1 SAC

1.1 Introduction to SAC

The Symbolic Analysis and Control package (`SAC`) is a toolbox that provides several routines for the analysis and control of nonlinear systems with or without time delays including, amongst others, properties like accessibility, observability, equivalence with the triangular form, and linearization by input/output injection, and control problems like the disturbance rejection and operations between noncommutative polynomials and matrix like the Smith form, Euclidean division, wedge product, and so on.

$$\chi^2 = \sum_{i=1}^{N} \left(\frac{y_i - (a + bx_i)}{\sigma_i} \right)^2$$

These tools are based on the so-called differential algebraic approach, which is a very promising method in the study of nonlinear time delay systems.

`SAC` is written in Maxima, an open-source free computer algebra system which can be compiled on many systems, including windows, linux, and MacOS X.

1.2 Definitions for SAC

1.2.1 Operators

`*^` [Operator]

Defines the non-commutative operator `*^`. This allows to multiply polynomials in $\mathcal{K}[\delta]$.

```
(%i1) load("sac.mc")$
(%i2) x(t-1)*_D *^ x(t-2);
(%o2)              x(t - 3) x(t - 1) _D
```

The factors may be matrices.

```
(%i3) matrix([_D^2,1],[_D,_D+x[2](t)])*^matrix([x[1](t)],[u(t)]);
                    [                          2                 ]
                    [           x (t - 2) _D  + u(t)             ]
(%o3)               [             1                              ]
                    [                                            ]
                    [ (x (t - 1) + u(t - 1)) _D + x (t) u(t) ]
                    [   1                            2           ]
```

The right factor may also be a p-form

```
(%i4) x[1](t)*_D*^(x[1](t)*del(x[2](t),u(t-2)));
(%o4)             - x (t - 1) x (t) del(u(t - 3), x (t - 1))
                      1         1                   2
```

Note that δ is written as _D.

`_d (w)` [Function]

Computes the differential form of a function or p-form w. Given $f(z_\tau)$, this routine computes df: $df = \sum_{i=1}^{n}\sum_{j=0}^{s}\frac{\partial f}{\partial z_i(t-j)}dz_i(t-j)$.

The partial derivatives are taken against the variables which explicitely depend on t.

```
(%i1) load("sac.mc")$
(%i2) _d(x[1](t-2)*u(t)+x[1](t));
(%o2)       del(x (t)) + x (t - 2) del(u(t)) + u(t) del(x (t - 2))
                 1        1                                 1
```

If f is a p-form, then it returns its differential, which is a (p+1)-form.

```
(%i3)  dlist:[x[1](t),(x[1](t)+x[2](t))*del(x[1](t)),
              (sin(u(t-1))+x[2](t-1)*u(t)^2)*del(x[2](t),u(t-1))];
(%o3) [x (t), (x (t) + x (t)) del(x (t)),
        1       2       1          1
                        2
       - (x (t - 1) u (t) + sin(u(t - 1))) del(u(t - 1), x (t))]
           2                                                2
(%i4) maplist(_d,dlist);
(%o4) [del(x (t)), - del(x (t), x (t)),
           1             1      2

(- 2 x (t - 1) u(t) del(u(t - 1), x (t), u(t)))
     2                           2

   2
 - u (t) del(x (t - 1), u(t - 1), x (t))]
             2                    2
```

`_D` [Variable]

Default value: `protected var`

_D represents the delay operator δ defined as $\delta f(t) = f(t-1)\delta$ and $\delta w(t) = w(t-1)$, where f is a function and $w(t)$ is a p-form.

_D is a reserved word, so it cannot be bonded to a value. It is used for the [non-commutative] polynomial product [ncprod], page 3.

