CorAL
The Correlations Analysis Library *

November 7, 2007

David Brown and Mike Heffner
Lawrence Livermore National Laboratory
brown170@llnl.gov, mheffner@llnl.gov

Scott Pratt and Li Yang
Dept. of Physics & Astronomy, Michigan State University
prattsc@msu.edu

Contents

*Supported by the U.S. Department of Energy, Grant No. DE-FG02-03ER41259.

1 Introduction

CorAL is a code base for analysis of 2-particle correlations at small relative momentum. It includes routines for reading, storing and manipulating three-dimensional correlation functions and source functions. Three-dimensional data can be stored in a Cartesian mesh or in terms of expansion coefficients using either Cartesian Harmonics or spherical harmonics as a basis. Routines are provided to translate between different three-dimensional realizations. CorALroutines provide both the means to calculate correlation functions from source functions, and to perform the inverse, by either fitting to a parameterized form, or to perform imaging.

Classes for wave functions and the associated kernels in CorAL are provided for the following pairs: $\pi^+\pi^-$, $\pi^+\pi^-$, pp, nn, pn, pK^+, $K^+\pi^+$, $K^+\pi^-$, $p\Lambda$, $\Xi^+\pi^-$ and $\Lambda\Lambda$.

CorAL is designed to exploit the simple equation linking three-dimensional source functions to three-dimensional correlation functions:

$$R(\mathbf{q}) \equiv C(\mathbf{q}) - 1 = \int d^3r \left\{|\phi(\mathbf{q},\mathbf{r})|^2 - 1\right\} S(\mathbf{r}), \tag{1}$$

by providing the obtain numerical arrays describing $C(\mathbf{q})$ from numerical representations of $S(\mathbf{r})$ for a variety of relative wave functions. A more detailed description of theory can be found in [1], and is included in the `doc/papers/` directory. Eq. (1) can be re-expressed with Correlation and source functions represented by expansion coefficients in either the spherical-harmonic or Cartesian-harmonic basis. In these bases, an angular function $F(\Omega)$ can be reproduced from coefficients via the expressions:

$$F(\Omega) = \sum_{\ell_x,\ell_y,\ell_z} \frac{(\ell_x+\ell_y+\ell_z)!}{\ell_x!\ell_y!\ell_z!} F_{\ell_x,\ell_y,\ell_z} n_x^{\ell_x} n_y^{\ell_y} n_z^{\ell_z} \tag{2}$$

$$F(\Omega) = (4\pi)^{1/2} \sum_{\ell,m} F_{\ell,m} Y_{\ell,m}(\Omega), \tag{3}$$

where $F_{\vec{\ell}}$ and $F_{\ell,m}$ are defined

$$F_{\vec{\ell}} = \frac{(2\ell+1)!}{\ell!} \int \frac{d\Omega}{4\pi} F(\Omega) \mathcal{A}_{\vec{\ell}}(\Omega), \tag{4}$$

$$F_{\ell,m} = (4\pi)^{-1/2} \int d\Omega F(\Omega) Y_{\ell,m}(\Omega). \tag{5}$$

Here $\mathcal{A}_{\vec{\ell}=\ell_x,\ell_y,\ell_z}$ and $Y_{\ell,m}(\Omega)$ are angular functions and are referred to as Cartesian and Spherical Harmonics. Forms and properties of Cartesian Harmonics can be found in the `doc` directory. Forms and properties for Spherical Harmonics can be found in standard texts, such as J.D. Jackson, *Classical Electrodynamics*.

The expansion coefficients for the correlation and source functions are related on a one-to-one basis.

$$\mathcal{R}_{\ell,m}(q) = \int 4\pi r^2 dr\ \mathcal{K}_\ell(q,r)\ \mathcal{S}_{\ell,m}(r), \tag{6}$$

$$\mathcal{R}_{\vec{\ell}}(q) = \int 4\pi r^2 dr\ \mathcal{K}_\ell(q,r)\ \mathcal{S}_{\vec{\ell}}(r). \tag{7}$$

For equations the kernel K_ℓ plays the role of the wave function in Eq. (1) linking the correlation and source functions:

$$\mathcal{K}_\ell(q,r) \equiv \frac{1}{2} \int d\cos\theta_{qr} \left[|\phi(q,r,\cos\theta_{qr})|^2 - 1\right] P_\ell(\cos\theta_{qr}). \tag{8}$$

The functions of CorAL are to :

- Derive source functions from output of theoretical models or parameterizations. Source functions would then be stored numerically in arrays. The arrays could either be three dimensional cartesian arrays or arrays of expansion coefficients for either basis mentioned above.
- Calculate wave functions and kernels for a variety of interaction types. Kernels could then be discretized and stored.

- Provide simple convolutions to generate correlation functions from source functions.
- Perform deconvolutions to generate source functions from experimentally determined source functions. This will done either through parameter fitting or through *imaging* which involves fitting to source functions whose radial dependences are defined by splines or other very general forms.

2 Directory Structure

The base directory of CorAL has four directories:
`doc/ include/ lib/ src/ examples/ codetests/`
The `doc` directory includes this document, plus some related papers along with a sub-directory `printAY` which has codes used to generate printouts showing the relationship between $Y_{\ell,m}$s and Cartesian harmonics.

The `include/` directory will store copies of all include files required by CorAL. The `lib/` directory contains the makefile used to compile the project and will store the static library `libcoral.a`.

The `src/` directory stores all source and header files in several sub-directories:
Amongst the header files are two catch-all files, `coral.h` and `Utilities/utilities.h` which allow the programmer to include all CorAL files with a simple command `#include "coral.h"`. The header file `utilities.h` is automatically included by `coral.h`, and is useful if the user wishes to use the code for other purposes than analyzing correlation functions.

The routines for generating `CWaveFunction` objects are in `src/WaveFunctions/`, code related to kernels are in `src/Source2CF/` and `src/SourceCalc/` contains code used for filling arrays with information about source functions. Imaging and fitting routines are stored in ?????????????????????????????????? and `src/Utilities` holds a panoply of useful codes, which can be applied to projects other than correlations.

Example codes are kept in `examples/`. These codes are meant to provide examples which can be useful in getting started with CorAL. The directory `codetests` also has some examples which can be used to test whether CorALis operating correctly.

3 Data Arrays

Data are stored in several classes of arrays, including three-dimensional Cartesian meshes, Spherical Harmonic expansions, Cartesian Harmonic expansions and in the future, basis splines. In the next section we will discuss a namespace for operations involving multiple arrays, such as those used to translate into different array types, or multiply and divide arrays.

3.1 Cartesian Meshes

Cartesian meshes are described by a three dimensional array of size `NXMAX, NYMAX, NZMAX` and granularities `DELX, DELY, DELZ`. The mesh is further defined by three parameters `XSYM, YSYM, ZSYM` which are booleans set to `true` for the corresponding reflection symmetries. When set to `false` the program assigns additional memory to describe data for negative values of `x, y, z`. The ℓ_x, ℓ_y, ℓ_z components refer to a range $\ell_x\Delta x < x < (\ell_x+1)\Delta x$, $\ell_y\Delta y < y < (\ell_y+1)\Delta y \cdots$.

