Kodiak

Kodiak is a C++ Library that implements a generic branch and bound algorithm for rigorous numerical approximations. Kodiak supports the following enclosure methods:

- Interval arithmetic (via the FLIB++ library).
- Bernstein polynomials.

Building Kodiak C++ Library

1. Install BOOST.
2. Install FILIB++. This library has to be configured with the following options before making and installing it: `./configure CFLAGS=-fPIC CPPFLAGS=-fPIC CXXFLAGS=-fPIC`
3. Type `make src`.
4. Examples of the use of the library are found in the directory `examples`. For building the examples, type `make examples`.

Mantas Markevicius, from University of York, UK, implemented a standalone interface for Kodiak in Python.

Building Kodiak's Python Interface

1. Build Kodiak C++ library.
2. Install Cython.
3. Type `make python`
4. Run standalone interface through the command `python python/kodiak.py`.

Building Kodiak's Self-Containing Executable

1. Install Kodiak's Python interface.
2. Install PyInstalled PyInstaller.
3. Type `make kodiak`
4. Run self-containing executable through the binary program `bin/kodiak`

If any of the libraries was installed in non standard directories, modify the files `src/Makefile` and `python/Makefile` accordingly.

Commands for Standalone Interface

Exiting the Interactive Interface

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

Program arguments

Program can be called with a file as arguments. In this case, it will evaluate each files and print the output:
```
./kodiak <file>
```
The same commands can be used if running from source:
```
python kodiak.py <file>
```

Files can be specified with an absolute or a relative path.
i.e. to solve a problem contained in a file *bar.kdk* which is in */home/foo/* folder:
```
./kodiak /home/foo/bar.kdk
```

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

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

Program can be given a flag to specify an output file, if the file already exists results are appended to it:
```
./kodiak -o <output_file>
```

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

Safe mode flag can be set to false the using command line arguments:
```
./kodiak -u
```

To save the syntactically correct input to a file:
```
./kodiak -s <save_file>
```

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

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

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

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

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

Interactive mode commands or data file syntax

To read a file, the file name can contain a relative path or an absolute path to file
`file <file>`

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

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

When writing data files, each expression must be followed by a ; semicolon i.e.
`var x in [0,1]; var y in [1,5];`

Supported Problem Types

Paving problems Paving problems only require variables, which can be defined with *var* keyword, i.e.:
`var var_name in [13, 42]`

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

There are also options to set bounds to precise and approximate representation of floating point number to **Kodiak**:
Precise bounds can be set using one the following commands:
rat(n,m), to input rational numbers.
dec(n,m), to input decimal numbers.
Where both n and m are integers.

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

Approximate representation of numbers can be entered using a *approx(n)* command, but this is discouraged and to do this the safe mode of **Kodiak** has to be set to false.

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

However, pavings without any constraints are not very interesting. Constraints can be added with the *cnstr* keyword, i.e.:
`cnstr x^2 + y < x`

More mathematical operations which can be used when defining equations can be found in the following sections of this document.

To have more control over the search space you can specify the paving mode. Possible options are *first*, *std*, *full*.
The search mode can be set using the following command:
`set paving mode = first`

More options to control the search space are described in the settings section of this guide.

Kodiak can be told to solve the paving problem, when all the parameters are described, using the *pave* keyword.

Bifurcation problems N.B. include short description about bifurcations?

Bifurcation problems require variables, just like paving problems, which can be defined with *var* keyword, i.e.:
`var var_name in [13, 42]`

But bifurcation problems also take parameters, which can be defined using *param* keyword, i.e.:
`param param_name in [-42, 42]`
Describing parameters supports exactly the same commands as variable descriptions.

Bifurcation problems also require differential expressions, which can be supplied to **Kodiak** using a *dfeq* keyword, i.e:
`dfeq x+sqrt(y)`

Constraints can also be supplied for bifurcation problems using the *cnstr* keyword, i.e. :
`cnstr x^2 + y < x`

Kodiak can be told to solve the paving problem, when all the parameters are described using the *bifurcation* keyword.

To solve a special type of bifurcation problems, equilibrium problems, the solve command is *equilibrium*.

Minimisation and Maximisation problems N.B. include description about optimization problems.

These types of problems take exactly the same arguments as paving problems. You can assign variables, constants, definitions and constraints to optimization problems.

You can issue solve commands *min* to solve a minimization problem, *max* to solve maximization problem and *minmax* to solve minimization and maximization.

Plotting

Both paving and bifurcation problems support plotting out the output using the **gnuplot** tool.

To plot out the last solved problem type in `plot` with the variables names following it, i.e.
`plot x y z`, would produce a 3D plot of the paving with the projection of these variables. 2D plots are also supported.

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

This paving can be loaded into **Kodiak** using *load paving* command, which is used like this:
`load paving foo`, would load the paving in file **foo.pav** into **Kodiak**.

Settings

Set the name of the problem

```
set name = name
```

Print out extra information when solving problems.

```
set debug = true|false
```

When safe input is true approx values are not allowed, the default value is *true*.

```
set safe input = true | false
```

Set the solver to use Bernstein Polynomials where possible.

```
set bp = true|false
```

Set precision for all the variables.

```
set precision = natural_number
```

Set resolution for all the variables .

```
set resolution = real_number
```

Set resolution for specific variable.

```
set resolution name = real_number
```

Set variable selection mode for Branch and Bound algorithm.

```
set selectvar = nat
```

Set maximum depth for the Branch and Bound algorithm.

```
set depth = nat
```

Print the output of the solver to file

```
set output = FILE
```

Cleans the state of the program, so that new problems can be entered

```
reset
```

If output was set using -o flag or set output command, this flag resets output back to the console

```
reset output
```

Problem definitions

Numerical declarations

```
num_decl = natural number, approx(real), rat(nat, nat), dec(nat,nat), hex_constant
```

Declaring variables

```
var name in [num_decl, num_decl]
```

Declaring parameters

```
param name in [num_decl, num_decl]
```

Declaring constants

```
const name = num_decl
```

Mathematical expression

```
math_exp = sin(approx(-1.1))*sqrt(rat(3,2))^3-x
```

Setting objective function for MinMax problems

```
objfn math_exp
```

Adding differential equations for bifurcation problems

```
dfeq math_exp
```

Adding constraints for all types of problems

```
cnstr math_exp bool_op math_exp
```

Let expressions used to define problems

```
objfn let name = math_exp in math_exp
cnstr let name = math_exp in math_exp
dfeq let name = math_exp in math_exp
```

Nested let expressions

```
objfn|dfeq|cnstr = let name = math_exp, name = math_exp in math_exp
```

Supported boolean operators Less than `<`
More than `>`
Less than or equal `<=`
More than or equal `>=`
Equal `=`

Supported mathematical operations Standard mathematical operator are supported `+ - * / ^`

Square root `sqrt(math_exp)`
Square or num_decl^2 `sq(math_exp)`
Sine `sin(math_exp)`
Cosine `cos(math_exp)`
Tangent `tan(math_exp)` Arc Sine `asin(math_exp)`
Arc Cosine `acos(math_exp)`
Arc Tangent `atan(math_exp)`
Absolute value `abs(math_exp)`
Natural logarithm `ln(math_exp)`
Exponential `exp(math_exp)`

Solve commands

Finding a paving for problem `pave`
Finding a bifurcation for problem `bifurcation` Finding an equilibrium for probelm `equilibrium`
Find minimum and maximum for the set differential equation `minmax`
Find minimum for the set differential equation `min`
Find maximum for the set differential equation `max`

Copyright notices: Kodiak (See LICENSE.pdf)