`ddt (f,S)` [Function]

`ddt (f,S,k)` [Function]

Given a system S:$\dot{x} = f(x_\tau, u_\tau)$ and a function $h(x_\tau, u_\tau^{(i)})$, find the time-derivative of h along the trajectories of S: $d\_dt(h,S) = \sum_{j=0}^{s}\left(\sum_{i=1}^{n}\frac{\partial h}{\partial x(t-j)}f(t-j) + \sum_{k=0}^{r}\frac{\partial h}{\partial u^{(k)}(t-j)}u^{(k+1)}(t-j)\right)$

```
(%i1) load("sac.mc")$
(%i2) S:systdef(matrix([x[2](t-1)],[u[1](t)]))$
(%i3) ddt(x[1](t)^2,S);
(%o3)                          2 x (t - 1) x (t)
                                  2         1
(%i4) ddt(x[1](t)^2,S,2);
                                                      2
(%o4)                  2 u (t - 1) x (t) + 2 x (t - 1)
                          1         1         2
(%i5) ddt(x[1](t)^2,S,3);
                         d
(%o5)          2 x (t) (-- (u (t - 1))) + 6 u (t - 1) x (t - 1)
                   1     dt   1               1         2

(%i6) ddt(u[1](t)*del(x[1](t-1)),S);
                d
(%o6)          (-- (u (t))) del(x (t - 1)) + u (t) del(x (t - 2))
                dt   1           1            1         2
```

See [Lie], page 10,

1.2.2 Functions

`antider` (*dw_1*, *dw_2*, ...) [Function]

This function returns the integral form of its argument, which can be a closed 1-form or a list of closed 1-forms.

```
(%i1) load("sac.mc")$

(%i1) load("sac.mc")$
(%i2) L:[del(u(t)),x[1](t-1)*del(x[1](t-1))]$
(%i3) antider(L);
                                        2
                                       x (t - 1)
                                        1
(%o3)                          [u(t), ---------]
                                           2
```

If one of the arguments is not a closed 1-form (even if it is integrable), then it throws an error.

```
(%i4) antider(x[1](t)*del(x[2](t-1)));

argument is not a [list of ] closed 1-form[s]
```

The integration is done using the routine `potential`, from the "vect" package. See [differential], page 4,

`coefpow` (*pol*) [Function]

Given a polynomial *pol*, $p[\delta] = \sum_i p_i \delta^i$, where p_i are scalar or matrix coefficients, it returns a list of the nonzero coefficients $p_i \in \mathcal{K}[\delta]$, and another list with the corre-

sponding exponents i, in ascending order, that is $[c, e]$ where $c = [p_i \mid p_i \neq 0]$, and $e = [i \mid p_i \neq 0]$, in ascending order.

```
(%i2) coefpow(_D^3+2);
(%o2)                          [[2, 1], [0, 3]]
(%i3) coefpow(matrix([_D^3+3],[_D]));
                          [ 3 ]  [ 0 ]  [ 1 ]
(%o159)                [[[   ], [   ], [   ]], [0, 1, 3]]
                          [ 0 ]  [ 1 ]  [ 0 ]
```

`dotfact (w)` [Function]
`dotfact (w, flag)` [Function]

Given a p-form $\omega = \sum_{i=1}^{s} a_i\, dz_i = a dz$ returns the column vectors $a = [a_1, \ldots, a_s]^T$ and $dz = [d(z_1), \ldots, d(z_s)]^T$.

If the argument is a column vector of p-forms v, it returns the matrices M and dz=[d(z_1), ...,d(z_s)] such that $\omega = M\, dz$.

If a second argument *flag* is given and is equal to 0, then the factorization does not include the _D operator.

```
(%i1) load("sac.mc")$

(%i2) w:_d(x[1](t-3)*u(t-2));
(%o2)          x (t - 3) del(u(t - 2)) + u(t - 2) del(x (t - 3))
                1                                          1
(%i3) dotfact(w);
                         [            3  ]
                         [ u(t - 2) _D   ]  [ del(x (t)) ]
(%o3)                   [[               ], [      1     ]]
                         [             2 ]  [            ]
                         [ x (t - 3) _D  ]  [ del(u(t))  ]
                         [  1            ]
(%i4) dotfact(w,0);
                         [ u(t - 2)  ]  [ del(x (t - 3)) ]
(%o4)                   [[          ], [      1         ]]
                         [ x (t - 3) ]  [                ]
                         [  1        ]  [ del(u(t - 2))  ]
```

