dVegas User's Guide

Nikolas Kauer
<nkauer@users.sourceforge.net>

Version 1.1.0, January 2002

Table of Contents

- Next chapter
- Table of contents

Chapter 1: Introduction

1.1: About dVegas

dVegas is a library for adaptive Monte Carlo integration based on the VEGAS algorithm invented by G.P. Lepage. The "d" in dVegas stands for "discrete" and "decoded", because it extends Lepage's original VEGAS program in that it can not only handle continuous, but also discrete dimensions. dVegas is written as a C++ class in an object-oriented programming style, in an attempt to make its structure transparent and manifest. dVegas code was written to be robust and accessible, thus being more suited for customization and extension than the somewhat cryptic "black box" code of the original VEGAS program and its derivatives. For this purpose, dVegas was rewritten from scratch, with carefully chosen, meaningful variable names and comments. Its functionality is also made available to Fortran and C programmers through special interfaces. A parallel dVegas library is provided for use on multiprocessor machines.

A detailed description of G.P. Lepage's algorithm, its implementation in dVegas, as well as comparisons with and information about several earlier implementations, can be found in the references listed in section 8.

1.2: Platform notes

For up-to-date information regarding your platform consider visiting the dVegas Web site:
http://hepsource.org/dvegas/platforms.html

1.2.1: GNU Compiler Collection (GCC)

The dVegas libraries will compile and link with recent releases of the GNU compiler collection (GCC). To build all libraries g++, g77 and gcc are needed.

1.2.1.1: Intel/Linux

On Intel/Linux 2.2.5, gcc version egcs-2.91 (formely egcs-1.1.2) is known to work. Use the `-fno-second-underscore` option with g77.

1.2.2: Commercial compilers

1.2.2.1: Compaq/DEC

On Compaq Alpha platforms, the Compaq (formerly Digital) C++, Fortran and C compilers (cxx, f77, cc) should work. Success has been reported for Compaq C++ V6.2 and Compaq Fortran V5.3 on Alpha/Tru64 Unix 4.0F.

1.3: How to get dVegas

The distribution can be downloaded here:
http://sourceforge.net/projects/hepsource/

1.4: Installation

Obtain the source distribution as described in section 1.3. Then unpack the distribution tarball, for example with

```
gunzip -c dvegas-1.0.0.tar.gz | tar xf -
```

and change into the top directory. Installation proceeds in four steps

1. Configuration: `./configure [--enable-fortran] [--enable-c] [--enable-getput]`
2. Build: `make`
3. Test: `make test`
4. Installation: `mv lib/*` *destdir*

which are described in more detail in the following sections.

The installation process works best with GNU **make**. If GNU **make** is available on your system I recommend you use GNU make instead of the vendor-supplied **make** program. If you have to use the latter consult section 1.4.6 for more information. If you can use GNU **make**, make sure that it is found before the vendor-supplied **make** and named 'make', so that the configure script can automatically detect it. You can use these commands, for example for tcsh:

```
setenv PATH ~/bin:${PATH}
ln -s /usr/local/bin/gmake ~/bin/make
```

1.4.1: Dependencies

dVegas needs random numbers to perform the numerical integration. The library, hence, relies on a random number generator that generates one or more random number streams.

Currently dVegas uses the CLHEP Random package to generate pseudo random numbers. If the CLHEP header files and library are installed in your compilers' search

paths and you are using GNU make everything should be automatic. If you do have the CLHEP library and header files installed, but they are not automatically included by the linker, just set the variables LIBDIRCLHEP and INCLDIRCLHEP in the top-level Makefile. Otherwise, you need to obtain a CLHEP distribution for your platform (or the source code) from

http://wwwinfo.cern.ch/asd/lhc++/DISTRIBUTION/clhep.html

first and install it on your system (or ask your system administrator to do it). If you're not using GNU make and your compilers do not find the CLHEP files automatically, you'll have to modify all Makefiles. See section 1.4.6 for details.

Note that dVegas is not tied to CLHEP, and other random number generators can be integrated easily if the need arises.

1.4.2: Configuration

Remove traces of previous configurations with
`make distclean`

Then configure your distribution with
`./configure` *[options]*

To see a list of all options with brief descriptions do:
`./configure --help`

Special options for dVegas:

`--enable-fortran`
: build Fortran interface

`--enable-c`
: build C interface

`--enable-getput`
: build "get/put" Fortran or C interface (serial only)

`--with-gnu`
: use GNU compiler collection

The options `--enable-fortran` and `--enable-c` are exclusive. For details about `--enable-getput` see section 3.2. It is currently only available for the Fortran interface, but can be made available for the C interface on demand. The configure script tries to detect a viable compiler combination automatically. If you want to use GCC, but it is not automatically selected, you should specify the `--with-gnu` option when running configure. If you want to select the compilers used to build dVegas manually see section 1.4.6 for more information.

1.4.3: Build

`make`
builds one or more static libraries and copies them into the *lib* subdirectory. Consult section 1.4.6 if something goes wrong here.

1.4.4: Test

The libraries built in the previous step can be tested with small sample programs with `make test`

This command creates a test executable for every library and tries to run it on your system. If the exit code indicates an error occured, the test fails. The diligent reader is encouraged to compare the obtained results with the expected results, which are recorded in the source files under subdirectory *test*.

1.4.5: Installation

The final installation step is to move the libraries in subdirectory *lib* into the library search path of the compiler (or use the `-L` linker flag to include this directory in the search path). Note that there is no make target `install`.

If you plan on using the C++ or C libraries you also have to copy the header files in subdirectory *include* to a suitable directory, so that the compiler can find them.

