SPK Compiler Output (C++)

This documentation describes the source code files produced by SPK Compiler.

Table of Contents

Related Articles

- SpkSourceML[1]
- SpkDataML[2]
- SpkReportML[3]
- Pred Library[4]

`NonmemPars.h`

This header file exports a namespace, `NonmemPars`, defining variables that contain the values of NONMEM control parameters.

Namespace Entries

`const int nTheta`

is the length of *theta* vector.

`const valarray<double> thetaUp`

contains the upper boundary value for *theta*.

`const valarray<double> thetaLow`

contains the lower boundary value for *theta*.

`const valarray<double> thetaIn`

contains the initial estimate for *theta*.

`const valarray<bool> thetaFixed`

contains a vector of boolean flags specifying whether the corresponding i-th element of *theta* is fixed to the initial estimate value or not. `thetaFixed[i] =` true causes *theta(i+1)* to be fixed.

`const int nEta`

is the length of *eta* vector, which determines the dimension of *Omega*.

`const valarray<double> etaIn`

is the initial estimate for *eta*.

`const enum covStruct omegaStruct`

is present only if the *population analysis* is requested in the given SpkSourceML document. The value is either `DIAGONAL`, `FULL` or `BLOCKDIAG`. `DIAGONAL` indicates that only the diagonal elements of *Omega* matrix are subject to optimization. `FULL` indicates that potentially all elements are subject to optimization (but remember, *Omega* is symmetric).

`const enum covStruct omegaStruct`

is present only if the *individual analysis* is requested in the given SpkSourceML document. The value is always `DIAGONAL` for this type of analysis. `DIAGONAL` indicates that potentially all lower triangle elements are subject to optimization (remember, *Omega* is symmetric). `BLOCKDIAG` indicates that the *Omega* matrix is potentially made up of multiple `DIAGONAL` and `FULL` blocks.

`const int omegaDim`

is the dimension of *Omega* (symmetric) matrix. This value is determined by the length of *eta*.

`const int omegaOrder`

is the number of elements in *Omega* matrix that are subject to optimization (ie. the order of *Omega*).

`const valarray<double> omegaIn`

is the initial estimates for the elements of *Omega* that are subject to optimization. Thus, the length of `omegaIn` is `omegaOrder`.

`const valarray<bool> omegaFixed`

contains a vector of boolean values specifying whether the corresponding i-th element of *Omega* should be fixed to the initial estimate value or not. If Omega is 3 by 3 and only the diagonal elements are subject to optimization, setting `omegaFixed[2] = true` causes the (3,3) element of the matrix to be fixed.

`const int nOmegaBlk`

is the number of blocks in the block diagonal representation of *Omega*.

`const valarray<covStruct> omegaBlockStruct`

contains a vector of `covStruct` values specifying whether the corresponding i-th block is `DIAGONAL` or `FULL`. If *Omega* is made up of 3 blocks, setting `omegaBlockStruct[1] = FULL` causes the second block to be recorded as `FULL`.

`const valarray<int> omegaBlockdims`

contains a vector of integers specifying the dimension corresponding i-th block.

`const valarray<bool> omegaBlockSameAsPrev`

contains a vector of boolean values specifying whether the corresponding i-th block of *Omega* should be constrained equal to the previous block. If *Omega* is made up of 3 blocks, setting `omegaBlockSameAsPrev[3] = true` causes the third block to be constrained equal to the second.

`const int nEps`

is the length of *eps* vector, which determines the dimension of *Sigma*. This field is present only if the *population analysis* is requested.

`const enum covStruct sigmaStruct`

is present only if the *population analysis* is requested. The value is either `DIAGONAL`, `FULL` or `BLOCKDIAG`. `DIAGONAL` indicates that only the diagonal elements of *Sigma* matrix are subject to optimization. `FULL` indicates that potentially all lower triangle elements are subject to optimization (remember, *Sigma* is symmetric). `BLOCKDIAG` indicates that the *Sigma* matrix is potentially made up of multiple `DIAGONAL` and `FULL` blocks.

`const int sigmaDim`

is the dimension of *Sigma* (symmetric) matrix. ex. For a 3 by 3 matrix, the value here is 3. This field is present only if the *population analysis* is requested.

