ATOM Simulation Specifications and Implementation Details

April 14, 2017

1 Specifications

ATOM provides easy-to-use general-purpose simulation interface.

A complete simulation consists of definitions of *object*s(*watch*es), *rule*s and optionally *assertion*s.

A simulation will use data from the system, applying rules to generate new data, and log the watched variables. (see below for details).

1.1 Predefined variables and constants

Predefined values are all globally accessible.

variable	meaning	vector?	comment
t	time	N	
g	gravitational acceleration	Y	constant
G	gravitational constant	N	constant
c	speed of light	N	constant
e	base of natural logarithm	N	constant
π(pi)	circles' circumference / diameter	N	constant

Constant value may be override. Constant means the value *should* not change during simulation.

1.2 Mathematical representations

Expressions with elementary operations (addition, subtraction, multiplication, division, and powering), trignometric and inverse trignometric functions, and logarithm must be implemented.

Vectors in expressions are handled as in mathematics, but may have different syntax with scalars.

1.3 Identifier

Identifier is anything used to distinguish objects. Identifier must be unique to any object within the same type. Identifier may be either input by user or assigned internally by an implementation. One object may have multiple identifiers. Identifers end with underscores are reserved.

1.4 Object

Object refers to any physical object involved in a physical process. *Object* in a simulation is made of fields (see *Watch*) and values.

1.5 Watch

Watch defines what variable are of interest in simulation. Watched variables are kept throughout the simulation process and will be written to output.

Watch specification is ordered.

1.6 Rule

*Rule*s provide information of interaction and define how interactions are applied to *object*s.

*Rule*s can be thought as extended form of forces **or** constraints physically and routines programmatically.

*Rule*s can be conditional or not and be applicable to a set of choice of *object* combinations or all combinations. Conditional *rule* may be associated with *rule* pack. A *rule* is conditional if it is only applicable when a predicate is met, and the predicate is not relevant to identity of an *object*. Conditional block will have all variables defined if they appeared new, no matter if they're explictly stated in all part of the pack. An implementation must at least provide syntax for adding *rule* for specific *object* with arbitrary number of *object*s referenced either by identifier or by number.

The simulation may skip the rest if all needed information are obtained before reaching the end (see *Watch*). Implementation shall check if the system is overdetermined, and should halt rather than silently solve the overdetermined system.

*Rule*s are resolved in the order of user input. Implmentation are entitled to choose how many variable to update in a single cycle as well as whether to update them in place or not. Implementation may simplify the resolving procedure if proved equivalent *mathematically*.

1.7 Assertion

Assertions are used to verify some condition or predicate through the simulation process. The simulator shall halt if an assertion fails. Assertions shall be verified immediately when simulator comes across it. If an assertion fails due to insufficient variable defined, simulator shall still report the error. [*Note*: Simulator shall report the error either statically or dynamically, yet is encouraged to do it as early as possible.]

1.8 Simulation

Simulation proceeds by increment time `t` by user defined step. Implementation may choose to use a different step no greater than the user set one.

Implementation shall at least support MKS, CGS and FPS unit system, and appropriate conversions.

The single-step error shall be at most 10^{-6}, with respect to your selected unit system absolutely **or** relatively. [*Example*: 1×10^{-7} may compare equal to 5×10^{-7} and 10^7 may compare equal to $10^7 + 1$.] For implementation with binary floating-point numbers, additional rounding error must be considered.

1.9 Standard library

1.9.1 mechanics/mass_point

This file defines common dynamic object with default values.

1.9.2 mechanics/rules

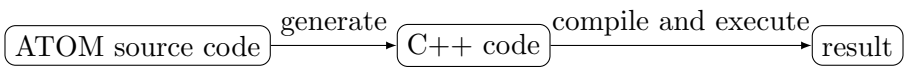
This file defines several common passive forces and law of real mass point motion.

1. Normal force applies $\mathbf{F_N}$ if $\mathbf{pos_i} - \mathbf{pos_j} = \mathbf{0}$ and $\mathbf{v_i} \times \mathbf{v_j} = \mathbf{0}$ where the subscript i and j refers to distinct *object*s. For tangent plane defined by $\mathbf{p} = r\mathbf{u} + s\mathbf{w} + \mathbf{p_0}$, the effect is that some $\mathbf{F_N}$ satisfying $\mathbf{F_N} \times (\mathbf{u} \times \mathbf{w}) = \mathbf{0}$ [*Note*: that is, $\mathbf{F_N} = k\mathbf{u} \times \mathbf{w}$] is applied so that $(\mathbf{a_i} - \mathbf{a_j}) \cdot (\mathbf{u} \times \mathbf{w}) = \mathbf{0}$, and $\mathbf{F_N}$ is either defined to the resolved value or $\mathbf{0}$ if not applicable.
2. Collision solver is triggered if $\mathbf{pos_i} - \mathbf{pos_j} = \mathbf{0}$ where the subscript i and j refers to distinct *object*s. The total momentum of the pair is conserved, with coefficient of restitution e defined by material.
3. Static friction $\mathbf{f}$ applies if $\mathbf{v_i} - \mathbf{v_j} = \mathbf{0}$ and $\mathbf{F_N} \neq \mathbf{0}$. For the plane defined above and the effect is that to the extent $\mathbf{f} \leq \mu_s \mathbf{F_N}$ and $\mathbf{f} \cdot (\mathbf{a} \times \mathbf{b}) = \mathbf{0}$, $\mathbf{f}$ is applied and hereby defined so that $\mathbf{v_i} = \mathbf{v_j}$, otherwise $\mathbf{f}$ is undefined.
4. Dynamic friction $\mathbf{f}$ applies if $\mathbf{v_i} - \mathbf{v_j} \neq \mathbf{0}$ and $\mathbf{F_N} \neq \mathbf{0}$ and the effect is $\mathbf{f} = -\frac{\mu \|\mathbf{F_N}\|}{\|\mathbf{v_i} - \mathbf{v_j}\|} \cdot (\mathbf{v_i} - \mathbf{v_j})$.
5. Newton's second law of motion, $\mathbf{RF} = m\mathbf{a}$.
6. Velocity is integration of acceleration w.r.t time, $\dot{\mathbf{v}} = \mathbf{a}$
7. Displacement is integration of velocity w.r.t time, $\dot{\mathbf{pos}} = \mathbf{v}$

2 Implementation

2.1 Overview

ATOM is implemented in static fashion, that is, code is statically analyzed, and generates code that is further compiled by another compiler.



