gTybalt 1.3.6

a free computer algebra system
26 October 2020

Stefan Weinzierl and Roberta Marani

Table of Contents

gTybalt

This is a tutorial that documents gTybalt 1.3.6, a free computer algebra system.

1 Introduction

Symbolic calculations, carried out by computer algebra systems, have become an integral part in the daily work of scientists. The advance in algorithms and computer technology has led to remarkable progress in several areas of natural sciences. gTybalt was developed as a tool for certain kind of calculations. The characteristics of these calculations are: First of all, these tend to be "long" calculations, e.g. the system needs to process large amounts of data and efficiency in performance is a priority. Secondly, the algorithms for the solution of the problem are usually developed and implemented by the scientists themselves. This requires support from the computer algebra system for a programming language which allows to implement complex algorithms for abstract mathematical entities. In other words, it requires support of object oriented programming techniques from the system. On the other hand, these calculations usually do not require that the computer algebra system provides sophisticated tools for all branches of mathematics. Thirdly, despite the fact that these calculations process large amounts of data, the time needed for the implementation of the algorithms usually outweights the actual running time of the program. Therefore convenient development tools are also important.

1.1 Main features of gTybalt

gTybalt is a free computer algebra system. It is distributed under the terms and conditions of the GNU General Public Licence.

The main features of gTybalt are:

- Object Oriented: gTybalt allows symbolic calculations within the C++ programming language.
- Efficiency for large scale problems: Solutions developed with gTybalt can be compiled with a C++ compiler and executed independently of gTybalt. This is particular important for computer-extensive problems and a major weakness of commercial computer algebra systems.
- Short development cycle: gTybalt can interpret C++ and execute C++ scripts. Solutions can be developed quickly for small-scale problems, either interactively or through scripts, and once debugged, the solutions can be compiled and scaled up to large-scale problems.
- High quality output: Mathematical formulae are visualized using TeX fonts and can easily be converted to LaTeX on a what-you-see-is-what-you-get basis.

Compared to other computer algebra systems, gTybalt does not try to cover every domain of mathematics. Some desirable algorithms, like symbolic integration are not implemented. However, the modular design of gTybalt allows to incorporate easily new algorithms.

The functionality of a computer algebra system can be divided into different modules, e.g. there will be a module, which displays the output, a second module analysis and interprets the input, a third module does the actual symbolic calculation. Writing a computer algebra system from scratch is a formidable task. Fortunately it is not required, since there are already freely available packages for specific tasks. gTybalt is based on several other packages and provides the necessary communication mechanisms among these packages. gTybalt is therefore a prototype of a "bazaar"-style program and an example of what is possible within the free software community. It should be clearly stated, that without these already existing packages gTybalt would never have been developed and my thanks go to the authors of these packages for sharing their programs with others. In particular gTybalt is build on the following packages:

- The TeXmacs-editor is used to display the output of formulae in high quality mathematical typesetting using TeX fonts.
- gTybalt can also be run from a text window. Then the library eqascii is used to render formulae readable in text mode.

- Any interactive program needs an interpreter for its commands. gTybalt uses the CINT C/C++ interpreter which allows execution of C++ scripts and C++ command line input.
- At the core of any computer algebra system is the module for symbolic and algebraic manipulations. This functionality is provided by the GiNaC-library.
- One aspect of computer algebra systems is arbitrary precision arithmetic. Here GiNaC (and therefore gTybalt) relies on the Class Library for Numbers (CLN library).
- Plotting functions is very helpful to quickly visualize results. The graphic abilities of gTybalt are due to the Root-package.
- The GNU scientific library is used for Monte Carlo integration.
- Optionally gTybalt can be compiled with support for the expansion of transcendental functions. This requires the nestedsums library to be installed.
- Optionally gTybalt can be compiled with support for factorization of polynomials. This requires the NTL library to be installed.

Additional documentation: Since gTybalt incoporates GiNaC, Root and TeXmacs, you are advised to read also the documentation on GiNaC, Root and TeXmacs.

1.2 License

2 Tutorial for gTybalt

2.1 Starting gTybalt

gTybalt can be run in the TeXmacs mode or in a simple text mode. To start gTybalt in text mode, type `gtybalt`. To quit, type `quit`. To use gTybalt within TeXmacs, first start TeXmacs with the command `texmacs`. You can then start a gTybalt session by clicking on the terminal symbol and selecting “gTybalt” from the pop-up menu. Alternatively you can start gTybalt from the “Text” menu via “Text → session → gTybalt”.