`const int sigmaOrder`

is the number of elements in *Sigma* matrix that are subject to optimization (ie. the order of *Sigma*). This field is present only if the *population analysis* is requested.

`const valarray<double> sigmaIn`

is the initial estimates for the elements of *Sigma* that are subject to optimization. Thus, the length of `simgaIn` is `sigmaOrder`. This field is present only if the *population analysis* is requested.

`const int nSigmaBlk`

is the number of blocks in the block diagonal representation of *Sigma*.

`const valarray<covStruct> sigmaBlockStruct`

contains a vector of `covStruct` values specifying whether the corresponding i-th block is `DIAGONAL` or `FULL`. If *Sigma* is made up of 3 blocks, setting `sigmaBlockStruct[1] = FULL` causes the second block to be recorded as `FULL`.

`const valarray<int> sigmaBlockdims`

contains a vector of integers specifying the dimension corresponding i-th block.

`const valarray<bool> sigmaBlockSameAsPrev`

contains a vector of boolean values specifying whether the corresponding i-th block of *Sigma* should be constrained equal to the previous block. If *Sigma* is made up of 3 blocks, setting `sigmaBlockSameAsPrev[3] = true` causes the third block to be constrained equal to the second.

`const int seed`

specifies the seed for random number generation. A value < 0 indicates no simulation to be done.

----- For ADVAN6 only from here -----

`const bool isPkFunctionofT`

inidicates as to whether the user's PK model is a function of T (continuous time variable). A value, true, indicates it does.

`const bool nCompartments`

specifies the number of compartments (including the output compartment that NONMEM implicitely adds).

`const bool nParameters`

specifies the number of PK parameters. Currently this is equal to the length of P vector variable appearing in the user's model.

`const bool defaultDoseComp`

specifies the default compartment that receives dose.

`const bool defaultObservationComp`

specifies the default observation compartment.

`const valarray<bool> initialOff`

is a vector of boolean values where the i-th value indicates as to whether the corresponding compartment is initiall off or not. True inidicates initially off.

`const valarray<bool> noOff`

is a vector of boolean values where the i-th value indicates as to whether the corresponding compartment can be turned off or not. True indicates no turn-off.

`const valarray<bool> noDose`

is a vector of boolean values where the i-th value indicates as to whehter the corresponding compartment would ever receive a dose or not. True indicates no dose at all.

`MontePars.h`

This header file exports a namespace, `MontePars` containing parameters that control Monte Carlo simulation. This file is generated only if *Monte Carlo simulation is requested* from the given SpkSourceML document.

`const enum { plain, grid, misser, analytic } method`

specifies the method for approximating (or computing) the integral.

NOTE: Do not assume the order of appearance of enum values to be the same as the one appeared in this document!

`plain`

Monte-Carlo approximation for integral

`analytic`

Closed form solution only valid for *LinearModel* (consult Brad Bell[5] for details)

`grid`

Approximation integral using evaluation on a uniform grid

`miser`

Approximation integral using Miser algorithm

`const int nEval`

is the number of components or the number of function evaluations. If method = grid, the value is taken as the number of random effects. Otherwise, the value must be one.

`const valarray<int> numberEval`

specifies the number of function evalautions. If method == grid, the length of the vector is equal to the number of random effects. numberEval[i] specifies the number of grid points in i-th random effect, and the corresponding total number of function evaluations is the product of the elements of numberEval. If method != grid, the length of the vector is 1 and numberEval[0] is taken as the total number of functions evaluations.

`IndData.h`

This header file exports the definition of a template class, `IndData`. `IndData` is a representation of the data set associated with *a single individual*.

The class object contains not only the individual's original data set but also placeholders for values computed during the process that correspond to each data record. The original data set values are declared read-only and others are most likely read-write. It also provides a number of methods for data operation.

Public Methods

`IndData( int nRecords, const vector<char*> &IDIn, const vector<double> &d1In, const vector<double> &d2In, ...)`

The constructor. The first argument, `nRecords` specifies the number of data records (for the individual). The arguments after following `nRecords` (ex. `IDIn`, `d1In`, `d2In`...) are vectors containing the values of data items from the data set such as ID, DV and TIME. Optional data items such as ID and MDV will appear in this list of arguments with their default values.

