Module 6

Ryo Kimura

6 Debugging

In this module we will discuss what you will likely spending the most time doing during coding; reading documentation and debugging.

6.1 Reading documentation

Documentation comes in many forms: official and unofficial, in online forums and in published user guides, in source code and in compiled API docs. Here we will cover some tips for reading and navigating documentation that you will likely encounter.

6.1.1 `man` pages

Most commands that can be run on the command line in Linux come with documentation in the form a ***`man`*** *page* that explains its basic functionality.

You can view the man page of any command with the `man` command (e.g., `man wget` to see the man page for the `wget` command). The corresponding man pages is then opened using a basic text file viewer called `less`. `less` uses single letter commands to navigate the man page; here are the most useful ones which should suffice to view most man pages:

`h`	display help	`/pattern`	search forward for `pattern`
↓, `j`	scroll down one line	`?pattern`	search backward for `pattern`
↑, `k`	scroll up one line	`n`	repeat previous search
space, `f`	scroll down one page	`N`	repeat previous search in reverse
`b`	scroll up one page	`q`	quit and exit
`g`	go to first line		

6.1.2 Gurobi documentation

Gurobi's official documentation is fairly well-written and easy to navigate:

- If you are looking for documentation on how to use a specific class/function, use the search bar in the top right. Otherwise, the easiest way to browse the documentation is via the navigation bar on the left side of the screen. See the Quick Start Guides for a concrete example on how to solve a model from start to finish. See the Reference Manual for specific details on the API.
- Make sure to set the Version and Languages appropriately on the left navigation bar.

- Most functions and classes are intuitively named, and the top page of the corresponding section in the Reference Manual has a high-level overview. This includes attribute settings, parameter settings, solver status codes, error codes, and callback codes.
- Most relevant documentation for the Python and C++ APIs are found in Python API Overview and C++ API Overview sections of the Reference Manual.

Documentation Link: https://www.gurobi.com/documentation/

6.1.3 CPLEX documentation

CPLEX's official documentation, which is part of IBM Knowledge Center, can be a little difficult to navigate if you are not used to it. Here are some tips:

- If you are looking for documentation on how to use a specific class/function, use the search bar in the top right. Otherwise, the easiest way to navigate is via the Table of Contents on the left side of the screen (if it's closed, hover over it and click on the pin icon.)
- Make sure you are looking at the documentation for the version of CPLEX you are using. This is indicated at the top of the page (e.g., Home » ILOG CPLEX Optimization Studio 12.9.0 » CPLEX). You can change the version of CPLEX documentation by using the Change version or product dropdown menu in the top left section of your screen.
- If you are looking for information on a parameter setting, go to CPLEX » Parameters of CPLEX, and look in Topical list of parameters or List of CPLEX parameters.
- If you are looking for a high-level overview of a particular feature/concept (e.g., callbacks, Benders annotation), look under CPLEX » User's Manual for CPLEX typically in Advanced programming techniques or Discrete optimization. If you want to know what function to use, CPLEX » Overview of the APIs of CPLEX can be useful.
- Most relevant documentation for the C++ API is found in CPLEX » CPLEX C++ Reference Manual » `optim.concert` in either the *Classes* or *Functions* section. The major exception is `IloCplex`, which is documented under `optim.cplex.cpp` instead of `optim.concert`.
- Most relevant documentation for the legacy Python API is in the `cplex._internal._subinterfaces` section. It is helpful to read the **Structure of the package** `cplex` section on the overview page (https://www.ibm.com/support/knowledgecenter/SSSA5P_latest/ilog.odms.cplex.help/CPLEX/Python/topics/PLUGINS_ROOT/ilog.odms.cplex.help/refpythoncplex/html/overview.html) to understand how the package is organized.
- Most relevant documentation for DOcplex is in `docplex.mp.model`, `docplex.mp.linear`, or `docplex.mp.solution` module pages. Note that DOcplex documentation is hosted on Github, **NOT** IBM Knowledge Center.

