Lelaps and GODL

Fast Detector Simulation Using Lelaps
Detector descriptions in GODL
Overview

- Introduction
- Lelaps
  - CEPack tool kit
  - Usage
  - Performance
- GODL geometry description
- Future
Introduction

- Lelaps is a fast detector simulation that swims particles through detectors with magnetic fields, accounting for multiple scattering and energy loss.
- It produces parameterized showers in EM and hadronic calorimeters.
- It converts gammas.
- It supports decays of certain short-lived particles ("V" decays).
- It does all this very fast.

- Lelaps consists of a set of C++ class libraries and a main program, which itself is called lelaps.
- The main class library is called CEPack, the actual simulation toolkit.
- Built-in support for LDMar01, SDJan03 and SDMar04.
- Also reads detector geometries in GODL format.
- Reads StdHep (uses IStdHep class) generator files and produces SIO or LCIO output files.
Most objects in the LD are cylinders.

There are some conical masks (not drawn).

TPC and barrel muon detector each contain a CENode with a set of concentric cylinder surfaces.

Muon endcaps use a CENode with a set of cylindrical slices.

Calorimeters use subid calculation for their segmentation.
Materials in CEPack

- All elements built in with default pressure/temperature/density.
- Any compound can be specified by chemical formula and density or (for gasses) temperature and pressure.
- Mixtures can be created by mixing elements and compounds (by volume or by weight).
- All needed quantities are calculated automatically
  - Constants needed for multiple scattering and energy loss
  - Radiation lengths (Tsai, PDG)
  - Interaction lengths (from a fit to element data)
  - Other constants needed for shower parametrization
Matprop

- Lelaps distribution comes with a little program called matprop:
  - `matprop U` # for elements (gasses at NTP)
  - `matprop SiO2/2.32` # for solids (and liquids) specify density (g/cc)
  - `matprop CO2/1/298.15` # for gasses specify pressure (atm) and temperature (K)
  - `matprop O2/STP` # for gasses at STP (0 C, 1 atm)
  - `matprop Ar/NTP` # for gasses at NTP (25 C, 1 atm)
  - `matprop H2//` # for gasses at NTP

- For a mixture (e.g. Air at 20 C) use (-g to indicate gas):
  - `matprop -g O2/1/293.15 20.946 N2/1/293.15 78.084 Ar/1/293.15 0.934`

- Just type matprop to get a list of options
- Prints out lots of material properties
- Matprop is available online: [http://www.slac.stanford.edu/comp/physics/matprop.html](http://www.slac.stanford.edu/comp/physics/matprop.html)
Multiple Scattering and $dE/dx$

- Multiple scattering is performed using the algorithm of Lynch and Dahl.
- Material is “saved up” along the track until there is enough.

- $dE/dx$ is calculated using the methods by Sternheimer and Peierls.

- All constants precalculated by the material classes.
Shower Parameterization

- Electromagnetic showers are parameterized using the algorithms of Grindhammer and Peters.
    (Paper is a 1993 conference contribution, submitted by request to the archive in 2000).

CEPack simulation of BaBar EM calorimeter in Moose (courtesy of Dominique Mangeol).
See also the BaBar web site (computing – simulation – fast simulation).
Shower Parameterization

- Hadronic showers are parameterized using code that is similar to the code for electromagnetic showers, with some modifications.
- Parameterized shower simulation was compared to Geant4.
- In general pretty good agreement for EM showers. Hadronic showers agree pretty well longitudinally, but not as well radially.
  - Hadronic shower parameterization has been tweaked since then.
EM Shower Parameterization and Geant4

Comparison of CEPack longitudinal profile (green) of a 10 GeV electron in an EM calorimeter with Geant4 (orange).
Decays and Gamma Conversions

- Supported unstable particles are $\pi^0$, $K^0$-short ($K^0$-long treated as stable), Lambda, Sigma$^+/-/0$, $\Xi^-, 0$ and Omega$^-$. Only decay modes $> 2\%$ supported ("V" decays)
Decays