See [differential], page 4,

`euclid (a, b)` [Function]

Given two poynomials a, b$\in \mathcal{K}[\delta)$, it performs the Euclid's division to find q, r$\in \mathcal{K}[\delta)$ such that a=qb+r, where the polynomial degree of pol.d(r) is strictly less than pol.d(b). This function returns a polynomial matrix M=[q,r] such that a = q b + r, and deg(r)<deg(b).

```
(%i1) load("sac.mc")$
(%i2) a:x(t)*_D^2+2$
(%i3) b:u(t)*_D-1$
(%i4) d:euclid(a,b);
            [ x(t) _D        x(t)       2 u(t) u(t - 1) + x(t) ]
(%o4)/R/    [ -------- + -------------  ---------------------- ]
            [ u(t - 1)   u(t - 1) u(t)      u(t) u(t - 1)      ]
(%i5) fullratsimp(d[1][1]*^b+d[1][2]);
                                      2
(%o5)                          x(t) _D  + 2
```

euclid() : ncalg.mc

`findel` (*matrix* `M`, *expr* `e`) [Function]
`findel` (*matrix* `M`, *expr* `e`, *int* `idr`) [Function]
`findel` (*matrix* `M`, *expr* `e`, *int* `idr`, *int* `idc`) [Function]

Given a matrix M, an element e, a row number *idr*, and a column number *idc*, returns a list of all the pairs [i,j] such that M[i,j]=e$, with i$\geq$ *idr*, j$\geq$ *idc*. If only one index is given, it will be assigned to *idr*, and *idc* will be set to 1. If no index is given, then *idr*=*idc*=1.

```
(%i1) load("sac.mc")$
(%i2) M:genmatrix(lambda([i,j],(2*i-j)),4,10);
              [ 1  0  - 1  - 2  - 3  - 4  - 5  - 6  - 7  - 8 ]
              [                                               ]
              [ 3  2   1    0   - 1  - 2  - 3  - 4  - 5  - 6 ]
(%o2)         [                                               ]
              [ 5  4   3    2    1    0   - 1  - 2  - 3  - 4 ]
              [                                               ]
              [ 7  6   5    4    3    2    1    0   - 1  - 2 ]
(%i3) findel(M,-3);
(%o3)                    [[1, 5], [2, 7], [3, 9]]
(%i4) findel(M,1,2);
(%o4)                    [[2, 3], [3, 5], [4, 7]]
(%i5) findel(M,-3,2,6);
(%o5)                        [[2, 7], [3, 9]]
```

`findmaxidx` (`f`, `s`) [Function]

Given an expression f, and a symbol s, this function returns max k such that $\partial f/\partial s[k](t-\ell) \neq 0$ for a delay $\ell \in \Re_+$.

```
(%i1) load("sac.mc")$
(%i2) w:matrix([sin(u[3](t-2)+u(t))+1],[x[4](t)])$
(%i3) findmaxidx(w,u);
(%o3)  3
(%i4) findmaxidx(w,x);
(%o4)  4
(%i5) findmaxidx(w,z);
(%o5)  minf
```

See [systdef], page 14,

`gradfnc` (*f*, *v*) [Function]

Given a list of variables [v_1, ..., v_s], it will return the partial derivative of function f(.) with respect to them:

$$gradfnc(f, v) = \left[\frac{\partial f}{\partial v_1}, \ldots, \frac{\partial f}{\partial v_s}\right]$$

```
(%i1) load("sac.mc")$
(%i2) gradfnc(((x[2](t-1))^2),[x[1](t-1),x[2](t-1)]);
(%o2)                [0, 2 x (t - 1)]
                      2
(%i3) gradfnc(matrix([x[2](t-1)^2],[x[1](t-1)*x[2](t)]),[x[1](t-1),x[2](t-1)]);
                      [   0    2 x (t - 1) ]
                      [           2        ]
(%o3)                       [                    ]
                      [ x (t)       0      ]
                      [  2                 ]
```

