APPENDIX B

Formula Language Reference

This appendix describes the Working Model 2D formula language.

NOTE: The formula language is a different system from *Working Model Basic* (WM Basic). Although they share similar syntax and numbering schemes, they are used for different purposes. WM Basic is a language system used to control Working Model 2D, while the formula language is a high-performance, "light-weight" language used by Working Model 2D objects during simulations. For more information on WM Basic, please refer to the accompanying *Working Model Basic User's Manual.*

B.1. About Formulas

Formulas follow standard rules of mathematical syntax, and strongly resemble the equations used in spreadsheets and programming languages. Formulas are composed of *identifiers*, *fields*, *operators*, and *functions*. The following sections discuss each category in detail.

Formulas can be up to 255 characters in length. Capitalization and spacing do not affect formulas, although identifiers and function names must not contain spaces. Parentheses behave as they do in standard algebraic manipulations.

B.2. Numeric Conventions

Numbers in Working Model 2D use the standard scientific notation of spreadsheets such as Lotus 1-2-3 and Excel, and computer languages such as BASIC, PASCAL and C.

Exponents

Exponents are displayed in the following way:

In Printed Text	In Working Model 2D
$123x10^{3}$	`123e3`
$1.001x10^{-22}$	`1.001e-22`

Angle Measures

All angles are expressed in radians. An angle of 360° has a radian measure of 2π.

NOTE: Angles in formulas are expressed in radians although the default display mode for Working Model 2D is degrees.

B.3. Identifiers

Identifiers are used in formulas to identify an object. There are five types of identifiers. When creating formulas, you can use one or more of these types:

```
body[3]

point[2]

constraint[44]

output[12]

input[5]
```

The number within the brackets is the object ID. Each object in Working Model 2D has a unique ID. To find the ID of an object, double-click the object to display the Properties window for that object. The ID appears with the proper formula syntax in the top of the Properties window. In addition, the identifier of an object is displayed in the Status bar when the pointer is over the object (see **Chapter 6, "The Workspace"**).

For example, `body[10]` is the ID for body #10.

Body[]

`Body[]` is the identifier for bodies, such as circles, polygons, and rectangles.

Point[]

`Point[]` is the identifier for point objects. Point objects are either isolated points, or the points which compose the endpoints of a constraint. `Point[11]` is the ID for point #11. (See "Body Fields" on page B–6 for polygon vertices.)

Constraint[]

`Constraint[]` is the identifier for constraint objects, including springs, ropes, joints, and pulleys.

Output[]

`Output[]` is the identifier for all meters.

Input[]

`Input[]` is the identifier for all input controls, including sliders, text boxes, and buttons.

If you use an identifier with an ID for an object that does not exist, the result will be a "Null" object. Null objects return 0.0 for all of their properties.

B.4. Fields

Each identifier in the Working Model 2D formula language can have fields. You use fields to access the values of basic properties such as position and velocity.

Fields are specified by a "Type", followed by a period (.) and a field name. To access the moment of a body with an ID of 3, you would enter the formula:

```
body[3].moment
```

The value returned from `body[3].moment` is a number that can be used in any formula.

Any value that has an x, y, and rotational component is returned as type vector. The vector type has three fields: `.x`, `.y`, and `.r` (rotation).

Fields are a way of accessing smaller components of some bigger object.

Sometimes you will use two fields in a row. To obtain the rotation of a body, you enter the following formula:

```
body[2].p.r
```

This equation has two hierarchical fields. First, `body[2].p` produces the position field of body #2, which is a vector. Next, the "`.r`" produces a rotation value (i.e., the orientation of the body) from the position field.

`body[2]`	body type
`body[2].p`	vector type
`body[2].p.r`	number type

The following is a list of all fields with the type of value that each field produces. See the following sections for field descriptions.

Type	**Field**	**Type Returned**
Vector	.x	number
	.y	number
	.r	number
Body	.p	Vector
	.v	Vector
	.a	Vector
	.mass	number
	.moment	number
	.charge	number
	.staticfric	number
	.kineticfric	number
	.elasticity	number
	.cofm	Point
	.width	number
	.height	number

	.radius	number
	.vertex[n].x	number
	.vertex[n].y	number
Point	.p	Vector
	.v	Vector
	.a	Vector
	.offset	Vector
	.body	Body
	.force	Vector
Constraint	.length	number
	.dp	Vector
	.dv	Vector
	.da	Vector
	.p1	Point
	.p2	Point
	.force	Vector
Output	.x	number
	.y1	number
	.y2	number
	.y3	number
	.y4	number
Input		number

Vector Fields

x, y, r

Notice that position, velocity, and acceleration are always returned as type "vector" in the above table. For example,

`point[4].a`	acceleration of point #4
`body[3].v`	velocity of mass center of body #3

are both of type vector. You cannot enter these formulas in a text field, because they are not numbers.

