Figure 1:

Kodiak, a C++ Library for Rigorous Branch and Bound Computation

Kodiak is a C++ library that implements a generic branch and bound algorithm for rigorous numerical approximations. Particular instances of the branch and bound algorithm allow the user to refine and isolate solutions to systems of nonlinear equations and inequalities, global optimization problems, and bifurcation sets for systems of ODEs. Kodiak utilizes interval arithmetic (via the *filib++* library) and Bernstein enclosure (for polynomials and rational functions) as self-validating enclosure methods. Symbolic operations support procedures such as automatic partial differentiation.

Authors

Cesar A. Munoz (cesar.a.munoz@nasa.gov), NASA Langley Research Center

Andrew P. Smith (andrew.smith@nianet.org), National Institute of Aerospace

Anthony Narkawicz, NASA Langley Research Center

Contributors

Mantas Markevicius (mm1080@york.ac.uk), University of York, UK

Documentation

Currently, the main documentation is contained in this file. Publications concerning the library and associated algorithms are in preparation. There are also numerous comments in the source code.

Support and Contribution

See the instructions in this file and the linked resources. For further assistance, or to report problems and bugs, please contact the authors. Contributions to the library are welcomed.

Obtaining *Kodiak*

The repository is located at: https://github.com/nasa/Kodiak/

License

The Kodiak C++ Library is released under NASA's Open Source Agreement. See the file LICENSE.pdf; see also the copyright notice at the end of this file.

Installation and Usage Options

Kodiak can either be installed from the source code, or, if available, a pre-built executable may be used instead.

The software can then be run in one of the following ways:

- Use the provided run-time interface, either interactively, or from an input file (either installation option)
- Encode a problem in a C++ program file, then compile and run it (only for installation from source code)

The following section assumes that you are installing the software from the source code; if you already have a pre-built or compiled executable, you can skip ahead to the section on using the software.

Installation

1. Prerequisites

It is recommended to run *Kodiak* on a Linux or Mac computer with the GNU C++ compiler; so far it has been successfully tested on Ubuntu Linux and Mac OS X. Use of Windows is not supported, although it ought to be feasible, and the authors would welcome any report of successfully running the software on Windows or any other system.

The following software should firstly be installed, if not already present (please follow the links for instructions and support):

- *Boost* libraries (required):
  http://www.boost.org/users/download/
  In addition to the headers, you need at least the library `serialization`. This library can be installed using `./bootstrap.sh --with-libraries=serialization` and then `sudo ./b2 install`. Finally, you need to define the environment variable `BOOST_ROOT` to point to the directory where Boost's `include` and `lib` directories were installed, e.g., `/usr/local`.
- *filib++* interval library (required):
  http://www2.math.uni-wuppertal.de/~xsc/software/filib.html
  This library should be configured with the following options before making and installing it:
  `./configure CFLAGS=-fPIC CPPFLAGS=-fPIC CXXFLAGS=-fPIC`
- Python language (recommended, necessary for the interface):
  http://www.python.org/
  There is a good chance that Python is already installed.
- *Cython*, a Python to C translator (recommended, necessary for the interface):
  http://cython.org/#download
- *PyInstaller* package utility (optional, necessary to build a standalone executable)
  http://www.pyinstaller.org/
- *gnuplot* plotting software (optional, necessary for graphical output)
  http://www.gnuplot.info/

2. Build Library and Examples

If necessary, unzip the zip file in order to extract the files. The following files and directories should be present:

- In the working directory: this readme file (in various formats), the license, and a makefile
- `src`: source code for the library
- `python`: code for the Python interface
- `examples`: example C++ files (`.cpp`) containing several problems
- `kdks`: example text files (`.kdk`) for use with the interface

If any of the prerequisite libraries were installed in non-standard directories, then the files `src/Makefile` and `python/Makefile` should be modified accordingly.

Type `make lib` to compile the library. The `lib` directory and a static library file are created.

Type `make examples` to compile the examples. An executable for each example is created in the `examples` directory.

3. Build Interface (Recommended)

Type `make python` to build the interface.

The program and interface can be run with the command:
`python python/kodiak.py`

4. Build Standalone Executable (Optional)

Type `make kodiak` to build a self-contained executable. The `bin` directory and a binary file are created.

It can be run with the command: `bin/kodiak`

Using the Interface

Starting the Interface

The program and interface can be run with the command:
`python python/kodiak.py`

If using the standalone executable, instead type:
`bin/kodiak`

You are now presented with a prompt and may issue commands (see below).

Exiting the Interface

To quit type `quit` or press `ctrl+c`.

Program Arguments

If not using the standalone executable, substitute
`python kodiak.py <arguments>` for
`bin/kodiak <arguments>` in the following:

The program can be called with a file or files as argument(s):
`bin/kodiak <file>`
It will process each file and print the output.