Wired picture of the decay chain:

\[ \Omega^- \rightarrow \Xi^0 \pi^- \]
\[ \Xi^0 \rightarrow \Lambda \pi^0 \]
\[ \Lambda \rightarrow p \pi^- \]
\[ \pi^0 \rightarrow \gamma \gamma \]

as simulated by Lelaps for the LCD LD model.
Conversions

Wired picture of a gamma conversion as simulated by Lelaps for the LCD LD model.
Lelaps (LDMar01)
Lelaps (SDJan03)
Lelaps: Usage

- Most common usage:
  - `lelaps -o foo.sio -E bar1.stdhep bar2.stdhep ...`
  - Reads one or more StdHep files and produces SIO output file.
  - Defaults to SDJan03. Use `-L LDMar04` for LD, etc.
- Also has built-in simple particle “gun”:
  - `lelaps -o foo.slcio -i 11 -m 10 -n 4 -N 1000`
  - Generates 1000 events with 4 electrons/event with 10 GeV each random in $4\pi$ and produces LCIO output file.
- Options for generation in x,y plane ($2\pi$) or along x, y, z directions
- Options for turning off energy loss, multiple scattering and/or showering in calorimeters
- Options for turning off decays and/or conversions and tracking their secondaries
Performance

- Tracking alone: 3 - 4 “typical” events/s (at 1 GHz) for LD, 2 events/s for SD.
- Adding parameterized showering costs 15% (SD) to 30% (LD).
- Adding decays and conversions adds 20%.
- LCIO output file (14 MB, compressed) adds another 40%.
  - SIO output takes much longer (factor 2 to 4 depending heavily on calorimeter segmentation). Could be optimized.
- Using a GODL detector description file adds some 20-30% because of ID calculation.
GODL – General Object Description Language

Langugae Features - Variables and Arrays

- GODL is an extensible typeless programming language.
  - Type determined by assignment:
    
    ```
    a = 2.4;      # real
    b = 2;        # integer
    c = "text";   # string
    d = true;     # boolean
    ```

- It has variables and operations that can be performed on them:
  
  ```
  a += 1;
  b = a * 12;
  d = c + " more text";
  e = false;
  b = e != true;
  ```

- Array-like constructs:
  
  ```
  i = 5; foo.i = 12;  # Same as foo.5 = 12;
  ```
GODL – Language Features
Operators

- Set of operators (some cannot be used in some contexts):
  - `+ - * / = += -= *= /= == < > <= >= != ! && ||`

- Reference operator `@`
  - `a = 12;`  
  - `b = @a; print(b, "\n");`  
  - `@a->(12)`

Useful for referencing objects multiple times without recreating them.
GODL – Language Features
Built-in Functions

- It knows about the usual set of math functions:
  - exp, log, sqrt, pow, cos, sin, tan, acos, asin, atan, atan2, cosh, sinh, tanh, acosh, asinh, atanh, log10, abs, fabs, ceil, floor, mod

- In addition:
  - list
    ```
    a = list(a, b, c, d);  # Creates unnamed list
    ```
  - print
    ```
    print(a, "\n");
    print(a, "\n", b, "\n");
    ```
  - argc, argv
    - When arguments are provided
  - unit
    - See later
GODL – Language Features
Control Constructs

- It has a limited set of control constructs:
  - C-style for and while loops:
    ```
    for (i = 0; i < 25; i += 1) {
        ...
    }
    while (true) {
        ...
        if (something) break;
    }
    ```
  - C-style if statements (no “else” yet)
    ```
    if (a < b) {
        ...
    }
    ```
GODL – Language Features
List Objects

- Variables can be list objects:
  
  ```
  a = foo(a, b, c, d);
  ```

- Lists can contain objects of any type, including other lists.

- To add objects to a list:
  
  ```
  a += e;
  a += f;
  ```

- Note that this is not necessarily the same as:
  
  ```
  a += e + f;
  ```

  which would first add f to e and then the result to a. If e and f are list objects, this adds to “a” a single list “e” which in turn contains “f”.