`nombrefn` (*pol*) [Function]

Computes the submodules \f$H_k\f$. After calling this routine, a specific \f$H_k\f$ can be recovered using assoc(k,S@hk).

```
(%i1) load("sac.mc")$
(%i2) f:matrix([s*(x[2](t)-x[1](t))+u(t)],[x[1](t)*(b-x[3](t))-x[2](t)],[x[1](t)*
                         [ u(t) + s (x (t) - x (t))  ]
                         [            2       1      ]
                         [                           ]
(%o2)                    [ x (t) (b - x (t)) - x (t) ]
                         [  1          3        2    ]
                         [                           ]
                         [   x (t) x (t) - a x (t)   ]
                        [    1     2         3      ]
(%i3) lorenz:systdef(f)$
(%i4) hk(lorenz);
(%o4) [[2, [del(x (t)), del(x (t))]], [3,
                2          3
              [x (t) del(x (t)) - b del(x (t)) + x (t) del(x (t))]], [inf, 0]]
                3         3              3        2         2
(%i5) lorenz@hk;
(%o5) [[2, [del(x (t)), del(x (t))]], [3,
                2          3
              [x (t) del(x (t)) - b del(x (t)) + x (t) del(x (t))]], [inf, 0]]
                3         3              3        2         2
(%i6) assoc(2,lorenz@hk);
(%o6)         [del(x (t)), del(x (t))]
```

`isaccessible` (*system S*) [Function]

Tests if the system S satisfies the strong accessibility condition: $\mathcal{H}_\infty = 0$.

```
(%i1) load("sac.mc")$
(%i2) f:matrix([x[2](t-2)*u[1](t)],[u[2](t-3)])$
(%i3) S:systdef(f)$
(%i4) isaccessible(S);
(%o4)                                  true
```

See [isclosed], page 9,

`isclosed` (*w_1*, *w_2*, ...) [Function]

This routine checks if a 1-form or list of 1-forms are closed, that is, every element of the list is a 1-form *w_i* satisfying d(*w_i*)=0 for all i=1,...

Any argument which is not a 1-form, will return false.

```
(%i1) load("sac.mc")$
(%i2) isclosed(u(t-1)*del(u(t-1)));
(%o2)                                  true
(%i3) isclosed(u(t)*del(u(t-1)));
(%o3)                                  false
```

If the argument is a list, it returns true only if all its elements are closed 1-forms.

```
(%i5) isclosed([ u(t-1)*del(u(t-1)), u(t)*del(u(t-1)) ]);
(%o5)                                  false
```

`isintegrable` (*w_1*, *w_2*, ...) [Function]

This routine checks if a 1-form or list of 1-forms is integrable using the Frobenious theorem. So far, it is not valid for time-delay systems.

```
(%i1) load("sac.mc")$
(%i2) isintegrable(x[2](t)*del(x[2](t))+del(x[3](t)));
(%o2)                                  true
(%i3) isintegrable(x[1](t)*del(x[2](t))+del(x[3](t)));
(%o3)                                  false
(%i4) isintegrable(x[1](t)*del(x[2](t))+del(x[3](t)),del(x[1](t)));
(%o4)                                  true
```

`isobservable` (*system S*) [Function]

Given a system S with output y, it checks the generic observability condition given by:

$rank_{\mathcal{K}[\delta)} \frac{\partial[y\ \dot{y}\cdots y^{(n-1)}]}{\partial x} = n$

```
(%i1) load("sac.mc")$
(%i2) f:matrix([x[2](t-2)*u[1](t)],[u[2](t-3)])$
(%i3) h:x[1](t-1)$
(%i4) S:systdef([f,h],[x,u,y])$
(%i5) isobservable(S);
(%o5)                                  true
```

