

MassInf Programmers' Notes

Version 2.30

John Skilling

Maximum Entropy Data Consultants Ltd

April 1999

	<i>Section</i>	<i>Index</i>	<i>Page</i>
1	Language	ANSI C	2
2	Formalism	MarkovChainMonteCarlo	2
2.1	Object	Natoms Coords Fluxes Valency Widths Blur	2
2.2	Prior	Alpha0 MaxAtoms CellNum CellWidth	3
2.3	Data	Ndata Data Acc	4
2.4	Response	Footprint	5
2.5	Likelihood	Chisq Chibar Noise Mockbar Evidence	5
3	DataAtom	ibits zbits nbits CheckAtom DataExch	6
4	Inferences	Infer ZeroInfer NextInfer MakeInfer Nsample	6
5	Control Parameters	Iseed ENSEMBLE Niter	7
6	Diagnostics	UserWrite Progress Limit Write	8
7	Error States	E\... CHECK	8
8	Precision		9
9	Program Supply	mcmc treelink coollink random peano2d peano3d	10
	Appendix	mcmc.h header file	11

1 Language

The internal programming language is ANSI C, selected because it has a defined standard, and because C++ extensions would be of only slight programming advantage. The user can compile with either ANSI C or C++.

2 Formalism

Massive Inference (MassInf) is a probabilistic procedure that uses Bayes' theorem

$$\underbrace{\text{Prob}(f)}_{\text{Prior}} \times \underbrace{\text{Prob}(D|f)}_{\text{Likelihood}} = \underbrace{\text{Prob}(D)}_{\text{Evidence}} \times \underbrace{\text{Prob}(f|D)}_{\text{Inference}} \quad (1)$$

INPUTS \rightarrow OUTPUTS

to infer an object f from noisy and (usually) incomplete data D . It generates both the posterior inference for the object, and the value of the evidence for that inference. The posterior is produced as a set of samples of f , calculated with an exploratory Markov chain Monte Carlo (MCMC) algorithm.

The outer procedure is named `MarkovChainMonteCarlo`, and this calls various user procedures that specify the details of the current application.

2.1 Object

The object being observed is modelled by a variable number N of ‘atoms’ or ‘point masses’, each of which has an amplitude or ‘flux’ z and a positional coordinate x . Thus the object is taken to be

$$f(x) = \sum_{j=0}^{N-1} z_j \sigma(x - x_j). \quad (2)$$

For example, MassInf is applicable to various forms of spectroscopy, where the object is naturally described by a set of discrete ‘lines’. For a two-dimensional example, the object might be made up of superposed points or patches of known shape.

Within the MassInf program, the object is encoded as

```
int      Natoms;    //Number of atoms N
unsigned* Coords;   //Positions x[0...N-1]
double*  Fluxes;    //Amplitudes z[0...(Valency*N-1)]
unsigned* Widths;   //Positional widths +/-dx[0...N-1]
```

As a technical refinement, the atoms have a common ‘valency’, being the number of associated amplitudes z .

```
int      Valency;   //Number of fluxes associated with one atom
```

Usually, the valency is unity, but higher values are allowed provided the responses of an atom’s various amplitudes do not overlap. For example, a coloured photograph might have three (red, green, blue) data channels responding to the appearance of atoms having different colours described by red, green and blue intensities; in this case the valency would be 3, and the fluxes would appear in z in order $(R, G, B)_{\text{atom}\#0}, \dots, (R, G, B)_{\text{atom}\#N-1}$; the convention is that valency rasters fastest.

Atomic positions are coded as ‘unsigned’ integers rather than ‘float’ or ‘double’ because having a finite number of a-priori equivalent locations has technical advantages. In most computers nowadays, the natural number of locations is 2^{32} .

The user must interpret these 2^{32} locations appropriately. If the object is one-dimensional (like a spectrum), then the positions x will map directly onto the appropriate range, being appropriately bunched up or spread out proportionate to any a-priori expectation of non-uniformity. If the object is multi-dimensional, the user must map the single coordinate x into the appropriate space. Because the MassInf algorithm runs more efficiently if atoms of neighbouring location x appear similarly in the data, it is recommended that the mapping preserve locality. For example, a square domain of $2^{16} \times 2^{16}$ cells might be covered by a 2^{32} -element Peano curve.

