

Mechanic's user guide

Mariusz Slonina

April 6, 2010

Contents

1	Introduction	2
2	Installation	3
3	Getting started	3
3.1	The Hello module	5
3.2	The setup system	6
3.3	Pixel-coordinate system	9
3.4	Modes	9
3.5	Data storage scheme	9
3.6	Checkpoints	10
4	Advanced topics	10
4.1	Case studies	10
4.2	The template system	10
4.3	The Echo module	11
4.4	The Mandelbrot module	13
4.5	The Method 6	15
4.6	Working with data	16
5	Short Developers guide	16
5.1	Coding style	16
5.2	Message interface	16
6	Troubleshooting	17
6.1	Known bugs and missing features	17
6.2	Error codes	17

1 Introduction

Handling numerical simulations is not a trivial task, either in one- or multi-cpu environments. It can be a very stressful job, especially when you deal with many sets of initial conditions (like in many dynamical problems) and is full of human-based mistakes.

Our main research is focused on studying dynamics of planetary systems, thus it requires numerical job to be carefully done. Most of the problems can be coded by hand, in fact, we did it in that way many times, however you can easily find that most of these task are in some way, or in some part, repeatable.

Let's have an example: We want to study the dynamics of a four body problem – we have a star, two massive planets and a big gap between them. We want to know the dynamical behaviour (stability or not) of a test earth-like body in the gap. We can observe the behaviour by using some values of semimajor axis and eccentricity of the small planet and check the state of the system after some time. Then, we can change these values by a small delta and observe the state of the system again. If we repeat it in some range of semimajor axes and in some range of eccentricities we will get a dynamical map of the planetary system (one can exchange semimajor axis and eccentricity by other orbital elements, too). Each pixel of the map is a standalone numerical simulation which takes some time on one cpu.

Now, the first approach is to use one cpu to do all the stuff. However, if computation of the pixel lasts too long (especially when the configuration is quite stable), the creation of the dynamical map is a very long process, and can take not one or two weeks, but one or two months.

There is a second approach. Let's say, we have not one, but 10 cpus. If we can handle sending initial conditions and receiving results, we can create the dynamical map of the system at least 10 times faster!

And that's the reason we created **Mechanic**. We needed some kind of a numerical interface or framework that will handle our dynamical studies. We started by creating simple MPI Task farm model, however we quickly realised that using MPI framework can be useful not only in image-based operations (dynamical map is a some kind of an image), but also in many numerical problems with huge sets of initial conditions, or even tasks like observations reductions, which lasts too long on single cpu. We found that our interface should handle such situations, too.

Now, **Mechanic** is a multi-purpose numerical framework and interface. It is written in C99 with help of MPI and HDF5 storage. It provides extensible user API and loadable module support – each numerical problem can be coded as a standalone module, loaded dynamically during runtime. **Mechanic** uses **LibReadConfig** (LRC) for handling configuration aspects and **Popt** library for command line (CLI) options.

Mechanic is in pre-alpha stage, this means, that there are some bugs in code, some parts are not finished, and some features are not implemented yet. However, we try to keep the **Master** branch as stable and useful as possible. Feel free to participate in the development, test the software and send bugs. The latest snapshot can be grabbed from <http://git.astri.umk.pl>. The **Experimental** branch contains all bleeding-edge stuff.

Mechanic is distributed under terms of BSD license. This means you can use our software both for personal and commercial stuff. We released the code to the public, because we believe, that the science and its tools should be open for everyone. If you find **Mechanic** useful for your research, we will be appreciated if you refer to this user guide and our project homepage: <http://mechanics.astri.umk.pl/project/mechanic>.

We assume you have some basic knowledge on C-programming and using Unix-shell.

2 Installation

Mechanic uses **Waf** build system, see <http://code.google.com/p/waf> for details. **Waf** is build in Python, you should have at least Python 2.3 installed on your system.

