Dvegas User's Guide

Nikolas Kauer <nkauer@users.sourceforge.net>

Version 2.0.0, April 2002

Table of Contents

- Next chapter
- Table of contents

Chapter 1: Essentials

1.1: About Dvegas

Dvegas facilitates adaptive Monte Carlo integration based on an enhanced and extended version of Peter Lepage's VEGAS algorithm. It allows to automatically take into account correlations between sets of dimensions, and allows to fully adapt the sampling of sums of integrals. The code is genuinely object-oriented, written in ISO/ANSI standard C++, makes extensive use of the C++ Standard Library and includes an interface to OmniComp.

OmniComp is an intuitive system that is easy to install and use which allows to accelerate Monte Carlo programs through distributed execution on workstation clusters or PC farms as well as multi- processor machines. It is based on omniORB, a high-performance, open-source CORBA implementation from AT&T Laboratories Cambridge. Notably, it does not require thread-safe integrand implementations.

The *D* in Dvegas stands for *decoded* and *deconstructed*, because the somewhat cryptic "black box" code of the original VEGAS program and its derivatives was decoded and the algorithm recoded from scratch, in a modern, highly modular, const-correct object-oriented way, aggressively applying modern software development fundamentals and best practices as layed out in Bjarne Stroustrup's "The C++ Programming Language", Andy Hunt and Dave Thomas's "The Pragmatic Programmer", Scott Meyers's "Effective C++" and Martin Fowler's "Refactoring". The resulting code is well-structured, accessible and robust, thus being well suited for easy and safe modification and extension.

A detailed description of the VEGAS algorithm, its enhanced and extended implementation in Dvegas, as well as comparisons with and information about several earlier implementations, can be found in the references listed in section 4.

1.2: Platform notes

In principle, Dvegas can be built with any sufficiently standard compliant C++ compiler on any platform. In practice, Dvegas has been sucessfully employed in the following environments:

1.2.1: GNU C++ compiler

Programs based on Dvegas will compile and link with recent releases (3.x) of the C++ compiler of the GNU compiler collection, which can be obtained at http://gcc.gnu.org.

1.2.1.1: Linux/i386

gcc-3.0 on Red Hat Linux 7 with Intel Pentium III, Intel Xeon or AMD Athlon is known to work.

1.2.1.2: Tru64 Unix/Alpha

gcc-3.0 on Compaq Tru64 Unix 5 with DEC Alpha EV6.7 is known to work.

1.2.1.3: Windows2000/i386

gcc-3.0 on Windows2000 with Pentium III is known to work. See README.win32 for special instructions.

1.2.2: Compaq C++ compiler

The Compaq C++ compiler **cxx** is available for Alpha platforms with either Tru64 Unix or Linux.
http://www.compaq.com/products/software/compilers/candcxx.html

1.2.2.1: Tru64 Unix/Alpha

Compaq C++ V6.3 on Compaq Tru64 Unix 5 with DEC Alpha EV6.7 is known to work.

If you are working on Linux/Alpha with either **g++** or **cxx** and find Dvegas to work please drop me a note.

For up-to-date information about supported platforms visit
http://hepsource.org/dvegas/platforms.html

1.3: How to get Dvegas

The Dvegas source code can be downloaded as .tar.gz archives or via CVS here: http://sourceforge.net/projects/hepsource/

The .tar.gz archives can, for example, be unpacked with
```
gunzip -c dvegas-2.0.0.tar.gz | tar xf -
```

1.4: Using Dvegas

How to use Dvegas in your programs and its capabilities are illustrated with the help of two demo programs (also see section 2). Copy the sample Makefile for your compiler to `Makefile`, possibly edit it, and run **make**. If more than one compiler is installed on your system, make sure the correct compiler is used, i.e. comes first in your path. The build process will attempt to create the following demo programs: **dvegas_demo** and **omnicomp_demo**

Note that in order to build **omnicomp_demo**, omniORB headers and libraries need to be available on your system and the corresponding `Makefile` variables set appropriately. Please refer to section 3 for more information.

