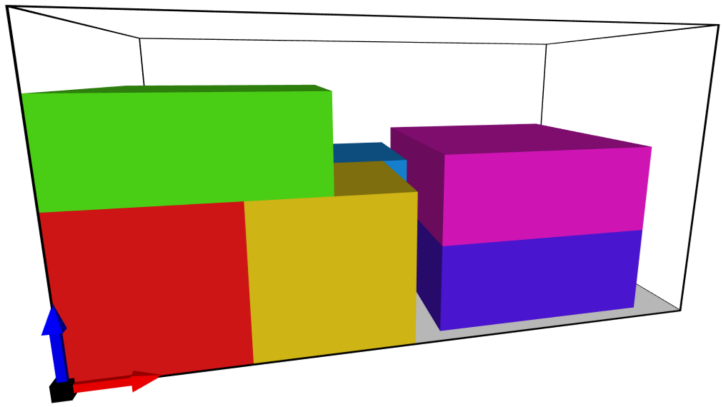
Tools for Loading Problems

April 1, 2019



1 Introduction

The goal of the ORTEC Scientific Benchmark platform[1] is to let people work on more practical combinatorial optimization problems. In order to allow you to get right to solving them, several `Python` tools have been developed to view and analyze instances and solutions. This document aims to give a short but complete description of the given tools, including examples of usage. The tools discussed in this document are:

- Conversion and Validity of Instance tool;
- Conversion and Validity of Solution tool;
- Applicable Constraints and Objectives tool;
- Solution Viewer tool.

All functionality of the tools can be used separately (such as loading and writing instance and solution data) and this document also serves as documentation for these functions.

2 Conversion and Validity of Instance Files

The instance data is available in JSON, XML, and YAML-format, and a conversion tool and validity checker are available for these formats. Conversion is done through an intermediary object known as `ThreeDinstance`, as seen in Figure 1. More on `ThreeDinstance` can be found in the accompanying document: "Instance Specification for Loading Problems". A short guide on how to use the tools is given in Section 2.1. Loading data will refer to the process of going from a JSON, XML, or YAML-file to a `ThreeDinstance` object, and is explained in Section 2.2. Writing data will refer to the converse, and is explained in Section 2.3. In Section 2.4 we explain how the instance file is checked for validity.

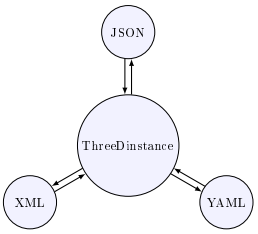


Figure 1: Conversion of data formats through `ThreeDinstance` object

2.1 Usage

The conversion functionality is accessible through the `osbl-instance` script, see Figure 2 for the command line manual. This manual page can be opened by supplying `-h` as command line argument to the script.

```
usage: osbl-instance [-h] --input INPUT_FILE [--type TYPE]
                     [--output OUTPUT_FILE] [--xml] [--yaml] [--json]
                     [--setname SETNAME] [--instancename INSTANCENAME]
                     [--reindex] [--remove_unused]

Convert and check loadbuilding instances

optional arguments:
  -h, --help            show this help message and exit
  --input INPUT_FILE, -I INPUT_FILE
                        The input file
  --type TYPE, -t TYPE  The type of the input file, choose one of: json, xml,
                        yaml
  --output OUTPUT_FILE, -O OUTPUT_FILE
                        The output file basename, extension is set by output
                        type
  --xml, -X             Create xml file
  --yaml, -Y            Create yaml file
  --json, -J            Create json file
  --setname SETNAME     Overwrite the set name
  --instancename INSTANCENAME
                        Overwrite the instance name
  --reindex, -R         Reindex ids to lowest possible
  --remove_unused, -U   Remove unused optional fields
```

Figure 2: Usage manual of the instance tool

Normal usage requires at least the input file to be specified, after this there are several options: if the input type is specified, it should be one of `json`, `xml`, or `yaml`, otherwise it will be deduced from the extension. If no output name is specified, it will be taken to be the set name followed by the instance name, separated by an underscore. More than one output type can be specified, or none at all. In

the latter case, the instance is only checked for validity. The set name and instance name arguments overwrite the respective fields. If the reindex flag is set, duplicate items will be grouped together, and objectives with unnecessarily high priorities will be reordered. If the remove unused flag is set, unused optional data fields will be discarded. As part of the conversion, defaults are set whenever one is available for data fields that are required for the given objectives and constraints.