`~IndData()`

The destructor.

`int getNRecords() const`

Returns the total number of data records associated with this individual.

`int getNObservs() const`

Returns the number of data records whose corresponding MDV (Missing Data Value) value is 0 (false).

`const valarray<double> getMeasurements() const`

Returns a vector of measurements. i.e. The DV values of which corresponding MDV values are 0.

`int getMeasurementIndex( j ) const`

Returns the index to an element of the measurement vector, i.e. y, that corresponds to the measurement, i.e. DV, value of the j-th row of the original data set. If the MDV field value of the j-th row of the original data set were 1, indicating the data record does not contain a valid measurement value, i.e. DV, the value returned by this method is -1.

`int getRecordIndex( int j' ) const`

Returns the index to a record in the original data set from which the j'th element of the measurement vector, i.e. y, originated.

`void replaceMeasurements( const valarray<double> & yi )`

Replace the internally kept values of measurements with the values of `yi`.

`void replacePred ( const valarray<double> & predIn )`

Replace the internally kept values of PRED (prediction) with the values of `predIn`.

`void replaceRes( const valarray<double> & ResIn )`

Replace the internally kept values of RES (residual) with the values of `ResIn`.

`void replaceWRes( const valarray<double> & WResIn )`

Replace the internally kept values of WRES (weighted residual) with the values of `WResIn`.

`void replacePPred( const valarray<double> & pPredIn )`

Replace the internally kept values of PPRED (population prediction) with the values of `pPredIn`.

`void replacePRes( const valarray<double> & pResIn )`

Replace the internally kept values of PRES (population residual) with the values of `pResIn`.

`void replacePWRes( const valarray<double> & pWResIn )`

Replace the internally kept values of PWRES (population weighted residual) with the values of `pWResIn`.

`void replaceIPred( const valarray<double> & iPredIn )`

Replace the internally kept values of IPRED (individualized prediction) with the values of `iPredIn`.

`void replaceIRes( const valarray<double> & iResIn )`

Replace the internally kept values of IRES (individualized residual) with the values of `iResIn`.

`void replaceIWres( const valarray<double> & iWResIn )`

Replace the internally kept values of IWRES (individualized weighted residual) with the values of `iWResIn`.

`void replaceCPred( const valarray<double> & cPredIn )`

Replace the internally kept values of CPRED (conditional prediction) with the values of `cPredIn`.

`void replaceCRes( const valarray<double> & cResIn )`

Replace the internally kept values of CRES (conditional residual) with the values of `cResIn`.

`void replaceCWRes( const valarray<double> & cWResIn )`

Replace the internally kept values of CWRES (conditional weighted residual) with the values of `cWResIn`.

`void replaceEta( const valarray<double> & etaIn )`

Replace the internally kept values of ETA (eta) with the values of `etaIn`.

`void replaceEtaRes( const valarray<double> & etaResIn )`

Replace the internally kept values of ETARES (eta residual) with the values of `etaResIn`.

`void replaceWEtaRes( const valarray<double< & etaWResIn )`

Replace the internally kept values of WETARES (weighted eta residual) with the values of `etaWResIn`.

`void replaceIEtaRes( const valarray<double> & iEtaResIn )`

Replace the internally kept values of IETARES (individualized eta residual) with the values of `iEtaResIn`.

`void replaceIWEtaRes( const valarray<double> & iEtaWResIn )`

Replace the internally kept values of IWETARES (individualized weighted eta residual) with the values of `iWEtaResIn`.

`void replacePEtaRes( const valarray<double> & pEtaResIn )`

Replace the internally kept values of PETARES (population eta residual) with the values of `pEtaResIn`.

`void replacePWEtaRes( const valarray<double> & pEtaWResIn )`

Replace the internally kept values of PWETARES (population weighted eta residual) with the values of `pEtaWResIn`.

`void replaceCEtaRes( const valarray<double> & cEtaResIn )`

Replace the internally kept values of CETARES (conditional eta residual) with the values of `cEtaResIn`.

`void replaceCWEtaRes( const valarray<double> & cEtaWResIn )`

Replace the internally kept values of CWETARES (conditional weighted eta residual) with the values of `cEtaWResIn`.