1.5: Random Number Generation

To allow easy substitution of the random number generator employed by Dvegas, the `RandomNumberGenerator` interface defined in `rng.h` is used. Three implementations are provided:

1.5.1: Built-in: ranlux.cpp

For convenience, the Dvegas distribution includes code in `ranlux.cpp` that provides a default random number generator based on Martin Luscher's chaos theory-improved RANLUX algorithm, which has excellent characteristics. To vary the seed go to the top of the file and modify the setting of `seedForGSLRandomNumberGenerator`.

1.5.2: Class Library for High Energy Physics (CLHEP)

The Random package of the CLHEP library provides C++ implementations for a variety of quality random number generators. If CLHEP headers and library are

available on your system, you can use any CLHEP random number generator by setting the corresponding `Makefile` variables appropriately, and linking with `clhep_rng.o` instead of `ranlux.o`. The sample Makefiles include an example. The default CLHEP random number generator is RANLUX. To select a different generator, simply replace `RanluxEngine` in the following line in `clhep_rng.cpp`:

```
RandomNumberGenerator *const randomNumberGenerator = new
CLHEPRandomNumberGenerator(new RanluxEngine(), 19780503,
3);
```

1.5.3: GNU Scientific Library (GSL)

The Random Number Generation module of the GSL library provides C implementations for a variety of quality random number generators. If GSL headers and library are available on your system, you can use any GSL random number generator by setting the corresponding `Makefile` variables appropriately, and linking with `gsl_rng.o` instead of `ranlux.o`. The sample Makefiles include an example. The default GSL random number generator is RANLUX. To select a different generator, simply replace `gsl_rng_ranlxs1` in the following line in `gsl_rng.cpp`:

```
RandomNumberGenerator *const randomNumberGenerator = new
GSLRandomNumberGenerator(gsl_rng_alloc(gsl_rng_ranlxs1),
19780503);
```

1.6: Troubleshooting and Support

For up-to-date information about Dvegas-related problems and solutions visit http://hepsource.org/dvegas/troubleshooting.html

Problems not addressed there should be sent to *hepsource-support@lists.sourceforge.net*. Comments and suggestions should be sent to *hepsource-devel@lists.sourceforge.net*.

1.7: Porting Dvegas

If you want to use Dvegas on a new platform or with a new compiler, please contact *hepsource-devel@lists.sourceforge.net*.

1.8: LGPL license

This library is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version.

This library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with this library; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

- Next chapter
- Table of contents

Chapter 2: Dvegas

2.1: Dvegas interface

```
namespace HepSource
{
class Dvegas
{
public:
  Dvegas(const int aDim, const int f, Integrand *const i);
  Dvegas(const int cDim, const int cBin, const int f, Integrand *const i,
         const Sampling sampling = IMPORTANCE);
  Dvegas(const int cDim, const int cBin, const vector<int>& dDimSizes, const int f,
         Integrand *const i, const Sampling sampling = IMPORTANCE);
  Dvegas(const int cDim, const int cBin, const int corrDim,
         const vector<int>& dDimSizes, const int aDim, const int f, Integrand *const
         const Sampling sampling = IMPORTANCE);
  Dvegas(const int cDim, const int cBin, const vector<int>& corrDim,
         const vector<int>& dDimSizes, const int aDim, const int f, Integrand *const
         const Sampling sampling = IMPORTANCE);
  Dvegas();
  virtual ~Dvegas();
```

```
CellAccumulators collectData(const Int64 numberOfShots);

void info() const;     // print info about best estimate
const bool isAdaptingContinuousDimensions() const;
const double chiSquarePerIteration() const;

void adapt(CellAccumulators& cellAcc);     // adapt weights
const double getCurvature() const;
void setCurvature(const double curvature);
const double getRootAmpl() const;
void setRootAmpl(const double rootAmpl);

void disableOptimize(const int setOfCorrelatedWeights);

void reset();          // discard all data
void resetWeights();
void resetEstimates();

void saveStateToFile(const string& filename = "dvegas.state") const;
void restoreStateFromFile(Integrand *const i, const string& filename = "dvegas.stat
void restoreWeightsFromFile(Integrand *const i, const string& filename = "dvegas.st
void saveStateToStream(ostream& os) const;
void restoreStateFromStream(Integrand *const i, istream& is);
void restoreWeightsFromStream(Integrand *const i, istream& is);
const string saveStateToString() const;
void restoreStateFromString(Integrand *const i, const string& s);
void restoreWeightsFromString(Integrand *const i, const string& s);

void activateCorrelationsDetector();

void attachOmniComp(OmniComp *const omniComp);
```

```
  void detachOmniComp();

  void printVersion() const;
};
}
```