Warning not tested for time-delay systems

`leftkernel` (*matrix M*) [Function]

Returns a basis for the left kernel of a matrix with entries in $\mathcal{K}[\delta)$.

```
(%i1) load("sac.mc")$
(%i2) M:matrix([_D],[u(t)],[1+_D]);
                                  [   _D   ]
                                  [        ]
(%o2)                             [  u(t)  ]
                                  [        ]
                                  [ _D + 1 ]
(%i3) leftkernel(M);
                        [              _D           ]
                        [ 1       - --------       0 ]
                        [           u(t - 1)         ]
(%o3)                   [                            ]
                        [      u(t - 1) + u(t) _D    ]
                        [ 0  - ------------------  1 ]
                        [        u(t - 1) u(t)       ]
```

`Lie` (*h*, *S*) [Function]
`Lie` (*f*, *S*, *k*) [Function]

Computes the k-th Lie derivative of h following the trajectory of S of the polynomial $h \in \mathcal{K}[\delta]$

```
(%i1) load("sac.mc")$
(%i2) f:matrix([x[2](t-2)*u(t)],[x[1](t)]);
                               [ x (t - 2) u(t) ]
                               [  2             ]
(%o2)                          [                ]
                               [     x (t)      ]
                               [      1         ]
(%i3) s:systdef(f,[x,u])$
(%i4) lie(x[2](t-3)*_D,s,2);
                             x (t - 5) u(t - 3) _D
(%o5)                         2
```

`lorebez` (`a`, *b*) [Function]

Let $a, b \in \mathcal{K}[\delta]$. We call $\alpha,\ \beta$ Ore polynomials if they satisfy the left-Ore condition: $\alpha\, a + \beta\, b = 0$, and we call them Bezout polynomials if they satisfy $\alpha\, a + \beta\, b = gcld(a, b)$ where glcd(a,b) stands for greatest left common divisor of (a,b).

```
(%i1) load("sac.mc")$
(%i2) a:_D^2+1$
(%i3) b:x(t)$
(%i4) lorebez(a,b);
                    [             1           ]
                    [ 0          ----         ]
                    [            x(t)         ]
(%o4)/R/            [                         ]
                    [                    2    ]
                    [      x(t - 2) + _D  x(t) ]
                    [ 1  - ------------------- ]
                    [         x(t - 2) x(t)     ]
(%i5) lorebez(a,b)*^matrix([a],[b]);
                                [ 1 ]
(%o5)                           [   ]
                                [ 0 ]
```

`maxd` (*f*) [Function]

Given a function, matrix, or p-form, it finds the maximum delay found in any time-dependant variable.

```
(%i1) load("sac.mc")$
(%i2) maxd(x[3](t-1)*u(t-4));
(%o2)                         4
(%i3) maxd(
```

`ncgrad` (*f*, *v*) [Function]

Computes the gradient of a function $\sum_{i=0}^{\tau_M} \partial f/\partial v(t-i)\delta^i$.

```
(%i1) load("sac.mc")$
(%i2) ncgrad(x[2](t-2)*x[1](t)+x[1](t-2),[x[1](t),x[2](t)]);
                              2                      2
(%o2)                    [_D  + x (t - 2), x (t) _D ]
                                2          1
```

`ncinverse` (*matrix M*) [Function]

