1 Vectors Library

The NASA PVS library contains three distinct vectors libraries

1. 2-dimensional vectors
2. 3-dimensional vectors
3. N-dimensional vectors

One might wonder why there should be 2D and 3D versions, when an N-dimensional version is available. The answer is that there are some notational conveniences for doing this. For example, in the 2D version we represent a vector as

```
Vector: TYPE = [#  x, y: real  #]
```

whereas in the N-dimensional library a vector is

```
Index     : TYPE = below(n)
Vector    : TYPE = [Index -> real]
```

where n is a formal parameter (posnat) to the theory. Thus, in the two dimensional case, the x-component of a vector v is v‘x whereas in the N-dimensional library it is v(0). Also certain operations are greatly simplified in the 2D case. The dot product is

```
*(u,v): real  =  u‘x * v‘x + u‘y * v‘y;          % dot product
```

in the 2-dimensional case, whereas in the N-dimensional case it is

```
*(u,v): real = sigma(0,n-1,LAMBDA i:u(i)*v(i));     % Dot Product
```

where sigma is a summation operator imported from the reals library.

In this appendix we will present the 2-dimensional version because that is what is used in the SATS work. However, the differences in the libraries are kept to a minimum. All operators, definitions, and lemmas are given identical names to simplify the use of these libraries.

1.1 2D Vectors

Two names are available for a vector type are provided in the theory vectors2D.

```
Vector    : TYPE = [#  x, y: real  #]
Vect2     : TYPE = Vector
```

The vector operators are defined as follows:

```
a      : VAR real
u,v,w : VAR Vector

-(v)  : Vector = (-v`x, -v`y);

+(u,v): Vector = (u`x + v`x, u`y + v`y);

-(u,v): Vector = (u`x - v`x, u`y - v`y);

*(u,v): real  =  u`x * v`x + u`y * v`y;          % dot product

*(a,v): Vector = (a * v`x, a * v`y);
```

A conversion is provided so that one can create 2D vectors as follows

```
(xv,yv)
```

rather than having to write

```
(# x := xv, y := yv #)
```

There are several functions and predicates provided such as

```
sqv(v): nnreal = v*v
norm(v): nnreal = sqrt(sqv(v))

zero_vector?(v) : MACRO bool = (norm(v) = 0  AND
                                v`x = 0 AND v`y = 0)

nz_vector?(v)   : MACRO bool = (norm(v) /= 0 AND
                                (v`x /= 0 OR v`y /= 0))

normalized?(v)  : MACRO bool = (norm(v) = 1)

zero    : Zero_vector =  (0,0) ;

^(nzv)         : Normalized = (1/norm(nzv))*nzv

parallel?(nzu,nzv): bool = ^(nzu)*^(nzv) = 1 OR
                           ^(nzu)*^(nzv) = -1

orthogonal?(u,v): bool =  u * v = 0 ;
```

There are several dozen lemmas available for manipulating vectors such as

```
add_assoc           : LEMMA u+(v+w) = (u+v)+w
add_move_right      : LEMMA u + w = v IFF u = v - w
add_cancel_left     : LEMMA u + v = u + w IMPLIES v = w
neg_distr_sub       : LEMMA -(v - u) = u - v
dot_eq_args_ge      : LEMMA u*u >= 0
dot_distr_add_right : LEMMA (v+w)*u = v*u + w*u
dot_scal_left       : LEMMA (a*u)*v = a*(u*v)
dot_scal_canon      : LEMMA (a*u)*(b*v) = (a*b)*(u*v)
sqv_scal            : LEMMA sqv(a*v) = sq(a)*sqv(v)
sqrt_sqv_norm       : LEMMA sqrt(sqv(v)) = norm(v)
norm_eq_0           : LEMMA norm(v) = 0 IFF v = zero
cauchy_schwartz     : LEMMA sq(u*v) <= sqv(u)*sqv(v)
```

1.2 Positions in 2D space

The theory positions2D enhances the vector space with constructs for specifying distances. One frequently wants to use a vector to designate a location in 2D space. To make this more explicit, the following type definition was added

```
Pos2D: TYPE = Vect2
```

though it is really just a synonym. Next it is useful to have a metric or distance function:

```
sq_dist(p1,p2: Pos2D): nnreal = sq(p1`x - p2`x) + sq(p1`y - p2`y)

dist(p1,p2: Pos2D)   : nnreal = sqrt(sq_dist(p1,p2))
```