To download the latest snapshot of **Mechanic** try

```
http://git.astri.umk.pl/?p=Mechanic.git
```

We try to keep as less requirements as possible to use **Mechanic**. To compile our software you need at least:

- MPI2 implementation (we prefer **OpenMPI**, and **Mechanic** was tested with it)
- HDF5, at least 1.8
- **LibReadConfig** with HDF5 support – LRC can be downloaded from our git repository, since it is a helper tool builded especially for **Mechanic**, but can be used independly. You need to compile it with `--enable-hdf` flag
- **Popt** library (should be already installed on your system)
- C compiler (`gcc 4.3` should do the job)

Compilation is similar to standard **Autotools** path:

```
./waf configure
./waf build
./waf install
```

The default installation path is set to `/usr/local`, but you can change it with `--prefix` flag.

Althought **Mechanic** requires MPI, it can be runned in a single-cpu environments (we call it "fake-MPI"). **Mechanic** should do its job both on 32 and 64-bits architectures with *nix-like system on board.

3 Getting started

To understand what **Mechanic** is and what it does, let us write a well-known "Hello World!". We will create small C library, let us call it **Hello** and save it in `mechanic_module_hello.c` file:

```
#include "mechanic.h"
#include "mechanic_module_hello.h"

int hello_init(moduleInfo* md){

    md->name = "hello";
    md->author = "Mariusz Slonina";
    md->date = "2010";
    md->version = "1.0";
    md->mrl = 3;

    return 0;
}
```

```

int hello_cleanup(moduleInfo* md){

    return 0;
}

int hello_pixelCompute(int node, moduleInfo* md, configData* d, masterData* r)
{

    r->res[0] = (double) r->coords[0];
    r->res[1] = (double) r->coords[1];
    r->res[2] = (double) r->coords[2];

    return 0;
}

int hello_slave_out(int nodes, int node, moduleInfo* md, configData* d,
    masterData* r){

    mechanic_message(MECHANIC_MESSAGE_INFO, "Hello from slave[%d]\n", node);

    return 0;
}

```

We need to compile the example code to a shared library. We can do that by calling

```

gcc -fPIC mechanic_module_hello.c -o mechanic_module_hello.o
gcc -shared mechanic_module_hello.o -o mechanic_module_hello.so

```

Mechanic need to know, where our module is, so we need to adjust `LD_LIBRARY_PATH` (it depends on shell you are using) to the place we saved our module.

We run **Mechanic** with our module by

```

mpirun -np 3 mechanic -p hello

```

This will tell **Mechanic** to run on three nodes, in a task farm mode, with one master node and two slaves. The master node will send default initial condition (pixel coordinates) to each slave and receive data in `masterData` structure (in this case the coordinates of the pixel).

The output should be similar to:

```

-> Mechanic
   v. 0.12-UNSTABLE-2
  Author: MSlonina, TCfA, NCU
  Bugs: mariusz.slonina@gmail.com
  http://mechanics.astri.umk.pl/projects/mechanic
!! Config file not specified/doesn't exist. Will use defaults.
-> Mechanic will use these startup values:
(...)
-> Hello from slave[1]
-> Hello from slave[2]

```

Two last lines were printed using our simple module. In the working directory you should find also `mechanic-master.h5` file. It is a data file written by the master node, and each run of **Mechanic** will produce such file. It contains all information about the setup of the simulation and data received from slaves.

If you try

```
h5dump -n mechanic-master.h5
```

you should see the following output:

```
HDF5 "mechanic-master.h5" {  
  FILE_CONTENTS {  
    group      /  
    dataset    /board  
    group      /config  
    dataset    /config/default  
    dataset    /config/logs  
    group      /data  
    dataset    /data/master  
  }  
}
```

which describes the data storage in master file.

The `Hello` module is included in `Mechanic` distribution as a simple example of using the software.

3.1 The Hello module

Let us go step-by-step through the `Hello` module. Each `Mechanic` module must contain the preprocessor directive

```
#include "mechanic.h"
```

The module specific header file

```
#include "mechanic_module_hello.h"
```

is optional. Since each module is a normal C code, you can also use any other headers and link to any other library during compilation.

Every function in the module is prefixed with the name of the module – thus, you should use unique names for your modules. The file name prefix, `mechanic_module_` is required for proper module loading.

The first three functions:

- `hello_init()`
- `hello_cleanup()`
- `hello_pixelCompute()`

are required for the module to work. `Mechanic` will abort if any of them is missing. The fourth one, `hello_slave_out()` is optional and belongs to the templateable functions group (see **The template system** (p.10)).

