et al.*, 2002), (Melchior *et al.*, 2004).

$$-\sum_{j=1}^{M}\operatorname{arctg}\frac{\omega_i}{\omega_{zpj}}=\phi_i,\quad i=1\ldots N \tag{3.5}$$

This is the system of equations that this function solves.

This reasoning makes clear that the function may also be used as an identification method. In that case ϕ will be the experimental phase data to fit. Once frequencies ω_{zpj} are found, it must be checked which correspond to zeros and which correspond to poles. All combinations must actually be tried, and the one with a gain closer to the experimental gain data will be chosen.

When the function is used for providing a controller, open-loop gain behaviour is not as important as controller stability. So all frequencies are made to correspond to stable poles and zeros (stable poles remain as they are, unstable poles are turned into stable poles).

Syntax (identification method)

```
C = crone2(w, gain, phase, n)
```

Arguments (identification method)

❖ *w* — vector with frequencies ω_1, ω_2,… ω_N in rad/s.

❖ *gain* — column vector with experimental gains (at frequencies found in *w*) in dB.

❖ *phase* — vector with experimental phases (at frequencies found in *w*) in degrees.

❖ *n* (optional) — desired number of poles and zeros of the model (corresponds to *M* in (3.4) and (3.5)). The actual number of poles and zeros may be less than *n* since spurious frequencies may appear during the resolution and be eliminated later. If this parameter is omitted, *n* is assumed to be *N* (the number of frequencies in vector *w*), for system (3.5) to become square (and easier to solve).

Syntax (finding a controller)

```
C = crone2(w, [], phase, n, w1)
```

Arguments (finding a controller)

❖ *w* — vector with frequencies ω_1, ω_2,… ω_N in rad/s.

❖ *phase* — vector with the phases, in degrees, the controller must have at frequencies found in *w* (ϕ as given by the right-hand side of equation (3.2)).

❖ *n* (optional) — desired number of poles and zeros of the controller (corresponds to *M* in (3.4) and (3.5)). The actual number of poles and zeros may be less than *n* since spurious frequencies may appear during the resolution and be eliminated later. If this parameter is omitted, *n* is assumed to be *N* (the number of frequencies in vector *w*), for system (3.5) to become square (and easier to solve).

❖ *w1* — frequency, in rad/s, at which the controller will have a unit gain (that is to say, a 0 dB gain).

Return values

❖ *C* — tf object given according to the applicable formulas referred to above.

Algorithm

The function begins by completing the parameters received. Since the code necessary for solving system (3.5) is very complicated, it was relegated to a function later in the file, called *resolution* and described below. The rest of the function filters (thrice) and interprets the results returned.

The first filter eliminates all real poles with a frequency clearly above the interest frequency range, since they will not affect the controller's phase; and all complex poles outside the interest frequency range, since pairs of complex conjugate poles, when found outside the interest frequency range, are certainly needless. Complex conjugate poles cause phase jumps of 180º, and it is possible that they shift the phase up or down 180º in order to reach the correct value; but the phase's average value will be corrected later, so eliminating all complex poles outside the interest frequency range does no harm. The code sweeps vector *wzp*, containing frequencies returned by *resolution*, with a *for* cycle, and stores values to keep in a temporary variable called *temp*. It is not possible to eliminate undesired values immediately, as that would interfere with the *for* cycle, that would try to reach positions at the end of the vector that would no longer exist. A limit two decades above the highest sampling frequency was chosen for keeping real poles.

After this all pairs of poles that eliminate each other's effect on phase are to be done away with in the second filter. Poles have effects on phase that compensate each other if their module is equal (so that they affect the phase at the same frequency) and they are symmetric in relation to the imaginary axis (which means that the sum of their arguments is π rad). As it is highly unlikely that exact equalities are found a tolerance was allowed for. It has the value of 1º, which appeared to be reasonable. Immediately eliminating elements from the vector would interfere with the two nested *for* cycles, just like before; so a temporary vector of flags called *flags* was used to store the number of those that are to be eliminated. When cycles are over only those solutions having no flag are kept; their number is found with function *setdiff*.

The function then proceeds to eliminate in the third filter complex poles, within the interest frequency range (those outside have already been eliminated), that are unnecessary. Function arc of tangent, a 180º-period function, is present in the system of equations solved by *resolution*, and this means that spurious sudden jumps of 180º may appear. This is a delicate operation because it is hard to see if a particular pair of complex conjugate poles is necessary or not. For that purpose the plant's phase variation between adjacent sampling frequencies is compared with the model's phase variation, to see if there are significant changes. The limit of what is significant was empirically found. Whenever significant changes occur poles are searched to find complex poles placed between the two sampling frequencies under examination. The order number of such poles is saved in variable *flags*, so that these may be done away with using *setdiff* as above.

Vector *wzp* holds a set of pole frequencies that actually may correspond to either poles or zeros. If no specification for gain is present, the choice is to assume that they are whatever keeps them stable. So *wzp* is swept, and frequencies of unstable poles are stored in vector *wz* with their sign changed, so as to become stable zeros (that have the very same effect on phase). Stable poles are stores in vector *wp*.

But if a specification for gain was given all possible 2^N combinations of poles and zeros have to be checked, looking for the one corresponding to a gain closer to the desired one. In order to obtain all such combinations, the first 2^N non-negative integers are written with binary digits—these are the numbers that may be written with N or less binary digits—using function *dec2bin* (which is given *N* as second parameter so that left zeros are added in order to obtain always *N* binary digits). When a certain frequency corresponds to a 0, it is assumed to be a zero; when it corresponds to a 1, it is assumed to be a pole. Combinations where complex conjugates are not both poles or both zeros are not taken into account. This is done by reckoning the sum of poles' imaginary parts, to see if it is zero (the sum of zeros' imaginary parts could also have been used). The controller's gain at frequencies where experimental data are available is obtained with function *bode*; the combination of zeros and poles minimising the sum of the square of the differences between desired and controller gains is chosen. The difference is reckoned after correcting both gain sets to ensure zero means, because the controller's static gain is not yet determined.

The periodic nature of function arc of tangent brings another problem that has yet to be dealt with: the controller's phase may differ from experimental data by 180º or by a multiple of 180º. Function *bode* is used to find the controller's phase at sampling frequencies so that it may be compared to the experimental phase. The difference, stored in variable *error*, must be roughly multiple of 90º. The quotient is stored in variable *difference*. If phase is below the data provided in *phase*, zeros must be added to raise it; if it is above, poles must be added to lower it. Such poles and zeros are placed at the origin.

Finally the controller's gain is determined. If no gain data was given, the frequency at which gain should be 1 had to be provided. The gain is found at that frequency (using function *bode*) and gain is set to the inverse. If gain data was given, the controller's gain is determined so that gains' average values are the same. The gain of the controller is once again found with *bode*.

Function *resolution*

The system of trigonometric equations (3.5) is rather complex, but it is possible to linearise it without any approximations. Its solutions may be found solving three problems sequentially. The first one is the linear set of equations

$$\sum_{k=1}^{M}(-1)^{\mathrm{C}\,(k/2)}\,\Phi_{i,k}\mathrm{S}_{1,k}A_i^k = -\Phi_{i,0},\ i = 1\ldots N \tag{3.6}$$

where

$$\Phi_{i,k} = \begin{cases}\mathrm{tg}(\phi_i),\ \exists_{p\in\mathbb{N}_0} k = 2p \\ 1,\ \exists_{p\in\mathbb{N}} k = 2p-1\end{cases} \tag{3.7}$$

$$A_i = \frac{\omega_i}{\omega_1} \tag{3.8}$$

These equations are solved in order to the $\mathrm{S}_{1,k}$.

The second one is the equation

$$y_1^M + \sum_{k=1}^{M}(-1)^k y_1^{M-k}\mathrm{S}_{1,k} = 0 \tag{3.9}$$

This equation is solved in order to y_1 and has *M* solutions.

The third one is the set of (independent) equations

$$\omega_{zpi} = \frac{\omega_1}{y_{1,i}}, \quad i = 1 \ldots N \tag{3.10}$$

solved in order to ω_{zpi}, the pole frequencies wanted.

This linearisation has a drawback: the matrix in system (3.6) is usually a very good example of a very badly conditioned matrix.

Resolution is usually performed by means of the operator \, conceived for solving linear systems of equations. The accuracy of the solution is checked. Whenever errors superior to 1° are found in any of the equations, or whenever an infinite or *NaN* value was obtained, symbolic resolution with Maple is used. This is performed in a separate function, found at the end of the file, called *resmaple*, because interpreting the solution requires many lines of code. By means of a *for* cycle different precision digits are experimented, from 10 to 100 with a step of 5, until Maple finds a solution with an error of less than 1° for all equations. If not even with 100 digits a good result is obtained a warning is shown, since it is likely that the controller will not achieve the desired performance.

The zeros of the polynomial of the second problem are found with command *roots*. It is to be noticed that the coefficients are stored in vector *system* from the independent (in position 1, not 0) up to the highest power. Since function *roots* uses the inverse convention the order must be reversed.

In the third problem those roots that are zero are despised, as they correspond to poles in infinity.

Function *resmaple*

This function calls programme Maple through function *maple* in order to solve system (3.6). This function returns a character string with the solution written in Maple's syntax. The first Maple command used is *linsolve*. If no solution is found (if the system is overdetermined this is to be expected) a solution is searched for by means of a singular value decomposition[6].

If once again no solution is found, the returned variable remains empty. If a solution was found there may be more than one solution; in that case the character string contains a general expression depending on the entries of a vector *_t*, represented by *_t[1][1]*, *_t[1][2]*, etc.. Unfortunately Matlab's command *sym* is unable to understand that notation; so it must be changed to Matlab's usual notation: *t(1,1)*, *t(1,2)*, etc.. For that purpose the string is checked for the presence of the underscore. If this character is found, it is removed; brackets are replaced by parenthesis; and followed parentheses are replaced by a comma. The sole problem is that this may destroy the syntax of an eventual *matrix([…])* command (the ellipsis stands here for the contents of the matrix). So if this word is found the brackets are restored in their positions. As this is over the solution depends on a variable *t*; its entries are set to 1 (any other value could be used; the purpose is simply to obtain a particular solution); the size is set to the maximum possible value.

Now it is possible to proceed as though there were only one solution: function *sym* is used to build a symbolic object from Maple's solution, and that symbolic object is numerically evaluated with function *eval*.

[6] For formulas see (Pina, p. 403-404, 1995).

References

See (Oustaloup, p. 180-183, 1991), (Valério, p. 101-104 and 115-116, 2005c).

3.2. Function *crone2z*

This function is the discrete counterpart of function *crone2* above. Its task is more difficult since a controller given by

$$C\left(z^{-1}\right)=k\frac{\prod_{k=1}^{m}\left(1-b_k z^{-1}\right)}{\prod_{k=1}^{n}\left(1-a_k z^{-1}\right)} \tag{3.11}$$

(instead of (3.3)) will have a frequency response given by

$$C\left(e^{j\omega}\right)=k\frac{\prod_{k=1}^{m}\left(1-b_k e^{-j\omega}\right)}{\prod_{k=1}^{n}\left(1-a_k e^{-j\omega}\right)}=k\frac{\prod_{k=1}^{m}\left(1-b_k\cos\omega+jb_k\sin\omega\right)}{\prod_{k=1}^{n}\left(1-a_k\cos\omega+ja_k\sin\omega\right)} \tag{3.12}$$

where T_s is the sampling time; and there is no way to have another discrete controller with poles but no zeros (or zeros but no poles) to have the same frequency response in what concerns phase, as was the case above. This means that if the number of zeros and poles is n then all n+1 combinations of numbers of poles D and numbers of zeros N that add up to n must be examined to check which will provide a better result. The one resulting in a more accurate reproduction of phase behaviour will be chosen. Incidentally this means that no guarantee of stability is given when the method is used for designing a controller. If this method is used for identifying a model both the gain and phase behaviours must be had into account when distributing zeros and poles.

From the discussion above we see that (3.5) is replaced by the system resulting of equalling (3.12) to ϕ_i.

Syntax (identification method)

```
C = crone2z(w, gain, phase, n, Ts)
```

Arguments (identification method)

- *w* — vector with frequencies ω_1, ω_2,… ω_N in rad/s.
- *gain* — vector with experimental gains (at frequencies found in *w*) in dB.
- *phase* — vector with experimental phases (at frequencies found in *w*) in degrees.
- *n* (optional) — desired number of delays of the model (corresponds to N+D in (3.11)). If this parameter is omitted, *n* is assumed to be N (the number of frequencies in vector *w*), for the system of equations to become square (and easier to solve). It may also be set to *[n1 n2]*. In that case *n1* will be the number of delays in the numerator and *n2* will be the number of delays in the denominator.

❖ *Ts* — sampling time in seconds.

It should be noticed that vectors *w*, *gain* and *phase* may be column vectors or row vectors, but they must all be of the same type.

Syntax (devising a controller)

```
C = crone2z(w, [], phase, n, Ts, w1)
```

Arguments (devising a controller)

❖ *w* — vector with frequencies ω_1, ω_2,… ω_N in rad/s.

❖ *phase* — vector with the phases, in degrees, the controller must have at frequencies found in *w* (ϕ as given by the right-hand side of equation (3.2)).

❖ *n* — desired number of delays of the controller (corresponds to *N*+*D* in (3.11)). This parameter may be set to *[]*. In that case *n* is assumed to be *N* (the number of frequencies in vector *w*), for the system of equations to become square (and easier to solve). It may also be set to *[n1 n2]*. In that case *n1* will be the number of delays in the numerator and *n2* will be the number of delays in the denominator.

❖ *Ts* — sampling time in seconds.

❖ *w1* — frequency, in rad/s, at which the controller will have a unit gain (that is to say, a 0 dB gain). The presence of this parameter is what actually decides if parameter *gain* is taken into account. So if you provide both *gain* and *w1* it is *w1* that matters.

It should be noticed that vectors *w* and *phase* may be column vectors or row vectors, but they must all be of the same type.

Return values

❖ *C* — tf object given according to the applicable formulas referred to above.

Algorithm

The system of equations resulting from equalling (3.12) to ϕ_i is, in the general case, numerically solved since linearisation is only possible if there are no poles or no zeros. In the first case, the phase of (3.12) is given by

$$\arg\left[C\left(e^{j\omega}\right)\right]=\sum_{k=1}^{m}\operatorname{arctg}\frac{b_k\sin\omega}{1-b_k\cos\omega} \tag{3.13}$$

This means that the discrete counterpart of (3.5) is

$$\sum_{k=1}^{m}\operatorname{arctg}\frac{b_k\sin\omega}{1-b_k\cos\omega}=\phi_i,\quad i=1\ldots N \tag{3.14}$$

Solving (3.14) may be in three steps, just like (3.5). The first is solving

$$\begin{aligned}&\sum_{k=1}^{M}\sum_{r=1}^{k}\left[(-1)^{\mathrm{C}(k/2)}\Phi_{i,k}\mathrm{T}_{k-r-1,M-k+1}\mathrm{S}_{1,r}A_i^r B_i^{k-r}\right]=\\&=-\Phi_{i,0}-\sum_{k=1}^{M}\left[(-1)^{\mathrm{C}(k/2)}\Phi_{i,k}\mathrm{T}_{k-1,M-k+1}B_i^k\right],\quad i=1\ldots N\end{aligned}\tag{3.15}$$

where

$$\begin{aligned}&\mathrm{T}_{-1,a}=1\\&\mathrm{T}_{0,a}=a\\&\mathrm{T}_{1,a}=\sum_{i=1}^{a}i\\&\mathrm{T}_{2,a}=\sum_{j=1}^{a}\sum_{i=1}^{j}i\\&\ldots\\&\mathrm{T}_{b,a}=\sum_{i=1}^{a}\mathrm{T}_{b-1,i},\ b\in\mathbb{N}_0\end{aligned}\tag{3.16}$$

$$A_i=\frac{\sin\omega_1}{\sin\omega_i}\tag{3.17}$$

$$B_i=\frac{\cos\omega_1}{\sin\omega_i}-\operatorname{cotg}\omega_i\tag{3.18}$$

$$\Phi_{i,k}=\begin{cases}\operatorname{tg}\left(\phi_i-M\dfrac{\pi}{2}\right),\ \exists_{p\in\mathbb{N}_0}k=2p\\1,\ \exists_{p\in\mathbb{N}}k=2p-1\end{cases}\tag{3.19}$$

These equations are solved in order to the $\mathrm{S}_{1,k}$.

The second is (3.9) again.

The third is

$$b_k=\frac{\operatorname{cosec}\omega_1}{y_{1,k}+\operatorname{cotg}\omega_1}\tag{3.20}$$

If instead of no poles C has no zeros, the resolution is exactly the same, save that instead of (3.19) we will have

$$\Phi_{i,k}=\begin{cases}\operatorname{tg}\left(-\phi_i-M\dfrac{\pi}{2}\right),\ \exists_{p\in\mathbb{N}_0}k=2p\\1,\ \exists_{p\in\mathbb{N}}k=2p-1\end{cases}\tag{3.21}$$

If both zeros and poles are to exist, a numerical method must be employed. But this may also be used even if the problem may be exactly solved as explained above. Since the matrix in (3.15) is always very poorly conditioned, this may even prove to be better than the exact solution. So this toolbox always solves the problem in both ways, keeping the better result.

The code of this file considers separately two cases: first, that in which only the total number of delays in the transfer function is given; then, that in which a specific distribution of delays between numerator and denominator is given.

In the first, all possible combinations of number of delays in the numerator and denominator adding up to *n* have to be tried. This is done with a *for* cycle. All parameters are kept in a single vector *a*, but as variable *ordnum* (this stands for order of the numerator) changes, so does the way this vector is interpreted: the first *ordnum*+1 coefficients are considered to belong to the numerator (the 1 is added to take the a_0 coefficient into account); the remaining ones are belong in the denominator. Minimisation is done with function *fminsearch*. This derivative-free method proved to have good results for this particular task (better than those obtained using derivative-based Newton method, for instance). The starting point is an *a* vector with all coefficients set to 1. This proved to be an acceptable option. The function minimised is found below within the file.

The values of the coefficients and the values of the performance index returned by *fminsearch* are also kept in a structure called *temp*, in fields *coef* and *J*, respectively. The most favourable value is chosen in the end; so no effort is taken to have stable poles and zeros.

After this, the two cases in which exact solutions are possible are considered anew. The function that handles the situation, *crone2zExact*, is also found below within the file. It returns the transfer function (not its coefficients, like the minimisation process) and the performance index. So in the end there are three performance indexes, *J*, *J1* and *J2*.

In the second case, when the structures of the numerator and the denominator are fixed beforehand, no *for* cycle is needed, and *crone2zExact* is only called if possible. Indexes *J1* and *J2* are set to infinity when no corresponding controller exists.

The smallest of all performance indexes is chosen, and, if necessary, the controller is built from parameters stored in *coef*; then an appropriate gain is found. If variable *w1* exists this means setting the gain to 1 at that frequency. In other cases the gain at frequencies *w* is found and the average is compared with the average value in *gain*.

Function *crone2zAux*

This function is minimised by *fminsearch* in *crone2z*. It reckons the frequency behaviour of a particular candidate solution. The vector of coefficients *coef* is split into two, so that the first *ordnum*+1 values are considered to belong to the numerator while the remaining ones are considered to belong in the denominator. If no data on gain was provided the performance index is the norm of the difference between the reckoned phase and the phase to mimic. If vector *gain* was provided the norm of the difference between the reckoned gain and the gain to mimic (both alleviated of the respective averages, since the gain has not been computed yet) is added. This means that this performance index equals an error of 1° in the phase to an error of 1 dB in the gain.

Function *crone2zExact*

This function uses the following convention to know if the transfer function it has to build has poles and no zeros or zeros and no poles: a negative value of its *M* parameter corresponds to the first case, and a positive value of *M* corresponds to the second. Thus in the first case *M* is the number of poles and in the second *M* is the number of zeros.

After ensuring that *phase* and *w* are column (and not row) vectors, function *resolution* is called. Then the controller is found from the parameters returned. This cannot be done as in the main body of function *crone2z* since parameters are returned in a way different from that resulting from minimising the performance index with *fminsearch*. Finally the performance index is found; since the transfer function already exists this is rather easy, since function *bode* can be used to find the frequency behaviour.

Function ***resolution***

This function is rather similar to that described above in section 3.1, save that, of course, equations implemented are those given above in this section.

Function ***resmaple***

This function is essentially the same as the function described above in section 3.1.

References

See the references for function *crone2* and (Valério *et al.*, 2004a), (Valério, p. 104-110, 2005c).

4. Functions ensuring a complex fractional order behaviour for the open-loop

The function described in this section reckons the parameters needed for building a controller that ensures an open-loop dynamic described by (1.2).