This routine computes the inverse of a matrix with entries in $\mathcal{K}[\delta)$, if it exists. Otherwise, signals an error.

```
(%i1) load("sac.mc")$
(%i2) M:matrix([1+_D,-_D],[_D,1-_D])*^matrix([x[2](t),u(t)],[1,-u(t-2)]);
      [ (x (t - 1) - 1) _D + x (t)    (u(t - 3) + u(t - 1)) _D + u(t)   ]
      [   2                    2                                        ]
(%o2) [                                                                ]
      [   (x (t - 1) - 1) _D + 1    (u(t - 3) + u(t - 1)) _D - u(t - 2) ]
      [     2                                                          ]
(%i3) ncinverse(M);
      [   (u(t) + u(t - 2)) _D - u(t - 2)  u(t) + (u(t) + u(t - 2)) _D ]
      [ - -----------------------------    -------------------------- ]
      [        u(t) + u(t - 2) x (t)          u(t) + u(t - 2) x (t)    ]
      [                          2                                2     ]
(%o3) [                                                                ]
      [        1 + (x (t) - 1) _D             x (t) + (x (t) - 1) _D   ]
      [              2                         2        2              ]
      [       --------------------          - ----------------------   ]
      [       u(t) + u(t - 2) x (t)          u(t) + u(t - 2) x (t)    ]
      [                          2                                2     ]
(%i4) ncinverse(M)*^M;
                                [ 1  0 ]
                                  [    ]
(%o4)                           [ 0  1 ]
(%i5)
```

`ncrowrank` (*matrix M*) [Function]

Returns the row rank of M over $\mathcal{K}[\delta]$.

```
(%i1) load("sac.mc")$
(%i2) ncrowrank(matrix([_D^2,1],[_D,1+_D],[_D,1-_D]));
(%o2)                                2
(%i3) M:matrix([u(t),u(t-1),_D],[u(t-1)*_D,u(t-2)*_D,_D^2]);
                     [    u(t)       u(t - 1)    _D  ]
(%o3)                [                               ]
                     [                             2 ]
                     [ u(t - 1) _D  u(t - 2) _D  _D  ]
(%i4) ncrowrank(M);
(%o4)                                1
```

`nctriangularize` (*matrix M*) [Function]

Returns a structure with 3 elements: P, S, and Q, such that for the given matrix M$\in \mathcal{K}[\delta)^{n\times m}$, P M Q = S. P and Q are unimodular matrices, while S is an upper-triangular matrix, whose elements of the main diagonal are normalized.

```
(%i1) load("sac.mc")$
```

/* buggy!! needs work! */

`p_degree` (*pol*) [Function]

Given a p-form $\omega \in \mathcal{E}^p$, it returns the integer p.

```
* (%i1) load("sac.mc")$
* (%i2) p_degree(del(x[1](t),x[3](t-1)));
* (%o2)            2
```

`protect` (*s*) [Function]

Some symbols are reserved for the use of the software. Assigning them would lead to weird and hard-to-track bugs. This command avoids this problem by reserving the symbol *s*. So far we have reserved *t*, *del*, *true*, and *false*.

```
(%i1) load("sac.mc")$
(%i2) protect(t);
(%o2) neverset
(%i3) t:1;
assignment: cannot assign to t
 -- an error. To debug this try: debugmode(true);
(%i4) unprotect(t)$
(%i5) t:1;
(%o5) 1
```

See [unprotect], page 17.

`psqswap` (*psqstruct* `s`, *list* [`r1,c1`], *list* [`r2,c2`]) [Function]

Given a PSQ structure, and 2 lists of indexes [*r1,c1*] and [*r2,c2*], swap rows and columns of the elements P, S, Q to have P', S', and Q' such that S'= P' S Q', where S' is obtained by swapping rows *r1* and *r2*, and columns *c1* and *c2*.

```
(%i1) load("sac.mc")$
(%i2) mypsq:new(PSQ (ident(2),matrix([a,b,c],[d,e,f]),ident(3)));
                                                    [ 1  0  0 ]
                        [ 1  0 ]      [ a  b  c ]   [         ]
(%o2)         PSQ(P = [      ], S = [         ], Q = [ 0  1  0 ])
                        [ 0  1 ]      [ d  e  f ]   [         ]
                                                    [ 0  0  1 ]
(%i3) psqswap(mypsq,[1,1],[2,3]);
                                                    [ 0  0  1 ]
                        [ 0  1 ]      [ f  e  d ]   [         ]
(%o3)         PSQ(P = [      ], S = [         ], Q = [ 0  1  0 ])
                        [ 1  0 ]      [ c  b  a ]   [         ]
                                                    [ 1  0  0 ]
```

