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 night 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;
  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.

3. 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 **const**. The hardcopies apply for parallelization.

```
⟨ VectorFunction class declaration 3 ⟩ ≡
  class 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; }
  };
```

This code is used in section 1.

4. This makes n copies of **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 paralelizing the code).

```
⟨ VectorFunctionSet class declaration 4 ⟩ ≡
  class 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( ); }
  };
```

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

$$\frac{1}{\sqrt{(2\pi)^n|\Sigma|}}\int f(x)e^{-\frac{1}{2}x^T\Sigma^{-1}x}\mathrm{d}x$$

then if we write $\Sigma = AA^T$ and $x = \sqrt{2}Ay$, we get integral

$$\frac{1}{\sqrt{(2\pi)^n|\Sigma|}}\int f\left(\sqrt{2}Ay\right)e^{-y^Ty}\sqrt{2^n}|A|\mathrm{d}y = \frac{1}{\sqrt{\pi^n}}\int f\left(\sqrt{2}Ay\right)e^{-y^Ty}\mathrm{d}y,$$

which means that a given function f we have to wrap to yield a function

$$g(y) = \frac{1}{\sqrt{\pi^n}}f\left(\sqrt{2}Ay\right).$$

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 seemlessly 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 desctructor; hence *delete_flag*. The second constructor takes a pointer to the function and differs from the first only by setting *delete_flag* to *true*.

⟨ **GaussConverterFunction** class declaration 5 ⟩ ≡

```
class GaussConverterFunction : public VectorFunction {
protected:
  VectorFunction *func;
  bool delete_flag;
  GeneralMatrix A;
  double multiplier;
public:
  GaussConverterFunction(VectorFunction &f, const GeneralMatrix &vcov);
  GaussConverterFunction(VectorFunction *f, const GeneralMatrix &vcov);
  GaussConverterFunction(const GaussConverterFunction &f);
  virtual ~GaussConverterFunction( )
  { if (delete_flag) delete func; }
  virtual VectorFunction *clone( ) const
  { return new GaussConverterFunction(*this); }
  virtual void eval(const Vector &point, const ParameterSignal &sig, Vector &out);
private:
  double calcMultiplier( ) const;
  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

```
#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
#ifdef __CROSS_COMPILATION__
#define M_PI  3.14159265358979323846
#endif
```

⟨ **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 14 ⟩;
⟨ **GaussConverterFunction** constructor code 2 15 ⟩;
⟨ **GaussConverterFunction** copy constructor code 16 ⟩;
⟨ **GaussConverterFunction** :: *eval* code 17 ⟩;
⟨ **GaussConverterFunction** :: *multiplier* code 18 ⟩;
⟨ **GaussConverterFunction** :: *calcCholeskyFactor* code 19 ⟩;

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

⟨ **ParameterSignal** constructor code 8 ⟩ ≡

```
ParameterSignal::ParameterSignal(int n)
: data(new bool[n]), num(n) {
  for (int i = 0; i < num; i++) data[i] = true;
}
```

This code is used in section 7.

9.

⟨ **ParameterSignal** copy constructor code 9 ⟩ ≡

```
ParameterSignal::ParameterSignal(const ParameterSignal &sig)
: data(new bool[sig.num]), num(sig.num) {
  memcpy(data, sig.data, num);
}
```

This code is used in section 7.

10. This sets *false* (no change) before a given parameter, and *true* (change) after the given parameter (including).

⟨ **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;
}
```
This code is used in section 7.

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

⟨ **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();
}
```
This code is used in section 7.

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

⟨ **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();
}
```
This code is used in section 7.

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

⟨ **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];
}
```
This code is used in section 7.

14. Here we construct the object from the given function f and given variance-covariance matrix $\Sigma =$*vcov*. The matrix A is calculated as lower triangular and yields $\Sigma = AA^T$.

⟨ **GaussConverterFunction** constructor code 1 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);
}
```
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.

⟨ **GaussConverterFunction** constructor code 2 15 ⟩ ≡
```
GaussConverterFunction::GaussConverterFunction(VectorFunction *f, const GeneralMatrix
        &vcov)
  : VectorFunction(*f), func(f), delete_flag(true), A(vcov.numRows(), vcov.numRows()),
      multiplier(calcMultiplier()) {      /* todo: raise if A.numRows() ≠ indim() */
    calcCholeskyFactor(vcov);
  }
```
This code is used in section 7.

16.