4.1. Function *crone3*

This function returns the parameters needed for building a controller *C* that will ensure, in a frequency range $\left[\omega_l;\omega_h\right]$, an open-loop dynamic behaviour given by

$$\begin{aligned} C(s)G(s) &= \mathrm{Re}\left[\frac{s^{a+jb}}{\omega_0}\right] = \mathrm{Re}\left[\frac{s^a e^{\log s^{jb}}}{\omega_0}\right] = \mathrm{Re}\left[\frac{s^a e^{jb\log s}}{\omega_0}\right] = \\ &= \mathrm{Re}\left[\frac{s^a\left[\cos(b\log s)+j\sin(b\log s)\right]}{\omega_0}\right] = \frac{s^a\cos(b\log s)}{\omega_0} \end{aligned} \tag{4.1}$$

It is possible to show that the gain and phase Bode diagrams of this transfer function are nearly linear. The slope of the gain may be positive or negative; the slope of the phase is always negative. To obtain a positive slope we may consider the open-loop transfer function

$$C(s)G(s) = \frac{s^{-a}}{\omega_0\cos(b\log s)} \tag{4.2}$$

For simplicity, this function automatically considers (4.1) if *b* is negative and (4.2) if *b* is positive. Thus the sign of *b* will be the sign of the slope of the phase Bode diagram.

The idea is to choose the differentiation order *a*+*jb* so that the closed-loop will have a resonance gain below a desired value or a damping coefficient above a desired value—and this even in the presence of uncertainty in the parameters of plant *G*.

Let us denote the phase and the gain of the open-loop by *x* and *y*, respectively; so

$$G(s)C(s) = ye^{jx} \tag{4.3}$$

The closed-loop transfer function is

$$\frac{G(s)C(s)}{1+G(s)C(s)} \tag{4.4}$$

This means that the closed-loop gain *g* will be

$$g = \left|\frac{ye^{jx}}{1+ye^{jx}}\right| = \frac{y}{\left|1+y\cos x + jy\sin x\right|} = \\ = \frac{y}{\sqrt{y^2\sin^2 x + 1 + 2y\cos x + y^2\cos^2 x}} = \frac{y}{\sqrt{1+y^2+2y\cos x}} \tag{4.5}$$

In what concerns the closed-loop damping coefficient, it will be (approximately) given by

$$\zeta = -\cos\frac{\pi^2}{2\arccos\dfrac{y^2+2y\left(\cos x - 1\right)+1}{2y}} \tag{4.6}$$

Level curves of these two functions may be seen in Figure 1. Controllers developed with this function have a Nichols plot touching the level curve corresponding to the worst admissible closed-loop resonance gain or damping coefficient, without going beyond. The slope is chosen so that, when plant parameters vary (within given expected ranges), the Nichols plot goes as little as possible beyond that limit curve.

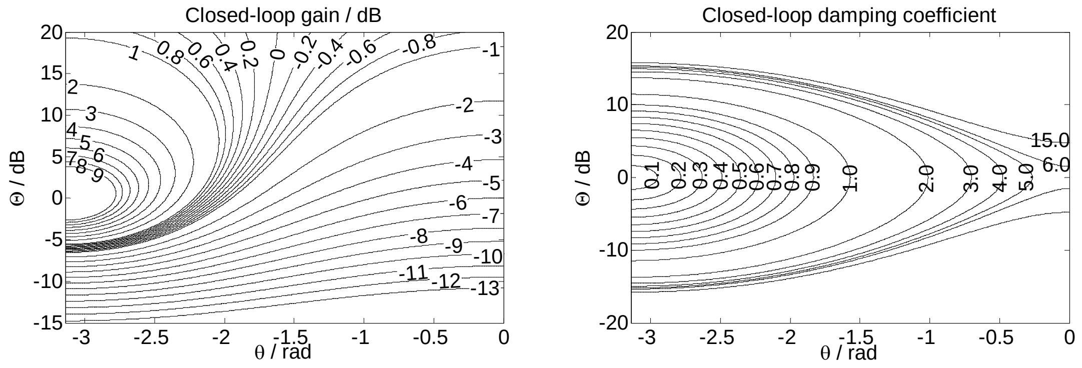


Figure 1. Nichols charts with curves of constant values of closed-loop gain and damping coefficient

Syntax

```
[a, b, w0, w, gainC, phaseC] = crone3(G,...
    vargain, varzeros, varpoles, wl, wh, spec, st)
```

See below for the possible use of an additional parameter, *n*.

Arguments

❖ *G* — LTI object with the plant to control (created with *tf*, *zpk*, or *ss*).

❖ *vargain* — two-position vector with the additive variations that the gain (in absolute value, as given by function *zpkdata*; *not* in dB) may undergo. One of the positions will contain the positive variation and the other the negative one; the order is irrelevant.

❖ *varzeros* — two-column matrix with the additive variations that the zeros (in rad/s, as given by function *zpkdata*) may undergo. The number of lines must be the number of zeros; each line will correspond to a zero. Zeros are ordered according to their frequency, from low to high frequencies. If zeros with the same frequency (such as complex conjugate zeros) are present, the order will be that of the argument, from negative to positive arguments. These are the criteria of function *sort*; for more information see the help of that built-in function. If there are complex conjugate zeros, the complex conjugates of the variations given for the zero that appears first will be assumed for its complex conjugate appearing later (so that line of the matrix is skipped). One of the columns of the matrix will contain the positive variations and the other the negative ones; the order is irrelevant: different lines may even have them in a different order.

❖ *varpoles* — two-column matrix with the additive variations that the poles (in rad/s, as given by function *zpkdata*) may undergo. The number of lines must be the number of poles; each line will correspond to a pole. Poles are ordered according to their frequency, from low to high frequencies. If poles with the same frequency (such as complex conjugate poles) are present, the order will be that of the argument, from negative to positive arguments. These are the criteria of function *sort*; for more information see the help of that built-in function. If there are complex conjugate poles, the complex conjugates of the variations given for the pole that appears first will be assumed for its complex conjugate appearing later (so that line of the matrix is skipped). One of the columns of the matrix will contain the positive variations and the other the negative ones; the order is irrelevant: different lines may even have them in a different order.

❖ *wl* — lower limit of $\left[\omega_l;\omega_h\right]$, given in rad/s.

❖ *wh* — upper limit of $\left[\omega_l;\omega_h\right]$, given in rad/s.

❖ *spec* — the performance specification the closed-loop must verify. It may be a closed-loop resonance gain in dB or a closed-loop damping coefficient.

❖ *st* (optional) — variable for choosing the type of performance specification. It should be one of the following strings:

- ‘g’ (default) — closed-loop resonance gain;
- ‘d’ — closed-loop damping coefficient.

Return values

❖ *a* — real part of the differentiation order, as seen in (4.1).

❖ *b* — imaginary part of the differentiation order, as seen in (4.1).

❖ *w0* — gain adjustment factor, as seen in (4.1).

❖ *w* — a column vector with fifty frequencies, in rad/s, logarithmically distributed in $\left[\omega_l;\omega_h\right]$.

❖ *gainC* — column vector with the gains, in dB, the controller must have at frequencies found in *w*.

❖ *phaseC* — column vector with the phases, in degrees, the controller must have at frequencies found in *w*.

These three variables allow synthesising the controller, namely using function *crone2* or function *crone2z*. The fifty frequencies of *w* are necessary for internal calculations in *crone3*, but are clearly too many for using with functions *crone2* or *crone2z*. Thus it is advisable to make a selection. Actually it is possible to call the function with syntax

```
[a, b, w0, w, gainC, phaseC, C] = crone3(G,...
    vargain, varzeros, varpoles, wl, wh, spec, st, n)
```

The additional parameter and the additional return value are:

❖ *n* — desired number of poles and zeros of the controller, if *G* is continuous; or desired number of delays of the controller, if *G* is discrete. Corresponds to parameter *n* in both functions *crone2* and *crone2z*.

❖ *C* — controller reckoned with *crone2* or *crone2z* from variables *w*, *gainC* and *phaseC*. However, given the excessive number of sampling frequencies it is very likely that the result will be poor.

Algorithm

First of all this function supplies a value for *st* if it was not given and obtains the relevant data (gain, poles, zeros, sampling time and variable) from plant *G*. Poles and zeros must be sorted so that it is possible to know which line of variables *varzeros* and *varpoles* corresponds to which zero or pole. This is done with built-in function *sort*; rules about the order in which information is to be found in *varzeros* and *varpoles* are set by what that function does. The variable of the transfer function may be *s*, *z* or z^{-1}, and so it is necessary to store that information as well.

Then the slope of the Nichols plot is found. For that purpose a vector *w* with frequencies is found using function *logspace*. The value of the slope is optimised with function *fminsearch*; that proved to be the best option. The optimisation is performed twice for two different initial values, one positive and one negative.

Then it is necessary to find the phase at which the curve corresponding to the provided value of *spec* has the slope *yslope*. This is done using function *fminsearch* again. Since the phase is the *x*-axis of the Nichols plot, this particular phase is stored into variable *x*—just as the slope, which is a derivative of the gain, is stored into variable *yslope*. Then the gain corresponding to *x* is found, and stored into variable *y* because gains are the *y*-axis of the Nichols plot. This is done using several *if*s, to deal with different specifications and different branches of the curves.

The next step is finding the complex order that leads to a Nichols plot that has a slope *yslope* and passes through point defined by coordinates *x* and *y*. First estimates are obtained linearising the equations (see function *crone3Aux3* for an explanation of the linearised formulas); then final values of *a* and *b* are found once more using *fminsearch*.

The final step is obtaining the frequency behaviour of the controller, that is equal to the frequency behaviour of the open-loop minus the frequency behaviour of the plant. For that purpose the frequency behaviour of the open-loop is found from *a* and *b* (the Bode diagram is assumed linear); a phase correction is necessary to ensure that it will lie somewhere in $\left[-360^{\text{o}};0^{\text{o}}\right]$; and a gain correction is necessary because the tangency to the desired curve may be found in other phase interval. To accurately verify this, the Nichols plot over an entire range of 360º is needed. The logarithms of the frequencies in which that is to be found are stored in *wf_0log* and *wf_360log*. Then the frequency behaviour between these frequencies is obtained (just like *gain* and *phase* were obtained) and stored in *tempGain* and *tempPhase*. Then, if the performance specification is not met (this is verified with the same formulas that will be described below in function *reckoning*), the gain is shifted down by the value corresponding to one period of 360º.

Only now is the frequency behaviour of *G* reckoned with *bode* and subtracted from the frequency behaviour of the open-loop. Lastly the value of *w0* is obtained by interpolation: we know the values of *gain* for several values of *w*, so this is just interpolating data to find the frequency for which gain will be $20\log_{10}\cosh\left(b\frac{\pi}{2}\right)$.

A controller is obtained with one of functions *crone2* or *crone2z* only if there are enough input and output arguments. The choice is done according to the sampling time of the plant, that is zero for frequency domain models. Thus the transfer function returned will be in the frequency domain if the plant is given in the frequency domain, and will be in the discrete-time domain if the plant is given in the discrete-time domain. As stated above, the fifty frequencies of *w* are certainly excessive for the algorithms of both functions *crone2* or *crone2z* and that is why results are unlikely to be acceptable.

Function *crone3Aux1*

This function is one of the three functions minimised by *fminsearch* in *crone3*. It reckons the adequacy of a particular open-loop's Nichols diagram slope under plant variations.

The first thing to do is to find the phase at which the curve corresponding to the provided value of *spec* has the slope *yslope*, and the corresponding gain. This is done exactly as in function *crone3*.

The next step is finding the complex order that leads to a Nichols plot that has a slope under consideration *yslope* and passes through point defined by coordinates *x* and *y*. This is again done exactly as in function *crone3*. If the real or imaginary parts are themselves complex, this means that no solution for the equations was found; the situation is deemed impossible, and the function returns *error* accordingly set to infinity.

It is now possible to reckon the frequency behaviour of the open-loop. This is done for frequencies found in vector *w* and the results are stored in variables *bodeGain* and *bodePhase*. These names reflect that it is possible with these vectors to draw the Bode diagram. (Actually it is possible to know the geometric location of a plant's Nichols diagram without knowing which frequency corresponds to each point, being thus impossible to draw the corresponding Bode diagram.) Phase and gain corrections that follow are exactly like those found in *crone3*.

All this is a preparation for what follows: now that the frequency behaviour of the open-loop in known, we will see how it will be damaged by plant uncertainties. This is done in three steps.

In the first step gain variations are considered. These affect not only the gain, but also the phase, if the gain changes sign. The new closed-loop resonance gain or damping coefficient is reckoned with function *reckoning*, and the variation towards the value given in *spec* is stored in vector *delta*.

In the second step zeros are considered. Two *for* cycles are used: one for sweeping all zeros, another for dealing with the columns of the matrix. Complex conjugate poles must be dealt with simultaneously, because if one is changed without changing the other they would not be conjugate anymore. So whenever a complex zero is found, the complex conjugate is dealt with; the complex conjugate of the variation of the other zero must be assumed so that the final result is still a pair of complex conjugates. Complex conjugates being ordered according to their phase, those with negative imaginary parts appear first; so if a complex zero with a positive imaginary part is found we may be sure that it was already dealt with before. That is why, right after the first *for* cycle, the one sweeping zeros, such zeros are skipped using a *continue* command. The transfer function built for assessing the effect of each variation has the nominal zero (or pair of zeros) as pole (or poles), to cancel their effect; the modified zero (or pair of zeros) is in the numerator. The frequency behaviour of this transfer function is obtained and put together with that of the open-loop; the new closed-loop resonance gain or damping coefficient is reckoned with function *reckoning*, and the variation towards the value given in *spec* is stored in vector *delta*.

In the third step poles are considered in the same manner as zeros were.

The last thing to do is to obtain, from vector *delta*, the deterioration of the closed-loop resonance gain or of the damping coefficient. For that purpose variations that are actually improvements are set to zero; a sum of squares is used.

Function *reckoning*

This function exists at the end of file *crone3Aux1* to save repetition of code. It receives the frequency behaviour of a function and reckons the corresponding closed-loop gain or closed-loop damping coefficient. What is important is the most unfavourable value, so the maximum closed-loop gain or minimum closed-loop damping coefficient is returned.

Function *crone3Aux2*

This function is minimised by *fminsearch* in both *crone3* and *crone3Aux1*. It exists for determining in what phase does a certain closed-loop gain curve or closed-loop damping coefficient curve have a given derivative. It receives a point *x* and reckons the derivative using the appropriate expression (which will be (4.10), (4.11), (4.12), (4.16) or (4.17), all below); the square of the difference to the desired derivative is returned.

The deduction of expressions referred to follows. Level curves of (4.5) are given by

$$\begin{aligned} & g=\frac{y}{\sqrt{1+y^2+2y\cos x}} \Leftrightarrow \frac{y^2}{g^2}=1+y^2+2y\cos x \Leftrightarrow \\ & \Leftrightarrow y^2\left(1-\frac{1}{g^2}\right)+y2\cos x+1=0 \end{aligned} \tag{4.7}$$

Two cases must be considered. If g is 1 (or 0 dB), we have

$$y2\cos x+1=0 \Leftrightarrow y=-\frac{1}{2\cos x} \tag{4.8}$$

If it is not,

$$y=\frac{-2\cos x\pm\sqrt{4\cos^2 x-4\left(1-\frac{1}{g^2}\right)}}{2\left(1-\frac{1}{g^2}\right)}=\frac{-\cos x\pm\sqrt{\cos^2 x-\left(1-\frac{1}{g^2}\right)}}{1-\frac{1}{g^2}} \tag{4.9}$$

Figure 1 shows that for g>1 (that is to say, if g>0 dB) the two signs correspond to the two branches of each curve: actually the minus sign corresponds to the lower part of the curve, while the plus sign corresponds to the upper part. For g<1 (or g<0 dB) there is only one branch; plotting both expressions shows that it is the minus sign that is correct.

Since these curves are plotted with a logarithmic scale in the y-axis, we now need the derivatives of these expressions given in dB. The results are as follows. For g<1,

$$\frac{d}{dx}20\log_{10}\frac{-\cos x-\sqrt{\cos^2 x-\left(1-\frac{1}{g^2}\right)}}{1-\frac{1}{g^2}}=$$

$$=\frac{20}{\log 10}\frac{1-\frac{1}{g^2}}{-\cos x-\sqrt{\cos^2 x-\left(1-\frac{1}{g^2}\right)}}\frac{\sin x-\frac{2\cos x(-\sin x)}{2\sqrt{\cos^2 x-\left(1-\frac{1}{g^2}\right)}}}{1-\frac{1}{g^2}}=$$

$$=\frac{20}{\log 10}\frac{\sin x+\frac{\sin 2x}{2\sqrt{\cos^2 x-\left(1-\frac{1}{g^2}\right)}}}{-\cos x-\sqrt{\cos^2 x-\left(1-\frac{1}{g^2}\right)}} \tag{4.10}$$

For g=1,

$$\frac{d}{dx}20\log_{10}-\frac{1}{2\cos x}=\frac{20}{\log 10}\frac{2\cos x}{-1}\frac{1}{4\cos^2 x}(-2\sin x)=$$
$$=\frac{20}{\log 10}\frac{\sin x}{\cos x}=\frac{20}{\log 10}\operatorname{tg} x \tag{4.11}$$

For g>1 and the lower part of the curve, formula (4.10) holds. For g>1 and the upper part of the curve,

$$\frac{d}{dx}20\log_{10}\frac{-\cos x+\sqrt{\cos^2 x-\left(1-\frac{1}{g^2}\right)}}{1-\frac{1}{g^2}}=$$
$$=\frac{20}{\log 10}\frac{\sin x-\frac{\sin 2x}{2\sqrt{\cos^2 x-\left(1-\frac{1}{g^2}\right)}}}{-\cos x+\sqrt{\cos^2 x-\left(1-\frac{1}{g^2}\right)}} \tag{4.12}$$

Level curves of (4.6) are given by

$$\begin{aligned}
&\zeta = -\cos\frac{\pi^2}{2\arccos\frac{y^2+2y\left(\cos x-1\right)+1}{2y}} \Leftrightarrow \\
&\Leftrightarrow \frac{\arccos-\zeta}{\pi^2} = \frac{1}{2\arccos\frac{y^2+2y\left(\cos x-1\right)+1}{2y}} \Leftrightarrow \\
&\Leftrightarrow \arccos\frac{y^2+2y\left(\cos x-1\right)+1}{2y} = \frac{\pi^2}{2\arccos-\zeta} \Leftrightarrow \\
&\Leftrightarrow y^2+2y\left(\cos x-1\right)+1 = 2y\cos\frac{\pi^2}{2\arccos-\zeta} \Leftrightarrow \\
&\Leftrightarrow y^2+y2\left(\cos x-1-\cos\frac{\pi^2}{2\arccos-\zeta}\right)+1=0
\end{aligned} \tag{4.13}$$

Let us make

$$\begin{aligned}
&\mathsf{B} = \cos x-1-\cos\frac{\pi^2}{2\arccos-\zeta} \\
&\Rightarrow \frac{d\mathsf{B}}{dx} = -\sin x
\end{aligned} \tag{4.14}$$

Then

$$y = \frac{-2\mathsf{B} \pm\sqrt{4\mathsf{B}^2-4}}{2} \Leftrightarrow y = -\mathsf{B} \pm\sqrt{\mathsf{B}^2-1} \tag{4.15}$$

Figure 1 shows that the two signs correspond to the two branches of each curve: actually the minus sign corresponds to the lower part of the curve, while the plus sign corresponds to the upper part.

Since these curves are plotted with a logarithmic scale in the *y*-axis, we now need the derivatives of these expressions given in dB. The results are as follows. For the lower part of the curves,

$$\begin{aligned}
&\frac{d}{dx}20\log_{10}\left[-\mathsf{B} -\sqrt{\mathsf{B}^2-1}\right] = \\
&= \frac{20}{\log 10}\frac{-\frac{d\mathsf{B}}{dx}-\frac{1}{2}\frac{2\mathsf{B}\frac{d\mathsf{B}}{dx}}{\sqrt{\mathsf{B}^2-1}}}{-\mathsf{B} -\sqrt{\mathsf{B}^2-1}} = \frac{20}{\log 10}\frac{\sin x+\frac{\mathsf{B}\sin x}{\sqrt{\mathsf{B}^2-1}}}{-\mathsf{B} -\sqrt{\mathsf{B}^2-1}}
\end{aligned} \tag{4.16}$$

For the upper part of the curves,

$$\frac{d}{dx}20\log_{10}\left[-\mathrm{B}+\sqrt{\mathrm{B}^{2}-1}\right]=\frac{20}{\log 10}\frac{\sin x-\dfrac{\mathrm{B}\ \sin x}{\sqrt{\mathrm{B}^{2}-1}}}{-\mathrm{B}+\sqrt{\mathrm{B}^{2}-1}} \tag{4.17}$$

Function *crone3Aux3*

This function is minimised by *fminsearch* in both *crone3* and *crone3Aux1*. It is intended to determine the differentiation order *a*+*jb* corresponding to a Nichols diagram that simultaneously verifies two conditions: its slope is equal to the desired slope, *yslope*; it passes through point (*x*;*y*). These conditions, if (4.1) is considered, may be expressed by

$$\begin{cases} y'=-20\dfrac{a}{b}\log_{10}e\,\mathrm{cotgh}\left(b\dfrac{\pi}{2}\right) \\ \left(20\log_{10}\cosh\left(b\dfrac{\pi}{2}\right);a\dfrac{\pi}{2}\right)\in\left\{(\xi;\upsilon):(\xi;\upsilon)=(x;y)+\lambda(1,y'),\quad \lambda\in\mathbb{R}\right\}\end{cases} \tag{4.18}$$

The straight line in the second equation may be rewritten as follows:

$$\begin{cases}\xi=x+\lambda \\ \upsilon=y+\lambda y'\end{cases}\Leftrightarrow\begin{cases}\lambda=\xi-x \\ \upsilon=y+\xi y'-xy'\end{cases},\quad \lambda\in\mathbb{R} \tag{4.19}$$

This second equation allows rewriting the second equation of (4.18):

$$\begin{cases} y'=-20\dfrac{a}{b}\log_{10}e\,\mathrm{cotgh}\left(b\dfrac{\pi}{2}\right) \\ a\dfrac{\pi}{2}=y-xy'+20y'\log_{10}\cosh\left(b\dfrac{\pi}{2}\right)\end{cases} \tag{4.20}$$

If (4.2) is considered instead of (4.1), the slope of the phase will change sign, and thus (4.20) becomes

$$\begin{cases} y'=20\dfrac{a}{b}\log_{10}e\,\mathrm{cotgh}\left(b\dfrac{\pi}{2}\right) \\ a\dfrac{\pi}{2}=y-xy'+20y'\log_{10}\cosh\left(b\dfrac{\pi}{2}\right)\end{cases} \tag{4.21}$$

This function *crone3Aux3* receives a pair of values for *a* and *b* and returns the sum of the square of the differences of the two members of the two equations in (4.20) or (4.21).