MassInf can also be used to display the object f itself (this being a special case of inference). However, there is a difficulty in that each sample of f is a set of delta functions, at locations that will nearly always differ in detail. Very many samples, usually well in excess of 2^{32} , may be needed before the average $\langle f \rangle$ takes its final locally smooth appearance on the 2^{32} cell computational grid. This is likely to be impractical. To circumvent this difficulty, MassInf uses an extra parameter

```
double    Blur;        //Dimensionless width coefficient
```

which lets each atom in a sample be given an appropriate width, proportionate to the local accumulated density of atoms with respect to their unsigned coordinates. These widths are output in the `Widths` which accompany the `Coords`, and enable a smooth display. Typically, a value `Blur=0.3` seems to be reasonable. `Blur=0` switches off the smoothing. `Blur=1` is likely to prove excessive. Note: these widths are only a presentational device.

2.2 Prior

The object f is described by its parameters N , x , z , each of which must be given its assignment of prior probability.

N , the number of atoms, can be assigned a Poisson distribution with given mean α , so that

$$\text{Prob}(N) = \frac{e^{-\alpha} \alpha^N}{N!}, \quad (3)$$

with prior mean and standard deviation

$$N = \alpha \pm \sqrt{\alpha}. \quad (4)$$

Alternatively, if the prior value of N cannot be judged with such relative precision (α usually being considerably greater than unity), N can be assigned a wider (geometric) distribution

$$\text{Prob}(N) = (1 - r)r^N, \quad r = \frac{a}{(a + 1)}, \quad (5)$$

with prior mean and standard deviation

$$N = a \pm \sqrt{a(a + 1)}. \quad (6)$$

Within the MassInf program, this ‘complexity’ hyperparameter is supplied as

```
double    Alpha0;     //alpha (if +ve); else -a (if -ve)
```

It is also possible to impose a maximum number of atoms through

```
unsigned MaxAtoms; //Usually 2^32 (coded as 0)
```

This modifies the Poisson distribution to binomial, and the geometric distribution similarly.

Each of the N locations x_0, \dots, x_{N-1} is assigned a uniform prior

$$\text{Prob}(x_i) = \text{constant (usually } 2^{-32}\text{)}. \quad (7)$$

Technically, the locations are grouped into cells, and the user can specify both the number (modulo 2^{32}) and the width of the cells, subject to the product not exceeding 2^{32} .

```
unsigned CellNum; //Usually 2^32 (coded as 0)
unsigned CellWidth; //Usually 1, must be > 0.
```

At most one atom can be placed in any cell, so that there is an intrinsic upper limit on the number of atoms. This very slightly modifies the formulae given above for the prior on N , as documented in the MassInf source code.

Each of the N amplitudes z_0, \dots, z_{N-1} is assigned an exponential prior

$$\text{Prob}(z_i) = \frac{\exp(-z_i/q)}{q}. \quad (8)$$

Here, the parameter q is sampled internally from a distribution designed to accommodate to the current problem. This parameter is automatic, and not under user control.

2.3 Data

There are (say) M data values D , each associated with a corresponding standard deviation uncertainty σ . These uncertainties are assumed to be Gaussian and uncorrelated: any correlation should preferably be removed beforehand by replacing the data with appropriate (singular vector) combinations. Thus the data are

$$\text{Data} = (D_k \pm \sigma_k), \quad k = 0 \dots M - 1. \quad (9)$$

Within MassInf, the data are encoded as

```
int      Ndata;    //Number of data M
double*  Dat;     //Data values D[0...M-1]
double*  Acc;     //Accuracy= (1/sigma)[0...M-1]
```

Reciprocal standard deviations are used to enable total uncertainty with infinite standard deviation to be supplied without overflow. Any such ‘un-measured’ data are irrelevant to the inference and are discarded from both the calculation and the resulting evidence value: they count towards `Ndata` because of addressing, but they do not count towards parameter M in any analysis.

2.4 Response

Any position x has an associated response $R(x)$, being the mock data that would be observed (apart from noise) if the object were simply a single unit amplitude at x . The mock data associated with a complete object of N atoms is thus

$$F_k = \sum_{j=0}^{N-1} z_j R_k(x_j). \quad (10)$$

Within `MassInf`, a user-defined data structure located at

```
DataStr* Footprint;
```

contains whatever ancillary information is needed to construct the response function $R(x)$.