GODL – Language Features

Units

- Variables can have units, and units are enforced across operations and in arguments to functions and list objects:
  ```
  m = _meter;     # _meter is a built-in unit
  unit("m");     # Declare as unit
  a = 2 m;
  b = 12 * a;
  area = a * b;
  area += 5;      # Error: incorrect units
  d = cos(area);  # Error: cos() only takes angles
  ```
GODL – Language Features

Units

- Available units (like CLHEP):
  - Basic units: _meter, _second, _joule, _coulomb, _kelvin, _mole, _candela, _radian, _steradian
  - Derived units: _angstrom, _parsec, _barn, _degree, _hertz, _becquerel, _curie, _electronvolt, _gram, _watt, _newton, _pascal, _bar, _atmosphere, _ampere, _volt, _ohm, _farad, _weber, _tesla, _gauss, _henry, _gray, _lumen, _lux

- Create new units:
  
m = _meter; g = _gram; # For convenience
  unit("m", "g") # Declare as units
  gcc = g/cm3; # New unit of density
  unit("gcc"); # Declare

- Automatically converts SI prefixes and powers:
  a = 1 cm2; # = 0.0001 _meter squared
GODL – Language Features
Miscellaneous

- Built-in constants:
  - \_pi (3.14...) has units of rad
  - \_e\_SI (electron charge 1.6\ldots10^{-19} \text{ C}), \_e (2.71...) dimensionless

- Debugging functions:
  - \texttt{verbose}: prints a lot of debugging information to stdout
  - \texttt{__printvars}: prints a list of all variables to stdout

- Control variables for \texttt{print()} function:
  - \texttt{printlevel\_:} (default 1) controls how much information to print
    (mostly for for object lists).
  - \texttt{precision\_:} controls how many digits are displayed for floating point numbers.
  - \texttt{fieldwidth\_:} controls how much space a printed number takes up.
GODL – Built-in List Objects

Materials

- Materials are declared using the element, material or mixture list objects (use the @ operator to pass by reference):
  - Si = element("Si");
  - vacuum = material("vacuum");
  - O2 = material(formula("O2"),
    pressure(1.0 atm),
    temperature(293.15 K));
  - Tyvek = material(name("Tyvek"),
    formula("CH2CH2"),
    density(0.935 g/cm3));
  - Air = mixture(part(@O2, 20.946),
    part(@N2, 78.084),
    part(@Ar, 0.934),
    by("volume");)
GODL – Built-in List Objects
Volumes and Placements

- First define a World Volume:
  \[
  \text{World} = \text{cylinder}(\text{radius}(700.0 \, \text{cm}), \text{length}(14.0 \, \text{m}), @\text{vacuum});
  \]

- Define another volume:
  \[
  \begin{align*}
  \text{em\_ec\_irad} & = 21.0 \, \text{cm}; \\
  \text{em\_ec\_orad} & = 125.0 \, \text{cm}; \\
  \text{em\_b\_irad} & = \text{em\_ec\_orad} + 2.0 \, \text{cm}; \\
  \text{em\_b\_orad} & = \text{em\_b\_irad} + \text{em\_thickness}; \\
  \text{em\_sampfrac} & = 0.02664; \\
  \text{em\_nlayers} & = 30; \\
  \text{em\_b\_length} & = 368.0 \, \text{cm};
  \end{align*}
  \]

  \[
  \text{EM\_Barrel} = \text{cylinder}(\text{name}("\text{EM\_Barrel}")), \\
  \quad \text{innerRadius}(\text{em\_b\_irad}), \\
  \quad \text{outerRadius}(\text{em\_b\_orad}), \\
  \quad \text{length}(\text{em\_b\_length}), @\text{SiW}, \\
  \quad \text{type}("\text{emcal}")), \text{nLayers}(\text{em\_nlayers}), \\
  \quad \text{samplingFraction}(\text{em\_sampfrac}));
  \]

- Add to World using placement:
  \[
  \text{World} += \text{placement}(@\text{EM\_Barrel});
  \]
GODL – Built-in List Objects
Volumes and Placements