1.4.6: Troubleshooting

For up-to-date information about build problems and solutions consult the dVegas Web site:
http://hepsource.org/dvegas/troubleshooting.html

If you are using a vendor-supplied **make** that supports the *export* keyword, then just uncomment the corresponding line in the top-level Makefile. Otherwise certain variables will not be exported and you may have to define them in not only in the top-level Makefile, but also in the Makefiles in subdirectories. This applies particularly to the variables `INCLDIRCLHEP`, `LIBCLHEP` and `LIBDIRCLHEP` if the CLHEP files are not found automatically.

You can preselect specific compilers on your system to be used to build dVegas, by defining these variables:
`CXX CXXCPP F77 CC`

The first two have to be defined. Here's an example:

```
CXX=cxx CXXCPP=/lib/cpp F77=f77 ./configure --enable-fortran
```

1.5: Compiling and Linking with dVegas

1.5.1: C++

Fast track: read last paragraph

Compiling and linking C++ programs with dVegas should be straightforward. Just append `-ldvegas` to the compiler command. If your compiler cannot find the library you need to specify the path first: `-L`*path* `-ldvegas`

If you want to run your program in parallel mode on a multiprocessor machine, you have to link with the pthreads version `-ldvegas_r`. Some compilers will automatically link with the right version if you have `-ldvegas` specified. See section 5.1 for platform-specific details. Depending on the compiler you might have to link explicitly with libpthread or set a compiler flag like `-pthread`.

If the dVegas library utilizes CLHEP---typically it will---you also need to link to the CLHEP library. You need to specify the library after the dVegas library like in this example: `-ldvegas -L/usr/local/lib/CLHEP -lCLHEP`

To get started just run `make test` in the top-level directory. It will compile and link one or more test programs displaying the commands it uses. If the tests pass you can most likely just adapt these commands for your own programs.

1.5.2: Fortran

Fast track: read last paragraph

Compiling and linking Fortran programs with dVegas is slightly more complicated. You need to link to the Fortran dVegas library: `-lf_dvegas` or `-L`*path* `-lf_dvegas`. In addition the Fortran dVegas library needs to resolve references to standard C++ libraries that have to be specified after it, e.g. `-lf_dvegas -L/usr/lib/cmplrs/cxx -lcxxstd -lcxx -lexc`

If the dVegas library utilizes CLHEP---typically it will---you also need to link to the CLHEP library. A complete appendix will then look something like this: `-lf_dvegas -L/usr/local/lib/CLHEP -lCLHEP -L/usr/lib/cmplrs/cxx -lcxxstd -lcxx -lexc`

If you want to run your program in parallel mode on a multiprocessor machine, you have to link with the pthreads version `-lf_dvegas_r`. Some compilers will automatically link with the right version if you have `-lf_dvegas` specified. See section 5.1 for platform-specific details. Depending on the compiler you might have to link explicitly with libpthread or set a compiler flag like `-pthread`.

To get started just run `make test` in the top-level directory. It will compile and link one or more Fortran test programs displaying the commands it uses. If the tests pass you can most likely just adapt these commands for your own programs. An attempt is made to figure out the right standard C++ libraries automatically. If the test programs don't link on your system search for comments in section 1.2. If you don't find answers there, see section 1.6.

1.5.3: C

Fast track: read last paragraph

Compiling and linking C programs with dVegas is slightly more complicated. You need to link to the C dVegas library: `-lc_dvegas` or `-L`*path* `-lc_dvegas`. In addition the C dVegas library needs to resolve references to standard C++ libraries that have to be specified after it, e.g. `-lc_dvegas -L/usr/lib/cmplrs/cxx -lcxxstd -lcxx -lexc`

If the dVegas library utilizes CLHEP---typically it will---you also need to link to the CLHEP library. A complete appendix will then look something like this: `-lc_dvegas -L/usr/local/lib/CLHEP -lCLHEP -L/usr/lib/cmplrs/cxx -lcxxstd -lcxx -lexc`

If you want to run your program in parallel mode on a multiprocessor machine, you have to link with the pthreads version `-lc_dvegas_r`. Some compilers will automatically link with the right version if you have `-lc_dvegas` specified. See section 5.1 for platform-specific details. Depending on the compiler you might have to link explicitly with libpthread or set a compiler flag like `-pthread`.

To get started just run `make test` in the top-level directory. It will compile and link one or more C test programs displaying the commands it uses. If the tests pass you can most likely just adapt these commands for your own programs. An attempt is made to figure out the right standard C++ libraries automatically. If the test programs don't link on your system search for comments in section 1.2. If you don't find answers there, see section 1.6.

1.6: Support

Problem reports, comments and suggestions regarding this dVegas distribution should be emailed to *nkauer@users.sourceforge.net*

1.7: Porting dVegas

If you plan to test this dVegas distribution on a new platform or with a new compiler, please contact *nkauer@users.sourceforge.net*.

1.8: LGPL license

- Next chapter
- Table of contents

Chapter 2: C++ Class

2.1: Constructor and main member functions

2.1.1: Constructor

```
Vegas(int cdim, int nsdim, int cnbin, int ddim, int adim,
      const int nvals[], int fn, Integrand fxn = 0, int ncpu = 1);
```

cdim: number of continuous dimensions
nsdim: number of non-separable continuous dimensions (first nsdim dimensions)
cnbin: number of bins for each continuous dimension
ddim: number of discrete dimensions
adim: number of auxilliary dimensions (no adaptation)
nvals[]: number of values for each discrete dimension (see below)
fn: number of functions to integrate, (weights are adapted for first function)
fxn: pointer to function (see next section)
ncpu: number of CPUs, default: 1 (see section 5.1)

The default integration/summation range is

- for continuous dimensions: (0, 1)

- for discrete dimensions: {0, 1, 2, ..., nvals[dim]-1}

Random numbers in (0, 1) are provided to facilitate the addition of auxilliary dimensions that are not VEGAS driven.