2.5 Likelihood

The difference between the mock data and the actual data is measured by the χ^2 statistic

$$\chi^2 = \sum_{k=0}^{M-1} \frac{(F_k - D_k)^2}{\sigma_k^2}. \quad (11)$$

This statistic controls the likelihood function that mediates the effect of the data upon the computations. The likelihood also includes a normalisation constant

$$Z = \prod_{k=0}^{M-1} \sqrt{2\sigma_k^2}. \quad (12)$$

If the scale of the standard deviation uncertainties is known, the likelihood is

$$\text{Prob}(D|f) = Z^{-1} \exp(-\chi^2/2). \quad (13)$$

Otherwise, with the overall uncertainty scale unknown, it is

$$\text{Prob}(D|f) = \frac{Z^{-1}}{\sqrt{2}} \Gamma(M/2) (\chi^2/2)^{-M/2} \quad (14)$$

where $\Gamma(u) = (u-1)!$ is the gamma function.

Within `MassInf`, likelihood quantities appear as

```
double   Chisq;    //current value chi^2(f)
double*  Chibar;   //chi^2 value averaged over posterior inference
double*  Noise;    //Std.dev. rescaling factor (if required)
double*  Mockbar;  //Average mock data [0...M-1]
```

If `Noise` is supplied as a `NULL` address, the standard deviation uncertainties are taken as known and there is no rescaling. Otherwise, the `Noise` address receives the rescaling factor and `Chibar` receives the number of measured data M

$$\begin{aligned} \text{Chibar} &= M, \\ \text{Noise} &= \sigma_{\text{estimated}}/\sigma_{\text{supplied}}. \end{aligned} \quad (15)$$

Finally, the vector `Mockbar` receives the mock data $\langle F \rangle$, averaged over the posterior inference. The evidence value (as its natural logarithm) is sent to

```
double*  Evidence; //log[e] Prob(Data)
```

3 DataAtom

`DataAtom` is the mandatory user-supplied procedure that calculates the mock data footprint $R(x)$ from an atom of unit strength at a given location. Its specification is

```
int      DataAtom( // + = OK, 0 = DoNotUse, - = error
unsigned coord,   //Input atom location
DataStr* Footprint, //Input definition of response
int*     ibits,   //Fragment coords
double*  zbits,   //Fragment quantities
int*     nbits); //Number of fragments
```

`DataAtom` should pick out from the given `Footprint` the number (`nbits`) of fragments of mock data that are produced from the given location (`coord`). Usually, `DataAtom` should signal success by returning a positive value. If, for some reason, the given location should not have an atom at all (this is a stronger condition than merely not interacting with the data so that `nbits=0`), `DataAtom` should return 0. Assuming success, `DataAtom` should return the corresponding mock data indices in the vector `ibits[0...nbits-1]` and the corresponding strengths in `zbits[0...nbits-1]`. In other words, the mock data contribution from unit amplitude at the supplied location would be

$$F(\text{ibits}[i]) = \text{zbits}[i] \quad \text{for } i = 0 \dots \text{nbits} - 1. \quad (16)$$

The indices returned in `ibits` must, of course, be in the range $[0 \dots \text{Ndata} - 1]$ of the data themselves. (If the valency exceeds 1, the extra strength vector(s) should be appended as if the dimensioning were `zbits[nbits][Valency]`.)

The optional user functions

```
int      CheckAtom(); //FOR ADVANCED USE
int      DataExch();  //FOR ADVANCED USE
```

should ordinarily be set to `NULL`. `CheckAtom` is used if particular distributions of flux within an atom's valency are to be prohibited. `DataExch` can be used to suggest which atoms and their valencies are usefully close, if the ordinary one-dimensional topology through the integer coordinate is inadequate.

4 Inferences

`MassInf` is primarily designed to infer the statistics of properties of the object f . These properties, together with their variances or other statistics, are accumulated in a user-defined structure located at

```
InferStr* Infer;
```

Before the first sample of f is generated, `MassInf` calls the user-defined procedure

```
ZeroInfer(Infer);
```

to ensure that any accumulators can be properly zeroed.

At the end of the computation (and within it, if internal storage becomes exhausted), `MassInf` sends the successive samples of f to the user by calling the user-defined procedure

```
NextInfer(Infer, Natoms, Coords, Fluxes, Widths);
```

once for each sample. A call to `NextInfer` is made with the atoms in increasing positional order, as it happens. These calls enable the user to accumulate whatever statistics are needed. Typically, they are sums, and sums of squares,

$$\sum \text{property}(f) \quad \text{and} \quad \sum \text{property}(f)^2 \quad (17)$$

(If the valency exceeds 1, extra amplitudes are inserted as if the dimensioning were `Fluxes[Natoms][Valency]`.) A smooth display can also be accumulated by plotting the atoms of each sample as smooth curves of given `Widths`, normalised to the given `Fluxes`.

