TRUST Baltik Project Tutorial V1.8.1beta

CEA Saclay

Support team: trust@cea.fr

June 2, 2020

Initialisation of TRUST environment

TRUST commands

- Source the TRUST environment:

  source /home/triou/env_TRUST_X.Y.Z.sh

- To know if the configuration is ok and where are the sources:
  ```
  $ echo $TRUST_ROOT
  ```

- Copy a TRUST test case:
  ```
  $ mkdir -p Formation_TRUST/yourname
  $ cd Formation_TRUST/yourname
  $ trust -copy upwind
  $ cd upwind
  ```

- Change "format lml" to "format lata" in the data file

Download & configure Eclipse (I)

Download Eclipse

- Go to the website of the Eclipse Foundation: http://www.eclipse.org/downloads/eclipse-packages/
- Click on **Eclipse Neon (4.6)** on the menu **More downloads**.
- Select **Eclipse IDE for C/C++ Developers** → **Linux 64-bits**
- Download the **eclipse-cpp-neon-3-linux-gtk-x86_64.tar.gz** package in your directory `Formation_TRUST/yourname`
- For OS older than CentOs7, Ubuntu16.04 and Fedora22, download **Eclipse Mars** version: **eclipse-cpp-mars-2-linux-gtk-x86_64.tar.gz**

Untar the downloaded Eclipse archive

```
$ cd Formation_TRUST/yourname
$ tar xfz eclipse-*.tar.gz
$ cd eclipse
```

Download & configure Eclipse (II)

For Ubuntu16.04, Fedora22, CentOs 7 and recents OS

Edit the *eclipse.ini* file by deleting the last 2 lines (Xms and Xmx) and adding the following lines:
Xms512m
Xmx2048m

For older OS

Edit the *eclipse.ini* file, by deleting the last 3 lines (MaxPermSize, Xms and Xmx) and adding the following ones:
Xmn256m
Xss2m
server
Xms512m
Xmx2048m

Create a TRUST platform project (I)

Initialize TRUST environnement

```
$ source /home/triou/env_TRUST_X.Y.Z.sh
$ echo $TRUST_ROOT/src
$ echo $exec_debug
```

Launch Eclipse

```
$ mkdir -p Formation_TRUST/yourname/workspace
$ cd Formation_TRUST/yourname/eclipse
$ ./eclipse &
```

- Workspace: Browse the directory `Formation_TRUST/yourname/workspace`
- Welcome : close x button

Create a TRUST platform project (II)

Create the project

- File → New → C++ Project
  ⇒ Project name: TRUST-X.Y.Z
  ⇒ Project type: "Executable" → "Empty Project"
  ⇒ Toolchains: "Linux GCC"
  ⇒ Finish

Import source files into the already created project

- From the "Project Explorer" tab, right click on TRUST-X-Y-Z → "Import..."
  ⇒ General → File System → Next
  ⇒ From directory: copy the string matching `$TRUST_ROOT/src/`
  ⇒ Check "Select All"
  ⇒ Into folder: TRUST-X.Y.Z
  ⇒ Finish
  ⇒ Wait to have 100% at the bottom right corner of the window (C/C++ indexer).

Create a TRUST platform project (III)

Configure the project and launch a computation

- From the "Project Explorer" tab, right click on TRUST-X.Y.Z → Properties
  ⇒ Builders: uncheck "CDT Builder" → OK → OK
- From the "Project Explorer" tab, right click on TRUST-X.Y.Z → "Debug As" → "Debug Configurations..."
  ⇒ Right click on "C/C++ Application" → New
  - In the "Main" tab:
    ⇒ Project: TRUST-X.Y.Z
    ⇒ "C/C++ Application": copy the string matching $exec_debug
    ⇒ "Apply"
  - In the "Arguments" tab:
    ⇒ "Program arguments" → specify the name of your datafile
    ⇒ "Working directory" → uncheck "Use default" and select the directory with path containing the datafile
    ⇒ "Apply"

  ⇒ "Debug"