2.2 Command line input

You can type in regular C++ statements which will be processed by CINT. For example

```
gTybalt> int i=1;
gTybalt> i++;
gTybalt> cout << "The increased number : " << i << endl;
The increased number : 2
```

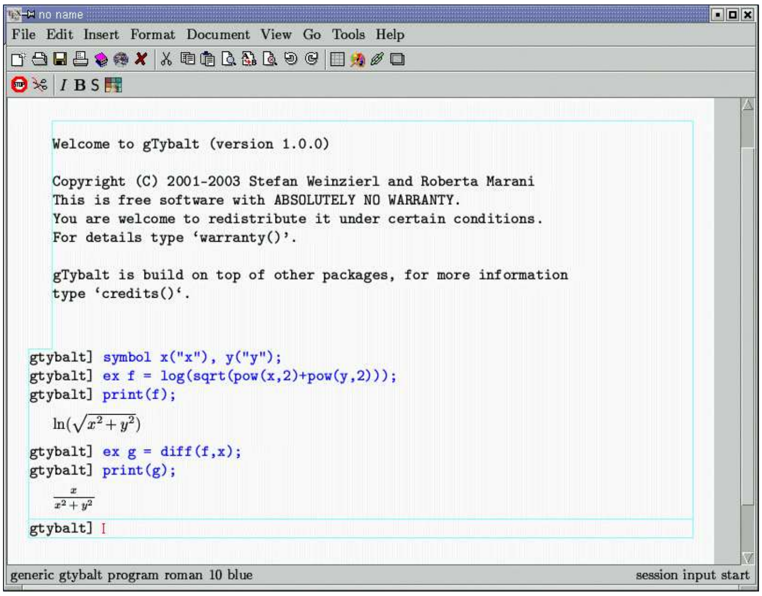
The functionality of gTybalt for symbolic and algebraic calculations is provided by the GiNaC-library. The syntax follows the one for the GiNaC-library. For example:

```
gTybalt> symbol a("a"), b("b");
gTybalt> ex e1=pow(a+b,2);
gTybalt> print(e1);

     2
(b+a)

gTybalt> ex e2=expand(e1);
gTybalt> print(e2);

 2    2
a  + b  + 2 a b
```

Here `print` is a gTybalt-subroutine, which prints a variable to the screen. By default, gTybalt does not print anything onto the screen, unless the user specifically asks for a variable to be printed. If gTybalt is running under TeXmacs, the output will be with TeX fonts. There is also a function `rawprint` which prints the variable `e2` as follows:

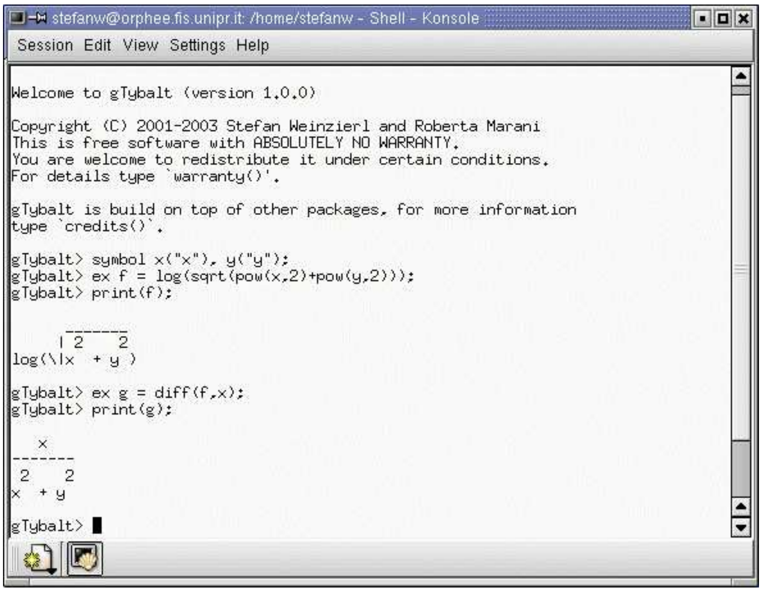
```
gTybalt> rawprint(e2);
a^2+b^2+2*a*b
```

The picture shows how the output of a further example will look like under TeXmacs.



A screenshot for gTybalt when running in TeXmacs mode.