2.2: Dvegas constructors

2.2.1: Constructor arguments

cDim: int

number of adapted continuous dimensions

cBin: int

resolution of adaptation (number of bins)

corrDim: vector<int>

defines sets of continuous dimensions for which the adaptation fully takes correlations into account
Example: cDim = 8 and corrDim = {3, 5}: set1 = {0, 1, 2}, set2 = {3, 4}, correlations not sampled for 5, 6, 7

dDimSizes: vector<int>

defines discrete dimensions
Example: dDimSizes = {3, 5}: two discrete dimensions with index1 in {0, 1, 2} and index2 in {0, 1, 2, 3, 4}

aDim: int

number of "auxilliary" (i.e. non- adapted) continuous dimensions

f: int

number of integrands to be evaluated (first integrand drives adapation, see below)

i: Integrand*

pointer to function that evaluates the integrand(s)

sampling: enum Sampling {NONE, IMPORTANCE, STRATIFIED}

Importance sampling concentrates sampled points in hypercubes that contribute most to the integral.
Stratified sampling concentrates sampled points in hypercubes that contribute most to the error of the integral.

The integration range for adapted and auxilliary continuous dimensions is `(0, 1)`.

Note that, except for `i`, all arguments can be zero (or empty `vector`), thus disabling the particular functionality.

2.2.2: Function that evaluates `f` integrands

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

x: double array (in)

contains `cDim` random floats in `(0, 1)` and thus covers the integration volume for the adapted continuous dimensions

k: int array (in)

contains `dDimSizes.size()` random integers in `[0, dDimSizes[j])` with `j = 0, ..., dDimSizes.size() - 1`
covering all possible index combinations for the discrete dimensions

weight: double (in)

weight of the sampled point, which may, for example, be used to fill histograms
Note that by default the weight is normalized in each iteration to yield the average integrand value--corresponding to the value of the integral (assuming unit volume)--rather than the sum.

aux: double array (in)

contains `aDim` random numbers in `(0, 1)` that have no effect on the adaptation

f: double array (out)

passes back `f` floats obtained by evaluating each integrand at the point specified by the input parameters
Note that the first integrand (i.e. values passed back through `f[0]`) drives the adaptation.
Depending on how similar the other integrands are to the first the adapted sampling may or may not be well suited for them.

2.2.3: Constructors

```
Dvegas(const int aDim, const int f, Integrand *const i);
```

"Crude", unadapted Monte Carlo sampling.

```
Dvegas(const int cDim, const int cBin, const int f, Integrand *const i,
       const Sampling sampling = IMPORTANCE);
```

Classic VEGAS case: the sampling is adapted for a number of continuous dimensions that are assumed to be independent.

```
Dvegas(const int cDim, const int cBin, const vector<int>& dDimSizes, const int f,
       Integrand *const i, const Sampling sampling = IMPORTANCE);
```

Full adaptation for sums of integrals.

```
Dvegas(const int cDim, const int cBin, const int corrDim,
       const vector<int>& dDimSizes, const int aDim, const int f, Integrand *const i,
       const Sampling sampling = IMPORTANCE);
```

Exclude "auxilliary" dimensions from adaptation.

```
Dvegas(const int cDim, const int cBin, const vector<int>& corrDim,
       const vector<int>& dDimSizes, const int aDim, const int f, Integrand *const i,
       const Sampling sampling = IMPORTANCE);
```

Relax independence assumption for continuous dimensions and fully sample correlations for selected dimensions.

```
Dvegas();
```

Use default constructor to subsequently initialize with `restoreStateFromStream()` or a similar method.