Creation of a Baltik project

Baltik commands

- Create your directory for your project:
  ```
  $ cd Formation_TRUST/yourname
  ```
- Fill your project from a basic project template:
  ```
  $ trust -baltik my_project
  $ cd my_project
  $ ls -l
  ```
- You can see that you have now:
  - three directories: share, src and tests, and
  - one "project.cfg" file.
- Copy the following TRUST .cpp file into your baltik project:
  ```
  $ cd src
  $ mkdir TRUST_modif
  $ cp $TRUST_ROOT/src/MAIN/mon_main.cpp TRUST_modif
  $ cd ..
  ```

Create your git repository

Git commands

- We want to create a git repository to store and manage your developpments.
- Initialize an empty git repository:
  `$ git init`
- Watch your working tree status:
  `$ git status`
- You can see your file and the three directories on the "untracked" files section. It means that they are not followed by the git repository for the moment.
- To add all your directories and files to the git repository, you have to prepare a commit:
  `$ git add --all`
  `$ git status`
- Now you can send your commit to add your files to your git repository:
  `$ git commit -m "Initial commit"`

Create your git repository

Git commands

- Watch your working tree status:
  `$ git status`
- There is nothing more to add to your git repository.

Baltik commands

- Edit your project file "project.cfg" to specify name, author and executable.
- Then configure your project:
  `$ baltik_build_configure -execute`
- The previous command launches the "baltik_build_configure" script and the "configure" script directly.

Create your git repository

Git commands

- Check the status of your git repository with the "--ignored" option to see the status of all files:
  `$ git status --ignored`
- You can see that the file "project.cfg" has been modified. And that there are new untracked files. It means that they are not on the git repository.
- To see only the changes on the git repository files:
  `$ git status -uno`
- Track changes via gitk (GUI interface of Git):
  `$ gitk &`
- You can see information about your first commit and actual untracked changes.

Make a basic build

- To make a basic build:
  `$ cd Formation_TRUST/yourname/my_project`
- Configure your project:
  `$ ./configure`
- Build your project in different modes:
  - Build an optimized (-03 option) version:
    `$ make optim`
  - Build a debug (-g -O0 option with asserts) version:
    `$ make debug`
- Initialize your baltik project environment:
  `$ source env_basic.sh`
- Check the executables files:
  `$ ls $exec`
  `$ ls $exec_opt`
  `$ ls $exec_debug`

Other builds

- List other options available for the make command:
  ```
  $ make help
  ```
- Build an optimized binary for profiling (option -pg -O3):
  ```
  $ make prof
  $ ls $exec_pg
  ```
- Build an optimized binary for test coverage (option -gcov -O3):
  ```
  $ make gcov
  $ ls $exec_gcov
  ```
- Notice that TRUST optimized binary for profiling or a TRUST optimized binary for test coverage must exist to compile your own optimized or debug or profiling or coverage executable.

Create a basic BALTIK project without dependency (I)

Initialize baltik environnement

```
$ source env_baltik.sh
$ echo $project_directory/src
```

Launch Eclipse

```
$ cd Formation_TRUST/yourname/eclipse
$ ./eclipse &
```

Create the project

- File → New → "Makefile Project with Existing Code"
  ⇒ Project name: MY_BALTIK
  ⇒ Existing Code Location: copy string matching `$project_directory/src`
  ⇒ Toolchain for Indexer Settings: "Linux GCC"
  ⇒ Finish
  ⇒ Wait to have 100% at the bottom right corner of the window (C/C++ indexer).

Create a basic BALTIK project without dependency (II)

Configure the BALTIK project and link it with TRUST

- From the "Project Explorer" tab, right click on MY_BALTIK → Properties
  ⇒ Builders: check "CDT Builder"
  ⇒ C/C++ Build :
  - Builder Settings: Build directory: `${workspace_loc:/MY_BALTIK}/../` or copy the string matching `$project_directory/`
  - Behavior: check "Build (Incremental build)": debug optim (instead of all)

  ⇒ Project References: check TRUST-X.Y.Z → OK

Build the BALTIK project

From the "Project Explorer" tab, right click MY_BALTIK → Index → Rebuild
⇒ Wait to have 100% at the bottom right corner of the window (C/C++ indexer).
Right click MY_BALTIK → Build Project

Create a basic BALTIK project without dependency (III)