Documentation Link: https://www.ibm.com/support/knowledgecenter/en/SSSA5P_latest/ilog.odms.studio.help/Optimization_Studio/topics/COS_home.html

6.1.4 Organization of CPLEX Legacy Python API

Cplex (`c = Cplex()`): class encapsulating a CPLEX problem

- **RootParameterGroup** (`c.parameters`): class containing all the CPLEX parameters
- **VariablesInterface** (`c.variables`): methods for adding, querying, and modifying variables
  - **AdvancedVariablesInterface** (`c.variables.advanced`): methods for advanced operations on variables *(protection, tightening bounds)*
- **LinearConstraintInterface** (`c.linear_constraints`): methods for adding, modifying, and querying linear constraints
  - **AdvancedLinearConstraintInterface** (`c.linear_constraints.advanced`): methods for handling lazy cuts and user cuts
- **QuadraticConstraintInterface** (`c.quadratic_constraints`): methods for adding, modifying, and querying quadratic constraints
- **IndicatorConstraintInterface** (`c.indicator_constraints`): methods for adding, modifying, and querying indicator constraints
- **SOSInterface** (`c.SOS`): class containing methods for Special Ordered Sets (SOS)
- **ObjectiveInterface** (`c.objective`): contains methods for querying and modifying the objective function
- **MultiObjInterface** (`c.multiobj`): methods for adding, querying, and modifying multiple objectives
- **MIPStartsInterface** (`c.MIP_starts`): contains methods pertaining to MIP starts
- **SolutionInterface** (`c.solution`): methods for querying the solution to an optimization problem
  - **ProgressInterface** (`c.solution.progress`): methods to query the progress of optimization
  - **InfeasibilityInterface** (`c.solution.infeasibility`): methods for computing degree of infeasibility in a solution vector
  - **MIPSolutionInterface** (`c.solution.MIP`): methods for accessing solutions to a MIP
  - **BasisInterface** (`c.solution.basis`): methods for accessing the basis of a solution
  - **SensitivityInterface** (`c.solution.sensitivity`): methods for sensitivity analysis
  - **SolnPoolInterface** (`c.solution.pool`): methods for accessing the solution pool
    - **SolnPoolFilterInterface** (`c.solution.pool.filter`): methods for solution pool filters
  - **AdvancedSolutionInterface** (`c.solution.advanced`): advanced methods for acessing solution information
  - **MultiObjSolnInterface** (`c.solution.multiobj`): methods for accessing solutions for multi-objective models

- **PresolveInterface** (`c.presolve`): methods for dealing with the presolved problem
- **OrderInterface** (`c.order`): methods for setting and querying a priority order for branching
- **ConflictInterface** (`c.conflict`): methods for identifying conflicts among constraints
- **AdvancedCplexInterface** (`c.advanced`): advanced control of a Cplex object *(basic presolve, pivoting, strong branching)*
- **InitialInterface** (`c.start`): methods to set starting information for an optimization algorithm to solve continuous problems (LP, QP, QCP)
- **FeasoptInterface** (`c.feasopt`): finds a minimal relaxation of the problem that is feasible
- **LongAnnotationInterface** (`c.long_annotations`): methods for adding, querying, and modifying long annotations
- **DoubleAnnotationInterface** (`c.double_annotations`): methods for adding, querying, and modifying double annotations
- **PWLConstraintInterface** (`c.pwl_constraints`): methods for adding, querying, and modifying PWL constraints

6.2 Debugging: Bug Prevention

Put simply, *debugging* is the process of fixing mistakes in your code. This is often a time-consuming process, and often constitutes the bulk of time spent coding. While in general debugging is a skill that must be developed over time with experience, there are some general strategies that are commonly used in the debugging process.

Some of the most important debugging strategies are *bug prevention strategies*, which aim to prevent bugs from occurring in the first place. We cover a few of the most common ones here: *incremental/modular development*, *use of standard libraries*, and *good coding practice*.

6.2.1 Incremental/Modular Development

One such bug prevention strategy is to *implement small chunks of the code at a time* and check they are working before moving on to the next one. This simple but effective strategy makes it easier to catch and pinpoint errors early, since you can narrow your focus to what was last changed. *Unit testing* is a systematic method for implementing this approach, see section 6.4 for more details.

A related strategy is to *structure your code in a "modular" manner*, i.e., implement smaller pieces of code that perform individual tasks which make up the whole, rather than trying to implement everything in a single main function. This encourages code re-use which makes the code more robust, since there are fewer places to change when you want to modify our adjust your code later. It also tends to result in simpler, more concise code, further reducing the change for errors.

6.2.2 Python/C++ Standard Libraries

Another bug prevention strategy is to *use functions from the language's standard library* whenever possible. Not only does it save development time, it eliminates a potential source of errors and bugs by using code that has been written and checked by experienced programmers.