The parameters `ddim` or `adim` can be set to zero if no discrete or auxilliary dimensions are needed.

The parameters of a constructed Vegas object can be modified with two member functions:

```
void set_parameters(int cdim, int cnbin, int ddim, int adim, const int nvals[], int f

void set_integrand(Integrand fxn);
```

Note that these functions are questionable from a design perspective and may not be supported in the next major version, i.e. versions 2.0.0 and higher.

If you plan to use the `get` and `put` methods (see section 2.6) directly rather than the `loop` method, just pass 0 for fxn.

2.1.2: pl_vegas

If a function pointer `fxn` of type `Integrand`:

```
typedef void (*Integrand)(const double x[], const int k[], const double& wgt,
                          const double aux[], double f[]);
```

is passed to the constructor or `set_integrand` then the function `pl_vegas` runs through `iterats` iterations with `ncall` shots each when called, modeling the behavior of Peter Lepage's VEGAS:

```
void pl_vegas(Vegas& vegas, int64 ncall, int iterats, int init = 0);
```

The array `x[]` contains the random numbers for the continuous dimensions, the array `k[]` contains the random numbers for the discrete dimensions, the array `aux[]` contains the auxilliary random numbers. The VEGAS weight (For each iteration the sum of the weights is normalized to 1.) is passed in variable `wgt`, so that it can be passed on to functions that fill histograms, for example. The function `*fxn` is expected to fill the array `f[]` with the function values that correspond to the arguments

`x[0]`, `x[1]`, ..., `k[0]`, `k[1]`, ... and `aux[0]`, `aux[1]`, ... The value of the first function is assigned to `f[0]`. The integration is optimized for this function according to the VEGAS algorithm. Secondary functions can simultaneously be integrated and their values are expected to be returned as `f[1]`, `f[2]`, etc.

The parameter `init` has the same meaning as in the original VEGAS program:

init = 0: fresh start (no prior information)
init = 1: use previous adaptation info, but discard shot data
init = 2: use previous adaptation info and build on previous shots data

The generic action for a single iteration is encapsulated in the member function:

```
void loop(int64 ncall_);
```

Note that `ncall`'s type is **int64**, corresponding to a 64-bit integer. **int64** might be a typedef to **long int** or **long long int** depending on your system. You can find that out by looking at the values of the macros `SIZE_LONG` and `SIZE_LONG_LONG`.

Both, class `Vegas` and the associated function `pl_vegas` are enclosed in namespace `MonteCarlo`.

2.1.3: Accessing results

Methods are provided to obtain the results/statistics for each integrated function `fn` (since the last adaptation):

```
double get_integral(int fn) const ;

double get_error(int fn) const ;

double get_redchi2(int fn) const ; // "reduced chi^2" = chi^2/(nr of estimates-1)

int get_niter() const ; // number of iterations (estimates)
```

or accumulated over all iterations:

```
double get_acc_integral(int fn) const ;

double get_acc_error(int fn) const ;

double get_acc_redchi2(int fn) const ;

int get_acc_niter() const ;
```

2.1.4: Saving and restoring the state of a Vegas object

The state of a Vegas object can be saved to file and restored later with the following methods:

```
void save(const char* filename = "vegas.dat") const ;

void save(std::ofstream& fout) const ;

void restore(Integrand fxn, const char* filename = "vegas.dat");

void restore(Integrand fxn, std::ifstream& fin);

void restore(std::ifstream& fin); // integrand set with set_integrand()
```

Saved are the parameters that can be set with `set_parameters` and all weights. This does not include the function pointer of type `Integrand` and the number of CPUs. The `restore` method does not check if the function to be integrated is the same as when `save` was called.

2.2: Sample program

```
#ifdef NO_CXX_HEADERS_FOR_C_LIB
#include <math.h>
#else
#include <cmath>
#endif
#include "dvegas.h"

#ifndef NO_CXX_HEADERS_FOR_C_LIB
// <cmath>
using std::pow ;
using std::sqrt ;
using std::exp ;
#endif

void fxn(const double x[], const int k[], const double& wgt, const double aux[], doub
{
  f[0] = x[0]*x[1]*x[2]*x[3]*x[4]*pow(2.0,5);
}

void fxn1(const double x[], const int k[], const double& wgt, const double aux[], dou
{
  if (k[0] < 99)
    f[0] =   1 * x[0]*x[1]*x[2]*x[3]*x[4]*pow(2.0,5);
  else if (k[0] == 99)
    f[0] = 100 * x[0]*x[1]*x[2]*x[3]*x[4]*pow(2.0,5);
}

int main()
```

```
{
  using MonteCarlo::Vegas ;
  using MonteCarlo::pl_vegas ;

  int nvals[2];
  Vegas vegas(5, 50, 0, 0, nvals, 1, &fxn);
  pl_vegas(vegas, 100000, 2); // result: 1.0

  nvals[0] = 100 ;
  vegas.set_parameters(5, 50, 1, 0, nvals, 1);
  vegas.set_integrand(&fxn1);
  pl_vegas(vegas, 100000, 1); // result: 199.0, factor sqrt(100)=10 error reduction w
  vegas.save();
  pl_vegas(vegas, 100000, 1, 1);

  nvals[1] = 3 ;
  vegas.set_parameters(5, 50, 2, 0, nvals, 1);
  pl_vegas(vegas, 100000, 2); // result: 597.0

  vegas.restore(&fxn1);
  vegas.save("same.dat");
  pl_vegas(vegas, 100000, 1, 1); // result: 199.0
}
```