2.2 Loading Data

There are three data loading methods available; one for each data format. They are implemented as three separate classes inheriting from the same base class `BaseToThreeDinstance`:

- `JSONtoThreeDinstance` for JSON files;
- `XMLtoThreeDinstance` for XML files;
- `YAMLtoThreeDinstance` for YAML files.

`BaseToThreeDinstance` provides a common interface to easily load files. Loading the data is done by instantiating an object of any of the three child classes, followed by calling its `CreateThreeDinstance` method. The constructor expects a file name as argument, which should be a string with the relative or absolute path of the data file. The `CreateThreeDinstance` method expects no arguments and returns a `ThreeDinstance`. An example that loads data from a JSON file is shown below:

```
from ortec.scientific.benchmarks.loadbuilding.instance.read.JSONtoThreeDinstance import JSONtoThreeDinstance
jsonToLB = JSONtoThreeDinstance("Instance_Example_001.json")
lbInstance = jsonToLB.CreateThreeDinstance()
```

2.3 Writing Data

Analogously to the data loading methods, there is a data writing method for each data format. Each is implemented as a separate class inheriting from the common base class `ThreeDinstanceToBase`:

- `ThreeDinstanceToJSON` for JSON files;
- `ThreeDinstanceToXML` for XML files;
- `ThreeDinstanceToYAML` for YAML files.

`ThreeDinstanceToBase` provides a common interface to easily write to files. Writing the data is done by instantiating an object of any of the three child classes, followed by calling its `WriteInstance` method. The constructor expects a `ThreeDinstance` as argument, and `WriteInstance` expects a file name as argument which should point to the relative or absolute path of the new file. The following Python snippet could follow the example code from section 2.2 to convert a JSON file to an XML file:

```
from ortec.scientific.benchmarks.loadbuilding.instance.write.ThreeDinstanceToXML import ThreeDinstanceToXML
lbToXML = ThreeDinstanceToXML(lbInstance)
lbToXML.WriteInstance("Instance_Example_001.xml")
```

2.4 Checking Data

Multiple aspects of a given `ThreeDinstance` can easily be checked for correctness using the methods mentioned in Figure 3. If multiple checks are desired the order of operations as seen in Figure 3 should be adhered to, to avoid unexpected behaviour and incorrect error reports. The checking of the required fields of the data is explained in section 2.4.1. The checking of optional fields of the data is explained in section 2.4.2. The reindexing step is explained in section 2.4.3. A short-hand for applying all checks exists as the `AllChecks` method. After loading the data all checks are performed, and if an error occurs, there is no conversion performed.

2.4.1 Required Fields

Certain data elements are always required, regardless of given constraints or objectives. The `IsValid` method checks whether all required fields are supplied, and whether their types are correct. A report is built on the errors that occurred. All components of the `ThreeDinstance` class also have their own `IsValid` method (which are invoked for `ThreeDinstance`'s call). In particular, this method will check whether:

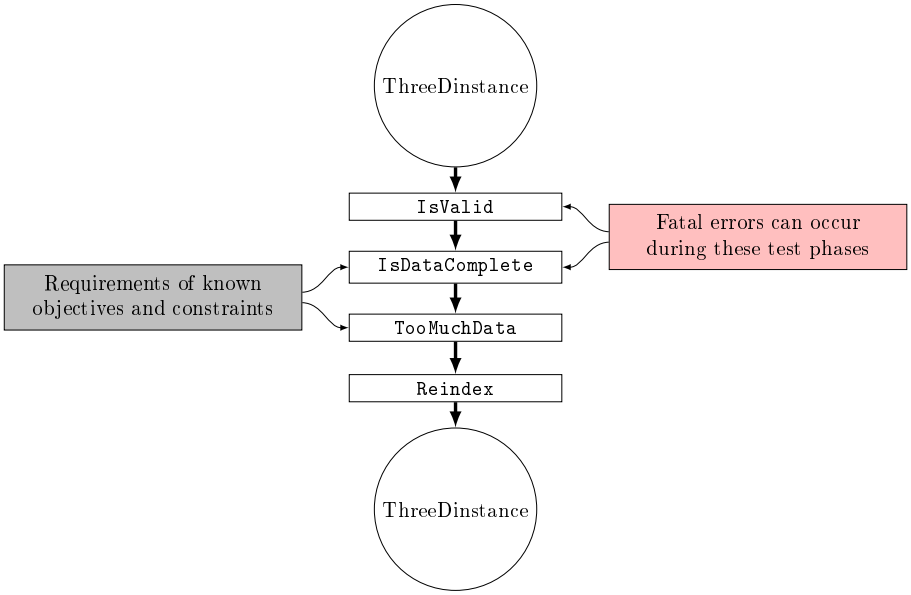


Figure 3: Complete checking of data files consists of four separate tests

- the description section is valid, which means:
  - it has a non-empty string set name;
  - it has a non-empty string instance name;
- each containerkind is valid, which means:
  - it has an integer id;
  - it has a non-negative integer quantity;
  - it has at least one loadingspace;
  - all of its loadingspaces are valid, which means:
    - they have integer ids;
    - their positions are lists of three non-negative integers;
    - their bounding boxes are lists of three positive integers;
  - there are no duplicate ids among its loadingspaces;
- each palletkind is valid, which means:
  - it has an integer id;
  - it has a non-negative integer quantity;
  - its bounding box is a list of three positive integers;
  - its position is a list of three non-negative integers;
  - it has exactly one valid loadingspace;
  - its orientations are a subset of {`lwH`, `WLh`, `Lwh`, `lWh`, `LWH`, `wLH`, `WlH`, `wlh`};
- each boxkind is valid, which means:
  - it has an integer id;
  - it has a non-negative integer quantity;
  - its bounding box is a list of three positive integers;
  - its position is a list of three non-negative integers;
  - it has exactly one valid loadingspace;
  - its orientations are a subset of {`HLW`, `LHw`, `hLw`, `lHW`, `hWL`, `Hlw`, `lwH`, `LhW`, `WLh`, `HwL`, `Lwh`, `lWh`, `whL`, `wHl`, `Whl`, `hwl`, `LWH`, `wLH`, `WlH`, `HWl`, `WHL`, `wlh`, `hlW`, `lhw`};
- each itemkind is valid, which means:
  - it has an integer id;
  - it has a non-negative integer quantity;
  - the bounding box is a list of three positive integers;

  - its orientations are a subset of {HLW, LHw, hLw, lHW, hWL, Hlw, lwH, LhW, WLh, HwL, Lwh, lWh, whL, wHl, Whl, hwl, LWH, wLH, WlH, HWl, WHL, wlh, hlW, lhw};
- the ids of the containerkinds, palletkinds, boxkinds, and itemkinds are unique with respect to other objects of their own type;
- each objective is valid, which means:
  - it has a string name equal to one of the defined objectives names;
  - it has a positive integer priority;
  - it has a positive float weight.
- each constraint is valid, which means:
  - it has a string name equal to one of the defined constraint names.
- constraint names do not appear more than once.

Note that objective names *are* allowed to appear more than once, for example when they occur with different priorities.

The `IsValid` method returns two values: a boolean value that equals `True` when there were no errors and `False` otherwise, and a string containing all the errors that occurred during testing. Errors are always fatal for this test. In the `AllChecks` method an assertion is used to ensure that upon normal termination this method did not fail.

2.4.2 Optional Fields

Certain data elements are only required when specific constraints and objectives are active for that instance (such as weights in presence of a maximum weight constraint). The constraints and objectives that can be used, along with the requirements they impose on the instance data, are defined in the `common/constraints` and `common/objectives` directories respectively. As soon as a class is constructed that inherits from either `BaseConstraint` or `BaseObjective` it is logged. The data checking aspect is two-fold for optional fields: all optional fields that are needed for constraints and objectives that are active for this instance must be supplied, and all fields that are not needed do not have to be supplied. The former of these cases is implemented by the `IsDataComplete` method, and the latter by the `TooMuchData` method.

In a `IsDataComplete` call we check whether all necessary optional fields are supplied. Whenever an optional field is missing while it is required by a constraint or objective two things can happen: the field is set to a default value (specified by the constraint or objective), or a fatal error occurs (when the default value is `None`). In both cases a warning/error will be stored. The method returns two values: a boolean value that equals `True` when there were no fatal errors and `False` otherwise, and a string containing all the errors/warnings that occurred during testing.