Many lemmas are available, including

```
dist_refl    : LEMMA dist(p,p) = 0
dist_sym     : LEMMA dist(p1,p2) = dist(p2,p1)
dist_eq_0    : LEMMA dist(p1,p2) = 0 IFF p1 = p2
dist_norm    : LEMMA dist(u,v) = norm(u-v)
sq_dist_le   : LEMMA sq_dist(v1,v2) <= sq_dist(p1,p2) IMPLIES
                        dist(v1,v2) <= dist(p1,p2)
dist_ge_x    : LEMMA dist(p1,p2) >= abs(p1`x - p2`x)
dist_ge_y    : LEMMA dist(p1,p2) >= abs(p1`y - p2`y)
dist_triangle: LEMMA sq(dist(p2,p0)) = sq(dist(p1,p0)) + sq(dist(p1,p2))
                                          - 2*(p1-p0)*(p1-p2)
```

The following predicates are available:

```
on_circle?(p,r): bool = dist(p,zero) = r

on_line?(p1,p2,p): bool =
```

```
              EXISTS (x : real) : p = p1 + x * (p2 - p1)

on_segment?(p1,p2,p): bool =
              EXISTS (x : { y: nnreal | y <= 1}) : p = p1 + x * (p2 - p1)
```

1.3 2D Lines

The theory lines2D provides convenient formalizations for lines in 2-dimensional space. The traditional way to defines a line L is by specifying two distinct points, $\vec{p_0}$ and $\vec{p_1}$, on it. A line L can also be defined by a point and a direction. Let $\vec{p_0}$ be a point on the line L and let $\vec{dv}$ be a nonzero vector specifying the direction of the line. This is equivalent to the two point definition, since we could just put $\vec{dv} = (\vec{p_1} - \vec{p_0})$. We can also add dynamics to our line. If we assume a particle is moving in a line with a constant velocity, then we can define this linear motion using the location of the point at time zero, a velocity vector and a time parameter t:

$$\vec{p_0} + t * \vec{\mathsf{vel}}$$

which provides the location of the particle at time t.

In the library, lines are defined as a tuple:

```
                                  %     Basic        |   Dynamic
                                  %------------------|-----------------
Line : TYPE = [# p: Vect2,        % point on the line| position at time 0
                 v: Nz_vect2 #]   % direction vector | velocity vector

Line2D: TYPE = Line
```

This enables one to represent a line using a point and a direction vector

```
  p(L) + v(L)                or              L`p + L`v
```

or using a point and a velocity vector

```
  p(L) + t v(L)              or              L`p + t * L`v
```

The following alternate field names are provided

```
p0 (L: Line): MACRO Vect2 = p(L)  % alternate field names
vel(L: Line): MACRO Vect2 = v(L)
```

For example

```
 L`p0 + t * L`vel
```

This can be appreviated using the following macro:

```
loc(L: Line)(tt: real): MACRO Vect2 = p(L) + tt*v(L)
```

Two functions are provided to calculate the velocity vector for different situations:

vel_from_tm: generates velocity vector from two points and transport time
vel_from_spd: generates velocity vector from two points and speed

These are defined as follows

```
vel_from_tm(p1,p2,t): { v | p2 = p1 + t*v } = 1/t*(p2 - p1)

vel_from_spd(p1,p2,s):  Vect2 = IF p1 = p2 then zero
                                ELSE s/dist(p1,p2)*(p2-p1)
                                ENDIF
```

Other useful lemmas include

```
vel_from_tm_rew     : LEMMA vel_from_tm(p1,p2,t) = 1/t*(p2 - p1)
vel_from_tm_eq_args : LEMMA vel_from_tm(p,p,t) = zero
vel_from_spd_lem    : LEMMA p1 /= p2 IMPLIES
           vel_from_spd(p1,p2,ps) = vel_from_tm(p1,p2,dist(p1,p2)/ps)
vel_from_spd_norm   : LEMMA p1 /= p2 IMPLIES
           vel_from_spd(p1,p2,s) = s*normalize(p2-p1)
```

Some predicates on lines are also provided:

```
L,L1,L2: VAR Line

on_line?(p,L): bool = EXISTS (x : real) : p = p(L) + x * v(L)

on_segment?(p,L): bool =
             EXISTS (x : { y: nnreal | y <= 1}) : p = p(L) + x * v(L)

orthogonal?(L1,L2): bool = ^(v(L1))*^(v(L2)) = 0

parallel?(L1,L2)  : bool = ^(v(L1))*^(v(L2)) = 1 OR ^(v(L1))*^(v(L2)) = -1
```