2.3: Limit macros

VEGAS_CONT_DIMMX

maximum number of continuous dimensions, default: 20

VEGAS_CONT_NBINMX

maximum number of bins for continuous dimensions, default: 50

VEGAS_DISC_DIMMX

maximum number of discrete dimensions, default: 5

VEGAS_DISC_NBINMX

maximum number of values for discrete dimensions, default: 100

VEGAS_AUX_DIMMX

maximum number of auxilliary dimensions (no adaptation), default: 5

VEGAS_FNMX

maximum number of functions to integrate, default: 5

2.4: Numeric types

The following type definitions are used in the class and function definitions:

```
#if SIZEOF_LONG == 8
typedef long int    int64 ;
#elif SIZEOF_LONG_LONG == 8
typedef long long int int64 ;
#endif

typedef double      float64 ;
```

The type `int64` is a 64-bit integer and is used to allow a maximum number of 9,223,372,036,854,755,807. (The typical 32-bit integer has a maximum of about 2 billion, which I found to be too small in some cases to achieve the desired precision.) The type `float64` is a 64-bit floating point number and is used in some member functions for variables that accumulate a large number of temporary results (of the order of `ncall`). 32-bit floating point numbers provide only a precision of typically 7 decimal digits, which can be insufficient when adding about 100 million numbers or more. For 64-bit floating point numbers with a precision of typically 15 decimal digits this problem occurs roughly at 10^16 additions. On most systems the type

`double` corresponds to 64-bit floating point numbers. In rare cases `float64` might correspond to `long double`. (Note that these problems can also occur in histogram packages.)

In a future dVegas release the issues addressed above will be double-checked in the `pre_loop` member function, and a warning or error will be issued when problems could arise. This kind of checking is facilitated in C++ by specializations of the numeric_limits template presented in <*limits*>. See section 22.2, Numeric Limits, in Bjarne Stroustrup's *The C++ programming language*, 3rd ed.

2.5: Enumerations

The enumeration `Sampling` specifies sampling techniques:

```
enum Sampling { importance, stratified } ;
```

Currently only importance sampling is implemented, since I use dVegas for calculations with high numbers of dimensions. Additional stratified sampling can lead to a substantially improved efficiency in low dimensions (cdim = 1, ..., 4) and will be implemented if the need arises.

The enumerations `Reset` and `Info` hold sets of options. They allow to specify elementary or composite options by name rather than a crypic (hexadecimal) number. The state of each elementary option is stored in a bit that is part of a *bit set* of related options.

The enumeration `Reset` specifies reset options:

```
enum Reset { r_data=0x1, r_weights=0x2, r_none=0x0, r_all=0x3 };
```

The enumeration `Info` holds a set of options for the `info` member function, which prints out information about the most recent iteration, as well as accumulated results and statistics. The `info` member function is a primary target for customization and the options listed below are intended to give an idea of the possibilities:

```
enum Info  { i_spec     =0x0001, i_int     =0x0002, i_adapt     =0x0004, i_grid     =
             i_spec_more=0x0010, i_int_more=0x0020, i_adapt_more=0x0040, i_grid_more=
```

```
            i_spec_all =0x0011, i_int_all =0x0022, i_adapt_all =0x0044, i_grid_all =
            i_all_short=0x000f, i_all_long=0x00ff };
```

Here, `spec` relates to information like parameters and settings, `int` to information about the integration itself like results, errors and statistics, `adapt` to information about the progression and efficiency of the adaptation cycles, and `grid` options would print out weight data (grid data for continuous dimensions). In each category the amount of information displayed can be regulated, e.g. by appending `_more` and `_all`.

In this release the `Info` argument of the `info` member function is just a dummy argument and the function outputs information in a style that I found useful. Based on feedback from other users a variety of output options will be implemented in future dVegas releases.

2.6: Vegas interface

```
class Vegas
{
public:
  void loop(int64 ncall_);  // accumulate data
  void* loop_worker() const ;  // thread function for parallel mode
  void adapt(); // adapt weights
  void reset(Reset flag); // discard data
  void info(Info flag) const ;  // print info

  void get(double x[], int k[], double* wgt_, double aux[], int icpu = 0);
  void put(const double f[], int icpu = 0);
  void pre_loop(int64 ncall_);
  void post_loop();

  Vegas(int cdim, int cnbin, int ddim, int adim, const int nvals[], int fn_, Integran
1);
  virtual ~Vegas() { }
```

```
// the following two functions are questionable from a design perspective and may n
// be available in the next major version (i.e. versions 2.0.0 and higher)
void set_parameters(int cdim, int cnbin, int ddim, int adim, const int nvals[], int
void set_integrand(Integrand fxn);

// results since last adaptation
double get_integral(int fn_) const ;
double get_error(int fn_) const ;
double get_redchi2(int fn_) const ; // "reduced chi^2" = chi^2/(nr of estimates-1)
int get_niter() const ; // nr of estimates/iterations

// accumulated results
double get_acc_integral(int fn_) const ;
double get_acc_error(int fn_) const ;
double get_acc_redchi2(int fn_) const ; // "reduced chi^2" = chi^2/(nr of estimates
int get_acc_niter() const ; // nr of estimates/iterations

// save and read weights information and configuration (possibly multiple times)
void save(const char* filename = "vegas.dat") const ;
void save(std::ofstream& fout) const ;
void restore(Integrand fxn, const char* filename = "vegas.dat");
void restore(Integrand fxn, std::ifstream& fin);
void restore(std::ifstream& fin); // integrand set with set_integrand()

friend std::ostream& operator<<(std::ostream& s, const Vegas& v);
friend std::istream& operator>>(std::istream& s,       Vegas& v);

// digital watermark with version information
void print_version() const ;

// use with caution
void set_niter(int i);
```

```
private:
  // ...
};
```

2.7: Vegas implementation

The complete C++ source code can be found in file *src/dvegas.cpp*.