It is possible to find a polynomial approximation of (4.20) to obtain approximate solutions. This is done before the function is called, so that it begins already near the solution. We know that

$$\sinh(x)=\sum_{n=0}^{+\infty}\frac{x^{2n+1}}{(2n+1)!}=x+\circ\left(x^2\right) \tag{4.22}$$

$$\cosh(x)=\sum_{n=0}^{+\infty}\frac{x^{2n}}{(2n)!}=1+\frac{x^2}{2}+\circ\left(x^2\right) \tag{4.23}$$

Thus

$$\operatorname{tgh}x=\frac{\sinh x}{\cosh x}\approx\frac{x}{1+\frac{x^2}{2}} \tag{4.24}$$

And since

$$\begin{aligned}
&h(x)=\log_{10}\cosh x \quad\Rightarrow\quad h(0)=0\\
&h'(x)=\frac{1}{\log 10}\frac{\sinh x}{\cosh x} \quad\Rightarrow\quad h'(0)=0\\
&h''(x)=\frac{1}{\log 10}\frac{\cosh^2 x-\sinh^2 x}{\cosh^2 x}=\frac{1}{\log 10}\frac{1}{\cosh^2 x} \quad\Rightarrow\quad h''(0)=\frac{1}{\log 10}
\end{aligned} \tag{4.25}$$

we will have

$$h(x)=\frac{1}{\log 10}\frac{x^2}{2}+\circ\left(x^2\right) \tag{4.26}$$

Using (4.24) and (4.26), system (4.20) may be approximated by

$$\begin{cases}
y'=20\dfrac{a}{-b\log 10\dfrac{b\frac{\pi}{2}}{1+\dfrac{\left(b\frac{\pi}{2}\right)^2}{2}}}\\[2ex]
a\dfrac{\pi}{2}=y-xy'+20y'\dfrac{\left(b\frac{\pi}{2}\right)^2}{2\log 10}
\end{cases}
\Rightarrow
\begin{cases}
\dfrac{-y'\log 10 b^2\frac{\pi}{2}}{1+\dfrac{\left(b\frac{\pi}{2}\right)^2}{2}}=20\dfrac{y-xy'+20y'\dfrac{\left(b\frac{\pi}{2}\right)^2}{2\log 10}}{\frac{\pi}{2}}\\[2ex]
a=\dfrac{y-xy'+20y'\dfrac{\left(b\frac{\pi}{2}\right)^2}{2\log 10}}{\frac{\pi}{2}}
\end{cases} \tag{4.27}$$

Let us make Q $=\left(b\frac{\pi}{2}\right)^2$; then the first equation becomes

$$\frac{-y'\log 10 Q}{1+\frac{Q}{2}} = 20y - 20xy' + \frac{200yQ}{\log 10} \Leftrightarrow$$

$$\Leftrightarrow -y'\log 10 Q = 20y + 10yQ - 20xy' - 10xyQ + \frac{200yQ}{\log 10} + \frac{100yQ^2}{\log 10} \Leftrightarrow \tag{4.28}$$

$$\Leftrightarrow Q^2 \frac{100y'}{\log 10} + Q\left(10y - 10xy' + \frac{200y'}{\log 10} + y'\log 10\right) + \left(20y - 20xy'\right) = 0$$

This quadratic is solved, and then it is possible to obtain the estimates

$$\begin{cases} b = -\dfrac{2\sqrt{Q}}{\pi} \\ a = \dfrac{y - xy' + \dfrac{10yQ}{\log 10}}{\dfrac{\pi}{2}} \end{cases} \tag{4.29}$$

The sign of *b* is negative because (4.20) corresponds to a negative slope of the phase diagram. The real solution for *b* is, of course, to be chosen.

Approximating (4.21) is pretty much the same; (4.28) and (4.29) will become

$$Q^2 \frac{100y'}{\log 10} + Q\left(10y - 10xy' + \frac{200y'}{\log 10} - y'\log 10\right) + \left(20y - 20xy'\right) = 0 \tag{4.30}$$

$$\begin{cases} b = \dfrac{2\sqrt{Q}}{\pi} \\ a = \dfrac{y - xy' + \dfrac{10yQ}{\log 10}}{\dfrac{\pi}{2}} \end{cases} \tag{4.31}$$

References

See (Oustaloup *et al.*, 2000), (Oustaloup, p. 283-284 and 290-292, 1991), (Valério, p. 116-120, 2005c).

5. Identification of fractional models

Functions described in this section allow finding a fractional model for a plant from its frequency behaviour. Usual least-squares methods may be employed with time response data.

Functions *crone2* and *crone2z*, described above in subsections 3.1 and 3.2, are also fit for identifying models, but integer ones.

5.1. Function *hartley*

This function finds a model for a plant given its frequency behaviour, using the algorithm developed by Hartley and Lorenzo (2003). The model may be either

$$G(s) = c_n s^{nQ} + c_{n-1} s^{(n-1)Q} + \ldots + c_2 s^{2Q} + c_1 s^Q + c_0 \tag{5.1}$$

or

$$G(s) = \frac{1}{c_n s^{nQ} + c_{n-1} s^{(n-1)Q} + \ldots + c_2 s^{2Q} + c_1 s^Q + c_0} \tag{5.2}$$

Syntax

```
G = hartley (w, gain, phase, Q, n, type)
```

Algorithm

- *w* — vector with frequencies in rad/s where the response of the plant is known.
- *gain* — vector with gains in dB at frequencies found in *w*.
- *phase* — vector with the phases in degrees at frequencies found in *w*.
- *Q* — the maximum common multiple of powers of *s*, as seen in (5.1) and (5.2).
- *n* — desired number of poles or zeros of the model. This parameter may be omitted, in which case the number of frequencies in *w* minus 1 will be assumed.
- *type* (optional) — the type of model. It may have one of two values:
  - 'num' — the form of (5.1) will be used;
  - 'den' (default) — the form of (5.2) will be used.

Return values

- *G* — vector with coefficients *c* (beginning with c_n) is returned.

Limitations

Elements of *G* should be real, although there are always residual complex parts. If these are not residual, that means that the model is very poor.

Algorithm

This function implements an identification method stemming from the fact that, for a plant given by (5.1),

$$\begin{bmatrix} G(j\omega_1) \\ G(j\omega_2) \\ \vdots \\ G(j\omega_M) \end{bmatrix} = \begin{bmatrix} 1 & (j\omega_1)^Q & (j\omega_1)^{2Q} & \cdots & (j\omega_1)^{nQ} \\ 1 & (j\omega_2)^Q & (j\omega_2)^{2Q} & \cdots & (j\omega_2)^{nQ} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & (j\omega_M)^Q & (j\omega_M)^{2Q} & \cdots & (j\omega_M)^{nQ} \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix} \tag{5.3}$$

where $\omega_1, \omega_2, \ldots, \omega_m$ are the sampling frequencies that will be given in vector *w*. So finding the coefficients means only solving (5.3). Should the model be that of (5.2), the only change is that the left-hand side of (5.3) will have the inverse of the frequency behaviour.

After finding missing parameters, the function ensures that *w*, *gain* and *phase* are column vectors, so that the matrix quantities of (5.3) will be properly built. Vector *g*, with the left-hand side of (5.3), is built from *gain* and *phase*. The matrix of (5.3) is stored in *W* by means of a *for* cycle. If the system is square, operator \ is used to solve it; otherwise a pseudo-inverse of *W* is obtained with *pinv* (as suggested in the reference below).

References

See (Hartley *et al.*, 2003).

5.2. Function *levy*

This function finds a model for a plant given its frequency behaviour $G(j\omega)$, using a variation of Levy's identification method. The model is of the form

$$\hat{G}(s) = \frac{b_m s^{mQ} + b_{m-1} s^{(m-1)Q} + b_{m-2} s^{(m-2)Q} + \ldots + b_1 s^Q + b_0}{a_n s^{nQ} + a_{n-1} s^{(n-1)Q} + a_{n-2} s^{(n-2)Q} + \ldots + a_1 s^Q + 1} \tag{5.4}$$

and is found minimising the square norm

$$E = G(j\omega)\left[a_n (j\omega)^{nQ} + \ldots + a_1 (j\omega)^Q + 1\right] - \left[b_m (j\omega)^{mQ} + \ldots + b_1 (j\omega)^Q + b_0\right] \tag{5.5}$$

at all frequencies.

Syntax

```
[G, J, handle] = levy (w, gain, phase, Q, n, m)
```

Arguments

❖ *w* — vector with frequencies in rad/s where the response of the plant is known.
❖ *gain* — vector with gains in dB at frequencies found in *w*.
❖ *phase* — vector with the phases in degrees at frequencies found in *w*.

❖ *Q* — the maximum common multiple of powers of *s*, as seen in (5.4).
❖ *n* — desired number of poles of the model.
❖ *m* — desired number of zeros of the model.

Return values

❖ *G* — structure with two fields, *num* and *den*, containing the coefficients of (5.4) (beginning with the highest order ones).
❖ *J* — index showing how accurate the identified model is, given by

$$J = \frac{1}{n_\omega}\sum_{i=1}^{n_\omega}\left|G\left(j\omega\right)-\hat{G}\left(j\omega\right)\right|^2 \tag{5.6}$$

where n_ω is the number of frequencies in parameter *w* and *G* is the plant that provided the frequency behaviour.
❖ *handle* — handle to a figure with a Bode plot, at frequencies *w*, of the provided data (shown with blue dots) and the performance of the identified model (shown as a continuous green line). If this output parameter does not exist, the figure is not drawn.

Algorithm

This function implements Levy's identification method generalised for the fractional case. This method finds the parameters, for an experimental frequency response given by $G\left(j\omega\right)=R\left(\omega\right)+jI\left(\omega\right)$, by solving

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix}\begin{bmatrix} b_0 \\ \vdots \\ b_m \\ a_1 \\ \vdots \\ a_n \end{bmatrix}=\begin{bmatrix} e \\ g \end{bmatrix} \tag{5.7}$$

where

$$A_{l,c}=\sum_{p=1}^{f}\left\{-\mathrm{Re}\left[\left(j\omega_p\right)^{lQ}\right]\mathrm{Re}\left[\left(j\omega_p\right)^{cQ}\right]-\mathrm{Im}\left[\left(j\omega_p\right)^{lQ}\right]\mathrm{Im}\left[\left(j\omega_p\right)^{cQ}\right]\right\},$$
$$l=0\ldots m \wedge c=0\ldots m \tag{5.8}$$

$$B_{l,c}=\sum_{p=1}^{f}\left\{\mathrm{Re}\left[\left(j\omega_p\right)^{lQ}\right]\mathrm{Re}\left[\left(j\omega_p\right)^{cQ}\right]R_p+\mathrm{Im}\left[\left(j\omega_p\right)^{lQ}\right]\mathrm{Re}\left[\left(j\omega_p\right)^{cQ}\right]I_p-\right.$$
$$\left.-\mathrm{Re}\left[\left(j\omega_p\right)^{lQ}\right]\mathrm{Im}\left[\left(j\omega_p\right)^{cQ}\right]I_p+\mathrm{Im}\left[\left(j\omega_p\right)^{lQ}\right]\mathrm{Im}\left[\left(j\omega_p\right)^{cQ}\right]R_p\right\}, \tag{5.9}$$
$$l=0\ldots m \wedge c=1\ldots n$$

$$C_{l,c} = \sum_{p=1}^{f} \left(\left\{ \mathrm{Im}\left[\left(j\omega_p \right)^{lQ} \right] I_p - \mathrm{Re}\left[\left(j\omega_p \right)^{lQ} \right] R_p \right\} \mathrm{Re}\left[\left(j\omega_p \right)^{cQ} \right] + \right.$$
$$\left. + \left\{ -\mathrm{Re}\left[\left(j\omega_p \right)^{lQ} \right] I_p - \mathrm{Im}\left[\left(j\omega_p \right)^{lQ} \right] R_p \right\} \mathrm{Im}\left[\left(j\omega_p \right)^{cQ} \right] \right), \quad (5.10)$$
$$l = 1 \ldots n \wedge c = 0 \ldots m$$

$$D_{l,c} = \sum_{p=1}^{f} \left[\left(R_p^2 + I_p^2 \right) \left\{ \mathrm{Re}\left[\left(j\omega_p \right)^{lQ} \right] \mathrm{Re}\left[\left(j\omega_p \right)^{cQ} \right] + \right. \right.$$
$$\left. \left. + \mathrm{Im}\left[\left(j\omega_p \right)^{lQ} \right] \mathrm{Im}\left[\left(j\omega_p \right)^{cQ} \right] \right\} \right], \quad l = 1 \ldots n \wedge c = 1 \ldots n \quad (5.11)$$

$$e_{l,1} = \sum_{p=1}^{f} \left\{ -\mathrm{Re}\left[\left(j\omega_p \right)^{lQ} \right] R_p - \mathrm{Im}\left[\left(j\omega_p \right)^{lQ} \right] I_p \right\}, \quad l = 0 \ldots m \quad (5.12)$$

$$g_{l,1} = \sum_{p=1}^{f} \left\{ -\mathrm{Re}\left[\left(j\omega_p \right)^{lQ} \right] \left(R_p^2 + I_p^2 \right) \right\}, \quad l = 1 \ldots n \quad (5.13)$$

The function always uses operator \ to solve system (5.7). A warning about a poor number of condition means almost certainly a poor result.

Performance index *J* is reckoned only if required. The frequency behaviours corresponding to the numerator and the denominator are found with *for* cycles and stored in *tempNum* and *tempDen* respectively. The result is compared to the behaviour found from variables *gain* and *phase*.

The Bode plot is also drawn only if required. Frequency behaviour values found for *J* are not reused, for they may be too few. So a new vector *wNew* with at least 50 frequencies (or more, if *w* has more than 50 frequencies) is found. These are in the same range of *w*, extended to reach the limits of the first and last decades considered (thus improving the look of the plots). Frequency behaviour is found again, just as for reckoning *J*, and the resulting gain and phase stored in *gainNew* and *phaseNew*. These are the values plotted.

References

See (Levy, 1959), (Valério *et al.*, 2005b), (Valério, p. 87-91, 2005c)

5.3. Function *vinagre*

This function finds a model for a plant given its frequency behaviour, using a variation of the identification method of function *levy*. The model is given by

$$\hat{G}(s) = \frac{b_m s^{mQ} + b_{m-1} s^{(m-1)Q} + b_{m-2} s^{(m-2)Q} + \ldots + b_1 s^Q + b_0}{a_n s^{nQ} + a_{n-1} s^{(n-1)Q} + a_{n-2} s^{(n-2)Q} + \ldots + a_1 s^Q + 1} \quad (5.14)$$

and is found minimising the norm

$$E' = wG(j\omega) \left[a_n (j\omega)^{nQ} + \ldots + a_1 (j\omega)^Q + 1 \right] - \left[b_m (j\omega)^{mQ} + \ldots + b_1 (j\omega)^Q + b_0 \right] \quad (5.15)$$

Weights *w* are frequency-dependent. If sampling frequencies are $\omega_i, i = 1 \dots f$, then

$$w = \begin{cases} \dfrac{\omega_2 - \omega_1}{2\omega_1^2} & \text{if } i = 1 \\ \dfrac{\omega_{i+1} - \omega_{i-1}}{2\omega_i^2} & \text{if } 1 < i < f \\ \dfrac{\omega_f - \omega_{f-1}}{2\omega_f^2} & \text{if } i = f \end{cases} \tag{5.16}$$

Weights are intended to improve the quality of the approximation for low frequencies.

Syntax

```
[G, J, handle] = vinagre (w, gain, phase, Q, n, m)
```

Arguments

- ❖ *w* — vector with frequencies in rad/s where the response of the plant is known.
- ❖ *gain* — vector with gains in dB at frequencies found in *w*.
- ❖ *phase* — vector with the phases in degrees at frequencies found in *w*.
- ❖ *Q* — the maximum common multiple of powers of *s*, as seen in (5.4).
- ❖ *n* — desired number of poles of the model.
- ❖ *m* — desired number of zeros of the model.

Return values

❖ *G* — structure with two fields, *num* and *den*, containing the coefficients of (5.4) (beginning with the highest order ones).

❖ *J* — index showing how accurate the identified model is, given by

$$J = \frac{1}{n_\omega} \sum_{i=1}^{n_\omega} \left| G(j\omega) - \hat{G}(j\omega) \right|^2 \tag{5.17}$$

where n_ω is the number of frequencies in parameter *w* and *G* is the plant that provided the frequency behaviour.

❖ *handle* — handle to a figure with a Bode plot, at frequencies *w*, of the provided data (shown with blue dots) and the performance of the identified model (shown as a continuous green line). If this output parameter does not exist, the figure is not drawn.

Algorithm

This function's code is similar to that of function *levy*.

References

See (Vinagre, p. 140-141, 2001), (Valério, 2005, p. 91-93).

5.4. Function *sanko*

This function finds a model for a plant given its frequency behaviour, using Sanathanan's and Koerner's recursive variation of the identification method of function *levy*. The model is given by

$$\hat{G}(s)=\frac{b_m s^{mQ}+b_{m-1}s^{(m-1)Q}+b_{m-2}s^{(m-2)Q}+\ldots+b_1 s^Q+b_0}{a_n s^{nQ}+a_{n-1}s^{(n-1)Q}+a_{n-2}s^{(n-2)Q}+\ldots+a_1 s^Q+1} \tag{5.18}$$

and is found minimising the norm

$$E'=wG(j\omega)\left[a_n(j\omega)^{nQ}+\ldots+a_1(j\omega)^Q+1\right]-\left[b_m(j\omega)^{mQ}+\ldots+b_1(j\omega)^Q+b_0\right] \tag{5.19}$$

Weights w are frequency-dependent and iteration-dependent. If sampling frequencies are $\omega_i, i=1\ldots f$, then

$$w=\begin{cases}1 & \text{if } L=1\\ \left|D_{L-1}(\omega_i)\right|^2 & \text{if } L>1\end{cases} \tag{5.20}$$

where L is the iteration number, and $D_{L-1}(\omega_i)$ is the denominator found in the previous iteration evaluated at frequency ω_i. This means that the first iteration will provide the same result obtained with function *levy* and that further iterations will try to improve that model.

Syntax

```
[G, J, handle] = sanko (w, gain, phase, Q, n, m, toler, NMaxIters)
```

Arguments

❖ *w* — vector with frequencies in rad/s where the response of the plant is known.
❖ *gain* — vector with gains in dB at frequencies found in *w*.
❖ *phase* — vector with the phases in degrees at frequencies found in *w*.
❖ *Q* — the maximum common multiple of powers of *s*, as seen in (5.4).
❖ *n* — desired number of poles of the model.
❖ *m* — desired number of zeros of the model.
❖ *toler* (optional) — tolerance for comparison of consecutive iterations. The function stops after two consecutive iterations resulting in vectors of parameters with norms closer than *toler*. This parameter's default value is $10^{-3-\log(n+m)}$.
❖ *NMaxIters* (optional) — maximum number of iterations; its default value is 100.

Return values

❖ *G* — structure with two fields, *num* and *den*, containing the coefficients of (5.4) (beginning with the highest order ones).

❖ *J* — index showing how accurate the identified model is, given by

$$J = \frac{1}{n_\omega} \sum_{i=1}^{n_\omega} \left| G(j\omega) - \hat{G}(j\omega) \right|^2 \tag{5.21}$$

where n_ω is the number of frequencies in parameter *w* and *G* is the plant that provided the frequency behaviour.

❖ *handle* — handle to a figure with a Bode plot, at frequencies *w*, of the provided data (shown with blue dots) and the performance of the identified model (shown as a continuous green line). If this output parameter does not exist, the figure is not drawn.

Algorithm

This function's code is similar to that of function *vinagre*, save for the *while* cycle that implements the iterations, and that stops by means of a *break* command.