Launch a computation

- From the "Project Explorer" tab, right click MY_BALTIK → "Debug As" → "Debug Configurations..."
  ⇒ C/C++ Application → New
  - In the "Main" tab:
    ⇒ Project: MY_BALTIK
    ⇒ C/C++ Application: `${workspace_loc:/MY_BALTIK}/../basic` or copy the string matching `$project_directory/basic`
    ⇒ "Apply"
  - In the "Arguments" tab:
    ⇒ Program arguments → specify the name of your datafile
    ⇒ Working directory → uncheck "Use default" and select the directory containing the datafile
    ⇒ "Apply"

  ⇒ Debug

Useful shortcuts in sources

Shortcuts

- Open a cpp file from Project Explorer tab:
  Double click on TRUST-X.Y.Z → Kernel → Framework → Probleme_base.cpp
- In the cpp file: Right click on method "initialize()"
  ⇒ F3: Opens Declaration
  ⇒ F4: Open Type Hierarchy
  ⇒ Ctrl+Alt+H: Open Call Hierarchy
  ⇒ "Alt+→" and "Alt+←": Move from a tab to another

Creation of a new cpp class

Baltik commands

- Create a new repository for your own classes:
  ```
  $ mkdir -p $project_directory/src/my_module
  $ cd $project_directory/src/my_module
  ```
- Create your first class "my_first_class" with template:
  ```
  $ baltik_gen_class my_first_class
  ```

Git commands

- Check the status of your repository:
  ```
  $ git status .
  ```
- Add your new class to your git repository to follow your modifications:
  ```
  $ git add my_first_class.*
  $ git commit -m "Add my_first_class src"
  ```

Creation of a new cpp class

Baltik commands

- Have a look at the 2 files my_first_class.h|cpp.
- Each time a source file is added to the project, you need to configure it:
  ```
  $ cd $project_directory
  $ ./configure
  ```
- Build your project with Eclipse or in the terminal.
- Edit the 2 files with vim|nedit|gedit|emacs.

Eclipse

- Edit the 2 files with Eclipse.
- For Eclipse use, you have to update your project to see your new files:
  → "Index/Rebuild" from "my_project" of "Project Explorer"
  → Click on "▶" button of "my_project" in the "Project Explorer"

Creation of a new cpp class

Baltik commands

- We want to change the inheritance of the class in order that it inherits from "Interprete_geometrique_base" class instead of "Objet_U".
- "Interprete_geometrique_base" class is the base class of all the keywords doing tasks on domains (eg: Mailler, Lire_fichier,...).
- You will:
  - add an "#include <Interprete_geometrique_base.h>" in my_first_class.h,
  - switch "Objet_U" to "Interprete_geometrique_base" in the .h and .cpp files,
  - rebuild your application.
  - An error will occur!
- You will have an error indicating a pure virtual function ("interpreter_") should be implemented.
- Look at the "Interprete_geometrique_base" class:
  - Eclipse: highlight the string "Interprete_geometrique_base" and push the F3 button of your keyboard to open the declaration file of this class
  - Or with the HTLM documentation: open the declaration file of the "Interprete_geometrique_base" class

Creation of a new cpp class

Baltik commands

- Notice the "interpreter()" method (which calls the "interpreter_()" method).
- This method is called each time a keyword is read in the data file (eg: "Read_file dom dom.geom", "Solve pb",...).
- Define the public method "interpreter_(Entree&)" in the include file and implement it (just print a message with "Cerr" like "- My first keyword!") into the cpp file.
- "Entree" is a TRUST class to read an input stream (from a file for example): "virtual Entree& interpreter_(Entree&);"
- Rebuild your project and fix your files until the binary of your project is built (named basic if you have not changed the name in the project.cfg file)
- Now we want to test our new class.
- Modify a test case into the build directory of your Baltik project:

```
$ cd $project_directory/build/
$ trust -copy Cx
ERROR...
```

Creation of a new cpp class

Baltik commands

- An error occurs because this test case is not in our baltik but in TRUST project so we have to launch the full environment (TRUST+our baltik).
  ```
  $ source ../full_env_basic.sh
  $ trust -copy Cx
  $ cd Cx
  ```
- Open the data file "Cx.data":
  ```
  $ vim|nedit|gedit|emacs Cx.data
  ```
- Just after the line where the problem is discretized, add the keywords "my_first_class" and "End".
  NB: Instead of "End", you can reduce the number of time step to only 1.
- Run your binary to check that this new keyword is recognized:

Creation of a new cpp class

With Eclipse:

→ In the project explorer, right click on "MY_BALTIK" and select "Debug As/Debug configurations..."
→ In "Main" tab, check "Disable auto build" then click on "Apply"
→ In "Arguments" tab, fill "Program arguments:" with "Cx"
→ "Working directory:" $workspace_loc:my_project/../Cx/ or "Formation_TRUST/yourname/my_project/build/Cx"
→ "Apply" and "Debug"
→ Click on "Yes" to change the kind of view
→ Click on "Resume" button to run the calculation until the end

On a terminal:

```
$ cd $project_directory/build/Cx/
$ exec=$exec_debug trust Cx
```

Nota bene: "Interprete_geometrique_base::interpreter_()" method is called first, which calls then the "my_first_class::interpreter_()" method.

Part 1

- The use of the class **Param** is recommended to read in the data file:

```
#include <Param.h>
Entree& A::readOn(Entree& is)
{
Nom opt;
int dim;
Cerr << "Reading parameters of A from a stream (cin or file)" << finl;
Param param(que_suis_je());
// Register parameters to be read:
param.ajouter("option",&opt);
param.ajouter("dimension",&dim,Param::REQUIRED);
// Mandatory parameter
// Read now the parameters from the stream is and produces an error
// if unknown keyword is read or if braces are not found at the
// beginning and the end:
param.lire_avec_accolades_depuis(is);
...
return is;
}
```

Modify your cpp class

Part 1

- To call it in the data file, you have to use the following syntaxe as the read of the parameters is done by the readOn method:

```
A a
Read a { dimension 3 option fast }
```

- In our case, the read of the parameters will be done by the interpreter() method so the syntax in the data file will be the following:

```
my_first_class { domaine dom option 0 }
# dom is the domain name #
```

- Add into the "interpreter_(Entree&)" method the read of these parameters into braces using the **Param** object.
- **Param** use is the recommended choice in this case (even though many current TRUST classes are still using the old fashion to read parameters), because it greatly simplifies the coding.

Modify your cpp class

Part 1

- Add "#include <Param.h>" into the cpp file.
- If help needed, have a look at the "Interprete_geometrique_base" sub-class "Extruder". The data file syntax is :

```
Extruder { domaine DomainName nb_tranches N direction X Y Z }
```

- Now we want to obtain the problem object using his name. You can have a look at the following method:
  Interprete_geometrique_base::associer_domaine (Nom & nom_dom)
- Look the HTML documentation. What is the task of this method?
- Once implementation is finished, add a check at the end of the method "interpreter_(Entree&)" and find how to print the domain name:

```
Cerr << "Option number " << option_number << " has been
read on the domain named " << ??? << finl;
```

Modify your cpp class: Part 1

With Eclipse:

- Build/fix/re-build the test case:
  → "Project" and "Build project"
- Run the test case:
  → "Run" and "Debug"

Or in a terminal:

- Build/fix/re-build the test case:

```
$ cd $project_directory
$ make debug
```

- Run the test case:

```
$ cd $project_directory/build/Cx/
$ export exec=$exec_debug
$ trust Cx
```

- In this case, TRUST runs with exec_debug.

Modify your cpp class

Part 2

- We are going to try to print information of the domain boundaries in our current project.
- Edit the "my_first_class.cpp" file and add into the "interpreter_()" method a loop on the boundaries.
- Look for help inside the "Domaine", "Zone", "Bord", "Frontiere" classes into the HTML documentation to access to the:
  - Number of boundaries (**nb_bords()** method)
  - Boundaries (**bord(int)** method)
  - Name of the boundaries (**le_nom()** method)
  - Number of faces of each boundary (**nb_faces()** method)
- You will print the infos with something like:

```
Cerr << "The boundary named " << ??? << " has " << ??? << "
faces." << finl;
```

- Now, we are going to try to calculate the sum of the VEF control volumes on the domain in our project.

