CRONE®

Toolbox

Fractional Systems Toolbox

Volume 1/5

Fractional Calculus Module

User's Guide

Equipe CRONE

Commande – Robotique
Ordres Non Entiers

Version 5.0
May 25, 2007

LAPS - UMR 5131 CNRS
Université Bordeaux 1- ENSEIRB
351 cours de la Libération - F33405 TALENCE Cedex - France
Tél : +33 (0)5 40 00 66 07 - Fax +33 (0)5 40 00 66 44
Email : accueil@laps.u-bordeaux1.fr - URL : www.laps.u-bordeaux1.fr

Equipe CRONE

Commande – Robotique
Ordres Non Entiers

CRONE® Toolbox

Fractional Systems Toolbox

Scientific Responsible:
Alain OUSTALOUP, Professor, ENSEIRB

Coordinator:
Pierre MELCHIOR, Associate Professor, ENSEIRB

Project Manager:
Nicolas PETIT, Engineer

Scientific Contributor and Developers:
Olivier ALTET, Doctor
Mohamed AOUN, PhD Student
Olivier COIS, Doctor
Nicolas PETIT, Engineer
Patrick LANUSSE, Associate Professor, ENSEIRB

Scientific Contributor:
François LEVRON, Associate Professor, University Bordeaux 1
Xavier MOREAU, Associate Professor, University Bordeaux 1
Jocelyn SABATIER, Associate Professor, IUT, Dpt GEII, University Bordeaux 1

Contact: pierre.melchior@laps.ims-bordeaux.fr; crone@laps.ims-bordeaux.fr

Contents

Preface

This software treats of fractional derivative, how to calculate it, how to synthesize it, its applications in mathematics and in engineer science, such as automatic, identification and control. The objective of this toolbox fits with the will to transfer, distribute and enhance the value on international level, as well as in teaching, as in research, or in the industry, of upstream concept developed in laboratory.

The toolbox "***CRONE Toolbox: Fractional Systems Toolbox***" has been developed since the begin of the nineties. It is the subject of several publications, thesis and articles, and has been registered at the "Agence pour la Protection des Programmes" (APP, Software Protection Agency) in 1993 and 1994 [APP94].

At the moment, the toolbox CRONE is made up of four modules, each modulus treats one of the application theme of fractional differentiation :
- ***Fractional calculus***
- ***Time Domain System Identification by Fractional Model***
- ***Frequency Domain System Identification by Fractional Model***
- ***CRONE Control System Design***.

These modules focus on a will to limit, for the beginning, on the scalar case, in order to ensure a progressive and incentive learning to the user.

Matlab was chosen for its numerous advantages : numeric calculation algorithm on complex matrix, high level programming language, graphical display functions, easy IHM creation (menus, capture area, etc …). The portability on other environment is also a significant advantage to facilitate the distribution of this toolbox.

Moreover, most of the university laboratories and the Research and Development industrial services use this software. It becomes in fact a worldwide standard of pluridisciplinary calculus software, particularly in the automatic domain.

The development of this toolbox was done in collaboration with the group PSA Peugeot-Citroën and the financial support of the Aquitaine region.

1 Introduction

Other software tools currently available cannot be used for systems with fractional derivatives (i.e. non integer or complex order derivatives). The "Fractional Calculus" module includes all algorithms which allow the use of fractional or complex derivation. The following modules of the CRONE Toolbox have been developed on the base of the theoretical work of the CRONE team of the LAP, gradually since the end of the eighties :

- "Fractional Derivative" unit,
- "Explicit Form System (Differential Equations)" unit,
- "Implicit Form System" unit,
- "Fractional Differentiator" unit,
- "Fractional Polynomial Roots" unit,
- "Laplace Transform" unit.

2 Principle

"Fractional Derivative" unit

This unit enables the user to compute the fractional or complex order derivative of time functions from literal expressions or from external data files.

The definition of a complex order derivative was given by Riemann-Liouville [MIL93] [SAM93] [COIS00] in the ninetieth century.
The *n* complex order integral of a complex function *f(t)* is defined by :

$$I^n f(t) = \frac{1}{\Gamma(n)} \int_0^t \frac{f(\tau)}{(t-\tau)^{1-n}} \, d\tau \text{, with } \begin{cases} t > 0 \\ n \in \mathbb{C} \\ \Re e(n) > 0 \end{cases}, \qquad (1)$$

where $\Gamma(n)$ is the Euler Gamma function extended to complex numbers.

$$\Gamma(n) = \int_0^{\infty} \mathrm{e}^{-x} x^{n-1} dx \ . \qquad (2)$$

Also, the *n* complex order derivative, such as $\mathrm{Re}(n)>0$, of a complex function *f(t)* is defined by :

$$D^n f(t) = \frac{1}{\Gamma(m-n)} \left(\frac{d}{dt}\right)^m \left(\int_0^t \frac{f(\tau)}{(t-\tau)^{1-(m-n)}} d\tau \right), \qquad (3)$$

with $\begin{cases} t > 0 \\ \mathrm{Re}(n) > 0 \\ m = \lfloor \mathrm{Re}(n) \rfloor + 1 \end{cases}$, $\lfloor \mathrm{Re}(n) \rfloor$ is the integer part of $\mathrm{Re}(n)$.

Remark
If $\mathrm{Re}(n)<0$, the *n* complex order integral of a complex function *f(t)* is then defined by :

$$I^n f(t) = D^{-n} f(t), \qquad (4)$$

also the *n* complex order derivative is defined by :

$$D^n f(t) = I^{-n} f(t). \qquad (5)$$

♦

The use of this first definition has many drawbacks. A second definition which computes integer, real or complex order derivatives is the work basis on which the computation algorithm of the fractional derivative of the CRONE toolbox is developed. The idea underlying this definition consists to generalize the integration notion illustrated by the surface under the plot by introducing the memory notion (and a forgetting factor) with a weighting higher for the oldest samples, the weighting being function of the order of the derivation.

The definition [OUS91], [MIL93], [SAM93], can developed into an algorithm which computes a fractional derivative with a data vector [GRU67] :

$$D_h^{(n)} f(t) = \frac{1}{h^n} \sum_{k=0}^{+\infty} \left[(-1)^k \binom{n}{k} f(t-kh) \right] \tag{6}$$

The calculation error is in *O(h)*, where h is the sample period. A second algorithm can improve the precision, and enables to have an error in $O(h^2)$, $O(h^3)$,… depending on the option chosen by the user. In this case the following relation is used :

$$\begin{aligned} D^{(n)} f(t) &= D_h^{(n)} f(t) + \frac{nh}{2} D_h^{(n+1)} f(t) \\ &+ \frac{(3n^2+5n)h^2}{24} D_h^{(n+2)} f(t) + \frac{(n^3+5n^2+6n)h^3}{48} D_h^{(n+3)} f(t) \\ &+ \frac{(15n^4+150n^3+485n^2+502n)h^4}{5760} D_h^{(n+4)} f(t) + O\left(h^5\right) \end{aligned} \tag{7}$$

Remark
The computation algorithm of the fractional derivative gives a result with an asymptotic error which can be estimate with the following relation:

$$D_h^n(f(t)) = \frac{1}{h^n} \sum_{k=0}^{\infty} (-1)^k \binom{n}{k} f(t-kh), \tag{8}$$

The Laplace transform of this expression is:

$$L\left[D_h^n(f(t))\right] = \frac{1}{h^n} \sum_{k=0}^{\infty} (-1)^k \binom{n}{k} F(p) \exp(-khp) = F(p) \left(\frac{1-\exp(-hp)}{h} \right)^n . \tag{9}$$

Using Taylor development of the exponential term, it becomes:

$$L\left[D_h^n(f(t))\right] = F(p) \left[\frac{1}{h} \left(1 - \left(1 - hp + \frac{h^2p^2}{2!} - \frac{h^3p^3}{3!} + O\left(h^4\right) \right) \right) \right]^n , \tag{10}$$

whence

$$L\left[D_h^n(f(t))\right] = p^n \left(1 - \frac{hp}{2!} + \frac{h^2p^2}{3!} - \frac{h^3p^3}{4!} + \frac{h^4p^4}{5!} + O\left(h^5\right) \right)^n F(p), \tag{11}$$

Using Taylor development of the term $(1-u)^n$, one finally finds :

$$L\left[D_h^n(f(t))\right] = p^n \left(1 - \frac{n}{2} hp + \left(\frac{n+3n^2}{24} \right) h^2p^2 - \left(\frac{n^3+n^2}{48} \right) h^3p^3 + O\left(h^4\right) \right) F(p) . \tag{12}$$

The inverse Laplace transform of this expression is:

$$D_h^n(f(t)) = D^n(f(t)) - \frac{n}{2} h D^{n+1}(f(t)) + \frac{n+3n^2}{24} h^2 D^{n+2}(f(t)) - \frac{n^3+n^2}{48} h^3 D^{n+3}(f(t)) + O(h^4), \quad (13)$$

Using the following expression:

$$D^n(f(t)) = D_h^n(f(t)) + a_1(n) h D_h^{n+1}(f(t)) + a_2(n) h^2 D_h^{n+2}(f(t)) + a_3(n) h^3 D_h^{n+3}(f(t)) + O(h^4), \quad (14)$$

Substituting the expression of $D^n(f(t))$ in the equation (13) by the equation (14) gives:

$$\begin{gathered} D_h^n(f(t)) = D_h^n(f(t)) + \left(a_1(n) - \frac{n}{2}\right) h D_h^{n+1}(f(t)) + \qquad , \\ \left(a_2(n) - a_1(n+1)\frac{n}{2} + \frac{n+3n^2}{24}\right) h^2 D_h^{n+2}(f(t)) + O(h^3) \end{gathered} \quad (15)$$

Whence

$$\begin{cases} a_1(n) - \dfrac{n}{2} = 0 \\ a_2(n) - a_1(n+1)\dfrac{n}{2} + \dfrac{n+3n^2}{24} = 0 \\ \vdots \end{cases} \quad (16)$$

The result is:

$$\begin{aligned} D^{(n)} f(t) &= D_h^{(n)} f(t) + \frac{nh}{2} D_h^{(n+1)} f(t) \\ &+ \frac{(3n^2+5n)h^2}{24} D_h^{(n+2)} f(t) + \frac{(n^3+5n^2+6n)h^3}{48} D_h^{(n+3)} f(t) \\ &+ \frac{(15n^4+150n^3+485n^2+502n)h^4}{5760} D_h^{(n+4)} f(t) + O(h^5) \end{aligned} \quad (17)$$

The error is directly linked to the sample period *h*.

♦

"Explicit Form System (Differential Equations)" unit

In a time description of the dynamic behavior of a scalar linear system, a differential equation is "fractional" when the first member is a linear combination of fractional derivatives of output signals, and the second member a linear combination of fractional derivatives of input signals [OUS91]:

$$\sum_{r=1}^{R} a_r D^{(n_r)} s(t) = \sum_{q=1}^{Q} a_q D^{(n_q)} e(t) . \qquad (18)$$

The second member is immediately calculable with algorithms from "Fractional derivative" unit. In the first member, the output s(t) is determined with the past of the signal by the following relation:

$$\sum_{r=1}^{R} \frac{a_r}{h^{n_r}} s(t) = \sum_{q=1}^{Q} a_q D^{(n_p)} e(t) - \sum_{r=1}^{R} a_r \left[\frac{1}{h^{n_r}} \sum_{k=1}^{+\infty} \left((-1)^k \binom{n_r}{k} s(t-kh) \right) \right] \cdot \qquad (19)$$

This unit enables to simulate the time response of such a differential equation with complex order derivations (so real and integer are included).

This unit was extended to the linear multivariable processes (MIMO). These systems are described by fractional differential equations with linked inputs and outputs [NAN96]:

$$\left\{ \sum_{v=1}^{V} \sum_{r=1}^{R} a_{r,v} D^{(n_{r,v})} s_v(t) = \sum_{w=1}^{W} \sum_{q=1}^{Q} a_{q,w} D^{(n_{q,w})} e_w(t) . \right. \qquad (20)$$

In the same way, the outputs are given at the time t according to the past of the outputs and inputs signals.

In the monovariable and multivariable cases, the algorithm uses the second definition of fractional order derivative [OUS91], [MIL93], [SAM93] :

$$D^{(n)} e(t) = \frac{1}{h^n} \sum_{k=0}^{+\infty} \left[(-1)^k \binom{n}{k} e(t-kh) \right] \cdot \qquad (21)$$

"Implicit Form System" unit

This unit allows to simulate the time response of a process described by a transfer function with implicit fractional derivative [OUS91], such as :

$$\frac{S(s)}{E(s)} = \frac{\sum_p \left[\prod_q \left(1+\tau_{p,q} s\right)^{n_{p,q}} \right]}{\prod_k \left(1+\tau_k s\right)^{n_k}} . \tag{22}$$

Example of implicit form system :

$$\frac{S(s)}{E(s)} = \frac{(1+2.89s)^{0.7}(1+5s)^{1.2+0.8i}(1+4.77s)^{0.4} + (1+3.65s)^{1.6}}{(1+8s)^{0.6+1.3i}(1+1.32s)^{3.8}} \tag{23}$$

♦

There is no simple form of differential equation of this transfer function with implicit fractional derivative.
Indeed, the inverse Laplace transform of the transfer function :

$$S(s) = (1+\tau s)^n E(s) \tag{24}$$

is:

$$s(t) = \tau^n \exp\left(\frac{-t}{\tau}\right) D^n_{imp,\tau}(e(t)). \tag{25}$$

Remark
Setting $s' = s + 1/\tau$, we obtain :

$$S(s'-1/\tau) = \tau^n s'^n \, E(s'-1/\tau) \tag{26}$$

The inverse Laplace transform of the equation is :

$$s(t)\exp\left(\frac{t}{\tau}\right) = \tau^n D^n \left(e(t)\exp\left(\frac{t}{\tau}\right)\right), \tag{27}$$

thus:

$$s(t) = \tau^n \exp\left(\frac{-t}{\tau}\right) D^n_{imp,\tau}(e(t)). \tag{28}$$

♦

To compute the time response, the output S(s) is expressed according to the input E(s):

$$S(s) = \sum_{p=0}^{P} \left[\prod_{q=0}^{Q_p} \left(1+\tau_{p,q} s\right)^{n_{p,q}} \left[\prod_{k=0}^{K} \left(1+\tau_k s\right)^{-n_k} (E(s)) \right]\right] \tag{29}$$

From this form, the output signal s(t) is then determined by:

$$\boxed{\begin{array}{l} s(t)=\sum_{p=0}^{P}\left\{\tau_{p,Q_p}^{n_{p,Q_p}}e^{\left(\frac{-t}{\tau_{p,Q_p}}\right)}D_{imp,\tau_{p,Q_p}}^{n_{p,Q_p}}\left[\cdots\left[\tau_{p,0}^{n_{p,0}}e^{\left(\frac{-t}{\tau_{p,0}}\right)}D_{imp,\tau_{p,0}}^{n_{p,0}}\left(f(t)\right)\right]\cdots\right]\right\} \\ avec\ \ f(t)=\left\{\tau_K^{-n_K}e^{\left(\frac{-t}{\tau_K}\right)}D_{imp,\tau_K}^{-n_K}\left[\cdots\left[\tau_0^{-n_0}e^{\left(\frac{-t}{\tau_0}\right)}D_{imp,\tau_0}^{-n_0}\left(e(t)\right)\right]\cdots\right]\right\} \end{array}}. \qquad (30)$$

Although this expression is rather heavy to write, it is relatively simple to program with the fractional derivative algorithm, defined by the relation:

$$D^{(n)}e(t)=\frac{1}{h^n}\sum_{k=0}^{+\infty}\left[(-1)^k\binom{n}{k}e(t-kh)\right]. \qquad (31)$$

"Fractional Differentiator" unit

Because the fractional derivative takes into account all the past of the function, its real time calculation cannot be carried out; only the fractional derivative of the functions with finished past can be calculated.

In the general case, it is advisable to calculate the fractional derivative with a recursive equation obtain through discretisation of a fractional derivative which truncates low and high frequencies.

This unit synthesizes a frequency-bounded fractional differentiator:

$$D_{fbl}(s) = C_0 \left(\frac{1+\frac{s}{\omega_b}}{1+\frac{s}{\omega_h}} \right)^n \tag{32}$$

The synthesis of the rational differentiator uses a recursive distribution of real zeros and poles and a rational order derivative unit:

$$D_{rational}(s) = C_0 \prod_k \left(\frac{1+\frac{s}{\omega_{bk}}}{1+\frac{s}{\omega_{hk}}} \right) \tag{33}$$

N is the number of zeros and poles and the recursion factors α and η are determined by the relations: $n = \frac{\log\alpha}{\log\alpha\eta}$, and $\alpha = \left(\frac{\omega_h}{\omega_b}\right)^{n/N}$ $\& \, \eta = \left(\frac{\omega_h}{\omega_b}\right)^{(1-n)/N}$ and $\frac{\omega_{hk}}{\omega_{bk}} = \alpha$ & $\frac{\omega_{b(k+1)}}{\omega_{hk}} = \eta$.

To compare the frequency responses of the three differentiators are displayed in a Bode diagram and Nichols charts:

- fractional differentiator :

$$D_{fractional}(s) = C_0 \left(\frac{s}{\omega_u} \right)^n \tag{34}$$

- frequency-bounded differentiator:

$$D_{fbl}(s) = C_0 \left(\frac{1+\frac{s}{\omega_b}}{1+\frac{s}{\omega_h}} \right)^n \tag{35}$$

- rational differentiator:

$$D_{rational}(s) = C_0 \prod_k \left(\frac{1+\dfrac{s}{\omega_{b\,k}}}{1+\dfrac{s}{\omega_{h\,k}}} \right) \tag{36}$$

This comparison makes it possible to evaluate the degradation related to the approximation of the fractional differentiator by a recursive distribution of zeros and poles.

"Fractional Polynomial Roots" unit

Defined as the denominator of the transmittance of a fractional differential equation equaled to zero, the characteristic equation can have integer, fractional, real or complex powers:

$$\sum_{l=1}^{L} a_l \, s^{n_l} = 0 \, . \tag{37}$$

This unit enables to calculate the roots of such an equation. Its resolution in the particular case of real fractional powers rests on the approximation of these powers by rational powers. A variable change makes it possible to be brought back to an integer degrees equation of fractional order variable. The roots of this equation are then given in accordance with a cut of the complex plane following $\mathbb{R}$- (the cut can be parameterized).

The algorithm which results from this is the one which uses the module "Fractional Polynomial Roots" of CRONE toolbox.

The poles of a system can be given by specifying either the denominator of transmittance, or the matrix of evolution A, or the vector of the eigenvalues.

"Laplace Transform" unit

The current algorithms of numerical calculation of Laplace transform or inverse Laplace transform, are not very satisfying. The Maple software, for example, allows calculation symbolic system of simple functions, but is inoperative for the functions met in the case of fractional derivation.

New methods must be developed to allow the user to solve fractional differential equations. Within this framework, a new method is proposed; this one calculates a numerical approximation of Laplace transform of a real or complex function, or inverse Laplace transform. The precision obtained with this method is however excellent only in the small times (or at the high frequencies). This is why this subject of search remains open. De nouvelles méthodes doivent être développées pour permettre la résolution d'équations différentielles généralisées.

It was demonstrated that the optimal expression with a precision point of view of the Laplace transform F(s) (reciprocally inverse Laplace transform f(t)) may be approximated by a finished sum of terms depending on the original function f(t) (resp. function F(s) symbolic system), in which the time variable t (resp. the variable symbolic system s) is replaced by a new variable proportional to its Laplace transform 1/s (resp. with its original 1/t).

Thus, the method consists in determining this series by a direct optimization aiming at minimizing the difference between its Taylor development and that of the Laplace transform (reciprocally, the inverse Laplace transform) of the original function f(t) (resp. the function F(s) symbolic system) written in the form of a polynomial sum of the variable t (variable s). This optimization is carried out by solving the system of Aitken.

We assume the function f(t) can be rewrite in this form :

$$f(t)=\sum_{k=0}^{2N-1}\left(\frac{a_k t^{(a-1+kb)}}{\Gamma(a+kb)}\right). \qquad (38)$$

with 2N the term numbers of the series.

The Laplace transform is then:

$$\mathrm{L}(f(t))=\sum_{k=0}^{2N-1}\left(\frac{a_k}{s^{(a+kb)}}\right). \qquad (39)$$

The coefficients a and b depend on properties of the function f(t):
- a depends on the asymptotic behavior of f(t) near zero $f(t)\approx At^{a-1}$.
- b is the common step of all powers of the function f(t) written in the form of series (for example, b=1 if the function is odd, and b=2 if the function is even).

We obtain an approximate expression:

$$\mathrm{L}(f(t))=\sum_{k=1}^{N}\left(A_k f(\frac{x_k}{s})\right), \qquad (40)$$

which leads to determine A_l and x_l so to solve the equation :

$$\sum_{l=1}^{N} A_l x_l^{(a-1+kb)} = \frac{\Gamma(a+kb)}{s}, \text{ with } 0 \le k \le 2N-1. \qquad (41)$$

Thus, the method consists in solving the system of N equations to determine N first terms of the series (eq. .41), and to calculate the Laplace transform of the function f(t). The precision obtained with this method is excellent, but only at small times.

For the inverse Laplace transform, the result is completely similar.
This new method and the associated developments will be presented in a specific article. We will present the method in it but also a comparison with the principal already existing methods of numerical calculation.

Bibliography

[APP94] Agence pour la Protection des Programmes - N° 93.30.006.00 (IDDN.FR.001.300006.00. R.P.1993.000.00000 - 28/07/1993) ; N° 94.11.015.00 (IDDN.FR.001.110015.00. R.P.1994.000.00000) - le16/03/1994.

[BAN93] M. Bansard Du formalisme de la dérivation généralisée à l'unité Outils Mathématiques de la toolbox CRONE, Thèse de Doctorat, Université Bordeaux I, 1993.

[COIS00] O. Cois et F. Levron - Systèmes à dérivées d'ordre complexe - IEEE Conférence Internationale Francophone d'Automatique, CIFA'2000, sous l'égide du GdR Automatique du CNRS et du GRAISYyHM - Lille, France, 5-8 Juillet 2000.

[GRU67] A.K. Grünwald, Über "begrenzte" Derivation und deren Anwendung, Z. Angew. Math. Phys., 12, 441-480, 1867.

[LEV99] F. Levron, J. Sabatier, A. Oustaloup and L. Habsieger - From partial differential equations of propagative recursive systems to non integer differentiation - Fractional Calculus and Applied Analysis (FCAA) : an international journal for theory and applications, Vol. 2, N° 3, pp 246-264, July 1999.

[MEL99] P. Melchior, P. Lanusse, F. Dancla et O. Cois, Valorisation de l'approche non entière par le logiciel CRONE, Colloque sur l'Enseignement des Technologies et des Sciences de l'Information et des Systèmes en EEA, CETSIS-EEA'99, Montpellier, 1999.

[MIL93] K.S. Miller and B. Ross, An introduction to the fractional calculus and fractional differential equations, A Wiley-Interscience Publication, 1993.

[NAN96] F. Nanot, "Dérivateur Généralisé et Représentation Généralisée des Systèmes Linéaires", Thèse de Doctorat, Université Bordeaux I, 1996.

[OLD74] K. B. Oldham and J. Spanier, The fractional calculus, Academic Press, New York and London, 1974.

[OUS83] A. Oustaloup, Systèmes asservis linéaires d'ordre fractionnaire, Masson, 1983.

[OUS97] A Oustaloup, P. Lanusse et P. Melchior, Le logiciel CRONE , Journées d'Etude sur les Logiciels pour le traitement de l'Image, du Signal et l'Automatique, Club EEA - Elisa'97, Nancy, Mars 1997.

[OUS91] A. Oustaloup, "La commande CRONE", Hermès 1991.

[OUS95] A. Oustaloup, La dérivation non entière: théorie, synthèse et applications, Editions Hermès, Paris, 1995.

[OUS00a] A. Oustaloup, F. Levron, B. Mathieu and F. Nanot - Frequency-Band Complex Non Integer Differentiator : Characterization and Synthesis - IEEE Transactions on Circuits and Systems, Vol. 47, N°1, pp 25-40, January 2000.

[OUS00b] A. Oustaloup, P. Melchior, P. Lanusse, O. Cois and F. Dancla, The CRONE toolbox for Matlab, 11th IEEE Inrternational Symposium on Computer-Aided Control System Design, CACSD, Anchorage, Alaska - USA, September 2000.

[SAM93] S.G. Samko, A.A. Kilbas and O.I. Marichev, Fractional integrals and derivatives: theory and applications, Gordon and Breach Science Publishers, 1993.

3 Object oriented programming

Definitions

Object-oriented programming may be seen as a collection of cooperating objects, as opposed to a traditional view in which a program may be seen as a list of instructions to the computer. In OOP, each object is capable of receiving messages, processing data, and sending messages to other objects. Each object can be viewed as an independent little machine with a distinct role or responsibility.
By way of "objectifying" software modules, object-oriented programming is intended to promote greater flexibility and maintainability in programming, and is widely popular in large-scale software engineering. By virtue of its strong emphasis on modularity, object oriented code is intended to be simpler to develop and easier to understand later on, lending itself to more direct analysis, coding, and understanding of complex situations and procedures than less modular programming methods.
The fundamental concepts of object oriented programming:

Class
A class defines the abstract characteristics of a thing (object), including the thing's characteristics (its attributes, fields or properties) and the thing's behaviors (the things it can do or methods or features). For example, the class Dog would consist of traits shared by all dogs, such as breed and fur color (characteristics), and the ability to bark (behavior). Classes provide modularity and structure in an object-oriented computer program. A class should typically be recognizable to a non-programmer familiar with the problem domain, meaning that the characteristics of the class should make sense in context. Also, the code for a class should be relatively self-contained. Collectively, the properties and methods defined by a class are called members.
Object
A particular instance of a class. The class of Dog defines all possible dogs by listing the characteristics and behaviors they can have; the object Lassie is one particular dog, with particular versions of the characteristics. A Dog has fur; Lassie has brown-and-white fur. In programmer jargon, the object Lassie is an instance of the Dog class. The set of values of the attributes of a particular object is called its state.The object consists of state and the behaviour that's defined in the object's class.
Method
An object's abilities. Lassie, being a Dog, has the ability to bark. So bark() is one of Lassie's methods. She may have other methods as well, for example sit() or eat(). Within the program, using a method should only affect one particular object; all Dogs can bark, but you need one particular dog to do the barking.
Message passing
"The process by which an object sends data to another object or asks the other object to invoke a method." Also known to some programming languages as interfacing. Lassie may give another dog one of her bones.

Interests

The OOP have four interests:

Inheritance

"Subclasses" are more specialized versions of a class, which inherit attributes and behaviors from their parent classes, and can introduce their own.
For example, the class Dog might have sub-classes called Collie, Chihuahua, and GoldenRetriever. In this case, Lassie would be an instance of the Collie subclass. Suppose the Dog class defines a method called bark() and a property called furColor. Each of its sub-classes (Collie, Chihuahua, and GoldenRetriever) will inherit these members, meaning that the programmer only needs to write the code for them once.
Each subclass can alter its inherited traits. For example, the Collie class might specify that the default furColor for a collie is brown-and-white. The Chihuahua subclass might specify that the bark() method produces a high-pitched by default. Subclasses can also add new members. The Chihuahua subclass could add a method called tremble(). So an individual chihuahua instance would use a high-pitched bark() from the Chihuahua subclass, which in turn inherited the usual bark() from Dog. The chihuahua object would also have the tremble() method, but Lassie would not, because she is a Collie, not a Chihuahua. In fact, inheritance is an "is-a" relationship: Lassie is a Collie. A Collie is a Dog. Thus, Lassie inherits the members of both Collies and Dogs.
Multiple inheritance is inheritance from more than one ancestor class, neither of these ancestors being an ancestor of the other. For example, independent classes could define Dogs and Cats, and a Chimera object could be created from these two which inherits all the (multiple) behavior of cats and dogs. This is not always supported, as it can be hard both to implement and to use well.

Encapsulation

Encapsulation conceals the functional details of a class from objects that send messages to it.
For example, the Dog class has a bark() method. The code for the bark() method defines exactly how a bark happens (e.g., by inhale() and then exhale(), at a particular pitch and volume). Timmy, Lassie's friend, however, does not need to know exactly how she barks. Encapsulation is achieved by specifying which classes may use the members of an object. The result is that each object exposes to any class a certain interface — those members accessible to that class. The reason for encapsulation is to prevent clients of an interface from depending on those parts of the implementation that are likely to change in future, thereby allowing those changes to be made more easily, that is, without changes to clients. For example, an interface can ensure that puppies can only be added to an object of the class Dog by code in that class. Members are often specified as public, protected or private, determining whether they are available to all classes, sub-classes or only the defining class.

Abstraction

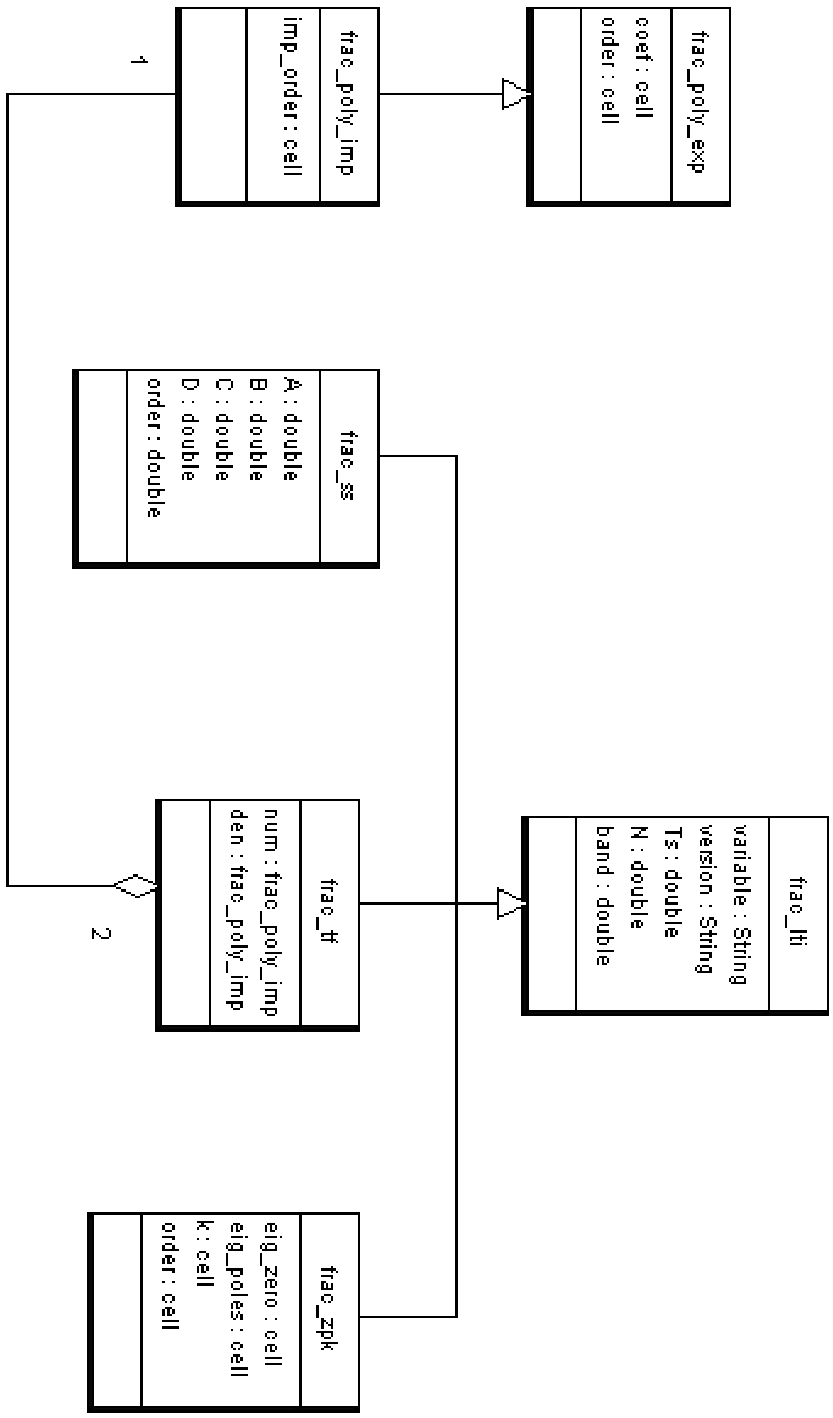
Abstraction is simplifying complex reality by modeling classes appropriate to the problem, and working at the most appropriate level of inheritance for a given aspect of the problem.
For example, Lassie the Dog may be treated as a Dog much of the time, a Collie when necessary to access Collie-specific attributes or behaviors, and as an Animal (perhaps the parent class of Dog) when counting Timmy's pets. Abstraction is also achieved through Composition. For example, a class Car would be made up of an Engine, Gearbox, Steering objects, and many more components. To build the Car class, one does not need to know how the different components work internally, but only how to interface with them, i.e., send messages to them, receive messages from them, and perhaps make the different objects composing the class interact with each other.

Polymorphism

Polymorphism allows you to treat derived class members just like their parent class's members. More precisely, Polymorphism in object-oriented programming is the ability of objects belonging to different data types to respond to method calls of methods of the same name, each one according to an appropriate type-specific behavior. One method, or an operator such as +, -, or *, can be abstractly applied in many different situations. If a Dog is commanded to speak(), this may elicit a Bark. However, if a Pig is commanded to speak(), this may elicit an Oink. They both inherit speak() from Animal, but their derived class methods override the methods of the parent class; this is Overriding Polymorphism. Overloading Polymorphism is the use of one method signature, or one operator such as "+", to perform several different functions depending on the implementation. The "+" operator, for example, may be used to perform integer addition, float addition, list concatenation, or string concatenation. Any two subclasses of Number, such as Integer and Double, are expected to add together properly in an OOP language. The language must therefore overload the concatenation operator, "+", to work this way. This helps improve code readability. How this is implemented varies from language to language, but most OOP languages support at least some level of overloading polymorphism. Many OOP languages also support Parametric Polymorphism, where code is written without mention of any specific type and thus can be used transparently with any number of new types. Pointers are an example of a simple polymorphic routine that can be used with many different types of objects.

Classes diagram

CRONE Toolbox is organized accounting to the class diagram shown below. Only attributes associated to each class are shown. Method are listed further.



frac_poly_exp class

Attributes

Attribute name	Description	Value
coef	Coefficients of the frac_poly_exp object	Cell Nu*Ny of double vector
order	Orders of the frac_poly_exp object	Cell Nu*Ny of double vector

Function's list

append
cancel_zero_coef (private)
char
clean
coef
commensurate
display
eig (a revoir)
enlarge
eq
get
horzcat
iscomplex
isempty
isnan
ldivide
match_same_order (private)
minus
mpower
mtimes
multi (private)
ne
order
parallel
plus
rdivide
roots (a revoir)
series
set
size
sort
subsasgn
subsref
times
transpose
uminus
vertcat

append

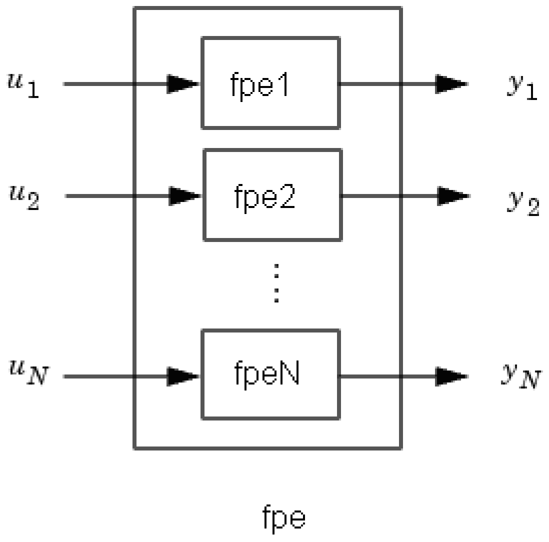
`Appends` frac_poly_exp objects by appending their inputs and outputs

Syntax

```
    fpe = append(fpe1,fpe2,...,fpeN)
```

Description

`append` appends the inputs and outputs of the LTI models fpe1,...,fpeN to form the augmented model fpe depicted below.



Arguments

The input arguments fpe1,..., fpeN are frac_poly_exp objects.

There is no limitation on the number of inputs.

Example

```
>> p1
s^2.2 +s^1.5
>> p2
s^0.6 +1
>> append(p1,p2)
Frac poly exp from input 1 to output:
#1 : s^2.2 +s^1.5
#2 : 0
Frac poly exp from input 2 to output:
#1 : 0
#2 : s^0.6 +1
```

cancel_zero_coef (private)

Removes the nul coefficents and their corresponding orders from a frac_poly_exp.

Syntax

```
res = cancel_zero_coef(fpe)
```

Description

`Cancel_zero_coef` removes the zero terms of af fpe. Zero terms can appear after arithmetic operations.
`Cancel_zero_coef` is private and called at the end of the function clean, after the call to 'sort' function. The order are suposed to be sorted and unique. The fpe is supposed to be of dimension one.
Due to numerical round-offs, a coefficient is supoposed to be zero if it is less than eps (see help eps).

Arguments

Argument in:
fpe: frac_poly_exp object

Argument out:
res: frac_poly_exp object

Example

```
>>P
s^6 + 0 s^5 + 3 s^3.7 + 2 s^3 - s^-0.2
>>Q = cancel_zero_coef(P)
s^6 + 3 s^3.7 + 2 s^3 -s^-0.2
```

char

Char converts frac_poly_exp object to a string, which makes it ready for display

Syntax

```
    st = char(fpe)
```

Arguments

Argument in*:*
fpe: frac_poly_exp object

Argument out*:*
st: string

Example

```
>>pol=frac_poly_exp([1,2,1],[3,2,0]);
>>char(pol)
s^3 + 2 s^2 +1

>>pol_multi=frac_poly_exp({[1,4]            [2,5];[1,4]
[2,5]},{[0.2,0] [3.5 1.5]; [0.2,0] [3.5 1.5]});
>>char(pol_multi)
's^0.2 + 4 '    '2 s^3.5 + 5 s^1.5 '
's^0.2 + 4 '    '2 s^3.5 + 5 s^1.5 '

>>pol_nan=frac_poly_exp(nan);
char(pol_nan)
NaN
```

clean

Sorts the orders of an frac_poly_exp in descending order, adds coefficients with the same order and removes all zeros coefficents

Syntax

```
res = clean(fpe)
```

Description

Clean uses the private functions `sort` and `cancel_zero_coef`. Due to round-offs, two orders are assumed to be equal if their difference is less than eps.