In a `TooMuchData` call, we check whether there were unnecessary optional fields supplied. Whenever an optional field is filled in despite not being required by a constraint or objective, a warning is given. The method never fails, and merely returns a string containing all the warnings.

2.4.3 Reindexing

The `Reindex` method will ensure that all ids of containers, pallets, boxes, items, and loading spaces are unique within their class for the given instance. It maintains the ordering of the original ids, but reindexes them to ensure they go through the positive integers without skipping numbers. The same is done for the priorities of the specified objectives.

3 Conversion and Validity of Solution Files

Similar to the instance file, the solution files can be in any of the JSON, XML and YAML-formats, and a conversion tool and validity checker are available for these three formats. See Figure 4 for the command line manual. Conversion is done through the intermediary object known as `ThreeDsolution`. More on this object can be found in the accompanying document: "Solution Specification for Loading Problems". A short guide on how to use the conversion tool is given in Section 3.1. Sections 3.2 and 3.3 respectively explain the reading and writing of solution files, and Section 3.4 explains the checking of solution files.

3.1 Usage

```
usage: osbl-solution [-h] --instance INPUT_FILE [--instancetype INSTANCE_TYPE]
                     --solution SOLUTION_FILE [--solutiontype SOLUTION_TYPE]
                     [--output OUTPUT_FILE] [--xml] [--yaml] [--json]
                     [--setname SETNAME] [--instancename INSTANCENAME]

Convert and check loadbuilding solutions

optional arguments:
  -h, --help            show this help message and exit
  --instance INPUT_FILE, -I INPUT_FILE
                        The instance file
  --instancetype INSTANCE_TYPE, -IT INSTANCE_TYPE
                        The type of the instance file, choose one of: json,
                        xml, yaml
  --solution SOLUTION_FILE, -S SOLUTION_FILE
                        The solution file
  --solutiontype SOLUTION_TYPE, -ST SOLUTION_TYPE
                        The type of the solution file, choose one of: json,
                        xml, yaml
  --output OUTPUT_FILE, -O OUTPUT_FILE
                        The output file basename, extension is set by output
                        type
  --xml, -X             Create xml file
  --yaml, -Y            Create yaml file
  --json, -J            Create json file
  --setname SETNAME     Overwrite the set name
  --instancename INSTANCENAME
                        Overwrite the instance name
```

Figure 4: Usage manual of the solution tool

Normal usage requires at least the instance and solution files to be specified, after this there are several options: if the input type either is specified, it should be one of `json`, `xml`, or `yaml`, otherwise it will be deduced from the extension. If no output name is specified, it will be taken to be the set name followed by the instance name, separated by an underscore. More than one output type can be specified, or none at all. In the latter case, the solution is only checked for validity.

3.2 Loading Data

There are three data loading methods available; one for each data format. They are implemented as three separate classes inheriting from the same base class `BaseToThreeSolution`:

- `JSONtoThreeDsolution` for JSON files;
- `XMLtoThreeDinstance` for XML files;
- `YAMLtoThreeDinstance` for YAML files.

The following Python snippet creates a `ThreeDsolution` object from a JSON file, given a `ThreeDinstance` object:

```
from ortec.scientific.benchmarks.loadbuilding.solution.read.JSONtoThreeDsolution import JSONtoThreeDsolution
jsonToSol = JSONtoThreeDsolution("Solution_Example_001.json")
lbSolution = jsonToSol.CreateThreeDsolution(lbInstance)
```

3.3 Writing Data

Analogously to the data loading methods, there is a data writing method for each data format. Each is implemented as a separate class inheriting from the common base class `ThreeDinstanceToBase`:

- `ThreeDinstanceToJSON` for JSON files;
- `ThreeDinstanceToXML` for XML files;
- `ThreeDinstanceToYAML` for YAML files.

The following Python snippet writes a `ThreeDsolution` to an XML file, and could follow the previous snippet to convert a solution file from JSON to XML:

```
from ortec.scientific.benchmarks.loadbuilding.solution.write.ThreeDsolutionToXML import ThreeDsolutionToXML
solToXML = ThreeDsolutionToXML(lbSolution)
solToXML.WriteSolution("Solution_Example_001.xml")
```

3.4 Checking Data

A `ThreeDsolution` object can be checked for validity by a simple call to `GetResults()` which returns a `ValidationResult` object. `ValidationResult` has two test functions `SolutionFormatIsValid` and `SolutionIsValid`, and it can return the objective with `GetObjective`. The quickest way is to call `PrintResults` and show the output from the validation.