The capabilities of the class are best explained by viewing the public members of the class:

```
class C3DArray{
 public:
  C3DArray(char *arrayparsfilename);
  C3DArray(int NXYZMAX,double DELXYZ,bool XSYM,bool YSYM,bool ZSYM);
```

```
  C3DArray(int NXMAX,double DELX,int NYMAX,double DELY,int NZMAX,double DELZ,
   bool XSYM,bool YSYM,bool ZSYM);
  ~C3DArray();

  int GetNXMAX();
  int GetNYMAX();
  int GetNZMAX();
  double GetDELX();
  double GetDELY();
  double GetDELZ();
  bool GetXSYM();
  bool GetYSYM();
  bool GetZSYM();
  void PrintPars();


  double GetElement(double x,double y,double z);
  double GetElement_NoInterpolation(double x,double y,double z);
  double GetElement(int isx,int ix,int isy,int iy,int isz,int iz);
  void SetElement(int isx,int ix,int isy,int iy,int isz,int iz,double value);
  void IncrementElement(int isx,int ix,int isy,int iy,int isz,int iz,
double value);
  void SetElement(double x,double y,double z,double value);
  void IncrementElement(double x,double y,double z,double increment);

  void ZeroArray();
  void MakeConstant(double c);
  void ScaleArray(double scalefactor);

  void Randomize(double c);
  void RandomizeGaussian(double c);

  void PrintArray();
  double GetBiggest();

  void ReadArray(char *dirname);
  void WriteArray(char *dirname);
};
```

Three constructors are used to initialize the arrays and set the parameters: `XSYM`, `YSYM`, `ZSYM`, `NXMAX`, `NYMAX`, `NZMAX`, `DELX`, `DELY`, `DELZ`, which can also be set by reading a parameters file. The parameters can be printed by calling `PrintPars()` and can be accessed via the function `GetXSYM()`,$\cdots$. If t`C3DArray(char *arrayparsfilename)` is used, the parameters are read from a file formatted in the form:

```
bool YSYM 1
int NXMAX 36
double DELZ 2.5
⋮
```

If any of the parameters are missing, default values will be used. The boolean parameter `IDENTICAL` can also be set, which is identical to setting the three symmetry parameters to `true`. For the reflection-symmetry parameters, `XSYM, YSYM and ZSYM` are set to `false` by default.

The functions that set, get and increment elements should probably not need much explanation. They are over-loaded so that one can access the values either by the indices directly, or through the coordinates. If values are accessed outside the range, e.g., x$>\Delta$x$\cdot$`NXMAX`, zero is returned. The arguments `isx,isy,isz` which appear in several functions refer to the sign of the argument. For $x > 0$ `isx=0`, while for $x < 0$, `isx=1`. When `XSYM='true'` memory is not allocated for `isx=1`.

`ZeroArray()` will set all the elements to zero and `MakeConstant(c)` sets all elements to `c`. The entire array

can be scaled by a constant with `ScaleArray`. Every element can be randomized according to a Gaussian distribution, $\exp -x^2/2c^2$, with `RandomizeGaussian(c)` or to a uniform distribution between $-c$ and c with `Randomize(c)`.

The element with the largest absolute value is returned with `GetBiggest()`.

The functions `WriteArray(char *dirname)` and `ReadArray(char *dirname)` will write and read arrays into the named directory. The same format is used for both reading and writing, so once arrays are written they can be read in easily. The directory will include a file, `3Darraypars.dat`, describing the dimensions and symmetry of the array, so only one array can be stored in a given directory. If the array being read has different dimensions or symmetry, the arrays will be deleted and re-created with the correct amount of allotted memory.

3.2 Cartesian Harmonics Data Arrays

Angular information can also be stored in terms of expansion coefficients describing expansions in terms of powers of unit vectors $\hat{e}$. For a given radial bin `ir`, coefficients F_{ℓ_x,ℓ_y,ℓ_z} can describe the function,

$$F(\Omega) = \sum_{\ell_x,\ell_y,\ell_z} \frac{(\ell_x+\ell_y+\ell_z)!}{\ell_x!\ell_y!\ell_z!} F_{\ell_x,\ell_y,\ell_z} e_x^{\ell_x} e_y^{\ell_y} e_z^{\ell_z}, \tag{9}$$

where the arrays satisfy the tracelessness constraint $F_{\ell_x,\ell_y,\ell_z+2} + F_{\ell_x+2,\ell_y,\ell_z+2} + F_{\ell_x,\ell_y+2,\ell_z} = 0$. With this constraint the elements for a given rank $\ell = \ell_x + \ell_y + \ell_z$ are determined by the $\ell_x = 0, 1$ elements. Thus, there are $(2\ell + 1)$ independent coefficients for each ℓ.

The public members of the class are:

```
class CCHArray{
 CCHArray(char *arrayparsfilename);
 CCHArray(int LMAXset,int NRADIALset,double RADSTEPset);
 CCHArray(int LMAXset,int NRADIALset,double RADSTEPset,
   bool XSYMset,bool YSYMset,bool ZSYMset);
 ~CCHArray();

 int GetLMAX();
 int GetNRADIAL();
 double GetRADSTEP();
 void SetLMAX(int LMAXset);
 void SetRADSTEP(double RADSTEPset);
 bool GetXSYM();
 bool GetYSYM();
 bool GetZSYM();
 void PrintPars();

 void ZeroArray();
 void ZeroArray(int lx,int ly,int lz);
 void ZeroArray(int ir);
 void ZeroArray_Partial(int LMAX_Partial);
 void ZeroArray_Partial(int LMAX_Partial,int ir);
 void ScaleArray(double scalefactor);
 void ScaleArray(double scalefactor,int ir);

 double GetElement(int lx,int ly,int lz,int ir);
 double GetElement(int lx,int ly,int lz,double r);
 void SetElement(int lx,int ly,int lz,int ir,double Element);
 void SetElement(int lx,int ly,int lz,double r,double Element);
 void IncrementElement(int lx,int ly,int lz,int ir,double increment);
```

```
void IncrementElement(int lx,int ly,int lz,double r,double increment);

void PrintArrayFixedIR(int ir);
void PrintArrayFixedIR(int LMAXPrint,int ir);
//FillRemainder functions use identity
//  A_{lx+2,ly,lz}+A_{lx,ly+2,lz}+A_{lx,ly,lz+2}=0
// to find entire array if (2L+1) values for lx=0,1 are known

double GetBiggest(int ir);

// Read and Write to a given directory, "AX" suffix means only lx=0,1 elements
// will be read/written. To write only up to WLMAX, use WriteShort()
void ReadAX(char *dirname);
void WriteAX(char *dirname);
void ReadAllA(char *dirname);
void WriteAllA(char *dirname);
void WriteShort(char *filename,int WLMAX);

void IncrementAExpArray(double x,double y,double z,double weight);
void IncrementAExpArrayFromE(double ex,double ey,double ez,
      double weight,int ir);
void AltIncrementAExpArrayFromE(double ex,double ey,double ez,
 double weight,int ir);
void AltAltIncrementAExpArrayFromE(double ex,double ey,double ez,
    double weight,int ir);
void IncrementMArrayFromE(double ex,double ey,double ez,double weight,int ir);
void IncrementAExpArrayFromThetaPhi(double theta,double phi,
     double weight,int ir);
void IncrementMArrayFromThetaPhi(double theta,double phi,
  double weight,int ir);

// Returns < ex^lx ey^ly ez^lz > where (ex,ey,ez) is unit vector
double GetMElementFromAExpArray(int lx,int ly,int lz,int ir);
// Given array, < ex^lx ey^ly ez^lz >, this returns the CH expansion
// element
double GetAExpElementFromMArray(int lx,int ly,int lz,int ir);

void FillRemainderX(int ir);
void FillRemainderY(int ir);
void FillRemainderZ(int ir);
void FillRemainderX();
void FillRemainderY();
void FillRemainderZ();

// For CH Expansion array, this returns
// F(Omega)=\sum_{lx,ly,lz} (L!/(lx!ly!lz!))
// F_{lx,ly,lz} e_x^lx e_y^ly e_z^lz
double AExpand(double ex,double ey,double ez,int ir);
double AExpand(double x,double y,double z);
double AExpand(double theta,double phi,int ir);

void Detrace(int ir);
void Detrace();

void Randomize(double mag,int ir);
void RandomizeA(double mag,int ir);
void Randomize(double mag);
```

```
 void RandomizeA(double mag);
 void RandomizeA_Gaussian(double mag,int ir);
 void RandomizeA_Gaussian(double mag);
};
```