- Next chapter
- Previous chapter
- Table of contents

Chapter 3: Fortran Interface

3.1: VEGAS mode (default)

3.1.1: Subroutines

Program snippet that shows how to define the integrand subroutine (named `fxn` here):

```
EXTERNAL fxn
[ main program ]

SUBROUTINE fxn(x, k, wgt, aux, f)
IMPLICIT NONE
DOUBLE PRECISION x(5), wgt, aux(1), f(1)
INTEGER k(1)
[...]
END
```

The array `x` contains the random numbers for the continuous dimensions, the array `k` contains the random numbers for the discrete dimensions, the array `aux` contains the auxilliary random numbers. The VEGAS weight (For each iteration the sum of the weights is normalized to 1.) is passed in variable `wgt`, so that it can be passed on to routines that fill histograms, for example. The subroutine `fxn` is expected to fill the array `f` with the function values that correspond to the arguments `x(1)`, `x(2)`, ..., `k(1)`, `k(2)`, ... and `aux(1)`, `aux(2)`, ... The value of the first function is assigned to `f(1)`. The integration is optimized for this function according to the VEGAS algorithm. Secondary functions can simultaneously be integrated and their values are expected to be returned as `f(2)`, `f(3)`, etc.

First this subroutine has to be called:

```
CALL dvegas_init(cdim, nsdim, cnbin, ddim, adim, nvals, fn, fxn)
```

which sets the following (internal) `INTEGER` variables:

cdim: number of continuous dimensions
nsdim: number of non-separable continuous dimensions (first nsdim dimensions)
cnbin: number of bins for each continuous dimension
ddim: number of discrete dimensions
adim: number of auxilliary dimensions (no adaptation)
nvals: array: number of values for each discrete dimension (see below)
fn: number of functions to integrate, (weights are adapted for first function)
fxn: integrand subroutine (this is a subroutine identifier, not an `INTEGER` variable)

The default integration/summation range is

- for continuous dimensions: (0, 1)
- for discrete dimensions: {0, 1, 2, ..., nvals(dim)-1}

Random numbers in (0, 1) are provided to facilitate the addition of auxilliary dimensions that are not VEGAS driven.

The parameters `ddim` or `adim` can be set to zero if no discrete or auxilliary dimensions are needed.

Optional initialization calls:

```
CALL dvegas_init_parallel(ncpu)
```

specify number of CPUs for multiprocessor machines (ncpu <= 4, see section 5.1)

```
CALL dvegas_init_output()
```

save status (init. parameters and weights) to file *dvegas.dat-new* after next `dvegas` call

```
CALL dvegas_init_files(outputfile, inputfile)
```

restore previously saved status from file *inputfile* and save new status to file *outputfile*

After dVegas has been initialized you can make multiple calls to `dvegas`:

```
CALL dvegas(ncall, iterats, init, integral, error, redchi2)
```

ncall: `INTEGER*8` (!), request `ncall` shots per iteration
iterats, init: `INTEGER`, request `iterats` iterations, `init` is explained below
integral, error, redchi2: `DOUBLE PRECISION` arrays, pass accumulated results, errors and "reduced chi^2" = chi^2/(nr of estimates-1)

The parameter `init` has the same meaning as in the original VEGAS program:

init = 0: fresh start (no prior information)
init = 1: use previous adaptation info, but discard shot data
init = 2: use previous adaptation info and build on previous shots data

3.1.2: Sample program

```
      PROGRAM TESTDVEGAS
      IMPLICIT NONE
      INTEGER nvals(1)
      INTEGER*8 ncalls
      DOUBLE PRECISION integral(1), error(1), redchi2(1)
      CHARACTER*20 outputfile, inputfile
      EXTERNAL fxn2
      nvals(1) = 100
      ncalls = 100000

      CALL dvegas_init(5, 0, 50, 1, 0, nvals, 1, fxn2)
*     2 iterations with init = 0, no initial adaptation
      CALL dvegas(ncalls, 2, 0, integral, error, redchi2)
*     result is 199.0
```

```
      WRITE(*,*) 'Result:'
      WRITE(*,*) 'integral = ', integral(1)
      WRITE(*,*) 'error    = ', error(1)
      WRITE(*,*) 'chi2/itn = ', redchi2(1)
      WRITE(*,*) ' '
*     now 1 iteration with init = 1, use previous adaptation
*     save grid
      CALL dvegas_init_output()
      CALL dvegas(ncalls, 1, 1, integral, error, redchi2)
*     now another run after restoring adaptation from file
      outputfile = 'run2.out'
      inputfile  = 'run2.in'
      CALL dvegas_init_files(outputfile, inputfile)
      CALL dvegas(ncalls, 1, 1, integral, error, redchi2)
      END

      SUBROUTINE fxn2(x, k, wgt, aux, f)
      IMPLICIT NONE
      DOUBLE PRECISION x(5), wgt, aux(1), f(1)
      INTEGER k(1)

      IF (k(1) .LT. 99) THEN
         f(1) =   1 * x(1)*x(2)*x(3)*x(4)*x(5)* 2.0**5
      ELSE IF (k(1) .EQ. 99) THEN
         f(1) = 100 * x(1)*x(2)*x(3)*x(4)*x(5)* 2.0**5
      END IF
      END
```

3.2: GetPut mode

3.2.1: Subroutines

First this subroutine has to be called:

```
      CALL dvegas_init(cdim, nsdim, cnbin, ddim, adim, nvals, fn)
```

which sets the following (internal) `INTEGER` variables:

cdim: number of continuous dimensions
nsdim: number of non-separable continuous dimensions (first nsdim dimensions)
cnbin: number of bins for each continuous dimension
ddim: number of discrete dimensions
adim: number of auxilliary dimensions (no adaptation)
nvals: array: number of values for each discrete dimension (see below)
fn: number of functions to integrate, (weights are adapted for first function)

The default integration/summation range is

- for continuous dimensions: (0, 1)
- for discrete dimensions: {0, 1, 2, ..., nvals(dim)-1}

Random numbers in (0, 1) are provided to facilitate the addition of auxilliary dimensions that are not

VEGAS driven.