Modify your cpp class

Part 2

- The information is in the "Zone_VF" class (a "Zone_dis" discretized zone) which can't be accessed from the domain, only from the problem.
- So we need to read another parameter in our data file:

```
my_first_class { domaine dom option 0 problem pb }
```

- Add the read of a new parameter problem (see "Extraire_plan::interpreter_(Entree&)" method for instance) into the "my_first_class.cpp" file.
- Then, remember the "equation" or "problem" UML diagram of the presentation's slides.
- Look for help inside the "Zone_VF", "Probleme_base" and "Equation_base" into the HTML documentation to access to the:
  - equation (**equation(int)** method)
  - discretized zone (**zone_dis()** method)
  - control volumes (**volumes_entrelaces()** method)

Modify your cpp class

Part 2

- You will need to cast the discretized zone returned by the **zone_dis()** method into a "Zone_VF" object.
- You will print the size of the control volumes array with something like:

```
Cerr << control_volumes.size() << finl;
```

- Where control_volumes is a **DoubleVect** returned by the **Zone_VF::volumes_entrelaces()** method.
- If you look at the "Problem" UML diagram of the presentation's slides, you will notice a better path to access to the discretized zone.
- What is this path ?
- Now, compute and print the sum of the control volumes with a "for" loop.

Add XData tags

- We want now to add XData tag to create the automated documentation of your new code.
- First we have to create this documentation for the first time.
  ```
  $ cd $project_directory
  $ make gui
  ```
- Open the documentation file:
  ```
  $ evince $project_directory/build/xdata/XTriou/doc.pdf &
  ```
- Now we will add comments in our cpp files to add information in the documentation.
- For this open the help of the TRAD_2 syntaxe:
  ```
  $ vim|nedit|gedit|emacs doc_TRAD_2 &
  ```

Add XData tags

- Add a first tag (in comments) into your cpp file just after the openning brace of the 'interpreter_()' method:

```
// XD english_class_name base_class_name TRUST_class_name
mode description
```

- The "english_class_name" and "TRUST_class_name" can be "my_first_class".
- The "base_class_name" is the name of the section in which will appear the information of your new class in the 'doc.pdf' file.
- The "mode" is to choose with the help of the doc_TRAD_2 file. Here we use "-3".

Add XData tags

- Then add at the end of the lines of type "param.ajouter...", an XD comment like:

  ```
  param.ajouter(...); // XD_ADD_P type description
  ```

- "type" can be (cf 'doc_TRAD_2' file): 'int', 'floattant', 'chaine', 'rien'...
- Compile the documentation:
  `$ make gui`
- Check that the documentation of your new class is in the new doc:
  `$ evince $project_directory/build/xdata/XTriou/doc.pdf &`
- To check that the GUI is validated:
  `$ make check_gui`
- Notice that you must have XD commands in all your cpp classes.

Adding prints

- Edit the "my_project/src/TRUST_modif/mon_main.cpp" file in your baltik project of Eclipse.
- Add this lines after "Process::imprimer_ram_totale(1);" :
  std::cout << "Hello World to cout." << std::endl;
  std::cerr << "Hello World to cerr." << std::endl;
  Cout << "Hello World to Cout." << finl;
  Cerr << "Hello World to Cerr." << finl;
  Process::Journal() << "Hello World to Journal." << finl;

With Eclipse: Rebuild the code

→ "Build project" from "my_project" of "Project Explorer"

Or in a terminal: Rebuild the code

```
$ cd $project_directory
$ make debug optim
```

Adding prints

- Create an empty data file:
  ```
  $ mkdir -p $project_directory/build/hello
  $ cd $project_directory/build/hello
  $ touch hello.data
  ```
- Run the code sequentially:
  ```
  $ trust hello
  ```
- Run the code in parallel and see the differences:
  ```
  $ trust hello 4
  ```
- "Cout" is equivalent to "std::cout" on the master process only. Use this output for infos about the physics (convergence, fluxes,...).
- "Cerr" is equivalent to "std::cerr" on the master process only. Use this output for warning/errors only.
- "finl" is equivalent to "std::endl" + "flush()" on the master process.