At the end of the computation, `MassInf` notifies the user by calling the user-defined procedure

```
MakeInfer(Infer, Nsample);
```

This reminds the user how many samples have been generated

```
int      Nsample; //Number of samples
```

and enables the accumulated properties to be properly presented. Typically, `MakeInfer` translates sums and sums of squares into means and standard deviations

$$\langle \text{property}(f) \rangle > \pm \sqrt{\text{variance}(\text{property})}. \quad (18)$$

An accumulated smooth display would be reduced to the mean $\langle f \rangle$ by dividing it by `Nsample`.

The user can opt not to produce any inferences by supplying `NULL` addresses for the functions `ZeroInfer`, `NextInfer` and `MakeInfer`. In that case, the calls will not be made, the functions need not be supplied, and the `InferStr` structure need not be set up.

5 Control Parameters

`MassInf` uses random numbers to govern its probabilistic exploration of plausible objects f . The random number generator is seeded by

```
int      Iseed;
```

which can take any integer value. If this value is positive or zero, the computation will be reproducible. If `Iseed` is given a negative value, the generator is instead seeded by some positive value determined by the current clock time, so that repeated computations will differ. Comparison of the results from such runs affords a rough way of assessing convergence. The `MarkovChainMonteCarlo` procedure returns the positive generator seed as its value, so that any particular run can always be reproduced, even if `Iseed` was set negative.

The `MassInf` algorithm can hold several samples simultaneously, each evolving by Markov chain Monte Carlo. This number is supplied in

```
int      ENSEMBLE;
```

ENSEMBLE=1 may often be sufficient, but larger values give potentially greater power because an ensemble of several members may escape from a position that could trap an individual member.

The amount of computation time to be devoted to each ensemble member is supplied in

```
int      Niter;
```

There is no adequate mathematical guidance to the value of `Niter`, so the user should be guided by experience of like problems in supplying a value sufficiently large to make convergence plausible.

6 Diagnostics

As a calculation proceeds, `MassInf` sends diagnostics by calling the user-defined procedure `UserWrite`, whose specification is

```
int      UserWrite(  
double   Progress, //Progress of calculation phase  
double   Limit,    //Progress limit of calculation phase  
int      Natoms,   //Current number of atoms  
double   Chisq,    //Current Chisquared value  
WriteStr* Write); //User's diagnostic pointers
```

The calculation proceeds in two phases. In the first ‘burn-in’ phase, the exploration attempts to arrive within the general area of plausible objects f . This phase is characterised by a `Progress` value that rises from 0 to its current `Limit` value of 1. Technically, `Progress` gives the reciprocal annealing temperature during burn-in.

In the second ‘equilibration’ phase, the algorithm attempts to explore the plausible objects of high posterior probability. This phase is characterised by a `Progress` value that counts iterations up to a maximum `Limit` value governed by the time taken to burn-in.

At each call, diagnostics `Natoms` and `Chisq` are available. Generally, `Natoms` should rise and `Chisq` should fall toward their equilibrium values during burn-in, and should oscillate around these values during exploration.

The user-defined structure located at

```
WriteStr* Write;
```

enables the user to provide diagnostic destination information, allied to any other information that may be helpful.

The user can opt to ignore these diagnostics by supplying a `NULL` address for the function `UserWrite`. In that case, the call will not be made, the function need not be supplied, and the `WriteStr` structure need not be set up.

7 Error States

Each procedure is of type `int`, and must return a defined value, often just 0. If the return value is negative, it is interpreted as an error code. `MassInf` will then immediately abort,

returning to its calling program with this value as its own (`int`) return value. Otherwise the `MarkovChainMonteCarlo` procedure will return the non-negative generator seed on successful completion.

`MassInf` itself can return a negative code if it detects a system error. There are several such error states (plus any that might be returned by the programmer's application). All are fatal.

`E_MALLOC` (=-130) Unable to allocate sufficient memory

This suggests an attempt to run an application with inadequate resources.

`E_MCMC_NUMBER` (=-200) `CellNum * CellWidth` exceeds 2^{32}

This indicates an out-of-range combination of parameters.