Arguments

Argument in:
fpe: frac_poly_exp

Argument out:
res: frac_poly_exp

Example

```
>>pol=frac_poly_exp([1 2 3 0 2 -1],[0.5 0.2 6 12 3
0.2]);
s^0.5 + 2s^0.2 + 3s^6 + 0s^12 + 2s^3 - 1s^0.2
>>clean(pol)
3 s^6 + 2 s^3 +s^0.5 +s^0.2
```

coef

Returns the coefficients of a frac_poly_exp object.

Syntax

```
c = coef(fpe)
```

Arguments

Argument in:
fpe: frac_poly_exp object

Argument out:
c: coefficient of P (cell)

Example

```
>> p=frac_poly_exp([1 2 3 2 -1],[0.5 0.2 6 3 0.2]);
>> c=coef(p)
[1x4 double]
>> c{1}
3     2     1     1
```

commensurate

Computes de step order of a frac_poly_exp.

Syntax

```
    [New_order,Comm_order]= commensurate(fpe)
```

Arguments

Argument in:
fpe : frac_poly_exp object

Argument out:
New_order: the integer order of T
Comm_order: the commensurate order (scalar)

Example

```
>> p
3 s^6 + 2 s^3 +s^0.5 +s^0.2
>> [comm_ord, new_ord]=commensurate(p)
new_ord =
    [1x4 double]
comm_ord =
    0.1000
>> new_ord{1}
ans =
    60    30     5     2

>>Pol
Frac poly exp from input 1 to output:
#1 : s^1.2 +s^0.8 +s^0.4 +s^0.2
#2 : s^5.6 +s^1.5 +s^0.9 +s^0.21
Frac poly exp from input 2 to output:
#1 : s^1.2 +s^0.8 +s^0.4 +s^0.2
#2 : s^5.6 +s^1.5 +s^0.9 +s^0.21

>>[ comm_order, new_order]=commensurate(pol);
>>comm_order
step_order = 0.0100
>>new_order
new_order =
{  [120   80   40   20] [560  150   90   21]  }
{  [120   80   40   20] [560  150   90   21]  }
```

display

Prints the frac_poly_exp object on the screen..

Syntax

```
display(fpe)
```

Arguments

Argument in:
fpe : frac_poly_exp object

Argument out:
none

Example

```
>>display(pol)
s^3 + 2 s^2 +1

>>display(pol_multi)
Frac poly exp from input 1 to output:
#1 : s^0.2 + 4
#2 : 2 s^3.5 + 5 s^1.5
Frac poly exp from input 2 to output:
#1 : s^0.2 + 4
#2 : 2 s^3.5 + 5 s^1.5

>>display(pol_nan) display :
The object is NaN

>>display(pol_vide) display :
The object is empty
```

enlarge

This function used on a an frac_poly_exp of dimension 1, duplicates it in order to form an m by n polynomial composed of this frac_poly_exp.

Syntax

```
res=enlarge(fpe,m,n)
```

Arguments

Argument in :
fpe : frac_poly_exp object
n, m : scalar

Argument out :
res : frac_poly_exp object

Example

```
>>pol1=frac_poly_exp([1,2,4],[3,2,0]);
s^3 + 2 s^2 + 4

>>pol3=enlarge(pol1,2,2)
Frac poly exp from input 1 to output:
#1 : s^3 + 2 s^2 + 4
#2 : s^3 + 2 s^2 + 4
Frac poly exp from input 2 to output:
#1 : s^3 + 2 s^2 + 4
#2 : s^3 + 2 s^2 + 4
```

eq

Tests the equality between two frac_poly_exp. Tested objects must have the same size.

Syntax

```
fpe1 == fpe2
bool=eq(fpe1, fpe2)
```

Arguments

Argument in :
fpe1 : frac_poly_exp
fpe2 : frac_poly_exp

Argument out :
bool : Boolean

Description

`eq(fpe1, fpe2)` compares each element of `fpe1` with the corresponding element of `fpe2`, and returns a logical 1 (true) if `fpe1` and `fpe2` are equal, or logical 0 (false) if they are unequal.

`eq(fpe1, fpe2)` is called for the syntax `fpe1 = = fpe2`

Examples

```
>>pol1=frac_poly_exp([1,2,4],[3,2,0]);
>>pol1bis=frac_poly_exp([2,4,8],[3,2,0]);
>>eq(pol1,pol1)
ans = 1
>>eq(pol1,pol1bis)
ans = 0

>>eq(pol_multi,pol_multi)
ans =  [1]    [0]
       [1]    [0]
```

frac_poly_exp

Creates a frac_poly_exp object (constructor) containing the following attributes:
- coef
- order

Syntax

```
Sys=frac_poly_exp()
Sys=frac_poly_exp(a)
Sys=frac_poly_exp(fpe)
Sys=frac_poly_exp(coef, order)
```

Description

frac_poly_exp() creates an empty explicit polynomial.
frac_poly_exp(a), where a is a scalar (or a NaN) creates an explicit polynomial. a is considered as the coefficient argument; the order is forced to 0. If a is a cell array of scalars frac_poly_exp(a) creates a multidimensional polynomial.
frac_poly_exp(fpe) returns fpe.
frac_poly_exp(coef, order) creates an explicit polynomial, the first row vector being the coefficients vector and the second the orders vector. If coef and order are cell array of row vector (same size) frac_poly_exp creates a multidimensional frac_poly_exp object.

Arguments

Arguments in:
a: scalar or cell array of scalar
fpe: frac_poly_exp object
coef: coefficients of the frac_poly_exp (row vector or cell array of row vector)
order: orders of the frac_poly_exp (row vector or cell array of row vector)
Coef and order must have the same size.

Argument out:
Sys: explicit fractional polynomial (frac_poly_exp object)

Examples

```
>>P_empty=frac_poly_exp
The object is empty
>>P_NaN=frac_poly_exp(nan)
The object is NaN
>>P_NaN2=frac_poly_exp([1,1],[NaN,0])
The object is NaN
>>P_scalar=frac_poly_exp(3)
3
>>P_scalar2=frac_poly_exp({[1] [2]})
Frac poly exp from input 1 to output:
#1 : 1
#2 : 2
```

```
>>P_copy = frac_poly_exp(P_scalar2)
Frac poly exp from input 1 to output:
#1 : 1
#2 : 2
>>P=frac_poly_exp([1,1],[0.2,0])
s^0.2 +1
>>P_multi=frac_poly_exp({[1,4][2,5]},{[0.2,0][3.5
1.5]})
Frac poly exp from input 1 to output:
#1 : s^0.2 + 4
#2 : 2 s^3.5 + 5 s^1.5
```

get

Query object attributes.

Syntax

```
res = get(fpe)
res = get(fpe,propertyName)
```

Description

`get(fpe)` returns all attibutes of the object and their current values.
`get(fpe,'coef')` returns the coefficients vector of the object identified by fpe.
`get(fpe,'order')` returns the orders vector of the object identified by fpe.
`get(fpe,’All’)` is the same than `get(fpe)`

Arguments

Argument in :
fpe: frac_poly_exp object
propertyName: string

Argument out :
res: row vector containing the coefficients or the orders of fpe, or cell of two row vector containing both of them.

Example

```
>>pol=frac_poly_exp([1,2,1],[3,2,0]);
>>coef=get(pol,'coef');
coef = [1 2 1]
>>coef=get(pol,'order');
coef = [3 2 0]

>>one_arg=get(pol);
one_arg ={[1 2 1], [3 2 0]}

>>all=get(pol,’All’);
all ={[1 2 1], [3 2 0]}
```

horzcat

Concatenate arrays of frac_poly_exp horizontally

Syntax

```
    fpe = [fpe1 fpe2 ...]
    fpe = horzcat(fpe1, fpe2, ...)
```

Description

`fpe = fpe(fpe1, fpe2, ...)` concatenates horizontally fpe1, fpe2, and so on. All fpe object in the argument list must have the same number of rows. horzcat concatenates N-dimensional fpe objects along the second dimension. The first and remaining dimensions must match.

Arguments

Argument in :
fpe1, fpe2: frac_poly_exp object

Argument out :
fpe: frac_poly_exp object

Examples

```
>>pol1=frac_poly_exp([1,2,4],[3,2,0]);
>>pol1bis=frac_poly_exp([2,4,8],[3,2,0]);

>>horzcat(pol1,pol1bis)
Frac poly exp from input 1 to output:
#1 : s^3 + 2 s^2 + 4
#2 : 2 s^3 + 4 s^2 + 8

>>horzcat(pol1,pol_nan,pol1)
The object is NaN

>>horzcat(pol1,pol_vide,pol1)
Frac poly exp from input 1 to output:
#1 : s^3 + 2 s^2 + 4
#2 : s^3 + 2 s^2 + 4
```

iscomplex

Determines whether frac_poly_exp has complex coefficients and/or orders. By convention, the NaN and the Empty polynomial are not complex.

Syntax

```
bool=iscomplex(fpe)
```

Arguments

Argument in :
fpe : frac_poly_exp

Argument out :
bool : boolean

Example

```
>>pol1=frac_poly_exp([1,2,4],[3,2,0]);
>>polcmplx1=frac_poly_exp([1,2+i,4],[3,2,0]);
>>polcmplx2=frac_poly_exp([1,2,4],[3+i,2,0]);

>>iscomplex(pol1)
ans= 0

>>iscomplex(polcmplx1)
ans= 1

>>iscomplex(polcmplx2)
ans= 1
```

isempty

Determines whether frac_poly_exp has empty coefficients and order.

Syntax

```
bool=isempty(fpe)
```

Arguments

Argument in :
fpe : frac_poly_exp object

Argument out :
bool : boolean

Example

```
>> fpe
s^2.2 +s^1.5
>> isempty(fpe)
0
>> test=frac_poly_exp
>> isempty(test)
1
```

isnan

Determines if the frac_poly_exp is nan

Syntax

```
    bool=isnan(fpe)
```

Arguments

Argument in :
fpe : frac_poly_exp object

Argument out :
bool : boolean

Example

```
>> fpe
s^2.2 +s^1.5
>> isnan(fpe)
0
>> test=frac_poly_exp(nan)
>> isnan(test)
1
```

ldivide

Creates a farctionnal transfer function (frac_tf) by dividing a frac_poly_exp by another frac_poly_exp.

Syntax

```
    res = ldivide(fpe1,fpe2)
    res = fpe1 \ fpe2
```

Description

Creates a fractionnal transfer function (frac_tf) with :
- numerator   : fpe2
- denominator : fpe1

`ldivide(fpe1, fpe2)` is called when the syntax `fpe1 \ fpe2` is used.

Arguments

Argument in :
fpe1, fpe2 : frac_poly_exp object

Argument out :
res : frac_tf object

Example

```
>> pol1
s^3 + 12 s + 4

>> pol1bis
3 s^3 + 12 s + 2

>> ldivide(pol1,pol1bis)
 transfer function :
( 3 s^3 + 12 s + 2  )
---------------------
 ( s^3 + 12 s + 4  )
```

match_same_order (private)

Adds the terms with the same order in a frac_poly_exp.

Syntax

```
    res = match_same_order(fpe)
```

Description

Check the orders of a frac_poly_exp and if some of them are alike adds their related coefficients.
This function is private and called at the end of the function sort.
The orders are suposed to be sorted and the frac_poly_exp of dimension one. Two coefficients are equal if their difference is less than eps().

Arguments

Argument in:
fpe: frac_poly_exp object

Argument out:
res: frac_poly_exp object

Example

```
>>fpe
s^6 + 2 s^0.2 + 3 s^6 + 2 s^3 - s^0.2
>>res = match_same_order(fpe)
4 s^6 + 2 s^3 +s^0.2
```

minus

Realizes the opration fpe1 - fpe2 and then calls clean. The ojects must have the same size.

Syntax

```
    res = minus(fpe1,fpe2)
    res = fpe1 – fpe2
```

Arguments

Argument in:
fpe1,fpe2: frac_poly_exp objects

Argument out:
res: frac_poly_exp object

Example

```
>>ans1=minus(pol1,pol1)
The object is empty

>>(3 s^3 + 12 s + 2) - (s^3 + 2 s^2 + 4)
2 s^3 - 2 s^2 + 12 s - 2

>>(3 s^3 + 12 s + 2) - (s^3 + 12 s^ + 4)
2 s^3 - 2
```

mpower

Realizes the opration (fpe)^r.

Syntax

```
res = fpe^r
res = mpower(fpe,r)
```

Description

`r` has to be an integer, if not mpower will take round(r) as argument.

Arguments

Argument in:
fpe: frac_poly_exp objects
r : integer

Argument out:
res: frac_poly_exp object

Example

```
>>pol1
(s^3 + 2 s^2 + 4)^2
>>mpower(pol1,2)
s^6 + 4 s^5 + 4 s^4 + 8 s^3 + 16 s^2 + 16
```

mtimes

Realizes the operation fpe1 x fpe2, when fpe1 and fpe2 can be matrix of appropriate dimensions : number of columns of fpe1 = number of rows of fpe2.

Syntax

```
    res = fpe1*fpe2
    res = fpe1*k
    res = k*fpe1
    res = mtimes(fpe1,fpe2)
    res = mtimes (k, fpe1)
    res = mtimes (fpe1, k)
```

Description

If `fpe1` is a frac_poly_exp of dimension `n x m` and `fpe2` is a frac_poly_exp of dimension `m x k`, then `res` is a frac poy_exp of dimension `n x k`.

Arguments

Argument in:
fpe1, fpe2: frac_poly_exp objects.
k : matrix of double.

Argument out:
res: frac_poly_exp object

Example

```
%(3 s^3 + 12 s + 2) * (s^3 + 2 s^2 + 4)
>>ans1=mtimes(pol1bis,pol1)
3 s^6 + 6 s^5 + 12 s^4 + 38 s^3 + 4 s^2 + 48 s + 8

% 4 * (s^3 + 2 s^2 + 4)
>>ans1=mtimes(2,pol1)
(4 s^3 + 8 s^2 + 16)
```

multi (private)

This function deals with multi-dimensionnal systems.

Syntax

```
varargout = multi(fun_name, nbr, varargin)
```

Description

This function deals with multi-dimensionnal systems. It catchs the size of the system and fills-in cells with the results of the function called on system of dimension one.
This function is private and is always called when a function is callde with multi-dimensionnal polynomials.

Arguments

Argument in:

fun_name	: Name of the function called
nbr	: Number of argout expected
nbrout_fpe	: Number of fpe expected in the argout
varargin	: Contain the arguments needed by the function "fun_name"

Argument out:
varargout : result depends on the function called

ne

Test for unequality

Syntax

```
    fpe1 ~= fpe1
    bool=ne(fpe1, fpe2)
```

Description

ne(fpe1, fpe2) is called when the syntax fpe1 ~= fpe2 is used.
fpe1 ~= fpe2 compares each element of fpe1 with the corresponding element of fpe2, and returns a logical 1 (true) if fpe1 and fpe2 are unequal, or logical 0 (false) if they are equal.
By convention, two NaN polynomials are not equal and two Empty polynomials are equal.

Arguments

Argument in :
fpe1 : frac_tf or frac_poly_exp
fpe2 : frac_tf or frac_poly_exp

Argument out :
bool : boolean

Example

```
>>pol1=frac_poly_exp([1,2,4],[3,2,0]);
>>pol1bis=frac_poly_exp([2,4,8],[3,2,0]);
>>ne(pol1,pol1)
ans = 0
>>ne(pol1,pol1bis)
ans = 1

>>ne(pol_multi,pol_multi)
ans =  [0]    [1]
       [0]    [1]
```

order

Returns the orders of a frac_poly_exp object. It calls `get(fpe,'order')`.

Syntax

```
    o = order(fpe)
```

Arguments

Argument in:
fpe: frac_poly_exp object

Argument out:
o: row vetor or cell of row vectors.

Example

```
>> p=frac_poly_exp([1 2 3 2 -1],[0.5 0.2 6 3 0.2]);
>> o=order(p)
[1x4 double]
>> o{1}
6.0000    3.0000    0.5000    0.2000
```

parallel

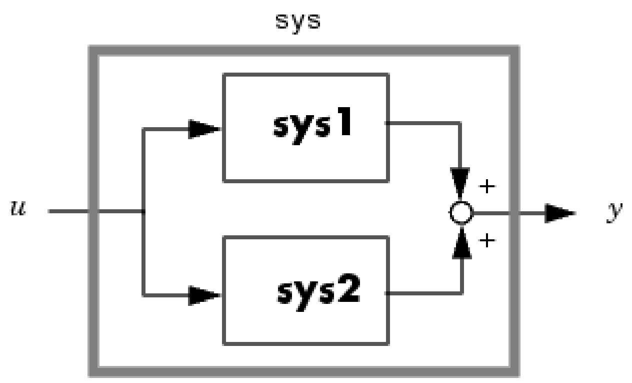
Parallel connection of two frac_poly_exp models

Syntax

```
    sys = parallel(sys1,sys2,in1,in2,out1,out2)
    sys = parallel(sys1,sys2)
```

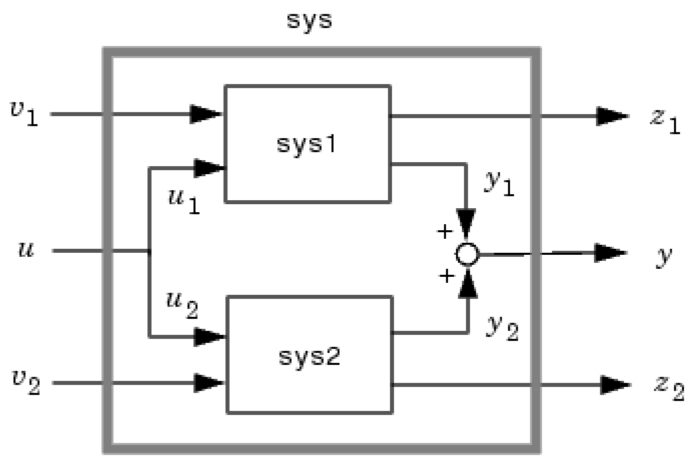
Description

parallel connects two frac_poly_exp models in parallel.
sys = parallel(sys1,sys2) forms the basic parallel connection shown below.



This command is equivalent to the direct addition
sys = sys1 + sys2

sys = parallel(sys1,sys2,in1,in2,out1,out2) forms the more general parallel connection.



The index vectors in1 and in2 specify which inputs u_1 of sys1 and which inputs u_2 of sys2 are connected. Similarly, the index vectors out1 and out2 specify which outputs y_1 of sys1 and which outputs y_2 of sys2 are summed up. The resulting model sys has $[v_1, u, v_2]$ as inputs and $[z_1, y, z_2]$ as outputs.

Arguments

Argument in :

sys1: fractional explicit polynomial (frac_poly_exp object)
sys2: fractional explicit polynomial (frac_poly_exp object)
in1: fractional explicit polynomial (vector)
in2: fractional explicit polynomial (vector)
out1: fractional explicit polynomial (vector)
out2: fractional explicit polynomial (vector)

Argument out :
sys : fractional explicit polynomial (frac_poly_exp object)

Example

```
>> sys
s^2.2 +s^1.5
>> parallel(sys,sys)
2 s^2.2 + 2 s^1.5

>> sys1
Frac poly exp from input 1 to output:
#1 : s^2.2 +s^1.5
#2 : s^2.2 +s^1.5
#3 : s^2.2 +s^1.5
#4 : s^2.2 +s^1.5
Frac poly exp from input 2 to output:
#1 : s^2.2 +s^1.5
#2 : s^2.2 +s^1.5
#3 : s^2.2 +s^1.5
#4 : s^2.2 +s^1.5
Frac poly exp from input 3 to output:
#1 : s^2.2 +s^1.5
#2 : s^2.2 +s^1.5
#3 : s^2.2 +s^1.5
#4 : s^2.2 +s^1.5
Frac poly exp from input 4 to output:
#1 : s^2.2 +s^1.5
#2 : s^2.2 +s^1.5
#3 : s^2.2 +s^1.5
#4 : s^2.2 +s^1.5
>> sys2
Frac poly exp from input 1 to output:
#1 : s^2.2 +s^1.5
#2 : s^2.2 +s^1.5
#3 : s^2.2 +s^1.5
Frac poly exp from input 2 to output:
#1 : s^2.2 +s^1.5
#2 : s^2.2 +s^1.5
#3 : s^2.2 +s^1.5
Frac poly exp from input 3 to output:
#1 : s^2.2 +s^1.5
#2 : s^2.2 +s^1.5
#3 : s^2.2 +s^1.5
```

```
Frac poly exp from input 4 to output:
#1 : s^2.2 +s^1.5
#2 : s^2.2 +s^1.5
#3 : s^2.2 +s^1.5

>> parallel(sys1,sys2,[1 2],[2 3],[3 4],[1 2])
Frac poly exp from input 1 to output:
#1 : s^2.2 +s^1.5
#2 : s^2.2 +s^1.5
#3 : s^2.2 +s^1.5
#4 : s^2.2 +s^1.5
#5 : 0
Frac poly exp from input 2 to output:
#1 : s^2.2 +s^1.5
#2 : s^2.2 +s^1.5
#3 : s^2.2 +s^1.5
#4 : s^2.2 +s^1.5
#5 : 0
Frac poly exp from input 3 to output:
#1 : s^2.2 +s^1.5
#2 : s^2.2 +s^1.5
#3 : 2 s^2.2 + 2 s^1.5
#4 : 2 s^2.2 + 2 s^1.5
#5 : s^2.2 +s^1.5
Frac poly exp from input 4 to output:
#1 : s^2.2 +s^1.5
#2 : s^2.2 +s^1.5
#3 : 2 s^2.2 + 2 s^1.5
#4 : 2 s^2.2 + 2 s^1.5
#5 : s^2.2 +s^1.5
Frac poly exp from input 5 to output:
#1 : 0
#2 : 0
#3 : s^2.2 +s^1.5
#4 : s^2.2 +s^1.5
#5 : s^2.2 +s^1.5
Frac poly exp from input 6 to output:
#1 : 0
#2 : 0
#3 : s^2.2 +s^1.5
#4 : s^2.2 +s^1.5
#5 : s^2.2 +s^1.5
```

plus

Realizes the opration fpe1 + fpe2. The ojects must have the same dimension.

Syntax

```
    res = fpe1 + fpe2
    res = plus(fpe1,fpe2)
```

Arguments

Argument in :
fpe1, fpe2 : frac_poly_exp objects.

Argument out :
res : frac_poly_exp object.

Example

```
>>(3 s^3 + 12 s + 2) + (s^3 + 2 s^2 + 4)
>>ans1=plus(pol1bis,pol1)
4 s^3 + 2 s^2 + 12 s + 6

>>ans2=pol_nan+pol_nan
The object is NaN
```

rdivide

Creates a transfer function

Syntax

```
    res = rdivide(fpe1,fpe2)
    res = fpe1 / fpe2
```

Description

Creates the transfer function with :
- numerator : fpe1
- denominator : fpe2

`rdivide(fpe1, fpe2)` is called when the syntax `fpe1 / fpe2 is used.`

Arguments

Argument in :
fpe1, fpe2 : frac_poly_exp object

Argument out :
res : frac_tf object

Example

```
>> pol1
s^3 + 12 s + 4

>> pol1bis
3 s^3 + 12 s + 2

>> rdivide(pol1,pol1bis)
 transfer function :
( s^3 + 12 s + 4  )
----------------------
 ( 3 s^3 + 12 s + 2  )
```

roots roots_s is not implemented yet

Computes the roots in s^{γ} and in s of a frac_poly_exp where γ is the commensurate order.

Syntax

```
    [roots,roots_s,comm_step]=roots(fpe)
```

Arguments

Argument in :
fpe : frac_poly_exp object.

Argument out :

roots	: roots in s^{γ} of fpe (cells) where γ is the commensurate order
comm_step	: commensurate order (complex vector)

roots_s : roots in s^{γ} of fpe (cells)

Example

```
>> fpe
s^4.5 +s^1.5
>> [r,ev,eo]=roots(fpe)
r =
    {1x1 cell}
ev =
    [22x1 double]
eo =
    [1.5000]

>> r{1}{1}
Empty matrix: 1-by-0

>> ev{1}
       0
        0 + 1.0000i
        0 - 1.0000i

>> eo{1}
1.5000
```

series

Series connection of two frac_poly_exp objects.

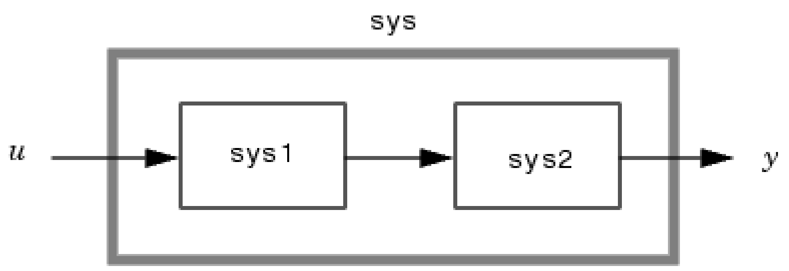
Syntax

```
    sys = series(sys1,sys2,in1,in2,out1,out2)
    sys = series(sys1,sys2)
```

Description

`series` connects two frac_poly_exp models in series.

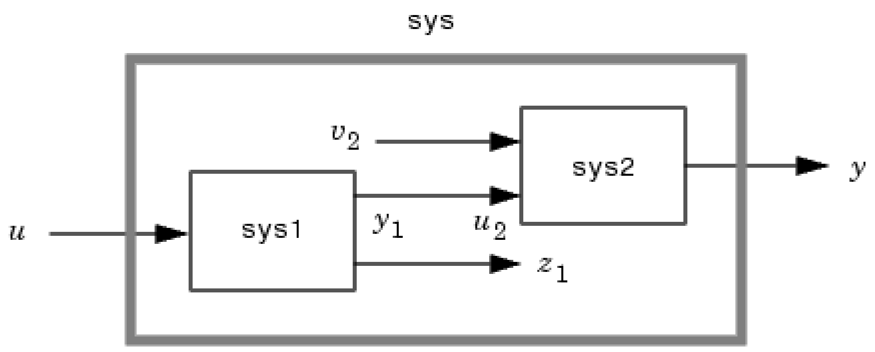
`sys = series(sys1,sys2)` forms the basic series connection shown below.



This command is equivalent to the direct multiplication

```
    sys = sys2 * sys1
```

`sys = series(sys1,sys2,outputs1,inputs2)` forms the more general series connection.



The index vectors outputs1 and inputs2 indicate which outputs y_1 of sys1 and which inputs u_2 of sys2 should be connected. The resulting model sys has u as input and y as output.

Arguments

Argument in :

sys1: fractional explicit polynomial (frac_poly_exp object)
sys2: fractional explicit polynomial (frac_poly_exp object)
in1: fractional explicit polynomial (vector)
in2: fractional explicit polynomial (vector)
out1: fractional explicit polynomial (vector)
out2: fractional explicit polynomial (vector)

Argument out :
sys : fractional explicit polynomial (frac_poly_exp object)

Example

```
>> sys
s^2.2 +s^1.5
>> series(sys,sys)
s^4.4 + 2 s^3.7 +s^3

>> sys1
Frac poly exp from input 1 to output:
#1 : s^2.2 +s^1.5
#2 : s^2.2 +s^1.5
#3 : s^2.2 +s^1.5
#4 : s^2.2 +s^1.5
Frac poly exp from input 2 to output:
#1 : s^2.2 +s^1.5
#2 : s^2.2 +s^1.5
#3 : s^2.2 +s^1.5
#4 : s^2.2 +s^1.5
Frac poly exp from input 3 to output:
#1 : s^2.2 +s^1.5
#2 : s^2.2 +s^1.5
#3 : s^2.2 +s^1.5
#4 : s^2.2 +s^1.5
Frac poly exp from input 4 to output:
#1 : s^2.2 +s^1.5
#2 : s^2.2 +s^1.5
#3 : s^2.2 +s^1.5
#4 : s^2.2 +s^1.5
>> sys2
Frac poly exp from input 1 to output:
#1 : s^2.2 +s^1.5
#2 : s^2.2 +s^1.5
#3 : s^2.2 +s^1.5
Frac poly exp from input 2 to output:
#1 : s^2.2 +s^1.5
#2 : s^2.2 +s^1.5
#3 : s^2.2 +s^1.5
Frac poly exp from input 3 to output:
#1 : s^2.2 +s^1.5
#2 : s^2.2 +s^1.5
#3 : s^2.2 +s^1.5
```

```
Frac poly exp from input 4 to output:
#1 : s^2.2 +s^1.5
#2 : s^2.2 +s^1.5
#3 : s^2.2 +s^1.5
>> series(sys1,sys2,[3 4],[1 2])
Frac poly exp from input 1 to output:
#1 : 2 s^4.4 + 4 s^3.7 + 2 s^3
#2 : 2 s^4.4 + 4 s^3.7 + 2 s^3
#3 : 2 s^4.4 + 4 s^3.7 + 2 s^3
#4 : 2 s^4.4 + 4 s^3.7 + 2 s^3
Frac poly exp from input 2 to output:
#1 : 2 s^4.4 + 4 s^3.7 + 2 s^3
#2 : 2 s^4.4 + 4 s^3.7 + 2 s^3
#3 : 2 s^4.4 + 4 s^3.7 + 2 s^3
#4 : 2 s^4.4 + 4 s^3.7 + 2 s^3
Frac poly exp from input 3 to output:
#1 : 2 s^4.4 + 4 s^3.7 + 2 s^3
#2 : 2 s^4.4 + 4 s^3.7 + 2 s^3
#3 : 2 s^4.4 + 4 s^3.7 + 2 s^3
#4 : 2 s^4.4 + 4 s^3.7 + 2 s^3
Frac poly exp from input 4 to output:
#1 : 2 s^4.4 + 4 s^3.7 + 2 s^3
#2 : 2 s^4.4 + 4 s^3.7 + 2 s^3
#3 : 2 s^4.4 + 4 s^3.7 + 2 s^3
#4 : 2 s^4.4 + 4 s^3.7 + 2 s^3
```

set

Allows to modify attrbutes of frac_poly_exp objects.

Syntax

```
set(fo,property,value)
```

Description

`fpe=set(fpe,'PropertyName',PropertyValue,...)` sets the named properties to the specified values on the object(s) identified by fpe.

Arguments

Argument in :
fpe : frac_poly_exp object
PropertyName: string
PropertyValue: property value depends on the property

PropertyName	PropertyValue
Coef	Vector of double
order	Vector of double

Argument out :
fo : frac_poly_exp object

Example

```
>> fpe
s^2.2 +s^1.5
>> set(fpe,'coef',[2 2])
2 s^2.2 + 2 s^1.5
>> set(fpe,'order',[1.2 0.1])
s^1.2 +s^0.1
```

size

The function returns the dimensions of frac_poly_exp objects.

Syntax

```
    d = size(sys)
    [m,n] = size(sys)
```

Arguments

Argument in :
sys: fpe objects

Argument out :
d: vector.
n,m: scalar

Example

```
>> pol
s^0.6 +1
>> d=size(pol)
d =
     1     1
>> [n,m]=size(pol)
n =
     1
m =
     1
```

sort

Sorts the orders of an explicit fractional polynomial in the descending order and then adds the coefficients with the same order by calling the private function `match_same_order`.

Syntax

```
    res = sort(fpe)
```

Arguments

Argument in:
fpe: frac_poly_exp object

Argument out:
res: frac_poly_exp object

Example

```
>> pol
s^0.5 + 2s^0.2 + 3s^6 + 0s^12 + 2s^3 - 1s^0.2

sort(pol)
3 s^6 + 2 s^3 - 2 s^0.5 +s^0.2
```

subsasgn

Allows the affectation of different attributes of an fpe. This function is called when the following syntax are used:

Syntax

```
    fpe(1,2) = fp
    fpe.coef = [1 3]
    res=subsasgn(fpe,index,value)
```

Description

index is a strucuture containing two attributes : type and subs. It can be of two types : « **.** » or « **( )** ».
- if index.type = '.' the value is the attribute specified by index.subs.
  Ex : fpe.order = [1 2 4 2]

- if index.type = '()' and index.subs=[n m]. The element n, m of frac_poly_exp object is assigned by the new frac_poly_exp specified by value.
  Ex : fpe(1,2) = pol

Arguments

Argument in :
fpe : frac_poly_exp object
index : structure
value : the new value depending on the attribute to change.

Argument out :
res : fpe object

Example

```
>>pol
s^3 + 2 s^2 +1

>>pol.coef={[2 1 2]}
2 s^3 +s^2 + 2

>>pol2
Frac poly exp from input 1 to output:
#1 : 2 s^3 +s^2 + 2
#2 : 2 s^3 +s^2 + 2
Frac poly exp from input 2 to output:
#1 : 2 s^3 +s^2 + 2
#2 : 2 s^3 +s^2 + 2

pol_multi(1,2)=frac_poly_exp([3 4 5],[3 2 1])
Frac poly exp from input 1 to output:
#1 : 2 s^3 +s^2 + 2
#2 : 3 s^3 + 4 s^2 + 5 s
Frac poly exp from input 2 to output:
```

```
#1 : 2 s^3 +s^2 + 2
#2 : 2 s^3 +s^2 + 2
```

subsref

Quick access to the differents attribute of an fpe.

Syntax

```
res = fpe(1,2)
res = fpe.coef
res=subsref(fpe1,index,value)
```

Description

index is a strucuture containing two attributes : type and subs. It can be of two type : « **.** » or « **( )** ». Indeed :

- if index.type = '.' the result is the attribute specified by index.subs.
  Ex : fpe.order

- if index.type = '()' and index.subs=[n m]. The element n, m of frac_poly_exp is returned.
  Ex : fpe(1,2)

Arguments

Argument in :
fpe : frac_poly_exp object
index : structure.

Argument out :
res : fpe object

Example

```
>>pol
s^3 + 2 s^2 +1

>>pol.coef
{ 1 2 1 }

>>pol2
Frac poly exp from input 1 to output:
#1 : s^3 + 2 s^2 +1
#2 : 2s^3 + 2 s^2 +1
Frac poly exp from input 2 to output:
#1 : 3s^3 + 2 s^2 +1
#2 : 4s^3 + 2 s^2 +1

>>pol_multi(1,2)
2s^3 + 2 s^2 +1
```

times

Realize a term by term multiplication. The ojects must have the same size.

Syntax

```
    res = fpe1.*fpe2
    res = times(fpe1,fpe2)
```

Arguments

Argument in:
fpe1, fpe2: frac_poly_exp objects

Argument out:
res: frac_poly_exp object

Example

```
%(3 s^3 + 12 s + 2) * (s^3 + 2 s^2 + 4)
>>ans1=times(pol1bis,pol1)
3 s^6 + 6 s^5 + 12 s^4 + 38 s^3 + 4 s^2 + 48 s + 8
```

transpose

Transposition of frac_poly_exp.

Syntax

```
    res = fpe’
    res = transpose(fpe)
```

Arguments

Argument in:
fpe : frac_poly_exp objects

Argument out:
res: frac_poly_exp object

Example

```
>> pol3
Frac poly exp from input 1 to output:
#1 : s^3 + 2 s^2 + 4
#2 : 3 s^3 + 12 s + 2
Frac poly exp from input 2 to output:
#1 : s^3 + 2 s^2 + 4
#2 : 3 s^3 + 12 s + 2

D=transpose(pol3)
Frac poly exp from input 1 to output:
#1 : s^3 + 2 s^2 + 4
#2 : s^3 + 2 s^2 + 4
Frac poly exp from input 2 to output:
#1 : 3 s^3 + 12 s + 2
#2 : 3 s^3 + 12 s + 2
```

uminus

Multiplies a frac_poly_exp by (-1)

.

Syntax

```
    Res = -fpe
    res = uminus(fpe)
```

Arguments

Argument in:

fpe: frac_poly_exp objects

Argument out:

res: frac_poly_exp object

Example

```
>> pol1
s^3 + 2 s^2 + 4
>> uminus(pol1)
-s^3 - 2 s^2 – 4
```

vertcat

Concatenates fpe objects vertically

Syntax

```
    fpe = [fpe1; fpe2; ...]
    fpe = vertcat(fpe1, fpe2, ...)
```

Description

`fpe = vertcat(fpe1, fpe2, ...)` vertically concatenates fpe1, fpe2, and so on. All frac_poly_exp objects in the argument list must have the same number of rows.
`vertcat` concatenates N-dimensional fpe objects along the first dimension. The second and remaining dimensions must match.

Arguments

Argument in :
fpe1, fpe2: frac_poly_exp objects

Argument out :
fpe: fractional frac_poly_exp objects

Examples

```
>> pol
s^2.2 +s^1.5
>> fpecat=vertcat(pol, pol, pol)
Frac poly exp from input 1 to output:
s^2.2 +s^1.5
Frac poly exp from input 2 to output:
s^2.2 +s^1.5
Frac poly exp from input 3 to output:
s^2.2 +s^1.5
```

frac_poly_imp class

Attributes

Attribute name	Description	Value
fpe	Polynomials of the frac_poly_imp object	Cell Nu*Ny of frac_poly_exp vector
Imp_order	Implicit orders of the frac_poly_imp object	Cell Nu*Ny of positive double vector

List of functions

char
clean
display
enlarge
eq
fpe
get
horzcat
iscomplex
isempty
isnan
ldivide
minus
mpower
mtimes
multi (private)
ne
plus
rdivide
roots
set
size
subsasgn
subsref
times
transpose
uminus
vertcat

char

Char converts frac_poly_imp object to a string, which makes it ready for display.

Syntax

```
    st = char(fpi)
```

Arguments

Argument in:
fpi: frac_poly_imp object

Argument out:
st: string

Example

```
>>pol=frac_poly_imp([1,2,1],[3,2,0],3);
>>char(pol)
(s^3 + 2 s^2 +1)^3

>>pol_multi=frac_poly_imp({[1,4] [2,5];[1,4]
[2,5]},{[0.2,0] [3.5 1.5]; [0.2,0] [3.5
1.5]},{2,3 ; 4,5});
>>char(pol_multi)
'(s^0.2 + 4)^2 '    '(2 s^3.5 + 5 s^1.5)^3 '
'(s^0.2 + 4)^4 '    '(2 s^3.5 + 5 s^1.5)^5 '

>>pol_nan=frac_poly_imp(nan);
char(pol_nan)
NaN
```

display

Displays the frac_poly_imp object on the screen..