The parameters describing the size of the array are `LMAX, NRADIAL, RADSTEP` and `XSYM, YSYM, ZSYM` and can be set either with the constructors or in the parameters file as described for the cartesian arrays in the previous subsection. These parameters are printed by calling `PrintPars()`.

Some of the functions are overloaded with and without dependences on the radial coordinate `r` or the radial index `ir`. The variations without radial dependencies operate on all the radial indices.

Individual elements can be accessed, set or incremented with the functions `GetElement(...), SetElement(...)` and `IncrementElement(...)`. The array is set to zero with `ZeroArray(...)` and `Randomize(...)` sets the elements to random values between -1 and 1.

The reading and writing routines `ReadAllA` and `WriteAllA` will read and write all elements of the array, whereas `ReadAX` and `Write AX` read and write only those components with $\ell_x = 0, 1$. This is useful for traceless arrays since the remaining components are easily generated from those with $\ell_x = 0, 1$. The function `FillRemainderX()` will accomplishes this feat using the tracelessness constraint above, which can be re-expressed as: $F_{\ell_x+2,\ell_y,\ell_z} = -F_{\ell_x,\ell_y+2,\ell_z} - F_{\ell_x,\ell_y,\ell_z+2}$. `FillRemainderY()` and `FillRemainderZ()` fill the remainders of the arrays starting with the $\ell_y = 0, 1$ and $\ell_z = 0, 1$ components respectively. The function `WriteShort(char *filename,int WLMAX)` writes the $\ell_x = 0, 1$ components (skipping those that are zero due to symmetry) up to $L \leq WLMAX$ to a single file in format that is convenient for graphing, but not sufficiently accurate for re-reading.

The value of the function expanded with Eq. (9) can be found with the functions `AExpand`.

The detracing operation is used by the array operations which multiply, divide and invert arrays. If one has an arbitrary non-detraced array M which describes a function

$$M(\Omega) = \sum_{\ell_x,\ell_y,\ell_z} \frac{\ell!}{\ell_x!\ell_y!\ell_z!} M_{\ell_x,\ell_y,\ell_z} e_x^{\ell_x} e_y^{\ell_y} e_z^{\ell_z}, \tag{10}$$

The `Detrace function` returns a detraced array that yields the identical angular function $M(\Omega)$.

Moments of unit-vector components $\langle e_x^{\ell_x} e_y^{\ell_y} e_z^{\ell_z} \rangle$ can be calculated from arrays of Cartesian-Harmonic coefficients with the functions `GetMElementFromAExpArray`. Similarly, if an array stores the moments, the equivalent element for an array of Cartesian-Harmonic expansion coefficients that gives those moments can be found with `GetAExpElementFromMArray(...)`.

The `IncrementAExpArray(...)` and `IncrementMexpArray()` can be used to generate arrays which will, when expanded, reproduce functions with the same angular distribution as the sampling of coordinates used to increment the function. For instance, the following set of calls will produce a numerator for a correlation function given a set of relative momenta `qx, qy` and `qz`,

```
CHArray *numerator;
numerator=new CHArray(parfilename);
for(ipair=0;ipair<NPAIRS;ipair++){
  numerator->IncrementAExpArray(qx[ipair],qy[ipair],qz[ipair],1.0);
}
```

Similarly an array of moments $\langle e_x^{\ell_x} e_y^{\ell_y} e_z^{\ell_z} \rangle$ can also be calculated with by using `IncrementMArray(...)` then scaling the array to make into an average.

If the object stores moments, `GetAExpElementFromMArray(...)` will return the Cartesian-Harmonic expansion coefficient, and if the object stores expansion coefficients, `GetMElementFromAExpArray(...)` will return specific moments.

3.3 Arrays for Expansions in Spherical Harmonics

A class is also provided to accommodate expansions in spherical harmonics. The $Y_{\ell,m}$ expansion coefficients are used to define angular functions,

$$F(\Omega) = (4\pi)^{1/2} \sum_{\ell,m} F_{\ell,m} Y_{\ell,m}(\Omega). \tag{11}$$

This class is rather incomplete. However, one can do all calculations using the Cartesian Harmonics of the previous subsection, then use one of the `CopyArray` functions described in the next section to translate expansion coefficients stored in a `CCHArray` arrays to those for a `CYlmArray`.

```
class CYlmArray{
  public:
    CYlmArray(int LMAXset,int NRADIALset);
    ~CYlmArray();
    int GetLMAX();
    complex<double> GetElement(int L,int M,int ir);
    void SetElement(int L,int M,int ir,complex<double>);
    void IncrementElement(int L,int M,int ir,complex<double> increment);
    void ScaleArray(double scalefactor);
    void ScaleArray(double scalefactor,int ir);
    void ZeroArray();
    void ZeroArray(int ir);
    void PrintArrayFixedIR(int ir);
    void PrintArrayFixedIR(int LMAXPrint,int ir);
};
```