Public Properties

`const vector<T>` *`xXx`*

`xXx` is replaced exactly by the labels of user-given (read only) data items such as ID, TIME and DV. It has a length of `nRecords`.

`vector<T> yYy`

`yYy` is replaced exactly by variable names that appear on the left hand side of assignment statements in model definitions. It has a length of `nRecords`.

`vector< vector<T> > THETA`

A read-write vector of `n`, `nTheta`-dimensional vectors, where `n` is the number of data records for the indiviual and `nTheta` is the size of *theta* vector.

`vector< vector<T> > ETA`

A read-write vector of `n` `nEta`-dimensional vectors, where `n` is the number of data records for the indiviual and `nTheta` is the size of *eta* vector.

`vector< vector<T> > EPS`

A read-write vector that contains `n` `nEps`-dimensional vectors, where `n` is the number of data records for the indiviual and `nTheta` is the size of *eps* vector. *This vector is present only if the population analysis is requested.*

`DataSet.h`

This header file exports the definition of a template class, `DataSet`. `DataSet` class is a reprentation of an entire data set (the population size >= 1).

Public Methods

`DataSet<class T>::DataSet()`

Constructor. The type of the template argument is somewhat restrictive at this point. `T` has to be a concrete type of CppAD[6].

`~DataSet()`

Destructor

`int getPopSize() const`

Returns the number of individuals in the population.

`const valarray<double> getAllMeasurements() const`

Returns a vector of measurements, i.e. DV values of which corresponding MDV values are 0 in the original data set.

`int getMeasurementIndex( int j ) const`

Returns the index to an element of the measurement vector, i.e. y, that corresponds to the measurement, i.e. DV, value of the j-th row of the original data set. If the MDV field value of the j-th row of the original data set were 1, indicating the data record does not contain a valid measurement value, i.e. DV, the value returned by this method is -1.

`int getMeasurementIndex( int i, int j ) const`

Returns the index to an element of the i-th individual's measurement vector, i.e. y, that corresponds to the measurement, i.e. DV, value of the j-th row of the i-th individual's original data set. If the MDV field value of the j-th row of the original data set were 1, indicating the data record does not contain a valid measurement value, i.e. DV, the value returned by this method is -1.

`int getRecordIndex( int j' ) const`

Returns the index to a record in the original data set from which the j'th element of the measurement vector, i.e. y, originated.

`int getRecordIndex( int i, int j' ) const`

Returns the index to a record in the i-th individual's original data set from which the j'th element of the measurement vector, i.e. y, originated.

`void expand( const valarray<double> & trancated, valarray<double> & expanded ) const`

Map `trancated` that is in the measurement-oriented space to the original record-oriented space. The measurement-oriented space refers to the space in which only the data records that contain Dependent Variable values. Whereas, the record-oriented space refers to the original data set that also contains rows missing DVs.

`const valarray<int> getNObservs() const`

Returns a vector of numbers of obervation records. The i-th element of the vector indicates the number of measurements for the i-th individual, where 0 <= i < n and n is the number of individuals in the population.

`int getNObservs(int i) const`

Returns the number of obervation records for the i-th individual.

`const valarray<int> getNRecords() const`

Returns a vector of numbers of records. The i-th element of the vector indicates the number of record for the i-th individual, where 0 <= i < n and n is the number of individuals in the population.

`int getNRecords(int i) const`

Returns the number of records for the i-th individual.

`friend ostream & operator<<( ostream& o, const DataSet<T> A )`

Extracts the data set into o in the following format:

```
<presentation_data columns="10" rows="12">
   <data_labels>
      <label name="ID">
      <label name="TIME">
      <label name="DV" synonym="CP">
      ...
   </data_label>
   <row position="">
      <value type="double" ref="ID">
         1
      </value>
      <value type="double" ref="TIME">
         0.0
      </value>
      <value type="double" ref="DV">
         0.0
      </value>
      ...
   </row>
   <row position="">
      <value type="double" ref="ID">
         1
      </value>
      <value type="double" ref="TIME">
         1.0
```

```
            </value>
            <value type="double" ref="DV">
               3.0
            </value>
            ...
         </row>

      </presentation_data>
```

The order in which <label>s in <data_labels> appear is arbitrary. However, the <value>s in a <row> are guaranteed to be listed in the same order as <label>s.