Syntax

```
    display(fpi)
```

Arguments

Argument in:
fpi : frac_poly_imp object

Argument out:
none

Example

```
>>display(pol)
(s^3 + 2 s^2 +1)^3

>>display(pol_multi)
Frac poly exp from input 1 to output:
#1 : (s^0.2 + 4 )^2
#2 : (2 s^3.5 + 5 s^1.5 )^3
Frac poly exp from input 2 to output:
#1 : (s^0.2 + 4 )^4
#2 : (2 s^3.5 + 5 s^1.5)^5

>>display(pol_nan) display :
The object is NaN

>>display(pol_vide) display :
The object is empty
```

enlarge

This function used on a frac_poly_imp of dimension 1, duplicates it in order to form an *m x n* matrix of frac_poly_imp objects.

Syntax

```
res=enlarge(fpi,m,n)
```

Arguments

Argument in :
fpi : frac_poly_imp object
n, m : scalar

Argument out :
res : frac_poly_imp object

Example

```
>>pol1=frac_poly_imp([1,2,4],[3,2,0],3);
(s^3 + 2 s^2 + 4)^3

>>pol3=enlarge(pol1,2,2)
Frac poly exp from input 1 to output:
#1 : (s^3 + 2 s^2 + 4)^3
#2 : (s^3 + 2 s^2 + 4)^3
Frac poly exp from input 2 to output:
#1 : (s^3 + 2 s^2 + 4)^3
#2 : (s^3 + 2 s^2 + 4)^3
```

eq

Tests the equality between two frac_poly_imp objects. Tested objects must have the same size.

Syntax

```
    fpe1 == fpe2
    bool = eq(fpe1, fpe2)
```

Arguments

Argument in :
fpe1 : frac_poly_imp
fpe2 : frac_poly_imp

Argument out :
bool : Boolean

Description

`eq(fpe1, fpe2)` compares each element of `fpe1` with the corresponding element of `fpe2`, and returns a logical 1 (true) if `fpe1` and `fpe2` are equal, or logical 0 (false) if they are unequal.

Examples

```
>>pol1=frac_poly_imp([1,2,4],[3,2,0],2);
>>pol2=frac_poly_imp([2,4,8],[3,2,0],2);
>>pol3=frac_poly_imp([1,2,4],[3,2,0],3);

>>eq(pol1,pol1)
ans = 1

>>eq(pol1,pol2)
ans = 0

>>eq(pol1,pol3)
ans = 0

>>eq(pol_multi,pol_multi)
ans =  [1]    [0]
       [1]    [0]
```

fpe

Returns the frac_poly_exp attribute of a frac_poly_imp object.

Syntax

```
    fpe = fpe(fpi)
```

Arguments

Argument in:
fpi: frac_poly_imp object

Argument out:
fpe: frac_poly_exp object

Example

```
>> p=frac_poly_imp([1 2 3 2 -1],[0.5 0.2 6 3 0.2],3);
>> fpe(pol)
3 s^6 + 2 s^3 +s^0.5 +s^0.2
```

frac_poly_imp

Creates a frac_poly_imp object (constructor) containing the following attributes:
- fpe
- imp_order

Syntax

```
Sys=frac_poly_imp()
Sys=frac_poly_imp(a)
Sys=frac_poly_imp(fpe)
Sys=frac_poly_imp(fpi)
Sys=frac_poly_imp(fpe,a)
Sys=frac_poly_imp(coef, order)
Sys=frac_poly_imp(coef, order, a)
```

Description

`frac_poly_imp()` creates an empty frac_poly_imp object.
`frac_poly_imp(a)`, where `a` is a scalar (or a NaN) creates an implicit polynomial. `a` is used as the coefficient argument to create a frac_poly_exp; the implicit order is set to 1. If `a` is a cell array of scalars `frac_poly_imp(a)` creates a multidimensional polynomial.
`frac_poly_imp(fpe)` creates a frac_poly_imp where `fpe` is a frac_poly_exp and the implicit order is set to 1.
`frac_poly_imp(fpi)` returns fpi.
`frac_poly_imp(coef, order)` creates an implicit polynomial, the two row vectors are used to create the frac_poly_exp and the implicit order is forced to 1. If `coef` and `order` are cell arrays of row vector (same size) `frac_poly_imp` creates a multidimensional frac_poly_imp object.
`frac_poly_imp(coef, order, a)` creates an implicit polynomial, the two row vectors are used to create the frac_poly_exp and the implicit order is set to `a`. If `coef, order` and `a` are cell array (same size) `frac_poly_imp` creates a multidimensional frac_poly_imp object.

Arguments

Arguments in:
a: scalar or cell array of scalar
fpe: frac_poly_exp object
fpi: frac_poly_imp object
coef: coefficients of the frac_poly_exp (row vector or cell array of row vector)
order: orders of the frac_poly_exp (row vector or cell array of row vector)
Coef and *order* must have the same size.

Argument out:
Sys: frac_poly_imp object

Examples

```
>>P_empty=frac_poly_imp
```

```
The object is empty

>>P_NaN=frac_poly_imp(nan)
The object is NaN

>>P_NaN2=frac_poly_imp([1,1],[NaN,0])
The object is NaN

>>P_scalar=frac_poly_imp(3)
( 3 )

>>P_scalar2=frac_poly_imp({[1] [2]})
Frac poly imp from input 1 to output:
#1 : ( 1 )
#2 : ( 2 )


>>P_fpe = frac_poly_imp(P_scalar2)
Frac poly imp from input 1 to output:
#1 : ( 1 )
#2 : ( 2 )

>>P_fpe2 = frac_poly_imp(P_scalar2,3)
Frac poly imp from input 1 to output:
#1 : ( 1 )^3
#2 : ( 2 )^3


>>P=frac_poly_imp([1,1],[0.2,0])
( s^0.2 +1 )

>>P_multi=frac_poly_imp({[1,4][2,5]},{[0.2,0][3.5
1.5]})

Frac poly imp from input 1 to output:
#1 : ( s^0.2 + 4  )
#2 : ( 2 s^3.5 + 5 s^1.5 )

>>P2=frac_poly_imp([1,1],[0.2,0],3)
( s^0.2 +1 )^3

>>P_multi2=frac_poly_imp({[1,4][2,5]},{[0.2,0][3.5
1.5]}, {2,3})
Frac poly imp from input 1 to output:
#1 : ( s^0.2 + 4  )^2
#2 : ( 2 s^3.5 + 5 s^1.5 )^3

P_copy=frac_poly_imp(P_multi2)
Frac poly imp from input 1 to output:
#1 : ( s^0.2 + 4  )^2
#2 : ( 2 s^3.5 + 5 s^1.5 )^3
```

get

Queries object attributes.

Syntax

```
res = get(fpi)
res = get(fpi,attribute)
```

Description

`get(fpi)` returns a cell containign all the attributes of the object and their current values.
`get(fpi,'fpe')` returns the fpe composing the fpi.
`get(fpi,'imp_order')` returns the implicit order of the fpi.
`get(fpi, 'All')` is the same than `get(fpi)`

Arguments

Argument in :
fpi: frac_poly_imp object
property: string

Argument out :
res: - *fpe* : frac_poly_exp object
- *a* : integer
- *{ fpe, a }* : cell array

Example

```
>>pol=frac_poly_imp([1,2,1],[3,2,0],3);
>> get(pol,'fpe');
s^3 + 2 s^2 +1
>> get(pol,'imp_order');
3

>>one_arg=get(pol);
[1x1 frac_poly_exp]    [3]

>>all=get(pol,’All’);
[1x1 frac_poly_exp]    [3]
```

horzcat

Concatenates arrays of frac_poly_imp horizontally.

Syntax

```
    fpi = [fpe1 fpe2 ...]
    fpi = horzcat(fpe1, fpe2, ...)
```

Description

`fpi = horzcat(fpe1, fpe2, ...)` concatenates horizontally `fpe1`, `fpe2`, and so on. All fpi objects in the argument list must have the same number of rows.
`horzcat` concatenates N-dimensional fpe objects along the second dimension. The first and remaining dimensions must match.

Arguments

Argument in :
fpe1, fpe2: frac_poly_imp object

Argument out :
fpe: frac_poly_imp object

Examples

```
>>pol1=frac_poly_imp([1,2,4],[3,2,0],3);
>>pol1bis=frac_poly_imp([2,4,8],[3,2,0],5);

>>horzcat(pol1,pol1bis)
Frac poly exp from input 1 to output:
#1 : (s^3 + 2 s^2 + 4)^3
#2 : (2 s^3 + 4 s^2 + 8)^5

>>horzcat(pol1,pol_nan,pol1)
The object is NaN

>>horzcat(pol1,pol_vide,pol1)
Frac poly exp from input 1 to output:
#1 : (s^3 + 2 s^2 + 4)^3
#2 : (s^3 + 2 s^2 + 4)^3
```

iscomplex

Determines whether frac_poly_imp has complex coefficients and/or orders. By convention, NaN and Empty frac_poly_imp objects are not complex.

Syntax

```
bool=iscomplex(fpi)
```

Arguments

Argument in :
fpi : frac_poly_imp

Argument out :
bool : boolean

Example

```
>>pol1=frac_poly_imp([1,2,4],[3,2,0],3);
>>polcmplx1=frac_poly_imp([1,2+i,4],[3,2,0],3);
>>polcmplx2=frac_poly_imp([1,2,4],[3+i,2,0],3);
>>polcmplx3=frac_poly_imp([1,2,4],[3,2,0],3+i);


>>iscomplex(pol1)
ans= 0

>>iscomplex(polcmplx1)
ans= 1

>>iscomplex(polcmplx2)
ans= 1

>>iscomplex(polcmplx3)
ans= 1
```

isempty

Determines whether a frac_poly_imp object is empty or not.

Syntax

```
    bool=isempty(fpi)
```

Arguments

Argument in :
fpi : frac_poly_imp object

Argument out :
bool : boolean

Example

```
>> fpi
(s^2.2 +s^1.5 )^3
>> isempty(fpi)
0
>> test=frac_poly_imp
>> isempty(test)
1
```

isnan

Determines if frac_poly_imp is nan

Syntax

```
    bool=isnan(fpi)
```

Arguments

Argument in :
fpi : frac_poly_imp object

Argument out :
bool : boolean

Example

```
>> fpi
(s^2.2 +s^1.5 )^3
>> isnan(fpi)
0
>> test=frac_poly_imp(nan)
>> isnan(test)
1
```

ldivide

Creates a fractionnal transfer function (frac_tf) by dividing a frac_poly_exp by another frac_poly_exp.

Syntax

```
    fpe1 \ fpe2
    res=ldivide(fpe1,fpe2)
```

Description

Creates a fractionnal transfer function (frac_tf) with :
- numerator : fpe2
- denominator : fpe1

Arguments

Argument in :
fpe1, fpe2 : frac_poly_imp object

Argument out :
res : frac_tf object

Example

```
>> pol1
(s^3 + 12 s + 4)^2

>> pol1bis
(3 s^3 + 12 s + 2 )^3

>> ldivide(pol1,pol1bis)
 transfer function :
( 3 s^3 + 12 s + 2  )^3
----------------------
 ( s^3 + 12 s + 4  )^2
```

minus

Realizes the opration fpe1 - fpe2 and calls clean method. The ojects must have the same size.
Both implicit orders must be equal to 1.

Syntax

```
    res = fpi1 – fpi2
    res = minus(fpe1,fpe2)
```

Arguments

Argument in:
fpe1,fpe2: frac_poly_imp objects

Argument out:
res: frac_poly_imp object

Example

```
>>ans1=minus(pol1,pol1)
The object is empty

>>(3 s^3 + 12 s + 2)^1 - (s^3 + 2 s^2 + 4)^1
(2 s^3 - 2 s^2 + 12 s – 2)^1
```

mpower

With fpi = (fpe)^k, mpower realizes res = (mpower(fpe,r))^k.

Syntax

```
    Res = fpi^r
    res = mpower(fpi,r)
```

Arguments

Argument in:
fpe: frac_poly_imp objects
r : integer

Argument out:
res: frac_poly_imp object

Example

```
>>pol1
(s^3 + 2 s^2 + 4)^1
>>mpower(pol1,2)
(s^6 + 4 s^5 + 4 s^4 + 8 s^3 + 16 s^2 + 16 )^1
```

mtimes

Multiplies two frac_poly_imp objects of appropriates dimensions.

Syntax

```
res = fpi1 * fpi2
res = k * fpi1
res = fpi1 * k
res = mtimes(fpi1,fpi2)
res = mtimes (k, fpe1)
res = mtimes (fpe1, k)
```

Description

If `fpe1` is a frac_poly_imp of dimension `n x m` and `fpe2` is a frac_poly_imp of dimension `m x k`, then `res` is a frac poy_imp of dimension `n x k`.

Arguments

Argument in:
fpe1, fpe2: frac_poly_imp objects

Argument out:
res: frac_poly_imp object

Example

```
%(3 s^3 + 12 s + 2)^1 * (s^3 + 2 s^2 + 4)^1
>>ans1=mtimes(pol1bis,pol1)
(3 s^6 + 6 s^5 + 12 s^4 + 38 s^3 + 4 s^2 + 48 s + 8)^1
```

multi (private)

This function deals with multi-dimensionnal systems.

Syntax

```
varargout = multi(fun_name, nbr, varargin)
```

Description

This function deals with multi-dimensionnal systems. It catchs the size of the system and fills-in cells with the results of the function `fun_name` call on system of dimension one.
This function is private and is always called when a function is called with multi-dimensionnal frac_poly_imp objects.

Arguments

Argument in:

fun_name	: Name of the function called
nbr	: Number of argout expected
nbrout_fpi	: Number of fpe expected in the argout
varargin	: Contain the arguments needed by the function "fun_name"

Argument out:

varargout : result depends on the function called

ne

Test for unequality

Syntax

```
    fpe1 ~= fpe2
    bool = ne(fpe1, fpe2)
```

Description

fpe1 ~= fpe2 compares each element of fpe1 with the corresponding element of fpe2, and returns a logical 1 (true) if fpe1 and fpe2 are unequal, or logical 0 (false) if they are equal.
By convention, two NaN polynomials are not equal and two Empty polynomials are equal.

Arguments

Argument in :
fpe1 : frac_poly_imp
fpe2 : frac_poly_imp

Argument out :
bool : boolean

Example

```
>>pol1=frac_poly_imp([1,2,4],[3,2,0],2);
>>pol2=frac_poly_imp([2,4,8],[3,2,0],2);
>>pol3=frac_poly_imp([1,2,4],[3,2,0],3);

>>ne(pol1,pol1)
ans = 0
>>ne(pol1,pol2)
ans = 1

>>ne(pol1,pol3)
ans = 1

>>ne(pol_multi,pol_multi)
ans =  [0]    [1]
       [0]    [1]
```

plus

Adds two frac_poly_exp objects of the same dimensions.
The implicit order of each frac_poly_imp must be equal to 1.

Syntax

```
    res=fpe1+fpe2
    res=plus(fpe1,fpe2)
```

Arguments

Argument in :
fpe1, fpe2 : frac_poly_imp objects.

Argument out :
res : frac_poly_imp object.

Example

```
>>(3 s^3 + 12 s + 2)^1 + (s^3 + 2 s^2 + 4)^1

>> plus(pol1bis,pol1)
(4 s^3 + 2 s^2 + 12 s + 6)^1

>> pol_nan+pol_nan
The object is NaN
```

rdivide

Creates a transfer function (frac_tf object)

Syntax

```
res=fpe1/fpe2
res=rdivide(fpe1,fpe2)
```

Description

Creates the transfer function with :
- numerator : fpe1
- denominator : fpe2

Arguments

Argument in :
fpe1, fpe2 : frac_poly_imp object

Argument out :
res : frac_tf object

Example

```
>> pol1
(s^3 + 12 s + 4 )^2

>> pol1bis
(3 s^3 + 12 s + 2 )^3

>> rdivide(pol1,pol1bis)
 transfer function :
( s^3 + 12 s + 4  )^2
----------------------
 ( 3 s^3 + 12 s + 2  )^3
```

set

Allows to modify attributes of frac_poly_imp object.

Syntax

```
    set(fpi,property,value)
    fpi=set(fpi,property,value)
```

Description

`set(fpi,’fpe’,fpe)` sets the frac_poly_exp object contained in the frac_poly_imp `fpi` to the value `fpe.`
`set(fpi,’imp_order’,a)` sets the imp_order (integer) contained in the frac_poly_imp `fpi` to the value `a.`

PropertyName	**PropertyValue**
fpe	Frac_poly_exp object
imp_order	integer

Arguments

Argument in :
fpi : frac_poly_imp object
PropertyName: string
PropertyValue: property value depends on the property

Argument out :
fpi : frac_poly_imp object

Example

```
>> fpe
(s^2.2 +s^1.5)^2
>> set(fpe,'fpe',frac_poly_exp([2 2],[3,1])
(2s^3 +2s^1)^2
>> set(fpe,'imp_order', 4)
(2s^3 +2s^1)^4
```

size

Returns the size of frac_poly_imp object.

Syntax

```
    d = size(sys)
    [m,n] = size(sys)
```

Arguments

Argument in :
sys: fpi objects

Argument out :
d: vector.
n,m: scalar

Example

```
>> pol
s^0.6 +1()^3
>> d=size(pol)
d =
     1     1
>> [n,m]=size(pol)
n =
     1
m =
   1
```

subsasgn

Allows the affectation of different attributes of an fpe.

Syntax

```
fpi(1,2) = fp
fpi.imp_order = 3
res=subsasgn(fpi,index,value)
```

Description

index is a strucuture containing two attributes : type and subs. It can be of two types : « **.** » or « **( )** ».

- if index.type = '.' the value is the attribute specified by index.subs.

  Ex : fpe.imp_order = 3

- if index.type = '()' and index.subs=[n m]. The element n, m of frac_poly_exp object is assigned by the new frac_poly_exp specified by value.

  Ex : fpe(1,2) = pol

Arguments

Argument in :

f*pi* : frac_poly_imp object
index : structure
value : the new value depending on the attribute to change.

Argument out :

res : fpi object

Example

```
>>pol
(s^3 + 2 s^2 +1)^3
>>pol.imp_order = { 6 }
(2 s^3 +s^2 + 2)^6

>>pol2
Frac poly exp from input 1 to output:
#1 : (2 s^3 +s^2 + 2)^3
#2 : (2 s^3 +s^2 + 2)^3
Frac poly exp from input 2 to output:
#1 : (2 s^3 +s^2 + 2)^3
#2 : (2 s^3 +s^2 + 2)^3

pol_multi(1,2)=frac_poly_imp([3 4 5],[3 2 1],3)
Frac poly exp from input 1 to output:
#1 : (2 s^3 +s^2 + 2)^3
#2 : (3 s^3 + 4 s^2 + 5 s)^5
Frac poly exp from input 2 to output:
#1 : (2 s^3 +s^2 + 2)^3
#2 : (2 s^3 +s^2 + 2)^3
```

subsref

Quick access to different attributes of an frac_poly_imp.

Syntax

```
res = fpi(1,2)
res = fpi.imp_order
res=subsref(fpi,index,value)
```

Description

index is a strucuture containing two attributes : type and subs. It can be of two type : « **.** » or « **( )** ». Indeed :
- if index.type = '.' the result is the attribute specified by index.subs.
  Ex : fpe.imp_order

- if index.type = '()' and index.subs=[n m]. The element n, m of frac_poly_exp is returned.
  Ex : fpe(1,2)

Arguments

Argument in :
fpi : frac_poly_imp object
index : structure.

Argument out :
res : frac_poly_imp object

Example

```
>>pol
(s^3 + 2 s^2 +1 ^)^3

>>pol.imp_order
{ 3 }

>>pol2
Frac poly exp from input 1 to output:
#1 : (s^3 + 2 s^2 +1)^1
#2 : (2s^3 + 2 s^2 +1)^2
Frac poly exp from input 2 to output:
#1 : (3s^3 + 2 s^2 +1)^3
#2 : (4s^3 + 2 s^2 +1)^4

>>pol_multi(1,2)
(2s^3 + 2 s^2 +1)^2
```

times

Realize a term by term multiplication. This function is called when .* operator is used.
The ojects must have the same size.

Syntax

```
res = fpe1 .* fpe2
res = times(fpe1,fpe2)
```

Arguments

Argument in:
fpe1, fpe2: frac_poly_imp objects

Argument out:
res: frac_poly_imp object

Example

```
%(3 s^3 + 12 s + 2)^1 * (s^3 + 2 s^2 + 4)^1
>>ans1=times(pol1bis,pol1)
(3 s^6 + 6 s^5 + 12 s^4 + 38 s^3 + 4 s^2 + 48 s + 8)^1
```

transpose

Matrix transposition of frac_poly_imp object.

Syntax

```
res = transpose(fpi)
res = fpi'
```

Arguments

Argument in:
fpe : frac_poly_imp objects

Argument out:
res: frac_poly_imp object

Example

```
>> pol3
Frac poly imp from input 1 to output:
#1 : (s^3 + 2 s^2 + 4)^3
#2 : (3 s^3 + 12 s + 2)^3
Frac poly imp from input 2 to output:
#1 : (s^3 + 2 s^2 + 4)^3
#2 : (3 s^3 + 12 s + 2)^2

D=transpose(pol3)
Frac poly imp from input 1 to output:
#1 : (s^3 + 2 s^2 + 4)^3
#2 : (s^3 + 2 s^2 + 4)^3
Frac poly imp from input 2 to output:
#1 : (3 s^3 + 12 s + 2)^3
#2 : 3 s^3 + 12 s + 2)^3
```

uminus

Multiplies a frac_poly_imp by (-1)
The implicit order must be equal to 1.

Syntax

```
    res = – fpi
    res = uminus(fpi)
```

Arguments

Argument in:
fpe: frac_poly_imp objects

Argument out:
res: frac_poly_imp object

Example

```
>> pol1
(s^3 + 2 s^2 + 4 )^1
>> uminus(pol1)
(-s^3 - 2 s^2 – 4)^1
```

vertcat

Concatenates frac_poly_imp objects vertically

Syntax

```
    fpi = [fpe1; fpe2; ...]
    fpe = vertcat(fpe1, fpe2, ...)
```

Description

`fpi = vertcat(fpe1, fpe2, ...)` vertically concatenates `fpe1`, `fpe2`, … All frac_poly_imp objects in the argument list must have the same number of rows.
`vertcat` concatenates N-dimensional `fpi` objects along the first dimension. The second and remaining dimensions must match.

Arguments

Argument in :
fpe1, fpe2: frac_poly_imp objects

Argument out :
fpi: fractional frac_poly_imp objects

Examples

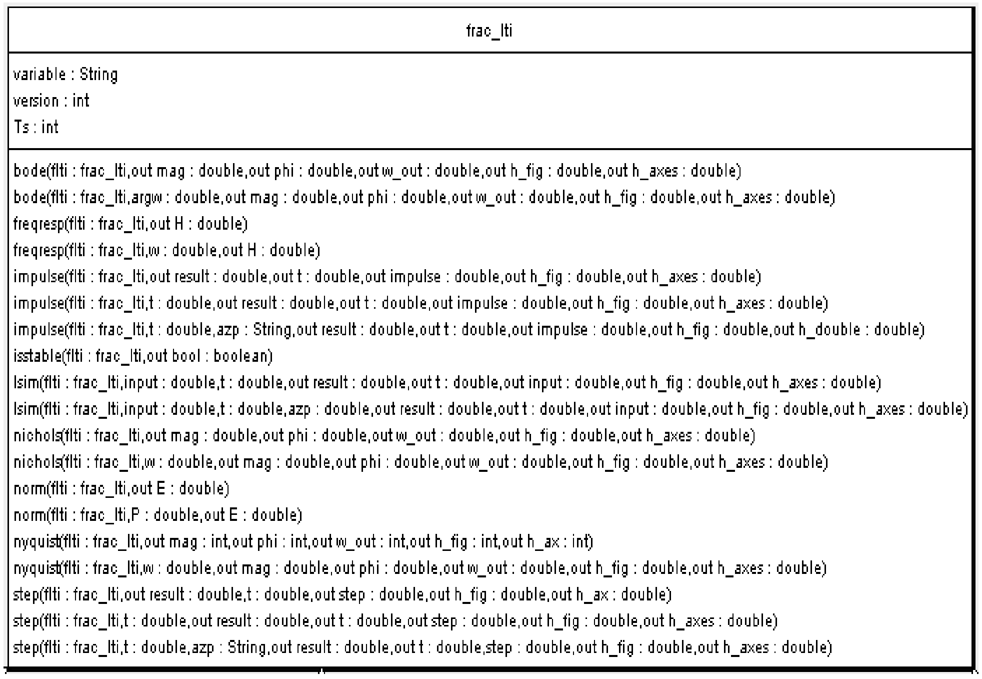
```
>> pol
(s^2.2 +s^1.5)^3
>> fpecat=vertcat(pol, pol, pol)
Frac poly exp from input 1 to output:
(s^2.2 +s^1.5)^3
Frac poly exp from input 2 to output:
(s^2.2 +s^1.5)^3
Frac poly exp from input 3 to output:
(s^2.2 +s^1.5)^3
```

Class frac_lti

Attributes

Attribute name	Description	Value
variable	Variable of frac_lti object	String can be 's', 'z', 'p', 'q'
version	Version of the CRONE toolbox objects	string
Ts	Sample time of discrete polynomial	Double, 0 if it is a continuous model
N	Number poles and zeros for a poles and zero approximation	Double (integer value)
band	Frequency band for a poles and zero approximation	1*2 double

Function’s list



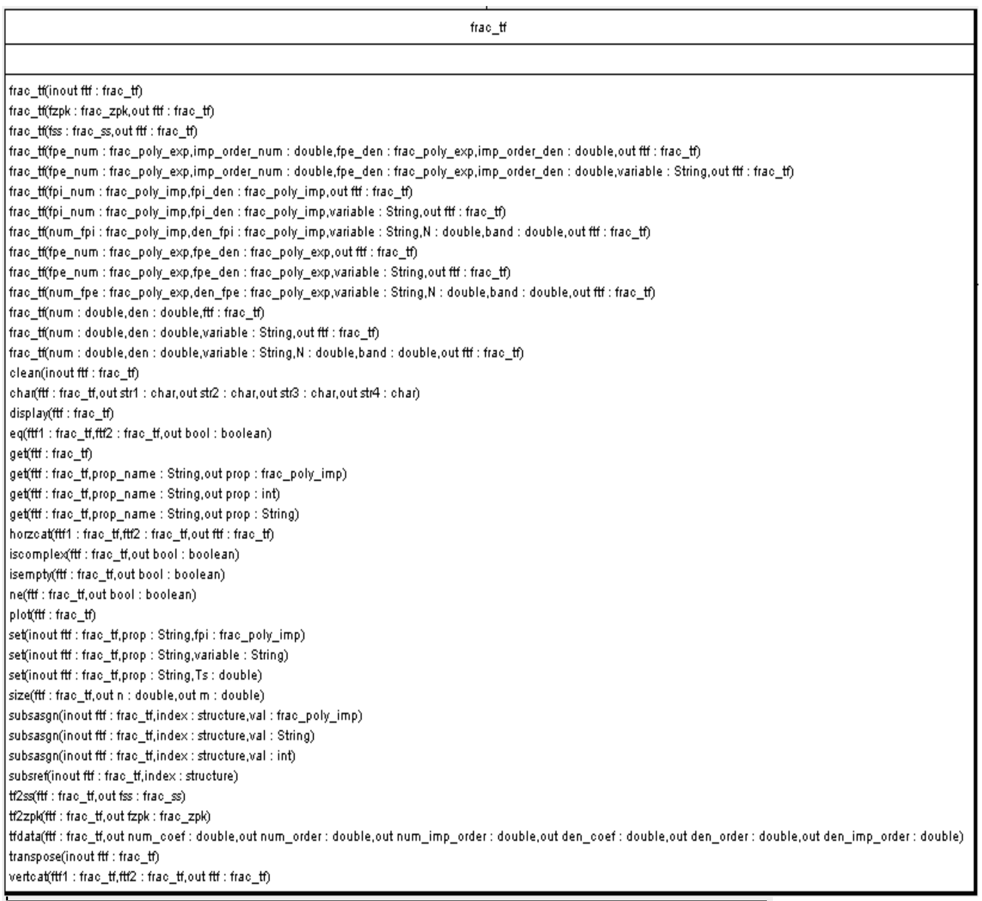
```
frac_lti

variable : String
version : int
Ts : int

bode(flti : frac_lti,out mag : double,out phi : double,out w_out : double,out h_fig : double,out h_axes : double)
bode(flti : frac_lti,argw : double,out mag : double,out phi : double,out w_out : double,out h_fig : double,out h_axes : double)
freqresp(flti : frac_lti,out H : double)
freqresp(flti : frac_lti,w : double,out H : double)
impulse(flti : frac_lti,out result : double,out t : double,out impulse : double,out h_fig : double,out h_axes : double)
impulse(flti : frac_lti,t : double,out result : double,out t : double,out impulse : double,out h_fig : double,out h_axes : double)
impulse(flti : frac_lti,t : double,azp : String,out result : double,out t : double,out impulse : double,out h_fig : double,out h_double : double)
isstable(flti : frac_lti,out bool : boolean)
lsim(flti : frac_lti,input : double,t : double,out result : double,out t : double,out input : double,out h_fig : double,out h_axes : double)
lsim(flti : frac_lti,input : double,t : double,azp : double,out result : double,out t : double,out input : double,out h_fig : double,out h_axes : double)
nichols(flti : frac_lti,out mag : double,out phi : double,out w_out : double,out h_fig : double,out h_axes : double)
nichols(flti : frac_lti,w : double,out mag : double,out phi : double,out w_out : double,out h_fig : double,out h_axes : double)
norm(flti : frac_lti,out E : double)
norm(flti : frac_lti,P : double,out E : double)
nyquist(flti : frac_lti,out mag : int,out phi : int,out w_out : int,out h_fig : int,out h_ax : int)
nyquist(flti : frac_lti,w : double,out mag : double,out phi : double,out w_out : double,out h_fig : double,out h_axes : double)
step(flti : frac_lti,out result : double,t : double,out step : double,out h_fig : double,out h_ax : double)
step(flti : frac_lti,t : double,out result : double,out t : double,out step : double,out h_fig : double,out h_axes : double)
step(flti : frac_lti,t : double,azp : String,out result : double,out t : double,step : double,out h_fig : double,out h_axes : double)
```

Class frac_tf

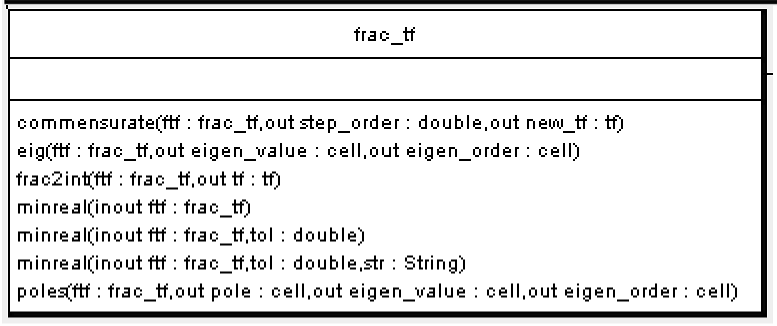
Attributes

Attribute name	Description	Value
num	Numerator of the frac_tf object	Frac_poly_imp matrix Nu*Ny
den	Denominator of the frac_tf object	Frac_poly_imp matrix Nu*Ny

Function's list



```
frac_tf

frac_tf(inout ftf : frac_tf)
frac_tf(fzpk : frac_zpk,out ftf : frac_tf)
frac_tf(fss : frac_ss,out ftf : frac_tf)
frac_tf(fpe_num : frac_poly_exp,imp_order_num : double,fpe_den : frac_poly_exp,imp_order_den : double,out ftf : frac_tf)
frac_tf(fpe_num : frac_poly_exp,imp_order_num : double,fpe_den : frac_poly_exp,imp_order_den : double,variable : String,out ftf : frac_tf)
frac_tf(fpi_num : frac_poly_imp,fpi_den : frac_poly_imp,out ftf : frac_tf)
frac_tf(fpi_num : frac_poly_imp,fpi_den : frac_poly_imp,variable : String,out ftf : frac_tf)
frac_tf(num_fpi : frac_poly_imp,den_fpi : frac_poly_imp,variable : String,N : double,band : double,out ftf : frac_tf)
frac_tf(fpe_num : frac_poly_exp,fpe_den : frac_poly_exp,out ftf : frac_tf)
frac_tf(fpe_num : frac_poly_exp,fpe_den : frac_poly_exp,variable : String,out ftf : frac_tf)
frac_tf(num_fpe : frac_poly_exp,den_fpe : frac_poly_exp,variable : String,N : double,band : double,out ftf : frac_tf)
frac_tf(num : double,den : double,ftf : frac_tf)
frac_tf(num : double,den : double,variable : String,out ftf : frac_tf)
frac_tf(num : double,den : double,variable : String,N : double,band : double,out ftf : frac_tf)
clean(inout ftf : frac_tf)
char(ftf : frac_tf,out str1 : char,out str2 : char,out str3 : char,out str4 : char)
display(ftf : frac_tf)
eq(ftf1 : frac_tf,ftf2 : frac_tf,out bool : boolean)
get(ftf : frac_tf)
get(ftf : frac_tf,prop_name : String,out prop : frac_poly_imp)
get(ftf : frac_tf,prop_name : String,out prop : int)
get(ftf : frac_tf,prop_name : String,out prop : String)
horzcat(ftf1 : frac_tf,ftf2 : frac_tf,out ftf : frac_tf)
iscomplex(ftf : frac_tf,out bool : boolean)
isempty(ftf : frac_tf,out bool : boolean)
ne(ftf : frac_tf,out bool : boolean)
plot(ftf : frac_tf)
set(inout ftf : frac_tf,prop : String,fpi : frac_poly_imp)
set(inout ftf : frac_tf,prop : String,variable : String)
set(inout ftf : frac_tf,prop : String,Ts : double)
size(ftf : frac_tf,out n : double,out m : double)
subsasgn(inout ftf : frac_tf,index : structure,val : frac_poly_imp)
subsasgn(inout ftf : frac_tf,index : structure,val : String)
subsasgn(inout ftf : frac_tf,index : structure,val : int)
subsref(inout ftf : frac_tf,index : structure)
tf2ss(ftf : frac_tf,out fss : frac_ss)
tf2zpk(ftf : frac_tf,out fzpk : frac_zpk)
tfdata(ftf : frac_tf,out num_coef : double,out num_order : double,out num_imp_order : double,out den_coef : double,out den_order : double,out den_imp_order : double)
transpose(inout ftf : frac_tf)
vertcat(ftf1 : frac_tf,ftf2 : frac_tf,out ftf : frac_tf)
```



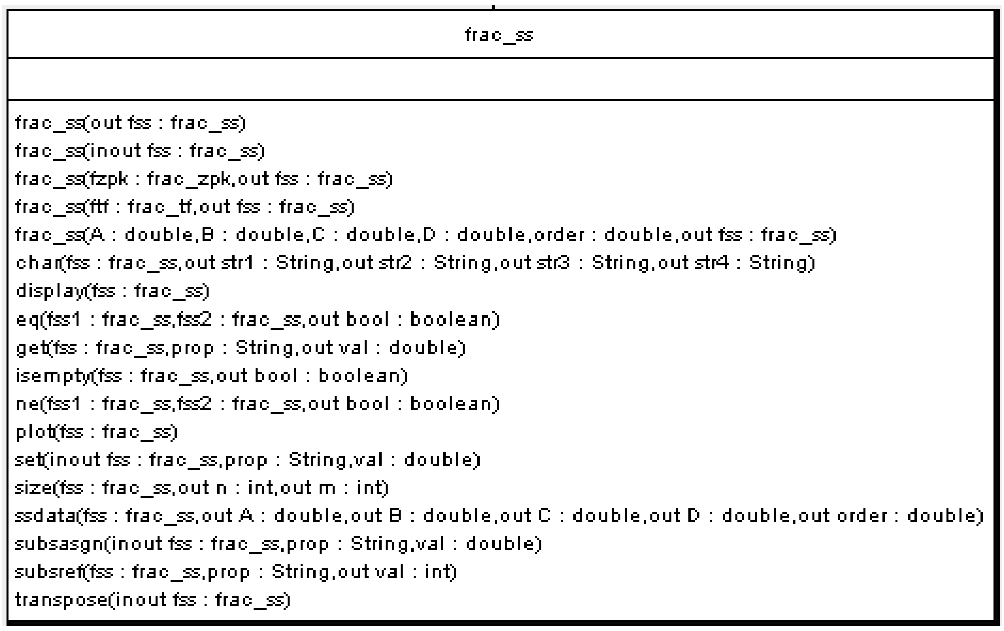
```
frac_tf

commensurate(ftf : frac_tf,out step_order : double,out new_tf : tf)
eig(ftf : frac_tf,out eigen_value : cell,out eigen_order : cell)
frac2int(ftf : frac_tf,out tf : tf)
minreal(inout ftf : frac_tf)
minreal(inout ftf : frac_tf,tol : double)
minreal(inout ftf : frac_tf,tol : double,str : String)
poles(ftf : frac_tf,out pole : cell,out eigen_value : cell,out eigen_order : cell)
```

Class frac_ss

Attributes

Attribute name	Description	Value
A	Variable of abstract_frac_poly object	Double matrix Nx* Nx
B	Version of the CRONE toolbox objects	Double matrix Nx*Nu
C	Sample time of discrete polynomial	Double matrix Ny*Nx
D	Number poles and zeros for a poles and zero approximation	Double matrix Ny*Nu
order	Frequency band for a poles and zero approximation	Double scalar