References

See (Sanathanan *et al.*, 1963), (Valério, p. 93-94, 2005c).

5.5. Function *lawro*

This function finds a model for a plant given its frequency behaviour, using Lawrence's and Roger's recursive variation of the identification method of function *levy*. This variation allows varying the number of frequencies accounted for in each iteration.

We define

$$v = \begin{bmatrix} b_0 & \cdots & b_m & a_0 & \cdots & a_n \end{bmatrix}^T \tag{5.22}$$

$$u = \begin{bmatrix} 1 & (j\omega)^q & \cdots & (j\omega)^{mQ} & -(j\omega)^Q G(j\omega) & \cdots & -(j\omega)^{nQ} G(j\omega) \end{bmatrix}^T \tag{5.23}$$

where $G(j\omega)$ is the frequency response at frequency ω. There will be several vectors *u*, one for each frequency. It may be proved that we may find consecutive approximations of *v* with

$$v_L = v_{L-1} + \mathbf{H}_L w_L^2 \left[u_L \left(\overline{G_L} - \overline{u_L^T} v_{L-1} \right) + \overline{u_L} \left(G_L - u_L^T v_{L-1} \right) \right] \tag{5.24}$$

$$\mathbf{H}_L = \mathbf{Z}_L \left(\mathbf{I} - \frac{\operatorname{Im}[u_L] \operatorname{Im}[u_L^T] \mathbf{Z}_L}{\frac{1}{2w_L^2} + \operatorname{Im}[u_L^T] \mathbf{Z}_L \operatorname{Im}[u_L]} \right) \tag{5.25}$$

$$\mathbf{Z}_L = \mathbf{H}_L \left(\mathbf{I} - \frac{\operatorname{Re}[u_L] \operatorname{Re}[u_L^T] \mathbf{H}_{L-1}}{\frac{1}{2w_L^2} + \operatorname{Re}[u_L^T] \mathbf{H}_{L-1} \operatorname{Re}[u_L]} \right) \tag{5.26}$$

where *L* is the iteration number. Initial values of matrix **H** and vector *v* may be obtained with values for a few frequencies:

$$\mathbf{H}_L^{-1} = \sum_{k=1}^{L} w_k^2 \left(u_k \overline{u_k^T} + \overline{u_k} u_k^T \right) \tag{5.27}$$

$$v_L = \mathbf{H}_L \sum_{k=1}^{f} w_k^2 \left(G_k \overline{u_k} + \overline{G_k} u_k \right) \tag{5.28}$$

or by letting

$$v_L = 0 \tag{5.29}$$

$$\mathbf{H}_L = \mathbf{I} \times x \tag{5.30}$$

where *x* is some suitable real value.

This function implements this algorithm, taking into account in each iteration one frequency more than in the previous. Initial values of matrix **H** and vector *v* may be provided. So the function is useful for improving previous estimations with new data and for avoiding numerical problems associated with inversion of matrixes.

Syntax

```
[G, J, handle] = lawro (w, gain, phase, Q, n, m, toler, H,...
    coefficients)
```

Arguments

❖ *w* — vector with frequencies in rad/s where the response of the plant is known.
❖ *gain* — vector with gains in dB at frequencies found in *w*.
❖ *phase* — vector with the phases in degrees at frequencies found in *w*.
❖ *Q* — the maximum common multiple of powers of *s*, as seen in (5.4).
❖ *n* — desired number of poles of the model.
❖ *m* — desired number of zeros of the model.
❖ *H* (optional) — initial value for matrix **H**, provided by (5.27). The default value is the identity matrix (that is, (5.30) with $x = 1$).
❖ *coefficients* (optional) — initial value for vector *v*, provided by (5.28). The default value is (5.29).

Return values

❖ *G* — structure with two fields, *num* and *den*, containing the coefficients of (5.4) (beginning with the highest order ones).
❖ *J* — index showing how accurate the identified model is, given by

$$J = \frac{1}{n_\omega} \sum_{i=1}^{n_\omega} \left| G(j\omega) - \hat{G}(j\omega) \right|^2 \tag{5.31}$$

where n_ω is the number of frequencies in parameter *w* and *G* is the plant that provided the frequency behaviour.

❖ *handle* — handle to a figure with a Bode plot, at frequencies *w*, of the provided data (shown with blue dots) and the performance of the identified model (shown as a continuous green line). If this output parameter does not exist, the figure is not drawn.

Limitations

This function has no provision for weights *w*, hence assumed to be unitary.

Algorithm

This function's code is a straightforward implementation of the above algorithm.

References

See (Lawrence *et al.*, 1979), (Valério, p. 94-99, 2005c).

6. Analysis and norms

Functions described in this section provide tools to analyse the behaviour of fractional plants.

6.1. Function *freqrespFr*

This function finds the frequency response of a fractional transfer function that may have one of the following forms:

$$F(s)=\frac{b_m s^{mQ}+b_{m-1}s^{(m-1)Q}+b_{m-2}s^{(m-2)Q}+\ldots+b_1 s^Q+b_0}{a_n s^{nQ}+a_{n-1}s^{(n-1)Q}+a_{n-2}s^{(n-2)Q}+\ldots+a_1 s^Q+1}e^{-\delta s} \tag{6.1}$$

$$F(s)=\left(k_P+\frac{k_I}{s^{\nu_I}}+k_D s^{\nu_D}\right)e^{-\delta s} \tag{6.2}$$

It is a generalisation of function *freqresp*.

Syntax

```
resp = freqrespFr(F, Q, w, delay)
```

Parameters

❖ *F* — parameters of (6.1), given by means of an LTI object equivalent to

$$F(s)=\frac{b_m s^m+b_{m-1}s^{m-1}+b_{m-2}s^{m-2}+\ldots+b_1 s+b_0}{a_n s^n+a_{n-1}s^{n-1}+a_{n-2}s^{n-2}+\ldots+a_1 s+1} \tag{6.3}$$

or of (6.2), given in a vector

$$\begin{bmatrix} k_P & k_I & \nu_I & k_D & \nu_D \end{bmatrix} \tag{6.4}$$

❖ *Q* (optional) — commensurate order of (6.1). The default value is 1. This parameter is irrelevant if *F* is given by a vector such as (6.4).
❖ *w* — vector with the frequencies at which the response of the plant is desired.
❖ *delay* (optional) — delay δ, as seen in (6.1) or in (6.2). The default value is 0.

Return values

❖ *resp* — complex-valued vector, with the dimension of *w*, with the frequency response of the plant at each frequency.

Algorithm

The frequency responses of (6.1) and (6.2) at frequency ω are found replacing *s* with $j\omega$.

After providing default values for non-existing parameters, the function checks the length of *F*: if it is 5, it is assumed to be of the form (6.4); if it is not 5, it is assumed to be of the form (6.3). This does not ensure that there will be no errors due to improperly provided parameters, but no further error-checking was deemed necessary. In the second case, only the numerator and denominator of the first object in *F* are considered (since MIMO plants are not covered); these are swept with *for* cycles.

The effects of the delay are added at the end.

6.2. Function *bodeFr*

This function plots the Bode diagram of a fractional transfer function that may have one of the following forms:

$$F(s)=\frac{b_m s^{mQ}+b_{m-1}s^{(m-1)Q}+b_{m-2}s^{(m-2)Q}+\ldots+b_1 s^Q+b_0}{a_n s^{nQ}+a_{n-1}s^{(n-1)Q}+a_{n-2}s^{(n-2)Q}+\ldots+a_1 s^Q+1}e^{-\delta s} \tag{6.5}$$

$$F(s)=\left(k_P+\frac{k_I}{s^{\nu_I}}+k_D s^{\nu_D}\right)e^{-\delta s} \tag{6.6}$$

It may also provide the data thereof. It is a generalisation of function *bode*.

Syntax

```
[gain, phase, w] = bodeFr(F, Q, w, delay)
```

Parameters

❖ *F* — parameters of (6.5), given by means of an LTI object equivalent to

$$F(s)=\frac{b_m s^m+b_{m-1}s^{m-1}+b_{m-2}s^{m-2}+\ldots+b_1 s+b_0}{a_n s^n+a_{n-1}s^{n-1}+a_{n-2}s^{n-2}+\ldots+a_1 s+1} \tag{6.7}$$

or of (6.6), given in a vector

$$\begin{bmatrix}k_P & k_I & \nu_I & k_D & \nu_D\end{bmatrix} \tag{6.8}$$

❖ *Q* (optional) — commensurate order of (6.5). The default value is 1. This parameter is irrelevant if *F* is given by a vector such as (6.8).

❖ *w* (optional) — vector with the frequencies at which the Bode diagram will be plot, or cell with the limits of the frequency range where the Bode diagram will be plot. In this last case, 10 frequencies per decade (with a minimum of 50) will be used. If empty, a range expected to be suitable is provided.

❖ *delay* (optional) — delay δ, as seen in (6.5) or in (6.6). The default value is 0.

Return values

❖ *gain* — gain of the plant, in absolute value, at frequencies in *w*.

❖ *phase* — phase of the plant, in degrees, at frequencies in *w*.

❖ *w* — frequencies, in rad/s, at which the gain and phase of the plant are those returned.

If there are no return arguments, a Bode diagram will be plot in the active figure.

Algorithm

If *w* is not provided, zeros and poles of the (integer) plant in *F* are stored in variable *temp*. They are found with functions *tzero* and *pole*. If this fails (which is checked with a *try* structure), the formula for the roots of an integer PID is used. The limits of the frequency range are set two decades below and above the frequency range where zeros and poles are found; only powers of 10 are employed. If there are zeros, the frequency range consists of four decades around 1 rad/s.

A frequency range is converted into a list of frequencies using ten frequencies per decade.

The frequency response is used by means of function *freqrespFr*.

6.3. Function *nyquistFr*

This function plots the Nyquist diagram of a fractional transfer function that may have one of the following forms:

$$F(s)=\frac{b_m s^{mQ}+b_{m-1}s^{(m-1)Q}+b_{m-2}s^{(m-2)Q}+\ldots+b_1 s^Q+b_0}{a_n s^{nQ}+a_{n-1}s^{(n-1)Q}+a_{n-2}s^{(n-2)Q}+\ldots+a_1 s^Q+1}e^{-\delta s} \tag{6.9}$$

$$F(s)=\left(k_P+\frac{k_I}{s^{\nu_I}}+k_D s^{\nu_D}\right)e^{-\delta s} \tag{6.10}$$

It may also provide the data thereof. It is a generalisation of function *nyquist*.

Syntax

```
[gain, phase, w] = nyquistFr(F, Q, w, delay)
```

Parameters

❖ *F* — parameters of (6.9), given by means of an LTI object equivalent to

$$F(s)=\frac{b_m s^{m}+b_{m-1}s^{m-1}+b_{m-2}s^{m-2}+\ldots+b_1 s+b_0}{a_n s^{n}+a_{n-1}s^{n-1}+a_{n-2}s^{n-2}+\ldots+a_1 s+1} \tag{6.11}$$

or of (6.6), given in a vector

$$\begin{bmatrix} k_P & k_I & \nu_I & k_D & \nu_D \end{bmatrix} \tag{6.12}$$

❖ *Q* (optional) — commensurate order of (6.9). The default value is 1. This parameter is irrelevant if *F* is given by a vector such as (6.12).

❖ *w* (optional) — vector with the frequencies at which the Nyquist diagram will be plot, or cell with the limits of the frequency range where the Nyquist diagram will be plot. In this last case, 10 frequencies per decade (with a minimum of 50) will be used. If empty, a range expected to be suitable is provided.

❖ *delay* (optional) — delay δ, as seen in (6.9) or in (6.10). The default value is 0.

Return values

❖ *gain* — gain of the plant, in absolute value, at frequencies in *w*.
❖ *phase* — phase of the plant, in degrees, at frequencies in *w*.
❖ *w* — frequencies, in rad/s, at which the gain and phase of the plant are those returned.

If there are no return arguments, a Nyquist diagram will be plot in the active figure.

Algorithm

This function is similar to function *bodeFr* above.

6.4. Function *nicholsFr*

This function plots the Nichols diagram of a fractional transfer function that may have one of the following forms:

$$F(s)=\frac{b_m s^{mQ}+b_{m-1}s^{(m-1)Q}+b_{m-2}s^{(m-2)Q}+\ldots+b_1 s^Q+b_0}{a_n s^{nQ}+a_{n-1}s^{(n-1)Q}+a_{n-2}s^{(n-2)Q}+\ldots+a_1 s^Q+1}e^{-\delta s} \tag{6.13}$$

$$F(s)=\left(k_P+\frac{k_I}{s^{\nu_I}}+k_D s^{\nu_D}\right)e^{-\delta s} \tag{6.14}$$

It may also provide the data thereof. It is a generalisation of function *nichols*.

Syntax

```
[gain, phase, w] = nicholsFr(F, Q, w, delay)
```

Parameters

❖ *F* — parameters of (6.13), given by means of an LTI object equivalent to

$$F(s)=\frac{b_m s^m+b_{m-1}s^{m-1}+b_{m-2}s^{m-2}+\ldots+b_1 s+b_0}{a_n s^n+a_{n-1}s^{n-1}+a_{n-2}s^{n-2}+\ldots+a_1 s+1} \tag{6.15}$$

or of (6.14), given in a vector

$$\begin{bmatrix} k_P & k_I & \nu_I & k_D & \nu_D \end{bmatrix} \tag{6.16}$$

❖ *Q* (optional) — commensurate order of (6.13). The default value is 1. This parameter is irrelevant if *F* is given by a vector such as (6.16).
❖ *w* (optional) — vector with the frequencies at which the Nichols diagram will be plot, or cell with the limits of the frequency range where the Nichols diagram will be plot. In this last case, 10 frequencies per decade (with a minimum of 50) will be used. If empty, a range expected to be suitable is provided.

❖ *delay* (optional) — delay δ, as seen in (6.13) or in (6.14). The default value is 0.

Return values

❖ *gain* — gain of the plant, in absolute value, at frequencies in *w*.
❖ *phase* — phase of the plant, in degrees, at frequencies in *w*.
❖ *w* — frequencies, in rad/s, at which the gain and phase of the plant are those returned.

If there are no return arguments, a Nichols diagram will be plot in the active figure.

Algorithm

This function is similar to function *bodeFr* above.

6.5. Function *sigmaFr*

This function plots the singular values diagram of a fractional transfer function matrix, with entries given by

$$F_{ij}(s)=\frac{b_m s^{mQ}+b_{m-1}s^{(m-1)Q}+b_{m-2}s^{(m-2)Q}+\ldots+b_1 s^Q+b_0}{a_n s^{nQ}+a_{n-1}s^{(n-1)Q}+a_{n-2}s^{(n-2)Q}+\ldots+a_1 s^Q+1} \tag{6.17}$$

It may also provide the data thereof. It is a generalisation of function *sigma*.

Syntax

```
[sv, w] = sigmaFr(F, Q, w)
```

Parameters

❖ *F* — parameters of the transfer function matrix, given by means of an LTI object with entries given by

$$F_{ij}(s)=\frac{b_m s^m+b_{m-1}s^{m-1}+b_{m-2}s^{m-2}+\ldots+b_1 s+b_0}{a_n s^n+a_{n-1}s^{n-1}+a_{n-2}s^{n-2}+\ldots+a_1 s+1} \tag{6.18}$$

❖ *Q* (optional) — commensurate order of (6.17). The default value is 1.
❖ *w* (optional) — vector with the frequencies at which the singular value diagram will be plot, or cell with the limits of the frequency range where the singular value diagram will be plot. In this last case, 10 frequencies per decade (with a minimum of 50) will be used.

Return values

❖ *sv* — singular values of the plant, in absolute value. Each frequency in *w* corresponds to a column in *sv*.
❖ *w* — frequencies, in rad/s, at which the gain and phase of the plant are those

returned.

If there are no return arguments, a singular value diagram will be plot in the active figure.

Algorithm

Frequencies are swept with a *for* cycle. For each, the frequency response is stored in variable *tempMatrix*. For that purpose, columns and rows are swept with *for* cycles (and since both *i* and *j* are used as dummy variables, the imaginary unit has to be used as *sqrt*(–1)) and the response is found just as in *freqrespFr* above. Singular values are found with function *svd*.

6.6. Function *normh2Fr*

This function returns the H_2 norm of a transfer function matrix. The H_2 norm of one such matrix **G** is defined as

$$\left\|\mathbf{G}(s)\right\|_2 = \sqrt{\frac{1}{2\pi}\int_{-\infty}^{+\infty} \mathrm{tr}\left[\mathbf{G}(j\omega)\overline{\mathbf{G}(j\omega)}^T\right] d\omega} \tag{6.19}$$

In simple words, the H_2 norm of a transfer function reflects how much it amplifies (or attenuates) its input over all frequencies.

For a MIMO system with *n* lines and *m* columns,

$$\left\|\mathbf{G}\right\|_2 = \sqrt{\sum_{j=1}^{n}\sum_{i=1}^{m}\left\|\mathbf{G}_{ij}\right\|_2^2} \tag{6.20}$$

This reduces the problem to finding the H_2 norm of SISO systems.

H_2 norm of integer systems

This may be found using function *normh2*.

H_2 norm of fractional systems

Let *Q* be the commensurate order of

$$G(s) = \frac{b_m s^{mQ} + b_{m-1}s^{(m-1)Q} + b_{m-2}s^{(m-2)Q} + \ldots + b_1 s^Q + b_0}{a_n s^{nQ} + a_{n-1}s^{(n-1)Q} + a_{n-2}s^{(n-2)Q} + \ldots + a_1 s^Q + 1} \tag{6.21}$$

and let

$$\left.\begin{array}{l} q = \mathrm{C}\ (Q) \\ p = Q - q \end{array}\right\} \Rightarrow p + q = Q \tag{6.22}$$

$$x = \omega^{1/Q} \Leftrightarrow \omega = x^Q \Rightarrow d\omega = Qx^{Q-1}dx \tag{6.23}$$

Since the complex conjugate may be obtained changing the sign of the imaginary part, we will have

$$\|G\|_2^2 = \frac{1}{\pi}\int_0^{+\infty} Qx^{Q-1}G(jx)G(-jx)\,dx \tag{6.24}$$

Let *A* and *B* be the polynomials in the numerator and denominator of product

$$G(jx)G(-jx) = \frac{A(x)}{B(x)} \tag{6.25}$$

Notice that the imaginary unit *j* has been considered as part of polynomials *A* and *B*. If we now let

$$\frac{x^q A(x)}{B(x)} = \frac{R(x)}{B(x)} + \sum_{k=0}^{q+\deg(A)-\deg(B)} a_k x^k \tag{6.26}$$

(where deg(*P*) represents the degree of polynomial *P*), then (6.24) becomes

$$\|G\|_2^2 = \frac{Q}{\pi}\int_0^{+\infty} x^{p-1}x^q \frac{A(x)}{B(x)}dx = \frac{Q}{\pi}\int_0^{+\infty} x^{p-1}\left(\frac{R(x)}{B(x)} + \sum_{k=0}^{q+\deg(A)-\deg(B)} a_k x^k\right)dx \tag{6.27}$$

Three cases are to be distinguished when reckoning (6.27).

First case

If

$$q + \deg(A) - \deg(B) > 0 \tag{6.28}$$

then

$$\|G\|_2 = \infty \tag{6.29}$$

Second case

If

$$q + \deg(A) - \deg(B) \le 0 \wedge p \ne 0 \tag{6.30}$$

let $B(x)$ have *b* different poles, $s_1, s_2, \ldots s_b$, and let m_k be the multiplicity of pole s_k. Then we may perform a partial fraction expansion of

$$\frac{x^q A(x)}{B(x)} = \sum_{k=1}^{b}\sum_{n=1}^{m_k} \frac{a_{k,n}}{(x+s_k)^n} \tag{6.31}$$

(Recall that a pole of multiplicity *m* will appear *m* times in the expansion.) Then (6.27) becomes

$$\|G\|_2^2 = \sum_{k=1}^{b} \sum_{n=1}^{m_k} \frac{(-1)^{n-1} Q a_k s_k^{p-n} \binom{p-1}{n-1}}{\sin(p\pi)} \tag{6.32}$$

Third case

If

$$q + \deg(A) - \deg(B) \leq 0 \wedge p = 0 \tag{6.33}$$

let s_1 be one of the poles of (6.31), chosen arbitrarily[7]. Then we may write

$$\frac{x^{q-1}A(x)}{B(x)} = \sum_{k=2}^{b} \frac{c_k}{(x+s_1)(x+s_k)} + \sum_{k=1}^{b} \sum_{n=2}^{m_k} \frac{d_{k,n}}{(x+s_k)^n} \tag{6.34}$$

Notice that poles with multiplicity 1 do not appear in the second summation. Expression (6.27) becomes

$$\|G\|_2^2 = \sum_{k=2}^{b} \left[\frac{Qc_k}{\pi(s_k - s_1)} \ln \frac{s_k}{s_1} \right] + \sum_{k=1}^{b} \sum_{n=2}^{m_k} \frac{Qd_{k,n} s_k^{1-n}}{\pi(-n+1)} \tag{6.35}$$

It should be noticed that, even though several different formulas are to be applied depending on the value of *Q*, the norm is a continuous function thereof.