To access individual components of these vectors, you must designate whether you want the x, y or rotational components.

To get a number, enter the following equation instead:

`body[3].v.x`	x velocity of mass center of body #3

body[3].v.x represents the x velocity of the mass center of body #3, which is a number, not a vector.

The subfield ".r" returns the rotational component of any vector.

Body Fields

p, v, a

These are the current values of position, velocity and acceleration. Each of these fields returns a value of type vector. Thus, to use any of these field you need to add one of the vector fields (x, y, r).

`body[1].p.x`	x position of mass center of body #1
`body[3].v.y`	y velocity of mass center of body #3
`body[37].a.r`	angular acceleration of body #37

mass, moment, charge, staticfric, kineticfric, elasticity

These are the current values of the various properties.

`body[3].charge`	charge of body #3
`body[14].mass`	mass of body #14

cofm

This field returns the kinematic properties of the center of mass of a body. The expression:

```
body[3].cofm
```

is the same type as points, so it has all the fields available to a point (see "Point Fields" on page B–7). For example, the expression:

```
body[3].cofm.p.x
```

returns the x coordinate of the center of mass. Similarly:

```
body[3].cofm.v.x
```

returns the x component of COM's velocity.

The next four fields (`width`, `height`, `radius`, `vertex[n]`) are called *geometry-based formula* and return the geometric information of bodies. You can use these fields to position endpoints of constraints precisely (see "Using Geometry-based Formulas (Point-based Parametrics)" on page 4–16).

width

Returns the width of a rectangle or a square. The `width` field is not valid for other body types. For squares, `width` is always identical to `height`.

height

Returns the height of a rectangle or a square. The `height` field is not valid for other body types. For squares, `height` is always identical to `width`.

radius

Returns the radius of a circle. The `radius` field is not valid for other body types.

vertex[n].x, vertex[n].y

For a polygon, `vertex[n].x` and `vertex[n].y` return the x and y coordinates of the *n*-th vertex, respectively. The number n ($n \geq 1$) corresponds to the vertex ID number shown in the Geometry window for the polygon. The coordinates are given in terms of the frame of reference of the polygon (see "Frame of Reference (FOR)" on page 3–10).

For a rectangle and square, the `vertex[1]` corresponds to the top right corner (when the body orientation is 0), and the subsequent indexing (`2` through `4`) returns the other vertices in a counter-clockwise order.

The expression `vertex[n]` is not valid for a circle.

Point Fields

p, v, a

These are the current values of position, velocity and acceleration. Each of these fields returns a value of type vector. Thus, to use any of these fields you need to add one of the vector fields (x, y, r).

`point[1].p.x`	x position of point #1

The position of a point is given in terms of the global coordinates.

offset

The `offset` field returns the vector containing the current configuration (`.x`, `.y`, and `.r`) of the point element in terms of the FOR (frame of reference) of the body to which the point is attached (local coordinates).

If the point element is attached to the background, the offset field is equivalent to the `.p` field of the point element. That is:

```
point[n].p.x = point[n].offset.x
```

and similarly for the `.y` and `.r` fields.

body

The `body` field returns the body to which the point element is attached. See "Body Fields" on page B–6 for associated fields.

force

The `force` field returns a vector representing the force acting on the point—to be more precise, the force acting on the body at the point. The components are given in terms of the global coordinates, regardless of what the point element is attached to.

Constraint Fields

length

This is the current distance between the two points of the constraint. To find the current length of a spring, you would enter:

`constraint[3].length`	length of constraint #3

dp, dv, da

These are the current values for the difference in position, velocity, and acceleration between the two points of the constraint. Each of these fields returns a value of type vector.

These values measure in the constraint's reference frame. The x value is measured along the line connecting the two points of a point to point constraint.

To find out how fast the length of a spring is changing (the difference in velocity between the two endpoints of the spring), you enter the following formula:

```
constraint[3].dv.x
```

p1, p2

Each of these fields returns `point` that serves as an endpoint of the constraint. The `p1` field returns the point element that was first created. See "Point Fields" on page B–7 for associated fields.

force

The force field returns the `vector` representing the constraint force. The field is equivalent to `constraintforce(n)` (see "Simulation Functions" on page B–21).