The following figure shows the corresponding output, when gTybalt runs in text mode.



A screenshot for gTybalt when running in text mode.

Within TeXmacs mode there is the possibility to print a session to a postscript file by choosing from the "File" menu the combination "File → Export → Postscript". It is also possible to generate for a session a corresponding LaTeX file via "File → Export → LaTeX". This is in particular useful if one would like to obtain for a displayed formula the corresponding LaTeX code.

2.3 Scripts

The standard behaviour of the C++/C interpreter CINT is to interpret any command immediately. There is also the possibility to put a few commands into a script and to load this file into a session. This is done through the following commandds:

```
.L file.C
.x file.C
```

The `.L` command loads a script into the session, but does not execute the script. This is useful for a script containing the definition of a function. The `.x` command loads and executes a script. As an example consider that the file '`hermite.C`' contains the following code:

```
ex HermitePoly(const symbol & x, int n)
{
 ex HKer=exp(-pow(x,2));
 return normal(pow(numeric(-1),n) * diff(HKer,x,n)/HKer);
}
```

This is just a function which calculates the n-th Hermite polynomial. Now try the following lines in gTybalt:

```
gTybalt> .L hermite.C
gTybalt> symbol z("z");
gTybalt> ex e1=HermitePoly(z,3);
gTybalt> print(e1);

                3
 - 12 z + 8 z
```

This prints out the third Hermite polynomial. As a further example let the file '`main.C`' contain the following code:

```
{
 symbol z("z");
 for (int i=0;i<5;i++)
  {
   print( HermitePoly(z,i));
  }
}
```

This is called an "un-named script". Un-named scripts have to start with an opening "`{`" and end with a closing "`}`". Then the following lines in gTybalt

```
gTybalt> .L hermite.C
gTybalt> .x main.C
```

will print the first five Hermite polynomials.

gTybalt offers also the possibility to execute automatically a script at start time. If there is a logon-file '`gtybalt_logon`' in the directory from where gTybalt is started, the commands in this file will be executed when gTybalt is started. The default commands are

```
{
 gROOT->ProcessLine("#include <iostream>");
 gROOT->ProcessLine("#include <string>");
 gSystem->Load("libgtybaltdict.so");
 gROOT->ProcessLine("using namespace GiNaC");
 gROOT->ProcessLine("greeting();");
}
```

and you are advised to copy these lines into your personal logon-file. A logon-file can be used to include very often used standard definitions. An example would be the following line:

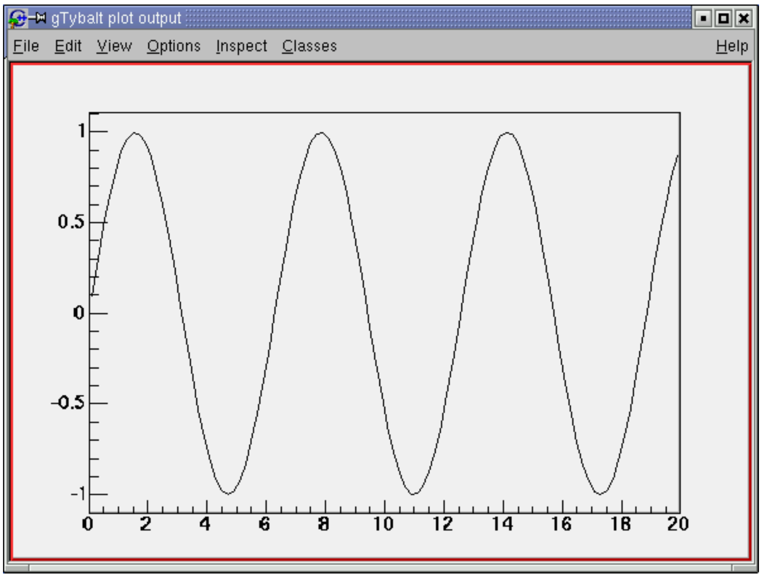
```
symbol x("x"), y("y"), z("z");
```

2.4 Plots

A function can be plotted as follows:

```
gTybalt> symbol x("x");
gTybalt> ex f1=sin(x);
gTybalt> plot(f1,x,0,20);
```

This will plot $\sin(x)$ in the intervall from 0 to 20. The plot will look like this:

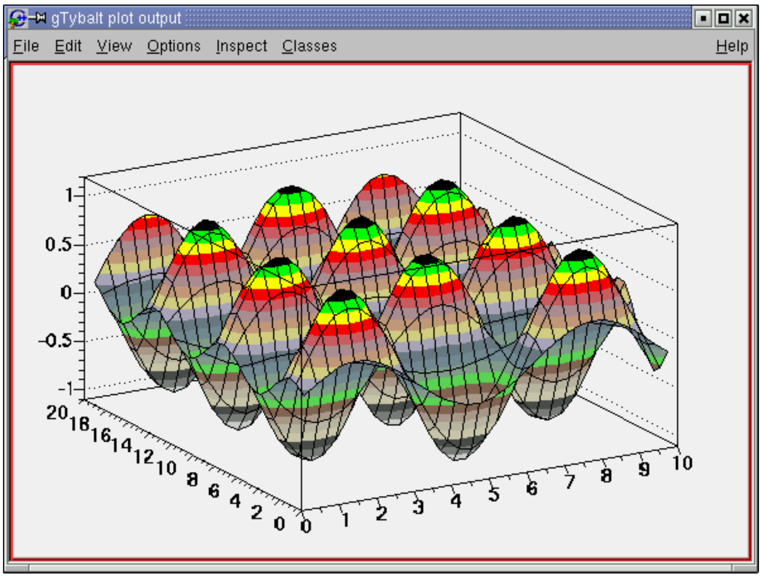


To clear the window with the plot, choose from the menubar of the plot "File → Quit ROOT".

Similar, a scalar function of two variables can be plotted as follows:

```
gTybalt> symbol x("x"), y("y");
gTybalt> ex f2=sin(x)*sin(y);
gTybalt> plot(f2,x,y,0,10,0,20);
```

This will plot $\sin(x)\sin(y)$ for x from 0 to 10 and y from 0 to 20. The plot will look like this:



To view the plot from a different angle, just grap the plot with the mouse and move it around. There is a wide variety of options on how to draw a graph. To access the draw panel, click on the right mouse button, when the mouse is placed inside the window containing the plot and choose "DrawPanel" from the pop-up menu. The options include among others lego- and contour-plots. The default option corresponds to the style "surf" and draws a (coloured) surface.

The plot can be saved to a file. For example, to save the plot as a postscript file, choose from the "File" menu the option "Save As canvas.ps".

2.5 Numerical integration

Functions can be integrated numerically by Monte Carlo integration. For example to evaluate

$$\int\limits_0^1 dx \int\limits_0^1 dy \int\limits_0^1 dz\; xyz$$

one types

```
gTybalt> symbol x("x"), y("y"), z("z");
gTybalt> ex f = x*y*z;
gTybalt> ex g = intnum(f,lst(x,y,z),lst(0,0,0),lst(1,1,1));
gTybalt> print(g);
0.12500320720479463077
```

The result of the integration can also be accessed with the help of the global variable `gTybalt_int_res`. In addition the global variables `gTybalt_int_err` and `gTybalt_int_chi2` give information on the error and the χ^2. For our example, one gets

```
gTybalt> print(gTybalt_int_res);
0.125003
gTybalt> print(gTybalt_int_err);
5.26234e-06
gTybalt> print(gTybalt_int_chi2);
0.385639
```

The Monte Carlo integration uses the adaptive algorithm VEGAS. gTybalt uses the implementation from the GNU Scientific Library. The algorithm first uses `gTybalt_int_iter_low` iterations with `gTybalt_int_calls_low` calls to obtain some rough information where the integrand is largest in magnitude. The results of this run are discarded, but the grid is kept. The algorithm then performs `gTybalt_int_iter_high` iterations with `gTybalt_int_calls_high` calls to obtain a Monte Carlo estimate. The default values are

```
gTybalt> print(gTybalt_int_iter_low);
5
gTybalt> print(gTybalt_int_calls_low);
10000
gTybalt> print(gTybalt_int_iter_high);
10
gTybalt> print(gTybalt_int_calls_low);
100000
```

The values of these variables can be adjusted by the user. For functions of one or two variables there are in addition the following simpler forms

```
double intnum(ex expr, ex x, double xmin, double xmax);
double intnum(ex expr, ex x, ex y, double xmin, double xmax,
                                   double ymin, double ymax);
```

2.6 Factorization

When gTybalt is compiled with the NTL library, gTybalt provides an interface to factorize univariate polynomials with integer coefficients. For example

```
gTybalt> symbol x("x");
gTybalt> ex f = expand( pow(x+2,13)*pow(x+3,5)*pow(x+5,7)*pow(x+7,2) );
gTybalt> ex g = factorpoly(f,x);
gTybalt> print(g);

          2        5        7         13
  (7 + x)  (3 + x)  (5 + x)  (2 + x)
```

If the first argument of the function `factorpoly` is not an univariate polynomial with integer coefficients, it returns unevaluated.