Function's list



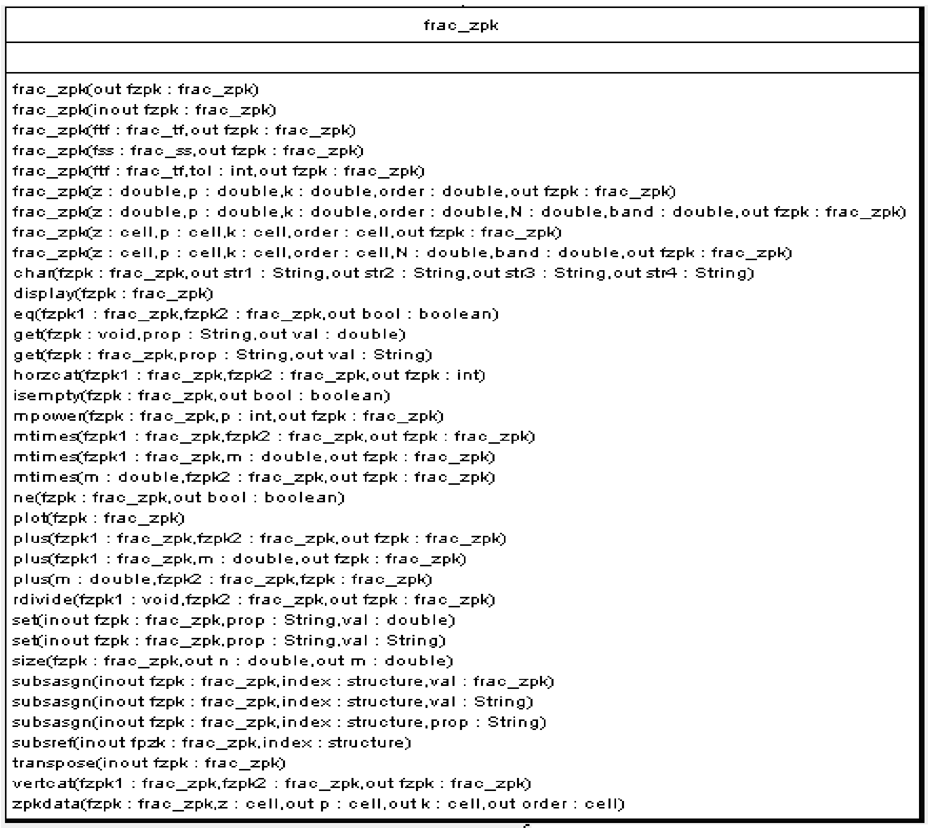
```
frac_ss

frac_ss(out fss : frac_ss)
frac_ss(inout fss : frac_ss)
frac_ss(fzpk : frac_zpk,out fss : frac_ss)
frac_ss(ftf : frac_tf,out fss : frac_ss)
frac_ss(A : double,B : double,C : double,D : double,order : double,out fss : frac_ss)
char(fss : frac_ss,out str1 : String,out str2 : String,out str3 : String,out str4 : String)
display(fss : frac_ss)
eq(fss1 : frac_ss,fss2 : frac_ss,out bool : boolean)
get(fss : frac_ss,prop : String,out val : double)
isempty(fss : frac_ss,out bool : boolean)
ne(fss1 : frac_ss,fss2 : frac_ss,out bool : boolean)
plot(fss : frac_ss)
set(inout fss : frac_ss,prop : String,val : double)
size(fss : frac_ss,out n : int,out m : int)
ssdata(fss : frac_ss,out A : double,out B : double,out C : double,out D : double,out order : double)
subsasgn(inout fss : frac_ss,prop : String,val : double)
subsref(fss : frac_ss,prop : String,out val : int)
transpose(inout fss : frac_ss)
```

Class frac_zpk

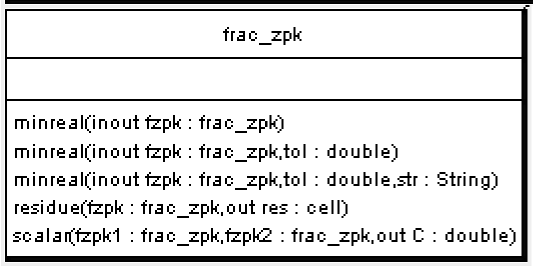
Attributes

Attribute name	Description	Value
Eig_zero	Zeros of the zpk form	cell Nu * Ny
Eig_poles	Poles of the zpk forml	cell Nu * Ny
k	Gain of the zpk form	cell Nu * Ny
order	Order of the zpk form	cell Nu * Ny

Function's list



```
frac_zpk

frac_zpk(out fzpk : frac_zpk)
frac_zpk(inout fzpk : frac_zpk)
frac_zpk(ftf : frac_tf,out fzpk : frac_zpk)
frac_zpk(fss : frac_ss,out fzpk : frac_zpk)
frac_zpk(ftf : frac_tf,tol : int,out fzpk : frac_zpk)
frac_zpk(z : double,p : double,k : double,order : double,out fzpk : frac_zpk)
frac_zpk(z : double,p : double,k : double,order : double,N : double,band : double,out fzpk : frac_zpk)
frac_zpk(z : cell,p : cell,k : cell,order : cell,out fzpk : frac_zpk)
frac_zpk(z : cell,p : cell,k : cell,order : cell,N : double,band : double,out fzpk : frac_zpk)
char(fzpk : frac_zpk,out str1 : String,out str2 : String,out str3 : String,out str4 : String)
display(fzpk : frac_zpk)
eq(fzpk1 : frac_zpk,fzpk2 : frac_zpk,out bool : boolean)
get(fzpk : void,prop : String,out val : double)
get(fzpk : frac_zpk,prop : String,out val : String)
horzcat(fzpk1 : frac_zpk,fzpk2 : frac_zpk,out fzpk : int)
isempty(fzpk : frac_zpk,out bool : boolean)
mpower(fzpk : frac_zpk,p : int,out fzpk : frac_zpk)
mtimes(fzpk1 : frac_zpk,fzpk2 : frac_zpk,out fzpk : frac_zpk)
mtimes(fzpk1 : frac_zpk,m : double,out fzpk : frac_zpk)
mtimes(m : double,fzpk2 : frac_zpk,out fzpk : frac_zpk)
ne(fzpk : frac_zpk,out bool : boolean)
plot(fzpk : frac_zpk)
plus(fzpk1 : frac_zpk,fzpk2 : frac_zpk,out fzpk : frac_zpk)
plus(fzpk1 : frac_zpk,m : double,out fzpk : frac_zpk)
plus(m : double,fzpk2 : frac_zpk,fzpk : frac_zpk)
rdivide(fzpk1 : void,fzpk2 : frac_zpk,out fzpk : frac_zpk)
set(inout fzpk : frac_zpk,prop : String,val : double)
set(inout fzpk : frac_zpk,prop : String,val : String)
size(fzpk : frac_zpk,out n : double,out m : double)
subsasgn(inout fzpk : frac_zpk,index : structure,val : frac_zpk)
subsasgn(inout fzpk : frac_zpk,index : structure,val : String)
subsasgn(inout fzpk : frac_zpk,index : structure,prop : String)
subsref(inout fpzk : frac_zpk,index : structure)
transpose(inout fzpk : frac_zpk)
vertcat(fzpk1 : frac_zpk,fzpk2 : frac_zpk,out fzpk : frac_zpk)
zpkdata(fzpk : frac_zpk,z : cell,out p : cell,out k : cell,out order : cell)
```

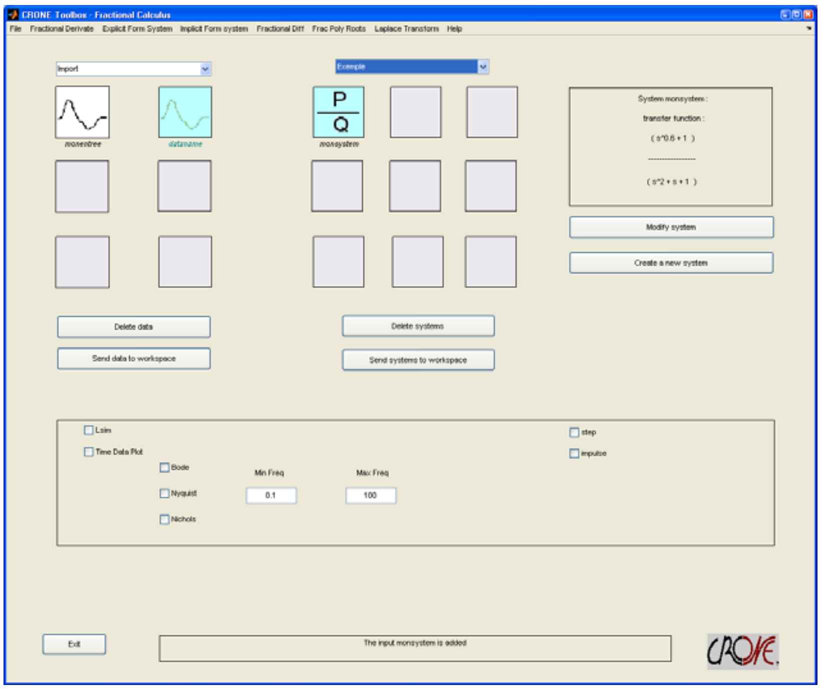


```
frac_zpk

minreal(inout fzpk : frac_zpk)
minreal(inout fzpk : frac_zpk,tol : double)
minreal(inout fzpk : frac_zpk,tol : double,str : String)
residue(fzpk : frac_zpk,out res : cell)
scalar(fzpk1 : frac_zpk,fzpk2 : frac_zpk,out C : double)
```

4 Graphic User Interface

The graphic user interface of the module ***Fractional Calculus*** is made up of pull-down menus and dialog boxes making it possible to enter the data and the parameters useful for calculations. The main window has a menu bar as follows organized:

File :	manages all data backup
Fractional Derivative :	computes a fractional order derivative
Explicit Form System :	computes the time and frequency response of a system described by a system of differential equations or by its transfer functions
Implicit Form System :	computes the time and frequency response of Implicit Form System described by its implicit fractional transfer functions
Fractional Differentiator :	synthesizes a rational differentiator
Fractional Polynomial Roots :	computes the fractional polynomial roots
Laplace Transform :	computes Laplace transform and inverse Laplace transform
Help :	help menu



```
System monsystem :
transfer function :
( s^0.6 + 1  )
-----------------
( s^2 + s + 1  )
```

Min Freq: 0.1

Max Freq: 100

```
The input monsystem is added
```

Figure 1 : main window of the module ***Fractional Calculus***

"File" Menu

File menu :

New session :	begins a new session
Open session :	opens a session from file of saved session
Save session as … :	saves all data of current session
Exit :	quits the module

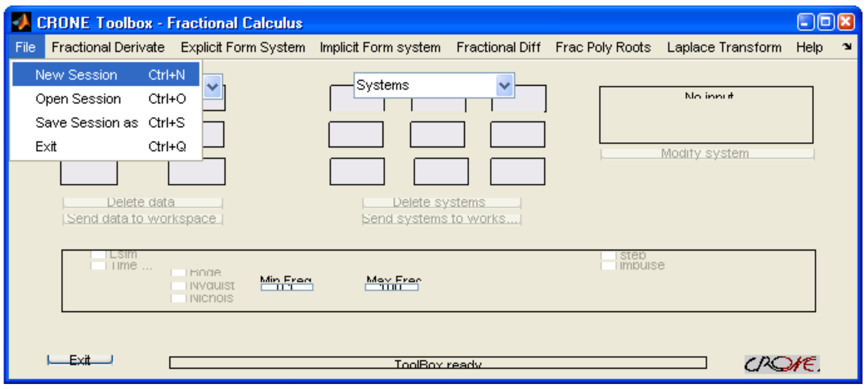


Figure 2 : ***File*** menu

The ***New*** command makes erase the session in progress and to start again a new session. The user can give a title to the session.

The ***Open*** command erase the session in progress and open a session from a file corresponding to a saved session, and in which are the data of the user.

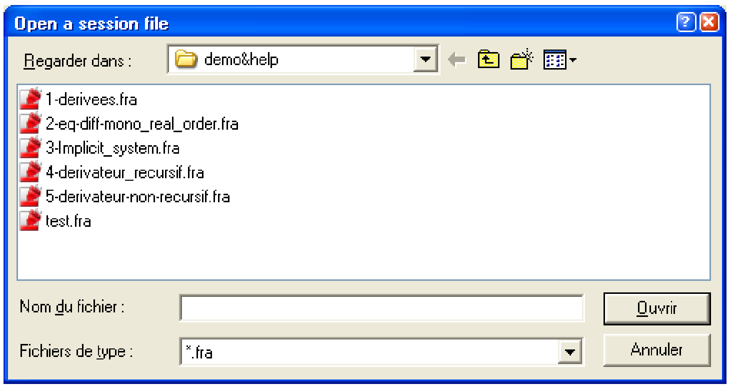


Figure 3 : Dialog box used to retrieve a file

If the selected file is not a session file, an error appears in the status bar:

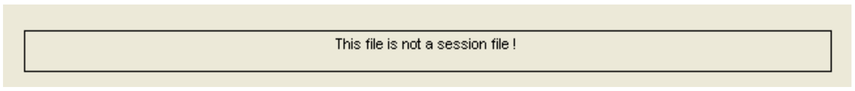

This file is not a session file !

Figure 4 : Status bar display

The ***Save as …*** command save data in a Matlab file (*.fra).

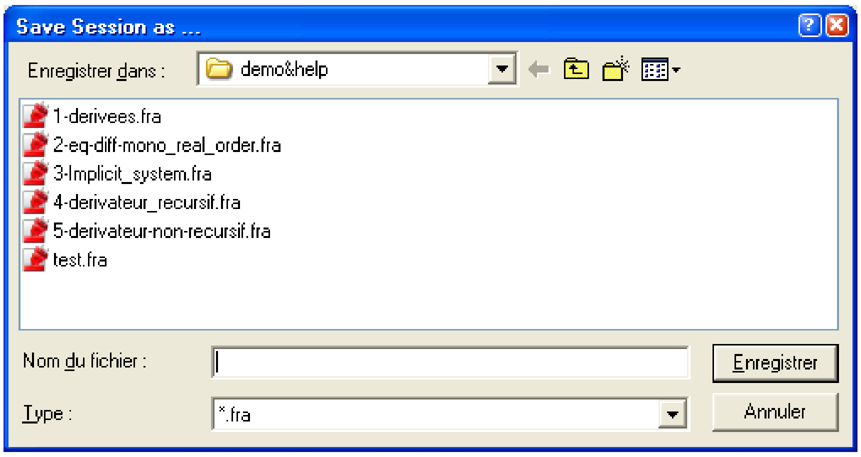
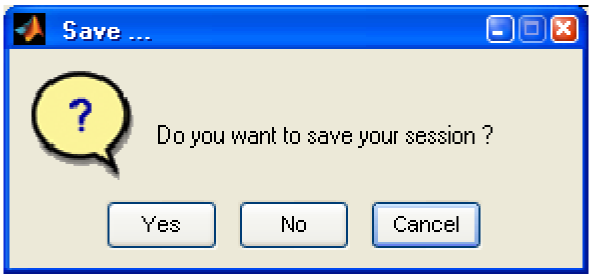


Figure 5 : Dialog box used to save a file

The ***Exit*** command quit the module ***Fractional Calculus***.

When data are present in memory, a message asks if the user wants to save them before carrying out a command.



Do you want to save your session ?

Figure 6 : Question dialog box

"Fractional Derivative" Menu

Fractional Derivative menu :

Data :	sets new data
Data processing :	data processing
Compute :	computes the fractional derivative
Option :	option about the tolerance

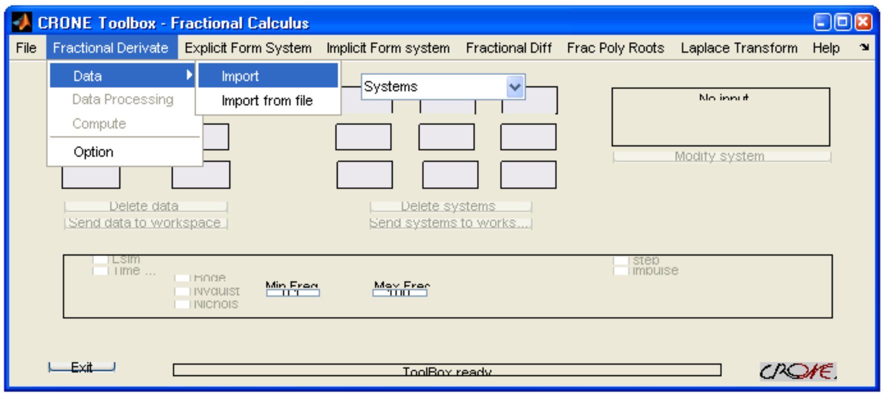
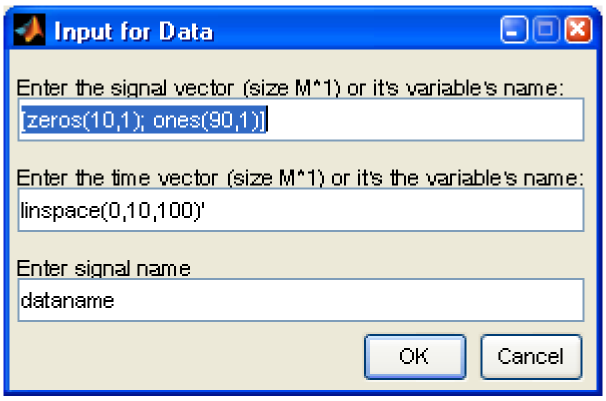


Figure 7 : ***Fractional Derivative*** menu

This menu computes the fractional order derivative of data vector.
This vector is entered by the user with the ***Data*** command.

Data command

The user can import data from Matlab workspace or from saved Matlab files (*.mat).



Input for Data

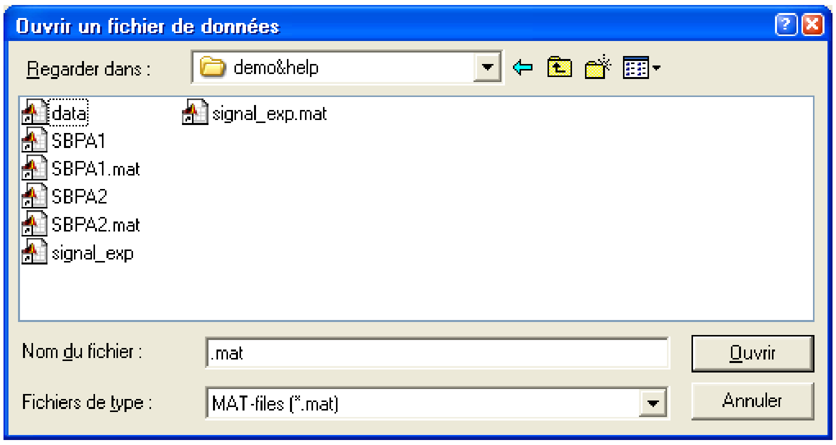
Enter the signal vector (size M*1) or it's variable's name:
[zeros(10,1); ones(90,1)]

Enter the time vector (size M*1) or it's the variable's name:
linspace(0,10,100)'

Enter signal name
dataname

OK Cancel

Figure 8 : Import dialog box

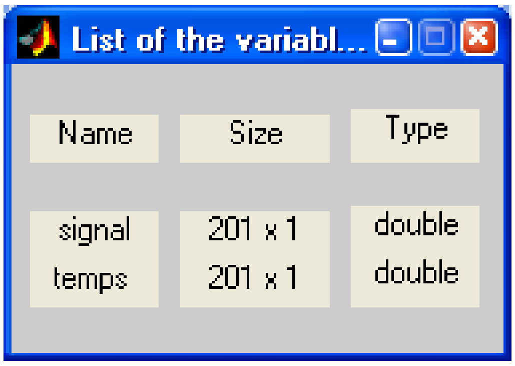


Nom du fichier : .mat

Fichiers de type : MAT-files (*.mat)

Figure 9 : Dialog box used to retrieve a file

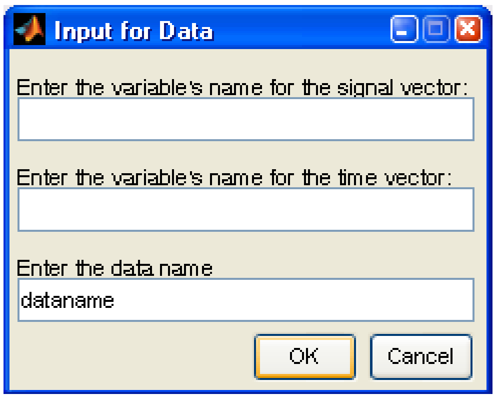
The list of variables included in the selected file appears in the following window :



Name	Size	Type
signal	201 x 1	double
temps	201 x 1	double

Figure 10 : List of variables

If a file was selected at the previous step, it is then possible to use the variables included in this file and displayed in a window (Figure 10).



Enter the variable's name for the signal vector:

Enter the variable's name for the time vector:

Enter the data name: dataname

Figure 11 : Import from file dialog box

Data Processing command

The ***Input Signal Processing*** command allows to process the signal in memory. It opens the datashaping window.

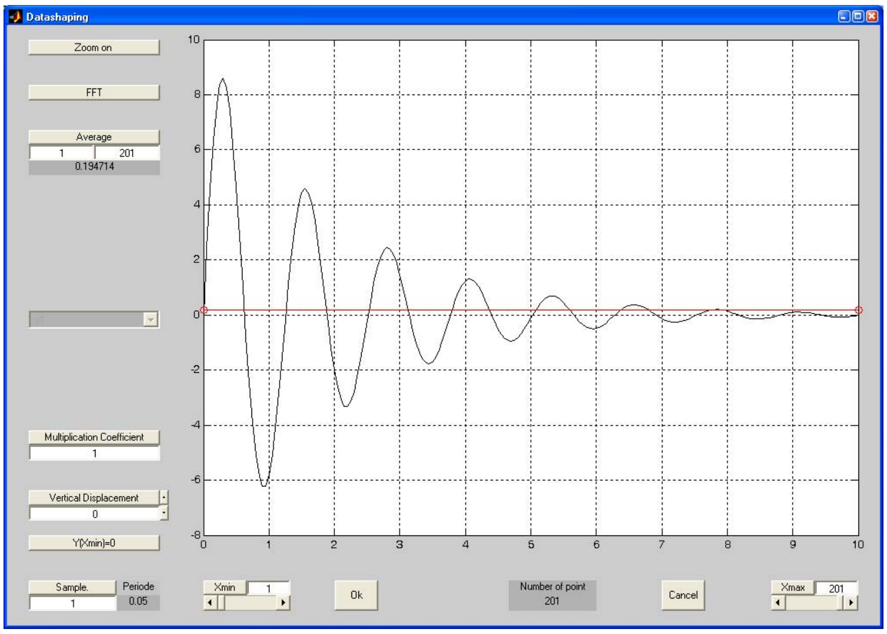
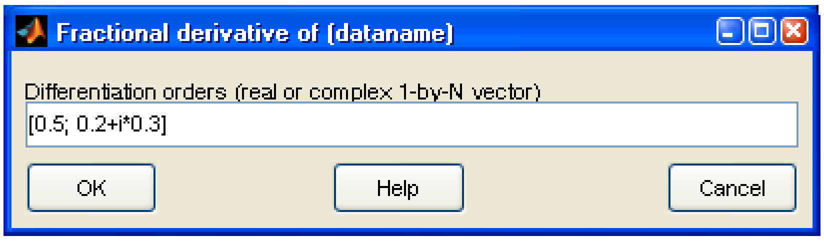


Figure 12 : Datashaping window

The ***FFT*** button gives the FFT of the signal.
The ***average*** button gives and plots the average of the signal from the point in the first box under the button to the point in the second box under the button.
The ***multiplication*** button multiplies the signal by the number given in the box under the button.
The ***vertical displacement*** moves the signal by the number given in the box under the button.
The ***sample*** gives which points will be used to plot the signal, for example if you choose 10 for the sample and you have 200 points for the signal only 20 of them will be used to plot the signal.
The ***Xmin*** button gives the begin point of the plot of the signal.
The ***Xmax*** button gives the ending point of the plot of the signal.

Compute command

The ***Compute*** command computes the fractional derivative.


Fractional derivative of (dataname)

Differentiation orders (real or complex 1-by-N vector)

[0.5; 0.2+i*0.3]

Figure 13 : Editable boxes to input orders

Once the derivation orders are edited, the result is drawn in a new figure.

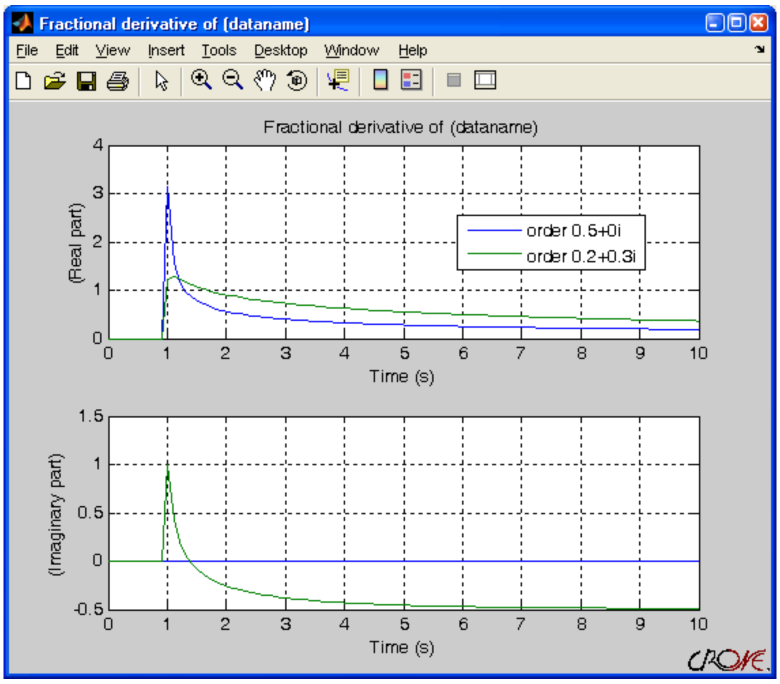
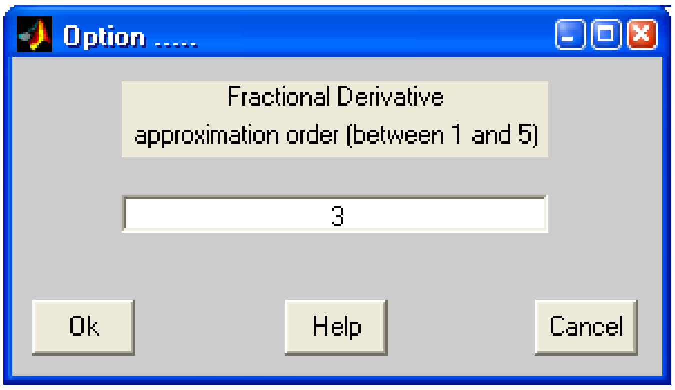


Figure 14 : Results of fractional derivatives

Option Command

The ***Option*** command permits to select the tolerance to obtain a better accuracy in terms of h^2, h^3, etc… (eq. (7).


Option .....

Fractional Derivative
approximation order (between 1 and 5)

3

Figure 15 : ***Option*** window

"Explicit Form System (Differential Equations)" Menu

Explicit Form System (Differential Equations) menu :

System definition :	sets the fractional differential equations
Data :	sets input signals
Data processing :	data processing
Output time responses :	computes output time responses
Frequency responses :	computes frequency responses
Eigenvalue and Poles :	computes eigenvalues, zeros and poles

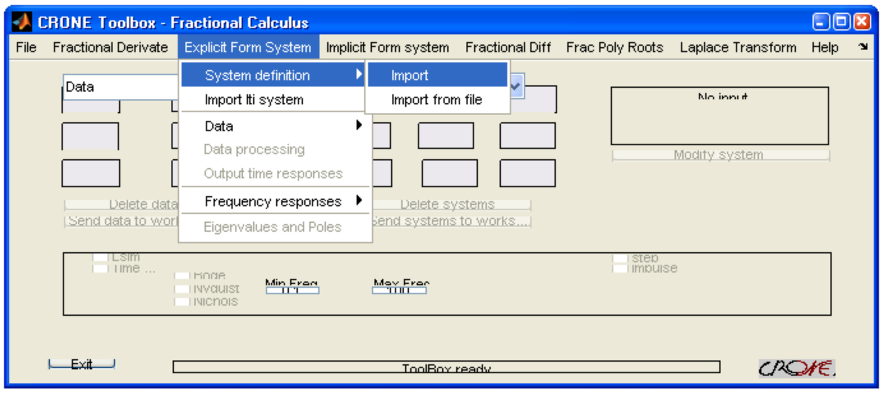
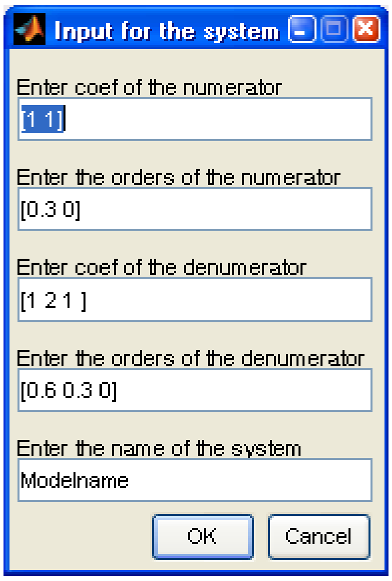


Figure 16 : ***Explicit Form System (Differential Equations)*** menu

System definition command

The user can import data from Matlab workspace or from saved Matlab files (*.mat) to enter the coefficients and the orders of the differential equations describing the system.


Input for the system

Enter coef of the numerator
[1 1]

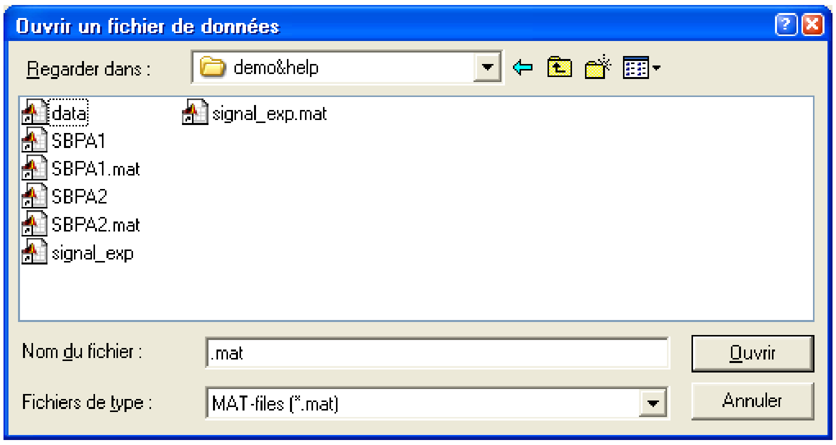
Enter the orders of the numerator
[0.3 0]

Enter coef of the denumerator
[1 2 1]

Enter the orders of the denumerator
[0.6 0.3 0]

Enter the name of the system
Modelname

Figure 17 : Import dialog box



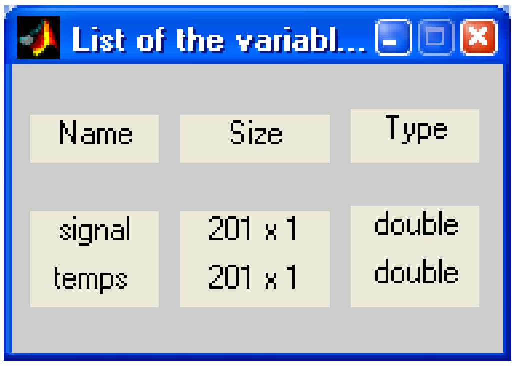
data
SBPA1
SBPA1.mat
SBPA2
SBPA2.mat
signal_exp
signal_exp.mat

Nom du fichier : .mat

Fichiers de type : MAT-files (*.mat)

Figure 18 : Dialog box used to retrieve a file

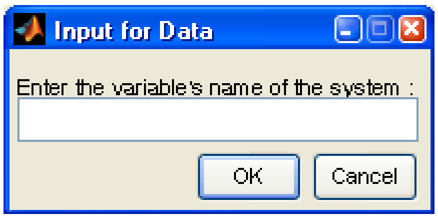
The list of variables included in the selected file appears in the following window:



Name	Size	Type
signal	201 x 1	double
temps	201 x 1	double

Figure 19 : List of variables

If a file was selected at the previous step, it is then possible to use the variables included in this file and displayed in a window (Figure 10).



Enter the variable's name of the system :

Figure 20 : Import from file dialog box

Data command

The user can import data from Matlab workspace or from saved Matlab files (*.mat).

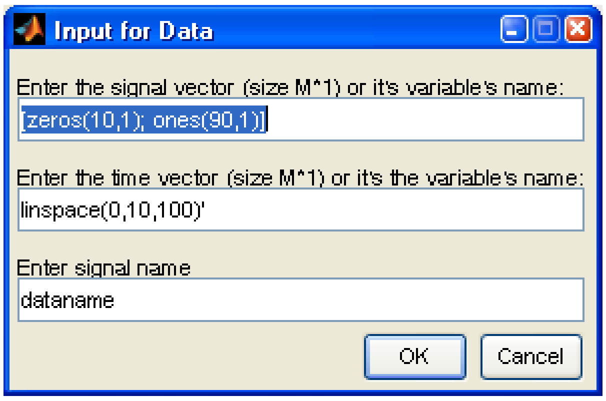

Input for Data
Enter the signal vector (size M*1) or it's variable's name:
[zeros(10,1); ones(90,1)]
Enter the time vector (size M*1) or it's the variable's name:
linspace(0,10,100)'
Enter signal name
dataname
OK
Cancel

Figure 21 : Import dialog box

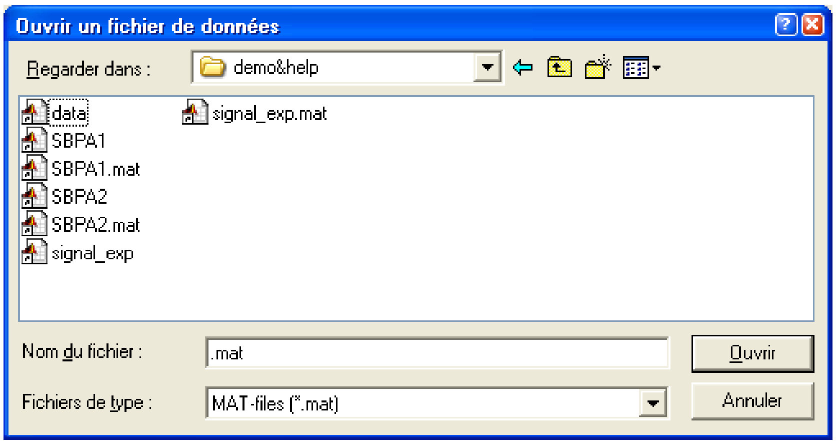
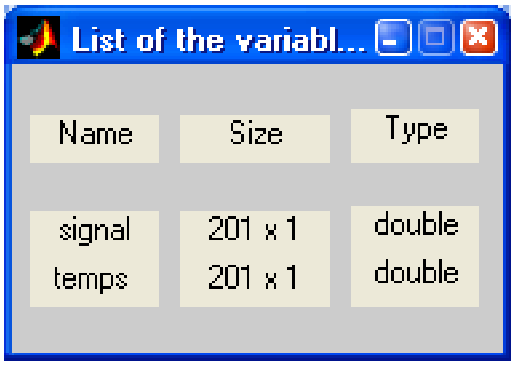


Figure 22 : Dialog box used to retrieve a file

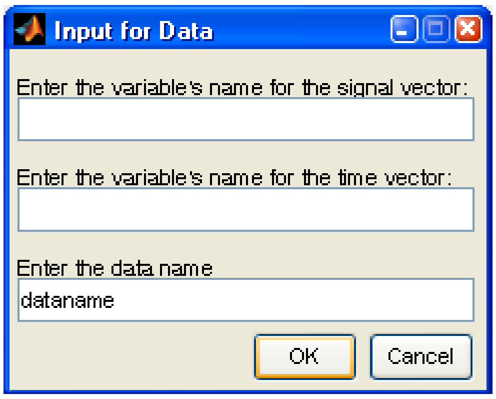
The list of variables included in the selected file appears in the following window:



Name	Size	Type
signal	201 x 1	double
temps	201 x 1	double

Figure 23 : List of variables

If a file was selected at the previous step, it is then possible to use the variables included in this file and displayed in a window (Figure 10).


Input for Data

Enter the variable's name for the signal vector:

Enter the variable's name for the time vector:

Enter the data name

dataname

OK Cancel

Figure 24 : Import from file dialog box

Data Processing command

The ***Input Signal Processing*** command allows to process the signal in memory. It opens the datashaping window.

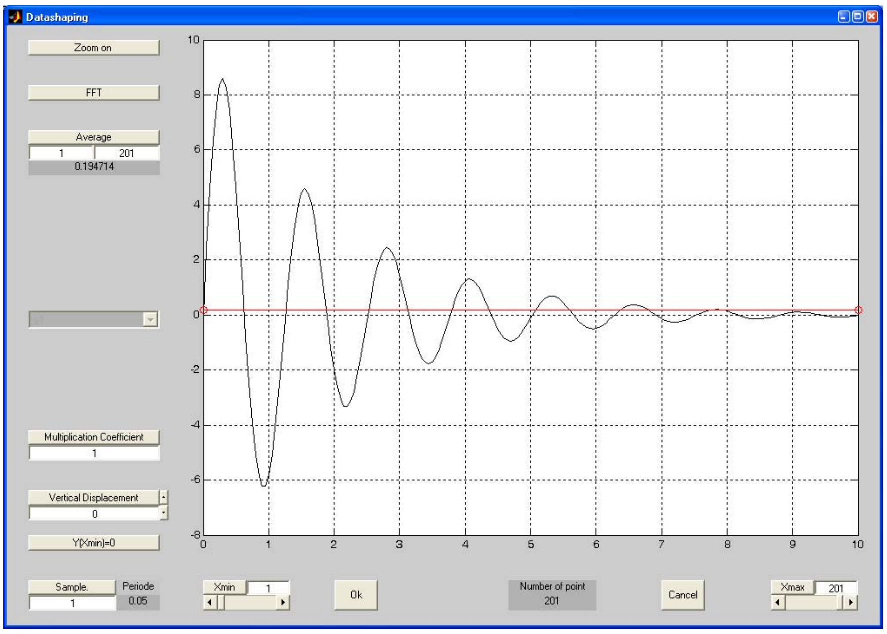


Figure 25 : Datashaping window

The ***FFT*** button gives the FFT of the signal.
The ***average*** button gives and plots the average of the signal from the point in the first box under the button to the point in the second box under the button.
The ***multiplication*** button multiplies the signal by the number given in the box under the button.
The ***vertical displacement*** moves the signal by the number given in the box under the button.
The ***sample*** gives which points will be used to plot the signal, for example if you choose 10 for the sample and you have 200 points for the signal only 20 of them will be used to plot the signal.
The ***Xmin*** button gives the begin point of the plot of the signal.
The ***Xmax*** button gives the ending point of the plot of the signal.

Output time responses command

The ***Compute*** command computes the output time response of the system.