`E_MCMC_FRAGS` (=-201) More than `Ndata` fragments (nbits)

`E_MCMC_RANGE` (=-202) ibits outside `[0,..,Ndata-1]`

`E_MCMC_VALENCY` (=-203) Valency footprints overlap

`E_MCMC_EXCH` (=-204) Too many fragments from `DataExch`

These indicate a wrongly programmed user procedure `DataAtom` (or `DataExch`).

`E_MCMC_FLUX` (=-205) Bad prediction of flux

`E_MCMC_CHISQ` (=-206) Bad prediction of chisquared

`E_MCMC_RESID` (=-207) Bad prediction of mock data residuals

These indicate a surprising loss of arithmetical precision. Exceptionally the cause might be innocent, but a commoner cause would be irreproducible output from the user procedures.

`E_MCMC_BIRTH` (=-208) System error, new atom failure

`E_MCMC_WORK` (=-209) System error, workspace corrupted

`E_MCMC_ATOM` (=-210) System error, wrong atom accepted

`E_TREEDATA` (=-261) System error, internal addressing

These system errors should never occur, and would indicate a system bug.

`E_RAN_ARITH` (=-299) Integer precision less than 32 bits

`E_PEANO` (=-444) Integer precision not 32 bits

The pseudo-random and Peano procedures check that integer precision is appropriate.

Because testing can be time-consuming, not all these conditions are checked. By default, `MarkovChainMonteCarlo` is supplied with a checking flag

```
static int CHECK=0;
```

switched OFF (zero). Full checking can be enabled by editing this to any non-zero value. The programmer may wish to debug an application in this stricter environment.

8 Precision

The `MassInf` program partly relies on the accumulation of small changes, so that all its calculations are carried out in double precision. In many applications, it may be possible to use single-precision transforms, but that is at the programmer's risk.

Some of the internal code requires integers to be stored in the usual two's-complement form.

9 Program Supply

MassInf is supplied to users as a suite of kernel programs

```
mcmc.[ch]    //Contains MarkovChainMonteCarlo
treelink.[ch] //Called by MarkovChainMonteCarlo
coollink.[ch] //Called by MarkovChainMonteCarlo
random.[ch]  //Called by MarkovChainMonteCarlo
```

The pseudo-random procedures in random.c are available for external use, if needed.
Peano curve procedures

```
peano2d.[ch] //Available for user application
peano3d.[ch] //Available for user application
```

are also supplied as an aid to 2- and 3-dimensional applications. They translate between a one-dimensional 32-bit coordinate and its 2- or 3-dimensional counterpart.

A particular application is defined by the suite of user programs that must include

```
userstr.h    //Defines DataStr,InferStr,ShowStr,WriteStr
```

The other user programs can be organised arbitrarily. As supplied, they comprise

```
userapp.[ch] //User application programs DataAtom (etc.)
applypgm.[ch] //Application shell
toy.c        //Main program for I/O
```

Finally

```
toy.dat
```

supplies a simple internally-documented test dataset formatted for this ‘toy’ application,
and

```
toy.log
```

gives the output. Note that different compilers and/or hardware may give results that differ in detail, because minor arithmetic errors amplify as exploration proceeds, even though long-term averages should remain stable.

Appendix: mcmc.h header file

```
/*+++++
*           Massive Inference
*
* Filename:  mcmc.h
*
* Purpose:   Header for mcmc.c
*
* History:   JS    22 Jan 1998, 16 Feb 1998, 6 Dec 1998,
*             23 Feb 1999, 27 Apr 1999, 24 May 1999
*
*           Copyright (c) 1997-1999 Massive Inference Techniques Ltd.
*-----
*/
#ifndef MCMCH
#define MCMCH
#include "userstr.h"      /* declare DataStr, InferStr, WriteStr */

extern int MarkovChainMonteCarlo(
        /* 0 +ve seed for random generator */

        int DataAtom( /* Mandatory */ /* 0 +ve = OK */
                    /* 0 = DO NOT USE this coord */
                    /* -ve = ERROR */
                    unsigned coord, /* I atom coordinate */
                    DataStr* Footprint, /* I definition of data footprint */
                    int* ibits, /* 0 fragment coords */
                    double* zbits, /* 0 fragment quantities */
                    int* nbits), /* 0 # fragments >= 0 */

        int CheckAtom( /* Optional */ /* 0 0 = DO NOT USE atom, else = OK */
                    DataStr* Footprint, /* I definition of data footprint */
                    int Valency, /* I # fluxes associated with atom */
                    const double* Fluxes), /* I fluxes associated with atom */

        int DataExch( /* Optional */ /* 0 +ve = # suggestions (0 or 1) */
                    unsigned coord, /* I 32-bit atom coordinate */
                    DataStr* Footprint, /* I definition of data footprint */
                    int* vatom, /* 0 orig valencies for suggest exch */
                    int* vexch, /* 0 exch valencies for suggest exch */
                    unsigned* xexch, /* 0 target coords for suggest exch */
                    int NEXCH), /* I MAX # suggestions */

        int ZeroInfer( /* Optional */
                    InferStr* Infer), /* 0 Initialised user integrals */

        int NextInfer( /* Optional */
                    InferStr* Infer, /* I 0 Running totals of user integrals*/
                    int Natoms, /* I # atoms in list */
                    const unsigned* Coords, /* I List of atom coordinates */
                    const double* Fluxes, /* I List of atom fluxes */

