Numerical Integration Module
1. Vector function. This is vector_function.h file
   This file defines interface for functions taking a vector as an input and returning a vector (with a different size) as an output. We are also introducing a parameter signalling: it is a boolean vector which tracks parameters which were changed from the previous call. The VectorFunction implementation can exploit this information and evaluate the function more efficiently. The information can be completely ignored.
   From the signalling reason, and from other reasons, the function evaluation is not const.

```
#ifndef VECTOR_FUNCTION_H
#define VECTOR_FUNCTION_H
#include "Vector.h"
#include "GeneralMatrix.h"
#include <vector>

ParameterSignal class declaration 2;
VectorFunction class declaration 3;
VectorFunctionSet class declaration 4;
GaussConverterFunction class declaration 5);
#endif
```

2. This is a simple class representing a vector of booleans. The items might be retrieved or changed, or can be set true after some point. This is useful when we multiply the vector with lower triangular matrix. True means that a parameter was changed.

```
ParameterSignal class declaration 2} ≡
class ParameterSignal {
  protected:
    bool *data;
    int num;

class public:
  ParameterSignal(int n);
  ParameterSignal(const ParameterSignal &sig);
  ~ParameterSignal() {
    delete[] data; }
  void signalAfter(int l);
  const bool &operator[](int i) const
  {
    return data[i]; }
  bool &operator[](int i)
  {
    return data[i]; }
};
```

This code is used in section 1.
This is the abstract class for vector function. At this level of abstraction we only need to know size of input vector and a size of output vector.

The important thing here is a clone method, we will need to make hard copies of vector functions since the evaluations are not \texttt{const}. The hardcopies apply for parallelization.

\begin{verbatim}
VectorFunction\{
protected:
  int in_dim;
  int out_dim;
public:
  VectorFunction(int idim, int odim) : in_dim(idim), out_dim(odim) {} 
  VectorFunction(const VectorFunction &func) : in_dim(func.in_dim), out_dim(func.out_dim) {} 
  virtual ~VectorFunction() {} 
  virtual VectorFunction *clone() const = 0;
  virtual void eval(const Vector &point, const ParameterSignal &sig, Vector &out) = 0;
  int indim() const 
  { return in_dim; } 
  int outdim() const 
  { return out_dim; }
\}
\end{verbatim}

This code is used in section 1.

This makes \( n \) copies of \texttt{VectorFunction}. The first constructor make exactly \( n \) new copies, the second constructor copies only the pointer to the first and others are hard (real) copies.

The class is useful for making a given number of copies at once, and this set can be reused many times if we need mupliple copis of the function (for example for parallelizing the code).

\begin{verbatim}
VectorFunctionSet\{
protected:
  std::vector(VectorFunction *) funcs;
  bool first_shallow;
public:
  VectorFunctionSet(const VectorFunction &f, int n);
  VectorFunctionSet(VectorFunction &f, int n);
  ~VectorFunctionSet();
  VectorFunction &getFunc(int i)
  { return *(funcs[i]); } 
  int getNum() const 
  { return funcs.size(); } 
\}
\end{verbatim}

This code is used in section 1.
5. This class wraps another VectorFunction to allow integration of a function through normally distributed inputs. Namely, if one wants to integrate
\[
\frac{1}{\sqrt{(2\pi)^n|\Sigma|}} \int f(x) e^{-\frac{1}{2}x^T \Sigma^{-1} x} dx
\]
then if we write \(\Sigma = AA^T\) and \(x = \sqrt{2}Ay\), we get integral
\[
\frac{1}{\sqrt{(2\pi)^n|\Sigma|}} \int f(\sqrt{2}Ay) e^{-y^T y} dy = \frac{1}{\sqrt{\pi^n}} \int f(\sqrt{2}Ay) e^{-y^T y} dy,
\]
which means that a given function \(f\) we have to wrap to yield a function
\[
g(y) = \frac{1}{\sqrt{\pi^n}} f(\sqrt{2}Ay).
\]
This is exactly what this class is doing. This transformation is useful since the Gauss–Hermite points and weights are defined for weighting function \(e^{-y^2}\), so this transformation allows using Gauss–Hermite quadratures seamlessly in a context of integration through normally distributed inputs.

The class maintains a pointer to the function \(f\). When the object is constructed by the first constructor, the \(f\) is not copied. If the object of this class is copied, then \(f\) is copied and we need to remember to destroy it in the deconstructor; hence delete_flag. The second constructor takes a pointer to the function and differs from the first only by setting delete_flag to true.

\((\text{GaussConverterFunction} \text{ class declaration 5}) \equiv\)
\[
\text{class GaussConverterFunction : public VectorFunction } \{
\text{protected:}
\text{ VectorFunction *func;}
\text{ bool delete\_flag;}
\text{ GeneralMatrix A;}
\text{ double multiplier;}
\text{public:}
\text{ GaussConverterFunction(VectorFunction \&f, const GeneralMatrix &vcov);}
\text{ GaussConverterFunction(VectorFunction *f, const GeneralMatrix &vcov);}
\text{ GaussConverterFunction(const GaussConverterFunction &f);}\]
\text{ virtual ~GaussConverterFunction()}
\text{ { if (delete\_flag) delete func; }\}
\text{ virtual VectorFunction *clone() const}
\text{ { return new GaussConverterFunction(*this); }\}
\text{ virtual void eval(const Vector &point, const ParameterSignal &sig, Vector &out);}
\text{private:}
\text{ double calcMultiplier() const;}
\text{ void calcCholeskyFactor(const GeneralMatrix &vcov);}\}
\]
This code is used in section 1.

6. End of vector_function.h file
7. This is `vector_function.cpp` file

```c
#include "vector_function.h"
#include <dynlapack.h>
#include <cmath>
#include <cstring>
#include <algorithm>
#ifdef __MINGW32__
#define __CROSS_COMPILATION__
#endif
#ifdef __MINGW64__
#define __CROSS_COMPILATION__
#endif
#define M_PI 3.14159265358979323846
```

This code is used in section 7.

8. Just an easy constructor of sequence of booleans defaulting to change everywhere.

```c
ParameterSignal constructor code 8
ParameterSignal copy constructor code 9
ParameterSignal::signalAfter code 10
VectorFunctionSet constructor 1 code 11
VectorFunctionSet constructor 2 code 12
VectorFunctionSet destructor code 13
GaussConverterFunction constructor code 1
GaussConverterFunction constructor code 2
GaussConverterFunction copy constructor code 16
GaussConverterFunction::eval code 17
GaussConverterFunction::multiplier code 18
GaussConverterFunction::calcCholeskyFactor code 19
```

9. This code is used in section 7.
10. This sets \textit{false} (no change) before a given parameter, and \textit{true} (change) after the given parameter (including).

\begin{verbatim}
(ParameterSignal::signalAfter code 10) ≡
  void ParameterSignal::signalAfter(int l)
  {
    for (int i = 0; i < std::min(l, num); i++) data[i] = false;
    for (int i = l; i < num; i++) data[i] = true;
  }
\end{verbatim}

This code is used in section 7.

11. This constructs a function set hardcopying also the first.

\begin{verbatim}
(VectorFunctionSet constructor 1 code 11) ≡
  VectorFunctionSet::VectorFunctionSet(const VectorFunction &f, int n)
  : funcs(n), first_shallow(false) {
    for (int i = 0; i < n; i++) funcs[i] = f.clone();
  }
\end{verbatim}

This code is used in section 7.

12. This constructs a function set with shallow copy in the first and hard copies in others.

\begin{verbatim}
(VectorFunctionSet constructor 2 code 12) ≡
  VectorFunctionSet::VectorFunctionSet(VectorFunction &f, int n)
  : funcs(n), first_shallow(true) {
    if (n > 0) funcs[0] = &f;
    for (int i = 1; i < n; i++) funcs[i] = f.clone();
  }
\end{verbatim}

This code is used in section 7.

13. This deletes the functions. The first is deleted only if it was not a shallow copy.

\begin{verbatim}
(VectorFunctionSet destructor code 13) ≡
  VectorFunctionSet::~VectorFunctionSet()
  {
    unsigned int start = first_shallow ? 1 : 0;
    for (unsigned int i = start; i < funcs.size(); i++) delete funcs[i];
  }
\end{verbatim}

This code is used in section 7.

14. Here we construct the object from the given function \( f \) and given variance-covariance matrix \( \Sigma = \text{vcov} \).

The matrix \( A \) is calculated as lower triangular and yields \( \Sigma = AA^T \).

\begin{verbatim}
(GaussConverterFunction constructor code 14) ≡
  GaussConverterFunction::GaussConverterFunction(VectorFunction &f, const GeneralMatrix &vcov)
  : VectorFunction(f), func(&f), delete_flag(false), A(vcov.numRows(), vcov.numRows()),
    multiplier(calcMultiplier()) { /* todo: raise if A.numRows() != indim() */
    calcCholeskyFactor(vcov);
  }
\end{verbatim}

This code is used in section 7.
15. Here we construct the object in the same way, however we mark the function as to be deleted.

\[
\text{GaussConverterFunction}\text{ constructor code 2 15} \equiv \\
\text{GaussConverterFunction}::\text{GaussConverterFunction}(\text{VectorFunction } f, \text{ const GeneralMatrix } \& vcv) \\
\text{ : VectorFunction } (f), \text{ func } (f), \text{ delete }\text{ flag}(true), \text{ A}(vcov.\text{ numRows }() , vcv.\text{ numRows }()), \\
\text{ multiplier } (\text{calcMultiplier }()) \{ /\ast \text{ todo: raise if } A.\text{ numRows }() \neq \text{ indim }() */ \\
\text{calcCholeskyFactor } (vcov); \\
\}
\]
This code is used in section 7.

16. \[
\text{GaussConverterFunction}\text{ copy constructor code 16} \equiv \\
\text{GaussConverterFunction}::\text{GaussConverterFunction}(\text{const GaussConverterFunction } \& f) \\
\text{ : VectorFunction } (f), \text{ func } (f.\text{func-clone }()), \text{ delete }\text{ flag}(true), \text{ A}(f.A), \text{ multiplier } (f.\text{multiplier }()) \} \\
\]
This code is used in section 7.

17. Here we evaluate the function \( g(y) = \frac{1}{\pi n} f(\sqrt{2} Ay) \). Since the matrix \( A \) is lower triangular, the change signal for the function \( f \) will look like \((0, \ldots, 0, 1, \ldots, 1)\) where the first 1 is in the same position as the first change in the given signal \( \text{sig} \) of the input \( y = \text{point} \).

\[
\text{GaussConverterFunction}\text{::eval code 17} \equiv \\
\text{void GaussConverterFunction}::\text{eval}(\text{const Vector } \& \text{point}, \text{ const ParameterSignal } \& \text{sig}, \text{Vector } \& \text{out}) \\
\{ \\
\text{ ParameterSignal } s(\text{sig}); \\
\text{ int } i = 0; \\
\text{ while } (i < \text{indim }() \land \neg \text{sig}[i]) \ i++; \\
\text{ s.signalAfter } (i); \\
\text{ Vector } x(\text{indim }()); \\
\text{ x.zeros }(); \\
\text{ A.multaVec } (x, \text{point}); \\
\text{ x.mult } (\text{sqrt } (2.0)); \\
\text{ func=eval } (x, s, \text{out}); \\
\text{ out.mult } (\text{multiplier }); \\
\}
\]
This code is used in section 7.