- Use loops to do repetitive tasks and if statements for conditionals:

```plaintext
Vertex_Barrel = cylinder(name("Vertex Barrel"),
    innerRadius(v_irad), outerRadius(v_orad),
    length(v_lenmax));

for (i = 1; i <= v_nlayers; i += 1) {
    vlen = v_leninner;
    if (i > 1) vlen = v_lenmax;
    Vertex_Barrel.i = cylinder(name("Vertex Barrel " + i),
        innerRadius(v_spacing * i),
        outerRadius(v_spacing * i + v_thickness),
        length(vlen), @Si,
        type("tracker"));
    Vertex_Barrel += placement(@Vertex_Barrel.i); # Notice hierarchy
}

World += placement(@Vertex_Barrel);
```
GODL – Built-in List Objects
Levels of Detail

- Specify levels of detail with “level” tag:

```
Had_Endcap = cylinder(name("Had Endcap"), level(1),
                  innerRadius(had_ec_irad),
                  outerRadius(had_ec_orad),
                  length(had_thickness), @StainlessPoly,
                  type("hadcal"), nslices(had_nlayers),
                  samplingFraction(had_sampfrac));
```

```
Had_Endcap += placement(@something, ..., level(max(0)), ...);
Had_Endcap += placement(@something_else, ..., level(min(1)), ...);
```

```
World += placement(@Had_Endcap, translate(0, 0, 0.5 * (had_b_length -
                  had_thickness)));
World += placement(@Had_Endcap, rotate(axis("y"), angle(180 degrees)),
                  translate(0, 0, -0.5 * (had_b_length - had_thickness)));
```
### GODL – Built-in List Objects
#### Levels of Detail

<table>
<thead>
<tr>
<th>Level syntax:</th>
<th>Create object when:</th>
<th>Used for:</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;not specified&gt;</td>
<td>always</td>
<td>Fundamental objects that are always present</td>
</tr>
<tr>
<td>level(min(2))</td>
<td>level &gt;= 2</td>
<td>Detailed objects that should not be simulated at lower levels</td>
</tr>
<tr>
<td>level(max(4))</td>
<td>level &lt;= 4</td>
<td>Fundamental objects that are replaced with other objects at higher levels</td>
</tr>
<tr>
<td>level(min(2), max(4))</td>
<td>2 &lt;= level &lt;= 4</td>
<td>Combinations: objects relevant only in a certain level range</td>
</tr>
<tr>
<td>level(range(2, 4))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>level(mask(0x1C))</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
ID calculation (such as CalorimeterID and TrackerID) is generally used for two purposes:

- To specify segmentation of a detector: hits with the same ID are combined to a single energy deposition.
- To specify an abbreviated version of the location of an energy deposition.

Problem: how can we change the amount or method of segmentation without changing the C++ source code of the simulator?

One solution: Specify the segmentation method in the geometry file and “interpret” it inside the simulator.

GODL API provides a simple, fast, interpreter to do that.

- In fact, it “compiles” the segmentation specification into byte-code, and runs the byte code for each hit.
GODL – Built-in List Objects
ID Calculation

- Example: Tracker ID. For the Vertex detector barrel we would use:

```plaintext
Vertex_Barrel.i = cylinder(name("VXD"),
    innerRadius(1.2 cm * i),
    outerRadius(1.2 cm * i + 0.01 cm),
    length(vlen), @Si,
    type("tracker"),
    idCode(code(tb_code),
        data("system", 1),
        data("id", i - 1)))
```

- The ID calculation in idCode is specified as a string in a “code” list object. The algorithm for the vertex and barrel trackers is:

```plaintext
tb_code = "x: fp0 y: fp1 z: fp2 layer: d3 id z H 0x40000000 mul or system 28 bitshift or stop"
```
GODL – Built-in List Objects
ID Calculation

- For the tracker end cap it is:
  "x: fp0 y: fp1 z: fp2 layer: d3 id 0x80000000 or z H
  0x40000000 mul or system 28 bitshift or stop";

- The “x: fp0” part means that the API routine that evaluates the byte code associated with the above expects x to be given in the first floating point “register”. Similarly, “layer” is provided as an integer in the fourth register.