Pred.h

`Pred.h` defines a template class, `Pred`, which is derived from an abstract class, `PredBase`. The purpose of this class is to provide a facility to execute the user's PRED model.

```
template <class spk_ValueType>
Pred : public PredBase
{
   public:
      Pred( const DataSet<spk_ValueType>* dataIn );
      ~Pred(){}
      virtual bool eval( int spk_thetaOffset, int spk_thetaLen,
                         int spk_etaOffset,   int spk_etaLen,
           int spk_epsOffset,   int spk_epsLen,
           int spk_fOffset,     int spk_fLen,
           int spk_yOffset,     int spk_yLen,
           int spk_i,
           int spk_j,
                         int & spk_m,
           const vector<spk_ValueType> & spk_indepVar,
           vector<spk_ValueType>       & spk_depVar ) = 0;
      int getNObservs( int who ) const;
      int getNRecords( int who ) const;
      int getMeasurementIndex( int who, int recordIndex ) const;
      int getMeasurementIndex( int recordIndex ) const;
      int getRecordIndex( int who, int measurementIndex ) const;
      int getRecordIndex( int measurementIndex ) const;
}
```

Methods to Implement

`Pred` must implement a pure virtual function of its super class PredBase[7], `eval()`.

Constructor

Arguments

`const DataSet<spk_ValueType>* dataIn`

A pointer to a DataSet object.

`virtual bool eval()`

`eval()` function evaluates the $PRED model at the evaluation point corresponding to the i-th individual's j-th data record.

For the complete specification of `eval()`, consult PredBase Specifiction[8].

Requirements

- The user's model definition which is in a psudo FORTRAN language shall retain the case-insensitive attribute when it is translated to C++.
- Prefix `spk_` is added to all non-user variables in order to avoid a conflict with user variable names. A non-user variable in this sense is a variable created by the system (i.e. non-user). Whereas, a user-variable is a variable that appear in the model definition.

Return Value

The function returns `true` if the MDV for the current data record is 0. `false` otherwise.

Arguments

`const int spk_thetaOffset`

is the index to the head of *theta* vector within `spk_indepVar`.

`const int spk_thetaLen`

is the length of *theta* vector. The vector elements are assumed to be placed from `spk_indepVar[spk_thetaOffset]` to `spk_indepVar[spk_theetaOffset + spk_thetaLen]`.

`const int spk_etaOffset`

is the index to the head of *eta* vector within `spk_indepVar`.

`const int spk_etaLen`

is the length of *eta* vector. The vector elements are assumed to be placed from `spk_indepVar[spk_etaOffset]` to `spk_indepVar[spk_etaOffset + spk_etaLen]`.

`const int spk_epsOffset`

is the index to the head of *eps* vector within `spk_indepVar`.

`const int spk_epsLen`

is the length of *eps* vector. The vector elements are assumed to be placed from `spk_indepVar[spk_thetaOffset]` to `spk_indepVar[spk_theetaOffset + spk_theetaLen]`.

`const int spk_fOffset`

is the index to the element in `spk_depVar` in which the prediction for the j-th data record for the i-th individual, F (in NONMEM's term), shall be placed.

`const int spk_fLen`

is the number of measurements for the i-th individual.

`const int spk_yOffset`

is the index to the element in `spk_depVar` in which the error model value for the j-th data record for the i-th individual, Y (in NONMEM's term), shall be placed.

`const int spk_yLen`

is the number of measurments for the i-th individual..

`const int spk_i`

is the index to the i-th individual within the population (0 indicates the first individual).

`const int spk_j`

is the index to the j-th data record of the i-th individual.

`const int &spk_m`

will be replaced by the index to the observation record that corresponds to the j-th data record of the i-th individual if the MDV is 0. The value is unspecified for MDV=1.

`const vector<T> & spk_indepVar`

is the vector containing independent variables: *theta*, *eta* and *eps*.

`vector<T> & spk_depVar`

is an output in which the predicted value for the i-th individual's j-th data record will be placed at the `spk_yOffset`-th element. The `spk_yOffset`-th elements will be are replaced by the error model value for the i-th individual's j-th data record.