⟨ **GaussConverterFunction** copy constructor code 16 ⟩ ≡
```
GaussConverterFunction::GaussConverterFunction(const GaussConverterFunction &f)
  : VectorFunction(f), func(f.func→clone()), delete_flag(true), A(f.A), multiplier(f.multiplier) { }
```
This code is used in section 7.

17. Here we evaluate the function $g(y) = \frac{1}{\sqrt{\pi^n}} f\left(\sqrt{2}Ay\right)$. 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 *sig* of the input $y =$*point*.

⟨ **GaussConverterFunction**::*eval* code 17 ⟩ ≡
```
void GaussConverterFunction::eval(const Vector &point, const ParameterSignal &sig, Vector
        &out)
{
  ParameterSignal s(sig);
  int i = 0;
  while (i < indim() ∧ ¬sig[i]) i++;
  s.signalAfter(i);
  Vector x(indim());
  x.zeros();
  A.multaVec(x, point);
  x.mult(sqrt(2.0));
  func→eval(x, s, out);
  out.mult(multiplier);
}
```
This code is used in section 7.

18. This returns $\frac{1}{\sqrt{\pi^n}}$.

⟨ **GaussConverterFunction**::*multiplier* code 18 ⟩ ≡
```
double GaussConverterFunction::calcMultiplier() const
{
  return sqrt(pow(M_PI, -1 * indim()));
}
```
This code is used in section 7.

19.

```
⟨ GaussConverterFunction :: calcCholeskyFactor code 19 ⟩ ≡
  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

$$\int f(x)\mathrm{d}x \approx \sum_{i=1}^{N} w_i x_i$$

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 paralelism, 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.

```
#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.

```
⟨ OneDQuadrature class declaration 22 ⟩ ≡
  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 lelel, int i) const = 0;
  };
```

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 **VectorFunction** as an argument, creates locally **VectorFunctionSet** and do calculation, second one takes as an argument **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 apriori and thus it is not applicable for adaptive quadratures, second for Monte Carlo type of quadrature, the level is a number of evaluations.

```
⟨ Quadrature class declaration 23 ⟩ ≡
  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 level, int tn, Vector &out) const = 0;
    virtual void integrate(VectorFunctionSet &fs, int level, Vector &out) const = 0;
    virtual int numEvals(int level) const = 0;
  };
```

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 ⟩ ≡
  template⟨typename _Tpit⟩ class QuadratureImpl;
  template⟨typename _Tpit⟩ class IntegrationWorker : public THREAD {
    const QuadratureImpl⟨_Tpit⟩ &quad;
    VectorFunction &func;
    int level;
    int ti;
    int tn;
    Vector &outvec;
  public:
    IntegrationWorker(const QuadratureImpl⟨_Tpit⟩ &q, VectorFunction &f, int l, int tii, int
        tnn, Vector &out)
    : quad(q), func(f), level(l), ti(tii), tn(tnn), outvec(out) { }
    ⟨ 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 ⟩ ≡
  void operator()()
  {
    _Tpit beg = quad.begin(ti, tn, level);
    _Tpit end = quad.begin(ti + 1, tn, level);
    Vector tmpall(outvec.length());
    tmpall.zeros();
    Vector tmp(outvec.length());      /* note that since beg came from begin, it has empty signal */
      /* and first evaluation gets no signal */
    for (_Tpit run = beg; run ≠ end; ++run) {
      func.eval(run.point(), run.signal(), tmp);
      tmpall.add(run.weight(), tmp);
    }
    {
      SYNCHRO syn(&outvec, "IntegrationWorker");
      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 beginnning 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.

```
⟨ QuadratureImpl class declaration 26 ⟩ ≡
  template⟨typename _Tpit⟩ class QuadratureImpl : public Quadrature {
    friend class IntegrationWorker⟨_Tpit⟩;
  public:
    QuadratureImpl(int d)
    : Quadrature(d) { }
    ⟨ QuadratureImpl::integrate code 27 ⟩;
    void integrate(const VectorFunction &func, int level, int tn, Vector &out) const
    {
      VectorFunctionSet fs(func, tn);
      integrate(fs, level, out);
    }
    ⟨ Quadrature::savePoints code 28 ⟩;
    _Tpit start(int level) const
    { return begin(0, 1, level); }
    _Tpit end(int level) const
    { return begin(1, 1, level); }
  protected:
    virtual _Tpit begin(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 workes and run it.