2.7 Expansion of transcendental functions

When gTybalt is compiled with the nestedsums library, gTybalt provides an interface to expand a certain class of transcendental functions in a small parameter. The class of functions comprises among others generalized hypergeometric functions, the first and second Appell function and the first Kampe de Feriet function. A hypergeometric function can be expanded as follows:

```
gTybalt> symbol x("x"), eps("epsilon");
gTybalt> transcendental_fct_type_A F21(x,lst(1,-eps),lst(1-eps),lst(1-eps),
         lst(1,-eps));
gTybalt> ex f = F21.set_expansion(eps,5);
gTybalt> rawprint(f);
-Li(3,x)*epsilon^3-Li(2,x)*epsilon^2-Li(4,x)*epsilon^4-Li(1,x)*epsilon
+Z(Infinity)
```

This expands the hypergeometric function ${}_2F_1(1,-\varepsilon;1-\varepsilon;x)$ in ε up to order 5 and agrees with the known expansion

$$ {}_2F_1(1,-\varepsilon;1-\varepsilon;x) = 1 - \varepsilon Li_1(x) - \varepsilon^2 Li_2(x) - \varepsilon^3 Li_3(x) - \varepsilon^4 Li_4(x) + \mathcal{O}(\varepsilon^5). $$

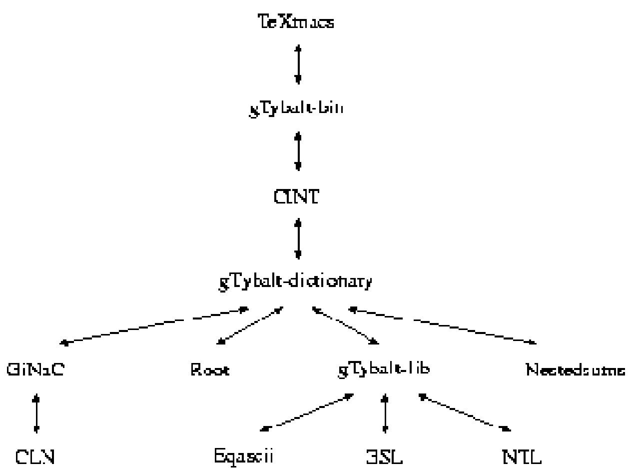
The algorithm for the expansion is based on an algebra for nested sums. Z(Infinity) represent the unit element in this algebra and is equal to 1.

3 Design of the program

This section gives some technical details on the design of the program and serves as a guide to the source code. The reader who is primarily interested in using the program just as an application may skip this section in a first reading. After a general overview of the system I discuss two technical points concerning threads and dynamic loading, where a few explanations might be useful to understand the source code.

3.1 Structural overview

A structural overview for gTybalt is shown in the following figure:



gTybalt consists of three parts, labelled `gTybalt-bin`, `gTybalt-dictionary` and `gTybalt-lib`, which ensure communication between the different modules on which gTybalt is based. The first part, `gTybalt-bin` is either called from TeXmacs (in TeXmacs-mode) or directly from the shell (when issuing the command "`gtybalt`" from a shell) and implements an event loop. This program reads input from the keyboard, sends the commands to the C++ interpreter CINT for execution and directs the output either to TeXmacs or to a text window.

The program CINT interprets the commands. For this purpose it uses a library called `gTybalt-dictionary`, which can be thought of as a look-up table where to find the actual implementations of the encountered function calls. The source code for this library is generated automatically during the build phase of gTybalt. A file "'`LinkDef.h`'" specifies which functions and classes to include into this library. The library is then generated from the header files for these functions and classes. The CINT interpreter is not 100% standard C++ compatible and there are certain constructs, which cannot be processed by CINT. Therefore the header files for the GiNaC-library are first copied to a temporary directory and then processed by a perl script, which comments out any parts which cannot be fed into CINT.

Finally, the library `gTybalt-lib` is an ordinary library, defining gTybalt-specific functions like `print`, `factorpoly` or `intnum`. It depends in turn on other libraries, like Eqascii, GSL or NTL, which are however not visible in the interactive interface.