The parameters `ddim` or `adim` can be set to zero if no discrete or auxilliary dimensions are needed.

To specify filenames for subsequent `dvegas_save` and `dvegas_restore` calls, use

```
CALL dvegas_init_files(outputfile, inputfile)
```

The defaults are *dvegas.dat-new* and *dvegas.dat*, respectively.

After dVegas has been initialized one can build iteration blocks as follows:

```
DO itn = 1, iterats
   [...]
   CALL dvegas_before_iteration(ncall, init)
   DO i = 1, ncall
      CALL dvegas_get(x, k, wgt, aux)
      f(1) = fxn(x, k, aux)
      CALL dvegas_put(f)
   END DO
   CALL dvegas_after_iteration()
END DO
CALL dvegas_result(integral, error, redchi2)
[or CALL dvegas_acc_result(integral, error, redchi2)]
[...]
CALL dvegas_end()
```

Subroutine argument types:

ncall: `INTEGER*8` (!)
init: `INTEGER`, explained below

x, aux, f: `DOUBLE PRECISION` arrays, random variables and function values, explained below
k: `INTEGER` array, random variables, explained below
wgt: `DOUBLE PRECISION`, VEGAS weight

integral, error, redchi2: `DOUBLE PRECISION` arrays, pass accumulated results, errors and "reduced chi^2" = chi^2/(nr of estimates-1)

The parameter `init` has the same meaning as in the original VEGAS program:

init = 0: fresh start (no prior information)
init = 1: use previous adaptation info, but discard shot data
init = 2: use previous adaptation info and build on previous shots data

The get and put subroutines retrieve the random variables and return the function values:

```
CALL dvegas_get(x, k, wgt, aux)

CALL dvegas_put(f)
```

The array `x` contains the random numbers for the continuous dimensions, the array `k` contains the random numbers for the discrete dimensions, the array `aux` contains the auxilliary random numbers. The VEGAS weight (For each iteration the sum of the weights is normalized to 1.) is passed in variable

`wgt`, so that it can be used, for example, for routines that fill histograms. The array `f` is expected to be filled with the function values that correspond to the arguments `x(1)`, `x(2)`, ..., `k(1)`, `k(2)`, ... and `aux(1)`, `aux(2)`, ... The value of the first function is assigned to `f(1)`. The integration is optimized for this function according to the VEGAS algorithm. Secondary functions can simultaneously be integrated and their values are expected to be passed as `f(2)`, `f(3)`, etc.

Persistence: The status of dVegas can be saved and restored later with these two calls:

```
      CALL dvegas_save()

      CALL dvegas_restore()
```

`dvegas_save` saves the dVegas status (init. parameters and weights) to file *dvegas.dat-new*. `dvegas_restore` restores a previously saved status from file *dvegas.dat*. Other filenames can be specified with `dvegas_init_files`, see above.

3.2.2: Sample program

```
      PROGRAM TESTDVEGAS
      IMPLICIT NONE
      INTEGER nvals(1)
      INTEGER*8 ncalls
      DOUBLE PRECISION integral(1), error(1), redchi2(1)
      CHARACTER*20 outputfile, inputfile

      INTEGER i, init, itn
      DOUBLE PRECISION fxn2
      DOUBLE PRECISION x(5), wgt, aux(1), f(1)
      INTEGER k(1)

      nvals(1) = 100
      ncalls = 100000

      CALL dvegas_init(5, 0, 50, 1, 0, nvals, 1)
*     2 iterations with init = 0, no initial adaptation
      DO itn = 1, 2
         init=1
         IF (i .EQ. 1) init=0
         CALL dvegas_before_iteration(ncalls, init)
         DO i = 1, ncalls
            CALL dvegas_get(x, k, wgt, aux)
            f(1) = fxn2(x, k)
            CALL dvegas_put(f)
         END DO
         CALL dvegas_after_iteration()
      END DO
      CALL dvegas_result(integral, error, redchi2)
*     CALL dvegas_acc_result(integral, error, redchi2)
*     result is 199.0
      WRITE(*,*) 'Result:'
```

```
      WRITE(*,*) 'integral = ', integral(1)
      WRITE(*,*) 'error    = ', error(1)
      WRITE(*,*) 'chi2/itn = ', redchi2(1)
      WRITE(*,*) ' '
*     now 1 iteration with init = 1, use previous adaptation
*     save grid
      CALL dvegas_before_iteration(ncalls, 1)
      DO i = 1, ncalls
         CALL dvegas_get(x, k, wgt, aux)
         f(1) = fxn2(x, k)
         CALL dvegas_put(f)
      END DO
      CALL dvegas_after_iteration()
      CALL dvegas_save()
*     now another run after restoring adaptation from file
      outputfile = 'run2.out'
      inputfile  = 'run2.in'
      CALL dvegas_init_files(outputfile, inputfile)
      CALL dvegas_restore()
      CALL dvegas_before_iteration(ncalls, 1)
      DO i = 1, ncalls
         CALL dvegas_get(x, k, wgt, aux)
         f(1) = fxn2(x, k)
         CALL dvegas_put(f)
      END DO
      CALL dvegas_after_iteration()
      CALL dvegas_save()
      CALL dvegas_end()
      END

      DOUBLE PRECISION FUNCTION fxn2(x, k)
      IMPLICIT NONE
      DOUBLE PRECISION x(5)
      INTEGER k(1)

      IF (k(1) .LT. 99) THEN
         fxn2 =   1 * x(1)*x(2)*x(3)*x(4)*x(5)* 2.0**5
      ELSE IF (k(1) .EQ. 99) THEN
         fxn2 = 100 * x(1)*x(2)*x(3)*x(4)*x(5)* 2.0**5
      END IF
      END
```