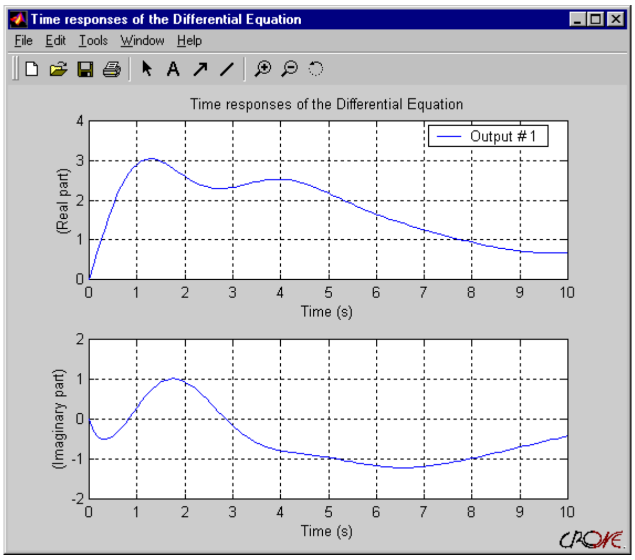


Figure 26 : plots of time responses

Frequency Responses & Eigenvalues and Poles command

The ***Frequency Responses*** submenu includes all commands to plot Bode diagram, Nichols charts and Nyquist plot.

The ***Eigenvalues and Poles*** commands display poles and eigenvalues of the system into a Matlab window.

"Implicit Form System" menu

Although these developments are still being developed, the menus and commands concerning the time and frequency simulation of the implicit form systems are envisaged.

This menu is organized as the ***Explicit Form System (Differential Equations)*** menu:

Implicit Form System menu:

System definition:	sets the implicit form system
Data:	sets input signals
Data processing:	data processing
Output time responses:	computes output time responses
Frequency responses:	computes frequency responses

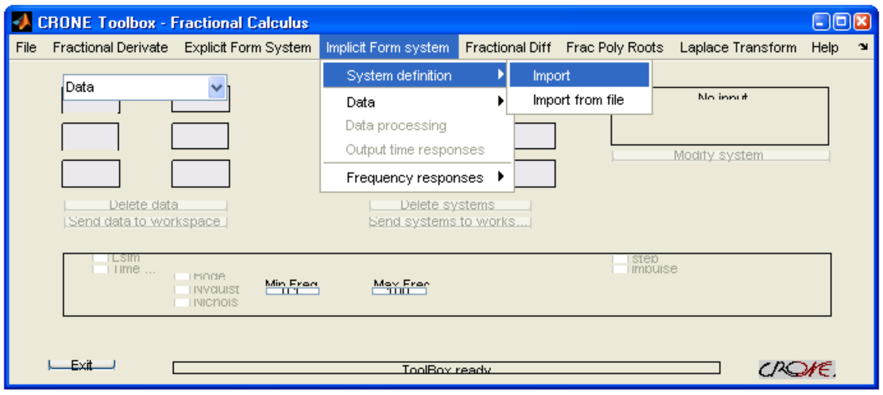
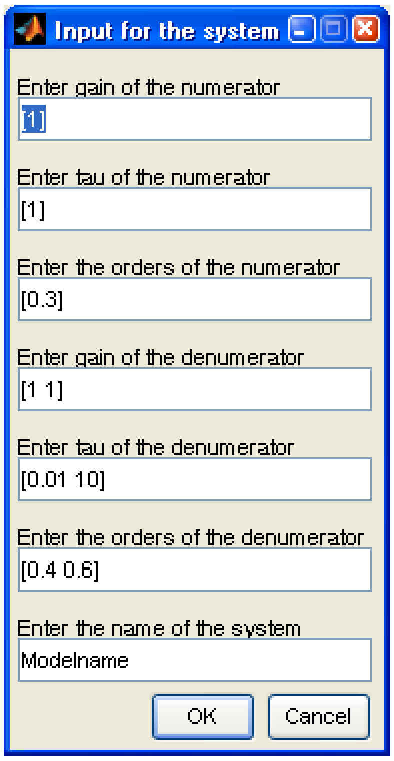


Figure 27 : ***Implicit Form System*** menu

System definition menu

The user can import data from Matlab workspace or from saved Matlab files (*.mat) to enter the coefficients and the orders of the differential equations describing the system.


Input for the system

Enter gain of the numerator
[1]

Enter tau of the numerator
[1]

Enter the orders of the numerator
[0.3]

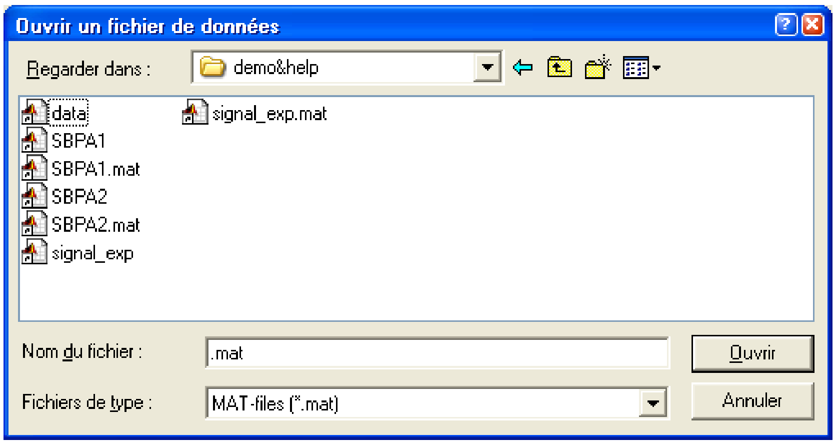
Enter gain of the denumerator
[1 1]

Enter tau of the denumerator
[0.01 10]

Enter the orders of the denumerator
[0.4 0.6]

Enter the name of the system
Modelname

Figure 28 : Import dialog box


Ouvrir un fichier de données

Regarder dans : demo&help

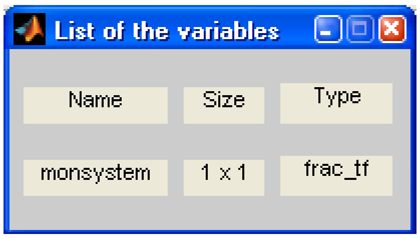
data
SBPA1
SBPA1.mat
SBPA2
SBPA2.mat
signal_exp
signal_exp.mat

Nom du fichier : .mat

Fichiers de type : MAT-files (*.mat)

Figure 29 : Dialog box used to retrieve a file

The list of variables included in the selected file appears in the following window :


List of the variables

Name	Size	Type
monsystem	1 x 1	frac_tf

Figure 30 : List of variables

If a file was selected at the previous step, it is then possible to use the variables included in this file and displayed in a window (Figure 10).

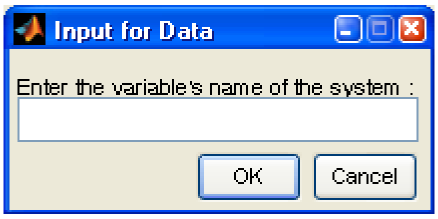
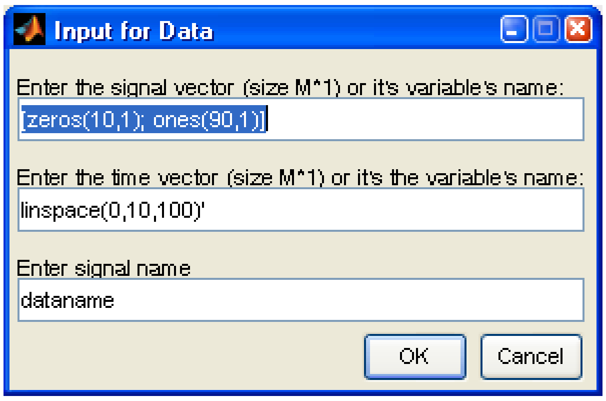


Figure 31 : Import from file dialog box

Data menu

The user can import data from Matlab workspace or from saved Matlab files (*.mat).



Enter the signal vector (size M*1) or it's variable's name:
[zeros(10,1); ones(90,1)]

Enter the time vector (size M*1) or it's the variable's name:
linspace(0,10,100)'

Enter signal name
dataname

Figure 32 : Import dialog box

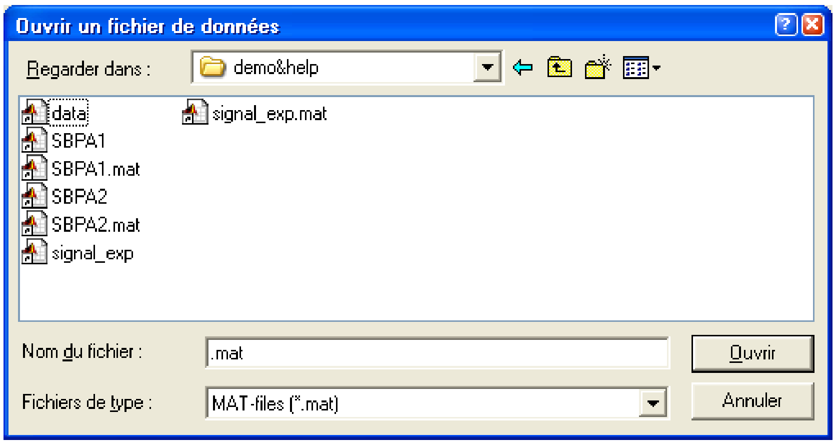
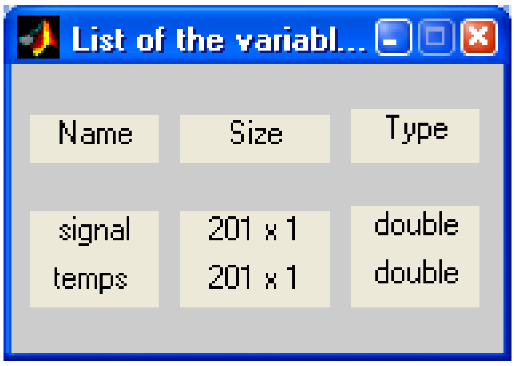


Figure 33 : Dialog box used to retrieve a file

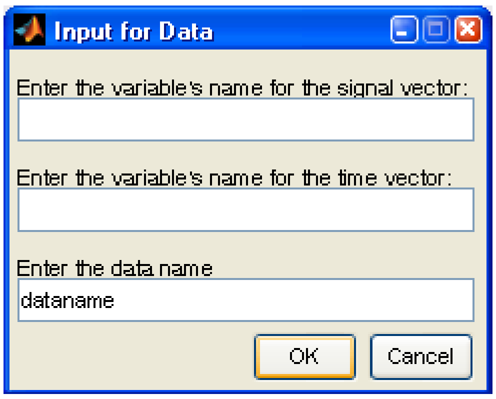
The list of variables included in the selected file appears in the following window:


List of the variabl...

Name	Size	Type
signal	201 x 1	double
temps	201 x 1	double

Figure 34 : List of variables

If a file was selected at the previous step, it is then possible to use the variables included in this file and displayed in a window (Figure 10).


Input for Data

Enter the variable's name for the signal vector:

Enter the variable's name for the time vector:

Enter the data name

dataname

Figure 35 : Import from file dialog box

Data Processing command

The ***Input Signal Processing*** command allows to process the signal in memory. It opens the datashaping window.

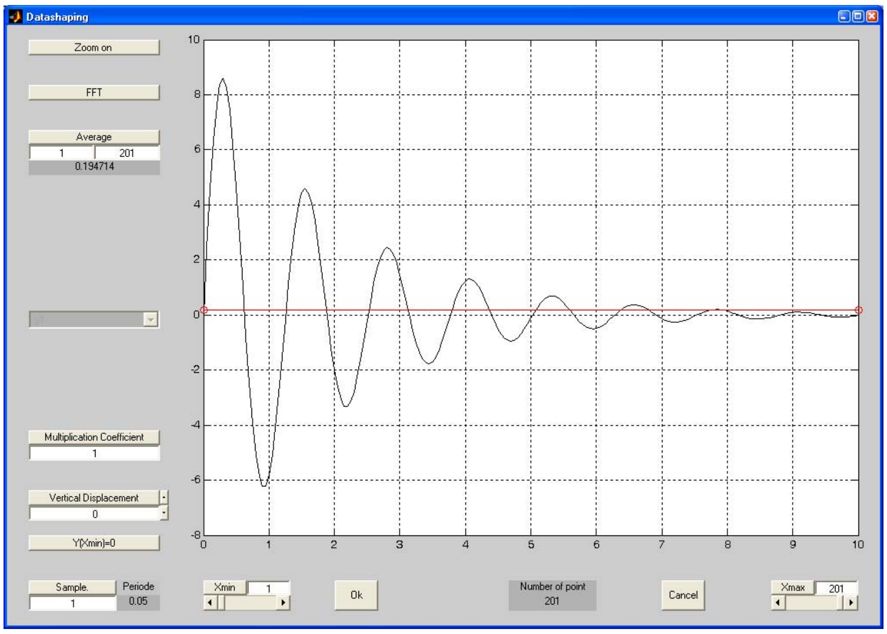


Figure 36 : Datashaping window

The ***FFT*** button gives the FFT of the signal.
The ***average*** button gives and plots the average of the signal from the point in the first box under the button to the point in the second box under the button.
The ***multiplication*** button multiplies the signal by the number given in the box under the button.
The ***vertical displacement*** moves the signal by the number given in the box under the button.
The ***sample*** gives which points will be used to plot the signal, for example if you choose 10 for the sample and you have 200 points for the signal only 20 of them will be used to plot the signal.
The ***Xmin*** button gives the begin point of the plot of the signal.
The ***Xmax*** button gives the ending point of the plot of the signal.

Output time responses command

The ***Compute*** command computes the output time response of the system.

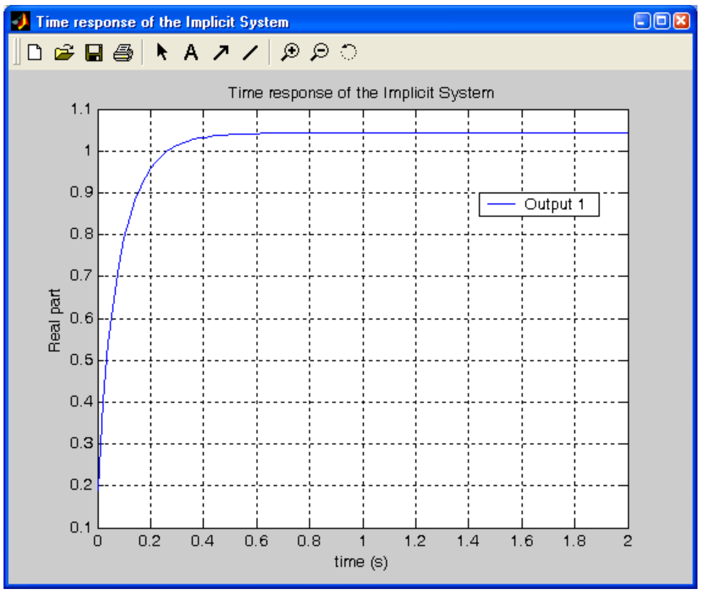


Figure 37 : plots of time responses

Frequency responses

The ***Frequency responses*** menu regroups the commands that draw the Bode, Nichols and Nyquist diagrams.

"Fractional Differentiator" menu

Fractional Differentiator menu:

Differentiator parameters:	sets differentiator parameters
View fractional differentiator:	displays differentiators
Modify Bode diagrams:	modifies parameters and Bode diagrams
Bode diagrams:	plots Bode diagrams
Unit choice:	sets units and axes properties

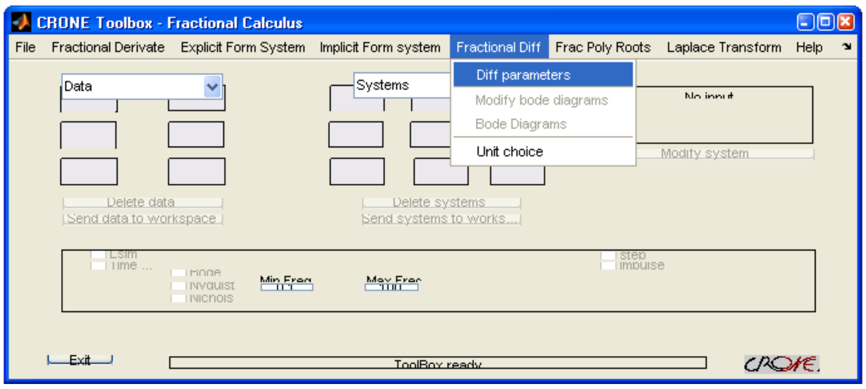
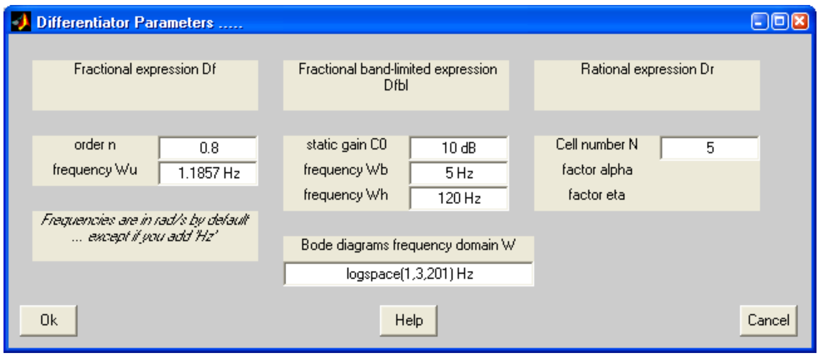


Figure 38 : ***Fractional Differentiator*** menu

Differentiator parameters command

The dialog window allows to set all the parameters of the differentiator. This window includes editable boxes for each parameter.


Differentiator Parameters .....

Fractional expression Df		Fractional band-limited expression Dfbl		Rational expression Dr	
order n	0.8	static gain C0	10 dB	Cell number N	5
frequency Wu	1.1857 Hz	frequency Wb	5 Hz	factor alpha	
		frequency Wh	120 Hz	factor eta	

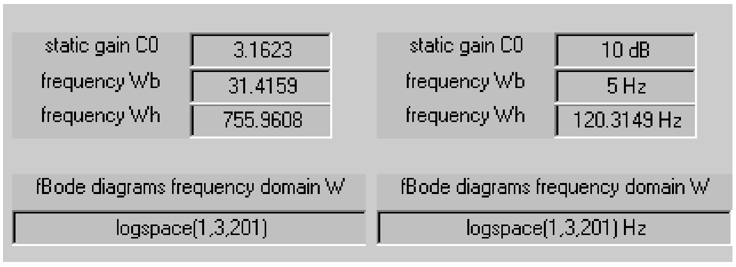
Frequencies are in rad/s by default ... except if you add 'Hz'

Bode diagrams frequency domain W: logspace(1,3,201) Hz

Figure 39 : Parameters editable boxes

The static gain can be specified in linear unit, or in dB; to do that, type the value followed by the text "dB". The frequencies can be also specified either in radian per second (rad/s - by default), or in Hertz; to do that, type the value followed by the text "Hz".

The recursive factors *alpha* and *eta* are deducted from the other parameters.



static gain C0	3.1623	static gain C0	10 dB
frequency Wb	31.4159	frequency Wb	5 Hz
frequency Wh	755.9608	frequency Wh	120.3149 Hz
fBode diagrams frequency domain W	logspace(1,3,201)	fBode diagrams frequency domain W	logspace(1,3,201) Hz

Figure 40 : Editable boxes for gain and frequencies

View fractional differentiator command

This command, will soon allow to display the differentiators:

- fractional differentiator

$$D_{fractional}(p) = C_0\left(\frac{p}{\omega_u}\right)^n \tag{42}$$

- fractional frequency band-limited differentiator :

$$D_{fbl}(p) = C_0\left(\frac{1+\dfrac{p}{\omega_b}}{1+\dfrac{p}{\omega_h}}\right)^n \tag{43}$$

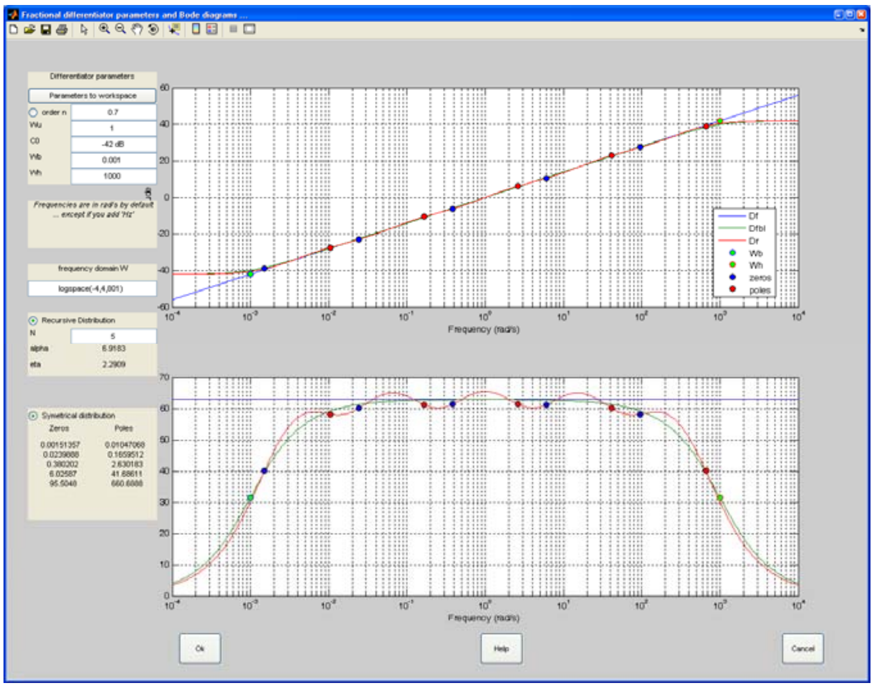
- rational differentiator :

$$D_{rational}(p) = C_0\prod_k\left(\frac{1+\dfrac{p}{\omega_{bk}}}{1+\dfrac{p}{\omega_{hk}}}\right). \tag{44}$$

Modify Bode diagrams command

The ***Modify Bode diagrams*** command synthesizes the differentiator in two ways: either by modifying the numerical values of the parameters, or
by moving the poles or the zeros ω_b and ω_h of the frequency-band or rational differentiator.

All editable boxes of differentiators (Figure 39) are displayed again on bode diagrams windows (Figure 41); after each modification, all plots are updated. Conversely, at each modification of the position of a point of the curve, the numerical values of the parameters are updated.


Fractional differentiator parameters and Bode diagrams ...

Differentiator parameters

Parameters to workspace

○ order n	0.7
Wu	1
C0	-42 dB
Wb	0.001
Wh	1000

Frequencies are in rad/s by default ... except if you add 'Hz'

frequency domain W

logspace(-4,4,801)

◉ Recursive Distribution

N	5
alpha	6.9183
eta	2.2909

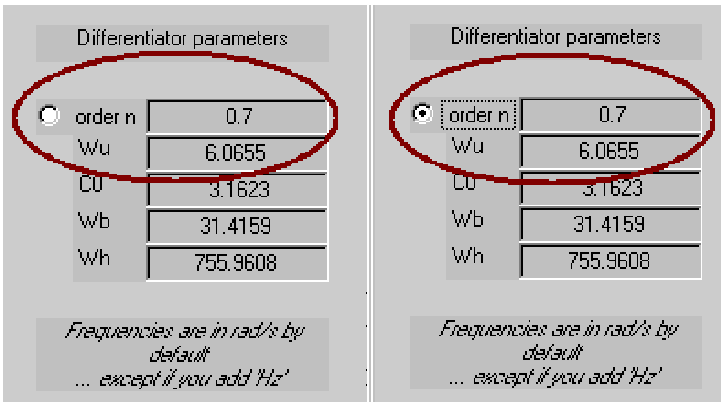
◉ Symetrical distribution

Zeros	Poles
0.00151357	0.01047068
0.0239888	0.1659512
0.380202	2.630183
6.02587	41.68611
95.5048	660.6888

Figure 41 : Plots of Bode diagrams

All the points of the curve can be dragged by the mouse directly on the plot; to do it, place the cursor over the chosen point, select it while pressing on the key and move the point. The new value of the frequency of one of the zeros or one of the poles is used to update the Bode diagrams and editable boxes.

When the zero ω_b or the pole ω_h is moved, only the new frequency of the moved point is modified. The magnitude and the phase in this case are refreshed according to the order and the gain at the unity frequency.
If the editable box of the order n is operational, the new position of the zero ω_b or the pole ω_h makes it possible to compute the new order n of the differentiator, thanks to the X-coordinate (the frequency) and to the Y-coordinate (magnitude or the phase) of the moved point (contrary to the previous case).


Differentiator parameters

○ order n	0.7
Wu	6.0655
C0	3.1623
Wb	31.4159
Wh	755.9608

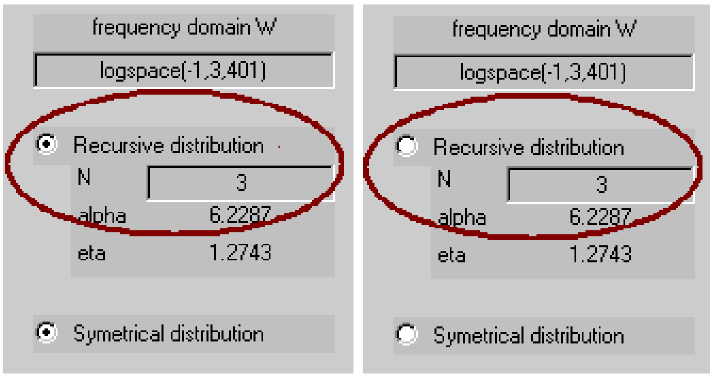
Frequencies are in rad/s by default ... except if you add 'Hz'

Differentiator parameters

◉ order n	0.7
Wu	6.0655
C0	3.1623
Wb	31.4159
Wh	755.9608

Frequencies are in rad/s by default ... except if you add 'Hz'

Figure 42 : Modification of the order
with new position of the points ω_b et ω_h
(selected radio button : right case)

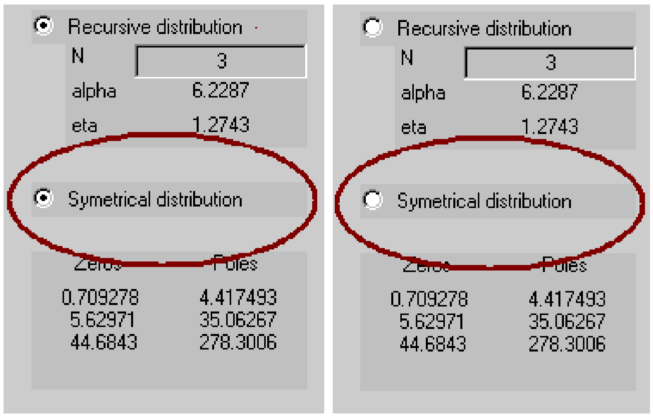


frequency domain W
logspace(-1,3,401)
[✓ Recursive distribution]
N 3
alpha 6.2287
eta 1.2743
[✓ Symetrical distribution]

frequency domain W
logspace(-1,3,401)
Recursive distribution
N 3
alpha 6.2287
eta 1.2743
Symetrical distribution

Figure 43 : ***Recursive distribution*** radiobutton

When one zero (or one pole) of the rational differentiator is moved, its symmetrical point compared to ω_b and ω_h is also moved in the same proportions; for example, when the first zero is moved towards the low frequencies, the last pole is moved in the same proportions towards the high frequencies, in an identical way for each zero and pole of the rational differentiator. To modify a single point without influencing "its symmetrical", select the ***Symmetrical distribution*** radiobutton to disable this functionality; selecting this radiobutton again validates the functionality.



[✓ Recursive distribution]
N 3
alpha 6.2287
eta 1.2743
[✓ Symetrical distribution]

Zeros	Poles
0.709278	4.417493
5.62971	35.06267
44.6843	278.3006

Recursive distribution
N 3
alpha 6.2287
eta 1.2743
Symetrical distribution

Zeros	Poles
0.709278	4.417493
5.62971	35.06267
44.6843	278.3006

Figure 44 : ***Symmetrical distribution*** radiobutton

Bode diagrams command

This command plots Bode diagrams of the differentiators into a new window (to print the plots for example).

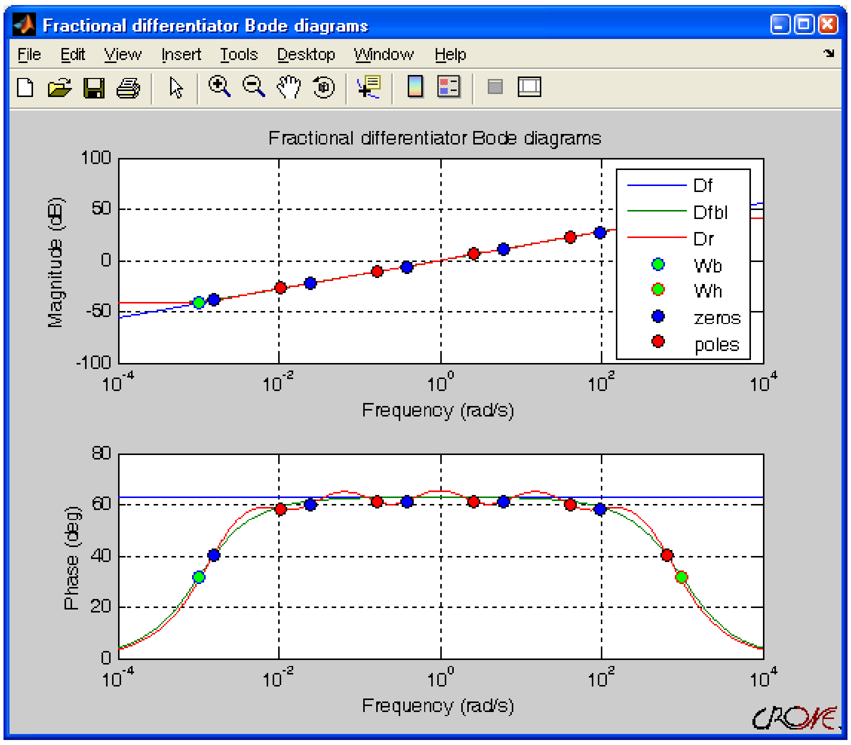
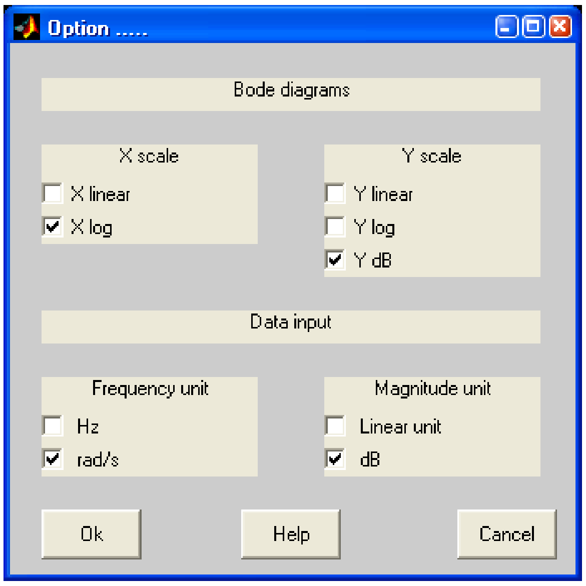


Figure 45 : Plots of Bode diagrams

Unit choice command

This command makes it possible to set the units and axes scale. Moreover, the frequency responses can be displayed according to the frequencies either in Hertz (Hz), or in radian per second (rad/s). The magnitude can be displayed either in linear scale, or in decibel (dB).



Bode diagrams

X scale
- [ ] X linear
- [✓] X log

Y scale
- [ ] Y linear
- [ ] Y log
- [✓] Y dB

Data input

Frequency unit
- [ ] Hz
- [✓] rad/s

Magnitude unit
- [ ] Linear unit
- [✓] dB

Figure 46 : ***Unit choice*** window

In this example, the frequency response is plotted on Bode diagrams (with logarithmic X-coordinate and linear Y-coordinate in dB) with the angular frequency (rad/s); the magnitude is displayed in dB.

"Fractional Polynomial Roots" menu

The ***Fractional Polynomial roots*** command computes the fractional polynomial roots.

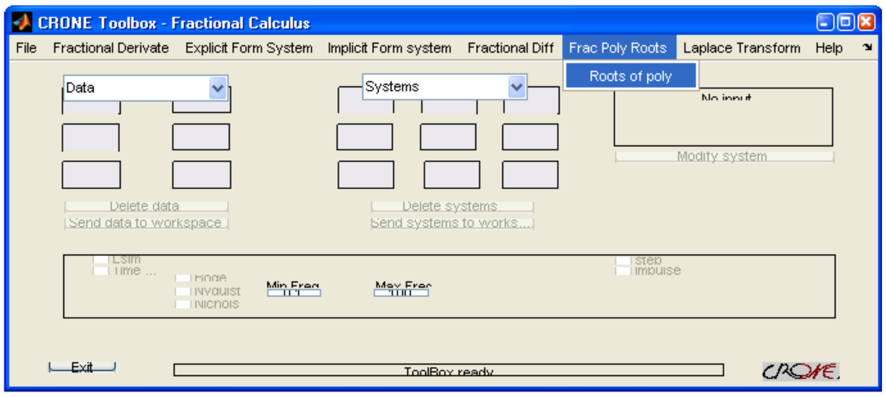
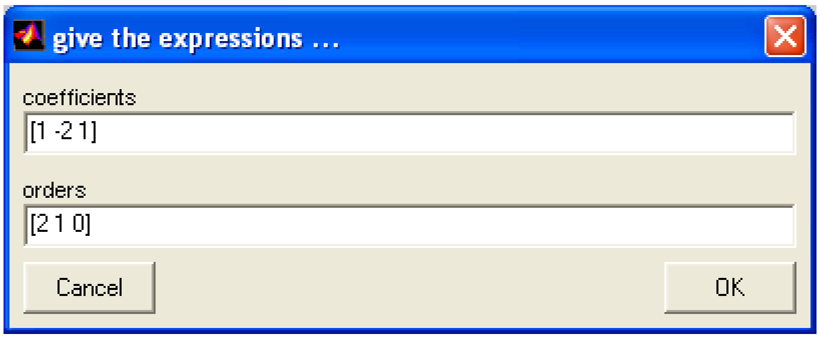


Figure 47 : ***Fractional Polynomial Roots*** menu

The coefficients and orders are set in the following input dialog box :


give the expressions ...

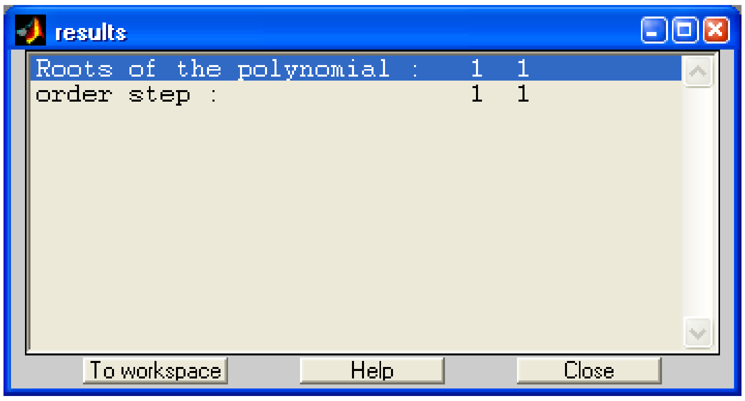
coefficients
[1 -2 1]

orders
[2 1 0]

Cancel   OK

Figure 48 : Input dialog box

The results are displayed into a window as this example:


results

```
Roots of the polynomial :     1   1
order step :                  1   1
```

To workspace   Help   Close

Figure 49 : Example of polynomial roots

"Laplace transform" menu

Laplace Transform menu:

Laplace transform:	computes Laplace transform
Inv Laplace transform:	computes inverse Laplace transform
Option:	method option

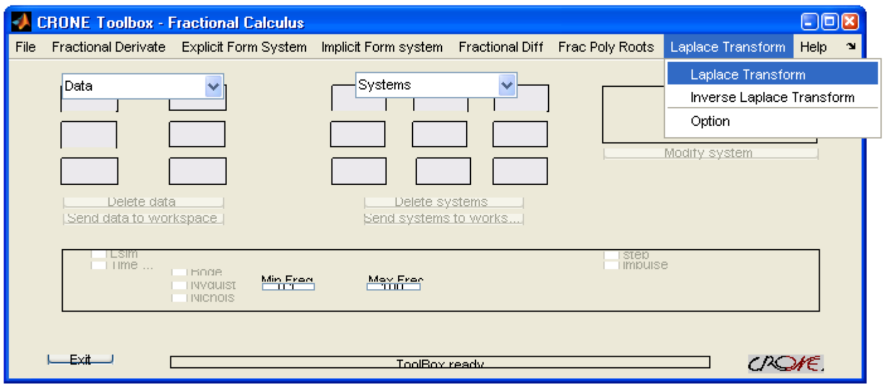
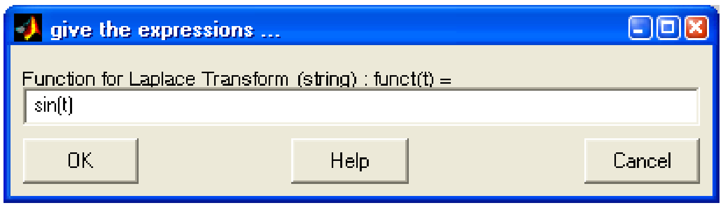


Figure 50 : ***Laplace transform*** menu

This menu makes it possible to compute either the Laplace transform of a function, or the inverse Laplace transform of a function. To carry out calculation, the frequency or time vector is required. The function must be written in matlab language knowing that the variables are vectors.



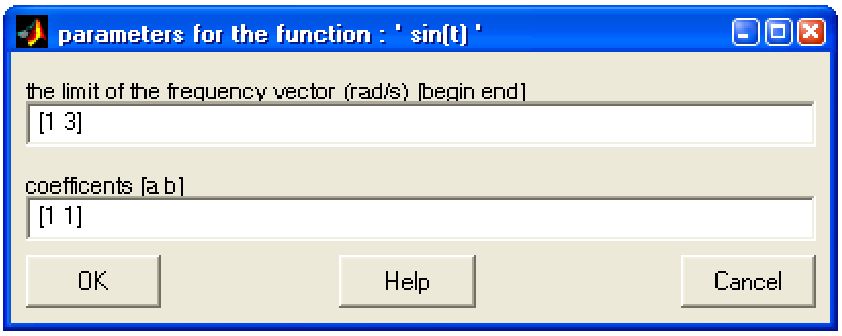
Function for Laplace Transform (string) : funct(t) =

sin(t)

Figure 51 : Function to transform

Then the user is asked to give the decade frequency limit of the band and the coefficient a and b of the function form. The frequency limits is in decade units so if you set the frequency limits at [1 3] the frequency range will be between 10 and 10^3. The coefficients a and b are the coefficients of the function form which is:

$$f(t) = \sum_{k=0}^{\infty} \left(\frac{a_k t^{(a-1+kb)}}{\Gamma(a+kb)} \right).$$



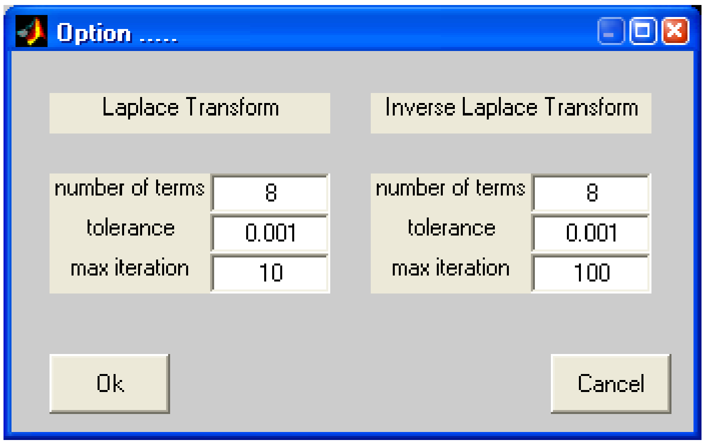
parameters for the function : ' sin(t) '

the limit of the frequency vector (rad/s) [begin end]
[1 3]

coefficents [a b]
[1 1]

Figure 52 : Frequency vector and parameters a & b of the function

The software then asks you to check your options.
The ***Option*** command sets tolerance and maximum iteration for the Aitken method.
The number of terms is the number N explained in the principles.
The tolerance is the maximum difference of the computed laplace transform between two iteration.