```
⟨ QuadratureImpl::integrate code 27 ⟩ ≡
  void integrate(VectorFunctionSet &fs, int level, Vector &out) const
  {     /* todo: out.length()==func.outdim() */     /* todo: dim == func.indim() */
    out.zeros();
    THREAD_GROUP gr;
    for (int ti = 0; ti < fs.getNum(); ti++) {
      gr.insert(new IntegrationWorker⟨_Tpit⟩(*this, fs.getFunc(ti), level, ti, fs.getNum(), out));
    }
    gr.run();
  }
```

This code is used in section 26.

28. Just for debugging.

```
⟨ Quadrature::savePoints code 28 ⟩ ≡
  void savePoints(const char *fname, int level) const
  {
    FILE *fd;
    if (Λ ≡ (fd = fopen(fname, "w"))) {      /* todo: raise */
      fprintf(stderr, "Cannot␣open␣file␣%s␣for␣writing.\n", fname);
      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", run.weight());
      for (int i = 0; i < dimen(); i++) fprintf(fd, "\t%16.12g", run.point()[i]);
      fprintf(fd, "\n");
    }
    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.

```
⟨ 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

$$\int_{-\infty}^{\infty} x^k e^{-x^2} \mathrm{d}x$$

for level k.

Note that if pluging 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.

⟨ **GaussHermite** class declaration 30 ⟩ ≡
```
class GaussHermite : public OneDPrecalcQuadrature {
public:
  GaussHermite( );
};
```
This code is used in section 21.

31. Just precalculated Gauss–Legendre quadrature. This quadrature integrates exactly integrals

$$\int_0^1 x^k \mathrm{d}x$$

for level k.

Check `precalc_quadrature.dat` for available levels.

⟨ **GaussLegendre** class declaration 31 ⟩ ≡
```
class GaussLegendre : public OneDPrecalcQuadrature {
public:
  GaussLegendre( );
};
```
This code is used in section 21.

32. This is just an inverse cummulative density function of normal distribution. Its only method *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.

⟨ **NormalICDF** class declaration 32 ⟩ ≡
```
class NormalICDF {
public:
  static double get(double x);
};
```
This code is used in section 21.

33. End of `quadrature.h` file

34. This is `quadrature.cpp` file.

```
#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 $\langle 0,1\rangle$ to the draw from Gaussina $N(0,1)$. This is done by a table lookup, the table is given by *normal_icdf_step*, *normal_icfd_data*, *normal_icdf_num*, and a number *normal_icdf_end*. In order to avoid wrong tails for lookups close to zero or one, we rescale input x by $(1-2*(1-end))=2*end-1$.

```
⟨ NormalICDF get code 38 ⟩ ≡
  double NormalICDF::get(double x)
  {
    double xx = (2 * normal_icdf_end - 1) * std::abs(x - 0.5);
    int i = (int) floor(xx/normal_icdf_step);
    double xx1 = normal_icdf_step * i;
    double yy1 = normal_icdf_data[i];
    double y;
    if (i < normal_icdf_num - 1) {
      double yy2 = normal_icdf_data[i + 1];
      y = yy1 + (yy2 - yy1) * (xx - xx1)/normal_icdf_step;
    }
    else {      /* this should never happen */
      y = yy1;
    }
    if (x > 0.5) return y;
    else return -y;
  }
```

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 of k level and dimension d takes the form

$$Qf = \sum_{i_1=1}^{n_k} \dots \sum_{i_d=1}^{n^k} w_{i_1} \cdot \dots \cdot w_{i_d} f(x_{i_1}, \dots, x_{i_d})$$

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

$$Qf = (Q_k \otimes \dots \otimes Q_k) f$$

Here we define the product quadrature iterator **prodpit** and plug it into **QuadratureImpl** to obtains **ProductQuadrature**.

```
#ifndef PRODUCT_H
#define PRODUCT_H
#include "int_sequence.h"
#include "vector_function.h"
#include "quadrature.h"
  ⟨ prodpit class declaration 41 ⟩;
  ⟨ ProductQuadrature class declaration 42 ⟩;
#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 implmented as pointers in order to allow for empty constructor.

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

⟨ **prodpit** class declaration 41 ⟩ ≡

```
class ProductQuadrature;
class prodpit {
protected:
  const ProductQuadrature *prodq;
  int level;
  int npoints;
  IntSequence *jseq;
  bool end_flag;
  ParameterSignal *sig;
  Vector *p;
  double w;
public:
  prodpit( );
  prodpit(const ProductQuadrature &q, int j0, int l);
  prodpit(const prodpit &ppit);
  ~prodpit( );
  bool operator≡(const prodpit &ppit) const;
  bool operator≠(const prodpit &ppit) const
  { return ¬operator≡(ppit); }
  const prodpit &operator=(const prodpit &spit);
  prodpit &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 40.