2.3: 64-bit numeric types

The following type aliases are defined:

```
#if (defined LONG_IS_INT64) && (!defined LONG_LONG_IS_INT64)
typedef long int Int64;
#elif (defined LONG_LONG_IS_INT64) && (!defined LONG_IS_INT64)
typedef long long int Int64;
#endif
typedef double Float64;
```

The type alias `Int64` corresponds to 64-bit integers, which allow for a maximum of 2^63 - 1 = 9,223,372,036,854,755,807 sampled points. Note that 32-bit integers only allow up to about 2 billion sampled points (2^31 - 1 = 2,147,483,647 to be precise), which in some cases may not suffice to achieve the desired precision. On 32-bit processor hardware including most Intel and AMD processors (i386 architecture), the macro `LONG_LONG_IS_INT64` should be defined. On 64-bit processor hardware, on the other hand, like DEC's Alpha, Intel's Itanium or AMD's Sledgehammer processor, the macro `LONG_IS_INT64` should be defined (see `Makefile`).

The type alias `Float64` corresponds to 64-bit floating point numbers and is used internally for variables that accumulate a large number of temporary results (of `O(numberOfShots)`). 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 platforms the type `double` corresponds to 64-bit floating point numbers. In rare cases `Float64` might correspond to `long double`. In this case the typedef should be adjusted accordingly. (Note that this issue is also relevant for histogram packages.)

2.4: Dvegas methods

```
CellAccumulators collectData(const Int64 numberOfShots);
void adapt(CellAccumulators& cellAcc);
```

Method `collectData()` samples `numberOfShots` points in the integration volume, i.e. advances the Monte Carlo

integration by one iteration and returns data accumulated for each cell in an object of type `CellAccumulators`. This object is subsequently passed to method `adapt()`, which adapts all weights, and thus improves the sampling in the next iteration.

```
void info() const;
```

At any time method `info()` can be used to print out information regarding the currently best estimate for the integral(s).

```
const double getCurvature() const;
void setCurvature(const double curvature);
const double getRootAmpl() const;
void setRootAmpl(const double rootAmpl);
```

The parameters `curvature` and `rootAmpl` ("root amplification") control the promotion of smaller weights, which damps the adaptation in order to avoid oscillations. See implementation of helper function `promoteSmallerWeights()` in `dvegas.cpp` for details.

```
void disableOptimize(const int setOfCorrelatedWeights);
```

If correlations are sampled, an optimization is performed before the sampling occurs. In rare cases the optimization step requires more time than is subsequently saved. In such cases the optimization can be turned off with the `disableOptimize()` method.

```
void reset();
void resetWeights();
void resetEstimates();
```

`reset()` discards all state-related internal data, reverting the object back to the state immediately after construction.

```
void saveStateToFile(const string& filename = "dvegas.state") const;
void restoreStateFromFile(Integrand *const i, const string& filename = "dvegas.state"
void restoreWeightsFromFile(Integrand *const i, const string& filename = "dvegas.stat
void saveStateToStream(ostream& os) const;
void restoreStateFromStream(Integrand *const i, istream& is);
```

```
void restoreWeightsFromStream(Integrand *const i, istream& is);
const string saveStateToString() const;
void restoreStateFromString(Integrand *const i, const string& s);
void restoreWeightsFromString(Integrand *const i, const string& s);
```

Methods are provided to save `Dvegas` objects to a stream, file or string and subsequently restore the saved state (or the most recent weights without the integral estimates of previous iterations).

```
void activateCorrelationsDetector();
```

By calling `activateCorrelationsDetector()` a separate module is activated that, during the following iteration, probes (for the first integrand) to what degree combinations of two dimensions are independent (as assumed by the VEGAS algorithm). If strong correlations are detected, they can then be fully sampled by setting `corrDim` accordingly. Circular correlations should be properly combined, e.g. the correlations {0, 1} and {1, 2} translate to the set {0, 1, 2}.

```
void attachOmniComp(OmniComp *const omniComp);
void detachOmniComp();
```

Before distributed integrand evaluation can be used to accelerate computations, an `OmniComp` object needs to be attached to the `Dvegas` object with `attachOmniComp` (see section 3 for more on `OmniComp`).

```
const bool isAdaptingContinuousDimensions() const;
const double chiSquarePerIteration() const;
```

These getter methods are used in `VEGAS()` for diagnostic warnings (see section 2.5).