3.2 Threads and the plotting routine

When plotting a function, it is desirable to have the window with the plot appearing on the screen, but at the same time still be able to work in the main window of gTybalt. Since there are now two possible actions which the user can take (e.g. typing new commands in the main window and modifying the plot inside the window with the plot) this is implemented using different threads. When starting gTybalt, the program will create a separate thread, which waits on a condition that a function should be plotted. When a plotting command is issued, CINT invokes a function, which just prepares some variables for the plot, signals that there is something to be plotted and then returns. Therefore after the return of this function the user can issue new commands in the main window of gTybalt. The thread waiting on the condition for plotting a function will wake up, plot the function and provide an event handler for events concerning the window with the plot. Therefore the user can now take actions in both the main window for gTybalt and the window with the plot. Once the window with the plot is cleared (by choosing from the menubar of the plot "File → Quit ROOT") the thread for plots will fall into sleep again and wait till another plotting command is issued. Thread safety is guarenteed by copying the relevant expressions for the function to be plotted to global variables and by the reference counting mechanism of GiNaC: The expression to be plotted will be pointed at by at least one (global) variable, therefore it will not be modified. While a plot is displayed on the screen, any command to plot another function will be ignored. The user must first clear the window with the plot.

3.3 Dynamic loading of modules and numerical integration

The default behaviour for numerical evaluation of a function uses the arbitrary precision arithmetic provided by the CLN library. For Monte Carlo integration, where a function needs to be evaluated many times, this is quite slow and therefore inefficient. It is also not needed, since statistical errors and not rounding errors tend to dominate the error of the final result. Therefore a different approach has been implemented for the numerical Monte Carlo integration: The function to be integrated is first written as C code to a file, this file is then compiled with a standard C compiler and the resulting executable is loaded dynamically (e.g. as a "plug-in") into the memory space of gTybalt and the Monte Carlo integration routine uses this compiled C function for the evaluations.

4 Installation

4.1 Prerequisities

gTybalt uses TeXmacs, Root, GiNaC and the GNU scientific library. These packages are needed before one starts to build gTybalt. TeXmacs depends on LaTeX and Guile Scheme, GiNaC on the CLN library. One therefore needs also these packages. The C++/C interpreter CINT is not needed as a separate package, since it is already included in Root.

Building gTybalt requires a ANSI-compliant C++-compiler. It is strongly recommended to use gcc. The version number of the gcc compiler can be obtained by

```
$ gcc -v
```

gTybalt 1.3.6 has been tested with the following versions for the different packages:

```
gcc 10.1.1,
GiNaC 1.8.0,
Root 6.20/06,
TeXmacs 1.99.12,
GSL 2.6,
CLN 1.3.4,
NTL 10.3.0,
nestedsums 1.5.2.
```

gTybalt has been tested with these versions and you are advised to use EXACTLY those. The NTL and nestedsums libraries are optional.

You can follow the steps below to build all relevant packages in the right order.

You need LaTex to be installed. You can check whether the LaTeX binaries exist in your path by

```
$ which latex
```

Usually LaTeX is already included in a standard Linux distribution.

You further need Guile Scheme to be installed. You can check by

```
$ which guile
```

If Guile is not installed, you can download it from

```
https://www.gnu.org/software/guile
```

and follow the installation instructions.

Installation of TeXmacs: Having LaTeX and Guile Scheme installed, you can build and install TeXmacs. TeXmacs can be obtained from

```
https://www.texmacs.org
```

Follow the installations instructions there.

Installation of Root: The package can be obtained from

```
ftp://root.cern.ch/root
```

Follow the installation instructions there. Note that the configuration command has a slightly different syntax as compared to the standard autotools configuration script. Further note that the Root package includes also the C++/C interpreter CINT.

Installation of the CLN library: GiNaC requires the CLN library, available from either one of the following FTP-sites:

```
ftp://ftp.santafe.edu/pub/gnu/
ftp://ftp.ilog.fr/pub/Users/haible/gnu/
ftp://ftpthep.physik.uni-mainz.de/pub/gnu/
```

Follow the installation instructions in the manual.

Installation of GiNaC: You find GiNaC at

```
https://www.ginac.de
```

Unpack the GiNaC source code and follow the installation instructions of GiNaC. There is no need to build the package ginac-cint, gTybalt uses the CINT implementation of Root and builds the necessary dictionary later.