Modify your cpp class

- "Journal()" prints to "datafile_000n.log" files. Use this output during parallel development to print plumbing infos which would be hidden during later production runs.
- During run, this output can be unactivated with:
  ```
  $ ls *.log
  $ trust -clean && trust hello 4 -journal=0
  $ ls *.log
  ```
- Now, we will print the sum of the control volumes of the test case Cx into a file.
  ```
  $ cd $project_directory/build/Cx
  ```
- We want to write in a file whose name is something like: DataFileName_result.txt, where "DataFileName" is the name of the data file (eg: Cx).
- For that, you will create an object of the class **Nom** and fill it by collecting the name of the data file with **Objet_U::nom_du_cas()** method.

Modify your cpp class

- Then complete the name of the file with the string "_result.txt" thanks to the "operator+=" method of the class **Nom**.
- Then you will create the file with the **SFichier** class and print the sum into this file.

With Eclipse: Run the test case

→ "Run" and "Debug"

Or in a terminal: Run the test case

```
$ cd $project_directory/build/Cx/
$ exec=$exec_debug trust Cx
```

- Then open the "Cx_result.txt" file.

Parallel exercise

Part 1

- Run your test case Cx in parallel mode:
```
$ cd $project_directory/build/Cx/
$ trust -partition Cx 2 # Partition in 2 subdomains
$ trust PAR_Cx 2 # 2 processes used
```
- Compare the files: Cx_result.txt, PAR_Cx_result.txt.
- Differences come from the fact that the 2 processors write into the file one after the other one. So the final content will be the value calculated on the last processor which will acces to the file.
- You can try to launch one more time the calculation, the result may differ.
- To have the entire sum, you can apply the **mp_sum()** method on the sum obtained and add the print in the .txt file.
- Compare it to the sum obtained in the sequential run.
- It is better but we counted several times faces that belongs to the joint and to the virtual zones.

Parallel exercise

Part 1

- To parallelize the algorithm, rewrite it with the help of the **mp_somme_vect(DoubleVect&)** method.
- Add this print in the .txt file.
- You should find the same value for the sequential and parallel calculation.

Part 2 Optional

- Create a "verifie" script to check the resulting value (sequential then parallel).
- Add a call to "compare_sonde" in your "verifie" script...

Part 3

- To validate parallelization in TRUST, you can use the command "compare_lata":

```
$ ls *lata
$ compare_lata Cx.lata PAR_Cx.lata
```

Parallel exercise

Part 3

- You can see that there is no differences and the maximal relative error encountered is about 4.e-12.
- Performances `$ ls *TU`
  `$ meld Cx.TU PAR_Cx.TU &`
  `$ meld Cx_detail.TU PAR_Cx_detail.TU &`

Part 4 Debog

- Copy a debog test case:
  `$ cd $project_directory/build`
  `$ trust -copy Debog_VEF`
  `$ cd Debog_VEF`
- Open the Debog_VEF.data file and search the "Debog" command.
- Sequential run:
  `$ trust Debog_VEF`
- You get "seq" and "faces" files.

Parallel exercise

Part 4 Debog

- Partitionning step and creation of the parallel data file:
  `$ trust -partition Debog_VEF 2`
- Verify the parallel data file, you must have now "Debog pb seq faces 1.e-6 **1**".
- Run in parallel:
  `$ trust PAR_Debog_VEF 2`
- You get debog*.log and DEBOG files.
- If a value of an array differs between the two calculations and the difference is greater than 1.e-6 then "ERROR" message appears in the log files else we will get "OK" (cf debog.log).
- Add a debog instruction in your file mon_main.cpp located in $project_directory/TRUST_modif, after the "Hello world" prints put:
  ```
  double var = 2.5;
  Debog::verifier("- Debog test message",var);
  ```
- Don't forget to add the "#include <Debog.h>"!

Parallel exercise

Part 4 Debog

- Then compile and do the sequential run.
- You can see a first message.
- Then do the parallel run and check the debog.log file.
- Becarefull the debog instruction in the data file must be between the "Discretize" and "Read pb" lines.
- For more information:
  `$ trust -doc &`
  → Open the TRUST Generic Guide
  → Click onto the TRUST Reference Manual
  → Search for "Debog" keyword.