Files can be specified with an absolute or a relative path, e.g., to solve a problem contained in a file `bar.kdk` which is in the `/home/foo/` directory:
`bin/kodiak /home/foo/bar.kdk`

Files **must** have a `.kdk` file extension.

To pass multiple files as arguments:
`bin/kodiak <file> <file> ...`
You can add as many as you like.

An output file can be specified; if the file already exists, results are appended to it:
`bin/kodiak -o <output_file>`

To read files and write the output to a different file:
`bin/kodiak -o <output_file> <file> ...`

To switch on unsafe input mode (permits floating-point approximations to real numbers):
`bin/kodiak -u`

To save the (syntactically correct) session input to a file:
`bin/kodiak -s <save_file>`

To run in quiet mode, with no output to console:
`bin/kodiak -q`

To continue the session after processing input files:
`bin/kodiak <file> ...  -c`

To start *Kodiak* in debug mode:
`bin/kodiak -d`

For more help on the command line arguments:
`bin/kodiak -h`

All the aforementioned flags can be combined, e.g.:
`bin/kodiak -o foo -d -s bar -u -c foobar.kdk`

Using Input Files

Input files can be created with a text editor; as noted, the filename extension must be `.kdk`. There are numerous examples in the `kdks` directory.

Anything after `#` on a line is a comment and is ignored by the program:

```
# this is a comment
file test.kdk # this is also a comment
```

Each expression must be followed by a semicolon, e.g.:
`var x in [0,1]; var y in [1,5];`

To read an input file (either from the prompt or from another input file), the filename can contain a relative path or an absolute path to the file:
`file <file>`

Commands in Input Files and Interactive Mode

The commands listed below can equivalently be used either interactively, from the command prompt, or else (if followed by a semicolon) in an input file.

Paving Problems (Systems of Equations and Inequalities)

Variables and their ranges can be declared with the `var` keyword, e.g.:
`var <var_name> in [13, 42]`

The lower and upper bounds can be any integer in the range [-2147483648, 2147483647] on a 64-bit machine.

The bounds can also be set to either precise or approximate representations of floating-point numbers. Precise bounds can be set using one the following commands, where a and b are integers:
`rat(a,b)`, to input rational numbers
`dec(a,b)`, to input decimal numbers

Numbers in the hexadecimal floating-point format are also accepted. The format follows the rules for the C definition of hex floats, e.g., `0x1ap-2` specifies the number 6.5.

Approximate representations of numbers in usual floating-point format can be entered using the `approx(a)` command, but this is discouraged and to do this the safe mode of *Kodiak* has to be set to false.

Constants, declared with the `const` keyword, and global definitions, with the `def` keyword, are also supported. The difference between them is that constants can only contain a single number, whereas definitions can hold entire equations.

Pavings without any constraints are not very interesting. Constraints (either equality or inequality) can be added to the problem with the `cnstr` command, e.g.:
`cnstr x^2 + y < x`

A list of mathematical operations which can be used in formulae can be found later in this document.

There are three possible options for the paving mode:

- `first`: this produces the first sub-box found which is feasible
- `std`: this paves the entirety of the feasible set
- `full`: this paves both the feasible and infeasible sets

The paving mode can be set as follows:
`set paving mode = <mode>`

For further options to control the search space, see **Settings**, later in this guide.

Finally, to solve the paving problem, after it and the settings have been declared, use the `pave` command.

Bifurcation Problems (Bifurcation Analysis of ODEs)

Variables and their ranges can be declared as above, e.g.:
`var <var_name> in [13, 42]`

Bifurcation problems also take parameters, which can be declared similarly using the `param` keyword, e.g.:
`param <param_name> in [-42, 42]`

One or more ordinary differential equations are required. An ODE in the form <expr>=0 can be declared with `dfeq <expr>`, e.g.:
`dfeq x+sqrt(y)`

Constraints can be declared as above, e.g.:
`cnstr x^2 + y < x`

To pave the equilibrium manifold, use the command `equilibrium`; to pave the bifurcation manifold (limit point and Hopf bifurcations), which is a subset thereof, use the command `bifurcation`.

Minimization and Maximization Problems (Global Optimization)

For optimization problems, variables, constants, definitions, and constraints can be declared, as above. In addition, an objective function is needed; it is declared with the `objfn` keyword, e.g., `objfn sin(x)+2*y`

The solve commands `min`, `max`, and `minmax` are used to find an enclosure for the minimum, the maximum, and both the minimum and maximum, respectively.

