Reduced Precision Emulator Manual

Release 5.1.dev0

Andrew Dawson, Peter Dueben

Jul 24, 2017

Contents

1 Overview **3**
1.1 Basic use of the reduced-precision type . . . 3
1.2 Controlling the precision . . . 3

2 Accessing the Source Code **5**

3 Building the Emulator **7**
3.1 Build requirements . . . 7
3.2 Building . . . 7
3.3 Integration . . . 8

4 Using the Emulator **9**
4.1 Using the emulator in your code . . . 9

5 Potential Sources of Error **11**
5.1 Initialization on assignment . . . 11
5.2 Changing `RPE_DEFAULT_SBITS` during execution . . . 12
5.3 Parallel and thread safety . . . 13
5.4 Non-equivalence of single and compound operations . . . 13

6 Developer Guide **15**
6.1 Getting Started . . . 15
6.2 Software Structure . . . 16
6.3 Working with *rpe* source code . . . 17
6.4 Code Generator . . . 34
6.5 Making a release . . . 39

7 API Reference **43**
7.1 RPE API: `rp_emulator.mod` . . . 43

8 Downloads **47**
8.1 Changelog . . . 47

9 Reduced Precision Emulator **51**
9.1 Citation . . . 51

This user guide provides documentation for using the reduced precision emulation Fortran library. If you have questions about the software or need help using it you can use the project mailing list or the Github issue tracker.

CHAPTER 1

Overview

The library contains a derived type: *rpe_var*. This type can be used in place of real-valued variables to perform calculations with floating-point numbers represented with a reduced number of bits in the floating-point significand.

Basic use of the reduced-precision type

The *rpe_var* type is a simple container for a double precision floating point value. Using an *rpe_var* instance is as simple as declaring it and using it just as you would a real number:

```
TYPE(rpe_var) :: myvar

myvar = 12
myvar = myvar * 1.287   ! reduced-precision result is stored in `myvar`
```

Controlling the precision

The precision used by reduced precision types can be controlled at two different levels. Each reduced precision variable has an `sbits` attribute which controls the number of explicit bits in its significand. This can be set independently for different variables, and comes into effect after it is explicitly set.

```
TYPE(rpe_var) :: myvar1
TYPE(rpe_var) :: myvar2

! Use 16 explicit bits in the significand of myvar1, but only 12 in the
! significand of myvar2.
myvar1%sbits = 16
myvar2%sbits = 12
```

For variables whose `sbits` attribute has not been explicitly set, there is a default global precision level, set by *RPE_DEFAULT_SBITS*. This will stand-in for the number of explicit significand bits in any variable where `sbits` has not been set. Setting *RPE_DEFAULT_SBITS* once to define the global default precision, and setting the precision of variables that require other precision manually using the `sbits` attribute is generally a good strategy. However, if you change *`RPE_DEFAULT_SBITS`* during execution, the document *Changing RPE_DEFAULT_SBITS during execution* lists some details you should be aware of.

The emulator can also be turned off completely by setting the module variable *`RPE_ACTIVE`* to `.FALSE.`.

CHAPTER 2

Accessing the Source Code

If you just want to access the source code, but you don't want to do any development for now, you can follow these instructions. The *Developer Guide* has instructions for those wishing to do development work with the code.

Released versions of the code can be downloaded from the *Downloads* page. If you want to follow the development branch of the code see the instructions in *Following the latest source*.

CHAPTER 3

Building the Emulator

Build requirements

Building the emulator requires:

- GCC (gfortran) >= 4.8 or Intel Fortran (ifort) >= 15.0.2
- GNU Make

The emulator has been tested with GCC 4.8.4 and GCC 4.9.2 and Intel Fortran 15.0.2 and is known to work correctly with these compilers. No testing of other compilers has been done by us, it might work with other compilers, it might not.

Building

The library is built using GNU Make and a Fortran compiler. The default action when invoking *make* is to build the static library *lib/librpe.a* and the Fortran module *modules/rp_emulator.mod*.

The code is tested with and set-up for building with GNU gfortran or Intel Fortran (ifort) compilers. Which compiler to use is determined by the value of the *F90* environment variable. If this variable is not set the Makefile will use *gfortran*. The default build is a simple invocation of *make*:

```
make
```

It is also possible to build a shared library *lib/librpe.so* using the *shared* target of the Makefile:

```
make shared
```

You can optionally specify a compiler on the command line:

```
F90=ifort make
```

If you want to use a compiler other than *gfortran* or *ifort* you will need to specify both the appropriate *F90* variable and the correct *FFLAGS* for your compiler. Compiler flags are used in the build to ensure the module *rp_emulator.mod* is placed in the *modules/* directory in the source tree.