Here are some modules in Python's standard library that may be especially useful:

- Built-in data structures: Python's built-in types `list`, `set`, and `dict` have many features and can be quite versatile
- Built-in functions: `int()` and `str()` provide explicit conversion to integer and string objects; `all()` and `any()` provide convenient iterators over iterables with boolean values; `min()`, `max()`, and `sum()` are self-explanatory
- `collections` implements common container datatypes, such as the factory function `namedtuple()` for creating tuple subclasses with named fields (e.g., `Point = namedtuple('Point', ['x', 'y'])`) and the `deque` class which implements a double-ended queue (the `heapq` module provides an implementation of priority queues)
- `itertools` provides several functions for creating iterators and enumeration; for example, `itertools.combinations(L)` returns an iterator over all combinations of elements from the list `L`
- `math` implements many useful functions commonly used in math
- `random` provides several convenient functions for pseudorandom number generation
- `decimal` implements a class for performing arbitrary precision decimal arithmetic
- `csv` and `json` provide convenient functions for interacting with CSV and JSON files
- `inspect` provides several useful functions for getting information about objects

C++ has several useful facilities as well:

- The Containers library includes class templates for common data structues such as static and dynamic continuous arrays (`<array>`, `<vector>`), stacks (`<stack>`), double-ended queues (`<deque>`), and associative containers (`<map>`, `<set>`)
- `<string>` includes the `basic_string` class for manipulating strings and conversion functions like the general-purpose `to_string` as well as the more specific `stoi`, `stol`, and `stof`
- The Utility library includes useful things including classes for dynamic memory management (`unique_ptr` in `<memory>`), date and time utilities (`<chrono>`), and generic tuples (`pair`, `tuple` in `<utility>`)
- `<algorithm>` implements many functions for common algorithmic tasks such as binary search (`binary_search`), sorting (`sort`), finding the first matching element (`find`), removing duplicate elements (`unique`), lexicographic comparison (`lexicographical_compare`), and constructing heaps (`make_heap`)
- `<cmath>` provides standard C library mathematical functions
- `<random>` provides utilities for pseudorandom number generation

6.2.3 Good Coding Practice

Finally, here are some miscellaneous strategies that get categorized under "good coding practice" that are really about bug prevention:

- Always initialize variables with some value. Many tricky bugs can stem from uninitialized variables.
- Define variables as close to the point in the source code when they get used as possible. This naturally encourages modular development and safeguards against unintended variable values.
- Use `try-catch` blocks to handle errors gracefully. Throw exceptions when you encounter an error in your code.
- Use `assert` statements to check assumptions about the state of the code. See https://github.com/emre/notes/blob/master/python/when-to-use-assert.md for more details on when to use assert (in Python).
- Avoid using C-style code in C++, e.g.
  - use `std::cout << k << std::endl` instead of `printf("%d\n", k);`
  - use `std::array<int>` and `std::vector<int>` instead of `int[]`
  - use `std::string` instead of `char*`
  - use references (`Model& p`) and smart pointers (`std::unique_ptr<Model> p`) instead of raw pointers (`Model *p`)
  - use `new` and `delete` instead of `malloc` and `free`

6.3 Debugging: Bug Search and Analysis

When your code does inevitably have a bug, you must search for where the bug took place, and then determine how best to fix it.

6.3.1 Binary Search

Norman Matloff, a professor from UC Davis, gives the following characterization of the debugging process in his *Guide to Faster, Less Frustrating Debugging* for C programming (see http://heather.cs.ucdavis.edu/~matloff/UnixAndC/CLanguage/Debug.html):

> When your program contains a bug, it is of course because somewhere there is something which you believe to be true but actually is not true. In other words:
>
> > **Finding your bug is a process of confirming that many things you believe are true, until you find one which is not true.**
>
> Here are examples of the types of things you might believe are true:
>
> - You believe that at a certain point in your source file, a certain variable has a certain value.
> - You believe that in a given if-then-else statement, the "else" part is the one that is executed.

- You believe that when you call a certain function, the function receives its parameters correctly.

So the processing of finding the location of a bug consists of confirming all these things! If you believe that a certain variable should have a certain value at a certain point, check it! If you believe that the "else" construct above is executed, check it!

Usually your belief will be confirmed, but eventually you will find a case where your belief will not be confirmed–and you will then know the location of the bug.

Since debugging is the process of searching where in the code your belief is not confirmed, Matloff suggests a *binary search* approach to efficiently narrow down the location of the bug (i.e., check that the belief is true at the halfway point in the source code, then check at the one-quarter or three-quarters points, and so on). While the strategy is simple, it can be very helpful to think of debugging as a search problem especially when you are not sure of the exact cause of the bug.