```

```

const unsigned* Widths),          /* I    List of atom widths */

int MakeInfer( /* Optional */
  InferStr* Infer,                /* I 0  Finalised user integrals */
  int      Nsample),             /* I   # samples in running totals */

int UserWrite( /* Optional */
  double   Progress,             /* I   Progress of calculation phase */
  double   Limit,                /* I   Progress limit of calc phase */
  int      Natoms,               /* I   # atoms */
  double   Chisq,                /* I   Chisquared */
  WriteStr* Write),             /* I   User's diagnostic pointers */

  int      Ndata,               /* I   # data */
const double* Data,             /* I   Data values [Ndata] */
const double* Acc,              /* I   Accuracies [Ndata] */
  double*   Mockbar,            /* (0) <Mock data> [Ndata] */

  int      ENSEMBLE,            /* I   # ensemble members */
  int      Niter,               /* I   # iterative cycles */
  int      Iseed,               /* I   Random seed */
  int      Valency,             /* I   # fluxes per atom */
  unsigned CellWidth,           /* I   Cell Coord width > 0; Cell has 0,1 atoms */
  unsigned CellNum,             /* I   # cells (0 means 2^32); Num*Width <= 2^32 */
  unsigned MaxAtoms,           /* I   Max # atoms (0 means infinity) */
  double   Alpha0,             /* I   Complexity (+ve = fixed, -ve = autoscaled)*/
  double   Blur,                /* I   Atomic blurring coefficient for display */

  DataStr* Footprint,           /* I   Atom responses */
  InferStr* Infer,              /* (0) User integrals */
  WriteStr* Write,              /* (0) User diagnostics */

  double*   Noise,              /* (0) Std.dev. noise estimate (NULL means x1) */
  double*   Chibar,             /* (0) <Chisquared> (rel. to Noise) */
  double*   Atoms,              /* (0) <# atoms> */
  double*   Evidence);          /* (0) log[e] Pr(Data) */

#undef  E_MCMC_NUMBER
#define E_MCMC_NUMBER -200 /* User error: CellNum * CellWidth > 2^32 */
#undef  E_MCMC_FRAGS
#define E_MCMC_FRAGS -201 /* User error: more than Ndata fragments */
#undef  E_MCMC_RANGE
#define E_MCMC_RANGE -202 /* User error: ibits[.] outside [0,Ndata) */
#undef  E_MCMC_VALENCY
#define E_MCMC_VALENCY -203 /* User error: valency footprints overlap */
#undef  E_MCMC_EXCH
#define E_MCMC_EXCH -204 /* User error: more than NEXCH fragments */
#undef  E_MCMC_FLUX
#define E_MCMC_FLUX -205 /* Numerical error: bad prediction of flux */
#undef  E_MCMC_CHISQ
#define E_MCMC_CHISQ -206 /* Numerical error: bad prediction of chisq */
#undef  E_MCMC_RESID
#define E_MCMC_RESID -207 /* Numerical error: bad prediction of resid */
#undef  E_MCMC_BIRTH

```

```
#define E_MCMC_BIRTH    -208 /* System error: birth prohibited */
#undef  E_MCMC_WORK
#define E_MCMC_WORK     -209 /* System error: workspace corrupted */
#undef  E_MCMC_ATOM
#define E_MCMC_ATOM     -210 /* System error: wrong atom accepted */
#undef  E_MCMC_MAXATOMS
#define E_MCMC_MAXATOMS -211 /* System error: too many atoms */

#endif
```