Unified source

The source code for the emulator is split across several files, and makes use of the C preprocessor to combine them during the build process. If you want to generate a unified source file for ease of use you can use the *source* target of the Makefile:

```
make source
```

This will generate the file `src/processed/rp_emulator.f90` which can be integrated into the source of other projects.

Integration

Assuming you did a full build, integration with your project is fairly straightforward, requiring two files to be present, one at compile time and one at link time.

You must make sure that the module file `rp_emulator.mod` is available to the compiler when compiling any source file that uses the module. You can do this by specifying an include flag at compile time. Alternatively you could place the module file in the same directory as your source files, but normally you would store it separately and use an include flag.

At link time the `librpe.a` (or `librpe.so`) library will also need to be available to the linker. You can use a combination of linker path and library options to make sure this is the case. Alternatively, you can directly specify the full path to `librpe.a` as an object to include in linking.

For example, let's say we have placed the module file at `$HOME/rpe/modules/rp_emulator.mod` and the library at `$HOME/rpe/lib/librpe.a`, our compilation command must tell the compiler to look in the right place for the module file:

```
gfortran -c -I$HOME/rpe/modules myprogram.f90
```

and our linker command must tell the linker which libraries to link and where to look for them:

```
gfortran -o myprogram.exe myprogram.o -L$HOME/rpe/lib -lrpe
```

The arguments are the same whether linking the static or shared library.

Unified source builds

If you wish, you can just build the unified source code of the emulator directly in your project.

CHAPTER 4

Using the Emulator

Using the emulator in your code

In any subroutine, module or program where you want to use the emulator, you need to import the emulator's module. For example, to include the emulator in a particular subroutine:

```
SUBROUTINE some_routine (...)
    USE rp_emulator
    ...
END SUBROUTINE some_routine
```

You can then use any of the emulator features within the subroutine.

CHAPTER 5

Potential Sources of Error

There are a few dark corners of both the library and Fortran itself which may catch out unsuspecting users.

Initialization on assignment

The emulator overloads the assignment operator allowing you to assign integers, real numbers, and other *rpe_var* instances to any *rpe_var* instance:

```fortran
PROGRAM assign_other_types

    USE rp_emulator
    IMPLICIT NONE

    TYPE(rpe_var) :: x

    ! Assign an integer value to x.
    x = 4

    ! Assign a single-precision real value to x.
    x = 4.0

    ! Assign a double-precision real value to x.
    x = 4.0d0

END PROGRAM
```

However, you cannot perform the same assignments at the time the *rpe_var* is defined. This is not allowed:

```fortran
PROGRAM assign_other_types

    USE rp_emulator
    IMPLICIT NONE

    TYPE(rpe_var) :: x = 4
```

```
END PROGRAM
```

Compiling this code with *gfortran* will give the following compilation error:

```
    TYPE(rpe_var) :: x = 4
                        1
Error: Can't convert INTEGER(4) to TYPE(rpe_var) at (1)
```

This is because this form of assignment does not actually call the overloaded assignment operator defined for *rpe_var* instances, instead it calls the default constructor for the derived type, which only allows the right-hand-side of the assignment to be another *rpe_var* instance.

Changing `RPE_DEFAULT_SBITS` during execution

You are able to change the value of the module variable *RPE_DEFAULT_SBITS* to anything you like at any time you like. However, you need to be aware of the potential inconsistencies you might introduce by doing so.

```
PROGRAM change_default_bits1

    USE rp_emulator
    IMPLICIT NONE

    TYPE(rpe_var) :: pi

    RPE_DEFAULT_SBITS = 16

    ! This value of Pi will be stored with only 16 bits in the mantissa.
    pi = 3.1415926535897932d0
    WRITE (*, '("RPE_DEFAULT_SBITS=16, pi=", F20.18)') pi%get_value()

    ! Doing this means that whilst any operations on Pi following will assume
    ! 4 bits of significand precision, the value currently stored still has 16
    ! bits of significand precision
    RPE_DEFAULT_SBITS = 4
    WRITE (*, '("RPE_DEFAULT_SBITS=4,  pi=", F20.18)') pi%get_value()

END PROGRAM
```

Output:

```
RPE_DEFAULT_SBITS=16, pi=3.141601562500000000
RPE_DEFAULT_SBITS=4,  pi=3.141601562500000000
```

To avoid any issues you may want to insert manual calls to *apply_truncation()* to ensure every variable used within the scope of the changed default is represented at the required precision.