18. This returns \( \frac{1}{\sqrt{\pi n}} \).

\[
\text{GaussConverterFunction}\text{::multiplier code 18} \equiv \\
\text{double GaussConverterFunction}::\text{calcMultiplier }() \text{ const } \\
\{ \text{ return } \text{sqrt } (\text{pow } (\text{M_PI}, -1 \ast \text{indim }())); \}
\]
This code is used in section 7.
19. \( \text{GaussConverterFunction}::\text{calcCholeskyFactor} \) code 19

```cpp
void GaussConverterFunction::calcCholeskyFactor(const GeneralMatrix &vcov)
{
    A = vcov;
    lapack_int rows = A.numRows();
    for (int i = 0; i < rows; i++)
        for (int j = i + 1; j < rows; j++) A.get(i, j) = 0.0;
    lapack_int info;
    dpotrf("L", &rows, A.base(), &rows, &info);  // todo: raise if info ≠ 1 */
}
```

This code is used in section 7.

20. End of `vector_function.cpp` file

21. Quadrature. This is `quadrature.h` file

This file defines an interface for one dimensional (non-nested) quadrature `OneDQuadrature`, and a parent for all multi-dimensional quadratures. This parent class `Quadrature` presents a general concept of quadrature, this is

\[
\int f(x)dx \approx \sum_{i=1}^{N} w_i x_i
\]

The class `Quadrature` just declares this concept. The concept is implemented by class `QuadratureImpl` which paralelizes the summation. All implementations therefore wishing to use the parallel implementation should inherit from `QuadratureImpl` and integration is done.

The integration concept relies on a point iterator, which goes through all \( x_i \) and \( w_i \) for \( i = 1, \ldots, N \). All the iterators must be able to go through only a portion of the set \( i = 1, \ldots, N \). This enables us to implement parallellism, for two threads for example, one iterator goes from the beginning to the (approximately) half, and the other goes from the half to the end.

Besides this concept of the general quadrature, this file defines also one dimensional quadrature, which is basically a scheme of points and weights for different levels. The class `OneDQuadrature` is a parent of all such objects, the classes `GaussHermite` and `GaussLegendre` are specific implementations for Gauss–Hermite and Gauss–Legendre quadratures resp.

```cpp
#ifndef QUADRATURE_H
#define QUADRATURE_H
#include <cstdlib>
#include "vector_function.h"
#include "int_sequence.h"
#include "sthread.h"
⟨OneDQuadrature class declaration 22⟩;
⟨Quadrature class declaration 23⟩;
⟨IntegrationWorker class declaration 24⟩;
⟨QuadratureImpl class declaration 26⟩;
⟨OneDPrecalcQuadrature class declaration 29⟩;
⟨GaussHermite class declaration 30⟩;
⟨GaussLegendre class declaration 31⟩;
⟨NormalICDF class declaration 32⟩;
#endif
```
22. This pure virtual class represents a concept of one-dimensional (non-nested) quadrature. So, one dimensional quadrature must return number of levels, number of points in a given level, and then a point and a weight in a given level and given order.

\begin{verbatim}
class OneDQuadrature {
  public:
    virtual ~OneDQuadrature() {};
    virtual int numLevels() const = 0;
    virtual int numPoints(int level) const = 0;
    virtual double point(int level, int i) const = 0;
    virtual double weight(int level, int i) const = 0;
  }
\end{verbatim}

This code is used in section 21.

23. This is a general concept of multidimensional quadrature. At this general level, we maintain only a dimension, and declare virtual functions for integration. The function take two forms; first takes a constant \texttt{VectorFunction} as an argument, creates locally \texttt{VectorFunctionSet} and do calculation, second one takes as an argument \texttt{VectorFunctionSet}.

Part of the interface is a method returning a number of evaluations for a specific level. Note two things: this assumes that the number of evaluations is known a priori and thus it is not applicable for adaptive quadratures, second for Monte Carlo type of quadrature, the level is a number of evaluations.

\begin{verbatim}
class Quadrature {
  protected:
    int dim;
  public:
    Quadrature(int d) : dim(d) {};
    virtual ~Quadrature() {};
    int dimen() const {
      return dim;
    }
    virtual void integrate(const VectorFunction &func, int tn, Vector &out) const = 0;
    virtual void integrate(VectorFunctionSet &fs, int level, Vector &out) const = 0;
    virtual int numEvals(int level) const = 0;
  }
\end{verbatim}

This code is used in section 21.
24. This is just an integration worker, which works over a given QuadratureImpl. It also needs the function, level, a specification of the subgroup of points, and output vector.

See (QuadratureImpl class declaration 26) for details.

IntegrationWorker class declaration 24 ≡

\[
\text{template\{typename _Tpit\} class QuadratureImpl;}
\]

\[
\text{template\{typename _Tpit\} class IntegrationWorker : public THREAD \{}
\]

\[
\quad \text{const QuadratureImpl\{_Tpit\} &quad;}
\]

\[
\quad \text{VectorFunction &func;}
\]

\[
\quad \text{int level;}
\]

\[
\quad \text{int ti;}
\]

\[
\quad \text{int tn;}
\]

\[
\quad \text{Vector &outvec;}
\]

\[
\}
\]

Public:

\[
\text{IntegrationWorker(const QuadratureImpl\{_Tpit\} &q, VectorFunction &f, int l, int tii, int tnn, Vector &out) \{}
\]

\[
\quad \\text{(IntegrationWorker::operator\{(\}) code 25);}\}
\]

This code is used in section 21.

25. This integrates the given portion of the integral. We obtain first and last iterators for the portion (beg and end). Then we iterate through the portion. and finally we add the intermediate result to the result outvec.

This method just everything up as it is coming. This might be imply large numerical errors, perhaps in future I will implement something smarter.

IntegrationWorker::operator\{(\}) code 25 ≡

\[
\text{void operator\{(\})}
\]

\[
\quad \_Tpit \ \text{beg} = \quad \text{quad.begin\{ti, tn, level\};}
\]

\[
\quad \_Tpit \ \text{end} = \quad \text{quad.begin\{ti + 1, tn, level\};}
\]

\[
\quad \text{Vector \ tmpall(outvec.length\{\});}
\]

\[
\quad \text{tmpall.zeros\{\};}
\]

\[
\quad \text{Vector \ tmp(outvec.length\{\}); \quad \text{\/* note that since beg came from begin, it has empty signal */}
\]

\[
\quad \text{\quad \text{\/* and first evaluation gets no signal */}}
\]

\[
\quad \text{for (_Tpit \ run = \ \text{beg}; \ run \neq \ \text{end}; \ ++\text{run}) \{}
\]

\[
\quad \quad \text{func.eval\{run.point\{\}, run.signal\{\}, tmp\};}
\]

\[
\quad \quad \text{tmpall.add\{run.weight\{\}, tmp\};}
\]

\[
\quad \}\}
\]

\[
\quad \text{SYNCHRO \ syn(&outvec, "IntegrationWorker");}
\]

\[
\quad \text{outvec.add(1.0, tmpall);}\}
\]

This code is used in section 24.
26. This is the class which implements the integration. The class is templated by the iterator type. We declare a method `begin` returning an iterator to the beginning of the `ti`-th portion out of total `tn` portions for a given level.

In addition, we define a method which saves all the points to a given file. Only for debugging purposes.

\[
\text{template\(\text{<typename}_{\text{Tpit}}\)} \text{ class QuadratureImpl : public Quadrature } \\
\text{ friend class IntegrationWorker\(\_\)Tpit)}; \\
\]

public:

\[
\text{QuadratureImpl\(\text{(int } d) \text{ \{} \text{ } \}} \\
\text{(QuadratureImpl\text{::integrate code 27}) } \\
\text{void integrate\(\text{(const VectorFunction } &\text{func, int level, int tn, Vector } &\text{out) const} \\
\text{\{VectorFunctionSet fs\(\text{(func, tn)}\);} \\
\text{integrate\(\text{(fs, level, out);}} \} \) \\
\text{(Quadrature::savePoints code 28)}; \\
\_\text{Tpit start\(\text{(int level) const} \\
\text{\{ return begin\(\text{(0, 1, level);}} \} \\
\_\text{Tpit end\(\text{(int level) const} \\
\text{\{ return begin\(\text{(1, 1, level);}} \} \\
\text{protected:} \\
\text{virtual } \_\text{Tpit begin\(\text{(int ti, int tn, int level) const = 0;} \}} \\
\); \\
\]

This code is cited in section 24.

This code is used in section 21.

27. Just fill a thread group with workers and run it.

\[
\text{(QuadratureImpl\text{::integrate code 27}) \equiv} \\
\text{void integrate\(\text{(VectorFunctionSet } &fs, \text{int level, Vector } &\text{out) const} \\
\text{\{ /* todo: out.length()==func.outdim() */ /* todo: dim == func.indim() */} \\
\text{out.zeros();} \)}} \\
\text{THREAD\_GROUP gr; } \\
\text{for \(\{\text{int ti = 0; ti < fs.getNum(); ti++}\} \{} \\
\text{gr.insert\(\text{(new IntegrationWorker\(\_\)Tpit\(\text{(*this, fs.getFunc(ti), level, ti, fs.getNum(), out)\))};} \\
\text{gr.run();} \} \\
\]

This code is used in section 26.

```c
§ Numerical Integration Module

⟨ Quadrature :: savePoints code 28 ⟩ ≡
void savePoints(const char *fname, int level) const
{
    FILE *fd;
    if (! fname)) { /* todo: raise */
        fprintf(stderr,"Cannot open file %s for writing.
        exit(1);
    }
    Tpit beg = begin(0, 1, level);
    Tpit end = begin(1, 1, level);
    for (Tpit run = beg; run != end; ++run) {
        fprintf(fd, "%16.12g
        fprintf(fd, "%16.12g", run.weight());
        for (int i = 0; i < dimen(); i++) fprintf(fd, "%16.12g", run.point()[i]);
    }
    fclose(fd);
}
```

This code is used in section 26.

29. This is only an interface to a precalculated data in file precalc_quadrature.dat which is basically 
C coded static data. It implements OneDQuadrature. The data file is supposed to define the following 
data: number of levels, array of number of points at each level, an array of weights and array of points. The 
both latter array store data level by level. An offset for a specific level is stored in offsets integer sequence. 
The implementing subclasses just fill the necessary data from the file, the rest is calculated here.

```c
⟨ OneDPreCalcQuadrature class declaration 29 ⟩ ≡
class OneDPreCalcQuadrature : public OneDQuadrature {
    int num_levels;
    const int *num_points;
    const double *weights;
    const double *points;
    IntSequence offsets;
    public:
    OneDPreCalcQuadrature(int nlevels, const int *npoints, const double *wts, const double *pts) :
        num_levels(nlevels), num_points(npoints), weights(wts), points(pts), offsets(num_levels) {
            calcOffsets();
    }
    virtual ~OneDPreCalcQuadrature() {}
    int numLevels() const
    { return num_levels; }
    int numPoints(int level) const
    { return num_points[level - 1]; }
    double point(int level, int i) const
    { return points[offsets[level - 1] + i]; }
    double weight(int level, int i) const
    { return weights[offsets[level - 1] + i]; }
    protected:
        void calcOffsets();
    }
```

This code is used in section 21.
30. Just precalculated Gauss–Hermite quadrature. This quadrature integrates exactly integrals
\[ \int_{-\infty}^{\infty} x^k e^{-x^2} \, dx \]
for level \( k \).