`relshift` (*f*) [Function]

The relative shift of a function \f$f(z_tau)\f$ is defined as the maximal forward time shift such that the resulting function is still causal. Mathematically, $rel\_shift(f(z_\tau)) = f(t) = max\{k \in Z^+ \mid (f(t+k) \in span_{\mathcal{K}[\delta)}\{dz\}\}$

```
(%i1) load("sac.mc")$
(%i2) relshift(x[3](t-1)*u(t-4));
(%o2)                              1
```

`showalltvars` (*expr*) [Function]

This function returns a list of all time-dependent variables appearing in `expr`.

```
(%i1) load("sac.mc")$
(%i2) w:q*u[1](t-1)*sin(x[2](t-2))*diff(u(t-1),t)*del(u[1](t));
                                              d
(%o2)        q u (t - 1) sin(x (t - 2)) (-- (u(t - 1))) del(u (t))
               1             2           dt                   1
(%i3) showratvars(w);
                                               d
(%o3)      [q, u (t - 1), sin(x (t - 2)), -- (u(t - 1)), del(u (t))]
               1              2           dt                   1
(%i4) showtvars(w);
                                         d
(%o4)          [x (t - 2), u (t - 1), -- (u(t - 1)), del(u (t))]
                 2           1         dt                   1
(%i5) showalltvars(w);
(%o5)               [x (t - 2), u (t - 1), u(t - 1), u (t)]
                      2           1                    1
```

See [showtvars], page 14,

`showtvars (`*expr*`)` [Function]

This function is like [showalltvars], page 13, but keeping `del()` or `diff()` operators.

```
(%i1) load("sac.mc")$
(%i2) w:q*u[1](t-1)*sin(x[2](t-2))*diff(u(t-1),t)*del(u[1](t))$
(%i3) showtvars(w);
                                         d
(%o3)          [x (t - 2), u (t - 1), -- (u(t - 1)), del(u (t))]
                 2           1         dt
```

`systdef (`*f*`)` [Function]
`systdef ([`*f*`,` *h*`])` [Function]
`systdef ([`*f*`,` *h*`], [`*n*`,`*v*`,`*z*`])` [Function]

$$\dot{x} = f(x_\tau,\ u_\tau)$$
$$y = h(x_\tau)$$

Given a system of the form where creates a structure to

$$x_\tau = x(t), x(t-1), \ldots, x(t-s)$$
$$u_\tau = u(t), u(t-1), \ldots, u(t-s)$$

store it.

There are three main forms of defining a system:

A) S:systdef(f); (no output, x=state, u=control)

```
(%i1) load("sac.mc")$
(%i2) f:matrix([s*(x[2](t)-x[1](t))+u(t)],[x[1](t)*(b-x[3](t))-x[2](t)],[x[1](t)*
(%i3) lorenz:systdef(f);
                              [     s (x (t) - x (t))     ]
                              [         2       1         ]
                              [                           ]
(%o3) sys(affine = true, f = [ x (t) (b - x (t)) - x (t) ],
                              [  1          3        2    ]
                              [                           ]
                              [   x (t) x (t) - a x (t)   ]
                              [    1     2         3      ]
     [    - s       s       0    ]
     [                           ]      [ 1 ]
     [ b - x (t)   - 1   - x (t) ]      [   ]
dF = [      3               1    ], g = [ 0 ],
     [                           ]      [   ]
     [   x (t)    x (t)    - a   ]      [ 0 ]
     [    2        1             ]
     [ s (x (t) - x (t)) + u (t) ]
     [     2       1        1    ]
     [                           ]
fg = [ x (t) (b - x (t)) - x (t) ], h = 0, n = 3, m = 1, p = 0,
     [  1          3        2    ]
     [                           ]
     [   x (t) x (t) - a x (t)   ]
     [    1     2         3      ]
statevar = [x (t), x (t), x (t)], controlvar = [u (t)], outputvar = y,
             1      2      3                     1
taumax = 0, hk)
```