Option .....

Laplace Transform		Inverse Laplace Transform	
number of terms	8	number of terms	8
tolerance	0.001	tolerance	0.001
max iteration	10	max iteration	100

Figure 53 : ***Option*** dialog boxes about Laplace transform and inverse Laplace transform

The result is then plotted.

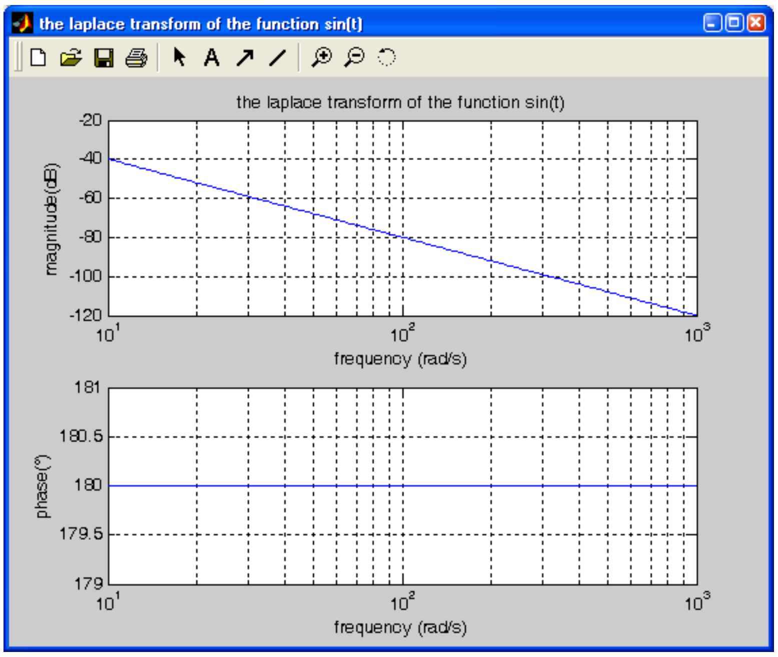


Figure 54 : Result of Laplace transform

"Help" menu

Help menu:

Differentiator parameters:		sets differentiator parameters
Contents	:	Describes briefly the unit
Index	:	Index of the help
About …		

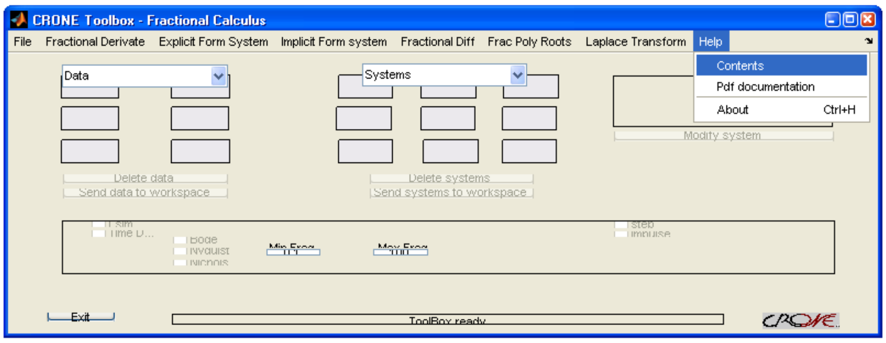


Figure 55 : ***Help*** menu

5 Reference

N.B.: This part correspond to an ancient version where the functions are described.To be reused.

In this paragraph, the significant functions of the ***Fractional Calculus*** module are presented by topic.

The organization of the variables was set up to standardize their use, and to simplify the lines of command by using structured variables.

In addition, the objective is to develop similar commands of the Matlab functions, and to make functions of ***CRONE toolbox***, an extension of the Matlab functions.

The following arithmetic operators have been overloaded for all fractional objects(frac_tf, frac_zpk, frac_ss, frac_poly_exp and frac_poly_imp)

+ and -	Add and subtract systems
*	Multiply systems
.*	Element-by-element multiplication
\	Left divide -- sys1\sys2 means inv(sys1)*sys2
/	Right divide -- sys1/sys2 means sys1*inv(sys2)
^	Powers of given system

.'

Transposition of input/output map

aitken

Syntax

```
[x,y,erreur] = aitken(u)
```

Description

This function solves the Aitken system [Levron98]
$\sigma(x^{indice} * y) = u$

Arguments

Argument in :
u : complex vector

Argument out :
x : complex vector
y : complex vector
erreur : string

app_imp1

Syntax

```
    [Co,Wzero,Wpole,Sys,Erreur] = app_imp1(n,Wb,Wh,Nz)
```

Description

This function computes an approximation, on a band frequency, of the derivate transfert function (s^n).
The approximation is a recursive pole-zero pair distribution.

Arguments

Argument in :
n : order of the derivate (scalar)
Wb : minimum frequency of the frequency band for the approximation (scalar)
Wh : maximum frequency of the frequency band for the approximation (scalar)
Nzp : number of zero and poles for the approximation (scalar)

Argument out :
Co : gain
Wzero : zeros of the approximation (vector)
Wpole : poles of the approximation (vector)
Sys : lti system of the approximation (lti)
erreur : error string (string)

Example

```
>> [Co,Wzero,Wpole,Sys,Erreur]=app_imp1(0.5,0.01,100,7)
Co =
  100.0000
Wzero =
    0.0139
    0.0518
    0.1931
    0.7197
    2.6827
   10.0000
   37.2759
Wpole =
    0.0268
    0.1000
    0.3728
    1.3895
    5.1795
   19.3070
   71.9686
Sys=
Zero/pole/gain:
100  (s+0.01389)  (s+0.05179)  (s+0.1931)  (s+0.7197)
(s+2.683) (s+10) (s+37.28)
-------------------------------------------------------
```

```
(s+0.02683)  (s+0.1)  (s+0.3728)  (s+1.389)  (s+5.179)
(s+19.31) (s+71.97)
Erreur =
```

append(frac_poly_exp)

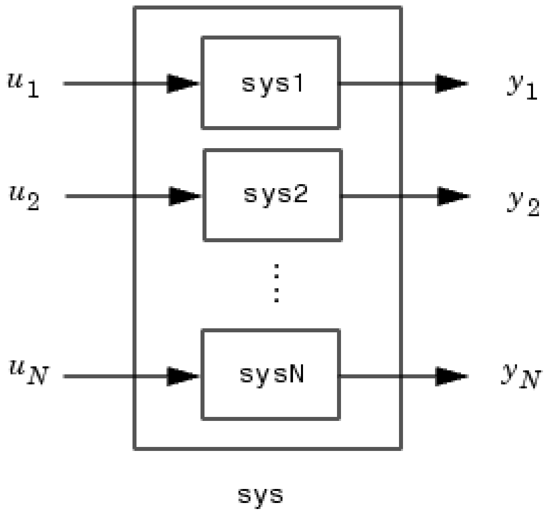
Group frac_poly_exp models by appending their inputs and outputs

Syntax

```
    sys = append(sys1,sys2,...,sysN)
```

Description

append appends the inputs and outputs of the LTI models sys1,...,sysN to form the augmented model sys depicted below.



Arguments

The input arguments sys1,..., sysN can be frac_poly_exp objects.

There is no limitation on the number of inputs.

Example

```
>> p1
s^2.2 +s^1.5
>> p2
s^0.6 +1
>> append(p1,p2)
Frac poly exp from input 1 to output:
#1 : s^2.2 +s^1.5
#2 : 0
Frac poly exp from input 2 to output:
#1 : 0
#2 : s^0.6 +1
```

binome

This function computes the generalized coefficients of the Newton binomial:
$factor = coef^{k}\ C_{n}^{k}$ (with k from 0 to leng-1)

Syntax

```
    factor=binome(n,leng,coef)
```

Arguments

Argument in :
n : order (complex vector)
leng : length of the output vector (integer scalar)
coef : coefficient

Argument out :
factor : Newton binomus (complex matrix)

Example

```
>> f=binome(0.7,7)
     1.0000
    -0.7000
    -0.1050
    -0.0455
    -0.0262
    -0.0173
    -0.0124

>> f=binome(0.7,7,1)
     1.0000
     0.7000
    -0.1050
     0.0455
    -0.0262
     0.0173
    -0.0124
```

binomial

This function computes the coefficients of the Newxton binomial :
$factor = C_n^k$

Syntax

```
    factor=binomial(n,k)
```

Arguments

Argument in :
n : order (integer scalar)
k : valor (integer scalar)

Argument out :
factor : Newton binomus (scalar)

Example

```
>> f=binomial(6,4)
    15
```

bode

Bode frequency response of fractional transfer functions

Syntax

```
bode(Tf)
bode(Tf ,W)
bode(Tf ,W)
[Mag ,Phase ,W]=bode(Tf)
[Mag ,Phase]=bode(Tf,W)
[Mag ,Phase]=bode(Tf,W,azp)
```

Description

Bode computes the magnitude and phase of the frequency response of fractional transfer functions. When invoked without left-hand arguments, bode produces a Bode plot on the screen.
Bode(T*f*) produces a Bode plot of the fractional transfer function T*f.* The frequency range is determined automatically based on the system poles and zeros.
Bode(Tf, W) explicitly specifies the frequency range or frequency points to be used for the plot. To focus on a particular frequency interval [Wmin,Wmax], set w = {Wmin,Wmax}. To use particular frequency points, set w to the vector of desired frequencies.
When invoked with left-hand arguments,

```
[Mag ,Phase ,W]=bode(Tf)
[Mag ,Phase]=bode(Tf,W)
```

return the magnitude and phase of the frequency response at the frequency W.

Arguments

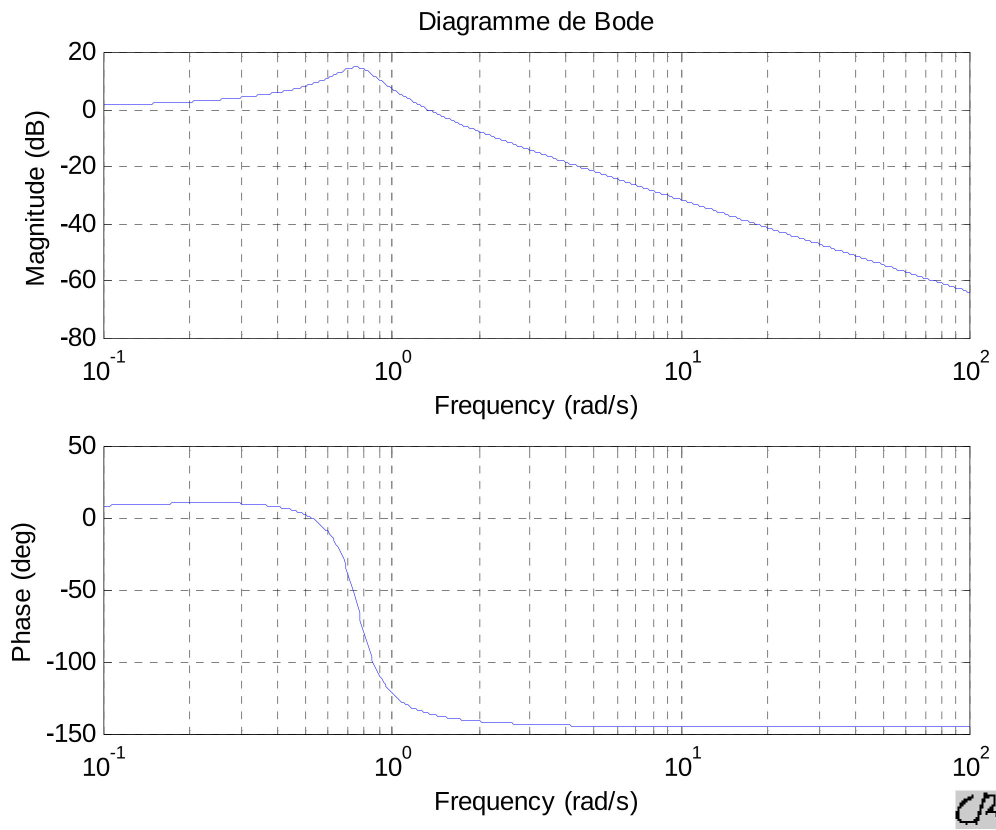
Argument in :
Tf : fractional object (frac_tf, frac_zpk, frac_ss or frac_poly_exp or frac_poly_imp)
W : frequency range (vector or cell)

Argument out :
Mag : magnitude (vector)
Phase : phase (vector)
W : frequency range (vector)

Example

```
» t can be
frac_poly_exp
s^2.2 +s^1.5

or frac_poly_imp
( s - 10  )^0.5

or frac_tf
  ( s^0.6 +1  )
------------------
( s^2.2 +s^1.5  )
```

```
or frac_zpk
2
----------------------------
 (s^0.5 - 0.01) (s^0.5 - 10)

or frac_ss
a =
        x1            x2
x1   10.01          -0.4
x2    0.25             0
b =
    x1
u1   4
u2   0
c =
    y1 y2
x1   0  2
d =
    y1
u1   0
order =
    0.5000

» bode(t)
» bode(t,{0.1,100})
» bode(t,logspace(-1,2,200))
gives a figure such as
```



```
>> [mag,phi,w]=bode(tft)
>> [mag,phi,w]=bode(tft,{0.1 100})
>> [mag,phi,w]=bode(tft,logspace(-1,2,20))
gives answer such as
mag =
33.3548
19.4582
11.3391
6.5930
3.8195
2.2012
1.2604
0.7166
0.4045
0.2270
0.1268
0.0706
0.0392
0.0218
0.0121
0.0067
0.0037
0.0021
0.0012
6.4313e-004


phi =
-134.2171
-134.5485
-135.0597
-135.7778
-136.7093
-137.8284
-139.0728
-140.3507
-141.5623
-142.6241
-143.4855
-144.1326
-144.5802
-144.8596
-145.0078
-145.0602
-145.0471
-144.9921
-144.9130
-144.8223


w =

  Columns 1 through 8
```

```
    0.1000      0.1438      0.2069      0.2976      0.4281
0.6158    0.8859    1.2743

  Columns 9 through 16

    1.8330      2.6367      3.7927      5.4556      7.8476
11.2884   16.2378   23.3572

  Columns 17 through 20

   33.5982   48.3293   69.5193  100.0000
```

char

Char converts fractional object to string.

Syntax

```
st = char(P)
```

Arguments

Argument in:
P: fractional object (frac_tf or frac_poly_exp or frac_poly_imp)

Argument out:
st: string

Example

```
>> fpe
s^2.2 +s^1.5
>> char(fpe)
ans =
s^2.2 +s^1.5

>> fpi
( s - 10  )^0.5
>> char(fpi)
ans =
 ( s - 10  )^0.5

>> tft
 transfer function :
  ( s^0.6 +1  )
------------------
( s^2.2 +s^1.5  )
>> [st1,st2,st3,st4]=char(tft)
st1 =
transfer function :
st2 =
  ( s^0.6 +1  )
st3 =
------------------
st4 =
( s^2.2 +s^1.5  )
```

clean

Sorts the orders of an explicit fractional object in the descending order and removes the null coefficients.

Syntax

```
Q = clean(P)
```

Arguments

Argument in:
P: frac_poly_exp, frac_poly_imp or frac_tf object

Argument out:
Q: frac_poly_exp, frac_poly_imp or frac_tf object

Example

```
>> p=frac_poly_exp([1 2 3 2 -1],[0.5 0.2 6 3 0.2]);
>> c = clean(p)
3 s^6 + 2 s^3 +s^0.5 +s^0.2
>>    tft=frac_tf(frac_poly_exp([1     1],[0     0.6]),
frac_poly_exp([1 1],[1.5 2.2]));
>> clean(tft)
 transfer function :
  ( s^0.6 +1  )
------------------
( s^2.2 +s^1.5  )
```

coef (frac_poly_exp)

Returns the coefficient of a frac_poly_exp object.

Syntax

```
    c = coef(P)
```

Arguments

Argument in:
P: frac_poly_exp object

Argument out:
c: coefficient of P (cell)

Example

```
>> p=frac_poly_exp([1 2 3 2 -1],[0.5 0.2 6 3 0.2]);
>> c=coef(p)
    [1x4 double]
>> c{1}
     3     2     1     1
```

commensurate(frac_poly_exp)

Computes de step order of a fractional explicit polynomial.

Syntax

```
    [New_order,Step_order]= commensurate(T)
    [New_order1, New_order2,Step_order]=
    commensurate(T1,T2)
```

Arguments

Argument in:
T : frac_poly_exp object

Argument out:
New_order: the integer order of T
New_order1: the integer orders of T1
New_order2: the integer orders of T2
Step_order: the step order (scalar)

Example

```
>> p
3 s^6 + 2 s^3 +s^0.5 +s^0.2
>> [new_ord,step_ord]=commensurate(p)
new_ord =
    [1x4 double]
step_ord =
    0.1000
>> new_ord{1}
ans =
    60    30     5     2

>> p1
s^2.2 +s^1.5
>> p2
s^0.6 +1
>> [new_ord1,new_ord2,step_ord]=commensurate(p1,p2)
new_ord1 =
    [1x2 double]
new_ord2 =
    [1x2 double]
step_ord =
    0.1000
>> new_ord1{1}
ans =
    22    15
>> new_ord2{1}
ans =
     6     0
```

commensurate(frac_tf)

Computes de step order of a fractional transfer function.

Syntax

```
[Step_order, new_tf]= commensurate(T)
```

Arguments

Argument in:
T : frac_tf object

Argument out:
Step_order: the step order (scalar)
new_tf: equivalent tf (tf object)

Example

```
>> tft
 transfer function :
  ( s^0.6 +1  )
------------------
( s^2.2 +s^1.5  )
>> [ord,ntf]=commensurate(tft)
ord =
    0.1000

ntf=
Transfer function:
  s^6 + 1
-----------
s^22 + s^15
```

den(frac_tf)

Quick access to the denominator of a fractional transfer function.

Syntax

```
D = den(T)
```

Arguments

Argument in :
T : fractional transfer function (frac_tf object)

Argument out :
D : T denominator (frac_poly_imp object)

Example

```
>> t
 transfer function :
    ( s^0.6 +1  )
---------------------
( s^2.2 +s^1.5 +1  )
>> den(t)
( s^2.2 +s^1.5 +1  )
```

dn

Syntax

```
D = dn(x,n,time)
```

Description

This function computes the fractional derivate of the data x to the order n, with n complex vector; time is the sampling period or the time vector.

Arguments

Argument in :
x : data (N*1 vector)
n : order (scalar)
t : time (scalar or N*1vector)

Argument out :
D : complex vector

Example

```
>> t=(0:0.1:0.9).';x=(1:10).';y=dn(t,x,0.5)
    3.1623
    4.7434
    5.9293
    6.9175
    7.7822
    8.5604
    9.2737
    9.9362
   10.5572
   11.1437
```

dnh

Syntax

```
D = dnh(x,n,time,level)
```

Description

This function computes the fractional derivate of the data x to the order n, with n complex vector; time is the sampling period or the time vector.
The variable 'level' is the approximation of the derivative ; the error is about h, h^2, etc ... according to level. [Levron2000]

Arguments

Argument in :
x : data (N*1 vector)
n : order (scalar)
t : time (scalar or N*1vector)
level : level (scalar ranging from 1 to 5)

Argument out :
D : complex vector

Example

```
>> t=(0:0.1:0.9).';x=(1:10).';y=dnh(t,x,0.5,3)
    4.3811
    4.9246
    6.1722
    7.1378
    7.9816
    8.7432
    9.4433
   10.0949
   10.7068
   11.2856
```

eig

Gives the eigenvalues of a fractional transfer function denominator or a fractional explicit polynomial and their commensurate orders.

Syntax

```
    [eigen_value,eigen_order]=eig(t)
```

Arguments

Argument in :
t : frac_tf or frac_poly_exp object
Argument out :
eigen_value : eigenvalues of *tf* (cell)
eigen_order : orders of the eigenvalues of *tf* (cell)

Example

```
» t
Transfer function:
    ( s^0.6 +1  )
---------------------
( s^2.2 +s^1.5 +1  )
or t
Fractional explicit polynomial
s^2.2 +s^1.5
>> eig(t)
ans =
    [22x1 double]
>> ans{1}
ans =
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
  -1.0000
  -0.6235 + 0.7818i
  -0.6235 - 0.7818i
   0.2225 + 0.9749i
   0.2225 - 0.9749i
   0.9010 + 0.4339i
   0.9010 - 0.4339i
```

eq

Test for equality

Syntax

```
A == B
D=eq(A, B)
```

Arguments

Argument in :
A : fractional object (frac_poly_exp, frac_poly_imp, frac_tf,, frac_zpk or frac_ss object)
B : fractional object (frac_poly_exp, frac_poly_imp, frac_tf,, frac_zpk or frac_ss object)

Argument out :
D : answer (boolean)

Examples

```
>> p1
s^2.2 +s^1.5
>> p2
s^0.6 +1
>> eq(p1,p1)
ans =
     1
>> eq(p1,p2)
ans =
     0
```

frac2int(frac_tf)

Quick access to the denominator of a fractional transfer function.

Syntax

```
    sys = frac2intT(fr)
```

Arguments

Argument in :
fr : fractional transfer function to approximate. (frac_tf object)

Argument out :
sys : T denominator (lti object)

Notice

If the fractional transfer function does not have its 'N' and 'band' property set the function will return an error.

Example

```
>> sys
 transfer function :
            ( s^0.5 +1  )
--------------------------------------
( s^2 - 3 s^1.5 + 6 s - 3 s^0.5 +1  )
with 5 zeros and poles and a band of [0.01 100] set for
a zero and pole approximation
>> sys=frac2int(sys)
Transfer function:
11 s^20 + 2848 s^19 + 2.968e005 s^18 + 1.609e007 s^17 +
4.944e008  s^16  + 9.026e009  s^15  + 1.011e011  s^14  +
7.062e011  s^13  +  3.097e012  s^12  +  8.556e012  s^11  +
1.501e013  s^10  +  1.676e013  s^9  +  1.188e013  s^8  +
5.305e012  s^7  +  1.493e012  s^6  +  2.635e011  s^5  +
2.86e010  s^4  +  1.849e009  s^3  +  6.884e007  s^2  +
1.358e006 s + 1.1e004
-------------------------------------------------------
7571 s^20 + 8.188e005 s^19 + 3.637e007 s^18 + 8.584e008
s^17 + 1.18e010 s^16 + 1.009e011 s^15 + 5.863e011 s^14
+  2.41e012  s^13  +  6.901e012  s^12+  1.328e013  s^11  +
1.663e013  s^10  +  1.328e013  s^9  +  6.901e012  s^8  +
2.41e012 s^7 + 5.863e011 s^6 + 1.009e011 s^5 + 1.18e010
s^4  +  8.584e008  s^3  +  3.637e007  s^2  +  8.188e005  s  +
7571
```

frac_poly_exp

Create an explicit fractional polynomial.

Syntax

```
Sys=frac_poly_exp(Coef,order,variable,N,band)
```

Description

frac_poly_exp(coef) creates an explicit polynomial with fractional orders. Coefficient is specified by “coef” a scalar in this case. The resulting system is a frac_poly_exp object.
frac_poly_exp(coef, order) creates an explicit polynomial with fractional orders. Coefficients and orders of this polynomial are specified by the vectors “coef” and “order” given in parameters to the function. The resulting system is a frac_poly_exp object.
frac_poly_exp(coef, order, variable) creates an explicit polynomial with fractional orders. Coefficients and orders of this polynomial are specified by the vectors “coef” and “order” given in parameters to the function and the variable of the polynomial is specified with the "variable" string. The resulting system is a frac_poly_exp object.
frac_poly_exp(coef, order, variable, N, band) creates an explicit polynomial with fractional orders. Coefficients and orders of this polynomial are specified by the vectors “coef” and “order” given in parameters to the function and the variable of the polynomial is specified with the "variable" string. The band and the number of poles and zeros for the zeros and poles approximation are also stored in the object with the "N" and "band" variables. The resulting system is a frac_poly_exp object.

Arguments

Argument in :
coef: coefficients of the polynomial (row vector or cell array of row vector)
order: orders of the polynomial (row vector or cell array of row vector)
variable: variable of the polynomial (string can be: 's', 'p', 'q', 'z')
N: number of pole and zero for the zero and pole approximation of the polynomial (scalar)
band: band for the zero and pole approximation of the polynomial (1*2 row vector)

Argument out :
sys : explicit fractional polynomial (frac_poly_exp object)

Examples

```
>> p=frac_poly_exp creates an empty frac_poly_exp
>> p
3 s^6 + 2 s^3 +s^0.5 +s^0.2
>> q=frac_poly_exp(p)
3 s^6 + 2 s^3 +s^0.5 +s^0.2
```

```
>> P=frac_poly_exp([1 1 1],[2 0.2 0])
s^2 +s^0.2 +1

>> P=frac_poly_exp([1 1 1],[2 0.2 0],'z')
z^2 +z^0.2 +1

>> P=frac_poly_exp([1 1 1],[2 0.2 0],'z',7,[0.01 1000])
z^2  +z^0.2  +1  but  the  object  also  contains  the
necessary   information   for   a   zeros   and   poles
approximation with 7poles and zeros on a band of [0.001
1000].

>>  P=frac_poly_exp({[1  1],[1  1];[1  1],[1  1]},{[0.4
0],[0.5 0];[0.7 0],[0.8 0]})
Frac poly exp from input 1 to output:
#1 : s^0.4 +1
#2 : s^0.5 +1
Frac poly exp from input 2 to output:
#1 : s^0.7 +1
#2 : s^0.8 +1

>>  P=frac_poly_exp({[1  1],[1  1];[1  1],[1  1]},{[0.4
0],[0.5 0];[0.7 0],[0.8 0]},'z')
Frac poly exp from input 1 to output:
#1 : z^0.4 +1
#2 : z^0.5 +1
Frac poly exp from input 2 to output:
#1 : z^0.7 +1
#2 : z^0.8 +1

>>  P=frac_poly_exp({[1  1],[1  1];[1  1],[1  1]},{[0.4
0],[0.5 0];[0.7 0],[0.8 0]},'z',7,[0.001 1000])
Frac poly exp from input 1 to output:
#1 : z^0.4 +1
#2 : z^0.5 +1
Frac poly exp from input 2 to output:
#1 : z^0.7 +1
#2  :  z^0.8  +1  but  the  object  also  contains  the
necessary   information   for   a   zeros   and   poles
approximation with 7poles and zeros on a band of [0.001
1000].
```

frac_poly_imp

Create an implicit fractional polynomial.

Syntax

```
sys =
frac_poly_imp(fpe,implicit_order,variable,N,band)
```

Description

Frac_poly_imp(*fpe, implicit_order*) creates an implicit fractional polynomial, i.e. a polynomial which is under the form $(fpe)^{implicit\_order}$ and the variable of the polynomial is specified with the "variable" string. The resulting system is a frac_poly_imp object. If imp_order is not equal to 1 then fpe must be of the form a*s+b else frac_poly_imp will return an error.
Frac_poly_imp(*fpe, implicit_order, variable*) creates an implicit fractional polynomial, i.e. a polynomial which is under the form $(fpe)^{implicit\_order}$. The resulting system is a frac_poly_imp object.
Frac_poly_imp(*fpe, implicit_order, variable, N, band*) creates an implicit fractional polynomial, i.e. a polynomial which is under the form $(fpe)^{implicit\_order}$ and the variable of the polynomial is specified with the "variable" string. The band and the number of poles and zeros for the zeros and poles approximation are also stored in the object with the "N" and "band" variables. The resulting system is a frac_poly_imp object.

Arguments

Argument in :
fpe: first order explicit polynomial or explicit polynomial if the implicit order is set to 0 (frac_poly_exp object(size(1*M) or cell array of frac_poly_exp object(size(1*M))
implicit_order: fractional polynomial order (row vector or cell array of row vectors)
variable: variable of the polynomial (string can be: 's', 'p', 'q', 'z')
N: number of pole and zero for the zero and pole approximation of the polynomial (scalar)
band: band for the zero and pole approximation of the polynomial (1*2 row vector)

Argument out :
sys: implicit fractional polynomial (frac_poly_imp object)

Example

```
>> frac_poly_imp creates an empty polynomial
>> fpi
```

```
( s - 10  )^0.5
>> fpi=frac_poly_imp(fpi)
( s - 10  )^0.5

>> fpe
s - 10
>> fpi=frac_poly_imp(fpe,0.5)
( s - 10  )^0.5

>> fp
Frac poly exp from input 1 to output:
#1 : s - 10
#2 : s - 10
#3 : s - 10
>> fpe2=frac_poly_imp(fp,[0.2 0.5 0.8])
( s - 10  )^0.2 ( s - 10  )^0.5 ( s - 10  )^0.8

>> fpe
s - 10
>> fpi=frac_poly_imp(fpe,0.5,'q')
( q - 10  )^0.5

>> fpe
s - 10
>> fpi=frac_poly_imp(fpe,0.5,'z',7,[0.01 100])
( z - 10  )^0.5  but the object also contains the
necessary information for a zeros and poles
approximation with 7zeros and poles on a band of [0.01
100].

>> fpe
s - 10
>>
fpimm=frac_poly_imp({fpe,fpe;fpe,fpe},{0.2,0.3;0.7,0.8}
)
Frac poly imp from input 1 to output:
#1 : ( s - 10  )^0.2
#2 : ( s - 10  )^0.3
Frac poly imp from input 2 to output:
#1 : ( s - 10  )^0.7
#2 : ( s - 10  )^0.8

>> fpe
s - 10
>>
fpimm=frac_poly_imp({fpe,fpe;fpe,fpe},{0.2,0.3;0.7,0.8}
,'z')
Frac poly imp from input 1 to output:
#1 : ( z - 10  )^0.2
#2 : ( z - 10  )^0.3
Frac poly imp from input 2 to output:
#1 : ( z - 10  )^0.7
#2 : ( z - 10  )^0.8

>> fpe
s - 10
```

```
>>
fpimm=frac_poly_imp({fpe,fpe;fpe,fpe},{0.2,0.3;0.7,0.8}
,'z',11,[0.001 1000])
Frac poly imp from input 1 to output:
#1 : ( z - 10  )^0.2
#2 : ( z - 10  )^0.3
Frac poly imp from input 2 to output:
#1 : ( z - 10  )^0.7
#2 : ( z - 10  )^0.8 but the object also contains the
necessary   information   for   a   zeros   and   poles
approximation with 11zeros and poles on a band of [0.0
1 1000].
```

frac_ss

Create a state space form.

Syntax

```
    Sys=frac_ss(A,B,C,D,order)
```

Description

frac_ss is used to create real- or complex-valued state-space models (frac_ss objects).

sys = ss(A,B,C,D,order) creates a state-space model

$$D^{order}x = Ax + Bu$$

$$y = Cx + Du$$

For a model with Nx states, Ny outputs, and Nu inputs: a is an Nx-by-Nx real- or complex-valued matrix. b is an Nx-by-Nu real- or complex-valued matrix. c is an Ny-by-Nx real- or complex-valued matrix. d is an Ny-by-Nu real- or complex-valued matrix.

Arguments

Argument in:

For a model with Nx states, Ny outputs, and Nu inputs
A: Nx-by-Nx real- or complex-valued matrix
B: Nx-by-Nu real- or complex-valued matrix
C: Ny-by-Nx real- or complex-valued matrix
D: Ny-by-Nu real- or complex-valued matrix
order: scalar

Argument out:

S*ys*: fractional transfer function. (frac_ss object)

Example

```
>> frac_ss Creates an empty frac_ss

>> fzpk
Fractionnal continuous-time zero-pole-gain system :
  2
  ----------------------------
  (s^0.5 - 0.01) (s^0.5 - 10)
>> frac_ss(fzpk)
a =
       x1          x2
x1   10.01        -0.4
x2    0.25           0
b =
    x1
u1   4
```

```
u2   0
c =
    y1 y2
x1   0  2
d =
    y1
u1   0
order =

    0.5000

>> ftf
 transfer function :
         ( 2  )
---------------------------
( s - 10.01 s^0.5 + 0.1  )
>> frac_ss(ftf)
a =
        x1            x2
x1   10.01          -0.4
x2    0.25             0
b =
    x1
u1   4
u2   0
c =
    y1 y2
x1   0  2
d =
    y1
u1   0
order =
    0.5000

>> A
   10.0100   -0.4000
    0.2500         0
>> B
     4
     0
>> C
     0     2
>> D
     0
>> order=0.5
    0.5000

>> frac_ss(A,B,C,D,order)
a =
        x1            x2
x1   10.01          -0.4
x2    0.25             0
b =
    x1
u1   4
u2   0
```

```
c =
    y1 y2
x1   0  2
d =
    y1
u1   0
order =
    0.5000

>> frac_ss(A,B,C,D,order,7,[0.01 100])
a =
        x1           x2
x1   10.01         -0.4
x2    0.25            0
b =
    x1
u1   4
u2   0
c =
    y1 y2
x1   0  2
d =
    y1
u1   0
order =
    0.5000
```

but the object also contains the necessary information for a zeros and poles approximation with 7 poles and zeros on a frequency band 0.01 to 100.

frac_tf

Create a fractional transfer function.

Syntax

```
Sys=frac_tf(P1,P2,variable,N,band)
```

Description

Frac_tf (*P1, P2*) creates a fractional transfer function, i.e. which is under the form $\frac{P1}{P2}$. The resulting system is a frac_tf object.
Frac_tf (*P1, P2, variable*) creates a fractional transfer function, i.e. which is under the form $\frac{P1}{P2}$.The resulting system is a frac_tf object.
Frac_tf (*P1, P2, N, band, variable*) creates a fractional transfer function, i.e. which is under the form $\frac{P1}{P2}$ and the variable of the polynomial is specified with the "variable" string. The band and the number of poles and zeros for the zeros and poles approximation are also stored in the object with the "N" and "band" variables. The resulting system is a frac_tf object.
In each case P1 and P2 can be frac_poly_exp or frac_poly_imp objects.

Arguments

Argument in:
P1: numerator (frac_poly_imp or frac_poly_exp)
P2: denominator (frac_poly_imp or frac_poly_exp)
variable: variable of the transfer function (string can be: 's', 'p', 'q', 'z')
N: number of pole and zero for the zero and pole approximation of the transfer function (scalar)
band: band for the zero and pole approximation of the transfer function (1*2 row vector)

Argument out:
S*ys*: fractional transfer function. (Frac_tf object)

Example

```
>> frac_tf Creates an empty frac_tf

>> fzpk
Fractionnal continuous-time zero-pole-gain system :
  2
  ----------------------------
  (s^0.5 - 0.01) (s^0.5 - 10)
>> frac_tf(fzpk)
 transfer function :
          ( 2  )
---------------------------
```

```
( s - 10.01 s^0.5 + 0.1  )
>> fss
a =
        x1           x2
x1   10.01         -0.4
x2    0.25            0
b =
    x1
u1   4
u2   0
c =
    y1 y2
x1   0  2
d =
    y1
u1   0
order =
    0.5000

>> frac_tf(fss)
 transfer function :
          ( 2  )
---------------------------
( s - 10.01 s^0.5 + 0.1  )

>> p1
s^2.2 +s^1.5
>> p2
s^0.6 +1
>> frac_tf([p1,p1;p2,p2],[p2,p2;p1,p1])
Frac tf from input 1 to output:
#1 :  transfer function :
( s^2.2 +s^1.5  )
------------------
  ( s^0.6 +1  )
#2 :  transfer function :
( s^2.2 +s^1.5  )
------------------
  ( s^0.6 +1  )
Frac tf from input 2 to output:
#1 :  transfer function :
  ( s^0.6 +1  )
------------------
( s^2.2 +s^1.5  )
#2 :  transfer function :
  ( s^0.6 +1  )
------------------
( s^2.2 +s^1.5  )

>> frac_tf([p1,p1;p2,p2],[p2,p2;p1,p1],'s')
Frac tf from input 1 to output:
#1 :  transfer function :
( s^2.2 +s^1.5  )
------------------
  ( s^0.6 +1  )
#2 :  transfer function :
```

```
( s^2.2 +s^1.5  )
------------------
  ( s^0.6 +1  )
Frac tf from input 2 to output:
#1 :  transfer function :
  ( s^0.6 +1  )
------------------
( s^2.2 +s^1.5  )
#2 :  transfer function :
  ( s^0.6 +1  )
------------------
( s^2.2 +s^1.5  )

>>      frac_tf([p1,p1;p2,p2],[p2,p2;p1,p1],'s',7,[0.01
100])
Frac tf from input 1 to output:
#1 :  transfer function :
( s^2.2 +s^1.5  )
------------------
  ( s^0.6 +1  )
#2 :  transfer function :
( s^2.2 +s^1.5  )
------------------
  ( s^0.6 +1  )
Frac tf from input 2 to output:
#1 :  transfer function :
  ( s^0.6 +1  )
------------------
( s^2.2 +s^1.5  )
#2 :  transfer function :
  ( s^0.6 +1  )
------------------
( s^2.2 +s^1.5  )
```

but the object also contains the necessary information for a zeros and poles approximation with 7 poles and zeros on a frequency band 0.01 to 100.

```
>> fpi
( s - 10  )^0.5
>> fpi2
( s - 0.1  )^0.8
>> frac_tf([fpi,fpi2;fpi,fpi2],[fpi2,fpi;fpi2,fpi])
Frac tf from input 1 to output:
#1 :  transfer function :
 ( s - 10  )^0.5
-----------------
( s - 0.1  )^0.8

#2 :  transfer function :
( s - 0.1  )^0.8
-----------------
 ( s - 10  )^0.5
Frac tf from input 2 to output:
#1 :  transfer function :
 ( s - 10  )^0.5
-----------------
```

```
( s - 0.1  )^0.8
#2 :  transfer function :
( s - 0.1  )^0.8
-----------------
 ( s - 10  )^0.5

>> frac_tf([fpi,fpi2;fpi,fpi2],[fpi2,fpi;fpi2,fpi],'s')
Frac tf from input 1 to output:
#1 :  transfer function :
 ( s - 10  )^0.5
-----------------
( s - 0.1  )^0.8
#2 :  transfer function :
( s - 0.1  )^0.8
-----------------
 ( s - 10  )^0.5
Frac tf from input 2 to output:
#1 :  transfer function :
 ( s - 10  )^0.5
-----------------
( s - 0.1  )^0.8
#2 :  transfer function :
( s - 0.1  )^0.8
-----------------
 ( s - 10  )^0.5

>>
frac_tf([fpi,fpi2;fpi,fpi2],[fpi2,fpi;fpi2,fpi],'s',9,[
0.001 1000])
Frac tf from input 1 to output:
#1 :  transfer function :
 ( s - 10  )^0.5
-----------------
( s - 0.1  )^0.8
#2 :  transfer function :
( s - 0.1  )^0.8
-----------------
 ( s - 10  )^0.5
Frac tf from input 2 to output:
#1 :  transfer function :
 ( s - 10  )^0.5
-----------------
( s - 0.1  )^0.8
#2 :  transfer function :
( s - 0.1  )^0.8
-----------------
 ( s - 10  )^0.5
```

but the object also contains the necessary information for a zeros and poles approximation with 9 poles and zeros on a frequency band 0.001 to 1000.

frac_zpk

Create a fractional zero pole gain form.