`int getNObservs( int i ) const`

Returns the number of obervations records for the i-th individual.

`int getNRecords( int i ) const`

Returns the number of total records for the i-th individual.

`int getMeasurementIndex( int j ) const`

Returns the index to an element of the measurement vector, i.e. y, that corresponds to the measurement, i.e. DV, value of the j-th row of the original data set. If the MDV field value of the j-th row of the original data set were 1, indicating the data record does not contain a valid measurement value, i.e. DV, the value returned by this method is -1.

`int getMeasurementIndex( int i, int j ) const`

Returns the index to an element of the i-th individual's measurement vector, i.e. y, that corresponds to the measurement, i.e. DV, value of the j-th row of the i-th individual's original data set. If the MDV field value of the j-th row of the original data set were 1, indicating the data record does not contain a valid measurement value, i.e. DV, the value returned by this method is -1.

`int getRecordIndex( int j ) const`

Returns the index to a record in the original data set from which the j'th element of the measurement vector, i.e. y, originated.

`int getRecordIndex( int i, int j ) const`

Returns the index to a record in the i-th individual's original data set from which the j'th element of the measurement vector, i.e. y, originated.

`OdePred.h`

`OdePred.h` defines a template class, `OdePred`, which is dervied from an abstract class, OdePredBase[9]. The purpose of this class is to provide a facility to execute the User's ODE model.

```
template <class spk_ValueType>
class OdePred : public OdePredBase<spk_ValueType>
{
public:
   OdePred( const DataSet<spk_ValueType>*  dataIn,
            int                            nPopSizeIn,
            bool                           isPkFunctionOfTIn,
            int                            nCompartmentsWithOutputIn,
            int                            nParametersIn,
            int                            defaultDoseCompIn,
            int                            defaultObservationCompIn,
            const std::valarray<bool>&     initialOffIn,
            const std::valarray<bool>&     noOffIn,
            const std::valarray<bool>&     noDoseIn,
            double                         tolRelIn
   );

  ~OdePred();

  int getNObservs( int who ) const;

  int getNRecords( int who ) const;

  const spk_ValueType lininterp( const string & devVar );

  virtual void initUserEnv( int spk_thetaOffset, int spk_thetaLen,
                            int spk_etaOffset,   int spk_etaLen,
                            int spk_epsOffset,   int spk_epsLen,
                            int spk_fOffset,     int spk_fLen,
                            int spk_yOffset,     int spk_yLen,
                            int spk_i,
                            int spk_j,
                            const vector<spk_ValueType>& spk_indepVar,
                            vector<spk_ValueType>      & spk_depVar );

  virtual void saveUserEnv( int spk_thetaOffset, int spk_thetaLen,
                            int spk_etaOffset,   int spk_etaLen,
                            int spk_epsOffset,   int spk_epsLen,
                            int spk_fOffset,     int spk_fLen,
                            int spk_yOffset,     int spk_yLen,
                            int spk_i,
                            int spk_j,
                            const std::vector<spk_ValueType>& spk_indepVar,
                            const std::vector<spk_ValueType>& spk_depVar );

  virtual void evalError(   int spk_thetaOffset, int spk_thetaLen,
                            int spk_etaOffset,   int spk_etaLen,
                            int spk_epsOffset,   int spk_epsLen,
                            int spk_i,
                            int spk_j,
                            const std::vector<spk_ValueType>& spk_indepVar);
  virtual void evalError();
```

```
  virtual void evalPk(      int thetaOffset, int thetaLen,
                            int etaOffset,   int etaLen,
                            int spk_i,
                            int spk_j,
                            const std::vector<spk_ValueType>& spk_indepVar );
  void evalPk( const spk_ValueType & t );

  virtual void evalDes(     int thetaOffset,     int thetaLen,
                            int spk_i,
                            int spk_j,
                            const std::vector<spk_ValueType>& spk_indepVar );
  void evalDes(             const spk_ValueType & t,
                            typename vector<spk_ValueType>::const_iterator a );
}
```

Methods to Implement

`OdePred(...)`

Arguments

`const DataSet<spk_ValueType> * dataIn`

A pointer to a DataSet object.

`int nPopSizeIn`

The population size (> 0).