Plotting

Both paving and bifurcation problems support graphical output using the *gnuplot* software. The projection of a paving into the space of selected variables is depicted. To plot the last solved problem in 2D or 3D, issue the `plot` command with two or three variables, e.g.:
`plot x y z`

To avoid having to redo calculations each time you start *Kodiak*, you can save the most recently produced paving with the `save paving` command. For example, `save paving foo` will save the paving to the file `foo.pav` in the current directory.

A paving can be loaded into *Kodiak* using the `load paving` command, e.g., `load paving foo` loads the paving in the file `foo.pav`.

Settings

To set the name of the problem: `set name = <name>`

To print out extra diagnostic information when solving problems: `set debug = true|false` (default: `false`)

To switch off safe input: `set safe input = true|false` (default: `true`). When safe input is on, approximate (i.e. floating-point) input values are not permitted.

To set the solver to use Bernstein enclosure in place of interval arithmetic where possible: `set bp = true|false` (default: `false`)

To set a resolution for all variables: `set resolution = <real_number>` As a termination criterion, the resolution is a non-negative number that specifies the smallest range (box width) where a bisection is considered for that variable.

To set a resolution for a specific variable: `set resolution <var_name> = <real_number>`

To set a precision for all variables: `set precision = <integer>`. If the precision (typically a small negative integer) is set to n, then the tolerance for the "almost certainly true" category of boxes is set to 10^n. It is not required to use this setting and this category of boxes, although doing so can speed up some problems.

To set the variable selection mode for branch and bound subdivision: `set selectvar = 0|1|2` (default: 1). This option is only used for minimization/maximization problems. If 0, round-robin is used to select a subdivision variable; if 1, a heuristic is applied to the objective function; if 2, a heuristic is applied to the constraint system.

To set the maximum depth (in the search tree) for the branch and bound algorithm: `set depth = <natural_number>`

To print the output of the solver to a file: `set output = <file>`

To clean the program state, so that new problems can be entered: `reset`

If output was set using the `-o` flag or the `set output` command, to reset output back to the console: `reset output`

Declaration and Expression Types

Numerical values: `<num_val> = <integer> | approx(<real>) | rat(<integer>, <integer>) | dec(<integer>, <natural_number>) | <hex_constant>`

Variable declarations: `var <name> in [<num_val>, <num_val>]`

Parameter declarations: `param <name> in [<num_val>, <num_val>]`

Constant declarations: `const <name> = <num_val>`

Mathematical expressions: `<math_expr> = <num_val> | <var> | <param> | <const> | <unary_operator>(<math_expr>) | <math_expr> <binary_operator> <math_expr> | let <name> = <math_expr> in <math_expr> | let <name> = <math_expr>, <name> = <math_expr>, ... in <math_expr>`

Example: `sin(approx(-1.1)*y)*sqrt(rat(3,2))^3-x`

Objective function (optimization problems): `objfn <math_expr>`

Differential equation of the form <math_expr> = 0 (bifurcation problems): `dfeq <math_expr>`

Constraint: `cnstr <math_expr> <relational_operator> <math_expr>`

Supported Relational (Boolean) Operations

`<`: less than
`>`: greater than
`<=`: less than or equal to
`>=`: greater than or equal to
`=`: equal to

Supported Mathematical Operations

`+`, `-`, `*`, `/`: addition, subtraction, multiplication, division
`^`: exponentiation (to an integer power)
`sq`, `sqrt`: square, square root
`sin`, `cos`, `tan`: trigonometric functions
`asin`, `acos`, `atan`: inverse trigonometric functions
`abs`: absolute value
`ln`, `exp`: natural logarithm, exponential

Solve Commands

`pave`: compute a paving for the feasible/infeasible set
`equilibrium`: compute a paving for the equilibrium set
`bifurcation`: compute a paving for the bifurcation set
`minmax`: compute an enclosure for the minimum and maximum
`min`: compute an enclosure for the minimum
`max`: compute an enclosure for the maximum

Using the Library

The *Kodiak* library can be used in your own C++ programs. A good way to start is to take one of the existing `.cpp` files in the `examples` directory and adapt to your purposes. You can either invoke the compiler directly with a link to the *filib++* and *Kodiak* libraries, or else add a new entry to the makefile there.

The commands and keywords listed above have member function equivalents (see the source code), although some of the names vary slightly. For example, there is a difference between, e.g., `sin` and `Sin`; the former is the operator for floating-point numbers and intervals, the latter is the operator for real expressions. In some cases, care must be taken with the order in which commands are issued. All variables should be declared before any variable resolutions are set.

Copyright Notices

See the file LICENSE.pdf for the license and copyright of the *Kodiak* library.

PLY (Python Lex-Yacc) Version 3.4

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of the David Beazley or Dabeaz LLC may be used to endorse or promote products derived from this software without specific prior written permission.

PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Version

Kodiak readme file, ver. Nov 2014

Logo

The Kodiak logo was designed by Mahyar Malekpour (NASA).