1.4 Intersecting Lines

The theory intersections2D provides some efficient methods for determining whether two lines intersect or not and the point of intersection if they do so. The theory is built around a function named cross:

$$\mathsf{cross}(p,q) = p_x * q_y - q_x * p_y$$

The following simple property hold for cross:

$$\mathsf{cross}(p,q) = -\mathsf{cross}(q,p)$$

There are three cases for two lines $L0$ and $L1$:

intersecting:	$\mathsf{cross}(L0_v, L1_v) \neq 0$	
parallel:	$\mathsf{cross}(L0_v, L1_v) = 0$ AND	$\mathsf{cross}(\Delta, L0_v) \neq 0$
same line:	$\mathsf{cross}(L0_v, L1_v) = 0$ AND	$\mathsf{cross}(\Delta, L0_v) = 0$

where $\Delta = L1_p - L0_p$. Correspondingly, the library provides the following predicates:

```
intersect?(L0,L1): bool = cross(L0`v,L1`v) /= 0

same_line?(L0,L1): bool = LET DELTA = L1`p - L0`p IN
                     cross(L0`v,L1`v) = 0 AND cross(DELTA,L0`v) = 0
```

Given two lines that intersect the function intersect_pt returns the intersection point:

```
intersect_pt(L0:Line2D,L1: Line2D | cross(L0`v,L1`v) /= 0): Pos2D =
                            LET DELTA = L1`p - L0`p,
                               ss = cross(DELTA,L1`v)/cross(L0`v,L1`v) IN
                               L0`p + ss*L0`v
```

Several key lemmas are provided:

```
intersection_lem     : LEMMA cross(L0`v,L1`v) /= 0 IMPLIES
                             LET DELTA = L1`p - L0`p,
                                ss = cross(DELTA,L1`v)/cross(L0`v,L1`v),
                                tt = cross(DELTA,L0`v)/cross(L0`v,L1`v)
                             IN
                          L0`p + ss*L0`v =  L1`p + tt*L1`v

pt_intersect         : LEMMA on_line?(p,L0) AND on_line?(p,L1) AND
                             NOT same_line?(L0,L1) IMPLIES
                                 intersect?(L0,L1)

intersect_pt_unique  : LEMMA intersect?(L0,L1) IMPLIES
                             pnot /= intersect_pt(L0,L1) AND
                             on_line?(pnot,L0)
                          IMPLIES
                             NOT on_line?(pnot,L1)

same_line_lem        : LEMMA p0 /= p1 AND
                            ( on_line?(p0,L0) AND on_line?(p0,L1) AND
                              on_line?(p1,L0) AND on_line?(p1,L1) )
                                 IMPLIES same_line?(L0,L1)

not_same_line        : LEMMA on_line?(p,L0) AND
                             NOT on_line?(p,L1)
                          IMPLIES
```

```
                              NOT same_line?(L0,L1)

   intersect_pt_lem     : LEMMA NOT same_line?(L0,L1) AND
                             on_line?(pnot,L0)  AND
                             on_line?(pnot,L1)
                          IMPLIES
                              intersect_pt(L0,L1) = pnot
```

1.5 Closest Approach

The theory closest_approach_2D provides some tools to calculate the point of closest approach (CPA) between two points that are dynamically moving in a straight line. This is an important computation for collision detection. For example, this can be used to calculate the time and distance of two aircaft (represented as line vectors) when they are at their closest point.

Suppose we have two time-parametric linear equations

$$\vec{p}(t) = \vec{p_0} + t\vec{u} \qquad \vec{q}(t) = \vec{q_0} + t\vec{v}$$

Minimum separation occurs at:

$$t_{\mathsf{cpa}} = -\frac{\vec{w_0}(\vec{u} - \vec{v})}{|\vec{u} - \vec{v}|^2}$$

where $\vec{w_0} = \vec{p_0} - \vec{q_0}$. The library provides a function time_closest:

```
time_closest(p0,q0,u,v): real =
   IF norm(u-v) = 0 THEN  % parallel, eq speed
      0
   ELSE
      -((p0-q0)*(u-v))/sq(norm(u-v))
   ENDIF
```

The following lemma gives an alternate way to calculate the function.

```
time_closest_lem: LEMMA norm(u-v) /= 0 AND
                        a = (u-v)*(u-v) AND
                        b = 2*(p0-q0)*(u-v)
                        IMPLIES
                          time_closest(p0,q0,u,v) = -b/(2*a)
```

The lemma time_cpa establishes that this time is indeed the point where the distance is at a minimum.

```
time_cpa: LEMMA t_cpa = time_closest(p0,q0,u,v)
             IMPLIES
                is_minimum?(t_cpa,(LAMBDA t: sq_dist(p0+t*u,q0+t*v)))
```

See

`http://geometryalgorithms.com/Archive/algorithm_0106/algorithm_0106.htm`

for a very nice discussion.