Syntax

```
[H2, status] = normh2Fr (F, Q, problems)
```

Arguments

❖ *F* — parameters of the transfer function matrix, given by means of an LTI object with entries given by

$$F_{ij}(s) = \frac{b_m s^m + b_{m-1} s^{m-1} + b_{m-2} s^{m-2} + \ldots + b_1 s + b_0}{a_n s^n + a_{n-1} s^{n-1} + a_{n-2} s^{n-2} + \ldots + a_1 s + 1} \tag{6.36}$$

❖ *Q* (optional) — commensurate order of (6.17). The default value is 1.

❖ *problems* (optional) — this affects how problems are reported by means of errors and warnings:

- ♦ 'none' — no errors or warnings are given;
- ♦ 'warnings' (default) — warnings are given when problems are found;
- ♦ 'errors' — errors are given when problems are found.

Return values

❖ *H2* — H_2 norm of the plant.

[7] Actually the better choice would be the one minimising numerical errors, but it is difficult to know beforehand which one is.

❖ *status* — this is a vector with two positions that may be 0 or 1: it is [0 0] if there are no stability and no causality problems; the first position becomes 1 if the system is unstable; the second position becomes 1 if the system in not causal.

Error messages

If *problems* is set to ‘errors’, the following error messages may appear:

❖ *A non-sensical imaginary number was obtained.* — This happens because of numerical problems during calculations.

❖ *The system is not stable: the result is a nonsense.*

❖ *According to the references (see the manual) the result should be infinite.* — This happens when some finite numerical value is found instead of (6.29).

Warnings

If *problems* is set to ‘warnings’, warnings similar to the error messages above may appear. The following warning may always appear:

❖ *Partial fraction expansion originated significant numerical errors.*

Algorithm

The main function implements (6.20), sweeping lines and columns with *for* cycles, and relegates the problem of finding the norm of a SISO plant to function *H2FrSISO*. In the end the result is checked to see if it is real; this is done with *angle* so that neglectable imaginary parts may be done away with. If there is a significant imaginary part, the result is left as is, and an error or warning (as appropriate) is given since it is probably meaningless.

Function *H2FrSISO*

This function implements formulas given above. Expansions (6.31) and (6.34) are found with function *residue*. To handle the second summation in (6.32), related to multiplicities, poles are swept with a *for* cycle and checked with an *if* structure. Things are more complicated when implementing (6.35). Matrix *simple* is used for holding, in its second column, simple poles and the first appearance of repeated ones—in the first column it contains the residues corresponding to those poles. Matrix *repeated* is used for holding, in its second column, further appearances (from the second on) of repeated poles—in the first column it contains the residues corresponding to those poles. A *for* cycle then sweeps *repeated* and creates a third column that contains the number of the repetition (2, 3, 4, etc.). Pole s_1 in (6.34) and (6.35) is the first in *simple*. A numerator and a denominator are rebuilt from the poles and residues in *simple*, and *residue* must be applied again to find the residues in the first summation of (6.35); since this may cause significant numerical errors, a check is performed and a warning given if appropriate. The second summation in (6.35) is performed only if *residue* is not empty (otherwise an error would occur).

Function *squarefreq*

This function exists to avoid repeating code. It receives a frequency ω a vector with the parameters of a polynomial $p(s)$ and returns the (imaginary) value $p(j\omega)\overline{p(j\omega)}$.

References

See (Malti *et al.*, 2003), (Valério, p. 127-131, 2005c).

6.7. Function *normhinfFr*

This function returns the H_∞ norm of a transfer function matrix. The H_∞ norm of one such matrix **G**, with entries given by

$$G_{ij}\left(s\right)=\frac{b_m s^{mQ}+b_{m-1}s^{(m-1)Q}+b_{m-2}s^{(m-2)Q}+\ldots+b_1 s^Q+b_0}{a_n s^{nQ}+a_{n-1}s^{(n-1)Q}+a_{n-2}s^{(n-2)Q}+\ldots+a_1 s^Q+1} \tag{6.37}$$

is defined as

$$\left\|\mathbf{G}\right\|_\infty=\sup_\omega \max \sigma\left[\mathbf{G}\left(j\omega\right)\right] \tag{6.38}$$

where $\sigma\left(A\right)$ represents the set of singular values of matrix *A*. (This set has a finite number of values, and thus has a maximum; on the other hand, the set resulting of sweeping all frequencies may have no maximum, but only a supreme value.) If *G* is SISO, (6.38) becomes simply

$$\left\|G\right\|_\infty=\sup_\omega \max\left|G\left(j\omega\right)\right| \tag{6.39}$$

In simple words, the H_∞ norm reflects how much it amplifies (or attenuates) its input at the frequency at which the amplification is maximal.

In this function, (6.38) is estimated by direct evaluation at several frequencies. Frequencies clearly above or below all the frequencies of poles and zeros need not be searched. The result is, of course, equal to or below the exact result—it can never be above.

Syntax

```
Hinf = normhinfFr (F, Q, w)
```

Parameters

❖ *F* — parameters of the transfer function matrix, given by means of an LTI object with entries given by

$$F_{ij}\left(s\right)=\frac{b_m s^m+b_{m-1}s^{m-1}+b_{m-2}s^{m-2}+\ldots+b_1 s+b_0}{a_n s^n+a_{n-1}s^{n-1}+a_{n-2}s^{n-2}+\ldots+a_1 s+1} \tag{6.40}$$

❖ *Q* (optional) — commensurate order of (6.17). The default value is 1.

❖ *w* (optional) — vector with the frequencies at which singular values will be found, or cell with the limits of the frequency range where singular values will be found. In this last case, 10 frequencies per decade (with a minimum of 50) will be used.

Return values

❖ *Hinf* — H_∞ norm of the plant.

Algorithm

This function uses *sigmaFr* to find the singular values in (6.38), and *max* to sort them. Since *sigmaFr* provides no default value for *w*, but allows for a cell with the limits of a frequency range, one such range is provided in *normhinfFr* with code similar to that of *bodeFr* and similar functions.

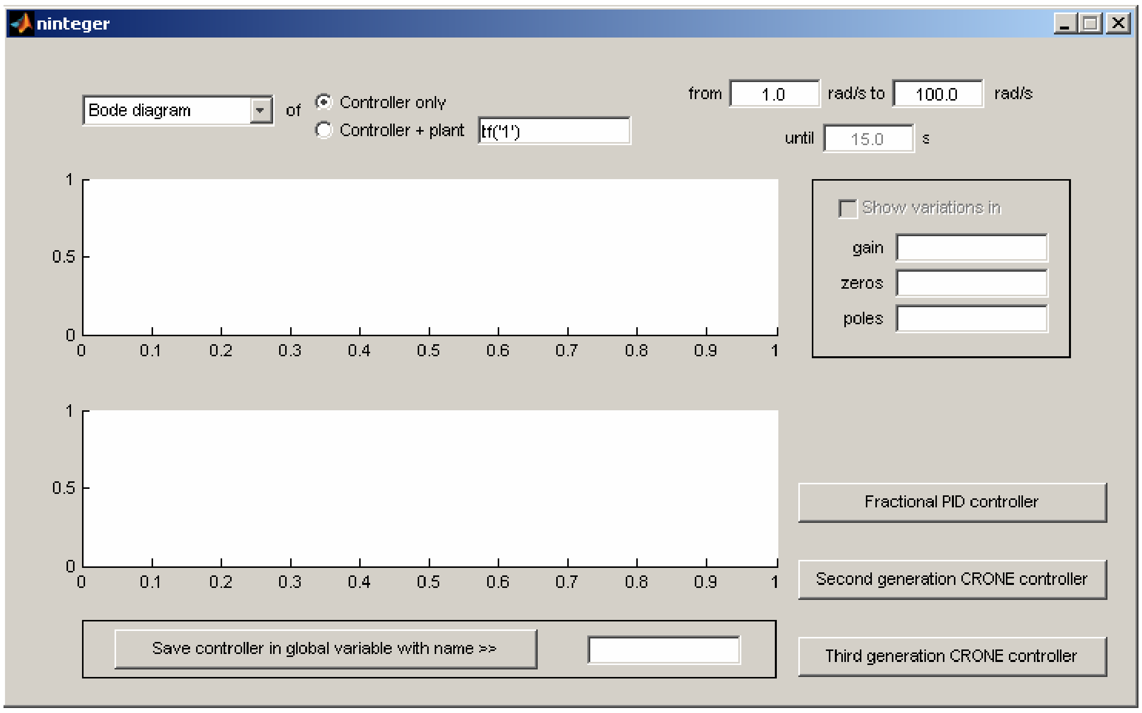
References

See (Petras *et al.*, 2002), (Valério, p. 131-132, 2005c).

7. Graphical interface

7.1. User manual

The graphical is accessible by tiping *ninteger* at the command prompt. The following dialogue will appear:


ninteger
Show variations in
gain
zeros
poles
Fractional PID controller
Second generation CRONE controller
Third generation CRONE controller
Save controller in global variable with name >>

Bode diagram of [✓ Controller only] Controller + plant tf('1')

from 1.0 rad/s to 100.0 rad/s

until 15.0 s

Figure 2. The main dialog of *ninteger* before being filled in.

Using the three buttons in the right other dialogues may be called for devising the controllers described in the previous sections. In what follows the foreknowledge of these is assumed.

After reckoning a controller this dialogue will be used for evaluating its performance, as will be seen below. It may be closed as any other window in your operating system.

Dialogue *nipidGui*

This dialogue allows developing fractional PIDs and is available pressing the button *Fractional order derivative controller* in the main window of *ninteger*. The type of approximation may be chosen with the two pop-up menus in the centre. The first pop-up menu has the options corresponding to parameter *formula* of function *nipid*. If a digital controller is chosen, the second pop-up menu will become active, allowing the user to choose parameter *expansion*. Controllers are always built using function *nipid*, save that function *newton* is used when option *Carlson* is chosen in the first pop-up menu and the necessary conditions are satisfied (these are given in section 2.4).

The fields of the dialogue allow entering the other parameters needed. Not all of them are necessary in all cases; only those that are allow a value to be entered. In particular:

❖ The field with the specification on the number of poles and zeros must always be filled in; it corresponds to parameter *n* in all cases.

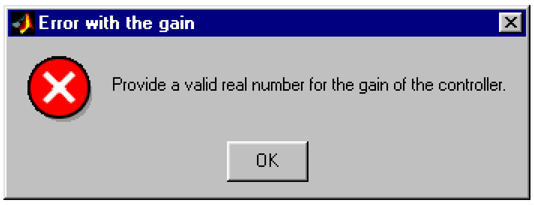
❖ The check-button allows choosing parameter *decomposition* ('all' if checked; 'frac' if not checked).

❖ Values for the gains (*kp*, *ki* and *kd*) and orders (*vi* and *vd*) must always be entered. If the option *Carlson* is chosen, a warning will appear stating that orders will be rounded to the nearest inverse of an integer (for instance, the value 0.3 will be rounded to 1/3).

❖ The fields in the sentence *from… rad/s to… rad/s* must be filled when a continuous controller is chosen; they correspond to parameters *wl* and *wh* respectively.

❖ The sampling time (*Ts*) must be filled when a digital controller is chosen.

Pressing the *OK* button calls the appropriate function. If an invalid number was entered in any field an error message will appear in this moment explaining as accurately as possible where the problem is.


Error with the gain

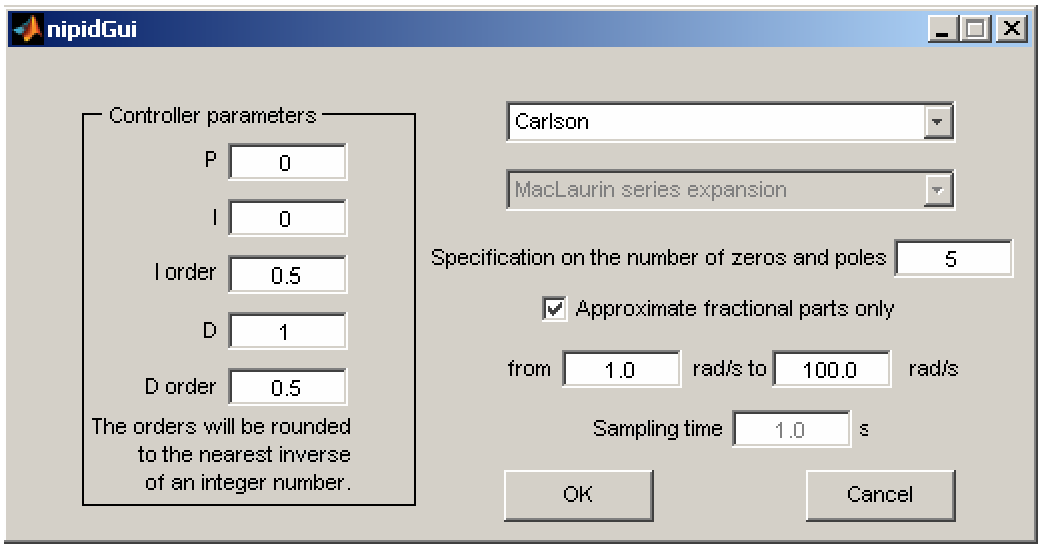
Provide a valid real number for the gain of the controller.

Figure 3. Example of an error message.

Button *OK* must be pressed before anything else can be done in MatLab. If the main dialogue was closed, this will also result in an error. In this case the message is more general; it runs «There was an error while building the controller.» This error may also appear if anything else went wrong (like overflows for instance) but the most usual reason is likely to be that the main dialogue was closed (which it should not have been).

Performance evaluation in the main dialogue

Let us suppose that a Carlson approximation of a fractional controller is devised in the *nipidGui* dialogue with the following parameters:


nipidGui

Controller parameters

P 0

I 0

I order 0.5

D 1

D order 0.5

The orders will be rounded to the nearest inverse of an integer number.

Carlson

MacLaurin series expansion

Specification on the number of zeros and poles 5

[✓] Approximate fractional parts only

from 1.0 rad/s to 100.0 rad/s

Sampling time 1.0 s

Figure 4. Dialogue *nipidGui* filled in.

Notice the warning shown, and that unnecessary fields are not active. The order of integration could, of course, have a different value.

When *OK* is pressed, and if nothing had been done before in the main dialogue, the Bode diagram of the controller from 1 to 100 rad/s will appear:

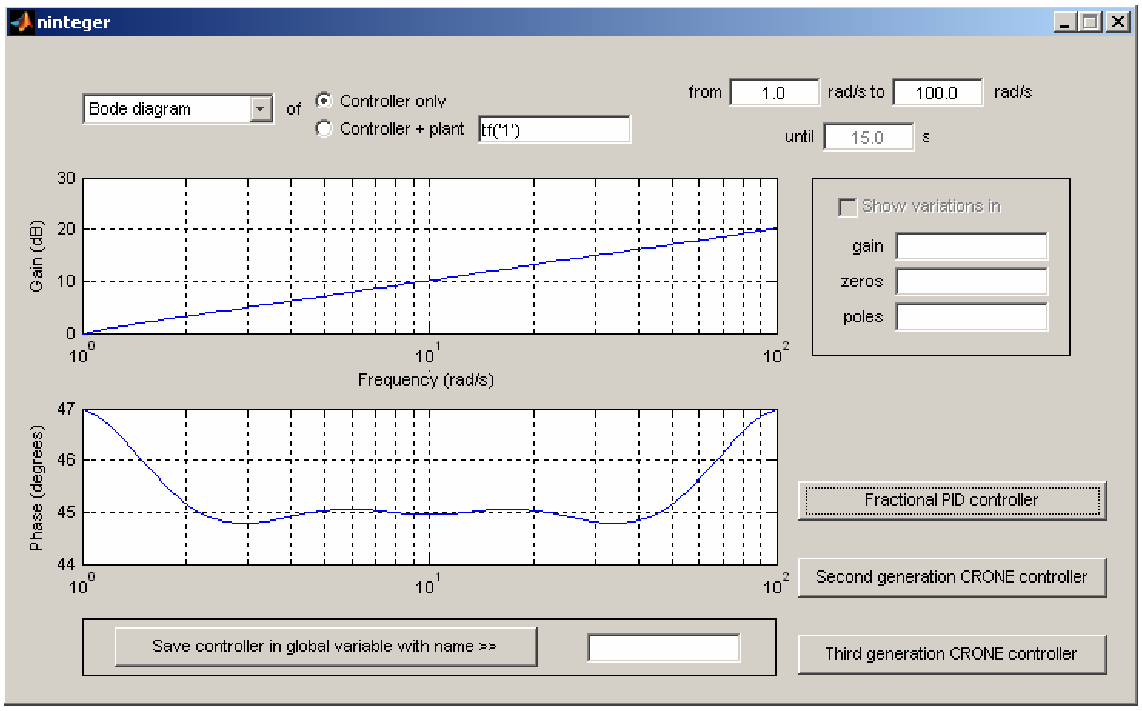


Figure 5. The main dialogue displaying a Bode diagram.

The main dialogue allows visualising six different characteristics of the controller:

- its Bode diagram;
- its Nichols diagram;
- its Nyquist diagram;
- its impulse response
- its step response;
- the placement of its poles and zeros.

The frequency behaviour (as shown in the Bode, Nichols and Nyquist diagrams) is sampled at fifty logarithmically spaced frequencies between the values entered in the fields of the sentence *from… rad/s to… rad/s*. Time responses are evaluated until the instant specified in the field of the sentence *until… s*; the number of sampling instants is chosen by MatLab's functions *impulse* and *step*. When poles and zeros are visualised, the first appear as red crosses and the latter as blue circles; an appropriate grid is drawn, and all the three above-mentioned fields are irrelevant.

Instead of viewing what happens with the controller it is also possible to view what happens to open-loop formed by the controller and by the plant for which is was designed. In that case the plant must be entered in the field next to the corresponding radio button. It must be an LTI system built with one of functions *tf*, *zpk* or *ss*. The expression may be entered in the field or the name of a variable in the workspace may be entered. For instance, introducing
```
tf([1, 1])
```
in the field or entering
```
plant = tf([1, 1])
```
in the command prompt and then

```
plant
```

in the field will have the same result. After filling the field in, the radio button labelled *Controller + plant* may be pressed.

If the field is not filled in or if what it is filled in with is not a suitable LTI plant, an error dialogue like the one of Figure 3 above will appear and the dialogue will revert to the situation in which only the controller (without plant) is considered. It is imperative that the plant be continuous if the controller is continuous and discrete if the controller is discrete; in the last case the sampling time must be same. If not, the error dialogue will appear. It will also appear if an open-loop is being (properly) visualised and then another controller with a different sampling time is devised. Only the new controller will be considered, since the extant plant is not compatible.

For saving the controller, introduce the name of the variable where you want it in the field at the bottom of the dialogue and press the button to its right. The variable will be global, so it is advisable that the name be not yet in use. An error dialogue like the one of Figure 3 above will appear if the name is not valid or if there is no controller to save.

Two final remarks. First, all this holds if the controller was devised not with the *nipidGui* dialogue but with some other. Second, it is possible to analyse the robustness of the controller by checking what happens if plant parameters undergo changes; but since such an analysis is usually performed in connection with third generation CRONE controllers, the issue will be covered after dealing with the dialogues for devising such controllers.

Dialogue *crone2Gui*

This dialogue allows developing controllers described above in chapter 3 and is available pressing the button *Second generation CRONE controller* in the main window of *ninteger*. It may be also used for identifying a plant from its frequency behaviour. Whatever the case, four things must be specified:

- ❖ how the phase behaviour must be;
- ❖ how the gain behaviour must be;
- ❖ at what frequencies phase (and eventually gain) behaviour is specified;
- ❖ what structure (namely number of zeros and poles) the controller must have.

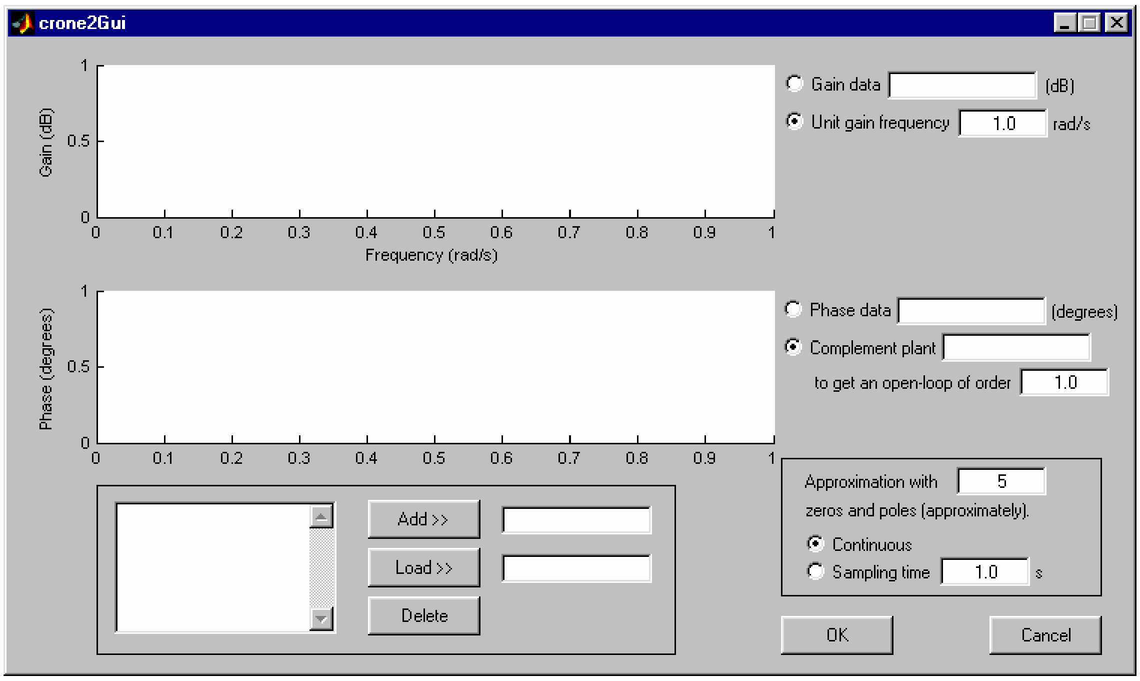


Figure 6. Dialogue *crone2Gui*.