In other circumstances this may not be a problem at all, for example around encapsulated subroutine calls. In the example below the procedure `some_routine()` takes no reduced precision types as arguments, but does work with reduced precision types internally, and in this case setting the default number of bits around the subroutine call is a useful way to set the default precision of all reduced precision variables within the subroutine (and within any routines it calls):

```
RPE_DEFAULT_SBITS = 4
CALL some_routine (a, b, c)
RPE_DEFAULT_SBITS = 16
```

Whatever you choose to do, you need to make sure you have considered this potential issue before you change the value of *RPE_DEFAULT_SBITS* at run-time.

Parallel and thread safety

The default number of bits for any reduced precision type is controlled by a module variable *RPE_DEFAULT_SBITS*. This is a mutable variable that can be changed dynamically during program execution if desired. If the application using the emulator is parallelised then the behaviour of the default bits setting needs to be considered.

For MPI parallelism, each MPI task will get its own separate instance of the *RPE_DEFAULT_SBITS* variable, and modifying it within a task will only affect the default precision within that task (unless programmed otherwise using message passing).

For threaded parallelism (e.g. OpenMP) the behaviour is less clear. Depending on the threading type used, the variable may be shared by threads, or each may have its own copy.

For parallel applications, care must be taken when changing the value of *RPE_DEFAULT_SBITS* at run time to make sure the implementation is safe.

Non-equivalence of single and compound operations

One would normally expect the following operations to yield identical results:

```
a * a * a
```

and

```
a ** 3
```

However, due to the way the emulator works by doing individual computations in full precision and reducing the precision of the result, these two would not necessarily yield the same result if `a` were a reduced precision variable. In the first example the compound multiplication would be done in two parts, the first part computes `a * a` and the precision of the temporary result is reduced, then the second part multiplies this reduced precision result by `a` and once again reduces the precision of the final result. However, in the second example a single operation is used to express the computation, this computation will be performed in full precision and the result will have its precision reduced. Whereas the first example uses reduced precision to store intermediate results, the second does not.

This is true for any operator that can be expressed as multiple invocations of other operators.

CHAPTER 6

Developer Guide

This developer guide provides extra documentation required for doing development work on the reduced precision emulator.

Getting Started

Requirements

In addition to the user requirements, there are extra dependencies for developers:

Core

- GCC (gfortran) >= 4.8 or Intel Fortran (ifort) >= 15.0.2
- GNU Make
- Git

Test suites

To run the test suites the following software is required:

- Python 2.7
- pFUnit (tested with version 3.1)

Code generator

Running the code generator requires:

- Python 2.7

- Jinja2

Documentation

If you want to build the documentation you will need:

- Python 2.7
- Sphinx >= 1.3.1

Get the source code

Follow the instructions in *Working with rpe source code* to set up your source code repository and version control tools.

Software Structure

The emulator library itself is a Fortran library that can be included in other programs to introduce reduced precision variables. However, the code repository contains several distinct components which are used to generate the final Fortran library.

The core library

The core of the emulator is written in Fortran, and is found in the file `src/rp_emulator.F90`. The emulator library provides two core derived types used to represent reduced precision floating point numbers. These types have accompanying overloaded operator definitions, allowing them to be used in the same way as the builtin real number type. For maximum ease of use, many of the builtin Fortran intrinsic functions are also overloaded to accept these reduced precision types.

The `src/rp_emulator.F90` file contains the module definition, the definition of the derived types, and the core functions and subroutines that control the emulator's functionality. However, it does not contain most of the overloaded operator and intrinsic function definitions. Instead there are a handful of C preprocessor directives in the file that pull in these definitions from external files in `src/include/`. The way in which these external file are generated is described below.

The code generator

A lot of the code required to complete the full emulator is repetetive boiler-plate code; code which is largely the same for a whole group of functions/subroutines. Because of the repetetive nature of the required Fortran implementations for all overloaded operators and intrinsic functions, it is more efficient to provide only a template of what the code should look like, and let the computer actually write the code. The code generator for the emulator is written in Python and is included in the `generator/` subdirectory.