Note that if plugging this one-dimensional quadrature to product or Smolyak rule in order to integrate a function \( f \) through normally distributed inputs, one has to wrap \( f \) to GaussConverterFunction and apply the product or Smolyak rule to the new function.

Check precalc_quadrature.dat for available levels.

\[ \text{GaussHermite class declaration} \]
\begin{verbatim}
class GaussHermite : public OneDPreCalcQuadrature {
 public:
   GaussHermite();
};
\end{verbatim}

This code is used in section 21.

31. Just precalculated Gauss–Legendre quadrature. This quadrature integrates exactly integrals
\[ \int_{0}^{1} x^k \, dx \]
for level \( k \).

Check precalc_quadrature.dat for available levels.

\[ \text{GaussLegendre class declaration} \]
\begin{verbatim}
class GaussLegendre : public OneDPreCalcQuadrature {
 public:
   GaussLegendre();
};
\end{verbatim}

This code is used in section 21.

32. This is just an inverse cumulative density function of normal distribution. Its only method \( \text{get} \) returns for a given number \( x \in (0, 1) \) a number \( y \) such that \( P(z < y) = x \), where the probability is taken over normal distribution \( N(0, 1) \).

Currently, the implementation is done by a table lookup which implies that the tails had to be chopped off. This further implies that Monte Carlo quadratures using this transformation to draw points from multidimensional \( N(0, I) \) fail to integrate with satisfactory precision polynomial functions, for which the tails matter.

\[ \text{NormalICDF class declaration} \]
\begin{verbatim}
class NormalICDF {
 public:
   static double get(double x);
};
\end{verbatim}

This code is used in section 21.

33. End of quadrature.h file
34. This is `quadrature.cpp` file.

```cpp
#include "quadrature.h"
#include "precalc_quadrature.dat"
#include <cmath>

(OneDPrecalcQuadrature::calcOffsets code 35); 
(GaussHermite constructor code 36); 
(GaussLegendre constructor code 37); 
(NormalICDF get code 38);

35. (OneDPrecalcQuadrature::calcOffsets code 35) ≡
    void OneDPrecalcQuadrature::calcOffsets()
    {
        offsets[0] = 0;
        for (int i = 1; i < num_levels; i++) offsets[i] = offsets[i - 1] + num_points[i - 1];
    }
This code is used in section 34.

36. (GaussHermite constructor code 36) ≡
    GaussHermite::GaussHermite()
    : OneDPrecalcQuadrature(gh_num_levels, gh_num_points, gh_weights, gh_points) {}
This code is used in section 34.

37. (GaussLegendre constructor code 37) ≡
    GaussLegendre::GaussLegendre()
    : OneDPrecalcQuadrature(gl_num_levels, gl_num_points, gl_weights, gl_points) {}
This code is used in section 34.
38. Here we transform a draw from univariate \((0, 1)\) to the draw from Gaussina \(N(0, 1)\). This is done by a table lookup, the table is given by \(\text{normal}\_icdf\_step, \text{normal}\_icfd\_data, \text{normal}\_icdf\_num\), and a number \(\text{normal}\_icdf\_end\). In order to avoid wrong tails for lookups close to zero or one, we rescale input \(x\) by \((1 - 2 * (1 - \text{end})) = 2 * \text{end} - 1\).

\[
\text{NormalICDF} \equiv \text{double NormalICDF :: get(double x) }
\]

\[
\text{get(x) = (2 * normal\_icdf\_end - 1) * std::abs(x - 0.5);} \\
i = (\text{int}) \text{floor}(xx / normal\_icdf\_step);} \\
xx = normal\_icdf\_step * i; \\
yy = normal\_icdf\_data[i]; \\
y = \begin{cases} 
\text{yy1} + (yy2 - yy1) * (xx - xx1) / normal\_icdf\_step; & (i < normal\_icdf\_num - 1) \\
\text{yy1} & \text{else}
\end{cases}
\]

This code is used in section 34.

39. End of quadrature.cpp file

40. Product quadrature. This is product.h file

This file defines a product multidimensional quadrature. If \(Q_k\) denotes the one dimensional quadrature, then the product quadrature \(Q_{k}^{d}\) of \(k\) level and dimension \(d\) takes the form

\[
Qf = \sum_{i=1}^{n_k} \cdots \sum_{i_d=1}^{n_k} w_{i_1} \cdots w_{i_d} f(x_{i_1}, \ldots, x_{i_d})
\]

which can be written in terms of the one dimensional quadrature \(Q_k\) as

\[
Qf = (Q_k \otimes \ldots \otimes Q_k) f
\]

Here we define the product quadrature iterator \(\text{prodpit}\) and plug it into \textbf{QuadratureImpl} to obtains \textbf{ProductQuadrature}.

```cpp
#ifndef PRODUCT_H
#define PRODUCT_H
#include "int_sequence.h"
#include "vector_function.h"
#include "quadrature.h"

//prodpit class declaration

class ProductQuadrature

#endif
```
41. This defines a product point iterator. We have to maintain the following: a pointer to product quadrature in order to know the dimension and the underlying one dimensional quadrature, then level, number of points in the level, integer sequence of indices, signal, the coordinates of the point and the weight.

The point indices, signal, and point coordinates are implemented as pointers in order to allow for empty constructor.

The constructor \texttt{prodpit(const ProductQuadrature &q, int j0, int l)} constructs an iterator pointing to \((j0,0,...,0)\), which is used by \texttt{begin} dictated by \texttt{QuadratureImpl}.

\begin{verbatim}
This code is used in section 40.
\end{verbatim}
42. The product quadrature is just \texttt{QuadratureImpl} with the product iterator plugged in. The object is constructed by just giving the underlying one dimensional quadrature, and the dimension. The only extra method is \texttt{designLevelForEvals} which for the given maximum number of evaluations (and dimension and underlying quadrature from the object) returns a maximum level yielding number of evaluations less than the given number.

\begin{verbatim}
(ProductQuadrature class declaration 42) ≡
class ProductQuadrature : public QuadratureImpl(prodpit)
{
    friend class prodpit;
    const OneDQuadrature &uquad;

    public:
    ProductQuadrature(int d, const OneDQuadrature &uq);
    virtual ~ProductQuadrature() {};
    int numEvals(int l) const
    {
        int res = 1;
        for (int i = 0; i < dimen(); i++) res *= uquad.numPoints(l);
        return res;
    }
    void designLevelForEvals(int max_eval, int &lev, int &evals) const;

    protected:
    prodpit begin(int ti, int tn, int level) const;
};
\end{verbatim}

This code is used in section 40.

43. End of product.h file

44. This is product.cpp file.

\begin{verbatim}
#include "product.h"
#include "symmetry.h"

(prodpit empty constructor 45);
(prodpit regular constructor 46);
(prodpit copy constructor 47);
(prodpit destructor 48);
(prodpit :: operator≡ code 49);
(prodpit ::operator= code 50);
(prodpit ::operator++ code 51);
(prodpit :: setPointAndWeight code 52);
(prodpit :: print code 53);
(ProductQuadrature constructor 54);
(ProductQuadrature :: begin code 55);
(ProductQuadrature :: designLevelForEvals code 56);
\end{verbatim}

45. (prodpit empty constructor 45) ≡

\begin{verbatim}
prodpit :: prodpit()
    : prody(Λ), level(0), npoints(0), jseq(Λ), end_flag(true), sig(Λ), p(Λ) {};
\end{verbatim}

This code is used in section 44.
46. This constructs a product iterator corresponding to index \((j_0, 0\ldots, 0)\).

\[
\text{\texttt{prodpit}} \text{ regular constructor 46} ≡ \\
\text{\texttt{prodpit}}::\text{prodpit} (\text{const ProductQuadrature} &q, \text{int} j_0, \text{int} l)
\]

This code is used in section 44.

47. Copy constructor, clear.

\[
\text{\texttt{prodpit}} \text{ copy constructor 47} ≡ \\
\text{\texttt{prodpit}}::\text{prodpit} (\text{const prodpit} &ppit)
\]

This code is used in section 44.

48.

\[
\text{\texttt{prodpit}} \text{ destructor 48} ≡ \\
\text{\texttt{prodpit}}::\sim\text{prodpit}()
\]

This code is used in section 44.

49.

\[
\text{\texttt{prodpit}} \text{::operator≡ code 49} ≡ \\
\text{\texttt{bool prodpit}}::\text{operator≡(const prodpit} &ppit)\text{ const}
\]

This code is used in section 44.
50.  
\[ \text{⟨prodpit :: operator = code 50⟩ ≡} \]
\[ \text{const prodpit &prodpit :: operator = (const prodpit &ppit)} \]
\[
\begin{aligned}
\{ & \text{prodq = ppit.prodq;} \\
& \text{end_flag = ppit.end_flag;} \\
& \text{w = ppit.w;} \\
& \text{if (jseq) delete jseq;} \\
& \text{if (sig) delete sig;} \\
& \text{if (p) delete p;} \\
& \text{if (ppit.jseq) jseq = new IntSequence(*ppit.jseq);} \\
& \text{else jseq = Λ;} \\
& \text{if (ppit.sig) sig = new ParameterSignal(*ppit.sig);} \\
& \text{else sig = Λ;} \\
& \text{if (ppit.p) p = new Vector(*ppit.p);} \\
& \text{else p = Λ;} \\
& \text{return *this;} \\
\end{aligned}
\]
This code is used in section 44.

51.  
\[ \text{⟨prodpit :: operator ++ code 51⟩ ≡} \]
\[ \text{prodpit &prodpit :: operator ++ ()} \]
\[
\begin{aligned}
& \text{/\* todo: throw if prodq ≡ Λ or jseq ≡ Λ or sig ≡ Λ or end_flag ≡ true */} \\
& \text{int i = prodq-dimen() - 1;} \\
& \text{(*jseq)[i]++;} \\
& \text{while (i ≥ 0 ∧ (*jseq)[i] ≡ npoints) {} \{} \\
& \text{(*jseq)[i] = 0;} \\
& \text{i--;} \\
& \text{if (i ≥ 0) (*jseq)[i]++;} \\
& \text{}} \\
& \text{sig.signalAfter(std::max(i, 0));} \\
& \text{if (i ≡ -1) end_flag = true;} \\
& \text{if (¬end_flag) setPointAndWeight();} \\
& \text{return *this;} \\
\end{aligned}
\]
This code is used in section 44.

52.  
\[ \text{⟨prodpit :: setPointAndWeight code 52⟩ ≡} \]
\[ \text{void prodpit :: setPointAndWeight()} \]
\[
\begin{aligned}
& \text{/\* todo: raise if prodq ≡ Λ or jseq ≡ Λ or sig ≡ Λ or */} \\
& \text{w = 1.0;} \\
& \text{for (int i = 0; i < prodq-dimen(); i++) {} \{} \\
& \text{(*p)[i] = (prodq-uqquad).point(level, (*jseq)[i]);} \\
& \text{w *= (prodq-uqquad).weight(level, (*jseq)[i]);} \\
& \text{}} \\
\end{aligned}
\]
This code is used in section 44.
53. Debug print.

(prodpirit :: print code 53) ≡

void prodpirit :: print( ) const
{
    printf("j=[");
    for (int i = 0; i < prodq-dimen(); i++) printf("%2d␣", (*jseq)[i]);
    printf("%
    for (int i = 0; i < prodq-dimen() - 1; i++) printf("%+4.3f␣", (*p)[i]);
    printf("%+4.3f)\n", (*p)[prodq-dimen() - 1]);
}

This code is used in section 44.

54. (ProductQuadrature constructor 54) ≡

ProductQuadrature :: ProductQuadrature(int d, const OneDQuadrature &uq)
: QuadratureImpl(prodpirit)(d), uquad(uq) { /* todo: check d ≥ 1 */
}

This code is used in section 44.

55. This calls prodpirit constructor to return an iterator which points approximately at $ti$-th portion out of $tn$ portions. First we find out how many points are in the level, and then construct an iterator $(j0,0,...,0)$ where $j0 = ti * npoints / tn$.

(ProductQuadrature :: begin code 55) ≡

prodpirit ProductQuadrature :: begin(int ti, int tn, int l) const
{
    /* todo: raise is l < dimen() */
    /* todo: check l ≤ uquad.numLevels() */
    int npoints = uquad.numPoints(l);
    return prodpirit(*this, ti * npoints / tn, l);
}

This code is used in section 44.

56. This just starts at the first level and goes to a higher level as long as a number of evaluations (which is $n_k^d$ for $k$ being the level) is less than the given number of evaluations.

(ProductQuadrature :: designLevelForEvals code 56) ≡

void ProductQuadrature :: designLevelForEvals(int max_evals, int &lev, int &evals) const
{
    int last_evals;
    evals = 1;
    lev = 1;
    do {
        last_evals = evals;
        evals = numEvals(lev);
        while (lev < uquad.numLevels() - 2 ∧ evals < max_evals);
        lev --;
        evals = last_evals;
    } while (lev < uquad.numLevels() - 2 ∧ evals < max_evals);
}

This code is used in section 44.

57. End of product.cpp file
58. Smolyak quadrature. This is smolyak.h file

This file defines Smolyak (sparse grid) multidimensional quadrature for non-nested underlying one dimensional quadrature. Let $Q_1^l$ denote the one dimensional quadrature of $l$ level. Let $n_l$ denote a number of points in the $l$ level. Than the Smolyak quadrature can be defined as

$$Q^d f = \sum_{l \leq |k| \leq l+d-1} (-1)^{|k|+d-|k|-1} \binom{d-1}{|k|-l} (Q_{k_1}^1 \otimes \ldots \otimes Q_{k_d}^1) f,$$

where $d$ is the dimension, $k$ is $d$-dimensional sequence of integers, and $|k|$ denotes a sum of the sequence.

Here we define smolpit as Smolyak iterator and SmolyakQuadrature.