2.5: VEGAS function

```
namespace HepSource
{
void VEGAS(Dvegas& dvegas, const Int64 numberOfShots, const int numberOfIterations,
           const int init = 0);
```

```
}
```

This function mimics the interface of the original VEGAS program. The parameter `init` has the following meaning:

init = 0

fresh start--discard all weight and estimate information

init = 1

use current weights, but discard prior estimates

init = 2

use current weights and take into account estimates of previous iterations

2.6: Examples

A comprehensive set of examples can be found in the file `dvegas_demo.cpp`.

- Next chapter
- Previous chapter
- Table of contents

Chapter 3: OmniComp

OmniComp is an intuitive system that is easy to install and use which allows to accelerate Monte Carlo programs through distributed execution on workstation clusters or PC farms as well as multi- processor machines. It is based on omniORB, a high-performance, open-source CORBA implementation from AT&T Laboratories Cambridge. Notably, it does not require thread-safe integrand implementations.

3.1: Using OmniComp

Usage examples:

```
host3> ssh -n host1 /path1/omnicomp_prog -w &
host3> ssh -n host2 /path2/omnicomp_prog -w &
host3> ssh -n host2 /path2/omnicomp_prog -w &
host3> /path3/omnicomp_prog
master process started (includes 1 worker)
3 additional worker(s) found
work done so far ...
0%  0%  0%  0%
```

Here, host1 and the local host are assumed to be single-processor machines, while host2 is assumed to be a dual-processor system. The working directory with the executable file **omnicomp_prog** is assumed to be shared between host1, host2 and the local host. This is convenient but not necessary (see **-n** option below). If **ssh** asks for a password on the command line try **ssh -f** instead of **ssh -n**.

Since the master process contains its own worker one can also start just one instance of the executable with no options and it will run like a regular, non-distributed program:

```
host> /path/omnicomp_prog
master process started (includes 1 worker)
no additional workers found
work done so far ...
0%
```

If different executables are required on different hosts, for example because they run different operating systems, the executables can be distinguished with a ".<key>" postfix. One needs to use the **-d** option in this case. It causes the postfix to be disregarded when the `*.workers` file

name is determined:

```
host3> ssh -n host1 /path1/omnicomp_prog.os1 -w -d &
host3> ssh -n host2 /path2/omnicomp_prog.os2 -w -d &
host3> ssh -n host2 /path2/omnicomp_prog.os2 -w -d &
host3> /path3/omnicomp_prog.os3 -d
```

One can disable the collocated worker in the local master process with the **-m** option. In this mode the master process controls the other workers, monitors progress and collects results, but does not participate in the computation itself.

If no shared directory mounted on all machines is available the distributed computation can be bootstrapped using the naming service (**-n** option). After setting up the environment for naming service as described in section 3.2.2, the server needs to be started on the host specified in `omniORB.cfg` by executing **omniNames**. Then, for example for tcsh:

```
host4> ssh -n host1 "tcsh -c '/path1/omnicomp_prog -w -n'" &
host4> ssh -n host2 "tcsh -c /path2/omnicomp_prog -w -n'" &
host4> ssh -n host3 "tcsh -c /path3/omnicomp_prog -w -n'" &
host4> /path4/omnicomp_prog -n
```

If your remote shell account is not set up to access the naming service you can also include the information on the command line:

```
host4> ssh -n host1 /path1/omnicomp_prog -w -n
       -ORBInitRef NameService=corbaname::names.example.edu &
host4> ssh -n host2 /path2/omnicomp_prog -w -n
       -ORBInitRef NameService=corbaname::names.example.edu &
host4> ssh -n host3 /path3/omnicomp_prog -w -n
       -ORBInitRef NameService=corbaname::names.example.edu &
host4> /path4/omnicomp_prog -n
```

Here, `names.example.edu` is the hostname of the system that provides the naming service.