B) S:systdef([f,h]); (output y=h(x))

```
(%i1) load("sac.mc")$

(%i2) f:matrix([x[2](t-2)*u[1](t)],[u[2](t-3)]);
                                [ x (t - 2) u (t) ]
                                [  2         1    ]
(%o2)                           [                 ]
                                [    u (t - 3)    ]
                                [     2           ]
(%i3) h:x[1](t-1)$

(%i4) S:systdef([f,h],[x,u,y]);
                           [            2 ]      [ x (t - 2)   0  ]
                           [ 0  u (t) _D  ]      [  2             ]
(%o4) sys(affine, f, dF = [     1         ], g = [                ],
                           [               ]      [              3 ]
                           [ 0      0      ]      [     0      _D  ]
     [ x (t - 2) u (t) ]
     [  2         1    ]
fg = [                 ], h = [ x (t - 1) ], n = 2, m = 2, p = 1,
     [    u (t - 3)    ]      [  1         ]
     [     2           ]
statevar = [x (t), x (t)], controlvar = [u (t), u (t)], outputvar = [y (t)],
            1      2                     1      2                    1
taumax = 3)
```

C) S:systdef([f,h],[n,v,z]); (n=state, v=control, z=output variables)

```
(%i1) load("sac.mc")$

(%i2) f:matrix([n[2](t)],[v[1](t)])$

(%i3) h:matrix([n[1](t)],[n[2](t-1)]);
                                  [   n (t)   ]
                                  [    1      ]
(%o3)                             [           ]
                                  [ n (t - 1) ]
                                  [  2        ]
(%i4) S:systdef([f,h],[n,v,z]);
                              [ n (t) ]       [ 0  1 ]      [ 0 ]
(%o4) sys(affine = true, f = [  2    ], dF = [      ], g = [   ],
                              [       ]       [ 0  0 ]      [ 1 ]
                              [   0   ]
     [ n (t) ]      [   n (t)   ]
     [  2    ]      [    1      ]
fg = [       ], h = [           ], n = 2, m = 1, p = 2,
     [ v (t) ]      [ n (t - 1) ]
     [  1    ]      [  2        ]
statevar = [n (t), n (t)], controlvar = [v (t)], outputvar = [z (t), z (t)],
             1      2                    1                    1      2
taumax = 0, hk)
```

`tshift` [Function]

`tshift`(*f*)

`tshift`(*f*,s)

Shifts in time its first argument, which can be any valid function, polynomial, matrix, p-form, or a list of these elements. If a second argument s is given, it shifts the first argument by s units of time.

```
(%i1) load("sac.mc")$
(%i2) tshift(x(t-2),4);
(%o2)    x(t - 6)
(%i3) tshift([matrix([x(t-1)],[u(t)]),x[3](t-3)]);
                             [ x(t - 2) ]
(%o6)                       [[          ], x (t - 4)]
                             [ u(t - 1) ]   3
```

`unprotect` (*s*) [Function]

Removes the protection of the protected symbol *s*. See [protect], page 13, for an example.

`wedge` (*p1-form1*, *p2-form*,...) [Function]

Computes the wedge product of its arguments $\Lambda : \mathcal{E}^{p_1} \times \cdots \times \mathcal{E}^{p_s} \to \mathcal{E}^{\Sigma p_i}$, which can be functions or p-forms.

```
(%i1) load("sac.mc")$
(%i2) wedge(del(x[1](t-1)), del(x[1](t-2),x[2](t)) );
(%o2)                - del(x (t - 2), x (t - 1), x (t))
                             1          1          2
```

Note that d(x)∧d(y) is written as d(x,y).

Appendix A Function and variable index