A toolkit for data-driven computing in Geant4

Presented by Marcus H. Mendenhall
Vanderbilt University

Why?

- There is a need for a unified package to allow management of tabulated datasets and smooth functions.
- Many physics issues depend intrinsically on derivatives of functions (E&M field solutions, e.g.).
- Fast, efficient integration and root-finding are useful tools to have universally available (solving Coulomb collisions, e.g.).
- Many instances arise in complex packages such as Geant4 where multiple entities need to share ownership of an object.

What?

- The package I will describe provides a unified platform which integrates:
  - Functions which provide their value and two derivatives
  - Reference-counted objects! No memory leaks.
  - Creation of cubic-spline interpolating functions in various transformed spaces (linear, log-log, etc.) with derivatives
  - Constructing functions of functions inline (binaries, composition, inverse functions, etc.)
  - Function metadata such as domain and information about 'interesting' points in the form of a sampling grid
  - Numerical algorithms which are highly efficient on functions with derivatives (adaptive integration, root finding)
  - Tools to adaptively convert expensive procedural functions into efficient splines to any specified accuracy

Example: using derivatives

```
// load an std::vector<G4double> zvals with values of z
// and Bzvals with the z component of the field at those points
// and create the interpolating_function from them.
// This is typically done once at instantiation of a class
static c2_factory<G4double> c2;
typedef c2_ptr<G4double> c2_p;
c2_p Bzfunc=c2.interpolating_function(zvals, Bzvals);

// now, the part below is usually done in a loop
// where the field is to be evaluated many times.

G4double bz, bzprime, bzprime2;
// compute fields at x, y, z near the axis

// get Bz on-axis and its derivatives
bz=Bzfunc(z, &bzprime, &bzprime2);

// x & y magnetic field from divergence equation
G4double bx = -x*bzprime/2, by=-y*bzprime/2;

// correct bz off-axis to maintain zero curl.
bz -= (x*x+y*y)*bzprime2/4.0;
```

Sampling Functions

- Problem:
  - Some function is very expensive to compute, and needs to be used repeatedly.
- Solution:
  - Generate an interpolating function representation of it
- Issues:
  - In general, one must know quite a bit about the function in advance to know how to sample it to provide bounded errors
  - This makes it hard to do without custom code
- Complete solution:
  - Use an error-controlling adaptive sampler
  - Allow function to carry some metadata about its structure

Example: sampling a smooth function

```
// enter with func being the c2_function to be plotted,
// and its domain set to the desired plotting range
static c2_factory<G4double> c2;
typedef c2_ptr<G4double> c2p;

// make sure our access to func is managed,
// if passed in from the outside
// This is always a safe thing to do.
c2p funcp=func;
// create an efficient representation,
// using all derivative information
c2p sample1=func.adaptively_sample(
  func.xmin(), func.xmax(), 1e-6, 1e-6);
// sample1 can be evaluated quickly and efficiently.
```


Example: sampling a classic ‘c’ function

```
// enter with func being the conventional function
static c2_factory<G4double> c2;
typedef c2_ptr<G4double> c2p;

// embed the normal ‘c’ function func in a c2_function
c2p classic=c2.classic_function(func);

// create an efficient representation,
// using no derivative information
c2p sample1=c2.interpolating_function().sample_function
  (classic,xmin, xmax, 1e-4, 1e-4);
// sample1 can be evaluated quickly and efficiently
// between xmin & xmax to 1e-4 absolute & relative error.
// If the function is nicer in log-log space, do:
c2p sample1=c2.log_log_interpolating_function().
  sample_function(classic,xmin, xmax, 1e-4, 0);
// Note that only relative errors make sense for log
// but the absolute error of the log is the relative error
// of the function
```