New share/Validation/Rapports_automatiques

Baltik commands

- Create a new prm file:
  ```
  $ cd Formation_TRUST/yourname/upwind
  $ trust -prm upwind
  ```
- Now you have a upwind.prm file.
- You have to add this prm validation file in your baltik:
  ```
  $ cd $project_directory
  $ cd share/Validation/Rapports_automatiques
  ```
- Create a new repository for your new prm validation form:
  ```
  $ mkdir -p upwind/src
  ```
- Add the needed files (data file, mesh & .prm file):
  ```
  $ cp Formation_TRUST/yourname/upwind/upwind.data upwind/src
  $ cp Formation_TRUST/yourname/upwind/upwind.geo upwind/src
  $ cp Formation_TRUST/yourname/upwind/upwind.prm upwind/src
  ```

New share/Validation/Rapports_automatiques

Git commands

- Add it to your git repository:
  ```
  $ git add upwind
  $ git commit -m "New prm"
  ```

Baltik commands

- Run this prm:
  ```
  $ cd upwind/
  $ Run_fiche
  ```
- Open the pdf report:
  ```
  $ evince build/rapport.pdf &
  ```

New tests/Reference/Validation

Baltik commands

- Create automatically the non-regression test case:
  ```
  $ cd $project_directory
  $ make check_optim
  ```
  Creation of upwind_jdd1
  Creation of upwind_jdd1/lien_fiche_validation
  Extracting test case (upwind.data) ...End.
  Creation of the file upwind_jdd1.lml.gz
  ...
- → You can see in the report table that PAR_upwind_jdd1 has crashed: "CORE" message.

New tests/Reference/Validation

Git commands

- Lets check the git status before solving this problem:
  ```
  $ git status -uno
  ```
- A new test case based on your PRM file has been created in the directory:
  $project_directory/tests/Reference/Validation/upwind_jdd1

Baltik commands

- Now we want to correct the error, so copy the test case:
  ```
  $ cd $project_directory/build
  $ trust -copy upwind_jdd1
  ```
  ERROR...
- We have to re-run the configure script to take into account the new test case:
  ```
  $ cd $project_directory
  $ ./configure
  $ cd build
  $ trust -copy upwind_jdd1
  ```

New tests/Reference/Validation

Baltik commands

- Now we will analyse the error:
  ```
  $ cd upwind_jdd1
  $ trust -partition upwind_jdd1
  $ trust PAR_upwind_jdd1 2
  ```
- Correct the data file PAR_upwind_jdd1.data and re-run it.
- If it's ok, update the data file in $project_directory/share/Validation/Rapports_automatiques/upwind/src ("Scatter ../upwind/DOM.Zones dom" → "Scatter DOM.Zones dom")
- To Relaunch the last test cases which do not run:
  ```
  $ cd $project_directory
  $ make check_last_pb_optim
  ```
  Changement du jeu de donnees...
  suite a une modification d'un jeu de donnees de la fiche de validation associee.
  ...
  Successful tests cases :1/1

New tests/Reference/Validation

Git commands

- Add this non-regression test in configuration:
  ```
  $ git status -uno
  $ git add
  tests/Reference/Validation/upwind_jdd1/upwind_jdd1.data
  ```
- Push the modifications on your git repository:
  ```
  $ git commit -m "New reference test"
  $ git log
  ```

New tests/Reference/Validation

Baltik commands

- To run all the non regression tests with a optimized binary:
  ```
  $ make check_all_optim
  ```
- To run all the non regression tests with a debug binary:
  ```
  $ make check_all_debug
  ```
- To create an archive to share your work:
  ```
  $ make distrib
  $ ls
  ```
- You have now an archive in tar.gz format of your baltik project.

Code coverage exercise

- Browse the TRUST ressources index file:
  `$ trust -index`
- Select the Test coverage link.
- Which is the less covered matrix class ?
- We want to run test cases using rational Runge-Kutta scheme of ordre 2.
  - For this go to the Doxygen documentation of RRK2 class to see the methods of this class.
  - Use the "trust -check function|class|class::method" command to find and launch tests cases.
  - For example:
    `$ trust -check RRK2::RRK2`

GDB exercise

With Eclipse:

Run a test case with GDB:
→ "Debug As" and "Debug configurations..." from "my_project"
→ in "Arguments", "Program arguments:" upwind
→ "Working directory:" Formation_TRUST/yourname/upwind/
→ "Apply" and "Debug"
For more information about GDB commands, refer to the help menu.