3.4 Array Operations Involving Multiple Arrays

In addition to the functionality described in the various member functions for the array objects, CorAL also provides functions involving more than one array through the namespace `ArrayCalc`:

```
namespace ArrayCalc{
  void CopyArray(CCHArray *A,CCHArray *B);
  void CopyArray(CCHArray *A,int ira,CCHArray *B,int irb);
  void CopyArray(C3DArray *A,C3DArray *B);


  void CalcMArrayFromAExpArray(CCHArray *A,CCHArray *M);
  void CalcMArrayFromAExpArray(CCHArray *A,int ira,CCHArray *M,int irm);
  void CalcAExpArrayFromMArray(CCHArray *M,CCHArray *A);
  void CalcAExpArrayFromMArray(CCHArray *M,int irm,CCHArray *A,int ira);
  void CalcAExpArrayFromXExpArray(CCHArray *X,CCHArray *A);
  void CalcAExpArrayFromXExpArray(CCHArray *X,int irx,CCHArray *A,int ira);
  void CalcXExpArrayFromAExpArray(CCHArray *A,CCHArray *X);
  void CalcXExpArrayFromAExpArray(CCHArray *A,int ira,CCHArray *X,int irx);
  void CalcYlmExpArrayFromAExpArray(CCHArray *A,int ir,
     CYlmArray *YlmArray,int irlm);
  void CalcAExpArrayFromYlmExpArray(CYlmArray *YlmArray,int irlm,
     CCHArray *A,int ira);
  void CalcAExpArrayFrom3DArray(C3DArray *threedarray,CCHArray *A);
  void Calc3DArrayFromAExpArray(CCHArray *A,C3DArray *threed);


  void AddArrays(CCHArray *A,CCHArray *B,CCHArray *C);
  void AddArrays(CCHArray *A,int ira,CCHArray *B,int irb,CCHArray *C,int irc);
  void AddArrays(C3DArray *A,C3DArray *B,C3DArray *C);
```

```
  // If C(Omega)=A(Omega)*B(Omega), this finds A in terms of A and B
  void MultiplyArrays(CCHArray *A,CCHArray *B,CCHArray *C);
  void MultiplyArrays(CCHArray *A,int ira,CCHArray *B,
     int irb,CCHArray *C,int irc);
  void MultiplyArrays(C3DArray *A,C3DArray *B,C3DArray *C);
  // If you know array is zero up to given Ls, or don't care to detrace,
  // this can save time
  void MultiplyArrays_Partial(int LMAXA,CCHArray *A,int ira,
     int LMAXB,CCHArray *B,int irb,
     int LMAXC,CCHArray *C,int irc);


  // If A(Omega)=B(Omega)*C(Omega), this finds C in terms of A and B
  void DivideArrays(CCHArray *A,CCHArray *B,CCHArray *C);
  void DivideArrays(CCHArray *A,int ira,CCHArray *B,int irb,
     CCHArray *C,int irc);
  void DivideArrays(C3DArray *A,C3DArray *B,C3DArray *C);


  void Detrace(CCHArray *M,CCHArray *A);
  void Detrace(CCHArray *M,int irm,CCHArray *A,int ira);


  bool CompareArrayParameters(C3DArray *threed,CCHArray *A);
  bool CompareArrayParameters(CCHArray *A,C3DArray *threed);
  bool CompareArrayParameters(CCHArray *A,CCHArray *B);
  bool CompareArrayParameters(C3DArray *threeda,C3DArray *threedb);
};
```

The `CopyArray(...)` functions check to make sure that the array parameters are equal before proceeding.

The `Calc*ArrayFrom*Array(...)` functions translate one type of format into another. The `AExp` qualifier is for arrays of Cartesian-Harmonic expansion coefficients, whereas `YlmExp` refers to expansion coefficients using spherical harmonics. The `3D` is for Cartesian meshes. These three forms of arrays were described previously.

The `AddArrays, MultiplyArrays and DivideArrays` functions perform as advertised. For functions $F(\mathbf{r})$ described by the expansion coefficients in the arrays, the results provide arrays describing the arrays in $\mathbf{r}$ space. For instance, `DivideArrays(A,B,C)` operates on Cartesian-Harmonic coefficients, the resulting array `C` would satisfy $C(\mathbf{r}) = A(\mathbf{r})/B(\mathbf{r})$ when expanded. These functions do some checking to make sure that the symmetries of the resulting arrays are appropriate, but this checking is not yet complete.

The `Detrace(A,B)` operation provides an array of Cartesian-Harmonic coefficients `B` that satisfies the tracelessness condition while expanding identically as `A`, i.e., $A(\mathbf{r}) = B(\mathbf{r})$.

The boolean functions `CompareArrayParameters` check whether arrays have identical dimensions and symmetries (`XSYM, YSMY` and `ZSYM`). For comparisons of `CCHArray` and `C3DArray` objects, only symmetries are tested.

4 Wave Functions

One of the basic elements of the code base are objects which derive from the `CWaveFunction` class. The main functionality of these classes is to calculate the squared wave function. The public members of the base class are:

```
class CWaveFunction{
  public:
    int GetNQMAX();
    double GetQ(int iq);
```

```
    void PrintCdelta(double Rx,double Ry,double Rz);
    double GetPsiSquared(double *pa,double *xa,double *pb,double *xb);
    double GetPsiSquared(double q,double r,double ctheta);
    virtual double CalcPsiSquared(int iq,double r,double ctheta);
    CWaveFunction();
    ~CWaveFunction();
}
```

Calculations for wave functions are based on stored parameters for specific magnitudes of the relative momentum. Examples of such information might be phase shifts. To view the values of q used for calculations, one can use the functions `GetNQMAX()` and `Get Q(int iq)`. The relative momenta are the canonical momenta as measured in the two-particle frame, i.e., in that frame the momenta are **q** and **-q**, and are measured in MeV/c.

The function `printCdelta` is not meant to be used often, as it provides a plot of the correlation function for a Gaussian source in terms of phase shifts, ignoring Coulomb, and assuming that $qR >> \hbar$. It is really only used for checking certain types of calculations.

The functions `GetPsiSquared(...)` and `CalcPsiSquared(iq,r,ctheta)` provide the squared wave function in terms of the relative momentum, or in terms of the two momenta of the particles. The two `GetPsiSquared(..` functions call `CalcPsiSquared` by interpolating for different values of `iq`.

New `CWaveFunction` objects must be created for each class of interaction, e.g., pp, $\pi^+\pi^-$ or pK^+. The function `CalcPsiSquared` is virtual as it is different for each class which derives from it, e.g., `CWaveFunction_pp`, `CWaveFunction_pipluspiplus`, etc.. The constructors are not listed in this class, though they all have the same form, e.g., `CWaveFunction_pp(char *parsfilename)`. The parameters file must have the form:

```
int NQMAX 20
double DELQ 4.0
double EPSILON 1.0
```

The momentum mesh is defined by `NQMAX` and `DELQ`, and `EPSILON` refers to a distance within which ϕ^2 is a constant. As long as the characteristic source size is larger than `EPSILON` resulting correlation functions should be independent of `EPSILON`. To better understand this, one can read the file "corrtail.pdf" and "longpaper.pdf" in "doc/papers/". This method requires good knowledge of experimental phase shifts. Unfortunately, such phase shifts are often provided as if the Coulomb interaction does not exists, so one must then modify the phase shifts to account for Coulomb which is done with the `CoulWave::phaseshifts_CoulombCorrect(.` utilities described in Sec. 7. It should be emphasized that this approximation can significantly affect the answer, and that stable results require that the phaseshifts and their derivatives are consistent to a high level. To reduce the sensitivity to this approximation, pp phase shifts for the s-wave are calculated by solving the Schrödinger equation with the Reid soft-core potential.