- Next chapter
- Previous chapter
- Table of contents

Chapter 4: C Interface

4.1: Functions

The integrand function has to be defined with the following arguments:

```
typedef void (*C_Integrand)(double x[], int k[], double* wgt,
                            double aux[], double f[]);

void fxn(double x[], int k[], double* wgt, double aux[], double f[]);
```

The array `x[]` contains the random numbers for the continuous dimensions, the array `k[]` contains the random numbers for the discrete dimensions, the array `aux[]` contains the auxilliary random numbers. The VEGAS weight (For each iteration the sum of the weights is normalized to 1.) is passed in variable `wgt`, so that it can be passed on to functions that fill histograms, for example. The function `*fxn` is expected to fill the array `f[]` with the function values that correspond to the arguments `x[0]`, `x[1]`, ..., `k[0]`, `k[1]`, ... and `aux[0]`, `aux[1]`, ... The value of the first function is assigned to `f[0]`. The integration is optimized for this function according to the VEGAS algorithm. Secondary functions can simultaneously be integrated and their values are expected to be returned as `f[1]`, `f[2]`, etc.

First this function has to be called:

```
void dvegas_init_(int* cdim, int* nsdim, int* cnbin, int* ddim, int* adim,
                  int nvals[], int* fn, C_Integrand fxn);
```

which sets the following (internal) variables:

cdim: number of continuous dimensions
nsdim: number of non-separable continuous dimensions (first nsdim dimensions)
cnbin: number of bins for each continuous dimension
ddim: number of discrete dimensions
adim: number of auxilliary dimensions (no adaptation)
nvals[]: number of values for each discrete dimension (see below)
fn: number of functions to integrate
fxn: pointer to integrand function

The default integration/summation range is

- for continuous dimensions: (0, 1)
- for discrete dimensions: {0, 1, 2, ..., nvals[dim]-1}

Random numbers in (0, 1) are provided to facilitate the addition of auxilliary dimensions that are not VEGAS driven.

The parameters `ddim` or `adim` can be set to zero if no discrete or auxilliary dimensions are needed.

Optional initialization calls:

```
void dvegas_init_parallel_(int* ncpu);
```
specify number of CPUs for multiprocessor machines (see section 5.1)

```
void dvegas_init_output_();
```
save status (init. parameters and weights) to file *dvegas.dat-new* after next `dvegas_` call

```
void dvegas_init_files_(char* outfile_, char* infile_, int outfile_len,
                        int infile_len);
```
restore previously saved status from file *inputfile* and save new status to file *outputfile*
Comment: The arguments `outfile_len` and `infile_len` are not really necessary in C, because C strings are NULL terminated. Fortran strings are unfortunately not terminated. Thus, in a C program it is sufficient that the values passed for `outfile_len` and `infile_len` are equal or **larger** than the actual string sizes. The exact values are irrelevant.

After dVegas has been initialized you can make multiple calls to `dvegas_`:

```
void dvegas_(longint* ncall, int* iterats, int* init, double integral[],
             double error[], double redchi2[]);
```
ncall: type longint, request `ncall` shots per iteration
iterats, init: request `iterats` iterations, `init` is explained below
integral[], error[], redchi2[]: accumulated results, errors and "reduced chi^2" = chi^2/(nr of estimates-1)

The parameter `init` has the same meaning as in the original VEGAS program:

init = 0: fresh start (no prior information)
init = 1: use previous adaptation info, but discard shot data
init = 2: use previous adaptation info and build on previous shots data

4.2: C header

The C header *include/c_dvegas.h* needs to be included in each program that uses dVegas. The C interface is implemented in *src/c_dvegas.cpp*. Note that c_dvegas does not stand for "C language dVegas", but rather for "C linkage dVegas", and is also the basis of the Fortran interface.

4.3: Sample program

```
#include <math.h>
#include <stdio.h>
#include "c_dvegas.h"

void fxn2(double x[], int k[], double* wgt, double aux[], double f[]);

int main()
{
  int cdim  =  5 ;
  int nsdim =  0 ;
  int cnbin = 50 ;
  int ddim  =  1 ;
  int adim  =  0 ;
  int nvals[1] = { 100 };
```

```
  int fn    =  1 ;

  longint ncalls = 100000 ;
  int iterats ;
  int init ;
  double integral[1];
  double error[1];
  double redchi2[1];
  char inputfile[21] = "run2.in" ;
  char outputfile[21] = "run2.out" ;

  dvegas_init_(&cdim, &nsdim, &cnbin, &ddim, &adim, nvals, &fn, fxn2);
  /* 2 iterations with init = 0, no initial adaptation */
  iterats = 2 ; init = 0 ;
  dvegas_(&ncalls, &iterats, &init, integral, error, redchi2);
  /* result is 199.0 */
  printf("Result:\n");
  printf("integral = %f\n", integral[0]);
  printf("error    = %f\n", error[0]);
  printf("chi2/itn = %f\n\n", redchi2[0]);
  fflush(NULL);
  /* now 1 iteration with init = 1, use previous adaptation */
  iterats = 1 ; init = 1 ;
  /* save grid                                              */
  dvegas_init_output_();
  dvegas_(&ncalls, &iterats, &init, integral, error, redchi2);
  /* now another run after restoring adaptation from file */
  dvegas_init_files_(outputfile, inputfile, 20, 20);
  dvegas_(&ncalls, &iterats, &init, integral, error, redchi2);
}

void fxn2(double x[], int k[], double* wgt, double aux[], double f[])
{
  if (k[0] < 99)
    f[0] =   1.0 * x[0]*x[1]*x[2]*x[3]*x[4]*pow(2.0,5.0);
  else if (k[0] == 99)
    f[0] = 100.0 * x[0]*x[1]*x[2]*x[3]*x[4]*pow(2.0,5.0);
}
```