Or in a terminal:

- Run a test case with GDB:
  ```
  $ cd Formation_TRUST/yourname/upwind/
  $ exec=$exec_debug trust -gdb upwind
  ```
- You are now in GDB.
- Add a breakpoint and stop into the SSOR preconditionner:
  ```
  (gdb) break SSOR::ssor
  ```

GDB exercise

- Run the test case:
  `(gdb) run upwind`
- Have a look at the stack
  `(gdb) where`
- Go to the next instruction:
  `(gdb) n`
- Print an array:
  `(gdb) print tab1`
- Or print matrice.tab1_ if "optimized out" message printed:
  `(gdb) print tab1[10]`
- Print only a value of an array:

```
(gdb) dumpint tab1 # Dump the array
(gdb) print tab1.size_array() # Array size
(gdb) up
(gdb) list 100
```

GDB exercise

- Print lines after the 100th line:

```
(gdb) print matrice
(gdb) print matrice.que_suis_je() # Kind of matrix ?
(gdb) print matrice.que_suis_je().nom_ # Kind of matrix ?
(gdb) up 5 # Move up 5 levels
(gdb) list 900
```

- Print others variables:

```
(gdb) # Pressure field
(gdb) print la_pression.que_suis_je().nom_
(gdb) # Pressure values (DoubleTab)
(gdb) print la_pression.valeurs()
(gdb) # DoubleTab dimension
(gdb) print la_pression.valeurs().nb_dim()
(gdb) # Dump the field values
(gdb) dumptab la_pression.valeurs()
```

Use Valgrind to find memory bugs

- Run a test case with Valgrind:
```
$ cd $project_directory
$ source env_basic.sh
$ cd build/Cx/
$ VALGRIND=1 trust Cx
```
- The Valgrind messages appear on the screen with the beginning of each line the same number. For example:
```
$ ==26645== ...
```
- The last line indicates if errors have occurred. An example with 0 error:
```
$ ==26645== ERROR SUMMARY: 0 errors from 0 contexts
(suppressed:  0 from 0)
```
- Now we will modify the sources in your baltik project to generate a Valgrind error on the Cx test case.

Use Valgrind to find memory bugs

- Edit the "my_first_class.cpp" file and remove the initialization of the sum to calcule the total of control volumes.
  In place of "double sum=0;", put only "double sum;".
- Rebuild your project and run the test case:
  ```
  $ cd $project_directory
  $ make debug optim
  $ cd build/Cx/
  ```
  - in mode optim:
    ```
    $ exec=$exec_opt trust Cx
    ```
    In this case, no error appears.
  - in mode debug:
    ```
    $ exec=$exec_debug trust Cx
    ```
    In this case also, no error appears.
  - in mode valgrind:
    ```
    $ VALGRIND=1 exec=$exec_opt trust Cx
    ```
    On the other hand, in this case, there are errors.
    ```
    $ ==7517== ERROR SUMMARY: 187 errors from 109 contexts
    ```

1. TRUST initialization
2. Eclipse initialization
3. Baltik initialization
   - Creation of a Baltik project
   - Creation of your git repository
   - Builds
   - Using Eclipse
4. Modify the cpp sources
   - Create a new cpp class
   - Modify your cpp class
   - Add XData tags
   - Adding prints
5. Parallel exercise
6. PRM file and validation test case
   - "trust -prm"
   - Validation test case
7. Code coverage exercise
8. Tools
   - GDB exercise
   - Use Valgrind to find memory bugs

For more

- You can find the commented solution of the exercise:
  ```
  $ cd $TRUST_ROOT/doc/TRUST/exercices/my_first_class
  ```
- You can practice on a tutorial:
  ```
  $ cd $TRUST_ROOT/doc/TRUST/exercices/
  $ evince equation_convection_diffusion/rapport.pdf &
  ```