Example: Using the root finder

```
c2_factory<G4double> c2;
typedef c2_ptr<G4double> G4_c2_ptr;

class G4ScreenedCoulombClassicalKinematics: public
    G4ScreenedCoulombCrossSectionInfo,
    public G4ScreenedCollisionStage {
public:
   G4ScreenedCoulombClassicalKinematics();
protected:
   // the c2_functions we need to do the work.
   c2_const_plugin_function_p<G4double> &phifunc;
   c2_linear_p<G4double> &xovereps;
   G4_c2_ptr diff;
};

G4ScreenedCoulombClassicalKinematics::G4ScreenedCoulombClassicalKinematics() :
    phifunc(c2.const_plugin_function()),
    xovereps(c2.linear(0., 0., 0.)),
    diff(c2.quadratic(0., 0., 0., 1.)-xovereps*phifunc)
{
}

xovereps.reset(0., 0.0, au/eps); // slope of x*au/eps term
phifunc.set_function(&(screen->EMphiData.get())); // install interpolating table
G4double xx1, phip, phip2;
G4int root_error;
xx1=diff->find_root(phifunc.xmin(), std::min(10*xx0*au,phifunc.xmax()),
   std::min(xx0*au, phifunc.xmax()), beta*beta*au*au, &root_error, &phip, &phip2)/au;
```

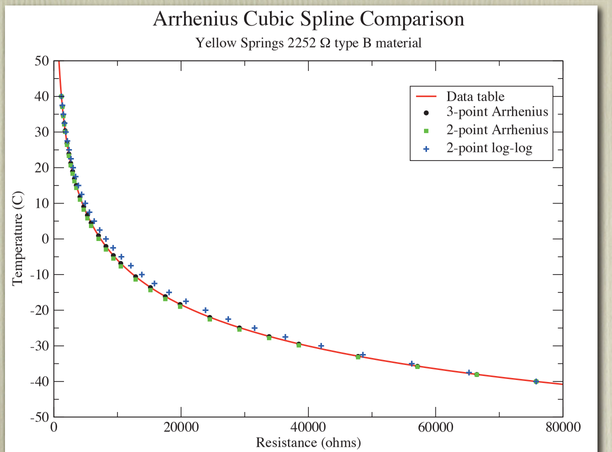
Precision interpolation

- Correct choice of interpolating method strongly affects how much information is needed to construct the interpolator
- choosing a coordinate system in which the function is approximately cubic (or simpler) allows cubic splines to have very sparse sets of knots
- c2_functions provide a rich choice of built in interpolators, and is easily extensible
- interpolators can be adaptively populated from a function to meet specified error tolerance

Example: Transformed Splines

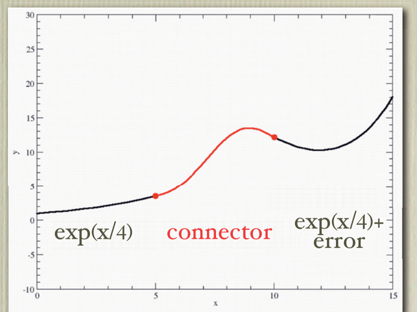
```
from c2_function import *

t0=273.15

a=arrhenius_interpolating_func().load(
  [-40+t0,40+t0],[75790, 1200], True,0,True,0)

#a=c2_function.arrhenius_interpolating_func().load(
# [-40+t0,t0, 40+t0],[75790, 7355, 1200], True,0,True,0)

# in c++, this would be: c2p b=c2.inverse_function(a)-c2.constant(t0);
b=c2inverse_function(a)-c2constant(t0)
r0 = b.xmin()
r1 = b.xmax()

xv=vectorDouble()
yv=vectorDouble()
b.adaptively_sample(r0,r1,1e-1,0,1,xv,yv)

out=open("therm.dat","w")
for x,y in zip(xv,yv):
    print >> out, x, y
out.close()
```