Each function should return an integer value, 0 on success and errcode on failure, which is important for proper error handling.

- `hello_init(moduleInfo* md)` is called on module initialization and you need to provide some information about the module, especially, `md->mrl`, which is the length of the results array sended from the slave node to master node. The `moduleInfo` type contains information about the module, and will be extended in the future. The structure is available for all module functions. The `moduleInfo` type has the following shape:

```
typedef struct {
    const char *name;
    const char *author;
    const char *date;
    const char *version;
    int mrl;
} moduleInfo;
```

- `hello_cleanup(moduleInfo* md)` currently does nothing, however, it is required for future development.
- `hello_pixelCompute(int node, moduleInfo* md, configData* d, masterData* r)` is the heart of your module. Here you can compute almost any type of numerical problem or even you can call external application from here. There are technically no contradictions for including Fortran based code. In this simple example we just assign coordinates of the simulation (see **Pixel-coordinate system** (p.9)) to the result array `r->res`. The array is defined in `masterData` structure, as follows:

```
typedef struct {
    MECHANIC_DATATYPE *res;
    int coords[3]; /* 0 - x 1 - y 2 - number of the pixel */
} masterData;
```

Currently, `MECHANIC_DATATYPE` is set to `double`, so we need to do proper casting from integer to double. The result array has the `md->mrl` size, in this case 3. The `masterData` structure is available for all module functions.

- `hello_slave_out(int nodes, int node, moduleInfo* md, configData* d, masterData* r)` prints formatted message from the slave node on the screen, after the node did its job. The `mechanic_message()` (see **Short Developers guide** (p.16)) is available for all modules, and can be used for printing different kinds of messages, i.e. some debug information or warning.

The `Mechanic` package contains few other modules:

- `Module` – the default module with all available functions included
- `Echo` – an extended version of the `Hello` module, which includes some advanced stuff on handling data files, and is an example of using template system
- `Mandelbrot` – a benchmark module, which computes The Mandelbrot fractal.

3.2 The setup system

`Mechanic` uses standard configuration path – first, we read defaults, then the config file and command line options. The latter two are optional, and if not present, the code will use defaults fixed at compilation time.

To find out what command line options are available, try

```
mpirun -np 3 mechanic --help
```

The configuration data is available to slave nodes by the structure:

```
typedef struct {  
    char* name;  
    char* datafile;  
    char* module;  
    int xres;  
    int yres;  
    int method;  
    int checkpoint;  
    int restartmode;  
    int mode;  
} configData;
```

3.2.1 Command line options

The full list of command line options is included below:

- `--help --usage -?` – prints help message
- `--name -n` – the problem name, it will be used to prefix all data files specific in given run
- `--config -c` – config file to use in the run
- `--module -p` – module which should be used during the run
- `--method -m` – pixel mapping method (0 – default, 6 – user-defined)
- `--xres -x` – x resolution of the simulation map
- `--yres -y` – y resolution of the simulation map
- `--checkpoint -d` – checkpoint file write interval

Restart (checkpoints) options are [TODO]:

- `--restart -r` – switch to restart mode
- `--rpath -b` – checkpoint file path

Mechanic can operate in different modes, see **Modes** (p.9) for details. You can switch between them by using:

- `-0` – masteralone mode
- `-1` – MPI task farm mode
- `-2` – multi task farm mode [TODO]

3.2.2 Config file

Mechanic uses LRC for handling config files. To load configuration from custom config file use `-c` or `--config` switch. If this option is set, but the file doesn't exist, **Mechanic** will abort. Sample config file is given below:

```
[default]
name = hello
xres = 4 #must be greater than 0
yres = 4 #must be greater than 0
method = 0 #single pixel -- 0, userdefined -- 6
module = hello # modules: hello, echo, mandelbrot, module
mode = 1 # masteralone -- 0, task farm -- 1, multi task farm -- 2

[logs]
checkpoint = 4
```

The config file options are equivalents of command line options. Any other option will be silently ommited. If any of the variables is missing, **Mechanic** will use defaults for each not found variable. Namespaces are mandatory and **Mechanic** will abort if missing. The errors are handled by LRC in this case.

You can include full or inline comments in your file, just after the comment mark `#`. The configuration is stored in the master file, see **Data storage scheme** (p.9).