```c
#errordefine SMOLYAK_H
#define SMOLYAK_H
#include "int_sequence.h"
#include "tl_static.h"
#include "vector_function.h"
#include "quadrature.h"

(smolpit class declaration 59);
(SmolyakQuadrature class declaration 60);
#errorendif
```
Here we define the Smolyak point iterator. The Smolyak formula can be broken to a sum of product quadratures with various combinations of levels. The iterator follows this pattern. It maintains an index to a summand and then a point coordinates within the summand (product quadrature). The array of summands to which the `isummand` points is maintained by the `SmolyakQuadrature` class to which the object knows the pointer `smolq`.

We provide a constructor which points to the beginning of the given summand. This constructor is used in `SmolyakQuadrature::begin` method which approximately divides all the iterators to subsets of equal size.

```cpp
(smolpit class declaration 59) ≡
class SmolyakQuadrature
{
    class smolpit {
        protected:
            const SmolyakQuadrature *smolq;
            unsigned int isummand;
            IntSequence *jseq;
            ParameterSignal *sig;
            Vector *p;
            double w;
    }

    public:
        smolpit();
        smolpit(const SmolyakQuadrature &q, unsigned int isum);
        smolpit(const smolpit &spit);
        ~smolpit();

        bool operator==(const smolpit &spit) const;
        bool operator!=(const smolpit &spit) const
        {
            return ~operator==(spit);
        }
        const smolpit &operator=(const smolpit &spit);
        smolpit &operator++();

        const ParameterSignal &signal() const
        {
            return *sig;
        }

        const Vector &point() const
        {
            return *p;
        }

        double weight() const
        {
            return w;
        }

        void print() const;

    protected:
        void setPointAndWeight();
    }
}
```

This code is used in section 58.
Here we define the class `SmolyakQuadrature`. It maintains an array of summands of the Smolyak quadrature formula:

\[ \sum_{l \leq |k| \leq l + d - 1} (-1)^{l + d - |k| - 1} \binom{d - 1}{|k| - l} (Q^1_{k_1} \otimes \ldots \otimes Q^1_{k_d}) f \]

Each summand is fully specified by sequence \( k \). The summands are here represented (besides \( k \)) also by sequence of number of points in each level selected by \( k \), and also by a cumulative number of evaluations. The latter two are added only for convenience.

The summands in the code are given by `levels`, which is a vector of \( k \) sequences, further by `levpoints` which is a vector of sequences of number of points in each level, and by `cumevals` which is the cumulative number of points, this is \( \sum_k \prod_{i=1}^d n_{k_i} \), where the sum is done through all \( k \) before the current.

The `levels` and `levpoints` vectors are used by `smolpit`.

```cpp
<SmolyakQuadrature class declaration (60)≡
class SmolyakQuadrature : public QuadratureImpl<smolpit> {
    friend class smolpit;
    int level;
    const OneDQuadrature &uquad;
    vector<IntSequence> levels;
    vector<IntSequence> levpoints;
    vector<int> cumevals;
    PascalTriangle psc;
public:
    SmolyakQuadrature(int d, int l, const OneDQuadrature &uq);
    virtual ~SmolyakQuadrature() {}
    virtual int numEvals(int level) const;
    void designLevelForEvals(int max_eval, int &lev, int &evals) const;
protected:
    smolpit begin(int ti, int tn, int level) const;
    unsigned int numSummands() const
    { return levels.size(); } 
private:
    int calcNumEvaluations(int level) const;
};

This code is used in section 58.
```

61. End of `smolyak.h` file
62. This is smolyak.cpp file.
#include "smolyak.h"
#include "symmetry.h"
(smolpit empty constructor 63);
(smolpit regular constructor 64);
(smolpit copy constructor 65);
(smolpit destructor 66);
(smolpit::operator≡ code 67);
(smolpit::operator= code 68);
(smolpit::operator++ code 69);
(smolpit::setPointAndWeight code 70);
(smolpit::print code 71);
(SmolyakQuadrature constructor 72);
(SmolyakQuadrature::numEvals code 73);
(SmolyakQuadrature::begin code 74);
(SmolyakQuadrature::calcNumEvaluations code 75);
(SmolyakQuadrature::designLevelForEvals code 76);

63. (smolpit empty constructor 63)≡
smolpit::smolpit()
: smolq(A), isummand(0), jseq(A), sig(A), p(A) { }
This code is used in section 62.

64. This constructs a beginning of isum summand in smolq. We must be careful here, since isum can be past-the-end, so no reference to vectors in smolq by isum must be done in this case.
(smolpit regular constructor 64)≡
smolpit::smolpit(const SmolyakQuadrature &q, unsigned int isum)
: smolq(q), isummand(isum), jseq(new IntSequence(q.dimen(), 0)), sig(new ParameterSignal(q.dimen())), p(new Vector(q.dimen()))
if (isummand < q.numSummands())
  setPointAndWeight();
}
This code is used in section 62.

65. (smolpit copy constructor 65)≡
smolpit::smolpit(const smolpit &spit)
: smolq(spit.smolq), isummand(spit.isummand), w(spit.w)
if (spit.jseq) jseq = new IntSequence(*(spit.jseq));
else jseq = Λ;
if (spit.sig) sig = new ParameterSignal(*(spit.sig));
else sig = Λ;
if (spit.p) p = new Vector(*(spit.p));
else p = Λ;
This code is used in section 62.
66. (smolpit destructor 66) ≡
    smolpit :: ~smolpit()
    {
        if (jseq) delete jseq;
        if (sig) delete sig;
        if (p) delete p;
    }
This code is used in section 62.

67. (smolpit::operator≡ code 67) ≡
    bool smolpit::operator≡(const smolpit &spit) const
    {
        bool ret = true;
        ret = ret & smolq ≡ spit.smolq;
        ret = ret & isummand ≡ spit.isummand;
        ret = ret & ((jseq ≡ Λ ∧ spit.jseq ≡ Λ) ∨ (jseq ≠ Λ ∧ spit.jseq ≠ Λ ∧ *jseq ≡ *(spit.jseq)));
        return ret;
    }
This code is used in section 62.

68. (smolpit::operator≡ code 68) ≡
    const smolpit &smolpit::operator≡(const smolpit &spit)
    {
        smolq = spit.smolq;
        isummand = spit.isummand;
        w = spit.w;
        if (jseq) delete jseq;
        if (sig) delete sig;
        if (p) delete p;
        if (spit.jseq) jseq = new IntSequence(*spit.jseq);
        else jseq = Λ;
        if (spit.sig) sig = new ParameterSignal(*spit.sig);
        else sig = Λ;
        if (spit.p) p = new Vector(*spit.p);
        else p = Λ;
        return *(this);
    }
This code is used in section 62.
69. We first try to increase index within the current summand. If we are at maximum, we go to a subsequent summand. Note that in this case all indices in jseq will be zero, so no change is needed.

\(\text{(smolpit::operator++ code 69)}\) ≡

```cpp
smolpit &smolpit::operator++()
{
    // todo: throw if smolq ≡ Λ or jseq ≡ Λ or sig ≡ Λ */
    const IntSequence &levpts = smolq->levpoints[isummand];
    int i = smolq->dimen() - 1;
    (*jseq)[i]++;
    while (i ≥ 0 ∧ (*jseq)[i] ≡ levpts[i]) {
        (*jseq)[i] = 0;
        i--;
        if (i ≥ 0) (*jseq)[i]++;
    }
    sig->signalAfter(std::max(i, 0));
    if (i < 0) isummand++;
    if (isummand < smolq->numSummands()) setPointAndWeight();
    return *this;
}
```

This code is used in section 62.

70. Here we set the point coordinates according to jseq and isummand. Also the weight is set here.

\(\text{(smolpit::setPointAndWeight code 70)}\) ≡