The list of frequencies may be obtained from a workspace variable by typing its name next to button *Load* >> and pressing it. The variable must be a column vector of positive real numbers (no negative or complex frequencies exist). In alternative it is possible to edit the list by adding frequencies (this is done typing them after button *Add* >> and pressing it) or deleting them (this done selecting those to delete and pressing *Delete*). Whatever the case, if there is an error, an error message similar to that of Figure 3 is shown explaining what the problem is.

Using the dialogue to devise a controller

If the objective is to control a plant, the sentence *Complement plant… to get an open-loop of order…* must be completed with a plant and a real number. The plant must be an LTI system built with one of functions *tf*, *zpk* or *ss*. The expression may be entered in the field or the name of a variable in the workspace may be entered. The phase of the plant will be plotted in blue, together with a set of red points. These correspond to the phase the controller must have at the specified frequencies to meet the specification, that is to say, they correspond to parameter *phase*.

It is, however, possible to input directly the phase behaviour that the controller must have. The name of the variable (which must be a column vector of real numbers) must be entered in the field next to the corresponding radio button before pressing this, or an error will occur. Instead of the name of a variable, the declaration thereof may be given (this is of course not practical for long vectors).

Frequency *w1*, at which the controller is to have a unit gain (0 dB), must be given in the field next to the corresponding radio button. Parameter *n* is given in the box above the *OK* and *Cancel* buttons, together with the sampling time, if that is the case. Pressing *OK* will call *crone2* or *crone2z* to reckon the controller. Incorrect values will cause an error dialogue to appear.

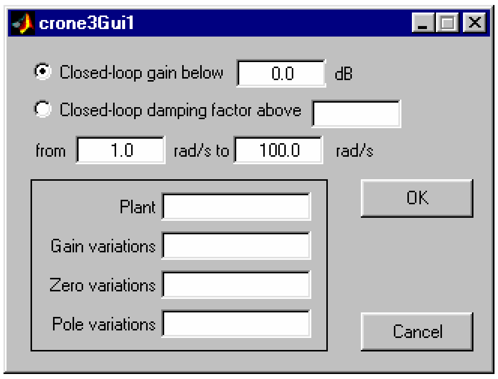
Using the dialogue to identify a plant

If the objective is to identify a plant, the lists of frequencies and phases are to be filled as explained above. The list of gains is given similarly: the field next to the corresponding radio button must be filled before pressing this.

It should be noticed that changing the list of frequencies after choosing to input phase and gain data will reset the radio buttons to their original values (complement a plant and specify a unit gain frequency), since provided data is likely to be useless with the new set of frequencies.

Dialogues *crone3Gui1* and *crone3Gui2*

Developing controllers described above in chapter 4 is a two-step task, carried out by these two dialogues, that are available pressing the button *Third generation CRONE controller* in the main window of *ninteger*. In the first dialogue the parameters of function *crone3* are given:


crone3Gui1

[✓ Closed-loop gain below] 0.0 dB
Closed-loop damping factor above
from 1.0 rad/s to 100.0 rad/s

Plant
Gain variations
Zero variations
Pole variations

Figure 7. Dialogue *crone3Gui1*.

❖ performance specification (*spec*) the closed-loop must verify (either a maximum closed-loop resonance gain or a minimum closed-loop damping coefficient);
❖ limits of the frequency range (*wl* and *wh*) in which the performance is evaluated;
❖ LTI object (*G*) with the plant to control (created with *tf*, *zpk*, or *ss*);
❖ variations that the gain, the zeros and the poles of the plant may undergo (*vargain*, *varzeros* and *varpoles*).

Names of workspace variables may be entered for these last four variables (if the plant has several poles and zeros it would not be expedient to introduce the corresponding matrixes in the fields). For details concerning the format of matrixes describing plant parameter variations see subsection 4.1.

When *OK* is pressed this data will be collected; eventual errors will lead to error dialogues like that of Figure 3 explaining what the problem is; and in the end a dialogue will appear as seen below in Figure 9.

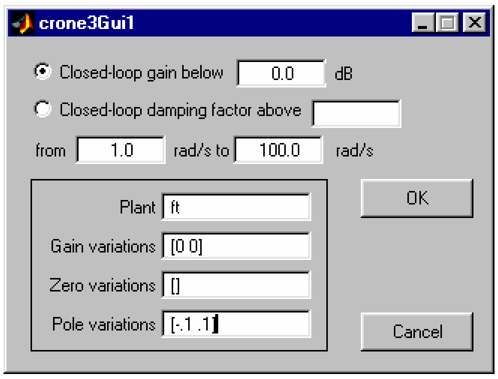
The leftmost list-box contains fifty frequencies logarithmically spaced between *wl* and *wh* (as entered in the previous plot). The corresponding frequency behaviour is shown plotted in a Bode diagram, but it is possible to see the Nichols or Nyquist diagrams instead. Fifty frequencies are surely too many for identifying a system with the desired behaviour, and so it is now necessary to choose among these fifty some few that will be used when the controller is synthesised. This is done by means of the two buttons *Add >>* and *Delete*; chosen frequencies are shown in the rightmost list-box, and the corresponding points in the plot appear in red. The dialogue allows specifying the value of *n* too. When *OK* is pressed, *crone2* or *crone2z* (depending on whether the plant given in the previous dialogue is continuous or discrete; in the latter case, the sampling time will be inherited by the controller) are used for reckoning the controller.

Robustness evaluation in the main dialogue

Let us suppose that the command

```
ft = tf([1, 1])
```

was given at the command prompt and that the two dialogues were filled in as follows:



crone3Gui1

[✓ Closed-loop gain below] 0.0 dB

Closed-loop damping factor above

from 1.0 rad/s to 100.0 rad/s

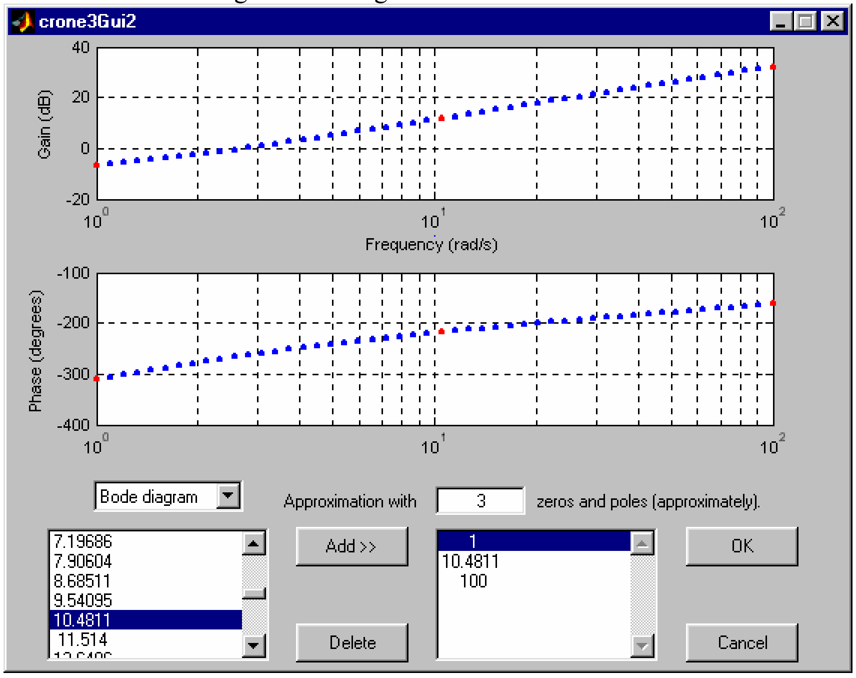
Plant: ft

Gain variations: [0 0]

Zero variations: []

Pole variations: [-.1 .1]

Figure 8. Dialogue *crone3Gui1* filled in.



crone3Gui2

Bode diagram

Approximation with 3 zeros and poles (approximately).

```
7.19686
7.90604
8.68511
9.54095
10.4811
11.514
```

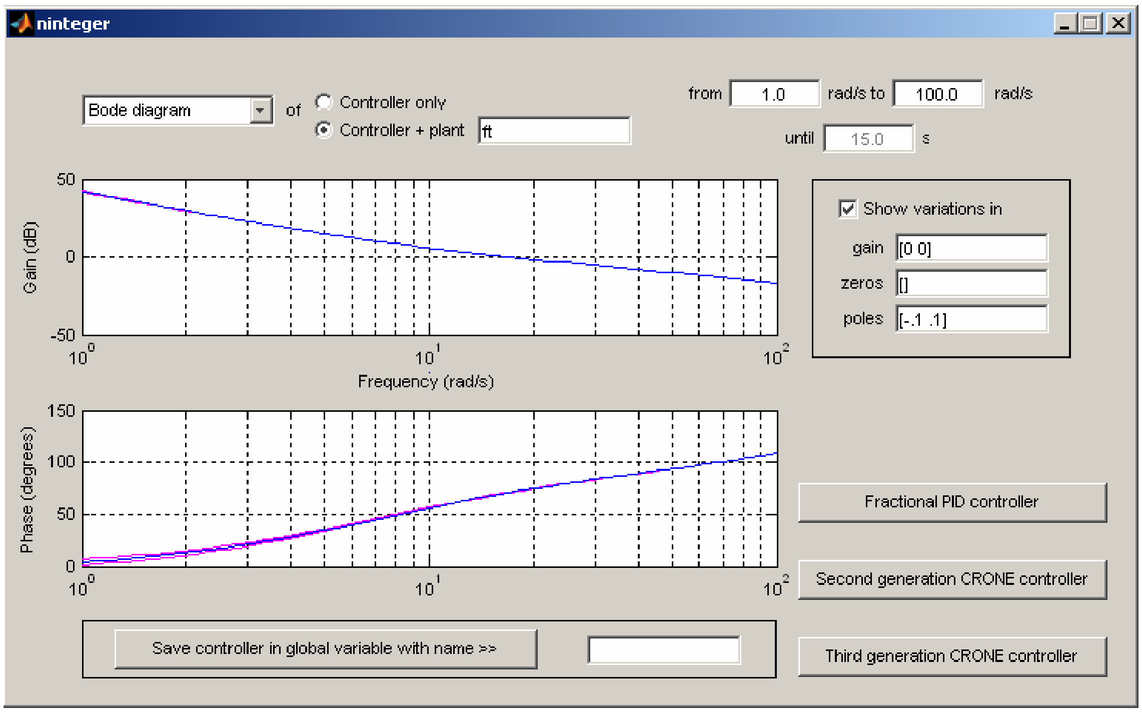
```
1
10.4811
100
```

Figure 9. Dialogue *crone3Gui2* filled in.

The main dialogue *ninteger* would appear with the plots seen in Figure 10, above. The plant and its possible variations of gain, zeros and poles are those previously introduced. The nominal behaviour appears in blue; all the other plots, corresponding to the possible variations of the plant, appear in magenta. By changing the plot to a Nichols plot it becomes clear that the specification was met[8], as seen in Figure 10, below. If placement of poles and zeros is chosen, nominal poles appear in red and their variations are shown in green, while nominal zeros appear in blue and their variations are shown in magenta. It is possible to change the ranges of variation of the plant parameters. If errors are found, an error dialogue similar to that of Figure 3 is shown explaining where the error is, and the plots are redrawn without taking parameter

[8] The plots are actually farther from the 0 dB closed-loop gain curve than what would be necessary. This is not out of unnecessary care from function *crone3*; it was consequence of errors introduced by *crone2* when approximating the behaviour prescribed by *crone3*.

variations into account (and the corresponding checkbox is accordingly unchecked).



ninteger
Bode diagram
of
Controller only
Controller + plant
Frequency (rad/s)
Gain (dB)
Phase (degrees)
Fractional PID controller
Second generation CRONE controller
Third generation CRONE controller
Save controller in global variable with name >>

[Bode diagram ▾] of ( ) Controller only [● Controller + plant] ft

from 1.0 rad/s to 100.0 rad/s

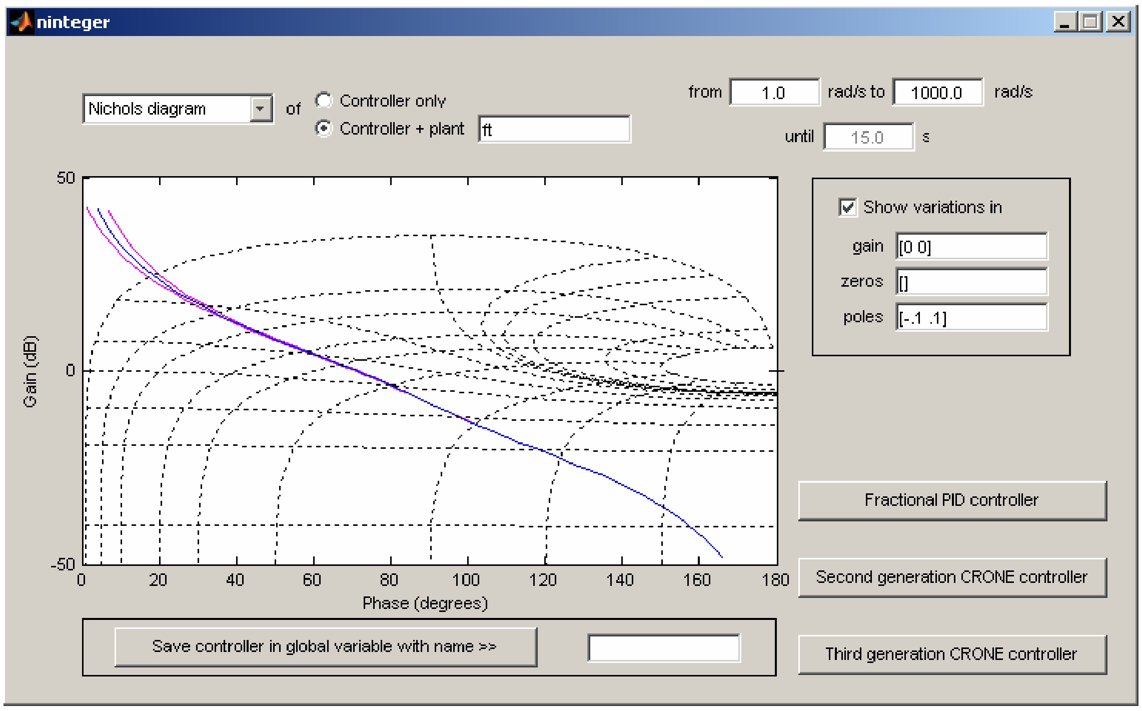
until 15.0 s

[✓] Show variations in

gain [0 0]

zeros []

poles [-.1 .1]



ninteger
Nichols diagram
of
Controller only
Controller + plant
Phase (degrees)
Gain (dB)
Fractional PID controller
Second generation CRONE controller
Third generation CRONE controller
Save controller in global variable with name >>

[Nichols diagram ▾] of ( ) Controller only [● Controller + plant] ft

from 1.0 rad/s to 1000.0 rad/s

until 15.0 s

[✓] Show variations in

gain [0 0]

zeros []

poles [-.1 .1]

Figure 10. Two views of resulting dialogue *ninteger*.

Robustness evaluation is always available whenever a controller and a plant are given (if the behaviour visualised is that of the controller without a plant, no plant parameter variations are obviously admitted).

7.2. Programmer manual

All the files pertaining to the graphical interface are inside the *gui* folder.
All dialogues are non-resizable, allow only one instance and have the command line accessibility set to *Callback*.

Function *isinteger*

This function checks if a number is integer or not. The need of such of a function was felt to increase the legibility of the code.
The syntax of this function is

```
output = isinteger(a)
```

Output will be 1 if *a* is integer or 0 if it is not. The result is obtained comparing *a* with its closest integer as returned by *round*. This means that the function also deals with matrixes: a matrix, of the same size of *a*, will be returned, having in each position the result for the corresponding element of *a*. It also means the case of a complex *a* is also dealt with: since *round* rounds both the real and imaginary parts, 1 will be returned only if both are integer.

Dialogue *ninteger*

Tags of objects in this dialogue are as follows:

Pop-up menu	popupmenu1
Radio button labelled *Controller only*	radiobutton1
Radio button labelled *Controller + plant*	radiobutton2
Field to the right of the above	edit_plant
Field for introducing frequency *wl*	edit_wl
Field for introducing frequency *wh*	edit_wh
Field for introducing the final time	edit_tf
Gain plot of the Bode diagram	plot1
Phase plot of the Bode diagram	plot2
Plot for all other diagrams and responses	plot
Button labelled Save controller in global…	pushbutton_Save
Field to the right of the above	edit_name
Checkbox	checkbox1
Field for introducing matrix *vargain*	edit_vargain
Field for introducing matrix *varzeros*	edit_varzeros
Field for introducing matrix *varpoles*	edit_varpoles
Button labelled Fractional PID controller	pushbuttonnipid
Button labelled Second generation CRONE controller	pushbutton2
Button labelled Third generation controller	pushbutton3

Opening function of *ninteger*

The opening function of this dialogue defines a default controller, a default plant and a default open-loop. These are stored as fields of the *handles* structures, which is the easiest and more elegant way of letting them be accessible to the whole dialogue; their names are respectively *C*, *plant* and *open_loop*. Since nothing exists in the beginning, they are both set to a null transfer function. A separate variable for the open-loop transfer function exist so that what should be plotted is always known.

Callback of radio button *radiobutton1*

If this button is hit the first task is to get its state, which is stored into variable *button_state*. If the button is now on, the other will be turned off. This is possible since there are but two; should there be more, the situation in which none is on would have to be dealt with differently. Variable *plant* is set to naught (since none is to be used), *open_loop* is set to *C* (as though the plant were unitary, since it is always the open-loop that is plotted), the checkbox and associated field become inactive (if there is no plant, there may be no plant parameter variations), and the plot is redrawn. Since the plot has to be redrawn in a variety of situations, the repetition of code is avoided using a function called *redraw*.

If the button is now off, the other will be turned on. This now requires a *try… catch* structure since it is easy that things go astray. The plant is obtained evaluating its string. The evaluation is carried out in the workspace using *evalin* to allow for variables in the workspace to be used. Variable *open_loop* is set to be equal to *C* multiplied by the plant. These commands come first since, should an error occur, what follows (enabling the checkbox and the associated fields) is not necessary. Then the plot is redrawn.

If this failed, the *catch* turns this button on again while turning the other off, turns off the *Enable* property of the checkbox and associated fields (in principle they were never enabled, but this is playing safe), and an appropriate error dialogue[9] appears.

Callback of radio button *radiobutton2*

The code of this callback is rather similar to that of radio button *radiobutton1*, save that the order is different.

Callback of field *edit_plant*

This callback also bears resemblances to that of radio button *radiobutton1*, but nothing is done if radio button *radiobutton2* is not on. If it is, then the open-loop is changed and the plot redrawn using *redraw*. If an error occurs, this means that was provided as plant is not a valid LTI plant. So buttons and fields are reset to the situation of controller only; the plant is set to naught and the open-loop to *C*; the plot is redrawn; and an error dialogue is shown.

Callback of fields *edit_wl*, *edit_wh* and *edit_tf*, of pop-up menu *popupmenu1*, and of checkbox *checkbox1*

If any of these fields is changed, or if a new type of plot is asked for, or if plant parameter variations are to be considered or not, then the plot is redrawn using function *redraw*.

Callback of fields *edit_vargain*, *edit_varzeros* and *edit_varpoles*

These callbacks limit themselves to redraw the plots using *redraw* as well, but only if their content is to be taken into account.

Function *redraw*

This function receives *handles* so as to have access to all the objects in the dialogue. The first thing it does is to obtain the values in fields *edit_wl*, *edit_wh* and

[9] It should be remarked that error dialogues are called with the modal option on, so that the user has no option but to pay them attention before doing anything else in MatLab. Their titles are descriptive themselves and tend to be different; different names allow more than one dialogue to appear simultaneously.

edit_tf. The first two are tested to check if they are positive and in the proper order. The latter is stored into a variable called *tfinal* (since *tf* is the name of a function). If anything is wrong an error dialogue is displayed, the plot is not altered, and control is returned using *return*. Then the value of the pop-up menu is stored into *plot_type*.