Classes were defined in such a way to easily accommodate adding additional classes of interactions. Currently, the classes that inherit from `CWaveFunction` are:

`CWaveFunction_pp`	proton-proton†
`CWaveFunction_pn`	proton-neutron†
`CWaveFunction_nn`	neutron-neutron†
`CWaveFunction_Xipi`	Ξ^0, π^+
`CWaveFunction_kpluspiminus`	$K^+\pi^-$ (Coulomb and K*)
`CWaveFunction_lambdalambda`	$\Lambda\Lambda$ (will prompt for scattering length)
`CWaveFunction_plambda`	$p\Lambda$
`CWaveFunction_piplusplus`	$\pi^+\pi^{+\dagger}$
`CWaveFunction_pipluspiminus`	$\pi^+\pi^{-\dagger}$
`CWaveFunction_pkplus`	proton-$K^{+\dagger}$
`CWaveFunction_ppiplus`	proton-$\pi^{+\dagger}$

†Uses fairly complete set of experimental phase shifts

5 Kernels and Convolutions

Kernels are used to provide a connection between both correlations functions and source functions when both are expressed in terms of expansion coefficients for either the Cartesian-Harmonic or the $Y_{\ell,m}$ basis. Since squared wave functions $|\phi(q,r,\cos\theta_{qr})|^2$ is rotationally invariant (**q** and **r** rotate together) there is a one-to-one correpondence between angular moments of the correlation and source functions with the same indices:

$$\mathcal{R}_{\ell,m}(\mathbf{q}) = \int 4\pi r^2 dr \mathcal{K}_\ell(q,r)\mathcal{S}_{\ell,m}(\mathbf{r}) \quad (12)$$

$$\mathcal{R}_{\ell_x,\ell_y,\ell_z}(\mathbf{q}) = \int 4\pi r^2 dr \mathcal{K}_\ell(q,r)\mathcal{S}_{\ell_x,\ell_y,\ell_z}(\mathbf{r}) \quad (13)$$

$$\mathcal{K}_\ell(q,r) = \frac{1}{2}\int d\cos\theta_{qr}|\phi(q,r,\cos\theta_{qr})|^2 \quad (14)$$

Thus, the kernel $\mathcal{K}_\ell(q,r)$ plays the basic role when linking source properties to correlations when analyzing with angular correlations and is determined by the wave function. All types of interactions provide resolving power for kernels, even for values of ℓ greater than those that matter for the interactions. See "doc/papers/longpaper.pdf".

The members of `CKernel` are:

```
class CKernel{
  public:
    CWaveFunction *wf;
    double GetValue(int ell,double q, double r);
    double GetValue(int ell,int iq,int ir);
    void Read(char *datadirname);
    void Write(char *datadirname);
    void Print();
    int GetLMAX();
    double GetDELR();
    double GetDELQ();
    int GetNQMAX();
    int GetNRMAX();
    void Calc();
    void Calc_ClassCoul(double ma,double mb,int zazb);
    void Calc_PureHBT();
    CKernel(CWaveFunction *wf,char *kparsfilename);
    ~CKernel();
    double GetPsiSquared(int iq,int ir,double ctheta);
    double GetPsiSquared(int iq,double r,double ctheta);
    double GetPsiSquared(double q,double r,double ctheta);
}
```

The constructor sets parameters but does not calculate the kernels. To calculate the kernels, one uses one of three calls:

`Calc`	Will use the wave function object to generate kernel
`Calc_ClassCoul`	Will calculate kernel for classical Coulomb interaction
`Calc_PureHBT`	Identical bosons, no Coulomb or strong interaction

Kernels are stored in meshes denoted by the parameters `ell`, `iq` and `ir` with values bounded by `LMAX`, `NRMAX` and `NQMAX`, and the granularities set by `DELR` and `DELQ`. They can be written to files with the `Write()` function which needs a character string as an argument to name the directory. The `Read()` function will read from the same format. In the named directory files will be written for each (`ell,q`) combination with filenames set by the values of `q`, e.g., `ell2_q52.dat` would store kernel information for `ell=2, q=52`. The first line of the file give `NRMAX` and `DELR`, with the subsequent lines giving the kernel values for `ir=0,1,...`. Thus, one should store information for different kernels in different directories.

Once wave functions or kernels are calculated and a source function is stored in an array, one can calculate the correlation function with functions in the namespace `Source2CF`:

```
namespace CS2CF{
  void s2c(C3DArray *s,CWaveFunction *wf,C3DArray *c);
  void s2c(CCHArray *s,CKernel *kernel,CCHArray *c);
};
```

One uses kernels to perform convolutions with angular decompositions, and wave-functions for convolutions connecting source and correlation functions stored in three-dimensional Cartesian meshes. However, since the kernels are a fairly efficient way of storing the calculated squared wave functions, you can also reproduce the squared wave functions with the functions `GetPsiSquared()`.

6 Calculating Source Functions

Source and correlation function information are both stored in the same types of arrays. Calculating the source from a model, e.g., blast-wave, requires creating a `CSourceCalc` object, or an object derived from this class. This objects stores parameters describing the source, e.g., Gaussian radii, but the actual source information is stored in arrays of the type described in Sec. 3. The arrays are filled with calls of the type `scalc->CalcS(sarray)`, where `scalc` would be a pointer to a source object and `sarray` would be a pointer to a the array storing the source function. The public members of the class are:

```
class CSourceCalc{
  public:
    parameterMap spars;
    virtual void CalcS(CCHArray *A);
    void ReadSPars(char *sparsfilename);
    void NormCheck(CCHArray *A);
    void CalcEffGaussPars(CCHArray *A);
    CSourceCalc::CSourceCalc();
};
```

All source parameters are stored as a `parameterMap` objects which are regular c++ maps of strings:

```
typedef map<string,string> parameterMap;
```

The namespace `parameter` includes a variety of functions which are mostly self explanatory:

```
namespace parameter {
    bool getB(parameterMap ,string ,bool);
    int getI(parameterMap ,string ,int);
    string getS(parameterMap ,string ,string);
    double getD(parameterMap ,string ,double);
    vector< double > getV(parameterMap, string, double);
    vector< string > getVS(parameterMap, string, string);
    vector< vector< double > > getM(parameterMap, string, double);
    void set(parameterMap&, string, double);
    void set(parameterMap&, string, int);
    void set(parameterMap&, string, bool);
    void set(parameterMap&, string, string);
    void set(parameterMap&, string, char*);
    void set(parameterMap&, string, vector< double >);
    void set(parameterMap&, string, vector< string >);
    void set(parameterMap&, string, vector< vector< double > >);
    void ReadParsFromFile(parameterMap&, char *filename);
    void PrintPars(parameterMap&);
};
```

The `ReadParsFromFile(paramtermap&, char *filename)` function will input parameters from a file. For example, the parameters used for reading in parameters for a `CSourceCalc_Gaussian` object could be spec-

ified in a file with the following lines:

```
double Pt 400
double DELPT 30
string OSCARfilename /usr/users/johnson/data/oscarfile.dat
int IDa 211
int IDb -211
```

For specific sources, one must use objects which derive their properties from `CSourceCalc`. Currently, there are three such objects, `CSource_Gaussian, CSource_Blast` and `CSource_OSCAR`, which can be used to generate source functions for Gaussian, blast-waves, and from files where phase space points have been recorded in the OSCAR format. Each type of object has quite different parameters:

Source Parameters

type	parameter	default	class/description
			CSource_Gaussian
double	Rx	4.0	One-particle Gaussian source sizes in c.o.m. frame
double	Ry	4.0	$\rho(\mathbf{r}) \sim e^{-x^2/2R^2}$
double	Rz	4.0	R_i For non-identicals, $R^2 = (1/2)(R\_a^2 + R\_b^2)$
double	Xoff	0.0	For non-identicals
double	Yoff	0.0	these are off-sets for separations
double	Zoff	0.0	of centroids of two Gaussians
double	Euler_Phi	0.0	Euler angles for
double	Euler_Theta	0.0	rotations of principle
double	Euler_Psi	0.0	axes
			Csource_Blast*
double	Rx	13	in-plane transverse radius
double	Ry	13	out-of plane (sharp cutoff radii)
double	Tau	12	Bjorken time (All emission at once, $\Delta\tau = 0$)
double	BetaX	0.7	Transverse velocities at surface
double	BetaY	0.7	for in/out-of-plane
double	T	110	Temperature in MeV
double	Pt	600	Total p_t of particles in MeV/c
double	Phi	0.0	Angle of emission relative to x axis
double	EtaG	2.0	Gaussian width of source rapidities ∞ for Bjorken
double	Ma	938.28	mass of first particle
double	Mb	139.58	mass of second particle
int	Nsample	1000	# of points to sample 1-particle distribution source sampling $\sim$Nsample^2
			CSource_OSCAR**
double	Pt	600	Total momentum of pair in MeV/c
double	DELPT	20	Range for accepting momentum of particles, i.e., $\lvert p_{t,a} - m_a P_t/(m_a + m_b)\rvert < \Delta P_t$
double	PHIMIN_DEG	0	Range for azimuthal angles, phase space
double	PHIMAX_DEG	360.0	points $\mathbf{p}_b$ and $\mathbf{r}_b$ will be rotated so that $\mathbf{p}_a$ and $\mathbf{p}_b$ are parallel once they pass filter
double	YMIN	-3.0	Range of rapidities, particles
double	YMAX	3.0	boosted to have same rapidity, $y - a = y_b$
int	IDa	211	Particle Data Book
int	IDb	211	IDs
double	Ma	139.58	Masses
double	Mb	139.58	
bool	AEQUALB	0	Set to 1 if particles are non-identical, but one uses same phase space points
int	NMAX	20000	Maximum # of phase space points (array size)
string	OSCARfilename	UNDEFINED	Location of phase space points
int	NEVENTSMAX	10000	Maximum # of events to be read from file

*Blast-wave calculations assume $\mathbf{u}_\perp = \gamma \mathbf{v}_\perp$ rise linearly from origin.

**All events for which one wishes to mix-and-match phase space points should stored in the same OSCAR file. Data corresponding to different impact parameters should be stored in different files.

7 Utilities

The directory `src/Utilities` contains code used for special functions, random-number generation, parameter maps, lorentz boosts, Clebsch Gordan coefficients, Monte-Carlo generation of momenta for boltzmann distributions and the "triangle function". We devote space below to describing some of the utilities which we think might likely be of some interest to a user:

The random number generation routines are fairly self explanatory:

```
class CRandom{
public:
  double ran(void);
  unsigned long int iran(unsigned long int imax);
  double gauss(void);
  void gauss2(double *g1,double *g2);
  CRandom(int seed);
  void reset(int seed);
  void generate_boltzmann(double mass,double T,double *p);
}
```

Here, `ran` returns a random real number between 0 and 1, `iran` returns an integer $0 \leq i < imax$, and gauss returns a gaussian random number consistent with the distribution $e^{-x^2/2}$. The function `gauss2` returns a pair of gaussian numbers which is efficient if one is creating more than one gaussian number since the fundamental algorithm generates pairs of numbers. The function `generate_boltzmann()` returns momenta consistent with a Boltzmann distribution.These routines use the **GSL** library with the "ranlxd1" choice of random number generators. The properties of the various algorithms are documented in the **GSL** documentation. By editing the code `src/Utilities/Random/gslrandom.cc`, one can easily switch to different classes of random number generators.

Including `src/Utilities/Misc/misc.h` and `src/Utilities/Misc/misc.cc` provides a user with several functions. Clebsch Gordan coefficients can be generated with `double cgc(double j1,double m1,double j2,double m2,double j,double m)`. A four vector `p[4]` can be boosted by a four velocity `u[4]` to `pprime[4]` with `void lorentz(double *u,double *p,double *pprime)`. The "triangle function", `double triangle(double M,double ma,double mb)`, defined by $\sqrt{M^4 + m_a^2 + m_b^2 - 2M^2m_a^2 - 2M^2m_b^2 - 2m_a^2m_b^2}/(4M^2)$, gives the relative momentum of a object of mass M decaying. The utility `bool comparestrings(char *s1,char *s2)` returns `true` if the strings are identical. Finally, the routine `outsidelong(...)` can be used to find the projections of the relative momentum in the pair frame from two four vectors $\mathbf{p}_a$ and $\mathbf{p}_b$. The header file `src/Utilities/Misc/misc.h` has the following prototypes:

```
  void lorentz(double *u,double *p1,double *p1prime);
  double cgc(double j1,double m1,double j2,double m2,double j,double m);
  bool comparestrings(char *s1,char *s2);
  double triangle(double m0,double m1,double m2);
  void outsidelong(double *pa,double *pb,
    double &qinv,double &qout,double &qside,double &qlong);
```

which are hopefully self-explanatory. Note that `outsidelong` includes the outwards boost which makes $q_{\rm inv}^2 = q_{\rm out}^2 + q_{\rm side}^2 + q_{\rm long}^2$. Thus, the returned value of $q_{\rm out}$ is shorter by a factor $\gamma = (1 - v_\perp^2)^{-1/2}$ than what most experimental groups use for their conventions. Also, the canonical relative momentum is used, $q = (1/2)(p'_a - p'_b)$ in the pair frame. This also differs from the usual convention for $\pi\pi$ interferometry, but follows the convention used for most other correlations such as pp analyses.

Special functions include spherical harmonics, Legendre Polynomials, Cartesian Harmonics, Coulomb wave functions and a variety of Bessel functions, which are all defined in `src/Utilities/SpecialFunctions/sf.h`. Many of the special functions are simply **GSL** routines in disguise.

Legendre polynomials and spherical harmonics are functions of the namespace `SpherHarmonics`:

```
namespace SpherHarmonics{
  double legendre(int ell,double ctheta);
  complex <double> Ylm(int ell,int m,double theta,double phi);
}
```

Bessel functions are incorporated as a namespace, with the function names being self-explanatory:

```
namespace Bessel{
  double J0(double x);
  double J1(double x);
  double Jn(int n, double x);

  double K0(double x);
  double K1(double x);
  double Kn(int n, double x);

  double Y0(double x);
  double Y1(double x);
  double Yn(int n, double x);

  double I0(double x);
  double I1(double x);
  double In(int n, double x);

  double j0(double x);
  double j1(double x);
  double jn(int n, double x);


  double y0(double x);
  double y1(double x);
  double yn(int n, double x);

  complex<double> h0(double x);
  complex<double> h1(double x);
  complex<double> hn(int n, double x);


  complex<double> hstar0(double x);
  complex<double> hstar1(double x);
  complex<double> hstarn(int n, double x);
};
```