```cpp
void smolpit::setPointAndWeight()
{
    // todo: raise if smolq ≡ Λ or jseq ≡ Λ or sig ≡ Λ or */
    int l = smolq->level;
    int d = smolq->dimen();
    int sumk = (smolq->levels[isummand]).sum();
    int m1exp = l + d - sumk - 1;
    w = (2 ∗ (m1exp/2) ≡ m1exp) ? 1.0 : -1.0;
    w *= smolq->psc.noverk(d - 1, sumk - l);
    for (int i = 0; i < d; i++) {
        int ki = (smolq->levels[isummand])[i];
        (*p)[i] = (smolq->uquad).point(ki, (*jseq)[i]);
        w *= (smolq->uquad).weight(ki, (*jseq)[i]);
    }
}
```

This code is used in section 62.
71. Debug print.

(\texttt{smolpit::print} code 71) \equiv
\begin{verbatim}
void smolpit::print() const
{
    printf("isum=%-3d: [", isummand);
    for (int i = 0; i < smolq.dimen(); i++)
        printf("%2d", (smolq.levels[isummand])[i]);
    printf("] j=[");
    for (int i = 0; i < smolq.dimen(); i++)
        printf("%2d", (*jseq)[i]);
    printf("] \%+4.3f* (", w);
    for (int i = 0; i < smolq.dimen() - 1; i++)
        printf("%+4.3f", (*p)[i]);
    printf("%+4.3f)\n", (*p)[smolq.dimen() - 1]);
}
\end{verbatim}

This code is used in section 62.

72. Here is the constructor of \texttt{SmolyakQuadrature}. We have to setup \texttt{levels}, \texttt{levpoints} and \texttt{cumevals}. We have to go through all $d$-dimensional sequences $k$, such that $l \leq |k| \leq l + d - 1$ and all $k_i$ are positive integers. This is equivalent to going through all $k$ such that $l - d \leq |k| \leq l - 1$ and all $k_i$ are non-negative integers. This is equivalent to going through $d + 1$ dimensional sequences $(k, x)$ such that $|(k, x)| = l - 1$ and $x = 0, \ldots, d - 1$. The resulting sequence of positive integers is obtained by adding 1 to all $k_i$.

(\texttt{SmolyakQuadrature} constructor 72) \equiv
\begin{verbatim}
SmolyakQuadrature::SmolyakQuadrature(int d, int l, const OneDQuadrature &uq)
: QuadratureImpl(smolpit(d), level(l), uquad(uq), psc(d - 1, d - 1))
{
    /* todo: check $l > 1$, $l \geq d$ */
    int cum = 0;
    SymmetrySet ss(l - 1, d + 1);
    for (symiterator si(ss); ¬si.isEnd(); ++si) {
        if ((*(si)[d] <= d - 1) {
            IntSequence lev((const IntSequence &)*si, 0, d);
            lev.add(1);
            levels.push_back(lev);
            IntSequence levpts(d);
            for (int i = 0; i < d; i++)
                levpts[i] = uquad.numPoints(lev[i]);
            levpoints.push_back(levpts);
            cum += levpts.mult();
            cumevals.push_back(cum);
        }
    }
}
\end{verbatim}

This code is cited in section 75.

This code is used in section 62.
73. Here we return a number of evaluations of the quadrature for the given level. If the given level is the current one, we simply return the maximum cumulative number of evaluations. Otherwise we call costly calcNumEvaluations method.

\[
\text{SmolyakQuadrature}::\text{numEvals} \quad \equiv \\
\text{int SmolyakQuadrature}::\text{numEvals}(\text{int } l) \text{ const}
\]

\[
\{ \\
\text{if } (l \neq \text{level}) \text{ return calcNumEvaluations}(l); \\
\text{else return cumevals[numSummands()} - 1];
\}
\]

This code is used in section 62.

74. This divides all the evaluations to \(tn\) approximately equal groups, and returns the beginning of the specified group \(ti\). The granularity of divisions are summands as listed by \(levels\).

\[
\text{SmolyakQuadrature}::\text{begin} \quad \equiv \\
\text{smolpit SmolyakQuadrature}::\text{begin}(\text{int } ti, \text{int } tn, \text{int } l) \text{ const}
\]

\[
\{ \\
/\ast \text{todo: raise is level } \neq l \ast/ \\
\text{if } (ti \equiv tn) \text{ return smolpit(*this, numSummands());} \\
\text{int totevals} = \text{cumevals[numSummands()]} - 1]; \\
\text{int evals} = (\text{totevals} * ti)/tn; \\
\text{unsigned int isum} = 0; \\
\text{while } (\text{isum} + 1 < \text{numSummands()} \land \text{cumevals[isum + 1]} < \text{evals}) \text{ isum++;} \\
\text{return smolpit(*this, isum);}
\}
\]

This code is used in section 62.

75. This is the same in a structure as \{SmolyakQuadrature constructor 72\}. We have to go through all summands and calculate a number of evaluations in each summand.

\[
\text{SmolyakQuadrature}::\text{calcNumEvaluations} \quad \equiv \\
\text{int SmolyakQuadrature}::\text{calcNumEvaluations}(\text{int } lev) \text{ const}
\]

\[
\{ \\
\text{int cum} = 0; \\
\text{SymmetrySet ss(lev - 1, dim + 1);} \\
\text{for } (\text{symiterator si(ss); } \neg \text{si.isEnd(); } ++\text{si}) \{ \\
\text{if } ((\ast\text{si})[\text{dim}] \leq \text{dim} - 1) \{ \\
\text{IntSequence lev}((\text{const IntSequence } \&)*\text{si, 0, dim}); \\
\text{lev.add(1);} \\
\text{IntSequence levpts(dim);} \\
\text{for } (\text{int } i = 0; i < \text{dim}; i++) \text{ levpts[i]} = uquad.numPoints(lev[i]); \\
\text{cum} += \text{levpts.mult();}
\}
\}
\text{return cum;}
\}
\]

This code is used in section 62.
76. This returns a maximum level such that the number of evaluations is less than the given number.

\[(\text{SmolyakQuadrature}::\text{designLevelForEvals} \text{ code 76}) \equiv\]
\[
\text{void SmolyakQuadrature::designLevelForEvals}(\text{int max_evals}, \text{int } \&\text{lev}, \text{int } \&\text{evals}) \text{ const}
\{
    \text{int } last_evals;
    \text{evals} = 1;
    \text{lev} = 1;
    \text{do } \{\text{lev}++;
    \text{last_evals} = \text{evals};
    \text{evals} = \text{calcNumEvaluations}(\text{lev});
    \} \text{ while } (\text{lev} < \text{uquad.numLevels()} \land \text{evals} \leq \text{max_evals});
    \text{lev}--;
    \text{evals} = \text{last_evals};
\}
\]
This code is used in section 62.

77. End of smolyak.cpp file
78. Quasi Monte Carlo quadrature. This is quasi_mcarlo.h file.

This defines quasi Monte Carlo quadratures for cube and for a function multiplied by normal density. The quadrature for a cube is named QMCarloCubeQuadrature and integrates:

\[ \int_{(0,1)^n} f(x)dx \]

The quadrature for a function of normally distributed parameters is named QMCarloNormalQuadrature and integrates:

\[ \frac{1}{\sqrt{(2\pi)^n}} \int_{(-\infty,\infty)^n} f(x)e^{-\frac{1}{2}x^T x}dx \]

For a cube we define qmcpit as iterator of QMCarloCubeQuadrature, and for the normal density multiplied function we define qmcnpit as iterator of QMCarloNormalQuadrature.

The quasi Monte Carlo method generates low discrepancy points with equal weights. The one dimensional low discrepancy sequences are generated by RadicalInverse class, the sequences are combined for higher dimensions by HaltonSequence class. The Halton sequence can use a permutation scheme; PermutationScheme is an abstract class for all permutation schemes. We have three implementations: WarnockPerScheme, ReversePerScheme, and IdentityPerScheme.

```cpp
#ifndef QUASI_MCARLO_H
#define QUASI_MCARLO_H
#include "int_sequence.h"
#include "quadrature.h"
#include <vector>
⟨PermutationScheme class declaration 79⟩
⟨RadicalInverse class declaration 80⟩;
⟨HaltonSequence class declaration 81⟩;
⟨QMCSpecification class declaration 82⟩;
⟨qmcpit class declaration 83⟩;
⟨QMCarloCubeQuadrature class declaration 84⟩;
⟨qmcnpit class declaration 85⟩;
⟨QMCarloNormalQuadrature class declaration 86⟩;
⟨WarnockPerScheme class declaration 87⟩;
⟨ReversePerScheme class declaration 88⟩;
⟨IdentityPerScheme class declaration 89⟩;
#endif
```

79. This abstract class declares permute method which permutes coefficient \(c\) having index of \(i\) fro the base \(base\) and returns the permuted coefficient which must be in \(0, \ldots, base - 1\).

```cpp
⟨PermutationScheme class declaration 79⟩ ≡
class PermutationScheme { public:
    PermutationScheme() {} 
    virtual ~PermutationScheme() {} 
    virtual int permute(int i, int base, int c) const = 0;
};
```

This code is used in section 78.
This class represents an integer number $num$ as $c_0 + c_1 b + c_2 b^2 + \ldots + c_j b^j$, where $b$ is $base$ and $c_0, \ldots, c_j$ is stored in $coeff$. The size of $IntSequence$ coeff does not grow with growing $num$, but is fixed from the very beginning and is set according to supplied maximum $maxn$.

The basic method is $eval$ which evaluates the RadicalInverse with a given permutation scheme and returns the point, and $increase$ which increases $num$ and recalculates the coefficients.

(RadicalInverse class declaration 80) ≡

```cpp
class RadicalInverse {
    int num;
    int base;
    int maxn;
    int j;
    IntSequence coeff;

public:
    RadicalInverse(int n, int b, int mxn);
    RadicalInverse(const RadicalInverse &ri) :
        num(ri.num), base(ri.base), maxn(ri.maxn), j(ri.j), coeff(ri.coeff) {}
    const RadicalInverse &operator=(const RadicalInverse &radi) {
        num = radi.num;
        base = radi.base;
        maxn = radi.maxn;
        j = radi.j;
        coeff = radi.coeff;
        return *this;
    }
    double eval(const PermutationScheme &p) const;
    void increase();
    void print() const;
};
```

This code is used in section 78.
81. This is a vector of RadicalInverses, each RadicalInverse has a different prime as its base. The static members primes and num_primes define a precalculated array of primes. The increase method of the class increases indices in all RadicalInverses and sets point pt to contain the points in each dimension.

(class HaltonSequence class declaration 81) ≡