- Reverse polish PostScript-like language with some limitations and some extras:
  - Some named variables must be provided by the simulator as standard arguments (x, y, z, layer)
  - Some named variables are provided using “data” object lists in the specification.
  - H is the Heaviside step function: 1 if argument positive, 0 otherwise.
GODL – Built-in List Objects
ID Calculation

- Slightly more work for the calorimeters. For the end caps we have:
  \[\text{cal\_code\_ec} = "x\ x\ \text{mul}\ y\ y\ \text{mul}\ \text{add}\ \sqrt{z}\ \text{atan2}\ \theta\ \text{seg}\ \text{mul}\ \pi\ \text{div}\ "\ +\ \text{standard\_code};\]

- For the barrel we have:
  \[\text{cal\_code\_b} = "x\ x\ \text{mul}\ y\ y\ \text{mul}\ \text{add}\ \sqrt{z}\ \text{atan2}\ \cos\ 1.0\ \text{add}\ \theta\ \text{seg}\ \text{mul}\ 2.0\ \text{div}\ "\ +\ \text{standard\_code};\]

- where standard\_code is:
  \[\text{standard\_code} = "\text{truncate\ 11\ bitshift}\ y\ x\ \text{atan2}\ m\ \phi\ \text{seg}\ \text{mul}\ 0.5\ \text{mul}\ _\pi\ \text{div}\ \text{truncate}\ \text{or}\ \text{system\ 21}\ \text{bitshift}\ \text{or}\ \text{system\ 28}\ \text{bitshift}\ \text{or}\ \text{stop}";\]

- We have to add standard argument specifications to this:
  \[\text{cal\_code\_ec} = "x:\ \text{fp0}\ y:\ \text{fp1}\ z:\ \text{fp2}\ \text{layer}:\ d3\ "{+}\ \text{cal\_code\_ec};\]
The GODL API consists of four classes: GODLParser, MCode, MStack and MVar.

There are (currently) 11 virtual functions that the API implementer must write. Example:

```cpp
virtual int constructCylinder(
    const char *nameForFutureReference,
    const char *objectName,
    double innerRadius, // length units: meter
    double outerRadius,
    double length,
    const char *materialRefName,
    const char *type,
    int nLayers,
    int nSlices,
    double samplingFraction,
    const MStack &IDCode)
```
GODL API

- Other functions:
  - `constructCone(...);`
  - `addField(...);`
  - `addPlacement(...);`
  - `constructPlacement(...);`
  - `rotate(...);`
  - `translate(...);`
  - `constructElement(...);`
  - `constructCompound(...);`
  - `constructMixture(...);`
  - `addMixture(...);`

- API reads `.godl` file and calls “construct” routines to construct objects and placements. It then calls rotate and translate on the placements and addMixtures to add materials to the mixtures. Finally it calls addPlacement to instantiate an actual placement of an object.
GODL - Status

- Parser/evaluator is essentially complete.
  - API layer to access the volume list exists.
  - Completely implemented in Lelaps V03-23-26.
  - Includes levels of detail and ID calculation.
  - SDMar04.godl file exists.
  - SDJan03.godl with two different levels of detail exists.
Future

- Lelaps and CEPack interfaces are not yet frozen!
- New features planned for CEPack
  - Combinatorial geometry
  - Shower continuation into next volume
- New features planned for GODL:
  - Add a number of standard geometrical shapes
  - Add support for combinatorial geometry
- Old features need to be tested more thoroughly
  - More tuning of hadronic showers
About the name Lelaps

Lelaps ("storm wind") was a dog with such speed that, once set upon a chase, he could not fail to catch his prey. Having forged him from bronze, Hephaestus gave him to Zeus, who in turn gave him to Athena, the goddess of the hunt. Athena gave Lelaps as a wedding present to Procris, daughter of Thespius, and the new bride of famous hunter Cephalus.

A time came when a fox created havoc for the shepherds in Thebes. The fox had the divine property that its speed was so great that it could not be caught. Procris sent Lelaps to catch the fox. But because both were divine creatures, a stalemate ensued, upon which Zeus turned both into stone. Feeling remorse, Zeus elevated Lelaps to the skies, where he now shines as the constellation Canis Major, with Sirius as the main star.
Introduction: Lelaps

…but clearly, Lelaps (the program) is not a dog!
The End