3.2.3 Examples

The general rule for running **Mechanic** is to use:

```
mpirun -np NUMBER_OF_CPUS_TO_USE mechanic [OPTIONS]
```

Here we provide and explain some simple examples:

- `mpirun -np 4 mechanic -p mandelbrot -x 200 -y 200 -n fractal` **Mechanic** will use 4 nodes in MPI task farm mode (one master and three slaves) and will compute the Mandelbrot fractal with resolution 200x200 pixels. The name of the run will be "fractal".
- `mpirun -np 4 mechanic -p mandelbrot -x 200 -y 200 -n fractal -0` This is a similar example, in this case **Mechanic** will compute the fractal in masteralone mode. Slave nodes will be terminated.
- `mpirun -np 4 mechanic -p mandelbrot -x 1 -y 1` Here we can do only one simulation using the Mandelbrot module. In this case, slave nodes 2 and 3 will be terminated (see **Modes** (p.9)).
- `mpirun -np 4 mechanic -p application -x 100 -y 1` We can also create a one-dimensional simulation map, by setting one of the axes to 1. This is especially useful in non-image computations, such as observation reduction – we can call **Mechanic** to perform tasks i.e. on 100 stars.
- `mpirun -np 1 mechanic [OPTIONS] mechanic [OPTIONS]` **Mechanic** will automatically switch to masteralone mode.

3.3 Pixel-coordinate system

Mechanic was created for handling simulations related to dynamical maps. Thus, it uses 2D pixel coordinate system (there are plans for extending it to other dimensions). This was the simplest way to show which simulations have been computed and which not. The map is stored in `/board` table in the master file. Each finished simulation is marked with 1, the ongoing or broken – with 0.

It is natural to use (x,y)-resolution option (either in the config file or command line) to describe the map of pixels for an image (like a dynamical map or the Mandelbrot fractal). However, one can use slice-based mapping, by using i.e. `100x1` or `1x100` resolution. In either case, the result should be the same. Setting $(x,y) = (1,1)$ is equivalent of doing only one simulation.

The mapping should help you in setting initial conditions for the simulation, i.e. we can change some values by using pixel coordinates or the number of the pixel. This information is available during the computation and is stored in `masterData` struct.

By default, the number of simulations is counted by multiplying x and y resolution. The simulations are currently done one-by-one, the master node does not participate in computations (except masteralone mode, see **Modes** (p.9)). You can change default behaviour by using `method = 6`, see **The Method 6** (p.15).

3.4 Modes

Mechanic can compute simulations both in single-cpu mode (masteralone) or multi-cpu mode (MPI task farm).

- Masteralone mode – This mode is especially useful if you run **Mechanic** in single-cpu environment. If the mode is used in multi-cpu environments and the size of MPI group is greater than 1, **Mechanic** will terminate all nodes but the master node.
- MPI Task farm – The classical, and default mode for **Mechanic**. This will use one master node and number of slave nodes to do simulations. The master node is responsible for sending/receiving data and storing them. If number of slave nodes is greater than number of simulations to do, all unused nodes will be terminated.

3.5 Data storage scheme

Mechanic writes data in the following scheme:

- `/config` – configuration file (written by LRC API)
- `/board` – simulation mapping
- `/data` – main data group
- `/data/master` – master dataset, contains data received from nodes.

The file can be viewed with `hdfutils`, i.e. `h5dump`. The master file has always `problemname-master.h5` name. If the master file exists in the current working dir, it will be automatically backedup.

The Parallel HDF has no support for MPI task farm, thus the only node allowed to write master file is the master node. However, the module can provide additional data files and operate on them, see **The Echo module** (p.11) for the example.

3.6 Checkpoints

Handling checkpoint files should be done in `Mechanic unstable-3` milestone.

4 Advanced topics

4.1 Case studies

There are some basic use cases of `Mechanic` and its template system:

- **Each slave does the same.** This is the simplest case of using `Mechanic`. The only thing to do is to define `pixelCompute()` function and return data to the master node with `masterData` structure. You can also do something more in `node_in/out` functions, but in that case it is not really necessary.
- **Each slave has different config file.** This time you need to read config file for each slave separately. This can be done with LRC in `slave_in()` function and config files named after slave number, i.e. `slave22`.
- **Each slave has different pixelCompute function.** At this point you need to create some subfunctions of `pixelCompute` and choose them accordingly to number of the slave, i.e. in the switch routine.
- **Each slave has both different config file and different pixelCompute.** Just combining two cases in simple switch routines and it should work.