```cpp
class HaltonSequence {
private:
    static int primes[];
    static int num_primes;

protected:
    int num;
    int maxn;
    vector<RadicalInverse> ri;
    const PermutationScheme &per;
    Vector pt;

public:
    HaltonSequence(int n, int mxn, int dim, const PermutationScheme &p);
    HaltonSequence(const HaltonSequence &hs)
        : num(hs.num), maxn(hs.maxn), ri(hs.ri), per(hs.per), pt(hs.pt) {}
    const HaltonSequence &operator=(const HaltonSequence &hs);
    void increase();
    const Vector &point() const
    { return pt; }
    const int getNum() const
    { return num; }
    void print() const;

protected:
    void eval();
};
```

This code is used in section 78.
82. This is a specification of quasi Monte Carlo quadrature. It consists of dimension $dim$, number of points (or level) $lev$, and the permutation scheme. This class is common to all quasi Monte Carlo classes.

\[
\text{(QMCSpecification class declaration 82) } \equiv \\
\text{class QMCSpecification } \\
\text{protected:} \\
\text{int dim;} \\
\text{int lev;} \\
\text{const PermutationScheme &per_scheme;} \\
\text{public:} \\
\text{QMCSpecification(int d, int l, const PermutationScheme &p)} \\
\text{: dim(d), lev(l), per_scheme(p) } \\
\text{virtual ~QMCSpecification()} \\
\text{()} \\
\text{int dimen() const} \\
\text{\{ return dim; \}} \\
\text{int level() const} \\
\text{\{ return lev; \}} \\
\text{const PermutationScheme &getPerScheme() const} \\
\text{\{ return per_scheme; \}} \\
\text{);} \\
\text{This code is used in section 78.}
83. This is an iterator for quasi Monte Carlo over a cube QMCarloCubeQuadrature. The iterator maintains HaltonSequence of the same dimension as given by the specification. An iterator can be constructed from a given number n, or by a copy constructor. For technical reasons, there is also an empty constructor; for that reason, every member is a pointer.

(qmcpit class declaration 83) ≡
class qmcpit {
protected:
    const QMCSpecification *spec;
    HaltonSequence *halton;
    ParameterSignal *sig;
public:
    qmcpit();
    qmcpit(const QMCSpecification &, int n);
    qmcpit(const qmcpit &, qpit);
    ~qmcpit();
    bool operator==(const qmcpit &, qpit) const;
    bool operator!=(const qmcpit &, qpit) const
        { return !operator==(qpit); }
    const qmcpit &operator=(const qmcpit &, qpit);
    qmcpit &operator++();
    const ParameterSignal &signal() const
        { return *sig; }
    const Vector &point() const
        { return halton->point(); }
    double weight() const;
    void print() const
        { halton->print(); };
}

This code is used in section 78.

84. This is an easy declaration of quasi Monte Carlo quadrature for a cube. Everything important has been done in its iterator qmcpit, so we only inherit from general Quadrature and reimplement begin and numEvals.

(QMCarloCubeQuadrature class declaration 84) ≡
class QMCarloCubeQuadrature : public QuadratureImpl(qmcpit), public QMCSpecification

public:
    QMCarloCubeQuadrature(int d, int l, const PermutationScheme &p)
        : QuadratureImpl(qmcpit)(d), QMCSpecification(d, l, p) {}
    virtual ~QMCarloCubeQuadrature() {}
    int numEvals(int l) const
        { return l; }

protected:
    qmcpit begin(int ti, int tn, int lev) const
        { return qmcpit(*this, ti * level() / tn + 1); }
};

This code is used in section 78.
85. This is an iterator for QMCarloNormalQuadrature. It is equivalent to an iterator for quasi Monte Carlo cube quadrature but a point. The point is obtained from a point of QMCarloCubeQuadrature by a transformation \( (0, 1) \rightarrow (-\infty, \infty) \) applied to all dimensions. The transformation yields a normal distribution from a uniform distribution on \( (0, 1) \). It is in fact \texttt{NormalICDF}.

(\texttt{qmcnpit} class declaration 85) \equiv

class qmcnpit : public qmcnpit {
protected:
    Vector *pnt;
public:
    qmcnpit();
    qmcnpit(const QMCSpecification &spec, int n);
    qmcnpit(const qmcnpit &qpit);
    ~qmcnpit();
    bool operator==(const qmcnpit &qpit) const
    { return qmcnpit::operator==(qpit); }
    bool operator!=(const qmcnpit &qpit) const
    { return !operator==(qpit); }
    const qmcnpit &operator=(const qmcnpit &qpit);
    qmcnpit &operator++();
    const ParameterSignal &signal() const
    { return *sig; }
    const Vector &point() const
    { return *pnt; }
    void print() const
    { halton-print();
      pnt-print(); }
};

This code is used in section 78.

86. This is an easy declaration of quasi Monte Carlo quadrature for a cube. Everything important has been done in its iterator \texttt{qmcnpit}, so we only inherit from general \texttt{Quadrature} and reimplement \texttt{begin} and \texttt{numEvals}.

(\texttt{QMCarloNormalQuadrature} class declaration 86) \equiv

class QMCarloNormalQuadrature : public QuadratureImpl(qmcnpit), public QMCSpecification {
public:
    QMCarloNormalQuadrature(int d, int l, const PermutationScheme &p)
    : QuadratureImpl(qmcnpit)(d), QMCSpecification(d, l, p) {}
    virtual ~QMCarloNormalQuadrature() {}
    int numEvals(int l) const
    { return l; }
protected:
    qmcnpit begin(int ti, int tn, int lev) const
    { return qmcnpit(*this, ti * level() / tn + 1); }
};

This code is used in section 78.
87. Declares Warnock permutation scheme.

\( \text{WarnockPerScheme class declaration} \equiv \)
\[
\text{class } \text{WarnockPerScheme} : \text{public PermutationScheme} \\
\text{public:}
\]
\[
\text{int permute} (\text{int } i, \text{int } base, \text{int } c) \text{ const;}
\]
This code is used in section 78.

88. Declares reverse permutation scheme.

\( \text{ReversePerScheme class declaration} \equiv \)
\[
\text{class } \text{ReversePerScheme} : \text{public PermutationScheme} \\
\text{public:}
\]
\[
\text{int permute} (\text{int } i, \text{int } base, \text{int } c) \text{ const;}
\]
This code is used in section 78.

89. Declares no permutation (identity) scheme.

\( \text{IdentityPerScheme class declaration} \equiv \)
\[
\text{class } \text{IdentityPerScheme} : \text{public PermutationScheme} \\
\text{public:}
\]
\[
\text{int permute} (\text{int } i, \text{int } base, \text{int } c) \text{ const}
\]{ return c; }
\]
This code is used in section 78.

90. End of quasi_mcarlo.h file
91. This is quasi_mcarlo.cpp file.

```c
#include "quasi_mcarlo.h"
#include <cmath>
RadicalInverse constructor code 92);
(RadicalInverse::eval code 93);
(RadicalInverse::increase code 94);
(RadicalInverse::print code 95);
(HaltonSequence static data 96);
(HaltonSequence constructor code 97);
(HaltonSequence::operator= code 98);
(HaltonSequence::increase code 99);
(HaltonSequence::eval code 100);
(HaltonSequence::print code 101);
(qmcpit empty constructor code 102);
(qmcpit regular constructor code 103);
(qmcpit copy constructor code 104);
(qmcpit destructor 105);
(qmcpit::operator≡ code 106);
(qmcpit::operator= code 107);
(qmcpit::operator++ code 108);
(qmcpit::weight code 109);
(qmcpit empty constructor code 110);
(qmcpit regular constructor code 111);
(qmcpit copy constructor code 112);
(qmcpit destructor 113);
(qmcpit::operator= code 114);
(qmcpit::operator++ code 115);
(WarnockPerScheme::permute code 116);
(ReversePerScheme::permute code 117);
```

92. Here in the constructor, we have to calculate a maximum length of coeff array for a given base and given maximum maxn. After allocation, we calculate the coefficients.