Output Fields

x

This is the value displayed on the x-axis or the abscissa of an output graph.

`output[6].x`	value displayed on x axis of output 6

y1, y2, y3, y4

These are the values displayed on the y axis of an output graph.

`output[6].y1`	value displayed on y1 axis of output 6
`output[6].y2`	value displayed on y2 axis of output 6
`output[6].y3`	value displayed on y3 axis of output 6
`output[6].y4`	value displayed on y4 axis of output 6

B.5. Operators

Operators include all of the common algebraic symbols (+, -, >, =). The following operators require one or two numbers. The letters "a" and "b" are used as place holders for any number or formula that evaluates to a number.

Numeric Operators

The following is a listing of numeric operators that are available for use in formula entry:

Operator	Input(s)	Output
- (negate)	a	-a
+ (plus)	a + b	a + b
- (minus)	a - b	a - b
* (multiply)	a * b	a x b
/ (divide)	a / b	a / b
% (mod)	a % b	a mod b
^ (power)	a ^ b	a^b
>	a > b	1 or 0
<	a < b	1 or 0
>=	a >=b	1 or 0
<=	a <=b	1 or 0
= (equal)	a = b	1 or 0
<>(not equal)	a <>b	1 or 0

These operators require numbers as their inputs. This means that you cannot add most formula elements that are not a number.

Incorrect:

`body[3] + point[3]`	cannot add a body to a point
`body[3].p - 34.5`	cannot subtract a number from a vector
`point[7].v + body[3]`	cannot add a vector to a body
`body[3].p > 44.0`	cannot compare a vector to a number

Correct:

```
body[3].p.x + point[3].p.x

body[3].p.x - 34.5

point[7].v.y - body[3].v.y

body[3].p.y > 44.0
```

```
body[3].p.y = 44.0

body[3].p.y != 44.0
```

- (negate)	Takes a single number and returns the negative of the number.
+ (plus)	Takes two numbers and returns the sum.
- (minus)	Takes two numbers and returns the difference.
* (multiply)	Takes two numbers and returns the product.
/ (divide)	Takes two numbers and returns the quotient.
% (mod)	Takes two numbers and returns the remainder of the first value divided by the second.
^ (power)	Takes two numbers and returns the first value raised to the power of the second value.
> (greater than)	Takes two numbers and returns the value 1 if the first value is greater than the second value. Otherwise, returns the value 0.
< (less than)	Takes two numbers and returns the value 1 if the first value is less than the second value. Otherwise, returns the value 0.
>= (greater than or equal to)	Takes two numbers and returns the value 1 if the first value is greater than or equal to the second value. Otherwise, returns the value 0.
<= (less than or equal to)	Takes two numbers and returns the value 1 if the first value is less than or equal to the second value. Otherwise, returns the value 0.
= (equal)	Takes two numbers and returns the value 1 if the two values are equal. Otherwise, returns the value 0. This operator *does not* assign any value to the left side of the equation. The formula:

```
body[3].p.y = 3
```

returns 1 if body #3's y position equals 3.0. This formula does not set any values of body #3's position.

<> (not equal)

Takes two numbers and returns the value 1 if the two values are not equal. Otherwise, returns the value 0.

Operator Precedence

Use parentheses to set the order of equation evaluation. All equations are normally evaluated from left to right. Precedence is given to operators in the following order. (operators listed in the same row have equal precedence):

() [] .	highest precedence
* / ^ %	
+ - (binary operators)	
< <= > >=	
=	lowest precedence

Arithmetic Operators

Operators with the highest precedence are applied first. For example, the following formula:

```
3 + 2 * 4
```

is evaluated as 3+(2*4) instead of as (3+2)*4. This is because the multiplication (*) operator has a higher precedence than the addition (+) operator.

Use parentheses to change the order of evaluation, or to assure yourself of the order of evaluation if you're not quite sure of the precedence of various operators. In the above example, you could enter the formula as

```
(3 + 2) * 4
```

to force evaluation of the addition before the multiplication.

You can nest parentheses, as in the formula

```
((3 + 2) * 4 + 10) / 2
```

Be sure to use parentheses, and not brackets ([]) or braces ({ }).