It is important to use the same set of omnicomp options (except for **-w** and **-m**) in all commands else errors will likely occur. If that happens the naming service can be cleaned up with **nameclt**, an omniORB client program to inspect and modify the naming service registry.

3.2: Getting started with OmniComp

3.2.1: omniORB headers and libraries

In order to build OmniComp executables one needs to link with omniORB libraries. These have been pre-built for a number of common platforms including Intel/Linux and can be downloaded for free at
http://www.uk.research.att.com/omniORB/omniORBForm.html

If no pre-built libraries are available for your platform they can easily be built from source with a

few steps:

1. Find the configuration that best matches your platform (and compiler!) in `./mk/platforms/` and uncomment the corresponding line in `./config/config.mk`, for example: `platform = alpha_osf1_5.0`
2. In the selected configuration file in `./mk/platforms/` edit the line that sets `PYTHON` and insert the path to your python interpreter, e.g. `/usr/local/bin/python`. If Python 1.5.2 or higher is not installed on your system follow the instructions in the file.
3. change to `./src` and **make export** (ignore the warnings) This step requires about 90MB and takes about 40 minutes on a PentiumII/333MHz.
4. **make clean**

You're done. The built libraries and binaries consume about 20MB disk space.

3.2.2: Environment variables

Example commands for shell initialization (assuming **tcsh** and Linux):

```
# omniorb libraries and executables
setenv OMNIORB_TOPDIR ${HOME}/omniorb/omni
setenv LD_LIBRARY_PATH
${LD_LIBRARY_PATH}:${OMNIORB_TOPDIR}/lib/i586_linux_2.0_glibc2.1
setenv PATH ${PATH}:${OMNIORB_TOPDIR}/bin/i586_linux_2.0_glibc2.1
```

If your omnicomp programs will be located in a shared directory that is mounted on all computers no further steps are necessary.

Otherwise the computation has to be bootstrapped with omniNames (option **-n**) and one also needs:

```
# omniorb naming service (omniNames)
setenv OMNINAMES_LOGDIR ${HOME}/omniorb/names_log
setenv OMNIORB_CONFIG ${HOME}/omniorb/omniORB.cfg
```

`OMNINAMES_LOGDIR` specifies the log directory for **omniNames** and is only required in the shell that is used to start **omniNames**. The log directory and files are created when **omniNames** is started for the first time.

File `omniORB.cfg` indicates on which computer **omniNames** will be run. Example with default port number:

```
ORBInitialHost names.example.edu
ORBInitialPort 2809
```

- Next chapter
- Previous chapter
- Table of contents

Chapter 4: References

(with hyperlinks to compressed postscript in HTML version)

1. N. Kauer, in preparation.
2. B. Stroustrup, The C++ Programming Language, 3rd ed., Addison-Wesley, 1997. Amazon
3. A. Hunt and D. Thomas, The Pragmatic Programmer, Addison-Wesley, 1999. Amazon
4. S. Meyers, Effective C++, 2nd ed., Addison-Wesley, 1997. Amazon
5. M. Fowler, Refactoring, Addison-Wesley, 1999. Amazon
6. M. Luscher, Comp. Phys. Comm. 79 (1994) 100. PS
7. G.P. Lepage, J. Comp. Phys. 27 (1978) 192. PS
   G.P. Lepage, Cornell preprint (1980) CLNS-80/447. PS
8. A. Duff, private communication.
9. R. Kreckel, Comp. Phys. Comm. 106 (1997) 258. PS
   R. Kreckel, Mainz preprint (1998) MZ-TH/98-54. PS
10. W.H. Press et al., Numerical Recipes in C, 2nd ed., Cambridge University Press (1992) 319. PS
11. S. Veseli, Comp. Phys. Comm. 108 (1998) 9. PS
12. T. Ohl, Comp. Phys. Comm. 120 (1999) 13. PS