42. The product quadrature is just **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 *designLevelForEvals* which for the given maximum number of evaluations (and dimension and underlying quadrature from the object) returns a maximum level yeilding number of evaluations less than the given number.

```
⟨ 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;
  };
```

This code is used in section 40.

43. End of `product.h` file

44. This is `product.cpp` file.

```
#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 ⟩;
```

45.

```
⟨ prodpit empty constructor 45 ⟩ ≡
  prodpit::prodpit( )
  : prodq(Λ), level(0), npoints(0), jseq(Λ), end_flag(true), sig(Λ), p(Λ) { }
```

This code is used in section 44.

46. This constructs a product iterator corresponding to index $(j0, 0 \ldots, 0)$.

```
⟨ prodpit regular constructor 46 ⟩ ≡
  prodpit::prodpit(const ProductQuadrature &q, int j0, int l)
  : prodq(&q), level(l), npoints(q.uquad.numPoints(l)), jseq(new IntSequence(q.dimen(),0)),
        end_flag(false), sig(new ParameterSignal(q.dimen())), p(new Vector(q.dimen())) {
    if (j0 < npoints) {
      (*jseq)[0] = j0;
      setPointAndWeight();
    }
    else {
      end_flag = true;
    }
  }
```

This code is used in section 44.

47. Copy constructor, clear.

```
⟨ prodpit copy constructor 47 ⟩ ≡
  prodpit::prodpit(const prodpit &ppit)
  : prodq(ppit.prodq), level(ppit.level), npoints(ppit.npoints), end_flag(ppit.end_flag), w(ppit.w) {
    if (ppit.jseq) jseq = new IntSequence(*(ppit.jseq));
    else jseq = Λ;
    if (ppit.sig) sig = new ParameterSignal(*(ppit.sig));
    else sig = Λ;
    if (ppit.p) p = new Vector(*(ppit.p));
    else p = Λ;
  }
```

This code is used in section 44.

48.

```
⟨ prodpit destructor 48 ⟩ ≡
  prodpit::~prodpit()
  {
    if (jseq) delete jseq;
    if (sig) delete sig;
    if (p) delete p;
  }
```

This code is used in section 44.

49.

```
⟨ prodpit::operator≡ code 49 ⟩ ≡
  bool prodpit::operator≡(const prodpit &ppit) const
  {
    bool ret = true;
    ret = ret & prodq ≡ ppit.prodq;
    ret = ret & end_flag ≡ ppit.end_flag;
    ret = ret & ((jseq ≡ Λ ∧ ppit.jseq ≡ Λ) ∨ (jseq ≠ Λ ∧ ppit.jseq ≠ Λ ∧ *jseq ≡ *(ppit.jseq)));
    return ret;
  }
```

This code is used in section 44.

50.

```
⟨ prodpit::operator= code 50 ⟩ ≡
  const prodpit &prodpit::operator=(const prodpit &ppit)
  {
    prodq = ppit.prodq;
    end_flag = ppit.end_flag;
    w = ppit.w;
    if (jseq) delete jseq;
    if (sig) delete sig;
    if (p) delete p;
    if (ppit.jseq) jseq = new IntSequence(*(ppit.jseq));
    else jseq = Λ;
    if (ppit.sig) sig = new ParameterSignal(*(ppit.sig));
    else sig = Λ;
    if (ppit.p) p = new Vector(*(ppit.p));
    else p = Λ;
    return *this;
  }
```

This code is used in section 44.

51.

```
⟨ prodpit::operator++ code 51 ⟩ ≡
  prodpit &prodpit::operator++()
  {    /* todo: throw if prodq ≡ Λ or jseq ≡ Λ or sig ≡ Λ or end_flag ≡ true */
    int i = prodq→dimen() − 1;
    (*jseq)[i]++;
    while (i ≥ 0 ∧ (*jseq)[i] ≡ npoints) {
      (*jseq)[i] = 0;
      i−−;
      if (i ≥ 0) (*jseq)[i]++;
    }
    sig→signalAfter(std::max(i, 0));
    if (i ≡ −1) end_flag = true;
    if (¬end_flag) setPointAndWeight();
    return *this;
  }
```

This code is used in section 44.