4.2 The template system

`Mechanic` uses some kind of a template system. It allows developer to use different sets of functions at different modes. Below we present list of available template functions and their possible overrides. For sake of clearness, we omitted function's arguments.

- `module_node_in(int mpi_size, int node, moduleInfo* md, configData* d)`
 - `module_master_in()`
 - `module_slave_in()`
- `module_node_out(int mpi_size, int node, moduleInfo* md, configData* d)`
 - `module_master_out()`
 - `module_slave_out()`
- `module_node_beforeSend(int node, moduleInfo* md, configData* d, masterData* r)`
 - `module_master_beforeSend()`
 - `module_slave_beforeSend()`
- `module_node_afterSend(int node, moduleInfo* md, configData* d, masterData* r)`
 - `module_master_afterSend()`

- module_slave_afterSend()
- module_node_beforeReceive(int node, moduleInfo* md, configData* d, masterData* r)
 - module_master_beforeReceive()
 - module_slave_beforeReceive()
- module_node_afterReceive(int node, moduleInfo* md, configData* d, masterData* r)
 - module_master_afterReceive()
 - module_slave_afterReceive()
- module_node_before_pixelCompute(int node, moduleInfo* md, configData* d, masterData* r)
 - module_master_beforePixelCompute()
 - module_slave_beforePixelCompute()
- module_node_after_pixelCompute(int node, moduleInfo* md, configData* d, masterData* r)
 - module_master_afterPixelCompute()
 - module_slave_afterPixelCompute()

Each template function is optional, so **Mechanic** will silently skip it if it is missing. Refer to **The Echo module** (p. 11) for a simple example of using the template system.

4.3 The Echo module

Here we present possible usage of the template system. We will use `node_in()` and `node_out()` functions as examples.

We implement `node_in()` and `node_out()` functions as follows:

```
int echo_node_in(int mpi_size, int node, moduleInfo* md, configData* d){
    mechanic_message(MECHANIC_MESSAGE_INFO, "NodeIN [%d]\n", node);

    return 0;
}

int echo_node_out(int mpi_size, int node, moduleInfo* md, configData* d,
    masterData* r){
    mechanic_message(MECHANIC_MESSAGE_INFO, "NodeOUT [%d]\n", node);

    return 0;
}
```

They will be used if no override is present. However, we can create overrides. For the master node we have:

```

int echo_master_in(int mpi_size, int node, moduleInfo* md, configData* d){

    return 0;
}

```

which will override the output of `node_in()` on the master node. We can create a much more complicated function, as for the `node_in()` at slave node:

```

int echo_slave_in(int mpi_size, int node, moduleInfo* md, configData* d,
    masterData* r){

    hid_t sfile_id, gid, string_type;
    hid_t dataset, dataspace;
    hid_t rank = 1;
    hsize_t dimsens_1d;
    herr_t serr;
    char sbase[] = "slave";
    char nodename[512];
    char gbase[] = "slave";
    char group[512];
    char oldfile[1028];

    char cbase[] = "Hello from slave ";
    char comment[1024];

    struct stat st;

    mechanic_message(MECHANIC_MESSAGE_INFO, "ECHO IN: %s\n", d->name);
    sprintf(nodename, "%s-%s%.5d.h5", d->name, sbase, node);
    sprintf(group, "%s%d", gbase, node);

    if (stat(nodename,&st) == 0) {
        sprintf(oldfile, "old-%s", nodename);
        rename(nodename,oldfile);
    }

    sfile_id = H5Fcreate(nodename, H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);
    gid = H5Gcreate(sfile_id, group, H5P_DEFAULT, H5P_DEFAULT, H5P_DEFAULT);

    sprintf(comment, "%s%.5d. ", cbase, node);
    string_type = H5Tcopy(H5T_C_S1);
    H5Tset_size(string_type, strlen(comment));

    rank = 1;
    dimsens_1d = 1;

    dataspace = H5Screate_simple(rank, &dimsens_1d, NULL);

    dataset = H5Dcreate(gid, "comment", string_type, dataspace, H5P_DEFAULT,
        H5P_DEFAULT, H5P_DEFAULT);
    serr = H5Dwrite(dataset, string_type, H5S_ALL, dataspace, H5P_DEFAULT,
        comment);

    H5Sclose(dataspace);
    H5Dclose(dataset);
    H5Gclose(gid);
    H5Fclose(sfile_id);

    return 0;
}

```

This function use advantage of HDF storage. Each slave will create its own data file and print a comment to it.