Comparison of splines



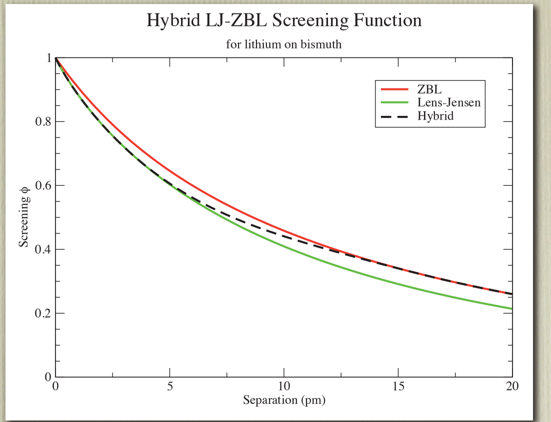
Joining Functions Smoothly

- Take any two c2_functions and smoothly connect automatically
- Often necessary to connect an approximate function to an asymptotic form
- Also useful for filling in regions of functions which are difficult to compute numerically (0/0, e.g.)




```
c2p conn=c2.connector_function(left_func.xmax(),
   left_func, right_func.xmin(), right_func, auto_center, center_val);
c2_piecewise_function<double> &piece=c2.piecewise_function();
c2p keep_piece(piece);
piece.append_function(left_func);
piece.append_function(conn);
piece.append_function(right_func);
```

Spliced screening functions



Marcus H. Mendenhall, February 13-15, 2008 VANDERBILT UNIVERSITY

Use of connector to splice theories

```
G4_c2_function &LJZBLScreening(G4int z1, G4int z2,
  size_t npoints, G4double rMax, G4double *auval){
// hybrid of LJ and ZBL, uses LJ if x < 0.5*auniv,
// ZBL if x > 1.5*auniv, and
// is very near the point where the functions naturally cross.
    G4double auzbl, aulj;
    c2p zbl=ZBLScreening(z1, z2, npoints, rMax, &auzbl);
    c2p lj=LJScreening(z1, z2, npoints, rMax, &aulj);

    G4double au=(auzbl+aulj)*0.5;
    lj->set_domain(lj->xmin(), 0.5*au);
    zbl->set_domain(1.5*au,zbl->xmax());

    c2p conn=c2.connector_function(lj->xmax(), lj,
      zbl->xmin(), zbl, true,0);
    c2_piecewise_function_p<G4double> &pw=c2.piecewise_function();
    c2p keepit(pw);
    pw.append_function(lj);
    pw.append_function(conn);
    pw.append_function(zbl);
    *auval=au;
    keepit.release_for_return();
    return pw;
}
```

Marcus H. Mendenhall, February 13-15, 2008 VANDERBILT UNIVERSITY

Example: complete program

```
#include "c2_factory.hh"
typedef c2_ptr<double> c2p;
static c2_factory<double> c2;
int main() {
    c2p a=c2.sin()+c2.linear(0,0,1.1);
    a->set_domain(0,10);
    c2_inverse_function_p<double> &b=c2.inverse_function(a);
    c2p keep_b(b);
    c2p hint;
    for(size_t loop=0; loop < 3; loop++) {
        switch(loop) {
            case 0:
                printf("\ninversion with no hinting\n");
                hint.unset_function();
                break;
            case 1:
                printf("\ninversion with rough hinting\n");
                hint=c2.interpolating_function().sample_function(
                     b, b.xmin(), b.xmax(), .1, 0.0, true, 0, true, 0);
                break;
            case 2:
                printf("\ninversion with detailed hinting\n");
                hint=c2.interpolating_function().sample_function(
                     b, b.xmin(), b.xmax(), .001, 0.0, true, 0, true, 0);
                break;
            default:
                break;
        }
        b.set_hinting_function(hint);
        const size_t ntest=10;
        const double testvals[ntest]={ 0, 0.01, 5, 5.0001, 5.001, 5.01, 5.1, 6, 10, 0.01 };
        for(size_t i=0; i<ntest; i++) {
            double xx=testvals[i];
            a->reset_evaluations();
            double yy=b(xx);
            printf("%8.4f %16.12f %4d \n", xx, yy, a->get_evaluations());
        }
    }
    return 0;
}
```

Marcus H. Mendenhall, February 13-15, 2008 VANDERBILT UNIVERSITY