The search process can be made easier by using good text editors and debuggers. See section 6.5 for more specfics.

6.4 Unit Testing

Unit testing is a method in which individual *units* of source code are tested to determine whether they are fit for use. Unit tests are typically minimal examples of code usage to test whether a particular piece of functionality actually works as intended.

For example, suppose you are implementing a Graph class to represent an undirected graph. You may then have unit tests to check the following functionality:

- Adding and removing nodes
- Creating an empty graph
- Computing connected components
- Computing the shortest s–t path

So why should you use unit tests? In short, unit tests provide an *automated* method to ensure that your code implements the intended functionality. Do you ever test newly implemented code using small scripts? If you put those in a unit test, you can easily check *all* of them at any time, and you can check that any changes you make later does not break any functionality you already have (e.g., *regression testing*).

For Python, **pytest** is recommended for its simple syntax and rich functionality; see https://docs.pytest.org/en/latest/ for details. The **unittest** module in the Python standard library also provides a unit testing framewor, though may be a bit harder to use; see https://docs.python.org/3/library/unittest.html for details.

For C++, **doctest** is recommended for its header-only implementation, straightforward syntax, and fast compile times; see https://github.com/onqtam/doctest for details.[1] **googletest** is another popular unit testing framework with extensive functionality but more involved setup and a slightly steeper learning curve; see https://github.com/google/googletest for details.

[1] Also see https://a4z.bitbucket.io/blog/2018/05/17/Speed-up-your-test-cycles-with-CMake.html for details on using doctest with CMake.

See https://mtlynch.io/good-developers-bad-tests/ for details on writing good unit tests (and how it's different from writing good source code). See https://people.orie.cornell.edu/snp32/orie_6125/testing/testing.html for more details on test-driven development more broadly.

6.5 Debuggers

Debuggers are programs that are designed to assist in searching for bugs as you run your code. Debuggers can provide a more convenient method to track the state of variables, check conditions, and analyze what is happening line by line.

gdb/ddd

6.5.1 How to Use GDB

The **GNU Project Debugger (GDB)** is a ubiquitous debugger for C and C++ programs. While the minimal command line interface[2] can be a bit intimidating, it can be a very powerful tool for figuring out where a bug takes place.

GDB works by executing a program while keeping track of the state of information associated with the program. It runs the program normally until it encounters a *breakpoint*, at which it will pause execution and allow you to step through the program line by line, continue to the next breakpoint, see the value of a variable, etc. The following steps highlight the general process of debugging with GDB:

1. Decide where to set a breakpoint in the source code (e.g., `example.cpp`), either by line number or by function name.
2. Compile your code with debugging flags enabled (e.g., `-g` for GCC).
3. Run `gdb example` on the command line.
4. Use `break <linenum>` or `break <funcname>` to set a breakpoint at line number `linenum` or call to function `funcname`. Use `info breakpoints` to show all breakpoints.
5. Use `watch <var>` to set a breakpoint that triggers whenever the value of `var` changes.[3]
6. Run through the program using various options:
   - Use `run` to start the program from the beginning to the next breakpoint.
   - Use `continue` to continue running the program to the next breakpoint.
   - Use `next` to step through code one line at a time, passing over functions.
   - Use `step` to step through code literally one line at a time.
   - Use `kill` to kill the program process. You can then restart the program from the beginning again with `run`.
7. Once you have reached a suitable point in the program, analyze the state using various options:

[2] Many IDEs are able to run GDB from the editor itself. There is also a graphical front-end to GDB called **DDD** that also may be of interest; see https://www.gnu.org/software/ddd/ for more details.

[3] NOTE: As of April 2019, `watch` breakpoints do not work in Windows Subsystem for Linux (WSL) for unknown reasons.

- Use `print <var>` to show value of `var`. Most helpful for simple types like `char`, `double`, etc.
- Use `backtrace` to show the current stack frame. This can sometimes help determine where the error took place. Use `up` to go up the stack (towards outer calls, closer to main) and `down` to go down the stack (towards inner, nested calls).
- Use `list` to print the next few lines of source code. Use `list 1` to go to the beginning of the file, and `list -` to go backwards.

8. Use `quit` to quit GDB.

See https://beej.us/guide/bggdb/ and https://thispointer.com/gdb-debugger-tutorial-series/ for more detailed guides on basic GDB use.