Note on Inequalities

Although the inequality operators have the same precedences, the return value of the formula:

```
if (0 < t <=1, 50, 100)
```

is actually equivalent to:

```
if ((0 < t) <= 1, 50, 100)
```

since the chain of the binary operators is evaluated from left to right. As a result, the above formula *always* returns 50 regardless of the value t (since ($0 < t$) returns 1 or 0, the entire first argument is always 1, or true). If you want the effect of "return 50 when t is between 0 and 1, or else return 100", you should type:

```
if(and(0 < t, t <= 1), 50, 100).
```

Please refer to "List of Functions" on page B–16 for detailed discussions on each function.

Vector Operators

The following operators will work on vectors.

Operator	Input(s)	Output
- (negate)	vector	vector
+ (plus)	vector,vector	vector
- (minus)	vector,vector	vector
* (multiply)	number,vector	vector
|| (magnitude)	vector	number

These operators require that their input types match those listed in the previous chart. Vector operators are useful for simplifying formulas. To display a meter showing the distance between two bodies, you would enter the following formula:

```
|body[3].p - body[2].p|
```

This formula contains two vector operators. First, the "-" operator was used to subtract the two positions of the bodies:

```
body[3].p - body[2].p
```
result is a vector

The chart above indicates that the minus (-) operator can be used on two vectors, and that the result is a vector. The result can then be used with the magnitude(II) operator to produce a number. The following table shows some of possibly common mistakes and corrections.

Incorrect	Correct
body[2].lal	lbody[2].al
lbody[2]l.a	lbody[2].al
lbody[2].a.xl	abs(body[2].a.x)
body[2].a.x + body[2].v	Unify both operands to vectors or numbers.

- (negate)

Takes a vector quantity and returns the negative of the quantity. The .x, .y, and .r fields of the vector are all negated.

`body[3].p.x`	value is 10.0
`-body[3].p.x`	value is -10.0
`(-body[3].p).x`	value is -10.0

In the last case, the value of body[3].p is negated as a complete vector.

+ (plus)

Takes two vectors and returns a vector which is the sum. The vector which is returned will have each of its fields (.x, .y, .r) equal to the sum of the corespondent fields of the two vectors being added.

- (minus)

Takes two vectors and returns a vector which is the difference. The vector which is returned will have each of its fields (.x, .y, .r) equal to the difference of the corespondent fields of the two vectors being added.

** (multiply)*

Takes a vector and a number and returns the scalar product. The vector which is returned will have each of its fields (.x, .y, .r) equal to the product of the number and the corresponding field of the multiplied vector.

|| (magnitude)

Takes a vector and returns a number which is the magnitude of the .x and .y fields. Magnitude is equal to the length of a line drawn from (0,0) to the (.x, .y) fields of the vector. The number returned from the magnitude function is equal to:

```
|v| = sqrt(v.x*v.x + v.y*v.y)
```

B.6. Functions

Functions take from zero to three arguments, and return a number or vector value. All functions accept their arguments in the form

```
function(arg1, arg2.....)
```

There are two kinds of functions available. Math functions perform standard mathematical operations. Simulation functions return information from Working Model 2D simulations.

List of Functions

Name	Inputs	Output
abs	number	number
and	number,number	1 or 0
angle	vector	number
acos	number	number
asin	number	number
atan	number	number
atan2	number,number	number
ceil	number	number
cos	number	number
exp	number	number
floor	number	number
if	number,number,number	number
ln	number	number
log	number	number
mag	vector	number
max	number,number	number
min	number,number	number
mod	number,number	number
not	number	1 or 0
or	number,number	1 or 0
pi		π
pow	number,number	number
rand		number
sign	number	1 or -1
sin	number	number
sqr	number	number
	vector	number
sqrt	number	number
tan	number	number
vector	number,number	vector

abs(x)

Takes a number and returns the absolute value of the number. Example:

```
abs(body[3].p.x)
```

returns the absolute value of body #3's x position.

and(x,y)

Logical AND operation. Takes two numbers and returns the value 1 if both numbers are not 0. Otherwise, returns the value 0. Example:

```
and(time>1 , body[2].v.y>10)
```

returns the value 1 if time is greater than 1 *and* body #2's y velocity is greater than 10.

angle(v)

Takes a vector and returns the angle the vector makes with the coordinate plane. For example, if a body has a velocity of 0 in the x direction, and 10 in the y direction, the body has a velocity that is in the direction of 90° or $\pi/2$ on the coordinate plane. The formula

```
angle(body[3].v)
```

would return the value of $\pi/2$.

acos(x)

Takes a number and returns the inverse cosine of the number. Values are returned in the range $[0,\pi]$.

asin(x)