The namespace `CoulWave` provides functions for calculating Coulomb partial waves. Here, `CWincoming` and `CWoutgoing` are the incoming/outgoing waves defined in terms of the regular and irregular solutions $F_L \pm iG_L$ respectively. The gamma function of a complex argument $\Gamma(z)$ is `cgamma()`.

```
namespace CoulWave{
  void GetFG(int L,double x,double eta,double *FL,double *GL);
  void GetFGprime(int L,double x,double eta,double *FL,double *GL,
    double *FLprime,double *GLprime);
  complex<double> CWincoming(int ell,double x,double eta);
  complex<double> CWoutgoing(int ell,double x,double eta);
  complex<double> cgamma(complex<double> cx);
  void phaseshift_CoulombCorrect(int ell,double q,double eta,
    double &delta,double &ddeltadq);
};
```

Here, the function `phaseshift_CoulombCorrect` provides a **crude** way to scale phase shifts to account for their distortion due to the Coulomb interaction. In many instances, models or parameterizations provide phase shifts assuming there is no Coulomb interaction, e.g., $\delta_s \sim qa$. This correction assumes that $\tan\delta_{\rm withCoul.} = \tan\delta_{\rm noCoul.} \cdot {\rm Gamow}_\ell(q)$, where the Gamow factor is $F_\ell(q, r=0)/qr|^2$ for a Coulomb wave function.

Cartesian harmonic functions are included as a class rather than as a namespace for the sake of efficiency.

Since many of the calculations use binomial and trinomial distributions, and since the `overlap` function might be used repeatedly, arrays are strategically stored so that subsequent calls can be performed more quickly. The public members of the `CCHCalc` object are:

```
class CCHCalc{
  public:
    CCHCalc();
    ~CCHCalc();
    double GetAFromE(int lx,int ly,int lz,double ex,double ey,double ez);
    double GetAFromThetaPhi(int lx,int ly,int lz,double theta,double phi);
    double GetMFromE(int lx,int ly,int lz,double ex,double ey,double ez);
    double GetMFromThetaPhi(int lx,int ly,int lz,double theta,double phi);


    double GetOverlap(int lx,int ly,int lz,int lxprime,int lyprime,int lzprime);
    double GetOverlap0(int lx,int ly,int lz,int lxprime,int lyprime,int lzprime);


    double Factorial(int n);
    double DoubleFactorial(int n);
    double Binomial(int lx,int ly);
    double Trinomial(int lx,int ly,int lz);
};
```

Here, the `GetA...` functions give the Cartesian harmonics $\mathcal{A}_{\vec{\ell}}$ as functions of either $\cos\theta$ and ϕ or in terms of unit vector components e_x, e_y, e_z. The `GetM` functions are simply moments of unit-vector components, $M_{\vec{\ell}} = e_x^{\ell_x} e_y^{\ell_y} e_z^{\ell_z}$. Unlike spherical harmonics, Cartesian harmonics are not orthogonal, and `GetOverlap0(...)` and `GetOverlap(...)` return $(1/4\pi)\int d\Omega A_{\vec{\ell}}(\Omega) A_{\vec{\ell}'}(\Omega)$. The function `GetOverlap(...)` stores the calculated values, creating the needed memories as they are calculated. This saves time but can use significant memory if one is calculating for $\ell \geq 50$. Thus, if one only plans to calculate a few overlaps, one should use `GetOverlap0(...)`. Binomial, factorial and trinomial functions also used stored values for increased speed.

8 Imaging

Kernels and wavefunctions provide the means to calculates correlation function objects if given source function objects. The reverse process is more difficult. Correlation functions and source functions are related to one another through the convolution,

$$\mathcal{R}_{\vec{\ell}}(q) = \int dr \mathcal{K}_\ell(q,r) S_{\vec{\ell}}(r). \tag{15}$$

If one views $\mathcal{S}$ as a vector with N_r components and $\mathcal{R}$ as a vector with N_q components, the kernel is a $N_r \times N_q$ matrix which relates the two vectors by

$$\mathcal{R}_i = \sum_j \mathcal{K}_{ij} \mathcal{S}_j, \tag{16}$$

where the $\vec{\ell}$ indices are suppressed. The term "imaging" denotes the problem of finding $\mathcal{S}$ from $\mathcal{R}$, i.e., inverting the kernel. Unfortunately, the kernel is usually highly singular, and even if $\mathcal{C}$ is measured to high accuracy, one can often not determine more than a handful of parameters of information describing $\mathcal{R}$.

Thus all imaging procedures involve fitting routines at their core. Even splines are themselves parameterized functions which are then fit.

```
class CParInfo{
  public:
    bool fixed; // 'true' if parameter is not allowed to vary
    // searches are confined to xmin < x < xmax
    double xmin,xmax,error,xbar;
    double bestx,currentx; // bestx is the value that vave smallest chi^2
```

```
    char *name;
    void Set(string sname,double xset,double error,
    double xminset,double xmaxset);
    CParInfo();
    ~CParInfo();
};

class CCF2SFit{
  public:
    /*
    calcflag = 1 if CF is a CCHArray object of specific lx,ly,lz
    and CSourceCalc object makes CCHArray objects
    calcflag = 2 if CF is a C3DArray object
    and CSourceCalc object makes objects
    calcflag =3,4 are used if source functions are calculated through
    intermediate MC lists, but no such implementations yet exist
    */
    void SetCalcFlag(int calcflagset);
    void SetMCSourceFlag(bool MCsourceflagset);

    void SetPar(string parstring,double value);
    void SetPar(string parstring,double value,double error,
    double xmin,double xmax);
    void AddPar(string parstring,double value,double error,
    double xmin,double xmax);
    void PrintPars();
    void PrintErrorMatrix();
    void PrintStepMatrix();
    void FixPar(string parname);
    void FreePar(string parname);
    void UseBestPars();
    void SetL(int lxset,int lyset,int lzset);

    void Metropolis(int maxcalls);
    void SteepestDescent(int maxtries);
    void Newton(int maxtries);
    void UpdateStepMatrix();
    void InitErrorMatrix();

    // This calculates source functions
    CSourceCalc *sourcecalc;

    // ----------- The objects below depend on calc_flag ----------
    // These are used to store information about the source
    CCHArray *sourceCH;
    C3DArray *source3D;
    CMCList *lista;
    CMCList *listb;

    // Wavefunctions or kernels
    CKernel *kernel;
    CKernelWF *kernelwf;
    CWaveFunction *wf;

    // These are arrays for storing correlation functions and errors
    C3DArray *cexp3D;
```

```
    C3DArray *cerror3D;
    C3DArray *ctheory3D;
    CCHArray *cexpCH;
    CCHArray *cerrorCH;
    CCHArray *ctheoryCH;

    CCF2SFit();
    CCF2SFit(CCHArray *sourceCHset,C3DArray *source3Dset,
    CMCList *listaset,CMCList *listbset,
    CKernel *kernelset,CKernelWF *kernelwfset,
    CWaveFunction *wfset,
    C3DArray *cexp3Dset,C3DArray *cerror3Dset,
    C3DArray *ctheory3Dset,CCHArray *cexpCHset,
    CCHArray *cerrorCHset,CCHArray *ctheoryCHset);
     CCF2SFit();

  protected:
    int calcflag;
    static bool MCsourceflag; // If the source has a Monte Carlo nature,
    // i.e., it fluctuates for a given parameter set, set this to true

    CParInfo **par;
    static int nmaxpars;
    int nfreepars,npars;
    double **ErrorMatrix;
    double currentchisquared,bestchisquared;

    int lx,ly,lz;
    CRandom *randy;
    void SwitchPars(int ipara,int iparb);
    void SwitchValues(double *a,double *b);

    int ncalls;
    double **StepMatrix;
    void Init();
    double GetChiSquared(double *x);
    void CalcErrorMatrixFromCurvature(double **C);
};

class CCF2SFit_Blast :  public CCF2SFit{
  public:
    CCF2SFit_Blast(CSourceCalc *scset,C3DArray *cexpset,
    C3DArray *cerrorset,C3DArray *ctheory3Dset,
    CCHArray *ctheoryset,CCHArray *sourceset,
    CKernel *kernelset);
};

class CCF2SFit_GX1D : public CCF2SFit{
  public:
    CCF2SFit_GX1D(CSourceCalc *scset,CCHArray *cexpset,
    CCHArray *cerrorset,CCHArray *ctheoryset,
    CCHArray *sourceset,CKernel *kernelset);
};

class CCF2SFit_3DGaussian :  public CCF2SFit{
  public:
    CCF2SFit_3DGaussian(CSourceCalc *scset,C3DArray *cexpset,
```

```
    C3DArray *cerrorset,C3DArray *ctheory3Dset,
    CCHArray *ctheoryset,CCHArray *sourceset,
    CKernel *kernelset);
};
```