52. This calculates the weight and sets point coordinates from the indices.

```
⟨ prodpit::setPointAndWeight code 52 ⟩ ≡
  void prodpit::setPointAndWeight()
  {    /* todo: raise if prodq ≡ Λ or jseq ≡ Λ or sig ≡ Λ or */    /* p ≡ Λ or end_flag ≡ true */
    w = 1.0;
    for (int i = 0; i < prodq→dimen(); i++) {
      (*p)[i] = (prodq→uquad).point(level, (*jseq)[i]);
      w *= (prodq→uquad).weight(level, (*jseq)[i]);
    }
  }
```

This code is used in section 44.

53. Debug print.

```
⟨ prodpit::print code 53 ⟩ ≡
  void prodpit::print( ) const
  {
    printf("j=[");
    for (int i = 0; i < prodq¬dimen( ); i++) printf("%2d␣", (*jseq)[i]);
    printf("]␣%+4.3f*(", w);
    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⟨prodpit⟩(d), uquad(uq) {      /* todo: check d ≥ 1 */
  }
```

This code is used in section 44.

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

```
⟨ ProductQuadrature::begin code 55 ⟩ ≡
  prodpit 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 prodpit(*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 {
      lev++;
      last_evals = evals;
      evals = numEvals(lev);
    } while (lev < uquad.numLevels( ) − 2 ∧ evals < max_evals);
    lev−−;
    evals = last_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_l^1 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

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

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**.

```
#ifndef 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 ⟩;
#endif
```

59. 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 divideds all the iterators to subsets of equal size.

```
⟨ 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.

60. Here we define the class **SmolyakQuadrature**. It maintains an array of summands of the Smolyak quadrature formula:

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

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 cummulative number of evaluations. The latter two are added only for conveniency.

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 nuber 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**.

```
⟨ 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(Λ), isummand(0), jseq(Λ), sig(Λ), p(Λ) { }
```

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.

```
⟨ smolpit::operator++ code 69 ⟩ ≡
  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.

```
⟨ smolpit::setPointAndWeight code 70 ⟩ ≡
  void smolpit::setPointAndWeight( )
  {     /* todo: raise if smolq ≡ Λ or jseq ≡ Λ or sig ≡ Λ or
        /* p ≡ Λ or isummand ≥ smolq→numSummands( ) */
    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.

```
⟨ smolpit::print code 71 ⟩ ≡
  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]);
  }
```

This code is used in section 62.

72. Here is the constructor of **SmolyakQuadrature**. We have to setup *levels*, *levpoints* and *cumevals*. We have to go through all d-dimensional sequences k, such that $l \le |k| \le l + d - 1$ and all k_i are positive integers. This is equivalent to going through all k such that $l - d \le |k| \le l - 1$ 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.

```
⟨ SmolyakQuadrature constructor 72 ⟩ ≡
  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 ≥ d */      /* todo: check l ≥ uquad.miLevel(), l ≤ uquad.maxLevel() */
    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);
      }
    }
  }
```

This code is cited in section 75.

This code is used in section 62.

73. Here we return a number of evalutions 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.

```
⟨ SmolyakQuadrature::numEvals code 73 ⟩ ≡
  int SmolyakQuadrature::numEvals(int l) const
  {
    if (l ≠ level) return calcNumEvaluations(l);
    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*.

```
⟨ SmolyakQuadrature::begin code 74 ⟩ ≡
  smolpit SmolyakQuadrature::begin(int ti, int tn, int l) const
  {    /* todo: raise is level ≠ l */
    if (ti ≡ tn) return smolpit(*this, numSummands());
    int totevals = cumevals[numSummands() − 1];
    int evals = (totevals * ti)/tn;
    unsigned int isum = 0;
    while (isum + 1 < numSummands() ∧ cumevals[isum + 1] < evals) isum++;
    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.

```
⟨ SmolyakQuadrature::calcNumEvaluations code 75 ⟩ ≡
  int SmolyakQuadrature::calcNumEvaluations(int lev) const
  {
    int cum = 0;
    SymmetrySet ss(lev − 1, dim + 1);
    for (symiterator si(ss); ¬si.isEnd(); ++si) {
      if ((*si)[dim] ≤ dim − 1) {
        IntSequence lev((const IntSequence &) *si, 0, dim);
        lev.add(1);
        IntSequence levpts(dim);
        for (int i = 0; i < dim; i++) levpts[i] = uquad.numPoints(lev[i]);
        cum += levpts.mult();
      }
    }
    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.