Syntax

```
Sys=frac_zpk(zero,pole,gain,order,variable,N,band)
```

Description

Frac_zpk (*zero,pole,gain,order*) creates a zero pole gain form, i.e. which is under the form $gain*\frac{\prod\left(s^{order}-zero\right)}{\prod\left(s^{order}-pole\right)}$. The resulting system is a frac_zpk object.

Frac_zpk (*zero,pole,gain,order,variable,N,band*) creates a zero pole gain form, i.e. which is under the form $gain*\frac{\prod\left(s^{order}-zero\right)}{\prod\left(s^{order}-pole\right)}$. The band and the number of poles and zeros for the zeros and poles approximation are also stored in the object with the "N" and "band" variables. The resulting system is a frac_zpk object.

Arguments

Argument in:
zero: zeros (row vector or cell array of row vectors)
pole: poles (row vector or cell array of row vectors)
gain: gain (scalar or cell array of scalar)
order: order (scalar or cell array of scalar)
variable: variable of the transfer function (string can be: 's', 'p', 'q', 'z')
N: number of pole and zero for the zero and pole approximation of the zero pole gain form (scalar)
band: band for the zero and pole approximation of the zero pole gain form (1*2 row vector)

Argument out:
S*ys*: fractional transfer function. (frac_zpk object)

Example

```
>> frac_zpk

>> ftf
 transfer function :
         ( 2  )
--------------------------
( s - 10.01 s^0.5 + 0.1  )

>> frac_zpk(ftf)
Fractionnal continuous-time zero-pole-gain system :
  2
  ---------------------------
  (s^0.5 - 10) (s^0.5 - 0.01)
```

```
>> fss
a =
        x1           x2
x1   10.01         -0.4
x2    0.25            0
b =
    x1
u1   4
u2   0
c =
    y1 y2
x1   0  2
d =
    y1
u1   0
order =
    0.5000

>> frac_zpk(fss)
Fractionnal continuous-time zero-pole-gain system :
  2
  ---------------------------
  (s^0.5 - 10) (s^0.5 - 0.01)

>> frac_zpk([0.01 10],[0.02 5 50],2,0.5)
Fractionnal continuous-time zero-pole-gain system :
  2 * (s^0.5 - 0.01) (s^0.5 - 10)
  -----------------------------------------
  (s^0.5 - 0.02) (s^0.5 - 5) (s^0.5 - 50)

>> frac_zpk([0.01 10],[0.02 5 50],2,0.5,7,[0.01 100])
Fractionnal continuous-time zero-pole-gain system :
  2 * (s^0.5 - 0.01) (s^0.5 - 10)
  -----------------------------------------
  (s^0.5 - 0.02) (s^0.5 - 5) (s^0.5 - 50)

>>   frac_zpk({[0.01   10],[0.01   10];[0.01   10],[0.01
10]},{[0.02  5  50],[0.02  5  50];[0.02  5  50],[0.02  5
50]},{2,3;7,8},{0.5,0.5;0.5,0.5})

Frac zpk from input 1 to output:
#1 : Fractionnal continuous-time zero-pole-gain system
  2 * (s^0.5 - 0.01) (s^0.5 - 10)
  -----------------------------------------
  (s^0.5 - 0.02) (s^0.5 - 5) (s^0.5 - 50)
#2 : Fractionnal continuous-time zero-pole-gain system
  3 * (s^0.5 - 0.01) (s^0.5 - 10)
  -----------------------------------------
  (s^0.5 - 0.02) (s^0.5 - 5) (s^0.5 - 50)
Frac zpk from input 2 to output:
#1 : Fractionnal continuous-time zero-pole-gain system
  7 * (s^0.5 - 0.01) (s^0.5 - 10)
  -----------------------------------------
  (s^0.5 - 0.02) (s^0.5 - 5) (s^0.5 - 50)
#2 : Fractionnal continuous-time zero-pole-gain system
  8 * (s^0.5 - 0.01) (s^0.5 - 10)
  -----------------------------------------
  (s^0.5 - 0.02) (s^0.5 - 5) (s^0.5 - 50)

>>   frac_zpk({[0.01   10],[0.01   10];[0.01   10],[0.01
10]},{[0.02  5  50],[0.02  5  50];[0.02  5  50],[0.02  5
50]},{2,3;7,8},{0.5,0.5;0.5,0.5},'z')

Frac zpk from input 1 to output:
#1 : Fractionnal continuous-time zero-pole-gain system
  2 * (z^0.5 - 0.01) (z^0.5 - 10)
  -----------------------------------------
  (z^0.5 - 0.02) (z^0.5 - 5) (z^0.5 - 50)
#2 : Fractionnal continuous-time zero-pole-gain system
  3 * (z^0.5 - 0.01) (z^0.5 - 10)
  -----------------------------------------
  (z^0.5 - 0.02) (z^0.5 - 5) (z^0.5 - 50)
Frac zpk from input 2 to output:
#1 : Fractionnal continuous-time zero-pole-gain system
  7 * (z^0.5 - 0.01) (z^0.5 - 10)
  -----------------------------------------
  (z^0.5 - 0.02) (z^0.5 - 5) (z^0.5 - 50)
#2 : Fractionnal continuous-time zero-pole-gain system
  8 * (z^0.5 - 0.01) (z^0.5 - 10)
  -----------------------------------------
  (z^0.5 - 0.02) (z^0.5 - 5) (z^0.5 - 50)

>>   frac_zpk({[0.01   10],[0.01   10];[0.01   10],[0.01
10]},{[0.02  5  50],[0.02  5  50];[0.02  5  50],[0.02  5
50]},{2,3;7,8},{0.5,0.5;0.5,0.5},'z',7,[0.01 100])

Frac zpk from input 1 to output:
#1 : Fractionnal continuous-time zero-pole-gain system
  2 * (z^0.5 - 0.01) (z^0.5 - 10)
  -----------------------------------------
  (z^0.5 - 0.02) (z^0.5 - 5) (z^0.5 - 50)
#2 : Fractionnal continuous-time zero-pole-gain system
  3 * (z^0.5 - 0.01) (z^0.5 - 10)
  -----------------------------------------
  (z^0.5 - 0.02) (z^0.5 - 5) (z^0.5 - 50)
Frac zpk from input 2 to output:
#1 : Fractionnal continuous-time zero-pole-gain system
  7 * (z^0.5 - 0.01) (z^0.5 - 10)
  -----------------------------------------
  (z^0.5 - 0.02) (z^0.5 - 5) (z^0.5 - 50)
#2 : Fractionnal continuous-time zero-pole-gain system
  8 * (z^0.5 - 0.01) (z^0.5 - 10)
  -----------------------------------------
  (z^0.5 - 0.02) (z^0.5 - 5) (z^0.5 - 50)
```

but the object also contains the necessary information for a zeros and poles approximation with 7 poles and zeros on a frequency band 0.01 to 100.

freqresp

Computes the frequency response of a fractional transfer function.

Syntax

```
G=freqresp(Tf,W)
```

Arguments

Argument in :
T*f* : fractional object (frac_tf, frac_zpk, frac_ss or frac_poly_imp)
W: frequency range (vector)

Argument out :
G : frequency response

Example

```
» t

Transfer function:
  ( s^0.6 +1  )
------------------
( s^2.2 +s^1.5  )

Fractional zpk
2
----------------------------
(s^0.5 - 0.01) (s^0.5 - 10)

Fractional state space
a =
        x1           x2
x1   10.01         -0.4
x2    0.25            0
b =
    x1
u1   4
u2   0
c =
    y1 y2
x1   0  2
d =
    y1
u1   0
order =
    0.5000

Fractional implicit polynomial
( s - 10  )^0.5

» freqresp(t,[1 10 50 100])
```

```
ans=
-0.7946 - 0.6787i
-0.0216 - 0.0153i
-0.0016 - 0.0011i
-5.2567e-004 -3.7052e-004i
```

gammac

Syntax

```
[Gz]=gammac(z)
```

Description

Complex Gamma function is valid in the entire complex plane. This routine uses the reflection formula to provide valid results for all z.

Arguments

Argument in :
z : complex matrix
Argument out :
Gz : complex matrix

Example

```
>> gammac will give a demo of the function
>> Gz=gammac(0.1+0.5i)
   0.0327 - 1.5758i
```

gammaic

Syntax

```
[Gz]=gammaic(x,z)
```

Description

Complex Incomplete Gamma function is valid in the entire complex plane. For real arguments, the result is the same results as Matlab function (within numerical error)

Arguments

Argument in :
x : real matrix
z : real matrix
Argument out :
Gz : complex matrix

Example

```
>> Gz=gammaic(0.1,0.5)
    0.3453
```

get(frac_poly_exp)

Query objects properties.

Syntax

```
get(fo,property)
```

Description

get(fo) returns all properties of the object and their current values.

get(fo,'PropertyName') returns the value of the property 'PropertyName' of the object identified by fo.

Arguments

Argument in :
fo : fractional object (frac_tf, frac_zpk, frac_ss, frac_poly_exp or frac_poly_imp)
property: property of the object (string)

Example

```
» fpe
s^2.2 +s^1.5
>> get(fpe)
Frac_poly_exp:
coef = 1*1 cell array
order = 1*1  cell array
N =
band = [ ]
>>c=get(fpe,'coef')
[1x2 double]
>> c{1}
1     1
>> o=get(fpe,'order')
[1x2 double]
>> o{1}
2.2000    1.5000
>> band=get(fpe,'band')
[]
>> N=get(fpe,'N')
[]
```

get(frac_poly_imp)

Query objects properties.

Syntax

```
get(fo,property)
```

Description

get(fo) returns all properties of the object and their current values.

get(fo,'PropertyName') returns the value of the property 'PropertyName' of the object identified by fo.

Arguments

Argument in :
fo : fractional object (frac_tf, frac_zpk, frac_ss, frac_poly_exp or frac_poly_imp)
property: property of the object (string)

Example

```
>> get(fpi)
Frac_poly_imp:
fpe = 1*1 cell array
imp_order = 1*1  cell array
N = 7
band = [1.000000e-002 100]
>> get(fpi,'fpe')
    [1x1 frac_poly_exp]
>> get(fpi,'imp_order')
    [0.5000]
>> get(fpi,'variable')
s
>> get(fpi,'N')
     7
>> get(fpi,'band')
    0.0100  100.0000
```

get(frac_ss)

Query objects properties.

Syntax

```
get(fo,property)
```

Description

get(fo) returns all properties of the object and their current values.

get(fo,'PropertyName') returns the value of the property 'PropertyName' of the object identified by fo.

Arguments

Argument in :
fo : frac_ss
property: property of the object (string)

Example

```
>> get(fss)
Frac_ss:
A = 2x2 matrix
B = 2x1 matrix
C = 1x2 matrix
D = 1x1 matrix
order = 0.5

>> A=get(fss,'A')
   10.0100   -0.4000
    0.2500         0

>> B=get(fss,'B')
     4
     0

>> C=get(fss,'C')
    0     2

>> D=get(fss,'D')
     0

>> o=get(fss,'order')
    0.5000
```

get(frac_tf)

Query objects properties.

Syntax

```
get(fo,property)
```

Description

get(fo) returns all properties of the object and their current values.

get(fo,'PropertyName') returns the value of the property 'PropertyName' of the object identified by fo.

Arguments

Argument in :
fo : frac_tf
property: property of the object (string)

Example

```
>> sys
 transfer function :
  ( s^0.6 +1  )
------------------
( s^2.2 +s^1.5  )
>> get(sys)
Frac_tf:
num = 1*1 frac_poly_imp
den = 1*1 frac_poly_imp
N = 5
band = [1.000000e-002 100]

>> get(sys,'den')
( s^2.2 +s^1.5  )

>> get(sys,'num')
( s^0.6 +1  )

>> get(sys,'variable')
s

>> get(sys,'N')
     5

>> get(sys,'band')
    0.0100  100.0000
```

get(frac_zpk)

Query objects properties.

Syntax

```
get(fo,property)
```

Description

get(fo) returns all properties of the object and their current values.

get(fo,'PropertyName') returns the value of the property 'PropertyName' of the object identified by fo.

Arguments

Argument in :
fo : frac_zpk
property: property of the object (string)

Example

```
>> fzpk
Fractionnal continuous-time zero-pole-gain system :
  2
  ---------------------------
  (s^0.5 - 0.01) (s^0.5 - 10)
>> get(fzpk)
Frac_zpk:
eig_zero = 1*1 cell array
eig_poles = 1*1  cell array
k = 1*1  cell array
order = 1*1  cell array
N =
band = [ ]

>> z=get(fzpk,'eig_zero')
    {[]}

>> p=get(fzpk,'eig_poles')
    [1x2 double]
>> p{1}
    0.0100   10.0000

>> k=get(fzpk,'k')
    [2]
>> k{1}
     2

>> o=get(fzpk,'order')
    [0.5000]
>> o{1}
```

```
    0.5000

>> get(fzpk,'N')
     []

>> get(fzpk,'band')
     []
```

horzcat

Concatenate arrays horizontally

Syntax

C = horzcat(A1, A2, ...)

Description

C = horzcat(A1, A2, ...) horizontally concatenates A1, A2, and so on. All fractional object in the argument list must have the same number of rows.

horzcat concatenates N-dimensional fractional objects along the second dimension. The first and remaining dimensions must match.

MATLAB calls C = horzcat(A1, A2,...) for the syntax C = [A1 A2 ...] when any of A1, A2, etc., is a fractional object.

Arguments

Argument in :
A1: fractional object (M*N1 fractional object)
A2: fractional object (M*N2 fractional object)

Argument out :
C: fractional object (M*N fractional object)

Examples

```
>> sys
s^2.2 +s^1.5
>> fpecat=horzcat(sys, sys, sys)
Frac poly exp from input 1 to output:
#1 : s^2.2 +s^1.5
#2 : s^2.2 +s^1.5
#3 : s^2.2 +s^1.5
```

impulse

Impulse response of fractional objects.

Syntax

```
Rep=impulse(f,Time, method)
```

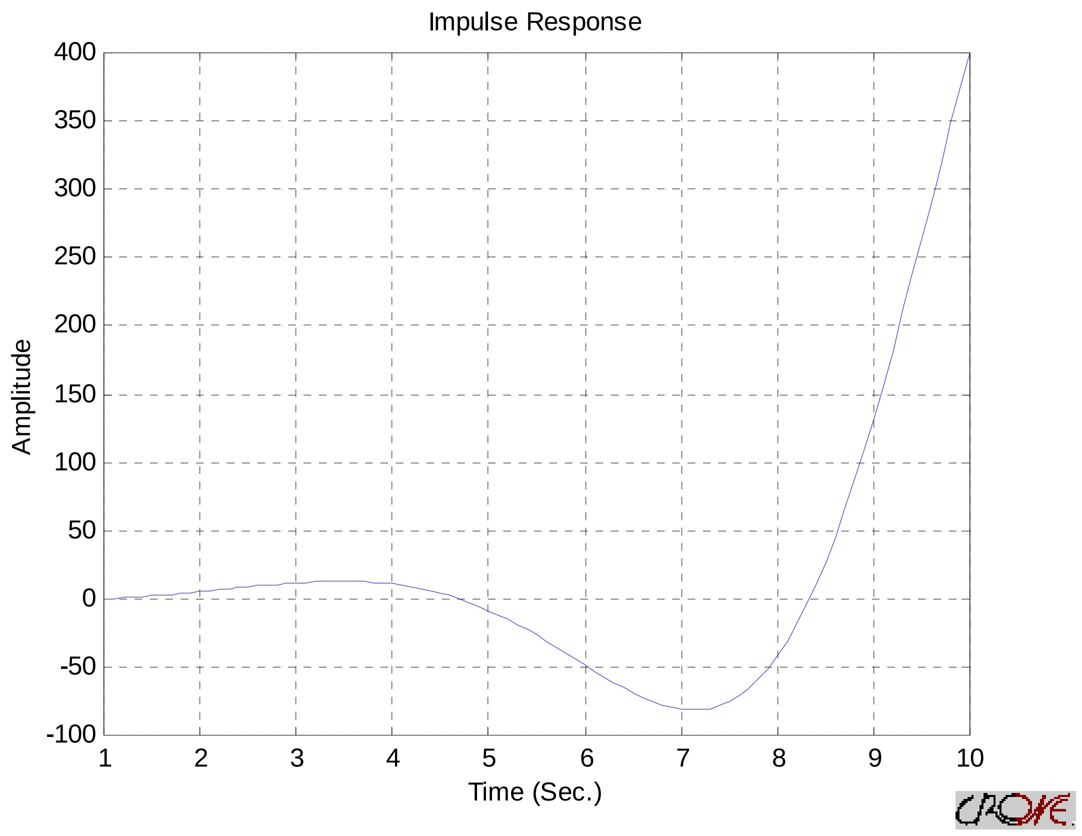
Arguments

Argument in :
f : fractional object (frac_tf or frac_poly_exp or frac_poly_imp)
Time : time vector (under the form Ti :Ts :Tf)
method : string (can be 'grun' for Grünwald, 'grunimp' for improved Grünwald, 'approx' for zero and pole approximation)

Argument out :
Rep : impulse response (Vector)

Example

```
» t can be
frac_poly_exp
s^2.2 +s^1.5

or frac_poly_imp
( s - 10  )^0.5

or frac_tf
  ( s^0.6 +1  )
------------------
( s^2.2 +s^1.5  )

or frac_zpk
2
----------------------------
 (s^0.5 - 0.01) (s^0.5 - 10)

or frac_ss
a =
        x1            x2
x1   10.01          -0.4
x2    0.25             0
b =
    x1
u1   4
u2   0
c =
    y1 y2
x1   0  2
d =
    y1
u1   0
order =
        0.5000

   » impulse(t,1:0.1:10)
```



iscomplex

Determine whether the frac_poly_exp, frac_poly_imp, frac_tf has complex coefficients and/or orders.

Syntax

```
    bool=iscomplex(sys)
```

Arguments

Argument in :
sys : frac_poly_exp or frac_poly_imp or frac_tf object

Argument out :
bool : answer (boolean)

Example

```
>> fpe
s^2.2 +s^1.5
>> iscomplex(fpe)
0
```

isempty

Determine whether the frac_poly_exp or frac_poly_imp or frac_tf has empty coefficients and/or order.

Syntax

```
    bool=isempty(sys)
```

Arguments

Argument in :
sys : frac_poly_exp, frac_poly_imp, frac_tf or frac_zpk object

Argument out :
bool : answer (boolean)

Example

```
>> fpe
s^2.2 +s^1.5
>> isempty(fpe)
0
>> test=frac_poly_exp
>> isempty(test)
1
```

lsim

Simulate implicit fractional objects response to arbitrary inputs.

Syntax

```
    y=lsim(sys,u,time,method)
```

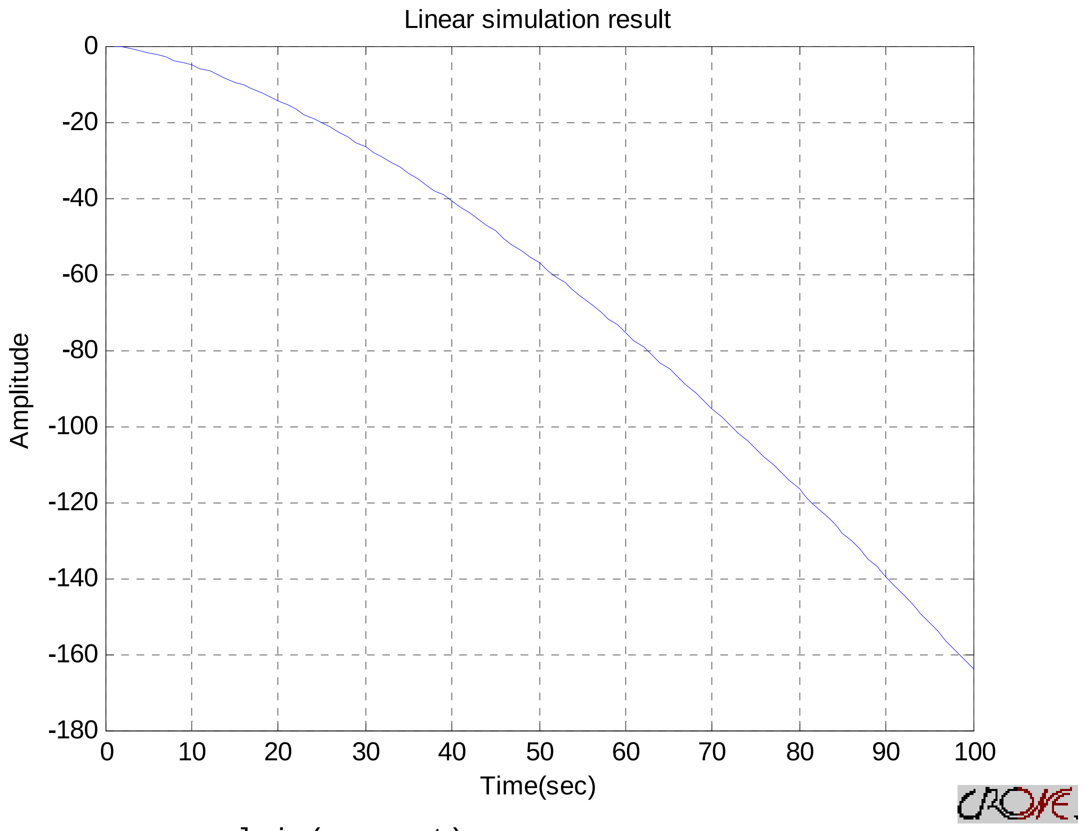
Arguments

Argument in :
sys : fractional object (frac_tf or frac_poly_exp or frac_poly_imp)
u : input signal (vector)
time : time vector (vector) or simple time(scalar)
method : string can be 'grun' for Grünwald, 'grunimp' for improved Grünwald or 'approx' for zero and pole approximation (only works on explicit system for the moment)

Argument out :
y : output response (complex matrix)

Example

```
» sys can be
frac_poly_exp
s^2.2 +s^1.5

or frac_poly_imp
( s - 10  )^0.5

or frac_tf
  ( s^0.6 +1  )
-----------------
( s^2.2 +s^1.5  )

or frac_zpk
2
---------------------------
 (s^0.5 - 0.01) (s^0.5 - 10)

or frac_ss
a =
        x1           x2
x1   10.01         -0.4
x2    0.25            0
b =
    x1
u1   4
u2   0
c =
    y1 y2
x1   0  2
d =
        y1
u1    0
order =
        0.5000

>> x=(1:10)';
>> t=(1:10)';
>> lsim(sys,x,t)
>> lsim(sys,x,t,'grun')
gives a figure such as
```



```
>> y=lsim(sys,x,t)
>> y=lsim(sys,x,t,'grun')
gives an answer such as
y =
             0
        4.0000
       -1.4000
        0.2900
        0.2340
        0.2152
        0.2013
        0.1898
        0.1799
        0.1714
```

minreal

Minimal realization or pole-zero cancelation

Syntax

```
    sysr = minreal(sys)
    sysr = minreal(sys,tol)
    sysr = minreal(sys,tol,str)
```

Description

sysr = minreal(sys) eliminates uncontrollable or unobservable state in state-space models, or cancels pole-zero pairs in transfer functions or zero-pole-gain models. The output sysr has minimal order and the same response characteristics as the original model sys.

sysr = minreal(sys,tol) specifies the tolerance used for state elimination or pole-zero cancellation. The default value is tol = sqrt(eps) and increasing this tolerance forces additional cancellations.

sysr = minreal(sys,tol,str) returns only a poles and zero simplification

Arguments

Argument in :
sys : fractional object (frac_tf or frac_zpk)
tol : tolerance
str : simplification string

Argument out :
sysr : fractional object (frac_tf or frac_zpk)

Example

```
>> test
 transfer function :
   ( 2 s^2 - 20 s^1.5 + 70 s - 100 s^0.5 + 48  )
-------------------------------------------------------
( s^3 - 23.1 s^2.5 + 207.1 s^2 - 909.3 s^1.5 + 2034.7 s
- 2192.4 s^0.5 + 882  )
>> minreal(test)
 transfer function :
              ( 2 s - 14 s^0.5 + 24  )
----------------------------------------------------
( s^2 - 20.1 s^1.5 + 144.8 s - 434.7 s^0.5 + 441  )
>> minreal(test,0.15)
 transfer function :
              ( 2 s - 14 s^0.5 + 24  )
----------------------------------------------------
( s^2 - 20.1 s^1.5 + 144.8 s - 434.7 s^0.5 + 441  )
>> minreal(test,0.15,'simplify')
```

```
 transfer function :
             ( 2 s - 14 s^0.5 + 24  )
----------------------------------------------------
( s^2 - 20.1 s^1.5 + 144.8 s - 434.7 s^0.5 + 441  )

>> test
Fractionnal continuous-time zero-pole-gain system :
2 * (s^0.5 - 1) (s^0.5 - 2) (s^0.5 - 3) (s^0.5 - 4)
-------------------------------------------------------
(s^0.5  -  1)  (s^0.5  -  2)  (s^0.5  -  2.1)  (s^0.5  -  5)
(s^0.5 - 6) (s^0.5 - 7)
>> minreal(test)
Fractionnal continuous-time zero-pole-gain system :
2 * (s^0.5 - 3) (s^0.5 - 4)
-------------------------------------------------------
(s^0.5 - 2.1) (s^0.5 - 5) (s^0.5 - 6) (s^0.5 - 7)
>> minreal(test,0.15)
Fractionnal continuous-time zero-pole-gain system :
2 * (s^0.5 - 3) (s^0.5 - 4)
-------------------------------------------------------
(s^0.5 - 2.1) (s^0.5 - 5) (s^0.5 - 6) (s^0.5 - 7)
>> minreal(test,0.15,'simplify')
Fractionnal continuous-time zero-pole-gain system :
2 * (s^0.5 - 3) (s^0.5 - 4)
-------------------------------------------------------
(s^0.5 - 2.1) (s^0.5 - 5) (s^0.5 - 6) (s^0.5 - 7)
```

ne

Test for inequality

Syntax

```
A ~= B
bool=ne(A, B)
```

Description

A ~= B compares each element of A with the corresponding element of B, and returns a logical 1 (true) if A and B are unequal, or logical 0 (false) if they are equal. Each input of the expression can be a fractional object.

ne(A, B) is called for the syntax A ~= B when either A or B is an object.

Arguments

Argument in :
A : fractional object (frac_tf or frac_poly_exp or frac_poly_imp)
B : fractional object (frac_tf or frac_poly_exp or frac_poly_imp)

Argument out :
bool : boolean

Example

```
» sys can be
frac_poly_exp
s^2.2 +s^1.5

or frac_poly_imp
( s - 10  )^0.5

or frac_tf
  ( s^0.6 +1  )
------------------
( s^2.2 +s^1.5  )

or frac_zpk
2
----------------------------
 (s^0.5 - 0.01) (s^0.5 - 10)

or frac_ss
a =
        x1            x2
x1   10.01          -0.4
x2    0.25             0
b =
    x1
u1   4
u2   0
```

```
c =
    y1 y2
x1   0  2
d =
    y1
u1   0
order =
    0.5000

>> sys~=(sys+sys)
ans =
     1
>> sys~=sys
ans =
     0
```

nichols

Compute Nichols frequency responses of fractionals models

Syntax

```
    nichols(Tf)
    nichols(Tf ,W)
    [Mag ,Phase ,W]=nichols(Tf)
    [Mag ,Phase]=nichols(Tf,W)
```

Description

Nichols computes the frequency response of fractional transfer functions and plots it in the Nichols coordinates.
Nichols(T*f*) produces a Nichols plot of the fractional transfer function T*f*. The frequency range is determined automatically based on the system poles and zeros.
Nichols(Tf, W) explicitly specifies the frequency range or frequency points to be used for the plot. To focus on a particular frequency interval [Wmin,Wmax], set w = {Wmin,Wmax}. To use particular frequency points, set w to the vector of desired frequencies.
When invoked with left-hand arguments,

```
    [Mag ,Phase ,W]=nichols(Tf)
    [Mag ,Phase]=nichols(Tf,W)
```

return the magnitude and phase of the frequency response at the frequency W.

Arguments

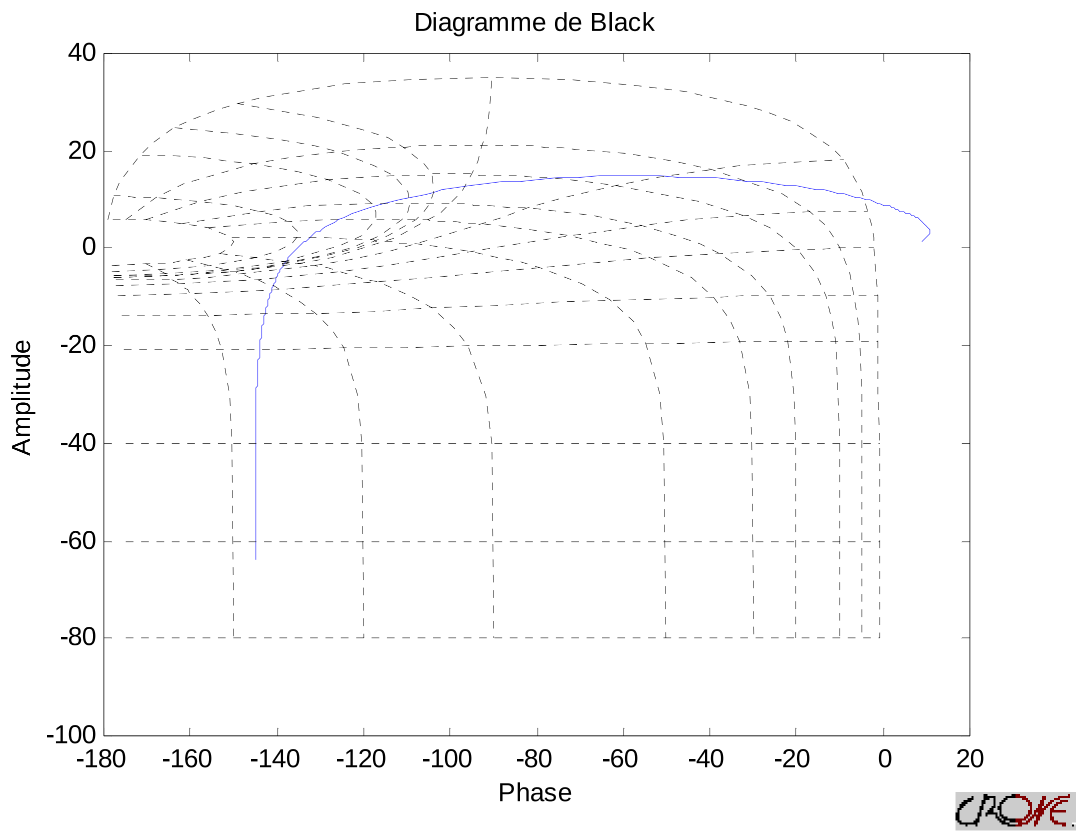
Argument in :
Tf : fractional object (frac_tf, frac_zpk, frac_ss or frac_poly_exp or frac_poly_imp)
W : frequency range (vector or cell)

Argument out :
G : magnitude (vector)
Phase : phase (vector)
W : frequency range (vector)

Example

```
» sys can be
frac_poly_exp
s^2.2 +s^1.5

or frac_poly_imp
( s - 10  )^0.5

or frac_tf
  ( s^0.6 +1  )
-----------------
( s^2.2 +s^1.5  )
```

```
or frac_zpk
2
----------------------------
 (s^0.5 - 0.01) (s^0.5 - 10)

or frac_ss
a =
        x1            x2
x1   10.01          -0.4
x2    0.25             0
b =
    x1
u1   4
u2   0
c =
    y1 y2
x1   0  2
d =
    y1
u1   0
order =
    0.5000

>> nichols(sys)
>> nichols(sys,{0.1 100})
>> nichols(sys,logspace(-1,2,100))
gives
```



```
>> [mag,phi,w]=nichols(sys)
>> [mag,phi,w]=nichols(sys,{0.1 100})
>> [mag,phi,w]=nichols(sys,logspace(-1,2,10))
gives an answer such as
mag =
0.0349
0.1194
0.4327
1.7053
7.4044
34.9802
175.1272
907.4797
4.7934e+003
2.5588e+004

phi =
144.2585
149.7569
157.3667
166.5000
175.6333
183.2431
188.7415
192.3654
194.6302
196.0045

w =
Columns 1 through 8
0.1000      0.2154      0.4642      1.0000      2.1544
4.6416   10.0000   21.5443
Columns 9 through 10
46.4159  100.0000
```

norm

Calculate the norm of fractional model.

Syntax

```
        [N] = Norm(Sys)
```

Arguments

Argument in:
Sys : fractional model(frac_tf, frac_ss, frac_zpk object)

Argument out:
N: norm of *Sys*.

Notice

If the fractional transfer function given in parameters to the function "norm" isn't stable, the result is "NaN".

Example

```
>> sys
frac_tf :
    ( 8 s^0.55 + 13  )
--------------------------
( s^1.1 + 3 s^0.55 + 2  )

or frac_ss:
a =
     x1 x2
x1   -3 -2
x2    1  0
b =
    x1
u1   4
u2   0
c =
    y1          y2
x1   2        3.25
d =
    y1
u1   0
order =
    0.5500

or frac_zpk :
8 * (s^0.55 + 1.625)
----------------------------
(s^0.55 + 2) (s^0.55 + 1)
>> norm(sys)
13.2201
```

You can find a nice example of norm with the script `Ortho.m` which can be found in `math/demo&help` .

num

Quick access to fractional transfer function numerator.

Syntax

```
N = num(T)
```

Arguments

Argument in :
T : fractional transfer function (frac_tf object)

Argument out :
N : numerator of *T* (frac_poly_imp object)

Example

```
>> tft
 transfer function :
  ( s^0.6 +1  )
------------------
( s^2.2 +s^1.5  )
>> num(tft)
( s^0.6 +1  )
```

nyquist

Compute Nyquist frequency response of fractional models

Syntax

```
    nyquist(Tf)
    nyquist (Tf ,W)
    [Mag ,Phase ,W]= nyquist (Tf)
    [Mag ,Phase]= nyquist (Tf,W)
```

Description

Nyquist computes the frequency response of fractional objects and plots it in the Nyquist diagram.
Nichols(T*f*) produces a Nyquist plot of the fractional object T*f*. The frequency range is determined automatically based on the system poles and zeros.
Nichols(Tf, W) explicitly specifies the frequency range or frequency points to be used for the plot. To focus on a particular frequency interval [Wmin,Wmax], set w = {Wmin,Wmax}. To use particular frequency points, set w to the vector of desired frequencies.
When invoked with left-hand arguments,

```
    [Mag ,Phase ,W]=nichols(Tf)
    [Mag ,Phase]=nichols(Tf,W)
```

return the magnitude and phase of the frequency response at the frequency W.