`bool isPkFunctionOfTIn`

The value of `true` indicates that the PK model is a function of T (i.e. continuous time variable).

`int nCompartmentsWithOutputIn`

The number of compartments, including the output compartment.

`int nParametersIn`

The number of PK parameters.

Note to Developer: The meaning of this parameter is not quite understood. Currently the value is not used by the system.

`int defaultDoseCompIn`

The integer value indicates the default dose compartment (>=1). For instance, the value of `1` indicates the first compartment is given doses.

`int defaultObservationCompIn`

The integer value indicates the default observation compartment (>=1). For instance, the value of `1` indicates the first compartment is taken observations.

`const valarray<bool> & initialOffIn`

An array of boolean values. If `initialOffIn[i]` were true, (i+1)-th compartment is considered initially OFF.

`const valarray<bool> & noOffIn`

An array of boolean values. If `noOffIn[i]` were true, (i+1)-th compartment is considered that it will be never turned off.

`const valarray<bool> & noDoseIn`

An array of boolean values. If `noDoseIn[i]` were true, (i+1)-th compartment is considered that it will be never receive doses.

`double tolRelIn`

The relative tolerance.

`~OdePred()`

Destructor

`int getNObservs( int who ) const`

The function returns the number of measurements for the individual indicated by `who` Measurements means DV values whose corresponding MDV values are 0.

`int getNRecords( int who ) const`

The function returns the number of data records for the individual indicated by `who`.

`const spk_ValueType lininterp( const string & devVar )`

`lininterp` linearly interpolates `devVar` at the current T (i.e. continuous time variable).

`virtual void initUserEnv(...)`

This function initializes the user's space so that their models are evaluated properly at a given evaluation point. The specifications for the arguments are the same as of `Pred::eval()`[10].

`virtual void saveUserEnv(...)`

This function saves the status of user's environment. The specifications for the arguments are the same as of `Pred::eval()`[11].

`virtual void evalError(...)`

This is a deplicated version of `evalError()`.

`void evalError()`

Evaluates the user's error model at the current state of the object.

Note to developers: Once the deplicated version is removed from OdePred[12] class, this function should be come "virtual".

`virtual void evalPk(...)`

This is a deplicated version of `evalPk()`.

`void evalPk( const spk_ValueType & t )`

Evaluates the user's error model at `t` and the current state of the object.

Note to developers: Once the deplicated version is removed from OdePred[13] class, this function should be come "virtual".

`virtual void evalDes(...)`

This is a deplicated version of `evalDes()`.

`void evalOde( const spk_ValueType& t, typename vector<spk_ValueType>::const_iterator a )`

Evaluates the user's differential equation model at `t`, `a` and the current state of the object.

Note to developers: Once the deplicated version is removed from OdePred[14] class, this function should be come "virtual".

`fitDriver.cpp`

This is the SPK job driver (for parameter estimation, statistics and simulation). When you run `Makefile.SPK` generated together by SPK Compiler, this file builds to an executable named `driver`.

```
Usage: driver FORCE_WARM_START

FORCE_WARM_START --- Forces the job to start from the status found in "checkpoint.xml"
```

Input File

`checkpoint.xml`

When the contents of this file is loaded into the job driver, the optimization will start from the status saved in the file (i.e. warm-start).

There are two scenarios where this file is loaded into the job driver:

- This file is found in the working directory *and* the user has requested the warm start.
- This file is found in the working directory *and* the first command line argument to this driver is *1*. The argument value overrides the user's request.

Return value

0

Successful/normal completion.

1

Normal completion for unknown problem(s).

2

Abnormal completion for unknown failures. This shall result in submitting a bugzilla report.

10

Normal completion but some known problem was detected during a file/directory access

12

Normal completion but some known prolem was detected during optimization.

13

Normal completion but some known problem was detected during statisitcs calculation.

14

Normal completion but some known user input error was detected.

15

Normal completion but some known programmer's error was detected.

16

Normal completion but some known problem was detected during data simulation.

100

Abnormal completion due to a unknown system failures. This shall result in submitting a bugzilla report.

101

The value, 101, is reserved for C++ compilation error.

102

Abnormal completion due to a unknown optimization failure. This shall result in submitting a bugzilla report.