```c
(RadicalInverse constructor code 92) ≡
RadicalInverse::RadicalInverse(int n, int b, int mxn)
  : num(n), base(b), maxn(mxn), coeff((int)(floor(log((double) maxn)/log((double) b)) + 2), 0) {
    int nr = num;
    j = -1;
    do {
      j++;
      coeff[j] = nr % base;
      nr = nr/base;
    } while (nr > 0);
  }
```

This code is used in section 91.
This evaluates the radical inverse. If there was no permutation, we have to calculate
\[
\frac{c_0}{b} + \frac{c_1}{b^2} + \ldots + \frac{c_j}{b^{j+1}}
\]
which is evaluated as
\[
\left( \ldots \left( \frac{c_j}{b} \cdot \frac{1}{b} + \frac{c_{j-1}}{b} \right) \cdot \frac{1}{b} + \frac{c_{j-2}}{b} \right) \ldots \cdot \frac{1}{b} + \frac{c_0}{b}
\]
Now with permutation \(\pi\), we have
\[
\left( \ldots \left( \pi \left( \frac{c_j}{b} \right) \cdot \frac{1}{b} + \pi \left( \frac{c_{j-1}}{b} \right) \right) \cdot \frac{1}{b} + \pi \left( \frac{c_{j-2}}{b} \right) \right) \ldots \cdot \frac{1}{b} + \pi \left( \frac{c_0}{b} \right)
\]

\[\text{RadicalInverse}::\text{eval code 93} \equiv \]
\[
\text{double RadicalInverse}::\text{eval(const PermutationScheme } \&p) \text{ const}
\{
\text{double res } = \text{0;}
\text{for (int } i = j; i \geq 0; i--) \{
\text{int cper } = p.\text{permute}(i, \text{base, coeff[i]});
\text{res } = (cper + \text{res})/\text{base;}
\}
\text{return res;}
\}
\]
This code is used in section 91.

We just add 1 to the lowest coefficient and check for overflow with respect to the base.

\[\text{RadicalInverse}::\text{increase code 94} \equiv \]
\[
\text{void RadicalInverse}::\text{increase()}
\{
\text{/* todo: raise if num } + 1 > \text{maxn */}
\text{num }++;
\text{int } i = 0;
\text{coeff[i]}++;
\text{while (coeff[i] } \equiv \text{base) }
\{
\text{coeff[i] } = \text{0;}
\text{coeff[i+i]}++;
\}
\text{if (i } > \text{j) } j = i;
\}
\]
This code is used in section 91.

Debug print.

\[\text{RadicalInverse}::\text{print code 95} \equiv \]
\[
\text{void RadicalInverse}::\text{print() const}
\{
\text{printf("n=%d b=%d c=", num, base);}
\text{coeff.print();}
\}
\]
This code is used in section 91.
96. Here we have the first 170 primes. This means that we are not able to integrate dimensions greater than 170.

\[ \text{HaltonSequence static data 96} \equiv \]
\[ \text{int HaltonSequence::num_primes} = 170; \]
\[ \text{int HaltonSequence::primes[]} = \{2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, \]
\[ 83, 89, 97, 101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173, 179, 181, 191, 193, 197, \]
\[ 463, 467, 479, 487, 491, 499, 503, 509, 521, 523, 541, 547, 557, 563, 569, 571, 577, 587, 593, 599, 601, 607, \]
\[ 613, 617, 619, 631, 641, 643, 647, 653, 659, 661, 673, 677, 683, 691, 701, 709, 719, 727, 733, 739, 743, 751, \]
\[ 757, 761, 769, 773, 787, 797, 809, 811, 821, 823, 827, 829, 839, 853, 857, 859, 863, 877, 881, 883, 887, 907, \]
\[ 911, 919, 929, 937, 941, 947, 953, 967, 971, 977, 983, 991, 997, 1009, 1013\};\]

This code is used in section 91.

97. This takes first \( \dim \) primes and constructs \( \dim \) radical inverses and calls \( \text{eval} \).

\[ \text{HaltonSequence constructor code 97} \equiv \]
\[ \text{HaltonSequence::HaltonSequence(int n, int mxn, int dim, const PermutationScheme &p)} : \text{num(n), mxn(mxn), per(p), pt(dim) \{ /* todo: raise if dim > num_primes */}
\[ \text{for (int i = 0; i < dim; i++) ri.push_back(RadicalInverse(num, primes[i], mxn));}
\[ \text{eval();}
\[ \}}\]

This code is used in section 91.

98.

\[ \text{HaltonSequence::operator = code 98} \equiv \]
\[ \text{const HaltonSequence & HaltonSequence::operator = (const HaltonSequence &hs) \{}
\[ \text{num = hs.num;}
\[ \text{mxn = hs.mxn;}
\[ \text{ri.clear();}
\[ \text{for (unsigned int i = 0; i < hs.ri.size(); i++) ri.push_back(RadicalInverse(hs.ri[i]));}
\[ \text{pt = hs.pt;}
\[ \text{return *this;}
\[ \}}\]

This code is used in section 91.

99. This calls \( \text{RadicalInverse::increase} \) for all radical inverses and calls \( \text{eval} \).

\[ \text{HaltonSequence::increase code 99} \equiv \]
\[ \text{void HaltonSequence::increase() \{}
\[ \text{for (unsigned int i = 0; i < ri.size(); i++) ri[i].increase();}
\[ \text{num++;}
\[ \text{if (num \leq mxn) eval();}
\[ \}}\]

This code is used in section 91.
100. This sets point $pt$ to radical inverse evaluations in each dimension.

\[
\text{HaltonSequence::eval code 100) } \equiv \\
\text{void HaltonSequence::eval( ) } \\
\{ \\
\text{ for (unsigned int } i = 0; i < ri.size(); i++) \text{ pt[i] = ri[i].eval(per);} \\
\}
\]

This code is used in section 91.

101. Debug print.

\[
\text{HaltonSequence::print code 101) } \equiv \\
\text{void HaltonSequence::print( ) const } \\
\{ \\
\text{ for (unsigned int } i = 0; i < ri.size(); i++) \text{ ri[i].print();} \\
\text{ printf("point=[\text{\_\_\_}\n");} \\
\text{ for (unsigned int } i = 0; i < ri.size(); i++) \text{ printf("\%7.6f_\n", pt[i]);} \\
\text{ printf("\}n");} \\
\}
\]

This code is used in section 91.

102. \text{qmcpit empty constructor code 102) } \equiv \\
\text{qmcpit::qmcpit( ) } : \text{spec(\(\Lambda\)), halton(\(\Lambda\)), sig(\(\Lambda\)) } \{ \}

This code is used in section 91.

103. \text{qmcpit regular constructor code 103) } \equiv \\
\text{qmcpit::qmcpit(const QMCSpecification &s, int n) } : \text{spec(s), halton(new HaltonSequence(n, s.level(), s.dimen(), s.getPerScheme())), sig(new ParameterSignal(s.dimen())) } \{ \}

This code is used in section 91.

104. \text{qmcpit copy constructor code 104) } \equiv \\
\text{qmcpit::qmcpit(const qmcpit &qpit) } : \text{spec(qpit.spec), halton(\(\Lambda\)), sig(\(\Lambda\)) } \{ \\
\text{ if (qpit.halton) halton = new HaltonSequence(*qpit.halton));} \\
\text{ if (qpit.sig) sig = new ParameterSignal(qpit.spec->dimen( ));} \\
\}

This code is used in section 91.

105. \text{qmcpit destructor 105) } \equiv \\
\text{qmcpit::~qmcpit( ) } \{ \\
\text{ if (halton) delete halton;} \\
\text{ if (sig) delete sig;} \\
\}

This code is used in section 91.
106. \( (\text{qmcpit}::\text{operator} \equiv \text{code} \ 106) \equiv \)
\[
\begin{align*}
\text{bool qmcpit}::\text{operator}(\text{const qmcpit} &qpit) \text{ const} \\
&\{ \\
&\quad \text{return } (\text{spec } \equiv qpit.\text{spec}) \land ((\text{halton } \equiv \Lambda \land qpit.\text{halton } \equiv \Lambda) \lor (\text{halton } \not\equiv \Lambda \land qpit.\text{halton } \neq \Lambda \land \text{halton}\text{-getNum]() } \equiv qpit.\text{halton}\text{-getNum()}) ); \\
&\}\n\end{align*}
\]
This code is used in section 91.

107. \( (\text{qmcpit}::\text{operator} = \text{code} \ 107) \equiv \)
\[
\begin{align*}
\text{const qmcpit} &\text{qmcpit}::\text{operator} =(\text{const qmcpit} &qpit) \\
&\{ \\
&\quad \text{spec } = qpit.\text{spec}; \\
&\quad \text{if } (\text{halton}) \text{ delete halton}; \\
&\quad \text{if } (qpit.\text{halton}) \text{ halton } = \text{new HaltonSequence}(*(qpit.\text{halton})); \\
&\quad \text{else halton } = \Lambda; \\
&\quad \text{return } *\text{this}; \\
&\}\n\end{align*}
\]
This code is used in section 91.

108. \( (\text{qmcpit}::\text{operator} ++ \text{ code} \ 108) \equiv \)
\[
\begin{align*}
\text{qmcpit} &\text{qmcpit}::\text{operator} ++() \\
&\{ \\
&\quad /\!\!* \text{ todo: raise if halton } \equiv \text{null } \lor \text{qmcq } \equiv \Lambda */ \\
&\quad \text{halton}\text{-increase}(); \\
&\quad \text{return } *\text{this}; \\
&\}\n\end{align*}
\]
This code is used in section 91.

109. \( (\text{qmcpit}::\text{weight} \text{ code} \ 109) \equiv \)
\[
\begin{align*}
\text{double qmcpit}::\text{weight}() \text{ const} \\
&\{ \\
&\quad \text{return } 1.0/\text{spec}\text{-level}(); \\
&\}\n\end{align*}
\]
This code is used in section 91.

110. \( (\text{qmcnpit empty constructor code} \ 110) \equiv \)
\[
\begin{align*}
\text{qmcnpit}::\text{qmcnpit}() \\
&: \text{qmcpit()}, \text{pnt}(\Lambda) \{} \} \\
\end{align*}
\]
This code is used in section 91.

111. \( (\text{qmcnpit regular constructor code} \ 111) \equiv \)
\[
\begin{align*}
\text{qmcnpit}::\text{qmcnpit}(\text{const QMCSpecification} &s, \text{int} n) \\
&: \text{qmcpit}(s, n), \text{pnt(new Vector}(s.\text{dimen}())) \{} \} \\
\end{align*}
\]
This code is used in section 91.
112. (qmcnpit copy constructor code 112) \(\equiv\)
\[\text{qmcnpit}::\text{qmcnpit}(const\ \text{qmcnpit}\ &qpit)\]
\[\begin{align*}
\text{qmcnpit}&::\text{qmcnpit}(qpit),\ pnt(\Lambda)\ \{ \\\n&\quad \text{if} \ (qpit.pnt)\ pnt = \text{new\ Vector}(*(qpit.pnt)); \\\n\} \]
This code is used in section 91.

113. (qmcnpit destructor 113) \(\equiv\)
\[\text{qmcnpit}::\sim\text{qmcnpit}()\]
\[\begin{align*}
&\quad \{ \\\n&\quad \quad \text{if} \ (pnt)\ \text{delete} \ pnt; \\\n&\quad \} \]
This code is used in section 91.

114. (qmcnpit ::operator= code 114) \(\equiv\)
\[\text{const\ qmcnpit} &\text{qmcnpit}::\text{operator}=(const\ \text{qmcnpit} &qpit)\]
\[\begin{align*}
&\quad \{ \\\n&\quad \quad \text{qmcpit}::\text{operator}=(qpit); \\\n&\quad \quad \text{if} \ (pnt)\ \text{delete} \ pnt; \\\n&\quad \quad \text{if} \ (qpit.pnt)\ pnt = \text{new\ Vector}(*(qpit.pnt)); \\\n&\quad \quad \text{else} \ pnt = \Lambda; \\\n&\quad \quad \text{return}\ *\text{this}; \\\n&\quad \} \]
This code is used in section 91.

115. Here we increase a point in Halton sequence and then store images of the points in NormalICDF function. (qmcnpit ::operator ++ code 115) \(\equiv\)
\[\text{qmcnpit} &\text{qmcnpit}::\text{operator}++()\]
\[\begin{align*}
&\quad \{ \\\n&\quad \quad \text{qmcpit}::\text{operator}++; \\\n&\quad \quad \text{for}\ (\text{int} \ i = 0;\ i < \text{halton-point().length();}\ i++)\ \{\text{pnt}[i] = \text{NormalICDF}::\text{get}\ (\text{halton-point()}[i]);\ \} \\\n&\quad \quad \text{return}\ *\text{this}; \\\n&\quad \} \]
This code is used in section 91.

116. Clear from code. (WarnockPerScheme::permute code 116) \(\equiv\)
\[\text{int\ WarnockPerScheme}::\text{permute}(\text{int} i,\ \text{int} base,\ \text{int} c)\ \text{const}\]
\[\begin{align*}
&\quad \{ \\\n&\quad \quad \text{return}\ (c + i)\ %\ base; \\\n&\quad \} \]
This code is used in section 91.
117. Clear from code.

\texttt{\textbf{ReversePerScheme::permute} code 117} \equiv

\begin{verbatim}
int ReversePerScheme::permute(int i, int base, int c) const
{
    return (base - c) % base;
}
\end{verbatim}

This code is used in section 91.

118. End of quasi_mcarlo.cpp file.