Using a generator is a big time saver, if you need to change 1 line of every intrinsic function implementation, you only need to modify that line in a few templates and let the code generator write all the actual Fortran code for you. Not only does this approach save time, it also has positive implications for code correctness. For example if 50 function implementations are produced by the generator, it is not possible for one of them to contain an error that the others don't, as would often be the case when hand-writing sucha a large number of essentially similar routines. You write the implementation carefully once, and the boring stuff is done automatically.

Due to its relative complexity, the *Code Generator* is documented separately.

Extra definitions

As well as code that is produced by the code generator, there are two more files in `src/include/` that can be edited manually: `interface_extras.i` and `implementation_extras.f90`. You can add arbitrary functions/subroutines to the `implementation_extras.i` file, with any required interface definitions in `interface_extras.i`, and they will be included in the main library source file automatically. Just make sure you make your interface public, as the default is private.

Tests

The emulator is provided with a suite of basic tests, in the `tests/` subdirectory. It is expected that any change you make will not cause any of the existing tests to fail, and ideally new features should be accompanied with new tests to make sure they work, and continue to work in the future.

The tests are split into two categories, units tests and integration tests. Unit tests should be relatively self-contained tests of the functionality of a particular small element of the code (code unit). Ideally these tests are for a single component in isolation from the rest of the system, although the nature of the library means this sometimes is not possible. Integration tests are more like realistic tests of the software in use, and should tests multiple aspects of the library together (in integration).

Working with *rpe* source code

Contents:

Introduction

These pages describe a git and github workflow for the rpe project.

There are several different workflows here, for different ways of working with *rpe*.

This is not a comprehensive git reference, it's just a workflow for our own project. It's tailored to the github hosting service. You may well find better or quicker ways of getting stuff done with git, but these should get you started.

For general resources for learning git, see *git resources*.

Install git

Overview

Debian / Ubuntu	`sudo apt-get install git`
Fedora	`sudo yum install git`
Windows	Download and install msysGit
OS X	Use the git-osx-installer

In detail

See the git page for the most recent information.

Have a look at the github install help pages available from github help

There are good instructions here: http://book.git-scm.com/2_installing_git.html

Following the latest source

These are the instructions if you just want to follow the latest *rpe* source, but you don't need to do any development for now.

The steps are:

- *Install git*
- get local copy of the rpe github git repository
- update local copy from time to time

Get the local copy of the code

From the command line:

```
git clone git://github.com/aopp-pred/rpe.git
```

You now have a copy of the code tree in the new `rpe` directory.

Updating the code

From time to time you may want to pull down the latest code. Do this with:

```
cd rpe
git pull
```

The tree in `rpe` will now have the latest changes from the initial repository.

Making a patch

You've discovered a bug or something else you want to change in rpe .. — excellent!

You've worked out a way to fix it — even better!

You want to tell us about it — best of all!

The easiest way is to make a *patch* or set of patches. Here we explain how. Making a patch is the simplest and quickest, but if you're going to be doing anything more than simple quick things, please consider following the *Git for development* model instead.

Making patches

Overview

```
# tell git who you are
git config --global user.email you@yourdomain.example.com
git config --global user.name "Your Name Comes Here"
# get the repository if you don't have it
git clone git://github.com/aopp-pred/rpe.git
# make a branch for your patching
cd rpe
git branch the-fix-im-thinking-of
git checkout the-fix-im-thinking-of
```

```
# hack, hack, hack
# Tell git about any new files you've made
git add somewhere/tests/test_my_bug.py
# commit work in progress as you go
git commit -am 'BF - added tests for Funny bug'
# hack hack, hack
git commit -am 'BF - added fix for Funny bug'
# make the patch files
git format-patch -M -C master
```

Then, send the generated patch files to the rpe mailing list — where we will thank you warmly.