Arguments

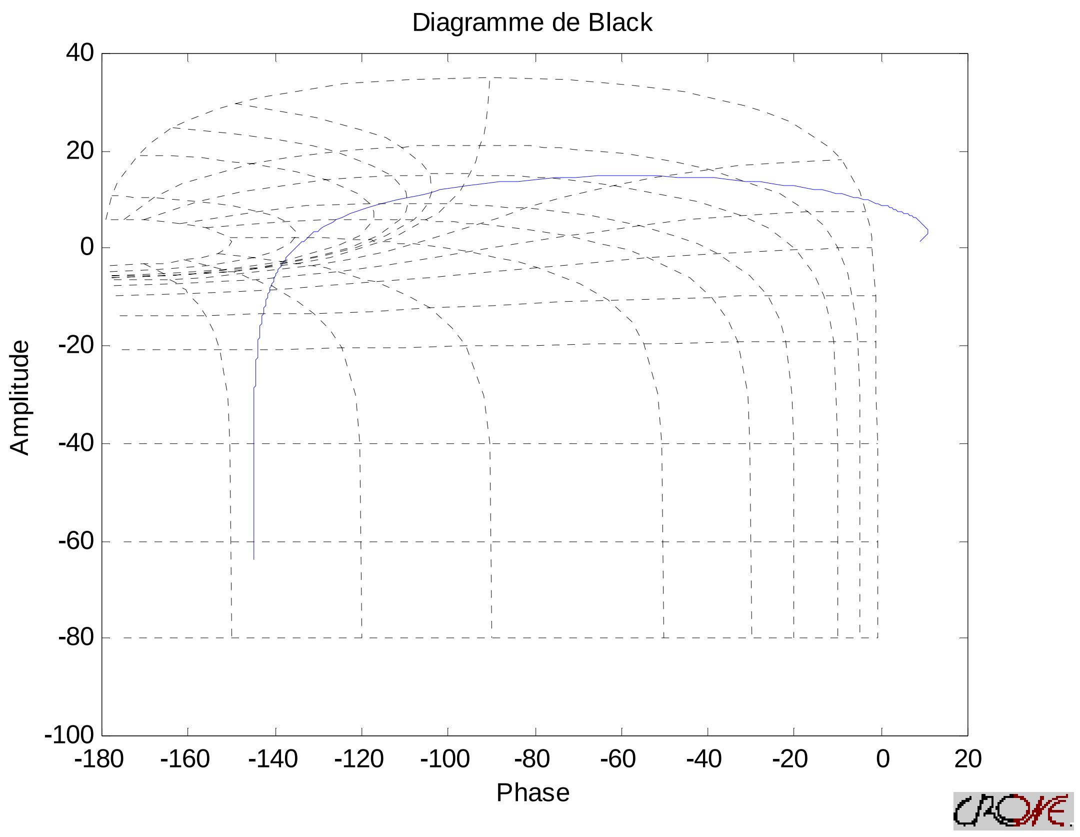
Argument in :
Tf : fractional object (frac_tf, frac_zpk, frac_ss or frac_poly_exp or frac_poly_imp)
W : frequency range (vector or cell)

Argument out :
Mag : magnitude (vector)
Phase : phase (vector)
W : frequency range (vector)

Example

```
» sys can be
frac_poly_exp
s^2.2 +s^1.5

or frac_poly_imp
( s - 10  )^0.5

or frac_tf
  ( s^0.6 +1  )
------------------
( s^2.2 +s^1.5  )
```

```
or frac_zpk
2
----------------------------
 (s^0.5 - 0.01) (s^0.5 - 10)

or frac_ss
a =
        x1            x2
x1   10.01          -0.4
x2    0.25             0
b =
    x1
u1   4
u2   0
c =
    y1 y2
x1   0  2
d =
    y1
u1   0
order =
    0.5000

>> nyquist(sys)
>> nyquist (sys,{0.1 100})
>> nyquist (sys,logspace(-1,2,100))
gives
```



```
>> [mag,phi,w]= nyquist (sys)
>> [mag,phi,w]= nyquist (sys,{0.1 100})
>> [mag,phi,w]= nyquist (sys,logspace(-1,2,10))
gives an answer such as
mag =
0.0349
0.1194
0.4327
1.7053
7.4044
34.9802
175.1272
907.4797
4.7934e+003
2.5588e+004

phi =
144.2585
149.7569
157.3667
166.5000
175.6333
183.2431
188.7415
192.3654
194.6302
196.0045

w =
Columns 1 through 8
0.1000      0.2154      0.4642      1.0000      2.1544
4.6416   10.0000   21.5443
Columns 9 through 10
46.4159                                      100.0000
```

order(frac_poly_exp)

Returns the orders of a frac_poly_exp object.

Syntax

```
    o = order(P)
```

Arguments

Argument in:
P: frac_poly_exp object

Argument out:
o: orders of P (cell)

Example

```
>> p=frac_poly_exp([1 2 3 2 -1],[0.5 0.2 6 3 0.2]);
>> o=order(p)
[1x4 double]
>> o{1}
6.0000    3.0000    0.5000    0.2000
```

parallel(frac_poly_exp)

Parallel connection of two frac_poly_exp models

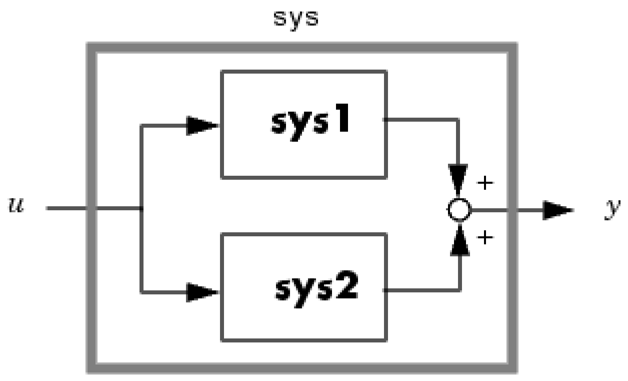
Syntax

```
sys = parallel(sys1,sys2,in1,in2,out1,out2)
sys = parallel(sys1,sys2)
```

Description

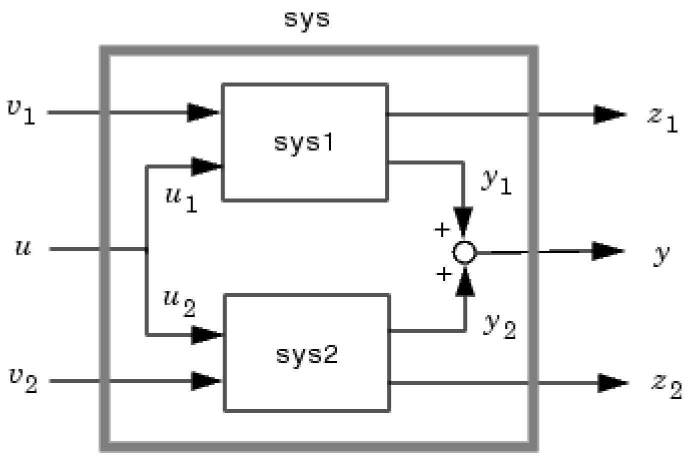
parallel connects two frac_poly_exp models in parallel.

sys = parallel(sys1,sys2) forms the basic parallel connection shown below.



This command is equivalent to the direct addition
sys = sys1 + sys2

sys = parallel(sys1,sys2,inp1,inp2,out1,out2) forms the more general parallel connection.



The index vectors inp1 and inp2 specify which inputs u_1 of sys1 and which inputs u_2 of sys2 are connected. Similarly, the index vectors out1 and out2 specify which outputs y_1 of sys1 and which outputs y_2 of sys2 are summed. The resulting model sys has [v_1, u, v_2] as inputs and [z_1, y, z_2] as outputs.

Arguments

Argument in :

sys1: fractional explicit polynomial (frac_poly_exp object)
sys2: fractional explicit polynomial (frac_poly_exp object)
in1: fractional explicit polynomial (vector)
in2: fractional explicit polynomial (vector)
out1: fractional explicit polynomial (vector)
out2: fractional explicit polynomial (vector)

Argument out :
sys : fractional explicit polynomial (frac_poly_exp object)

Example

```
>> sys
s^2.2 +s^1.5
>> parallel(sys,sys)
2 s^2.2 + 2 s^1.5

>> sys1
Frac poly exp from input 1 to output:
#1 : s^2.2 +s^1.5
#2 : s^2.2 +s^1.5
#3 : s^2.2 +s^1.5
#4 : s^2.2 +s^1.5
Frac poly exp from input 2 to output:
#1 : s^2.2 +s^1.5
#2 : s^2.2 +s^1.5
#3 : s^2.2 +s^1.5
#4 : s^2.2 +s^1.5
Frac poly exp from input 3 to output:
#1 : s^2.2 +s^1.5
```

```
#2 : s^2.2 +s^1.5
#3 : s^2.2 +s^1.5
#4 : s^2.2 +s^1.5
Frac poly exp from input 4 to output:
#1 : s^2.2 +s^1.5
#2 : s^2.2 +s^1.5
#3 : s^2.2 +s^1.5
#4 : s^2.2 +s^1.5
>> sys2
Frac poly exp from input 1 to output:
#1 : s^2.2 +s^1.5
#2 : s^2.2 +s^1.5
#3 : s^2.2 +s^1.5
Frac poly exp from input 2 to output:
#1 : s^2.2 +s^1.5
#2 : s^2.2 +s^1.5
#3 : s^2.2 +s^1.5
Frac poly exp from input 3 to output:
#1 : s^2.2 +s^1.5
#2 : s^2.2 +s^1.5
#3 : s^2.2 +s^1.5
Frac poly exp from input 4 to output:
#1 : s^2.2 +s^1.5
#2 : s^2.2 +s^1.5
#3 : s^2.2 +s^1.5

>> parallel(sys1,sys2,[1 2],[2 3],[3 4],[1 2])
Frac poly exp from input 1 to output:
#1 : s^2.2 +s^1.5
#2 : s^2.2 +s^1.5
#3 : s^2.2 +s^1.5
#4 : s^2.2 +s^1.5
#5 : 0
Frac poly exp from input 2 to output:
#1 : s^2.2 +s^1.5
#2 : s^2.2 +s^1.5
#3 : s^2.2 +s^1.5
#4 : s^2.2 +s^1.5
#5 : 0
Frac poly exp from input 3 to output:
#1 : s^2.2 +s^1.5
#2 : s^2.2 +s^1.5
#3 : 2 s^2.2 + 2 s^1.5
#4 : 2 s^2.2 + 2 s^1.5
#5 : s^2.2 +s^1.5
Frac poly exp from input 4 to output:
#1 : s^2.2 +s^1.5
#2 : s^2.2 +s^1.5
#3 : 2 s^2.2 + 2 s^1.5
#4 : 2 s^2.2 + 2 s^1.5
#5 : s^2.2 +s^1.5
Frac poly exp from input 5 to output:
#1 : 0
#2 : 0
#3 : s^2.2 +s^1.5
#4 : s^2.2 +s^1.5
```

```
#5 : s^2.2 +s^1.5
Frac poly exp from input 6 to output:
#1 : 0
#2 : 0
#3 : s^2.2 +s^1.5
#4 : s^2.2 +s^1.5
#5 : s^2.2 +s^1.5
```

poles(frac_tf)

Compute the poles of an frac_tf.

Syntax

```
[roots,eigen_value,order_step]=poles(p)
```

Arguments

Argument in :
P : frac_tf object.

Argument out :

roots	: poles of P (cells)
eigen_value	: eigenvalues of *P*(complex vector)
order_step	: step order (complex vector)

Example

```
>> sys
 transfer function :
  ( s^0.6 +1  )
------------------
( s^2.2 +s^1.5  )
>> [r,ev,eo]=roots(sys)
r =
    {1x1 cell}
ev =
    [22x1 double]
eo =
    [0.1000]

>> r{1}{1}
Empty matrix: 1-by-0

>> ev{1}
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
  -1.0000
  -0.6235 + 0.7818i
  -0.6235 - 0.7818i
   0.2225 + 0.9749i
   0.2225 - 0.9749i
   0.9010 + 0.4339i
   0.9010 - 0.4339i

>> eo{1}
0.1000
```

residue(frac_zpk)

Convert between partial fraction expansion and polynomial coefficients

Syntax

res = residue(sys)

Description

The residue function converts a zero, pole, gain form to a residue representation.

res = residue(sys) finds the residues, poles, and direct term of a partial fraction expansion of a frac_zpk.

Definition

If there are no multiple roots, then

$$\frac{b(s)}{a(s)} = \frac{r_1}{s-p_1} + \frac{r_2}{s-p_2} + \ldots + \frac{r_n}{s-p_n} + k(s)$$

If p(j) = ... = p(j+m-1) is a pole of multiplicity m, then the expansion includes terms of the form

$$\frac{r_j}{s-p_j} + \frac{r_{j+1}}{(s-p_j)^2} + \ldots + \frac{r_{j+m-1}}{(s-p_j)^m}$$

Arguments

Argument in :
sys : frac_zpk object.

Argument out :
res : residue (cells of frac_zpk)

Examples

```
>> fzpk
Fractionnal continuous-time zero-pole-gain system :
  2
  ---------------------------
  (s^0.5 - 0.01) (s^0.5 - 10)
>> res=residue(fzpk)
    [2x1 frac_zpk]
>> res{1}
Frac zpk from input 1 to output:
Fractionnal continuous-time zero-pole-gain system :
  -0.2002
  --------------
  (s^0.5 - 0.01)
```

```
Frac zpk from input 2 to output:
Fractionnal continuous-time zero-pole-gain system :
  0.2002
  --------------
  (s^0.5 - 10)
>> res=residue([fzpk, fzpk])
    [2x1 frac_zpk]    [2x1 frac_zpk]
>> res{1}
Frac zpk from input 1 to output:
Fractionnal continuous-time zero-pole-gain system :
  -0.2002
  --------------
  (s^0.5 - 0.01)
Frac zpk from input 2 to output:
Fractionnal continuous-time zero-pole-gain system :
  0.2002
  --------------
  (s^0.5 - 10)
>> res{2}
Frac zpk from input 1 to output:
Fractionnal continuous-time zero-pole-gain system :
  -0.2002
  --------------
  (s^0.5 - 0.01)
Frac zpk from input 2 to output:
Fractionnal continuous-time zero-pole-gain system :
  0.2002
  --------------
  (s^0.5 - 10)
```

roots

Compute the roots of an explicit fractional polynomial.

Syntax

```
[roots,eigen_value,order_step]=roots(p)
```

Arguments

Argument in :
P : frac_poly_exp, frac_poly_imp object.

Argument out :

roots	: roots of P (cells)
order_step	: step order (complex vector)
eigen_value	: eigenvalues of *P*(complex vector)

Example

```
>> fpe
s^2.2 +s^1.5
>> [r,ev,eo]=roots(fpe)
r =
    {1x1 cell}
ev =
    [22x1 double]
eo =
    [0.1000]

>> r{1}{1}
Empty matrix: 1-by-0

>> ev{1}
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
  -1.0000
  -0.6235 + 0.7818i
  -0.6235 - 0.7818i
   0.2225 + 0.9749i
   0.2225 - 0.9749i
   0.9010 + 0.4339i
   0.9010 - 0.4339i

>> eo{1}
0.1000
```

scalar

Computes the scalar product of two fractional transfer functions.

Syntax

```
    [C] = scalar(SYS1,SYS2)
```

Arguments

Argument in :
  SYS1, SYS2: fractional transfer functions (frac_tf or frac_zpk objects)

Argument out :
  C: scalar product of sys1 and sys2.

Example

```
>> t
    ( s^0.6 +1  )
---------------------
( s^2.2 +s^1.5 +1  )
>> s
 transfer function :
       ( 1  )
---------------------
( s^0.3 -s^0.1 +1  )
>> scalar(s,t)

ans =

    0.5038
```

series(frac_poly_exp)

Series connection of two frac_poly_exp models

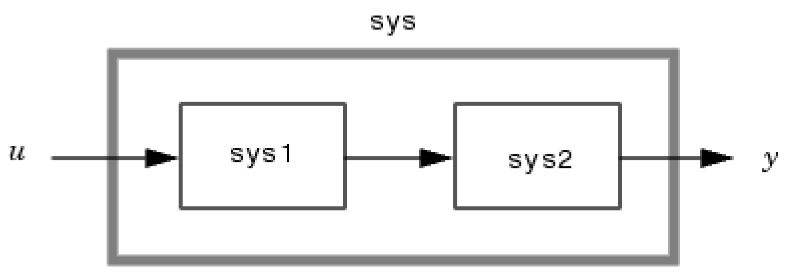
Syntax

```
sys = parallel(sys1,sys2,in1,in2,out1,out2)
sys = parallel(sys1,sys2)
```

Description

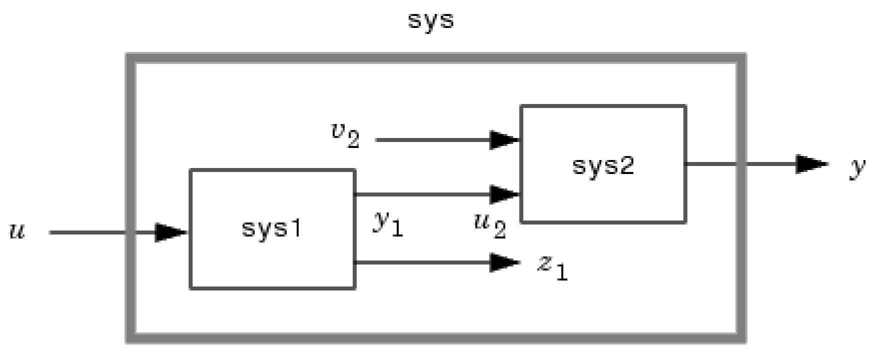
series connects two frac_poly_exp models in series.

sys = series(sys1,sys2) forms the basic series connection shown below.



This command is equivalent to the direct multiplication
sys = sys2 * sys1

sys = series(sys1,sys2,outputs1,inputs2) forms the more general series connection.



The index vectors outputs1 and inputs2 indicate which outputs y_1 of sys1 and which inputs u_2 of sys2 should be connected. The resulting model sys has u as input and y as output.

Arguments

Argument in :

sys1: fractional explicit polynomial (frac_poly_exp object)
sys2: fractional explicit polynomial (frac_poly_exp object)
in1: fractional explicit polynomial (vector)
in2: fractional explicit polynomial (vector)
out1: fractional explicit polynomial (vector)
out2: fractional explicit polynomial (vector)

Argument out :
sys : fractional explicit polynomial (frac_poly_exp object)

Example

```
>> sys
s^2.2 +s^1.5
>> series(sys,sys)
s^4.4 + 2 s^3.7 +s^3

>> sys1
Frac poly exp from input 1 to output:
#1 : s^2.2 +s^1.5
#2 : s^2.2 +s^1.5
#3 : s^2.2 +s^1.5
#4 : s^2.2 +s^1.5
Frac poly exp from input 2 to output:
#1 : s^2.2 +s^1.5
#2 : s^2.2 +s^1.5
#3 : s^2.2 +s^1.5
#4 : s^2.2 +s^1.5
Frac poly exp from input 3 to output:
#1 : s^2.2 +s^1.5
#2 : s^2.2 +s^1.5
#3 : s^2.2 +s^1.5
#4 : s^2.2 +s^1.5
Frac poly exp from input 4 to output:
#1 : s^2.2 +s^1.5
#2 : s^2.2 +s^1.5
#3 : s^2.2 +s^1.5
#4 : s^2.2 +s^1.5
>> sys2
Frac poly exp from input 1 to output:
#1 : s^2.2 +s^1.5
#2 : s^2.2 +s^1.5
#3 : s^2.2 +s^1.5
Frac poly exp from input 2 to output:
#1 : s^2.2 +s^1.5
#2 : s^2.2 +s^1.5
#3 : s^2.2 +s^1.5
Frac poly exp from input 3 to output:
#1 : s^2.2 +s^1.5
#2 : s^2.2 +s^1.5
#3 : s^2.2 +s^1.5
Frac poly exp from input 4 to output:
#1 : s^2.2 +s^1.5
```

```
#2 : s^2.2 +s^1.5
#3 : s^2.2 +s^1.5
>> series(sys1,sys2,[3 4],[1 2])
Frac poly exp from input 1 to output:
#1 : 2 s^4.4 + 4 s^3.7 + 2 s^3
#2 : 2 s^4.4 + 4 s^3.7 + 2 s^3
#3 : 2 s^4.4 + 4 s^3.7 + 2 s^3
#4 : 2 s^4.4 + 4 s^3.7 + 2 s^3
Frac poly exp from input 2 to output:
#1 : 2 s^4.4 + 4 s^3.7 + 2 s^3
#2 : 2 s^4.4 + 4 s^3.7 + 2 s^3
#3 : 2 s^4.4 + 4 s^3.7 + 2 s^3
#4 : 2 s^4.4 + 4 s^3.7 + 2 s^3
Frac poly exp from input 3 to output:
#1 : 2 s^4.4 + 4 s^3.7 + 2 s^3
#2 : 2 s^4.4 + 4 s^3.7 + 2 s^3
#3 : 2 s^4.4 + 4 s^3.7 + 2 s^3
#4 : 2 s^4.4 + 4 s^3.7 + 2 s^3
Frac poly exp from input 4 to output:
#1 : 2 s^4.4 + 4 s^3.7 + 2 s^3
#2 : 2 s^4.4 + 4 s^3.7 + 2 s^3
#3 : 2 s^4.4 + 4 s^3.7 + 2 s^3
#4 : 2 s^4.4 + 4 s^3.7 + 2 s^3
```

set(frac_poly_exp)

Query objects properties.

Syntax

```
set(fo,property,value)
```

Description

fo=set(fo,'PropertyName',PropertyValue,...) sets the named properties to the specified values on the object(s) identified by H.

Arguments

Argument in :
fo : fractional explicit polynomial (frac_poly_exp object)
PropertyName: property name of the object (string)
PropertyValue: property value (depends on the property)

Argument out :
fo : fractional explicit polynomial (frac_poly_exp object)

Example

```
>> fpe
s^2.2 +s^1.5
>> set(fpe,'coef',[2 2])
2 s^2.2 + 2 s^1.5
>> set(fpe,'order',[1.2 0.1])
s^1.2 +s^0.1
>> set(fpe,'Ts',0)
s^2.2 +s^1.5
>> set(fpe,'N',7)
s^2.2 +s^1.5
>> set(fpe,'band',[0.1 100])
s^2.2 +s^1.5
>> set(fpe,'variable','z')
z^2.2 +z^1.5
```

set(frac_poly_imp)

Query objects properties.

Syntax

```
set(fo,property,value)
```

Description

fo=set(fo,'PropertyName',PropertyValue,...) sets the named properties to the specified values on the object(s) identified by H.

Arguments

Argument in :
fo : frac_poly_imp object
PropertyName: property name of the object (string)
PropertyValue: property value (depends on the property)

Argument out :
fo : frac_poly_imp object

Example

```
>> fpi
( s - 10  )^0.5
>> set(fpi,'fpe',frac_poly_exp([1.5 -6],[1 0]))
( 1.5 s - 6  )^0.5
>> set(fpi,'imp_order',0.8)
( s - 10  )^0.8
>> set(fpi,'variable','z')
( z - 10  )^0.5
>> set(fpi,'N',8)
( s - 10  )^0.5
>> set(fpi,'band',[0.01 100])
( s - 10  )^0.5
```

set(frac_ss)

Query objects properties.

Syntax

```
set(fo,property,value)
```

Description

fo=set(fo,'PropertyName',PropertyValue,...) sets the named properties to the specified values on the object(s) identified by H.

Arguments

Argument in :
fo : frac_ss
PropertyName: property name of the object (string)
PropertyValue: property value (depends on the property)

Argument out :
fo : frac_ss

Example

```
>> set(fss,'A',[1 2; 3 4])
a =
    x1 x2
x1   1  2
x2   3  4
b =
    x1
u1   4
u2   0
c =
    y1 y2
x1   0  2
d =
    y1
u1   0
order =
    0.5000

>> set(fss,'B',[2;3])
a =
        x1          x2
x1   10.01        -0.4
x2    0.25           0
b =
    x1
u1   2
u2   3
```

```
c =
    y1 y2
x1   0  2
d =
    y1
u1   0
order =
    0.5000

>> set(fss,'C',[1,4])
a =
       x1            x2
x1   10.01          -0.4
x2    0.25             0
b =
    x1
u1   4
u2   0
c =
    y1 y2
x1   1  4
d =
    y1
u1   0
order =
    0.5000

>> set(fss,'D',[5])
a =
        x1            x2
x1   10.01          -0.4
x2    0.25             0
b =
    x1
u1   4
u2   0
c =
    y1 y2
x1   0  2
d =
    y1
u1   5
order =
    0.5000

>> set(fss,'order',0.8)
a =
        x1            x2
x1   10.01          -0.4
x2    0.25             0
b =
    x1
u1   4
u2   0
c =
    y1 y2
```

```
x1   0  2
d =
    y1
u1   0
order =
    0.8000
```

set(frac_tf)

Query objects properties.

Syntax

```
set(fo,property,value)
```

Description

fo=set(fo,'PropertyName',PropertyValue,...) sets the named properties to the specified values on the object(s) identified by H.

Arguments

Argument in :
fo : fractional object (frac_tf)
PropertyName: property name of the object (string)
PropertyValue: property value (depends on the property)

Argument out :
fo : fractional object (frac_tf)

Example

```
>>> tft
 transfer function :
  ( s^0.6 +1  )
------------------
( s^2.2 +s^1.5  )
>> num
( s^2.2 +s^1.5  )
>> set(tft,'num',num)
 transfer function :
( s^2.2 +s^1.5  )
------------------
( s^2.2 +s^1.5  )
>> den
( s^0.6 +1  )
>> set(tft,'den',den)
 transfer function :
( s^0.6 +1  )
--------------
( s^0.6 +1  )
>> set(tft,'variable','z')
 transfer function :
  ( z^0.6 +1  )
------------------
( z^2.2 +z^1.5  )
>> set(tft,'N',7)
 transfer function :
  ( s^0.6 +1  )
```

```
------------------
( s^2.2 +s^1.5  )
>> set(tft,'band',[0.01 100])
 transfer function :
  ( s^0.6 +1  )
------------------
( s^2.2 +s^1.5  )
```

set(frac_zpk)

Query objects properties.

Syntax

```
set(fo,property,value)
```

Description

fo=set(fo,'PropertyName',PropertyValue,...) sets the named properties to the specified values on the object(s) identified by H.

Arguments

Argument in :
fo : frac_zpk
PropertyName: property name of the object (string)
PropertyValue: property value (depends on the property)

Argument out :
fo : frac_zpk

Example

```
>> fzpk
Fractionnal continuous-time zero-pole-gain system :
  2
  ----------------------------
  (s^0.5 - 0.01) (s^0.5 - 10)
>> set(fzpk,'eig_zero',{[0.1 1]})
Fractionnal continuous-time zero-pole-gain system :
  2 * (s^0.5 - 0.1) (s^0.5 - 1)
  ----------------------------
  (s^0.5 - 0.01) (s^0.5 - 10)
>> set(fzpk,'eig_poles',{[0.1 1 100]})
Fractionnal continuous-time zero-pole-gain system :
  2
  ------------------------------------------
  (s^0.5 - 0.1) (s^0.5 - 1) (s^0.5 - 100)
>> set(fzpk,'k',{5})
Fractionnal continuous-time zero-pole-gain system :
  5
  ----------------------------
  (s^0.5 - 0.01) (s^0.5 - 10)
>> set(fzpk,'order',{1.2})
Fractionnal continuous-time zero-pole-gain system :
  2
  ----------------------------
  (s^1.2 - 0.01) (s^1.2 - 10)
>> set(fzpk,'N',7)
Fractionnal continuous-time zero-pole-gain system :
  2
  ----------------------------
  (s^0.5 - 0.01) (s^0.5 - 10)
>> set(fzpk,'band',[0.01 100])
Fractionnal continuous-time zero-pole-gain system :
  2
  ----------------------------
  (s^0.5 - 0.01) (s^0.5 - 10)
```

size

Fractional objets dimensions.

Syntax

```
d = size(sys)
[m,n] = size(sys)
```

Arguments

Argument in :
sys: fractional objects (frac_tf, frac_zpk, frac_ss, frac_poly_exp, frac_poly_imp objects)

Argument out :
d: vector.
n,m: scalar

Example

```
>> tft
 transfer function :
  ( s^0.6 +1  )
------------------
( s^2.2 +s^1.5  )
>> d=size(tft)
d =
     1     1
>> [n,m]=size(tft)
n =
     1
m =
     1
```

sort(frac_poly_exp)

Sort the orders of an explicit fractional polynomial in the descending order

Syntax

```
Q = sort(P)
```

Arguments

Argument in:
P: frac_poly_exp object

Argument out:
Q: frac_poly_exp object

Example

```
>> p=frac_poly_exp([1 2 3 2 -1],[0.5 0.2 6 3 0.2]);
>> c=sort(p)
3 s^6 + 2 s^3 +s^0.5 +s^0.2
```

ss2tf(frac_ss)

Convert state-space filter parameters to transfer function form

Syntax

tf = ss2tf(sys)

Description

tf = ssdata(sys) returns the transfer function tf.

Arguments

Argument in:
sys : state space form (frac_ss object)

Argument out:
tf : frac_tf

Example

```
>> fss
a =
        x1            x2
x1   10.01          -0.4
x2    0.25             0
b =
    x1
u1   4
u2   0
c =
    y1 y2
x1   0  2
d =
    y1
u1   0
order =
    0.5000

>> ss2tf(fss)
 transfer function :
          ( 2  )
---------------------------
( s - 10.01 s^0.5 + 0.1  )
```

ssdata(frac_ss)

Access state space data

Syntax

[A,B,C,D,order] = ssdata(sys)

Description

[A,B,C,D,order] = ssdata(sys) extracts the matrix (or multidimensional array) data A,B,C,D and order from the state-space model sys.

Arguments

Argument in:
sys : state space form (frac_ss object)

Argument out:
A: matrix
B: matrix
C: matrix
D: matrix
order: scalar

Example

```
>> fss
a =
        x1            x2
x1   10.01          -0.4
x2    0.25             0
b =
    x1
u1   4
u2   0
c =
    y1 y2
x1   0  2
d =
    y1
u1   0
order =
    0.5000

>> [A,B,C,D,order]=ssdata(fss)
A =
   10.0100   -0.4000
    0.2500         0
B =
     4
     0
C =
     0     2
D =
     0
order =
    0.5000
```

step

Step response of fractional transfer function.

Syntax

```
    Rep=step(sys,Time,method)
```

Arguments

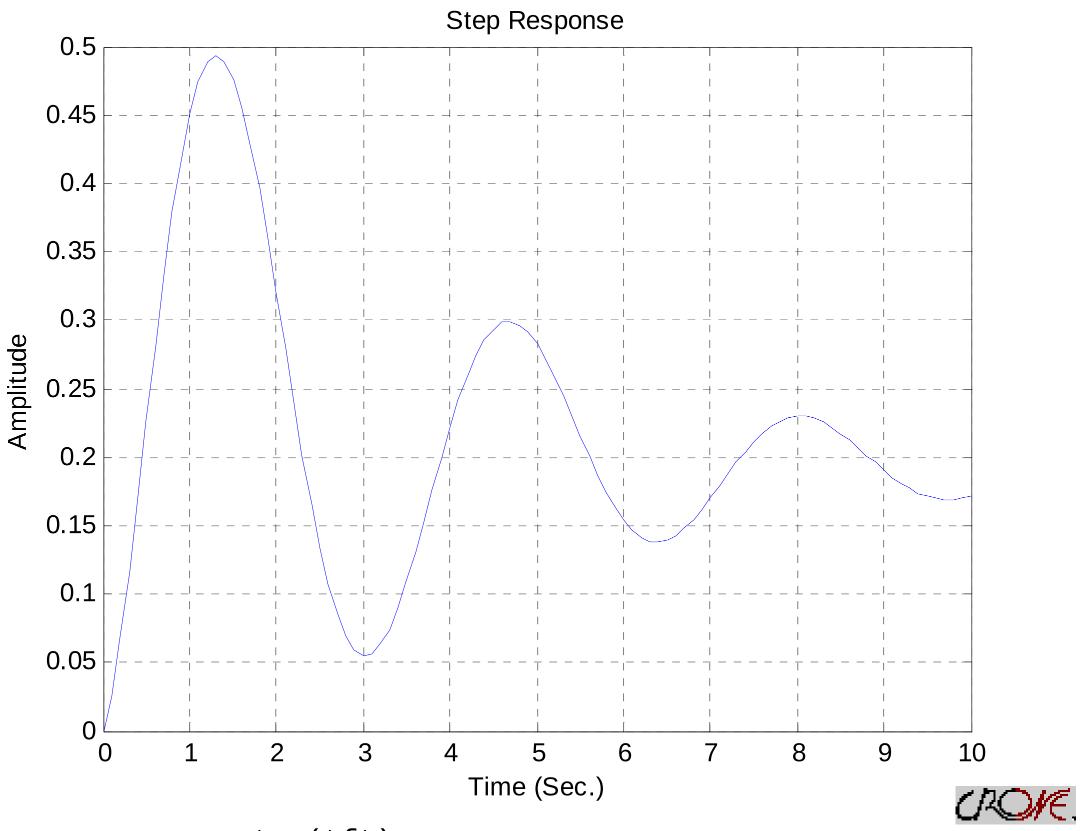
Argument in :
Sys : frac_lti object.
Time : time vector (under the form Ti :Ts :Tf)
method : string (can be 'grun' for Grünwald, 'grunimp' for improved Grünwald, 'approx' for zero and pole approximation)

Argument out :
Rep : step response (Vector)

Example

```
» sys can be
frac_poly_exp
s^2.2 +s^1.5

or frac_poly_imp
( s - 10  )^0.5

or frac_tf
  ( s^0.6 +1  )
------------------
( s^2.2 +s^1.5  )



or frac_zpk
2
----------------------------
 (s^0.5 - 0.01) (s^0.5 - 10)

or frac_ss
a =
        x1           x2
x1   10.01         -0.4
x2    0.25            0

b =
    x1
u1   4
u2   0

c =
    y1 y2
x1   0  2
```

```
d =
    y1
u1   0

order =
    0.5000

>> step(sys)
>> step(sys,1:0.1:10)
>> step(sys,1:0.1:10,'grunimp')
gives
```



```
>> y=step(tft)
>> y=step(tft,1:0.5:10)
>> y=step(tft,1:0.5:10,'grunimp')
gives an answer such as
y =
         0
    0.3389
    0.8214
    1.1554
    1.3270
    1.4370
    1.4997
    1.4825
    1.3965
    1.2845
    1.1734
```

```
                1.0697
                0.9792
                0.9107
                0.8679
                0.8476
                0.8443
                0.8535
                0.8704
```

tf2ss

Convert transfer function parameters to state-space form.

Syntax

```
    ss= tf2ss(sys)
```

Arguments

Argument in:
  sys : fractional transfer function(frac_tf object)

Argument out:
  ss: fractional transfer function (frac_ss object)

Example

```
>> sys
 transfer function :
          ( 2  )
---------------------------
( s - 10.01 s^0.5 + 0.1  )
>> tf2ss(sys)
a =
        x1           x2
x1   10.01         -0.4
x2    0.25            0
b =
    x1
u1   4
u2   0
c =
    y1 y2
x1   0  2
d =
    y1
u1   0
order =
    0.5000
```

tf2zpk

Convert transfer function parameters to zero-pole-gain form.

Syntax

```
    zpk= tf2zpk(sys)
```

Arguments

Argument in:
sys : fractional transfer function(frac_tf object)

Argument out:
zpk: fractional transfer function (frac_zpk object)

Example

```
>> sys
 transfer function :
          ( 2  )
---------------------------
( s - 10.01 s^0.5 + 0.1  )
>> tf2zpk(sys)
Fractionnal continuous-time zero-pole-gain system :
  2
  ---------------------------
  (s^0.5 - 10) (s^0.5 - 0.01)
```

tfdata(frac_tf)

Access transfer function data

Syntax

[num_coef, num_order, den_coef, den_order, num_imp_ord, den_imp_ord] = frac_tfdata(sys)

Description

[num_coef,num_order,den_coef,den_order,num_imp_ord,den_imp_ord]=frac_tfdata(sys) returns the numerator(s) and denominator(s) of the transfer function for SISO models only

Arguments

Argument in:
sys : fractional transfer function(frac_tf object)

Argument out:
num_coef: coefficients for the numerator of the transfer (cell)
num_order: orders for the numerator of the transfer (cell)
den_coef: coefficients for the denominator of the transfer (cell)
den_order: orders for the denominator of the transfer (cell)
num_imp_ord: implicit orders for the numerator of the transfer (cell)
den_imp_ord: implicit orders for the denominator of the transfer (cell)

Example

```
>> sys
 transfer function :
  ( s^0.6 +1  )
------------------
( s^2.2 +s^1.5  )
>> [nc,no,dc,do,nio,dio]=tfdata(sys)
nc =
 [1x2 double]
no =
 [1x2 double]
dc =
 [1x2 double]
do =
 [1x2 double]
nio =
 [1]
dio =
 [1]
```

tolord

This functions sets the tolorence on the orders approximations.

Syntax

```
tol = tolord
```

Arguments

Argument out:
tol: tolerence (scalar)

uncommensurate

Computes de fractional transfer from a LTI system.

Syntax

```
[New_tf]= uncommensurate(tf, step_order)
```

Arguments

Argument in:
tf : fractional transfer function(tf object or polynomial)
Step_order : the step order (scalar)

Argument out:
New_tf: fractional transfer function (frac_tf or frac_poly_exp object)

Example

```
>> sys=tf([1 0 0 0 0 0 1],[1 0 0 0 0 1 0 0 0 0 1])
 Transfer function:
   s^6 + 1
--------------
s^10 + s^5 + 1
 >> uncommensurate(sys,0.1)
 transfer function :
  ( s^0.6 +1  )
-----------------
( s +s^0.5 +1  )
>> t=uncommensurate([1 0 0 0 5 0 3 2 6],0.1)
s^0.8 + 5 s^0.4 + 3 s^0.2 + 2 s^0.1 + 6
```

vertcat

Concatenate fractional objects vertically

Syntax

C = vertcat(A1, A2, ...)

Description

C = vertcat(A1, A2, ...) horizontally concatenates A1, A2, and so on. All fractional object in the argument list must have the same number of rows.

vertcat concatenates N-dimensional fractional objects along the second dimension. The first and remaining dimensions must match.

MATLAB calls C = vertcat(A1, A2,...) for the syntax C = [A1 A2 ...] when any of A1, A2, etc., is a fractional object.

Arguments

Argument in :
 A1: fractional object (M*N1 fractional object)
 A2: fractional object (M*N2 fractional object)

Argument out :
 C: fractional object (M*N fractional object)

Examples

```
>> sys
s^2.2 +s^1.5
>> fpecat=vertcat(sys, sys, sys)
Frac poly exp from input 1 to output:
s^2.2 +s^1.5
Frac poly exp from input 2 to output:
s^2.2 +s^1.5
Frac poly exp from input 3 to output:
s^2.2 +s^1.5
```

zpk2tf

Convert zero-pole-gain form parameters to transfer function.

Syntax

```
    tf= zpk2tf(sys)
```

Arguments

Argument in:
sys : zero pole gain form (frac_zpk object)

Argument out:
tf: fractional transfer function (frac_tf object)

Example

```
>> sys
Fractionnal continuous-time zero-pole-gain system :
  2
  ----------------------------
  (s^0.5 - 0.01) (s^0.5 - 10)
>> zpk2tf(sys)
 transfer function :
          ( 2  )
---------------------------
( s - 10.01 s^0.5 + 0.1  )
```

zpkdata(frac_zpk)

Access zero-pole-gain data

Syntax

[z,p,k,order] = zpkdata(sys)

Description

[z,p,k,order] = zpkdata(sys) returns the zeros z, poles p, gain(s) k, and orders order of the zero- pole-gain model sys.

Arguments

Argument in:
sys : zero, pole, gain form (frac_zpk object)

Argument out:
z: zeros of the zero, pole, gain form (cell)
p: poles of the zero, pole, gain form (cell)
k: gain of the zero, pole, gain form (cell)
order: orders for the zero, pole, gain form (cell)

Example

```
>> fzpk
Fractionnal continuous-time zero-pole-gain system :
  2
  ----------------------------
  (s^0.5 - 0.01) (s^0.5 - 10)
>> [z,p,k,ord]=zpkdata(fzpk)
z =
{[]}
p =
[1x2 double]
k =
[2]
ord =
[0.5000]
```