Installation of the GNU scientific library: You find the GNU scientific library at

```
https://www.gnu.org/software/gsl
```

Unpack the source code and follow the installation instructions of the GNU scientific library.

Optionally gTybalt can be compiled with support for factorization of polynomials. In this case, the NTL library has to be installed. The NTL library is available from

```
https://www.shoup.net/ntl
```

Configure the NTL library with the option "NTL_STD_CXX=on". This puts the classes of the NTL library into a namespace "NTL". This option is needed by gTybalt.

Optionally gTybalt can be compiled with support for the expansion of transcendental functions. In this case, the nestedsums library has to be installed. The nestedsums library is available from

```
https://particlephysics.uni-mainz.de/weinzierl/nestedsums
```

Follow the installations instructions there.

4.2 Building gTybalt

gTybalt can be obtained from

```
https://particlephysics.uni-mainz.de/weinzierl/gtybalt
```

As with any autoconfiguring GNU software, installation proceeds through the following steps:

```
$ ./configure
$ make
[become root if necessary]
$ make install
```

The "configure" script can be given a number of options to enable and disable various features. For a complete list, type:

```
$ ./configure --help
```

A few of the more important ones:

- '`--prefix=`*PREFIX*': install architecture-independent files in *PREFIX* [defaults to '`/usr/local`']
- '`--exec-prefix=`*EPREFIX*': install architecture-dependent files in *EPREFIX* [defaults to the value given to '`--prefix`']
- '`--with-nestedsums`': If the nestedsums library is installed, one may give this option to enable support for the expansion of transcendental functions.
- '`--with-ntl`': If the NTL library is installed, one may give this option to enable support for the factorization of polynomials.
- '`--with-ntl-prefix=`*PREFIX*': If the NTL library is installed in an unusual location, the value of *PREFIX* informs the configure script where to find the NTL library.
- '`--with-gmp-prefix=`*PREFIX*': The NTL library can be compiled with the GMP library (GNU Multiple Precision Arithmetic Library). If the GMP library is installed in an unusual location, the value of *PREFIX* informs the configure script where to find the GMP library.

If you install the executables and the libraries in non-standard directories, you have to set the *PATH* and the *LD_LIBRARY_PATH* variables correspondingly.

4.3 Example of a complete installation

Here is an example of a complete installation. We assume that LaTex is already installed and that a suitable compiler exists. We further assume that we do not have root privileges, therefore we have to install the programs in an unusual directory, which we take to be `/home/username/local`. We further decide to build only shared libraries (where this is possible) and we would like to set up gTybalt with support for factorization and for the expansion of transcendental functions. This requires to install the NTL library and the nestedsums library. We decide to build the NTL library in conjunction with the GMP (GNU Multi-Precision library) for enhanced performance. Furthermore we assume that we have unpacked all relevant source packages into separate directories. The commands to build a specific package are entered from the top-level directory of the specific package. For concreteness we assume that we have a Linux PC with a Bourne compatible shell.

We start with Guile:

```
$ ./configure --prefix=/home/username/local
$ make
$ make install
```

We then set the environment variables (probably in a file like '`.bash_profile`'):

```
export GUILE_LOAD_PATH=/home/username/local
export PATH=/home/username/local/bin:$PATH
export LD_LIBRARY_PATH=/home/username/local/lib:$LD_LIBRARY_PATH
```

Building TeXmacs is done in a similar way

```
$ ./configure --prefix=/home/username/local
$ make
$ make install
```

In the next step we build the Root package:

```
$ ./configure linux --prefix=/home/username/local
  --etcdir=/home/username/local/etc
$ gmake
$ gmake install
```

and we set the variable *ROOTSYS* (probably in a file like '`.bash_profile`'):

```
export ROOTSYS=/home/username/local
```

The GMP library is build as follows:

```
$ ./configure --prefix=/home/username/local
$ make
$ make install
```

To build the NTL library we issue the commands:

```
$ cd src
$ ./configure PREFIX=/home/username/local NTL_STD_CXX=on
  NTL_GMP_LIP=on GMP_PREFIX=/home/username/local
$ make
$ make install
```

Note the slight modified form how the arguments are passed to the configure script. To build the CLN-library:

```
$ ./configure --prefix=/home/username/local --enable-static=no
$ make
$ make install
```

To build GiNaC:

```
$ ./configure --prefix=/home/username/local --enable-static=no
$ make
$ make install
```

To build the GNU scientific library:

```
$ ./configure --prefix=/home/username/local --enable-static=no
$ make
$ make install
```

To build the nestedsums library:

```
$ ./configure --prefix=/home/username/local --enable-static=no
$ make
$ make install
```

And finally we build gTybalt:

```
$ ./configure --prefix=/home/username/local --enable-static=no
  --with-nestedsums --with-ntl
  --with-ntl-prefix=/home/username/local
  --with-gmp-prefix=/home/username/local
$ make
$ make install
```

This completes the installation procedure.