- Next chapter
- Previous chapter
- Table of contents

Chapter 5: Parallel dVegas

5.1: Shared memory multiprocessor machines (SMP)

If a **Posix threads** implementation exists for your platform you can link your programs with the parallel dVegas libraries *libdvegas_r.a*, *libf_dvegas_r.a*, *libc_dvegas_r.a* (for C++, Fortran and C, respectively) to take full advantage of multiprocessor machines. "_r" stands for reentrant. Note that these libraries are not reentrant themselves, rather the "_r" suffix is introduced to remind you that when using the parallel libraries the integrand function/subroutine has to be **reentrant**, aka thread-safe and MT-safe. Reentrant essentially means that several copies of a function can be executed simultaneously without interfering with each other. If this is not the case the program will crash or yield arbitrary results.

C++ and C functions that do not write to global variables or use local variables declared "static" are typically reentrant. Fortran subroutines can present more problems. Some advice:

Variables that appear in a `SAVE` statement or are initialized in a `DATA` statement are always allocated statically and will corrupt the threaded version. Make sure no values are assigned to `COMMON` block variables in the subroutines called by `fxn`.

If you call Fortran subroutines in `fxn` make sure that all local variables are put on the run-time stack by using the corresponding compiler

option, typically `-automatic`. The default behavior is very likely static allocation. (The GNU Fortran compiler is an exception to the rule, with automatic allocation being the default. It has an option `-fno-automatic`.)

Note: Static variables (default, -static) are always initialized to default (or `DATA` statement) values before the subroutine is executed. Variables put on the run-time stack (`-automatic`) are NOT initialized automatically before the subroutine is executed and thus don't have a well-defined value until they get a value assigned when an assignment statement is executed. Using uninitialized variables in expressions (e.g. on the right side of an assignment statement) triggers floating exception signals in threads.

The following sections contain platform specific notes about writing code and creating excetables with the parallel dVegas libraries. Please report any problems (and solutions) you find when using these libraries, so that we can address them. See section 1.6 for details.

For documented platforms you might want to run `make test` and look at the test programs for the parallel libraries and check out how they are compiled.

5.1.1: Intel/Linux

No special issues known. Just link with `-lpthread`.

5.1.2: Alpha/Tru64 Unix

Use `-pthread` option for all Compaq compilers when compiling or linking. Use `-automatic` when compiling or linking with the Compaq Fortran compiler.

5.2: Workstation clusters

A dVegas library for distributed computing is not yet available. However, adapting the SMP code for workstation clusters is straightforward. The standard choice in scientific computing would be the Message Passing Interface (MPI). However, I believe that a CORBA-based solution would be easier to install and use, as well as more flexible and transparent. In spring 2002, I plan to integrate dVegas into a truly component-oriented, multi-process, CORBA-based architecture for Monte Carlo computations using the free, high-performance implementation from AT&T Laboratories Cambridge (omniORB). See omnicomp at hepsource.org for a preview. If you are familiar with MPI and would like to contribute code for MPI, please contact me.

- Next chapter
- Previous chapter
- Table of contents

Chapter 6: Python Class and Interface

Python (www.python.org) is an interpreted, interactive, object-oriented programming language. It is easy to learn, yet expressive, has a clear, elegant syntax, intuitive error checking facilities and does not require a compilation step. A Python module with a class implementing the dVegas algorithm and interface is located in subdirectory *python/*. The module uses Numerical Python for efficient array handling. The NumPy package can be obtained from http://numpy.sourceforge.net. An example that demonstrates the use of *dVegas.py* is included in file *demo.py*.

The directory *python/cpp/* contains a Python module *dVegas.py* that wraps around the C++ implementation. It can be used in cases where the performance of the Python implementation is insufficient. To that end the shared library *_dvegasmodule.so* has to be created using commands similar to

```
unix> g++ -c -DSIZEOF_LONG_LONG=8 -I/usr/local/python/include/python1.5
      -I/usr/local/python/include/python1.5/numeric -I../../include -fPIC _dvega
unix> g++ -shared -o _dvegasmodule.so _dvegasmodule.o ../../src/dvegas.o -lCLHEP
```

Chapter 7: Frequently Asked Questions

1. **How do I obtain version information for a dVegas library?** The command `strings libdvegas.a | grep "dVegas version"` displays the version on most Unix systems.

- Previous chapter
- Table of contents

Chapter 8: References

(with hyperlinks to compressed postscript in HTML version)

1. N. Kauer, in preparation.
2. A. Duff, private communication.
3. R. Kreckel, Comp. Phys. Comm. 106 (1997) 258. PS
   R. Kreckel, Mainz preprint (1998) MZ-TH/98-54. PS
4. W.H. Press et al., Numerical Recipes in C, 2nd ed., Cambridge University Press (1992) 319. PS
5. G.P. Lepage, J. Comp. Phys. 27 (1978) 192. PS
   G.P. Lepage, Cornell preprint (1980) CLNS-80/447. PS
6. S. Veseli, Comp. Phys. Comm. 108 (1998) 9. PS
7. T. Ohl, Comp. Phys. Comm. 120 (1999) 13. PS

- Previous chapter
- Table of contents