4 Finding Applicable Constraints and Objectives

In the presence of certain constraints and objectives, specific optional fields may become required. An example is that containers must have a maximum weight, and items a weight, in the presence of the maximum weight constraint. Another example is that containers require a cost in the presence of the container costs objective. A tool is available, through the `osbl-find-constraints-objectives` script, that checks whether we have sufficient data to impose certain constraints or objectives on a given instance. It reports all default values that have been assumed and then the objectives and constraints that have all their requirements fulfilled.

In section 4.1 a short guide is presented on how to use the tool. The two types of requirements that constraints and objectives can impose are explained in section 4.2. Section 4.3 explains how to add new constrains and objectives to the framework.

4.1 Usage

The sufficiency check is available through the `osbl-find-constraints-objectives` script, see Figure 5 for the command line usage manual. This manual page can be opened by supplying `-h` as command line argument to the script.

```
usage: osbl-find-constraints-objectives [-h] --input INPUT_FILE [--type TYPE]

Report applicable constraints and objectives

optional arguments:
  -h, --help            show this help message and exit
  --input INPUT_FILE, -I INPUT_FILE
                        The input file
  --type TYPE, -t TYPE  The type of the input file
```

Figure 5: Usage manual of the constraints and objectives tool

An example of output of this script can be found in Figure 6.

```
The following objective(s) can be applied given the data from the file:
        - fill_rate
        - weight_distribution
        - worst_fill_rate
The following objective(s) can not be applied given the data from the file:
        - container_costs
The following constraint(s) can be applied given the data from the file:
        - maximum_weight
        - support
```

Figure 6: Example output of the constraints and objectives tool

4.2 Checking Data Requirements

Each constraint or objective can impose requirements on items, boxes, pallets, containers, and loading spaces. A requirement can in this case mean one of two things: an `ExistenceRequirement` or a `PropositionalRequirement`.

An `ExistenceRequirement` represents that some constraint or objective needs a certain optional data field to be specified. Its data members consists of a field name, a default value, and a type cast. The default value can be `None`, meaning no default is assumed. When the field name does not occur in the data file, or if it is left unspecified, and a default value is specified the default is used. If instead no default is specified, an error is thrown. After the data has been loaded, a type cast is attempted. If the type cast fails an error is also thrown.

A `PropositionalRequirement` imposes, as implied by the name, a proposition onto a data field. It consists of a field name, a proposition, and an error message. If the proposition does not hold, the error message is thrown.

4.3 Adding New Constraints and Objectives

The objective and constraint system is designed to be as modular as possible: objectives and constraints can be added in the `common/constraints` and `common/objectives` directories without having to edit any other code: new optional fields are automatically read from, written to, and checked for validity by all scripts. Every new constraint and objective class definition is automatically logged if it is flagged as being 'active', which is true by default for children of `BaseConstraint` and `BaseObjective`. The complete list of supported optional fields can be obtained by reading all field requirements of these constraints and objectives.

`BaseConstraint` and `BaseObjective` have a common parent: `BaseRequirement`. It provides an interface for denoting optional field requirements for instances and their solutions. A `BaseRequirement` object has five fields: a list of requirements for itemkinds, boxkinds, palletkinds, containerkinds, and loadingspaces. Where a requirement can be either an `ExistenceRequirement` or a `PropositionalRequirement`, as described in Section 4.2. If any of the `ExistenceRequirements` on the same data field are conflicting (either in default or in type cast) an error is raised. See Figure 7 for the class diagram.

Adding new constraints or objectives can be done quite easily by inheriting from `BaseConstraint` or `BaseObjective` respectively. Each new constraint needs to override the member function `Validate`, which, given a `ThreeDsolution` returns a boolean value and a string representing whether or not the constraint holds and a small report accumulated during processing. Each new objective analogously needs to override the member function `Evaluate`, which, given a `ThreeDsolution`, returns the value the solution has with respect to this objective. Both objectives and constraints need a string data member `name` denoting its unique name. Both can optionally contain a list of data requirements to impose on the instance or solution. The new class definitions should go in `common/constraints` and `common/objectives` respectively.