\begin{verbatim}
__CROSS_COMPILE__ : 7.
__MINGW32__ : 7.
__MINGW64__ : 7.
_Tpit : 24, 25, 26, 27, 28.
A : 5.
abs : 38.
add : 25, 72, 75.
b : 80, 92.
base : 19, 79, 80, 87, 88, 89, 92, 93, 94, 95, 116, 117.
begin : 25, 28.
begin : 25, 26, 28, 41, 42, 55, 59, 60, 74, 84, 86.
calcCholeskyFactor : 2, 14, 15, 19.
calcMultiplier : 5, 14, 15, 18.
calcNumEvaluations : 60, 73, 75, 76.
calcOffsets : 29, 35.
clear : 98.
clone : 3, 5, 11, 12, 16.
coeff : 80, 92, 93, 94, 95.
cper : 93.
cum : 72, 75.
cumevals : 60, 72, 73, 74.
d : 23, 26, 42, 54, 60, 70, 72, 82, 84, 86.
data : 2, 8, 9, 10.
delete_flag : 5, 14, 15, 16.
designLevelPerEval : 42, 56, 60, 76.
dim : 22, 75, 81, 82, 97.
dimen : 22, 28, 42, 46, 51, 52, 53, 55, 64, 69, 70, 71, 82, 103, 104, 111.
dpotrf : 19.
dot : 25, 26, 28.
dot_flag : 41, 45, 46, 47, 49, 50, 51, 52.
eval : 4, 5, 17, 25, 30, 81, 93, 97, 99, 100.
evals : 42, 56, 60, 74, 76.
eval : 28.
f : 4, 5, 11, 12, 14, 15, 16, 24.
false : 10, 11, 14, 46.
fclose : 28.
fd : 28.
first_shallow : 4, 11, 12, 13.
floor : 38, 92.
fname : 28.

fopen : 28.
fwrite : 28.
fs : 23, 26, 27.
func : 3, 5, 14, 15, 16, 17, 23, 24, 25, 26.
funes : 4, 11, 12, 13.
GaussConverterFunction : 5, 14, 15, 16, 17, 18, 19, 30.
GaussHermite : 21, 30, 36.
GaussLegendre : 21, 31, 37.
GeneralMatrix : 5, 14, 15, 19.
get : 19, 32, 38, 115.
getfunc : 4, 27.
getNum : 4, 27, 81, 106.
getPerScheme : 83, 103.
gh_num_levels : 36.
gh_num_points : 36.
gh_points : 36.
gh_weights : 36.
gl_num_levels : 37.
gl_num_points : 37.
gl_points : 37.
gl_weights : 37.
gr : 27.
hs : 81, 98.
i : 2, 4, 8, 10, 11, 12, 13, 17, 19, 22, 28, 29, 35, 38.
IdentityPerScheme : 78, 89.
idim : 3.
in_dim : 3.
increase : 80, 81, 94, 99, 108.
info : 19.
insert : 27.
integrate : 23, 26, 27.
IntegrationWorker : 24, 26, 27.
IntSequence : 29, 41, 46, 47, 50, 59, 60, 64, 65, 68, 69, 72, 75, 80.
\end{verbatim}
isEnd: 72, 75.
isum: 59, 64, 74.
isummand: 59, 63, 64, 65, 67, 68, 69, 70, 71.
j: 10, 80.
jseq: 11, 45, 46, 47, 48, 49, 50, 51, 52, 53, 59, 63, 64, 65, 66, 67, 68, 69, 70, 71.
j0: 41, 46.
ki: 70.
l: 2, 10, 24, 11, 42, 46, 55, 60, 70, 72, 73, 74.

lapack_int: 19.
last_evals: 56, 76.
leve: 22.
length: 25, 115.
lev: 42, 56, 60, 72, 75, 76, 82, 84, 86.
level: 22, 23, 24, 25, 26, 27, 28, 29, 41, 42, 45, 46, 47, 52, 60, 70, 72, 73, 74, 82, 84, 86, 103, 109.
levels: 60, 70, 71, 72, 74.
levpoints: 60, 69, 72.
levpts: 60, 72, 75.
log: 92.
M_PI: 1, 18.
max: 51, 69.
max_eval: 42, 60.
max_evals: 56, 76.
maxLevel: 72.
mxn: 80, 81, 92, 94, 97, 98, 99.
mempcy: 9.
milevel: 72.
min: 10.
mult: 17, 72, 75.
multaVec: 17.
multiplier: 5, 14, 15, 16, 17.
mxn: 80, 81, 92, 97.
m1exp: 70.

n: 2, 3, 7, 11, 12, 80, 81, 83, 85, 92, 97, 103, 111.

levels: 29.
normal_icdf_data: 38.
normal_icdf_end: 38.
normal_icdf_num: 38.
normal_icdf_step: 38.
normal_icdf_data: 38.

NormalICDF: 32, 38, 85, 115.
nowerk: 70.
npoints: 20, 41, 45, 46, 47, 51, 55.
nr: 92.
null: 108.
num: 2, 8, 9, 10, 80, 81, 92, 94, 95, 97, 98, 99.
num_levels: 29, 35.
num_points: 29, 35.
num_primes: 81, 96, 97.
num_Evals: 23, 42, 56, 60, 73, 84, 86.
QUADRATURE_H: 21.
QuadratureImpl: 21, 24, 26, 40, 41, 42, 54, 60, 72, 84, 86.
QUASI_MONTE_CARLO_H: 78.
radi: 80.
RadicalInverse: 78, 80, 81, 92, 93, 94, 95, 97, 98, 99.
res: 42, 93.
ret: 49, 67.
ReversePerScheme: 78, 88, 117.
ri: 80, 81, 97, 98, 99, 100, 101.
rows: 19.
run: 25, 27, 28.
s: 17, 83, 103, 111.
savePoints: 28.
setPointAndWeight: 41, 46, 51, 52, 59, 64, 69, 70.
si: 72, 75.
sig: 2, 3, 5, 9, 17, 41, 45, 46, 47, 48, 50, 51, 52, 59, 63, 64, 65, 66, 68, 69, 70, 83, 85, 102, 103, 104, 105.
signalAfter: 2, 10, 17, 51, 69.
smolpit: 58, 59, 60, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 74.
smolyq: 59, 63, 64, 65, 67, 68, 69, 70, 71.
SMOLYAK_H: 58.
SmolyakQuadrature: 58, 59, 60, 64, 72, 73, 74, 75, 76.
spit: 41, 59, 65, 67, 68.
sqrt: 17, 18.
ss: 72, 75.
start: 13, 26.
std: 4, 10, 38, 51, 69.
stderr: 28.
sum: 70.
sunk: 70.
symiterator: 72, 75.
SymmetrySet: 72, 75.
syn: 25.
SYNCHRO: 25.
THREAD: 24.
THREAD_GROUP: 27.
ti: 24, 25, 26, 27, 42, 55, 60, 74, 84, 86.
tti: 24.
tmp: 25.
tmpall: 25.
tn: 23, 24, 25, 26, 42, 55, 60, 74, 84, 86.
tnn: 24.
totevals: 74.
true: 2, 3, 8, 10, 12, 15, 16, 45, 46, 49, 51, 52, 67.

uq: 42, 54, 60, 72.
uquad: 42, 46, 52, 54, 55, 56, 60, 70, 72, 75, 76.
vecw: 5, 14, 15, 19.
vector: 4, 60, 81.
VECTOR_FUNCTION_H: 1.
VectorFunction: 1, 2, 4, 5, 11, 12, 14, 15, 16, 23, 24, 26.
VectorFunctionSet: 4, 11, 12, 13, 23, 26, 27.
w: 41, 50.
WarnockPerScheme: 78, 87, 116.
weights: 20.

u: 41, 50.
weights: 20.

x: 17, 32, 38.
xx: 38.
xx1: 38.
y: 38.

y: 38.
yy1: 38.
yy2: 38.
zeros: 17, 25, 27.
(GaussConverterFunction::calcCholeskyFactor code 19) Used in section 7.
(GaussConverterFunction::eval code 17) Used in section 7.
(GaussConverterFunction::multiplier code 18) Used in section 7.
(GaussConverterFunction class declaration 5) Used in section 1.
(GaussConverterFunction constructor code 1 14) Used in section 7.
(GaussConverterFunction constructor code 2 15) Used in section 7.
(GaussConverterFunction copy constructor code 16) Used in section 7.
(GaussHermite class declaration 30) Used in section 21.
(GaussHermite constructor code 36) Used in section 34.
(GaussLegendre class declaration 31) Used in section 21.
(GaussLegendre constructor code 37) Used in section 34.
HaltonSequence::eval code 100) Used in section 91.
HaltonSequence::increase code 99) Used in section 91.
HaltonSequence::operator= code 98) Used in section 91.
HaltonSequence::print code 101) Used in section 91.
HaltonSequence class declaration 81) Used in section 78.
HaltonSequence constructor code 97) Used in section 91.
HaltonSequence static data 96) Used in section 91.
IdentityPerScheme class declaration 89) Used in section 78.
IntegrationWorker::operator()()() code 25) Used in section 24.
IntegrationWorker class declaration 24) Used in section 21.
NormalICDF class declaration 32) Used in section 21.
NormalICDF get code 38) Used in section 34.
OneDPreCalcQuadrature::calcOffsets code 35) Used in section 34.
OneDPreCalcQuadrature class declaration 29) Used in section 21.
OneDQuadrature class declaration 22) Used in section 21.
ParameterSignal::signalAfter code 10) Used in section 7.
ParameterSignal class declaration 2) Used in section 1.
ParameterSignal constructor code 8) Used in section 7.
ParameterSignal copy constructor code 9) Used in section 7.
PermutationScheme class declaration 79) Used in section 78.
ProductQuadrature::begin code 55) Used in section 44.
ProductQuadrature::designLevelForEvals code 56) Used in section 44.
ProductQuadrature class declaration 42) Used in section 40.
ProductQuadrature constructor 54) Used in section 44.
QMCSpecification class declaration 82) Used in section 78.
QMCCarloCubeQuadrature class declaration 84) Used in section 78.
QMCCarloNormalQuadrature class declaration 86) Used in section 78.
Quadrature::savePoints code 28) Used in section 26.
QuadratureImpl::integrate code 27) Used in section 26.
Quadrature class declaration 23) Used in section 21.
RadicalInverse::eval code 93) Used in section 91.
RadicalInverse::increase code 94) Used in section 91.
RadicalInverse::print code 95) Used in section 91.
RadicalInverse class declaration 89) Used in section 78.
RadicalInverse constructor code 92) Used in section 91.
ReversePerScheme::permute code 117) Used in section 91.
ReversePerScheme class declaration 88) Used in section 78.
SmolyakQuadrature::begin code 74) Used in section 62.
SmolyakQuadrature::calcNumEvaluations code 75) Used in section 62.
SmolyakQuadrature::designLevelForEvals code 76) Used in section 62.
Numerical Integration Module

\{SmolyakQuadrature::numEvals code 73\} Used in section 62.
\{SmolyakQuadrature class declaration 60\} Used in section 58.
\{SmolyakQuadrature constructor 72\} Cited in section 75. Used in section 62.
\{VectorFunctionSet class declaration 4\} Used in section 1.
\{VectorFunctionSet constructor 1 code 11\} Used in section 7.
\{VectorFunctionSet constructor 2 code 12\} Used in section 7.
\{VectorFunctionSet destructor code 13\} Used in section 7.
\{VectorFunction class declaration 3\} Used in section 1.
\{WarnockPerScheme::permute code 116\} Used in section 91.
\{WarnockPerScheme class declaration 87\} Used in section 78.
\{prodpit::operator++ code 51\} Used in section 44.
\{prodpit::operator≡ code 49\} Used in section 44.
\{prodpit::operator= code 50\} Used in section 44.
\{prodpit::print code 53\} Used in section 44.
\{prodpit::setPointAndWeight code 52\} Used in section 44.
\{prodpit destructor 48\} Used in section 44.
\{prodpit empty constructor 45\} Used in section 44.
\{prodpit regular constructor 46\} Used in section 44.
\{qmcnpit::operator++ code 115\} Used in section 91.
\{qmcnpit::operator= code 114\} Used in section 91.
\{qmcnpit class declaration 85\} Used in section 78.
\{qmcnpit copy constructor code 112\} Used in section 91.
\{qmcnpit destructor 113\} Used in section 91.
\{qmcnpit empty constructor code 110\} Used in section 91.
\{qmcnpit regular constructor code 111\} Used in section 91.
\{qmcpit::operator++ code 108\} Used in section 91.
\{qmcpit::operator≡ code 106\} Used in section 91.
\{qmcpit::operator= code 107\} Used in section 91.
\{qmcpit::weight code 109\} Used in section 91.
\{qmcpit class declaration 83\} Used in section 78.
\{qmcpit copy constructor code 104\} Used in section 91.
\{qmcpit destructor 105\} Used in section 91.
\{qmcpit empty constructor code 102\} Used in section 91.
\{qmcpit regular constructor code 103\} Used in section 91.
\{smolpit::operator++ code 69\} Used in section 62.
\{smolpit::operator≡ code 67\} Used in section 62.
\{smolpit::operator= code 68\} Used in section 62.
\{smolpit::print code 71\} Used in section 62.
\{smolpit::setPointAndWeight code 70\} Used in section 62.
\{smolpit class declaration 59\} Used in section 58.
\{smolpit copy constructor 65\} Used in section 62.
\{smolpit destructor 66\} Used in section 62.
\{smolpit empty constructor 63\} Used in section 62.
\{smolpit regular constructor 64\} Used in section 62.
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector function</td>
<td>1</td>
</tr>
<tr>
<td>Quadrature</td>
<td>2</td>
</tr>
<tr>
<td>Product quadrature</td>
<td>21</td>
</tr>
<tr>
<td>Smolyak quadrature</td>
<td>40</td>
</tr>
<tr>
<td>Quasi Monte Carlo quadrature</td>
<td>58</td>
</tr>
<tr>
<td></td>
<td>78</td>
</tr>
</tbody>
</table>