9 Installing, Compiling and Running CorAL

CorALwill be downloaded as a file `coral.tar.gz`. After decompression, the project will appear in a directory `CorAL/`, which can be moved at the user's discretion.

9.1 Compiling CorAL

CorALwill not function without the installation of the GSL (Gnu Scientific Library). This can be found by first dialing into:
http://www.gnu.org/software/gsl
GSL routines are used for random numbers, Bessel functions, Coulomb wave functions, spherical harmonics and Clebsch-Gordan coefficients. The code is only tested for the GNU compiler (g++, version 4.0 or higher). **GSL** and g++ are available for Linux, Windows and Mac OS X. Rather than using makefiles, CorALuses a replacement for the `make` command, `scons`, which provides a simple means to ensure that object files, libraries, and executables are recompiled on a must-compile basis. The package is available at:
http://www.scons.org

CorALCompilation requires a few environmental variables, which allow CorALto find header files, source files and libraries. These variables are:

VARIABLE	DESCRIPTION	EXAMPLE
CORAL_HOME	base directory, where `src/`, `include/`, `examples`... reside	`/Users/Barbara/CorAL/`
CORAL_GSLPATH	root of `../lib/gsl` and `../include/gsl`	`/usr/local/`
CORAL_X11PATH	root of `../lib/X11` and `../include/X11`	`/usr/X11R6/`
CORAL_CCFLAGS	optimization flags for g++	`-O2`

If the environment variables are not set, CorALwill do its best to guess the appropriate values given your operating system. ?????? can it even guess CORAL_HOME ???????

To compile the libraries, one need only go into the CorAL home directory enter:
~/CorAL% `scons`
If all works, the code will be compiled and three static libraries will appear in `lib/`: `libcoral.a`, `libcoralutils.a` and `libxgraph.a`. The xgraph library is non-essential and is mainly used for the code tests, so if it fails to compile, or if you simply have no need for it, you can delete the one pertinent line in the `SConstruct` file in the CorAL home directory.

9.2 Code Tests and Examples

For the purpose of testing CorAL, several source files are located in the `codetests/` directory. After changing into that directory, one can enter:
~CorAL/codetests%% `scons wftestX`
which will create the executable `wftestX`. Running this should open an X-window comparing a stored pp correlation function with one being calculated by `wftestX`. The procedure can be repeated with several other files in `codetests/`.

The CorAL package includes a `examples/` directory. Within that directory are several example codes. With these codes you can additionally

To incorporate CorAL functionality into your code, you need only add an include statement to your code, then compile. If you wish to use the static CorAL library (`libcoral.a` in `lib/`), you need to add the line:

```
#include "coral.h"
```

to the beginning of your source code, then compile with the command:

```
g++ -I ${CORALHOME}/include -L ${CORALHOME}/lib -lcoral -lgsl -lgslcblas myprog.cc -o myprog,
```

where `CORALHOME` is the root CorAL directory. If you would prefer to recompile the source code along with your main program, you would add the line:

```
#include "coral.cc"
```

to your main routine rather than `coral.h`, and compile with the command:

```
g++ -I ${CORALHOME}/include -lgsl -lgslcblas myprog.cc -o myprog,
```

The codes in the `sample/` directory can be compiled with `make` (e.g. `make wfsample`) using the `makefile` found in that directory. The header file `coral.h` simply contain include statements to the remaining headers.

The result of the compilation will be a shared library file `libcoral.a` which will reside in the `lib` directory. The `make` command will also copy all header and source files from the `src/` tree into the `include/` directory (unless new versions already exist). If new source files are added to the `src` directory, the makefile must be edited. To make a new makefile, simply run the shell script:

```
makemaker.sh > makefile,
```

then change the `${OPT}` variable in the makefile if needed. In the makefile in the `samples/` directory, one can also recompile the source with command `make coral`, which changes to the `lib/` directory, compiles CorAL and returns to the original directory.

Not all source code found in the `src/` tree is compiled into CorAL, although any `*.cc` or `*.h` file will be used as a dependency in the makefile. If you add source files, they should be listed with an include statement in one of the following source and header files which are included in the compilation: `coral.cc coral.h misc.cc, misc.h, arrays.cc, arrays.h, sf.cc, sf.h, parametermap.cc, parametermap.h, wavefunctio wavefunction.h, source2CF.cc, source2CF.h, kernel.cc, kernel.h`. For instance, if you add a new source file, `wf_lambdaxi.cc`, which is used to calculate wave functions for Λs and Ξs, you would add a line to the file `wavefunction.cc` so that it would become:

```
#include "wavefunction.h"
#include "wfcommon.cc"
#include "planewave.cc"
#include "partwave.cc"
#include "pipi_phaseshifts.cc"
#include "wf_pp.cc"
#include "wf_pkplus.cc"
#include "wf_pipluspiplus.cc"
#include "wf_pipluspiminus.cc"
#include "wf_nn.cc"
#include "wf_lambdaxi.cc
```

In fact, there do exist several `wf_...` files in the `src/` tree which are not included as they are either in development or are untested.

10 Known Issues

GSL has a bug in the Coulomb partial wave routines for versions 1.7 and earler. The sign of the wave function for larger (¿3) values of $x = qr/\hbar$ oscillates between the correct and incorrect sign. The authors have provided a fix to the source code (see below). If you do not have this fix to **GSL**, you can edit the file `src/'/SpecialFunctions/CoulWave/coulwave.cc` and comment away the line `#define NO_GSLCOULWAVE_BUG`. If you want to change the **GSL** source file, then recompile, a bug fix was provided by the author and is described in the text file "gslbugfix.txt" which is included in `doc/`.

References

[1] M. A. Lisa, S. Pratt, R. Soltz and U. Wiedemann, arXiv:nucl-ex/0505014.