As soon as this is known the configuration of the dialogue is changed. This is obtained with a *switch* structure that deals with each possibility. Plots *plot1* and *plot2* are set visible if a Bode diagram is wanted, and invisible otherwise; the visibility of plot *plot* is of course set to the opposite value. Plots which will not be used are cleaned of curves and grids by plotting a single point, so that these will not appear behind the desired plots. Also fields *edit_wl*, *edit_wh* and *edit_ts* are enabled or not according to what was requested.

Then the checkbox is tested; if it is enabled and on (this is kept in a flag called *flag_var* for future use), we know that there is a plant (whose zeros, poles and gain are obtained with *zpkdata* and sorted for future use; the latter is stored into *k* since the name *gain* will be used for the modulus of the frequency response), and the fields are read. This is done with *try… catch* structures to cope with the possibility of there being an error when *evalin* is executed (in the base workspace so that names of variables may be used). Inside the *try* part a test is performed to see if the contents and dimensions are correct (in what concerns the number of columns, if there are no zeros *varzeros* does not need to have two columns, and if there are no poles *varpoles* also does not). The *catch* part displays an error dialogue but does not interrupt the plots: the checkbox is unchecked to signify that parameter variations will not be plotted and *flag_var* is set to false.

All this is done irrespective of the existence of anything to plot. That is only checked at this moment; if the steady state gain of the open loop, returned by *dcgain*, is zero, and there are neither poles nor zeros, as given by *pole* and *tzero*, this means that no controller was yet defined, and the control is returned using *return*.

There are then two cases to consider: *flag_var* may be false (in which case there is only one line in each plot) or true. If it is false, a new *switch* deals with the several possible plots. Frequency responses are obtained with *bode* (and stored in *gain* and *phase*) in the case of Bode and Nichols plots; the Nyquist plot, that does not require gain and phase but the complex value of the response, is obtained with *freqresp* (and stored in *response*); time responses are obtained with *impulse* or *step* (and stored in *out*); zeros and poles with *zpkdata* (and stored in *zeros_open_loop* and *poles_open_loop*, so as to distinguish these from *zeros* and *poles* that are those from the plant). In this last case the sampling time is checked to see what the most appropriate grid is; if it is zero, this means that the transfer function is continuous and a normal grid is set; if it larger (or -1, in the case of an unspecified sampling time), a *zgrid* is asked for. Labels are always provided to render all plots clear.

If *flag_var* is true, the code that follows is a mixture of the above *switch* with the code found in *crone3Aux1*. The sampling time and variable of the plant are stored so that corrections to the zeros and poles may be built. Frequency responses are reckoned with a fixed set of frequencies and time responses with a fixed set of time instants so that they may be stacked and plot against the same x-axis. The building of *correction* transfer functions requires no comment since it is similar to what is found in *crone3Aux1*. In the end, and beyond the nominal behaviour, another variable (or variables) with a 2 at the end (thus *gain2*, *phase2*, *response2*, *out2*) contains the several possible departures from it, inasmuch only one variations takes place at a time. These are plot in magenta to be readily distinguishable; the nominal behaviour is plotted later so that it will be in front. If zeros and poles are to be plot, the variations of those that stem from the plant are likewise stored in *zeros2* and *poles2*. These variables must be obtained by means of *for* cycles, that sweep vectors *zeros* and *poles*, since complex conjugate zeros and poles must be taken into account: those that come last (those with positive imaginary parts) are to be added to the complex conjugate of the variations of the previous one. Altered zeros appear in magenta; altered poles appear in green.

Callback of button *pushbutton_Save*

In this callback the first thing to check is if there is a controller to save; this is seen, as above, by looking at the steady-state gain of the controller, using function *dcgain*. If it is zero, an error dialogue appears.

If it is not, the string in field *edit_name* is obtained are stored in variable *name*. What follows is inside a *try… catch* structure to detect errors. A variable with the name *name* is declared as global using function *eval*. Then the controller is saved into that variable. Finally the variable is declared as global also in the workspace, so that it will be accessible there. This is done using function *evalin*.

Should there be an error, it is likewise that the cause is an invalid name provided in the field. The error message is deliberately ambiguous so that other cases be not excluded.

Callback of buttons *pushbuttonnipid* and *pushbutton2*

The callback of all these buttons is very similar. For the second button, the attempt to build a controller is framed by a *try… catch* structure, above all to deal with the case in which the main dialogue was closed. It might be reopened but handles will not be the same and errors would occur when trying to access the objects. So, if there is an error, dialogue *ninteguer* is brought to the front (or reopened, if it was closed) and an error dialogue is shown.

This is what happens if there is an error doing the following. The proper dialogue is called and the returned controller stored in *temp*. It is not immediately stored in *handles.C* since the dialogue might have been cancelled; if that was the case, the returned controller will be the null transfer function. An existing controller would be lost if *handles.C* were used. In this manner the subsequent steps are only taken if *temp* is not the null controller. Fields *C* and *open_loop* are set to *temp*. But a plant might exist with the radio button *Controller + plant* turned on. The verification is done simply by calling the callback of field *edit_plant*. This prevents unnecessarily repeating code. Finally the plot is redrawn. (If the callback of *edit_plant* did find a plant, then the plot is already redrawn, but there is no harm in doing it again. The processing time is not relevant and the code is easier to understand.)

Callback of *pushbutton3*

This callback must call two dialogues and transfer data among them. Some of the variables returned from *crone3Gui1* are passed to *crone3Gui2* (notice that *crone3Gui1* returns the plant, but only its sampling time needs to be passed on) and this dialogue returns the new controller, stored in *temp* just as above. If it is a valid controller, and beyond changing fields *C* and *open_loop* of *handles*, fields *edit_plant*, *edit_vargain*, *edit_varzeros* and *edit_varpoles* are filled with the strings returned by *crone3Gui1*, while the radio-buttons, the checkbox and associated fields are set to allow for the plant and its variations to be taken into account when the plots are redrawn.

Dialogue *nipidGui*

Tags of objects in this dialogue are as follows:

Field with the proportional gain	edit_P
Field with the integral gain	edit_I
Field with the integral order	edit_I_order
Field with the derivative gain	edit_D
Field with the derivative order	edit_D_order
Text warning about rounding offs when using Carlson method	text_warn
First pop-up menu	popupmenu1
Second pop-up menu	popupmenu2
Field with the value of *n*	edit_n
Checkbox	checkbox_decomp
Field for introducing frequency *wl*	edit_wl
Field for introducing frequency *wh*	edit_wh
Field for introducing the sampling time	edit_Ts
Button labelled *OK*	pushbutton_OK
Button labelled *Cancel*	pushbutton_Cancel

Opening function of *nipidGui*

The opening function defines a null transfer function as the default return value, stored as field *output* of *handles*, and instructs the dialogue to wait for an answer to be provided using *uiwait*.

Callback of pop-up menu *popupmenu1*

This callback reconfigures the dialogue according to the approximation chosen. This is done with a *switch* structure that verifies the value of *formula*, the number of the option in the pop-up menu. Changes consist in changing the enabled fields and the contents of the second pop-up menu and hiding or showing the warning on Carlson's method.

Callback of button *pushbutton_Cancel*

This callback resumes control with *uiresume*. Closing the window will be handled by closing function.

Callback of button *pushbutton_Ok*

This is where the calculations are performed. Values of fields are obtained and checked. If anything is wrong an error dialogue is brought up and control is returned with *return*.

The approximation specified in *popupmenu1* is obtained and stored in variable *formula*. In this manner *wl* and *wh* are only read if a continuous controller is to be built, and the sampling time and the second pop-up menu are only read if not (in this case *formula* is converted into the corresponding string using an appropriate list). Finally a *switch* structure calls the desired function. If Carlson method was chosen, the conditions stated in section 2.4 are checked. If any of them is met, function *newton* is used, as explained in the text accompanying formulas (2.41) and (2.42). Otherwise, function *nipid* is called.

Control is resumed with *uiresume*.

Output function of *nipidGui*

This function returns *handles.output* and closes the figure by deleting the handle stored in *handles*; but this is only done if *handles* is not empty. If the dialogue is called more than once (by pressing the corresponding button in the main dialogue several times), several instances of this function will run simultaneously, but only one window will appear. This will be closed when the first instance of the function terminates and the contents of *handles* will disappear, becoming an empty variable. If that is the case, the null transfer function is returned as default to avoid errors.

Dialogue *crone2Gui*

Tags of objects in this dialogue are as follows:

Gain plot of the Bode diagram	axes_gain
Phase plot of the Bode diagram	axes_phase
List-box with the frequencies	listbox1
Radio button labelled *Controller only*	radiobutton1
Button labelled *Add >>*	pushbutton_add
Field to the right of the above	edit_add
Button labelled *Load >>*	pushbutton_load
Field to the right of the above	edit_load
Button labelled *Delete*	pushbutton_delete
Radio button labelled *Gain data*	radiobutton_Gain_data
Field to the right of the above	edit_gain
Radio button labelled *Unit gain frequency*	radiobutton_Unit_gain
Field to the right of the above	edit_w1
Radio button labelled *Phase data*	radiobutton_phase
Field to the right of the above	edit_phase
Radio button labelled *Complement plant*	radiobutton_complement
Field to the right of the above	edit_plant
Field for introducing the order of the open-loop	edit_order
Field with the value of *n*	edit_n
Radio button labelled *Continuous*	radiobutton_continuous
Radio button labelled *Sampling time*	radiobutton_Ts
Field to the right of the above	edit_Ts
Button labelled *OK*	pushbutton_OK
Button labelled *Cancel*	pushbutton_Cancel

Opening function of *crone2Gui*

The opening function is similar to that of *nipidGui*. In addition it sets labels for the axis of the plots and initialises several fields of *handles* that will be necessary in what follows, namely *w* (list of frequencies), *phase* (list of phases), *gain* (list of gains) and *plant* (the plant whose phase will be complemented if no separate *phase* is given).

Callback of push button *pushbutton_add*

This function gets the value in of field *edit_add* and checks it. If it is no positive real, an error dialog is displayed; if it is, it is added to the list of frequencies in the listbox *listbox1*. Function *unique* ensures that no repeated frequencies will be added; the contents of the field are furthermore deleted. Radio buttons *radiobutton_Gain_data* and *radiobutton_phase* are turned off if they are on, because if they are this means that vectors provided as phase or gain have a number of entries equal to the number of frequencies in the listbox. Since the listbox is going to be increased, they will have the wrong length, and are thus useless. The callback of field *edit_plant* is executed, just as it is when the button is turned on manually: this redraws the plots, because if a plant is provided the phase plot will be different, taking into account that a plant may have been provided.

Callback of button *pushbutton_load*

In this function the name of the variable given in the corresponding field is obtained. A *try… catch* structure deals with the case of an invalid name or an invalid content; an error dialogue is displayed in the *catch* part. Even if the contents are valid, they cannot be negative, complex, character strings, or a row vector. The rest is similar to the above function.

Callback of button *pushbutton_delete*

This function gets the indexes of selected elements in the listbox and eliminates them using function *setdiff*. It is necessary to deselect all elements before changing the list of strings, because if an element is selected but the final list is so short that the element no longer exists in the end an error would otherwise occur.

Function *redraw*

This is a good time to explain how this function works. First of all it checks for a list of frequencies: if *handles.w* does not exist, nothing can be done.

Then the gain plot is handled. Only if radio button *radiobutton_Gain_data* is on is there data to plot in *handles.gain*. The limits of the vertical axis are left as they are; those of the frequency axis are set to the limits of the decades in which frequencies are comprised. Otherwise the plot is erased by plotting a single value, and then setting the limits of the axis to their defaults, thereby hiding the plotted point. Labels for the axes are always added. If there is a plot, the frequency axis is logarithmic, and there is a grid; this is not the case if there is nothing to plot.

The phase plot is handled next. There are three possibilities. If radio button *radiobutton_phase* is on, there is data to plot in *handles.phase*. If it is not and *handles.plant* has no plant (which is checked, as usual, by verifying if its steady state gain is zero), there is nothing to plot, and the existing plot is deleted. But if *radiobutton_phase* is not on and there is a plant, both the phase of the plant (obtained with *bode* between the extremes of vector *handles.w*) and the contents of *phase* are plotted. The case when there is only one point to plot is taken into account by means of *try… catch* structures.

Callback of radio button *radiobutton_Gain_data*

If this button is hit the first task is to get its state, which is stored into variable *button_state*. If the button is now on, radio button *radiobutton_Unit_gain* will be turned off. This is possible since only these two work together; should they be more, the situation in which none is on would have to be dealt with differently. What follows requires a *try… catch* structure. The contents of field *edit_gain* are evaluated in the usual manner described above; the *catch* part deals with errors in this operation. Additional errors (like complex numbers, characters, matrixes, vectors with a length that does not match *w*) are handled by *if*s. Whatever the other, the other button is set to on: its being on is the default value, and the validity of the unit gain frequency is not checked until *OK* is pressed. So if radio button *radiobutton_Gain_data* is on, this means that valid data for the gain is available.

If the button is now off, the other is turned on. Whatever the case, the plot may need to be refreshed; this is done with function *redraw*.

Callback of field *edit_gain*

If this field is edited, action is taken only if *radiobutton_Gain_data* is on. The corresponding code is rather similar to the one described above.

Callback of radio button *radiobutton_Unit_gain*

The code of this callback is rather similar to that of radio button *radiobutton_Gain_data*, save that the order is different.

Callback of radio buttons radiobutton_phase and radiobutton_complement and of field edit_phase

The code of these callbacks is rather similar to that of the callbacks of radio buttons *radiobutton_Gain_data* and *radiobutton_Unit_gain* and field *edit_gain*, respectively. In this instance the button which is on by default is *radiobutton_complement*. If *radiobutton_phase* is on, this means that valid data for the phase is available.

The major difference is that whenever *radiobutton_complement* is turned on, the callback of field *edit_plant* is executed. This is because the presence of a valid plant must be ascertained. But even if no valid plant is given, *radiobutton_complement* will remain on, since one of the two must be on (this in spite of an error message as will be seen below).

Callback of field *edit_plant*

This callback consists in a *try… catch* structure. The *try* part reads, in the usual manner, the name of the plant given in the field, finds its phase behaviour (using *bode*), gets the order for the open-loop provided in field *edit_order*, and sets *handles.phase* to the difference between what the open-loop should be and what the plant is.

Errors in the open-loop order are handled by an *if* inside this *try* part (and not by the *catch* part), so that the error message may be different. The plant and the phase fields of *handles* are set to naught because without an open-loop order phase data still does not exist. The plot is redrawn to erase anything that might exist and control is returned with *return*.

Callback of field *edit_order*

The previous callback is called since it handles changes in this field as well.

Callback of radio buttons radiobutton_continuous and radiobutton_Ts

These merely ensure that one and only one of these buttons is on at a given instant.

Callback of button *pushbutton_Cancel*

This callback resumes control with *uiresume*. Closing the window will be handled by closing function.

Callback of button *pushbutton_Ok*

This is where the calculations are performed. The first thing done is checking the existence of a frequency list and of a list of phases. Then the values of fields *edit_n* and *edit_Ts* (this only if the corresponding radio button is on) are obtained and checked. If anything is wrong an error dialogue is brought up and control is returned with *return*. A continuous controller is marked with a 0 in variable *Ts*.

Then the value of radio button *radiobutton_Gain_data* is checked. If it is on, function *crone2* or function *crone2z* (depending on the value of *Ts*) are called with *handles.gain* as second argument. If it is off, field *w1* is read in the usual manner (and with the usual verification) and function *crone2* or function *crone2z* are called without second argument.

Whatever the case, and unless an error occurs, the controller is stored in *handles.output*. Control is resumed with *uiresume*.

Output function of *crone2Gui*

This function is similar to that of *nipidGui*.

Dialogue *crone3Gui1*

Tags of objects in this dialogue are as follows:

Radio button labelled Closed-loop gain below	radiobutton1
Field to the right of the above	edit_spec1
Radio button labelled Closed-loop damping factor above	radiobutton2
Field to the right of the above	edit_spec2
Field for introducing frequency *wl*	edit_wl
Field for introducing frequency *wh*	edit_wh
Field for introducing the plant	edit_plant
Field for introducing *vargain*	edit_vargain
Field for introducing *varzeros*	edit_varzeros
Field for introducing *varpoles*	edit_varpoles
Button labelled *OK*	pushbutton_OK
Button labelled *Cancel*	pushbutton_Cancel

Opening function of *crone3Gui1*

The opening function is similar to that of *nipidGui*, but in this case several output variables are needed.

Callback of radio buttons *radibutton1* and *radiobutton2*

These merely ensure that one button and only one is on at a given instant.

Callback of button *pushbutton_Cancel*

This callback resumes control with *uiresume*. Closing the window will be handled by closing function.

Callback of button *pushbutton_Ok*

This is where the calculations are performed. The first thing done is obtaining the values of fields *spec1* or *spec2* (depending on the radio button which is on), *wl* and *wh* in the usual manner, and checking their content. Variables that will be part of the output become fields of *handles*, while *wl* and *wh* are stored in local variables. Then field *edti_plant* is read into *handles.strings.plant*, so that the string entered may be restored later in dialogue *ninteger*. The plant is obtained in the usual manner and its zeros and poles (obtained with *zpkdata*) are stored in local variables. This is necessary to check that matrixes with pole and zero variations have the right size. A *try… catch* structure is used, as is always the case when *evalin* is employed.

Fields *vargain*, *varzeros* and *varpoles* are read into the corresponding sub-fields of *handles*.strings; the corresponding values are also obtained with *evalin*. Their size is checked; the last two must match the number of zeros or poles of the plant. It should be noticed that if there are no zeros variable *varzeros* does not need to have two columns; the same happens with poles and *varpoles*.

Which radio button is on is checked again so that *crone3* is called with the proper value of parameter *st*. Finally control is resumed with *uiresume*.

Output function of *crone3Gui1*

This function is similar to that of *nipidGui*, save that there are nine outputs in this case.

Dialogue *crone3Gui2*

Tags of objects in this dialogue are as follows:

Gain plot of the Bode diagram	plot1
Phase plot of the Bode diagram	plot2
Plot for Nichols and Nyquist diagrams	plot
Pop-up menu	popupmenu1
Field for introducing *n*	edit_n
Leftmost list-box with all the frequencies	listbox1
Rightmost list-box with the chosen frequencies	listbox2
Button labelled *Add >>*	pushbutton_add
Button labelled *Delete*	pushbutton_delete
Button labelled *OK*	pushbutton_OK
Button labelled *Cancel*	pushbutton_Cancel

Opening function of *crone3Gui1*

The opening function bears similarities to that of *nipidGui*, but in this case it is necessary to handle the inputs to the dialogue, obtained through *varargin*. A field of *handles* called *indexes* is used for knowing which frequencies have been selected for being used with *crone2* (or *crone2z*): it will contain the indexes of such frequencies in variable *handles.w*. This is practical since it is not necessary to deal with real numbers with long decimal parts that would be hard to track. The leftmost list-box is filled with such frequencies (converted to strings using *num2str*) and function *redraw* is called for plotting the data from which a choice is to be made.

Function *redraw*

This is the simplest of all *redraw* functions since it only has to get the type of plot and plot it. The type of plot is the value chosen in the pop-up menu. Plotting commands are found inside a *switch* structure that deals with the three available cases. The Nyquist diagram is not the best way of visualising this data that consists of separate points, but was included for completeness. In all cases, the plots which are not shown are cleared just as in *redraw* from *ninteger*.

Callback of *popupmenu1*

This callback merely calls *redraw* to redraw the plot.

Callback of *pushbutton_add*

This callback gets the list of selected frequencies in list-box *listbox1* and adds them to *handles.indexes* using *union* so that no repeated values will appear. The contents of the other list-box are updated and so are the plots.

Callback of *pushbutton_delete*

This callback gets the list of selected frequencies in list-box *listbox2* and takes them off *handles.indexes* using *setdiff*. It is necessary to deselect all elements before changing updating the contents of the other list-box, because if an element is selected but the final list is so short that the element no longer exists in the end an error would otherwise occur. Finally the plots are redrawn.

Callback of button *pushbutton_Cancel*

This callback resumes control with *uiresume*. Closing the window will be handled by closing function.

Callback of button *pushbutton_Ok*

This is where the calculations are performed. The first thing done is checking whether some frequencies were selected; if none were, no calculations may be performed. Parameter *n* is found in the usual manner and, depending on the value of *Ts*, *crone2* or *crone2z* are called and the returned controller stored in *handles.output*. Control is resumed with *resume*.

Output function of *crone3Gui1*

This function is similar to that of *nipidGui*.

8. Simulink

A Simulink library called *Nintblocks* is provided with two blocks that implement functions *nid* and *nipid*.