103

Abnormal completion due to a unknown statisitcs calculation failure. This shall result in submitting a bugzilla report.

104

Abnormal completion due to some unknown user input error.

105

Abnormal completion due to some programmer's error.

106

Abnormal completion due to a unknown data simulation fatal failure. This shall result in submitting a bugzilla report.

File Output

`checkpoint.xml`

The optimizer's state information from the last successful iteration is saved in this file. This file is always generated as long as fitDriver executes an iteration of optimization.

Screen Output

Standard Error

Plain error messages that are directed to the standard error and cannot be caught by the driver normally still go to the standard error. Such messages may include an error generated as a rusult of violation of an assertion statement.

Standard Output

The optimizer's tracing information which is generated for each iteration of optimization is directed to the standard output as long as tracing level is requested to be greater than 0 by the user. At the very end of the execution, an exit value is also printed in the following format: `exit code = INT`, where `INT` is one of the return values defined above.

`monteDriver.cpp`

This is a driver for a post-optimality process (ex. Monte Carlo).

Return value

0

Successful/normal completion.

10

Normal completion but some known problem was detected during file/directory access

100

Abnormal completion due to a unknown file/directory access failure.

200

Abnormal completion due to a known post-optimality failure.

7

Abnormal completion for other known errors.

300

Abnormal completion due to a unknown post-optimality failure.

8

Abnormal completion for other unknown failures.

`Makefile.SPK`

The primary goals of this Make file is to define rules to build two versions of executables. One is an executable hooked to the production version of libraries and the other is to the test version of libraries. An executable is either for the parameter optimization or for the Monte Carlo integration.

Targets

proc *(default)*

Compile SPK-Compiler-generated C++ source code files, link to *production* libraries (ie. libspk, libspkpred, and libopt) and build a driver, named `driver`. The *production* libraries are expected to be found in `/usr/local/lib/spkprod/`.

test

Compile SPK-Compiler-generated C++ source code files, link to *test* libraries (ie. libspk, libspkpred, and libopt) and build a driver, named `driver`. The *production* libraries are expected to be found in `/usr/local/lib/spktest/`.

clean

Delete all artifacts generated by SPK Compiler except for itself, `Makefile.SPK`.

Source Code Files

Common source

The source code files commonly needed by the two processes: optimization and Monte Carlo. The files are expected to be found in the current directory.

- NonmemPars.h
- IndData.h
- DataSet.h
- Pred.h

Optimization-specific source

The source code file only needed by the optimization process. The file is expected to be found in the current directory.

- fitDriver.cpp

Monte Carlo-specific source

The source code files only needed by the Monte Carlo process.

The following header is expected be found in the current directory.

- MontePars.h

The following files are expected to be found in `/usr/local/src/spktest/ml/`.

- monteDriver.cpp
- AnalyticalIntegral.h
- AnalyticalIntegral.cpp
- GridIntegral.h
- GridIntegral.cpp
- GridIntegral.h
- MapBay.h
- MapBay.cpp
- MapBay.h
- MontePopObj.h
- MontePopObj.cpp

Notes

1. ../sourceML/sourceML.html
2. ../dataML/dataML.html
3. ../reportML/reportML.html
4. http://192.168.2.2:8080/soft/v0.1/specs/spkpredLib/hierarchy.html
5. mailto:brad@apl.washington.edu
6. http://www.coin-or.org/CppAD/
7. http://192.168.2.2:8080/soft/v0.1/specs/spkpredLib/classPredBase.html
8. http://192.168.2.2:8080/soft/v0.1/specs/spkpredLib/classPredBase.html#a0
9. http://192.168.2.2:8080/soft/v0.1/specs/spkpredLib/classOdePredBase.html
10. http://192.168.2.2:8080/soft/v0.1/specs/spkpredLib/classPredBase.html#a0
11. http://192.168.2.2:8080/soft/v0.1/specs/spkpredLib/classPredBase.html#a0
12. http://192.168.2.2:8080/soft/v0.1/specs/spkpredLib/classOdePredBase.html
13. http://192.168.2.2:8080/soft/v0.1/specs/spkpredLib/classOdePredBase.html
14. http://192.168.2.2:8080/soft/v0.1/specs/spkpredLib/classOdePredBase.html