2.2 Typing

[*Note*: ATOM in this section means this very implementation of ATOM, for sake of simplicity.]

ATOM is strongly typed. Types in ATOM designates a specific arrangement of data. Each type in this ATOM has two forms, namely scalar form (basic form) and packed form (vector form or array form).

2.2.1 Cast

Cast is always carried out in a bit to bit fashion. It is therefore always possible to cast a type to another type.

[*Note*: User should carefully handle padding according the specific platform.]

2.3 Mathematics

2.3.1 Expressions and vectors in expressions

Expressions resembles they are in mathematical context but indices on functions like $\sin^2()$ are not yet accepted.

Symbols are treated as scalars unless declared to be vectors or otherwise specified. The declaration may either occur in *object* definition and *rule* definition. Vector addition and subtraction can be done in the way it was, but vector plus or minus scalar will be rejected. Vectors have different syntax on multiplication and some other operations.

Vector scalar multiplication is handled via method `.mul`. The method `.dot` and `.cross` are for inner product and for cross product, respectively. Vector also provides `.mag` for magnitude, `.angle` for angle between two vectors, `.unit` for obtaining unit vector, `.proj` to obtain projection over another vector. Users are generally encouraged to use these functions for better performance.

All the methods provided with vector that returns a vector returns a new vector.

2.3.2 Differentiation and integration

Differentiation and integration are special operations handled by `diff()` and `int()` functions. These functions accept two argument, where the first is the expression and the second is the variable to take derivative with respect to, which is optional and default to `t`. Alternatively, the prime notation may be applied to any variable to obtain derivative w.r.t. time. However, second-order derivative cannot be used since `''` are considered to have special meanings.

2.4 Source format and syntax

Complete source files consist of exactly three parts, namely objects, rules and start directive, one after another. A source file may contain exactly one start directive which must be the last non blank line. One line may contain exactly one statement. The source file is newline-sensitive and indent-sensitive. Specially everything after `#` till newline are treated as comment.

2.4.1 Accessing watched variables of another object

Watched variables of other *object*s can be accessed by adding `$n.` where `n` is an nonnegative integer. `$1`, `$2` ... point to other *object*s arranged in ascending order of definition. The implementation will provide arguments to the largest number given as sorted permutation. [*Note*: Explicit state of `$0` is unnecessary and not recommended.]

2.4.2 Units and literals

The default unit system is the SI system. Default system is applied to numbers when a specific dimension is needed and user has not provided one. User can set default system with compiler option.

User may use explicit unit literals anywhere when a dimension is needed. This number will be converted to the default system automatically by the compiler.

2.4.3 Protected names

There are a few names that are used in the implementation or have special meaning or are reserved for future use, so that they must not be used as identifiers.

```
 None   True   False   global   using
object  rule   const    real   vector
  if    else   assert   start    step
watch    def     Inf     NaN      Pi
```

Beside those listed above, identifiers shall not clash with names of specification mandated global variables.

2.4.4 Using directive

Using directive `"using" <filename>`

Using directives are used to import data from other file, which are done by textual replacement. Using directives may appear anywhere in a source file.

2.4.5 Object section

Object section may only contain *object* definitions.

***Object* definition**
`"object" ":" [<type spec>] <name> [["," [<type spec>] <name>] ...]`

***Object* initialization** `"object" <name> ":" [<name> ":="] <value>`

Object must be defined before first reference. Values used to initialize *object*s must be known at compile time. [*Example*: An *object*'s property `b.x` may be initialized to `a.x`, provided `a` is initialized before `b`.] Type specification may be omitted if the variable is `real`.

2.4.6 Rule section

Vector declarations, *rule* and conditional *rule* definitions and assertions may appear in this section.

***Rule* definition** `"rule" ":" <eqn>+`

Vector declaration `"vector" <name>`

Conditional rule definition `"rule" ":" "if" <condition> ":" <eqn>+`

Assertion `"assert" <condition>`

2.4.7 Start directive

Start directive `"inc" <var> ["till" <value>] ["by" <value>]`

2.5 Processing the source

2.5.1 Determining the order to update each variable

When the simulator come across a *rule*, it has to determine the variable(s) to solve for.

The simulator follows the following precedence rules:

1. unwatched temporary variables
2. watched variables
3. global variables
4. constant values (for comparison only, results in an error)

[*Example*: For a *rule* `F = -2 * v`, `F` will be updated since `v` has lower precedence.][*Note*: temporary variables shall not have the same name with global variables.]

When there are multiple variables with same precedence, the order of presence or definition is used.

[*Example*: For *rule* $\dot{\mathbf{pos}} = \mathbf{v}$, under default watches, it would use **v** to update **pos**, because **v** is *watched* before **pos**.]

If some equation cannot be resolved itself alone, the compiler would silently try to combine it with sequent equations until it can. However, users may use paired curly braces as hints to compiler, which will always be checked, and results in an error if impossible.

The exact order is determined as a part of compilation.

2.6 Code generation

THIS PAGE INTENTIONALLY LEFT BLANK