```
⟨ SmolyakQuadrature::designLevelForEvals code 76 ⟩ ≡
  void SmolyakQuadrature::designLevelForEvals(int max_evals, int &lev, int &evals) const
  {
    int last_evals;
    evals = 1;
    lev = 1;
    do {
      lev++;
      last_evals = evals;
      evals = calcNumEvaluations(lev);
    } while (lev < uquad.numLevels( ) ∧ evals ≤ max_evals);
    lev--;
    evals = 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:

$$\int_{\langle 0,1\rangle^n} f(x)\mathrm{d}x$$

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

$$\frac{1}{\sqrt{(2\pi)^n}}\int_{(-\infty,\infty)^n} f(x)e^{-\frac{1}{2}x^Tx}\mathrm{d}x$$

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; *PermutattionScheme* is an abstract class for all permutaton schemes. We have three implementations: **WarnockPerScheme**, **ReversePerScheme**, and **IdentityPerScheme**.

```
#ifndef QUASI_MCARLO_H
#define QUASI_MCARLO_H
#include "int_sequence.h"
#include "quadrature.h"
#include "Vector.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 $0,\ldots,base-1$.

```
⟨ 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.

80. This class represents an integer number *num* as $c_0+c_1b+c_2b^2+\ldots+c_jb^j$, 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 ⟩ ≡

```
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 **RadicalInverse**s, 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 **RadicalInverse**s and sets point *pt* to contain the points in each dimension.

⟨ **HaltonSequence** class declaration 81 ⟩ ≡

```
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.

```
⟨ QMCSpecification class declaration 82 ⟩ ≡
  class QMCSpecification {
  protected:
    int dim;
    int lev;
    const PermutationScheme &per_scheme;
  public:
    QMCSpecification(int d, int l, const PermutationScheme &p)
    : dim(d), lev(l), per_scheme(p) { }
    virtual ~QMCSpecification( )
    { }
    int dimen( ) const
    { return dim; }
    int level( ) const
    { return lev; }
    const PermutationScheme &getPerScheme( ) const
    { return per_scheme; }
  };
```

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 &s, 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 $\langle 0,1\rangle \to \langle -\infty,\infty\rangle$ aplied to all dimensions. The transformation yields a normal distribution from a uniform distribution on $\langle 0,1\rangle$. It is in fact **NormalICDF**.

```
⟨ qmcnpit class declaration 85 ⟩ ≡
  class qmcnpit : public qmcpit {
  protected:
    Vector *pnt;
  public:
    qmcnpit( );
    qmcnpit(const QMCSpecification &spec, int n);
    qmcnpit(const qmcnpit &qpit);
    ~qmcnpit( );
    bool operator≡(const qmcnpit &qpit) const
    { return qmcpit::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 **qmcnpit**, so we only inherit from general **Quadrature** and reimplement *begin* and *numEvals*.

```
⟨ QMCarloNormalQuadrature class declaration 86 ⟩ ≡
  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.

```
⟨ WarnockPerScheme class declaration 87 ⟩ ≡
  class WarnockPerScheme : public PermutationScheme {
  public:
    int permute(int i, int base, int c) const;
  };
```

This code is used in section 78.

88. Declares reverse permutation scheme.

```
⟨ ReversePerScheme class declaration 88 ⟩ ≡
  class ReversePerScheme : public PermutationScheme {
  public:
    int permute(int i, int base, int c) const;
  };
```

This code is used in section 78.

89. Declares no permutation (identity) scheme.

```
⟨ IdentityPerScheme class declaration 89 ⟩ ≡
  class IdentityPerScheme : public PermutationScheme {
  public:
    int permute(int i, int base, int c) const
    { return c; }
  };
```

This code is used in section 78.

90. End of `quasi_mcarlo.h` file

91. This is `quasi_mcarlo.cpp` file.

```
#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 ⟩;
⟨ qmcnpit empty constructor code 110 ⟩;
⟨ qmcnpit regular constructor code 111 ⟩;
⟨ qmcnpit copy constructor code 112 ⟩;
⟨ qmcnpit destructor 113 ⟩;
⟨ qmcnpit::operator= code 114 ⟩;
⟨ qmcnpit::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.

```
⟨ 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.

93. 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

$$\left(\ldots\left(\left(\frac{c_j}{b}\cdot\frac{1}{b} + \frac{c_{j-1}}{b}\right)\cdot\frac{1}{b} + \frac{c_{j-2}}{b}\right)\ldots\right)\cdot\frac{1}{b} + \frac{c_0}{b}$$

Now with permutation π, we have

$$\left(\ldots\left(\left(\frac{\pi(c_j)}{b}\cdot\frac{1}{b} + \frac{\pi(c_{j-1})}{b}\right)\cdot\frac{1}{b} + \frac{\pi(c_{j-2})}{b}\right)\ldots\right)\cdot\frac{1}{b} + \frac{\pi(c_0)}{b}$$

⟨ **RadicalInverse** :: *eval* code 93 ⟩ ≡

```
double RadicalInverse::eval(const PermutationScheme &p) const
{
  double res = 0;
  for (int i = j; i ≥ 0; i--) {
    int cper = p.permute(i, base, coeff[i]);
    res = (cper + res)/base;
  }
  return res;
}
```

This code is used in section 91.