Now, after all pixel have been computed, we tell our master node to copy slave data files to the master data file, as shown below:

```
int echo_master_out(int nodes, int node, moduleInfo* md, configData* d,
    masterData* r){

    int i = 0;
    hid_t fname, masterfile, masterdatagroup;
    herr_t stat;
    char groupname[512];
    char filename[512];

    stat = H5open();

    mechanic_message(MECHANIC_MESSAGE_INFO, "ECHO MASTER IN: %s\n", d->datafile);
    masterfile = H5Fopen(d->datafile, H5F_ACC_RDWR, H5P_DEFAULT);
    masterdatagroup = H5Gopen(masterfile, "data", H5P_DEFAULT);

    for (i = 1; i < nodes; i++) {
        sprintf(groupname, "slave%d", i);
        sprintf(filename, "%s-slave%d.h5", d->name, i);

        fname = H5Fopen(filename, H5F_ACC_RDONLY, H5P_DEFAULT);
        stat = H5Ocopy(fname, groupname, masterdatagroup, groupname,
            H5P_DEFAULT, H5P_DEFAULT);
        if (stat < 0) mechanic_message(MECHANIC_MESSAGE_ERR, "copy error\n");

        H5Fclose(fname);
    }

    H5Gclose(masterdatagroup);
    H5Fclose(masterfile);
    stat = H5close();

    mechanic_message(MECHANIC_MESSAGE_INFO,
        "Master process [%d] OVER & OUT.\n", node);

    return 0;
}
```

At the end of simulation, the slave node will print customized message:

```
int echo_slave_out(int mpi_size, int node, moduleInfo* md, configData* d,
    masterData* r){

    mechanic_message(MECHANIC_MESSAGE_INFO, "SLAVE[%d] OVER & OUT\n", node);

    return 0;
}
```

4.4 The Mandelbrot module

This module shows how to use basic api of `Mechanic` to compute any numerical problem, in that case – the Mandelbrot fractal.

We use here only 3 functions: `mandelbrot_init()`, `mandelbrot_cleanup()` and `mandelbrot_pixelCompute()`. There is an additional function, `mandelbrot_generateFractal()`, which shows that you can even add external functions to your module, since it is a standard C code.

In addition, the module returns the number of node that computed the pixel.

```

#include "mechanic.h"
#include "mechanic_module_mandelbrot.h"

int mandelbrot_init(moduleInfo *md){

    md->name = "mandelbrot";
    md->author = "MSlonina";
    md->date = "2010";
    md->version = "1.0";
    md->mrl = 4;

    return 0;
}

int mandelbrot_cleanup(moduleInfo *md){

    return 0;
}

int mandelbrot_pixelCompute(int slave, moduleInfo *md, configData* d,
    masterData* r){

    double real_min, real_max, imag_min, imag_max;
    double scale_real, scale_imag;
    double c;

    real_min = -2.0;
    real_max = 2.0;
    imag_min = -2.0;
    imag_max = 2.0;
    c = 4.0;

    scale_real = (real_max - real_min) / ((double) d->xres - 1.0);
    scale_imag = (imag_max - imag_min) / ((double) d->yres - 1.0);

    r->res[0] = real_min + r->coords[0] * scale_real;
    r->res[1] = imag_max - r->coords[1] * scale_imag;

    r->res[2] = mandelbrot_generateFractal(r->res[0], r->res[1], c);

    r->res[3] = (double) slave;

    return 0;
}

int mandelbrot_generateFractal(double a, double b, double c){

    double temp, lengthsq;
    int max_iter = 256;
    int count = 0;
    double zr = 0.0, zi = 0.0;

    do{

        temp = zr*zr - zi*zi + a;
        zi = 2*zr*zi + b;
        zr = temp;
        lengthsq = zr*zr + zi*zi;
        count++;

    } while ((lengthsq < c) && (count < max_iter));

    return count;
}