Takes a number and returns the inverse sine of the number. Values are returned in the range $[-\pi/2, \pi/2]$.

atan(x)

Takes a number and returns the inverse tangent of the number. Values are returned in the range $[-\pi/2, \pi/2]$.

atan2(y,x)

Takes two numbers and returns the inverse tangent of y/x. This function is useful because unlike the atan function, it can generate an angle in the correct quadrant. Values are returned in the range $[-\pi, \pi]$.

ceil(x)

Takes a number and returns the smallest integer no smaller than the number.

cos(x)

Takes a number and returns the cosine of the number.

exp(x)

Takes a number and returns the exponential of the number. (*e* raised to the value of the number).

floor(x)

Takes a number and returns the largest integer no larger than the number.

if(x,y,z)

Takes three numbers. If the value of the first number (x) is not equal to 0, then returns the value of the second number (y). Otherwise, returns the value of the third number (z). Example:

```
if(time>1, 20, 0)
```

returns the value 20 if time is greater than 1, otherwise returns the value 0.

Typically, the first argument of an `if` function is a relation (such as `x >` y) or a logical operation (such as `and(a, b)`). You can write nested if-statements by recursively using other `if()` functions as its own arguments.

For example, shown below is a somewhat naive C-code segment which returns the maximum of three numbers a, b, and c:

```
{
    if (a > b) {
        if (a > c)
            return a ;
        else
            return c ;
    }
    else {
        if (b > c)
            return b ;
        else
            return c ;
    }
}
```

In the formula language of Working Model 2D, the above segment can be translated into a single line as follows:

```
if(a>b,if(a>c,a,c),if(b>c,b,c))
```

ln(x)

Takes a number and returns the natural logarithm of the number.

log(x)

Takes a number and returns the base 10 logarithm of the number.

mag(v)

Takes a vector and returns the magnitude of the vector. Result is the same as |v|.

max(x,y)

Takes two numbers and returns the larger of the two numbers. Example:

```
max(body[1].a.x , body[2].a.x)
```

returns the larger x acceleration of either body #1 or body #2.

If you wish to find the maximum of three numbers `a`, `b`, and `c`, you could recursively use the `max()` functions as follows:

```
max(max(a,b),c)
```

min(x,y)

Takes two numbers and returns the smaller of the two. Example:

```
min(body[1].v.x , body[2].v.x)
```

returns the smaller x velocity of either body #1 or body #2.

As in the `max()` function, you could find the minimum of three numbers `a`, `b`, and `c` as:

```
min(min(a,b),c)
```

mod(x,y)

Takes two numbers and returns the remainder when the first value is divided by the second.

not(x)

Logical NOT operation. Takes a number and returns the value 0 if the number is *not* 0. Otherwise, returns the value 1.

or(x,y)

Logical OR operation. Takes two numbers and returns the value 1 if at least one of the numbers is not 0. Returns 0 if and only if both numbers are 0. Example:

```
or(time>1 , body[2].v.r>10)
```

returns the value 1 if time is greater than 1 *or* body #2's angular velocity is greater than 10.

pow(x,y)

Takes two numbers and returns the value of x raised to the power of y; i.e., returns x^y.

pi()

Returns the value of π.

rand()	Returns a random value between 0 and 1.
sign(x)	Takes a number and returns the value 1 if the number is greater than or equal to zero. Otherwise, returns the value -1.
sin(x)	Takes a number and returns the sine of the number.
sqr(x)	Takes a number or a vector. If the input is a number, returns the square (x * x) of the number. If the input is a vector, returns the sum of the .x field squared and the .y field squared.
sqrt(x)	Takes a number and returns the square root of the number.
tan(x)	Takes a number and returns the tangent of the number.
vector(x,y)	Takes two numbers and returns a vector composed of the two numbers. The first number (x) becomes the .x field of the vector. The second number becomes the .y field of the vector.

Simulation Functions

Simulation functions are used to extract data from the simulation. These functions are used in the various meters and vectors of Working Model 2D.

Name	Inputs	Output
constraintforce	number	vector
	number,number	vector
	number,number,number	vector
frame		number
frictionforce	number,number	vector
groupcofm	number	vector
kinetic		number
length	number,number	number
normalforce	number,number	vector
section	number,vector	number

constraintforce(x)

Takes the ID number of a constraint (x), and returns a vector describing the current force being applied by the constraint. To find the compression in a spring, use the formula:

```
constraintforce(3).x
```

In point to point constraints, the .x component of the force vector is always measured along the line connecting the two endpoints. In constraints that apply a torque, the .r component of the constraint force contains the value of applied torque.