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

⟨ **RadicalInverse** :: *increase* code 94 ⟩ ≡

```
void RadicalInverse::increase( )
{   /* todo: raise if num + 1 > maxn */
  num++;
  int i = 0;
  coeff[i]++;
  while (coeff[i] ≡ base) {
    coeff[i] = 0;
    coeff[++i]++;
  }
  if (i > j) j = i;
}
```

This code is used in section 91.

95. Debug print.

⟨ **RadicalInverse** :: *print* code 95 ⟩ ≡

```
void RadicalInverse::print( ) const
{
  printf("n=%d␣b=%d␣c=", num, base);
  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.

⟨ **HaltonSequence** static data 96 ⟩ ≡
```
int HaltonSequence::num_primes = 170;
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,
    199, 211, 223, 227, 229, 233, 239, 241, 251, 257, 263, 269, 271, 277, 281, 283, 293, 307, 311, 313, 317, 331,
    337, 347, 349, 353, 359, 367, 373, 379, 383, 389, 397, 401, 409, 419, 421, 431, 433, 439, 443, 449, 457, 461,
    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 *eval*.

⟨ **HaltonSequence** constructor code 97 ⟩ ≡
```
HaltonSequence::HaltonSequence(int n, int mxn, int dim, const PermutationScheme &p)
: num(n), maxn(mxn), per(p), pt(dim) {      /* todo: raise if dim > num_primes */
    /* todo: raise if n > mxn */
  for (int i = 0; i < dim; i++) ri.push_back(RadicalInverse(num, primes[i], maxn));
  eval();
}
```
This code is used in section 91.

98.

⟨ **HaltonSequence::operator=** code 98 ⟩ ≡
```
const HaltonSequence &HaltonSequence::operator=(const HaltonSequence &hs)
{
  num = hs.num;
  maxn = hs.maxn;
  ri.clear();
  for (unsigned int i = 0; i < hs.ri.size(); i++) ri.push_back(RadicalInverse(hs.ri[i]));
  pt = hs.pt;
  return *this;
}
```
This code is used in section 91.

99. This calls **RadicalInverse**::*increase* for all radical inverses and calls *eval*.

⟨ **HaltonSequence**::*increase* code 99 ⟩ ≡
```
void HaltonSequence::increase()
{
  for (unsigned int i = 0; i < ri.size(); i++) ri[i].increase();
  num++;
  if (num ≤ maxn) eval();
}
```
This code is used in section 91.

100. This sets point pt to radical inverse evaluations in each dimension.

```
⟨ HaltonSequence::eval code 100 ⟩ ≡
  void HaltonSequence::eval( )
  {
    for (unsigned int i = 0; i < ri.size( ); i++) pt[i] = ri[i].eval(per);
  }
```

This code is used in section 91.

101. Debug print.

```
⟨ HaltonSequence::print code 101 ⟩ ≡
  void HaltonSequence::print( ) const
  {
    for (unsigned int i = 0; i < ri.size( ); i++) ri[i].print( );
    printf("point=[␣");
    for (unsigned int i = 0; i < ri.size( ); i++) printf("%7.6f␣", pt[i]);
    printf("]\n");
  }
```

This code is used in section 91.

102.

```
⟨ qmcpit empty constructor code 102 ⟩ ≡
  qmcpit::qmcpit( )
  : spec(Λ), halton(Λ), sig(Λ) { }
```

This code is used in section 91.

103.

```
⟨ qmcpit regular constructor code 103 ⟩ ≡
  qmcpit::qmcpit(const QMCSpecification &s, int n)
  : 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.

```
⟨ qmcpit copy constructor code 104 ⟩ ≡
  qmcpit::qmcpit(const qmcpit &qpit)
  : spec(qpit.spec), halton(Λ), sig(Λ) {
    if (qpit.halton) halton = new HaltonSequence(*(qpit.halton));
    if (qpit.sig) sig = new ParameterSignal(qpit.spec→dimen( ));
  }
```

This code is used in section 91.

105.

```
⟨ qmcpit destructor 105 ⟩ ≡
  qmcpit::~qmcpit( )
  {
    if (halton) delete halton;
    if (sig) delete sig;
  }
```

This code is used in section 91.

106.