4.4 Troubleshooting

This section gives some advice if the installation procedure does not work as it should.

A very common problem during compilation is that the building process for the dictionary fails, either through a segmentation fault or through excessive compilation time (> 5 min for a 1 GHz PC). A typical print-out might look likes this:

```
Making all in dictionary
make[2]: Entering directory `/home/stefanw/build/gtybalt-1.0.0/dictionary'
rootcint -f gtybaltdict.cc -c ../ginac_headers/ginac.h
  ../gtybalt-lib/routines.h ../gtybalt-lib/print.h ../gtybalt-lib/plot.h
  ../gtybalt-lib/globals.h ../gtybalt-lib/integrate.h
  ../gtybalt-lib/eqostream.h   LinkDef.h
make[2]: *** [gtybaltdict.cc] Segmentation fault
make[2]: *** Deleting file `gtybaltdict.cc'
make[2]: Leaving directory `/home/stefanw/build/gtybalt-1.0.0/dictionary'
make[1]: *** [all-recursive] Error 1
make[1]: Leaving directory `/home/stefanw/build/gtybalt-1.0.0'
make: *** [all-recursive-am] Error 2
```

In many cases this is due to incompatibility between CINT and the compiler. The solution is to use the recommended versions of Root and gcc.

If the versions of Root and gcc are correct, read the file debug/README and follow the instructions there.

A second problem is related to libtool and manifests itself as follows:

```
Making all in dictionary
make[2]: Entering directory `/home/stefanw/build/gtybalt-1.1.2/dictionary'
  ...
mv -f .libs/gtybaltdict.lo gtybaltdict.lo
/bin/sh ../libtool --mode=link gcc  -g -O2   -o libgtybaltdict.la
  -version-info 0:0:0  gtybaltdict.lo ../gtybalt-lib/libgtybalt.la
```

```
libtool: link: ‘gtybaltdict.lo’ is not a valid libtool object
make[2]: *** [libgtybaltdict.la] Error 1
make[2]: Leaving directory ‘/home/stefanw/build/gtybalt-1.1.2/dictionary’
make[1]: *** [all-recursive] Error 1
make[1]: Leaving directory ‘/home/stefanw/build/gtybalt-1.1.2’
make: *** [all] Error 2
```

If the GNU autotools (autoconf, automake and libtool) are installed on the system a simple solution is to replace the call to configure by a call to autogen.sh, e.g. to build gTybalt as follows:

```
./autogen.sh
make
make install
```

Appendix A Bibliography

- *GiNaC: An open framework for symbolic computation within the C++ programming language*, Christian Bauer, Alexander Frink and Richard Kreckel, `https://www.ginac.de`
- *CLN: A Class Library for Numbers*, Bruno Haible, `https://www.ginac.de/CLN`
- *ROOT: An object-oriented data analysis framework*, Rene Brun and Fons Rademakers, `https://root.cern.ch`
- *CINT: A C/C++ Interpreter*, Masaharu Goto, `https://root.cern.ch/root/Cint.html`
- *GNU TeXmacs*, Joris van der Hoeven, `https://www.texmacs.org`
- *Eqascii*, Przemek Borys, `https://dione.ids.pl/~pborys/software/linux`
- *GNU Scientific Library*, Mark Galassi, Jim Davies, James Theiler, Brian Gough, Gerard Jungman, Michael Booth and Fabrice Rossi, `https://www.gnu.org/software/gsl`
- *Nestedsums Library*, Stefan Weinzierl, `https://particlephysics.uni-mainz.de/weinzierl/nesteds`
- *NTL Library*, Victor Shoup, `https://www.shoup.net/ntl`
- *GMP Library: GNU Multiple Precision Arithmetic Library*, Torbjorn Granlund and Kevin Ryde, `https://gmplib.org`

Concept Index