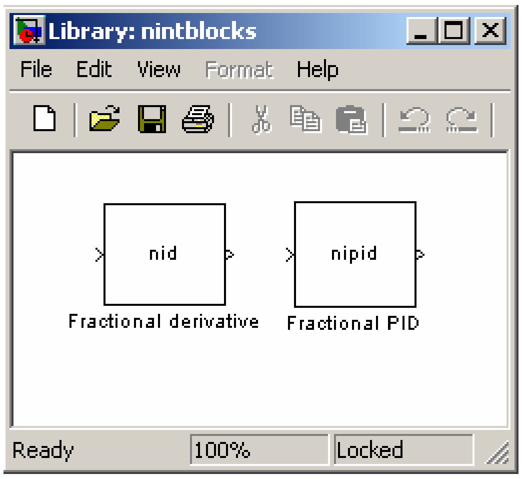
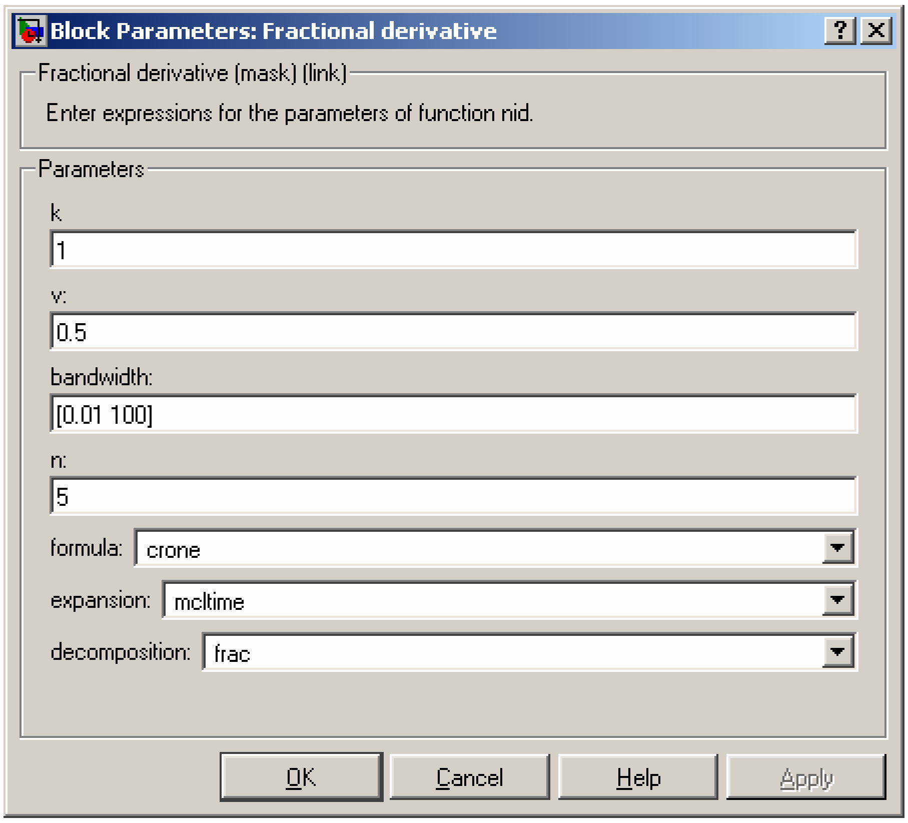


Figure 11. Simulink library

8.1. Block *Fractional derivative*

This block implements function *nid*. The parameters are provided by means of the following dialogue:


Block Parameters: Fractional derivative

Fractional derivative (mask) (link)

Enter expressions for the parameters of function nid.

Parameters

k: 1

v: 0.5

bandwidth: [0.01 100]

n: 5

formula: crone

expansion: mcltime

decomposition: frac

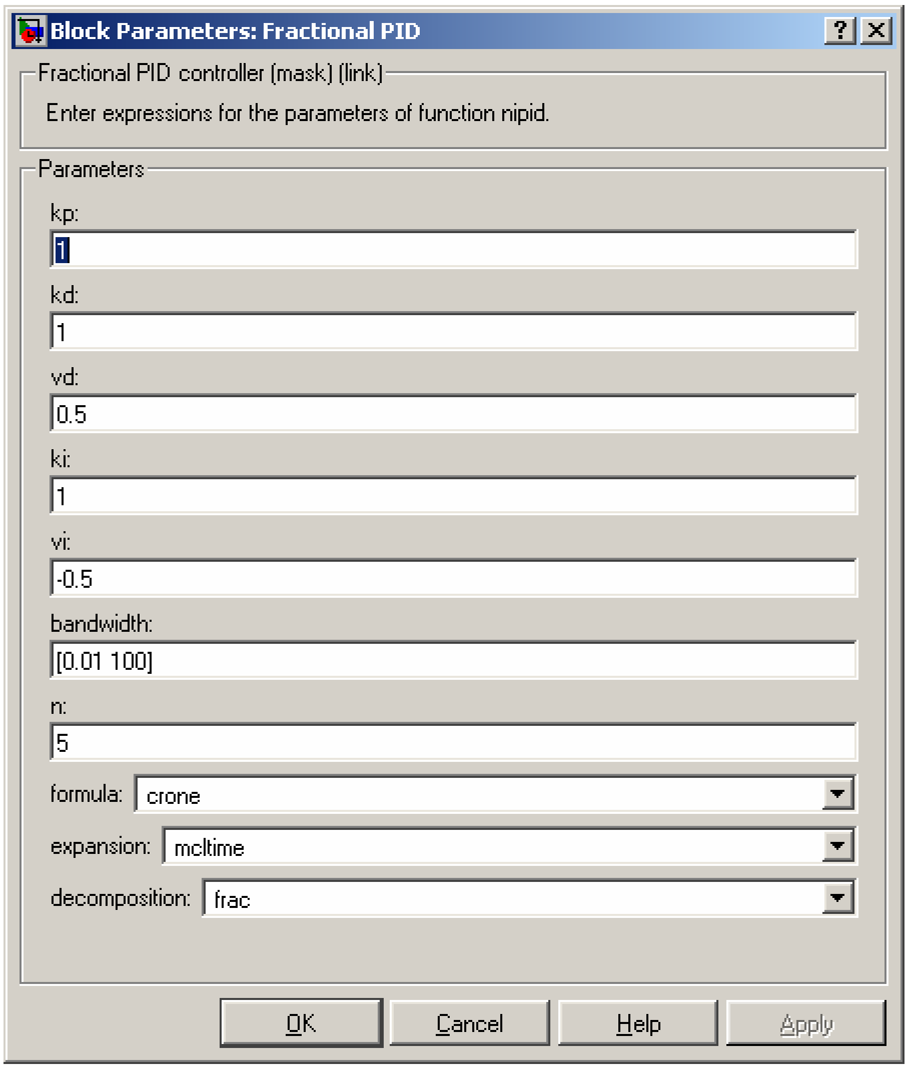
Figure 12. Dialogue of block *Fractional derivative*

Implementation

The block is implemented with a masked subsystem in which there is an LTI block calling function *nid*. In the Initialization tab of the mask of the subsystem some lines of code are needed to convert the result of the dialogue's pop-up menus into the strings required by *nid*.

8.2. Block *Fractional PID*

This block implements function *nipid*. The parameters are provided by means of the following dialogue:


Block Parameters: Fractional PID

Fractional PID controller (mask) (link)

Enter expressions for the parameters of function nipid.

Parameters

kp: 1

kd: 1

vd: 0.5

ki: 1

vi: -0.5

bandwidth: [0.01 100]

n: 5

formula: crone

expansion: mcltime

decomposition: frac

Figure 13. Dialogue of block *Fractional PID*

Implementation

The block is similar to the one above save that the function implemented is *nipid*.

9. Functions for continued fractions

In this section the information concerning four functions for working with continued fractions is included. So as to tidy the toolbox up these functions are to be found in a separate folder, called *cfractions*.

9.1. Function *contfrac*

This function returns a continued fraction, truncated after n terms, that approximates the real number x. The function's name is an acronym of continued fraction. The corresponding formulas are

$$x = x_0 = \left[\mathrm{C}\left(x_0\right); \frac{1}{\mathrm{C}\left(x_i\right)} \right]_{i=1}^{+\infty} \tag{6.41}$$

$$x_k = \frac{1}{x_{k-1} - \mathrm{C}\left(x_{k-1}\right)}, \quad k > 0 \tag{6.42}$$

Syntax

```
[a, b] = contfrac (x, n)
```

Arguments

❖ x — real number to expand.
❖ n — number of terms up to which the approximation is found.

Return values

❖ a — coefficients of the continued fraction as given by (6.42).
❖ b — coefficients of the continued fraction as given by (6.42).
These are arranged in the following general form:

$$x = a_0 + \cfrac{b_1}{a_1 + \cfrac{b_2}{a_2 + \cfrac{b_3}{a_3 + \ldots}}} \tag{6.43}$$

Obviously the coefficients of vector b will all be 1 (save b_0 which is 0). Coefficients a_0 and b_0 are found in the first position of vectors a and b. Other coefficients follow in the due order.

Limitations

Rational numbers correspond, in theory, to a continued fraction with a finite number of coefficients. However, due to floating point internal representations, sometimes integers are not recognised as such, and the difference between two numbers very close to an integer is expanded. As the difference is small, denominators in the continued fraction will be very large. This may help to recognise such cases.

Algorithm

This function implements formulas (6.41) and (6.42). The problem with test *if x==0* was already discussed in the user manual. If the test gives no problem, the expansion may be performed with a number of coefficients less than *n*. If that is the case, vectors *a* and *b* are padded with zeros in the end.

References

See (Radok, 1999), (Valério, p. 182-184, 2005c).

9.2. Function *contfracf*

This function reckons an approximation for the function

$$f(x) = \frac{c_{10} + c_{11}x + c_{12}x^2 + c_{13}x^3 + \dots}{c_{00} + c_{01}x + c_{02}x^2 + c_{03}x^3 + \dots} \tag{6.44}$$

given by the continued fraction

$$f(x) = \left[0; \frac{c_{10}}{c_{00}}, \frac{c_{j+1,0}x}{c_{j,0}} \right]_{j=1}^{+\infty} \tag{6.45}$$

$$c_{j,i} = c_{j-1,0}c_{j-2,i+1} - c_{j-2,0}c_{j-1,i+1} = -\begin{vmatrix} c_{j-2,0} & c_{j-2,i+1} \\ c_{j-1,0} & c_{j-1,i+1} \end{vmatrix} \tag{6.46}$$

truncated after *n* terms. The *f* in the function's name stands for function.

Syntax

```
[a, b] = contfracf (c, n)
```

Arguments

❖ *c* — matrix with two lines, the first with the coefficients of the polynomial in the numerator, and the second with the coefficients of the polynomial in the denominator, as seen in (6.44). The first column should have the independent coefficients, the second the coefficients in *x*, and so forth.
❖ *n* — number of terms up to which the approximation is found.

Return values

❖ *a* — coefficients of the continued fraction as given by (6.42).
❖ *b* — coefficients of the continued fraction as given by (6.42).
These are arranged in the following general form:

$$f(x) = a_0 + \cfrac{b_1}{a_1 + \cfrac{b_2 x}{a_2 + \cfrac{b_3 x}{a_3 + \ldots}}} \tag{6.47}$$

Coefficients a_0 and b_0 are both 0; these are found in the first position of vectors *a* and *b*. Other coefficients follow in the due order.

Algorithm

This function implements formula (6.46) by means of two nested *for* loops. The outermost sweeps the lines of matrix *c*; it begins with 2, since lines 0 and 1 correspond to the coefficients of the polynomials that define the function to expand. The innermost cycle deals with the several coefficients within a line. Since matrix indexes begin with 1 in MatLab, 1 must be added whenever indexes are used.

References

See (Radok, 1999), (Valério, p. 184-185, 2005c).

9.3. Function *contfraceval*

This function reckons the value of a continued fraction given by (6.43) according to the following recursive formulas:

$$\begin{aligned}
&P_{-1} = 1\\
&P_0 = a_0\\
&P_k = a_k P_{k-1} + b_k P_{k-2}, \quad k > 0\\
&Q_{-1} = 0\\
&Q_0 = 1\\
&Q_k = a_k Q_{k-1} + b_k Q_{k-2}, \quad k > 0\\
&R_k = \frac{P_k}{Q_k}
\end{aligned} \tag{6.48}$$

Syntax

```
out = contfraceval (a, b, n)
```

Arguments

❖ *a* — vector with the coefficients of (6.43). The first element should have coefficient a_0, the second coefficient a_1, and so forth.

❖ *b* — vector with the coefficients of (6.43). The first element should have coefficient b_0, the second coefficient b_1, and so forth. b_0 is never used; it is a good idea to set it to 0 to avoid confusion.

❖ *n* — number of terms up to which the continued fraction is to be evaluated. It may be a vector, in which case all the corresponding approximations are returned.

Return value

❖ *out* — value of the continued fraction up to the number of terms specified. If *n* is a vector, a line vector with the same dimension is returned.

Algorithm

This function implements formulas (6.48). All intermediate approximations are stored in variable *temp*. Since denominator Q may become too large, it is possible to set it from times to times to 1, while dividing P by Q. This preserves the result avoiding round-off errors. However, this normalisation option was not implemented so that the code might remain simple. In the end, only those values of *temp* asked for in variable *n* are returned.

References

See (Radok, 1999), (Wall, p. 15, 1948), (Valério, p. 184-185, 2005c).

9.4. Function *contfracfeval*

This function returns the value of a continued fraction, as returned by function *contfracf* for approximating *f*(*x*), in point *x*.

Syntax

```
out = contfracfeval (a, b, n, x)
```

Arguments

❖ *a* — vector with the coefficients of (6.47). The first element should have coefficient a_0, the second coefficient a_1, and so forth.
❖ *b* — vector with the coefficients of (6.47). The first element should have coefficient b_0, the second coefficient b_1, and so forth. b_0 is never used; it is a good idea to set it to 0 to avoid confusion.
❖ *n* — number of terms up to which the continued fraction is to be evaluated. It may be a vector, in which case all the corresponding approximations are returned.
❖ *x* — point in which the approximation of *f*(*x*) is to be evaluated.

Return value

❖ *out* — value of the continued fraction in point *x* up to the number of terms specified. If *n* is a vector, a line vector with the same dimension is returned.

Algorithm

This function simply multiplies vector *b* by *x* whenever appropriate and then calls function *contfraceval*.

References

See (Radok, 1999), (Wall, p. 15, 1948), (Valério, p. 184-185, 2005c).

10. Final Remarks

Devising a controller is not always a straightforward task, and some iterations and parameter tuning may often prove necessary. This toolbox was conceived as a helper, not as a substitute to human intelligence. This is particularly apparent if flaws in approximations are considered. Approximations of fractional derivatives do not always have constant phases, linear gains and exponential time responses—and sometimes fine-tuning to get some of these characteristics will only make the other worse. The algorithms for second generation CRONE controllers often get stuck due to numerical problems—and though a judicious choice of parameters usually makes the trick, what that choice may be is not always crystal clear. And so on and so forth…

In what concerns the graphical user interface, it is surely more appealing to use than typing commands at the prompt—but problems that may arise are often hidden behind generic error warnings that tell nothing about where the source of the problem is. Neither is there another alternative: realising from data and results where the hydra is hiding requires wit, and an expert rule-based system for doing that would be probably as large as the toolbox itself.

So this section is intended as a *caveat* against the perils of merely feeding functions with random parameters without backing all efforts with a reasonable dose of intellectual activity.

Bibliography

CARLSON, G. E.; HALIJAK, C. A. — Approximation of fractional capacitors $(1/s)^{1/n}$ by a regular Newton process. IEEE transactions on circuit theory. 7 (1964) 210-213.

COIS, O, LANUSSE, P., MELCHIOR, P., DANCLA, F. and OUSTALOUP, A. —. Fractional systems toolbox for Matlab: applications in system identification and Crone CSD. In 41st IEEE conference on Decision and Control. Las Vegas: IEEE, 2002.

HARTLEY, Tom; LORENZO, Carl — Fractional-order system identification based on continuous order-distributions. Signal processing. 83 (2003) 2287-2300.

LAWRENCE, P. J.; ROGERS, G. J. — Sequential transfer-function synthesis from measured data. Proceedings of the IEE. 126:1 (1979) 104-106.

LEVY, E. — Complex curve fitting. IRE transactions on automatic control. 4 (1959) 37-43.

LUBICH, C. — Discretized fractional calculus. SIAM journal of mathematical analysis. 17 (1986) 704-719.

MACHADO, J. A. Tenreiro — On the implementation of fractional-order control systems through discrete-time algorithms. In Bánki Donát Polytechnic műszaki főiskola: jubilee international conference proceedings. Bánki Donát: Bánki Donát Polytechnic, 1999. p. 39-44.

MACHADO, J. A. Tenreiro — Discrete-time fractional-order controllers. Fractional calculus & applied analysis. 1 (2001) 47-66.

MALTI, Rachid; AOUN, Mohamed; COIS, Olivier; OUSTALOUP, Alain; LEVRON, François — H_2 norm of fractional differential systems. In Proceedings of ASME 2003 design engineering technical conferences and Computers and information in engineering conference. Chicago: ASME, 2003.

MATSUDA, K.; FUJII, H. — H_∞ optimized wave-absorbing control: analytical and experimental results. Journal of guidance, control and dynamics. 16:6 (1993) 1146-1153.

MELCHIOR, P.; PETIT, N.; LANUSSE, P.; AOUN, M.; LEVRON, F.; OUSTALOUP, A. — Matlab based Crone toolbox for fractional systems. In Fractional differentiation and its application. Bordeaux: IFAC, 2004.

OUSTALOUP, Alain — La commande CRONE: commande robuste d'ordre non entier. Paris: Hermès, 1991.

OUSTALOUP, Alain; LEVRON, François; MATHIEU, Benoît; NANOT, Florence. — Frequency-band complex noninteger differentiator: characterization and synthesis. IEEE transactions on circuits and systems I. 47 (2000) 25-39.

OUSTALOUP, A., MELCHIOR, P., LANUSSE, P., COIS, O. and DANCLA, F. — The CRONE toolbox for Matlab. In 41st IEEE conference on Decision and Control. Las Vegas: IEEE, 2002.

PETRAS, I.; HYPIUSOVA, M. — Design of fractional-order controllers via H_∞ norm minimisation. In MIKLES, J.; VESELY, V. — Selected topics in modelling and control. Bratislava: Slovak University of Technology Press, 2002. vol. 3, p. 50-54.

PINA, Heitor — Métodos numéricos. Lisboa: McGraw-Hill, 1995.

PODLUBNY, Igor — Fractional differential equations: an introduction to fractional derivatives, fractional differential equations, to methods of their solution and some of their applications. San Diego: Academic Press, 1999.

RADOK, Rainer — Continued fractions. 199[9?]. http://mpec.sc.mahidol.ac.th/numer/step1aa.htm

SANATHANAN, C. K.; KOERNER, J. — Transfer function synthesis as a ratio of two complex polynomials. IEEE transactions on automatic control. 8 (1963) 56.58

VALÉRIO, Duarte — Fractional order robust control: an application. In Student Forum. Porto: EUCA, 2001a. p. 25-28.

VALÉRIO, Duarte Pedro Mata de Oliveira — Controlo robusto de ordem não inteira: síntese em frequência. Lisboa: Instituto Superior Técnico da Universidade Técnica de Lisboa, 2001b. Master's thesis.

VALÉRIO, Duarte; COSTA, José Sá da — Time domain implementations of non-integer order controllers. In Controlo 2002. Aveiro: 5th Portuguese Conference on Automatic Control, 2002. 353-358.

VALÉRIO, Duarte; SÁ DA COSTA, José — A method for identifying digital models and its application to non-integer order control. In Controlo 2004. Faro: APCA, 2004a.

VALÉRIO, Duarte; SÁ DA COSTA, José — Ninteger: a fractional control toolbox for Matlab. In First IFAC workshop on fractional differentiation and its applications. Bordeaux: IFAC, 2004b.

VALÉRIO, Duarte; SÁ DA COSTA, José — Time-domain implementation of fractional order controllers. IEE proceedings: control theory & applications. 2005a. Accepted for publication.

VALÉRIO, Duarte; SÁ DA COSTA, José — Levy's identification method extended to fractional order transfer functions. In Fifth EUROMECH Nonlinear Dynamics conference. Eindhoven: EUROMECH, 2005b.

VALÉRIO, Duarte Pedro Mata de Oliveira — Fractional robust system control. Lisboa: Instituto Superior Técnico da Universidade Técnica de Lisboa, 2005c. Doctorate thesis.

VINAGRE, B. M.; PODLUBNY, I.; HERNÁNDEZ, A.; FELIU, V. — Some approximations of fractional order operators used in control theory and applications. Fractional calculus & applied analysis. 3 (2000) 231-248.

VINAGRE JARA, Blas Manuel — Modelado y control de sistemas dinámicos caracterizados por ecuaciones íntegro-diferenciales de orden fraccional. Madrid: Universidad Nacional de Educación a Distancia, 2001. Doctorate thesis.

WALL, H. S. — Analytic theory of continued fractions. Princeton: D. Van Nostrand Company, 1948.

Appendix

This Appendix contains alternative code for functions *rad2deg* and *deg2rad* that may be used if toolbox Map is not available.

```
function deg = rad2deg(rad)
% function deg = rad2deg(rad)
% deg = 180 * real(rad) / pi;
deg = 180 * real(rad) / pi;

function rad = deg2rad(deg)
% function rad = deg2rad(deg)
% rad = pi * real(deg) / 180;
rad = pi * real(deg) / 180;
```