```
⟨ qmcpit::operator≡ code 106 ⟩ ≡
  bool qmcpit::operator≡(const qmcpit &qpit) const
  {
    return (spec ≡ qpit.spec) ∧ ((halton ≡ Λ ∧ qpit.halton ≡ Λ) ∨ (halton ≠ Λ ∧ qpit.halton ≠
        Λ ∧ halton→getNum( ) ≡ qpit.halton→getNum( )));
  }
```

This code is used in section 91.

107.

```
⟨ qmcpit::operator= code 107 ⟩ ≡
  const qmcpit &qmcpit::operator=(const qmcpit &qpit)
  {
    spec = qpit.spec;
    if (halton) delete halton;
    if (qpit.halton) halton = new HaltonSequence(∗(qpit.halton));
    else halton = Λ;
    return ∗this;
  }
```

This code is used in section 91.

108.

```
⟨ qmcpit::operator++ code 108 ⟩ ≡
  qmcpit &qmcpit::operator++( )
  {     /∗ todo: raise if halton ≡ null ∨ qmcq ≡ Λ ∗/
    halton→increase( );
    return ∗this;
  }
```

This code is used in section 91.

109.

```
⟨ qmcpit::weight code 109 ⟩ ≡
  double qmcpit::weight( ) const
  {
    return 1.0/spec→level( );
  }
```

This code is used in section 91.

110.

```
⟨ qmcnpit empty constructor code 110 ⟩ ≡
  qmcnpit::qmcnpit( )
  : qmcpit( ), pnt(Λ) { }
```

This code is used in section 91.

111.

```
⟨ qmcnpit regular constructor code 111 ⟩ ≡
  qmcnpit::qmcnpit(const QMCSpecification &s, int n)
  : qmcpit(s, n), pnt(new Vector(s.dimen( ))) { }
```

This code is used in section 91.

112.

```
⟨ qmcnpit copy constructor code 112 ⟩ ≡
  qmcnpit::qmcnpit(const qmcnpit &qpit)
  : qmcpit(qpit), pnt(Λ) {
    if (qpit.pnt) pnt = new Vector(*(qpit.pnt));
  }
```

This code is used in section 91.

113.

```
⟨ qmcnpit destructor 113 ⟩ ≡
  qmcnpit::~qmcnpit( )
  {
    if (pnt) delete pnt;
  }
```

This code is used in section 91.

114.

```
⟨ qmcnpit::operator= code 114 ⟩ ≡
  const qmcnpit &qmcnpit::operator=(const qmcnpit &qpit)
  {
    qmcpit::operator=(qpit);
    if (pnt) delete pnt;
    if (qpit.pnt) pnt = new Vector(*(qpit.pnt));
    else pnt = Λ;
    return *this;
  }
```

This code is used in section 91.

115. Here we inccrease a point in Halton sequence ant then store images of the points in **NormalICDF** function.

```
⟨ qmcnpit::operator++ code 115 ⟩ ≡
  qmcnpit &qmcnpit::operator++( )
  {
    qmcpit::operator++( );
    for (int i = 0; i < halton→point( ).length( ); i++) (*pnt)[i] = NormalICDF::get(halton→point( )[i]);
    return *this;
  }
```

This code is used in section 91.

116. Clear from code.

```
⟨ WarnockPerScheme::permute code 116 ⟩ ≡
  int WarnockPerScheme::permute(int i, int base, int c) const
  {
    return (c + i) % base;
  }
```

This code is used in section 91.

117. Clear from code.

```
⟨ ReversePerScheme::permute code 117 ⟩ ≡
  int ReversePerScheme::permute(int i, int base, int c) const
  {
    return (base − c) % base;
  }
```

This code is used in section 91.

118. End of `quasi_mcarlo.cpp` file.

⟨ **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.
⟨ **QMCarloCubeQuadrature** class declaration 84 ⟩ Used in section 78.
⟨ **QMCarloNormalQuadrature** class declaration 86 ⟩ Used in section 78.
⟨ **Quadrature** :: *savePoints* code 28 ⟩ Used in section 26.
⟨ **QuadratureImpl** :: *integrate* code 27 ⟩ Used in section 26.
⟨ **QuadratureImpl** class declaration 26 ⟩ Cited in section 24. Used in section 21.
⟨ **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 80 ⟩ 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.

⟨ **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** class declaration 41 ⟩ Used in section 40.
⟨ **prodpit** copy constructor 47 ⟩ 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.

Numerical Integration Module