For pin joints, the x and y components are given in terms of the global coordinate axes.

constraintforce(x,y)

Takes the ID number of a constraint (*x*), and the ID number of a body (*y*). Returns the amount of force being applied by the constraint on the body as a vector. This function is used by meters which measure gravity, air resistance, electrostatic and custom force fields. The ID numbers for these four constraints are constant, and are described in the next section.

The gravity constraint always uses constraint ID #10002. To measure the force imposed on a body by the linear gravity constraint, use the formula

```
constraintforce(10002, 3).y
```

In this case, the .y suffix is used to get the value of force in the y (up and down) direction.

constraintforce(x,y,z)

Takes the ID number of a constraint (*x*), and the ID numbers of two bodies (*y* and *z*). Returns the amount of force being applied by the constraint between the two bodies.

This function only returns values for forces which are applied to each pair of bodies (planetary gravity, electrostatics, and custom force fields). The ID numbers for these constraints are constant, and are described in the next section. The gravity constraint always uses constraint ID #10002. To measure the force of gravity between two specific bodies in a planetary system, use the formula

```
constraintforce(10002,3,5).x
```

As with point to point constraints, the .x value of the vector measures the force applied along the line that connects the center of mass of the two bodies.

frame()

Returns the current frame number. The initial conditions are defined as being frame zero.

frictionforce(x,y)

Takes the ID numbers of two bodies (x and y) and returns the friction force of the first object acting upon the second. The value is returned as a vector.

groupcofm(x)

Takes the ID number of a group (x) and returns the center of mass of all bodies in the group. Currently, the only defined group is group #0, which is the group that contains all bodies.

kinetic()

Returns the total kinetic energy of all bodies in the simulation as a number.

length(x,y)

Takes the ID numbers of two bodies (x and y) and returns the length of the line connecting their centers of mass.

normalforce(x,y)

Takes the ID numbers of two bodies (x and y) and returns the contact force of the first object acting upon the second. The value is returned as a vector.

Working Model 2D considers the contact force as the sum of the normal force and the collision force. Please see **"A.7. Simulating Collisions"** for more information.

section(x,v)

Takes the ID number of a body (x) and a vector quantity (y). Returns the cross sectional width of the body in the direction of the vector. For example, `section(body[1], vector(1,0))` will return the vertical cross section width of `body[1]`. This function is used by the air resistance force field to approximate the drag on bodies.

B.7. Predefined Values

Variables

There are several predefined variables that you can use in formulas.

Name	**Type**
`time or t`	`number`
`self`	`mass`
`other`	`mass`
`ground`	`mass`

time or t

Returns the current time in the simulation. Time always begins at 0.0 in frame #0.

self

Returns a body type when placed inside a force field equation. When force field equations are evaluated for each body in the simulation, "self" assumes the value of the current body on which the force field is being applied. For example, the equation for a linear gravitational field is

```
Fy:  - self.mass * 9.81
```

A force is applied to each body in the simulation. The value of "self" assumes the value of the specific body onto which force is being applied. Thus, in this case each body has a force applied equal to -9.81 times its own mass.

When force fields are evaluated for each pair of bodies (pair-wise fields), the value of "self" assumes the body of the first body in each pair.

other

Returns a body type when placed inside a force field equation. When force field equations are evaluated for each pair of bodies in the simulation (pair-wise fields), "other" assumes the value of the second body of each pair.

For example, the force field equation for planetary gravity is

```
-self.mass * 6.67e-11 / sqr(self.p -
other.p) * other.mass
```

or more commonly: $\frac{Gm_1m_2}{r^2}$.

This equation is applied to each pair of bodies in the simulation. As the equation is applied to each pair of bodies, "self" assumes the value of the first body and "other" assumes the value of the second in the pair.

ground

Returns a body type for the background. This is essentially a body at location 0,0 that never moves.

Constants

Four ID numbers are reserved for the global force fields of gravity, electrostatics, air resistance, and the custom force field. You will see these ID numbers in the formulas used in meters to measure forces produced by these constraints. These ID numbers are as follows:

Force Field	Reserved ID
gravity	10002
electrostatics	10004
air resistance	10006
custom force field	10008

If you create a meter to measure the force of gravity on a body, you will see a formula such as

```
constraintforce(10002, 3).y
```

This formula gives the y component of the force applied by constraint #10002 on body #3. The value 10002 is automatically inserted in the formula for this meter as the constraint ID for the gravity force.