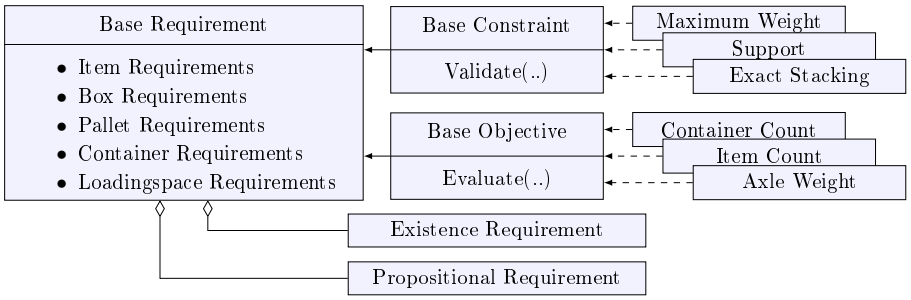


Figure 7: Class Diagram of Optional Requirements for Constraints and Objectives

5 Solution Viewer

Finally, a visualization tool is available for viewing solutions. When run, it will plot each loadingspace and its contents separately with appropriate labels. The vpython library is used to generate the plots. Once created, plots can be explored by rotation, zooming, and panning. Three different coloring options are available.

5.1 Usage

See Figure 8 for the helper page for this tool. The tool requires an instance and solution to work. If the instance and solution types are not specified, they will be deduced from the file extensions. Furthermore, three different color settings are available: file, correct, and distinct. If the first (default) setting is used, the color attributes will be read from the solution file, otherwise the tool will generate appropriate colors. The opacity setting sets the default opacity.

```
usage: osbl-solution-viewer [-h] --instance INPUT_FILE
                            [--instancetype INSTANCE_TYPE] --solution
                            SOLUTION_FILE [--solutiontype SOLUTION_TYPE]
                            [--color COLOR] [--opacity OPACITY]

Visualize loadbuilding solutions

optional arguments:
  -h, --help            show this help message and exit
  --instance INPUT_FILE, -I INPUT_FILE
                        The instance file
  --instancetype INSTANCE_TYPE, -IT INSTANCE_TYPE
                        The type of the instance file, choose one of: json,
                        xml, yaml
  --solution SOLUTION_FILE, -S SOLUTION_FILE
                        The solution file
  --solutiontype SOLUTION_TYPE, -ST SOLUTION_TYPE
                        The type of the solution file, choose one of: json,
                        xml, yaml
  --color COLOR, -C COLOR
                        The coloring option, choose one of: file, correct,
                        distinct
  --opacity OPACITY, -O OPACITY
                        The default opacity
```

Figure 8: Usage manual of the solution viewer tool

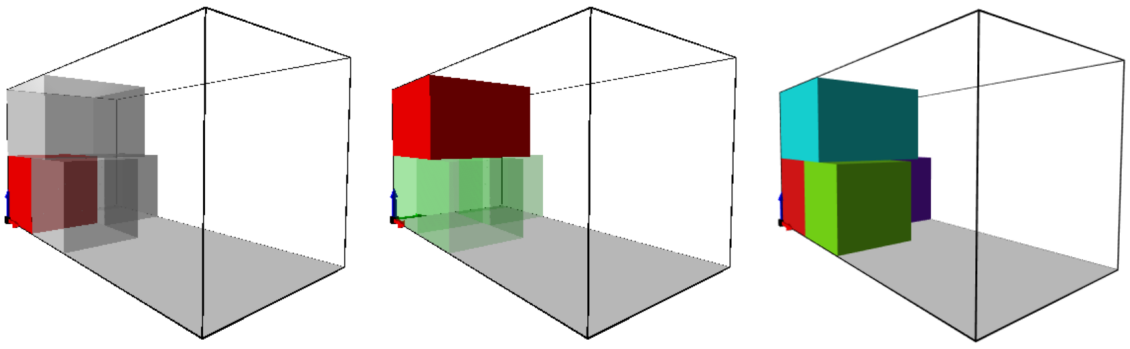


Figure 9: Left to right: colors read from solution file, colors by correctness of placement, and distinct colors

References

[1] ORTEC. (2018). ORTEC Scientific Benchmarks. http://benchmarks.ortec.com.