```

The header file:

```
#ifndef MECHANIC_MODULE_MANDELBROT_H
#define MECHANIC_MODULE_MANDELBROT_H

#include <stdio.h>
#include <stdlib.h>

int mandelbrot_generateFractal(double a, double b, double c);

#endif
```

4.5 The Method 6

You can change default **Mechanic** pixel mapping and simulation handling by setting **method = 6** . In this case, you need to provide additional functions in your module. If any of them is missing, **Mechanic** will abort.

The **farmResolution()** simply returns number of simulations to do.

```
int module_farmResolution(int x, int y, moduleInfo* md, configData* d){

    return x*y;
}
```

The **pixelCoordsMap()** operates on **t** index and should return 0 on success, errcode otherwise. The default behaviour is to map pixels on 2D board, as shown below:

```
int module_pixelCoordsMap(int t[], int numofpx, int xres, int yres, moduleInfo* md,
    configData* d){

    if (numofpx < yres) {
        t[0] = numofpx / yres;
        t[1] = numofpx;
    }

    if (numofpx > yres - 1) {
        t[0] = numofpx / yres;
        t[1] = numofpx % yres;
    }

    return 0;
}
```

The **pixelCoords()** assigns pixel coordinates to **masterData r** structure. The default behaviour is to copy **t** index to **r->coords** .

```
int module_pixelCoords(int node, int t[], moduleInfo* md, configData* d,
    masterData* r){

    r->coords[0] = t[0];
    r->coords[1] = t[1];
    r->coords[2] = t[2];

    return 0;
}
```

4.6 Working with data

Using h5dump, pytables, hdfview, vitables, vim and sed for first steps of visualization of the data.

5 Short Developers guide

Some notes for developing the code.

5.1 Coding style

- We try to follow ANSI C standard as close as possible, however we use also some advantages of C99. Thus, you should use ANSI C where possible, and C99 where necessary.
- We use ANSI C comment style. Because of Doxygen documenting style, use one asteriks * at start of your non-documentation comments.
- We use 2 spaces indenting style (you can easily map tabs to 2 spaces).
- Try not to use globals. Avoid them where possible.
- Remember, the code should be readable by humans. For the machines, it does not matter.
- We follow PEAR coding standards, see <http://pear.php.net/manual/en/standards.php>

If you are a lucky vim user, try settings below:

```
:set textwidth=79
:set shiftwidth=2
:set tabstop=2
:set smarttab
:set expandtab
:set list
:highlight OverLength ctermbg=red ctermfg=white guibg=#592929
:match OverLength /\%80v.* /
:let c_space_errors = 1
```

5.2 Message interface

Mechanic provides some functions that should be used instead of standard `printf` and `exit`:

- `void mechanic_message(int type, char* fmt, ...)`

A wrapper for any message printing. The available types are:

- `MECHANIC_MESSAGE_INFO` – information style
- `MECHANIC_MESSAGE_CONT` – continuation of any message style (when the message is too long, and you want to split it)
- `MECHANIC_MESSAGE_ERR` – error only
- `MECHANIC_MESSAGE_WARN` – warning only
- `MECHANIC_MESSAGE_DEBUG` – debug only

The `*fmt` is a standard `printf` format, and dots `...` are arguments for it.

- `mechanic_error(int stat)`
Handles error states, see **Error codes** (p. 17) for available error codes.
- `mechanic_abort(int node)`
A wrapper for `MPI_Abort`.
- `mechanic_finalize(int node)`
A wrapper for `MPI_Finalize`.

6 Troubleshooting

6.1 Known bugs and missing features

Known bugs and missing features:

- restart/checkpoint system
- HDF error handling in a better way
- extension to the template system: overrides specific for given node

6.2 Error codes

In case of emergency **Mechanic** tries to properly finalize all nodes and returns error codes as described below:

- 911 – MPI related error
- 912 – HDF related error
- 913 – Module interface related error
- 914 – Setup interface related error
- 915 – Memory allocation related error
- 999 – Any other error