In detail

1. Tell git who you are so it can label the commits you've made:

   ```
   git config --global user.email you@yourdomain.example.com
   git config --global user.name "Your Name Comes Here"
   ```

2. If you don't already have one, clone a copy of the rpe repository:

   ```
   git clone git://github.com/aopp-pred/rpe.git
   cd rpe
   ```

3. Make a 'feature branch'. This will be where you work on your bug fix. It's nice and safe and leaves you with access to an unmodified copy of the code in the main branch:

   ```
   git branch the-fix-im-thinking-of
   git checkout the-fix-im-thinking-of
   ```

4. Do some edits, and commit them as you go:

   ```
   # hack, hack, hack
   # Tell git about any new files you've made
   git add somewhere/tests/test_my_bug.py
   # commit work in progress as you go
   git commit -am 'BF - added tests for Funny bug'
   # hack hack, hack
   git commit -am 'BF - added fix for Funny bug'
   ```

   Note the `-am` options to `commit`. The `m` flag just signals that you're going to type a message on the command line. The `a` flag — you can just take on faith — or see why the -a flag?.

5. When you have finished, check you have committed all your changes:

   ```
   git status
   ```

6. Finally, make your commits into patches. You want all the commits since you branched from the `master` branch:

   ```
   git format-patch -M -C master
   ```

   You will now have several files named for the commits:

   ```
   0001-BF-added-tests-for-Funny-bug.patch
   0002-BF-added-fix-for-Funny-bug.patch
   ```

Send these files to the rpe mailing list.

When you are done, to switch back to the main copy of the code, just return to the `master` branch:

```
git checkout master
```

Moving from patching to development

If you find you have done some patches, and you have one or more feature branches, you will probably want to switch to development mode. You can do this with the repository you have.

Fork the rpe repository on github — *Making your own copy (fork) of rpe*. Then:

```
# checkout and refresh master branch from main repo
git checkout master
git pull origin master
# rename pointer to main repository to 'upstream'
git remote rename origin upstream
# point your repo to default read / write to your fork on github
git remote add origin git@github.com:your-user-name/rpe.git
# push up any branches you've made and want to keep
git push origin the-fix-im-thinking-of
```

Then you can, if you want, follow the *Development workflow*.

Git for development

Contents:

Making your own copy (fork) of rpe

You need to do this only once. The instructions here are very similar to the instructions at http://help.github.com/forking/ — please see that page for more detail. We're repeating some of it here just to give the specifics for the rpe project, and to suggest some default names.

Set up and configure a github account

If you don't have a github account, go to the github page, and make one.

You then need to configure your account to allow write access — see the `Generating SSH keys` help on github help.

Create your own forked copy of rpe

1. Log into your github account.
2. Go to the rpe github home at rpe github.
3. Click on the *fork* button:

Now, after a short pause and some ‘Hardcore forking action’, you should find yourself at the home page for your own forked copy of rpe.

Set up your fork

First you follow the instructions for *Making your own copy (fork) of rpe*.

Overview

```
git clone git@github.com:your-user-name/rpe.git
cd rpe
git remote add upstream git://github.com/aopp-pred/rpe.git
```

In detail

Clone your fork

1. Clone your fork to the local computer with `git clone git@github.com:your-user-name/rpe.git`
2. Investigate. Change directory to your new repo: `cd rpe`. Then `git branch -a` to show you all branches. You’ll get something like:

   ```
   * master
   remotes/origin/master
   ```

   This tells you that you are currently on the `master` branch, and that you also have a `remote` connection to `origin/master`. What remote repository is `remote/origin`? Try `git remote -v` to see the URLs for the remote. They will point to your github fork.

   Now you want to connect to the upstream rpe github repository, so you can merge in changes from trunk.

Linking your repository to the upstream repo

```
cd rpe
git remote add upstream git://github.com/aopp-pred/rpe.git
```

`upstream` here is just the arbitrary name we’re using to refer to the main rpe repository at rpe github.

Note that we’ve used `git://` for the URL rather than `git@`. The `git://` URL is read only. This means we that we can’t accidentally (or deliberately) write to the upstream repo, and we are only going to use it to merge into our own code.

Just for your own satisfaction, show yourself that you now have a new 'remote', with `git remote -v show`, giving you something like:

```
upstream     git://github.com/aopp-pred/rpe.git (fetch)
upstream     git://github.com/aopp-pred/rpe.git (push)
origin       git@github.com:your-user-name/rpe.git (fetch)
origin       git@github.com:your-user-name/rpe.git (push)
```

Configure git

Overview

Your personal git configurations are saved in the `.gitconfig` file in your home directory.

Here is an example `.gitconfig` file:

```
[user]
        name = Your Name
        email = you@yourdomain.example.com

[alias]
        ci = commit -a
        co = checkout
        st = status
        stat = status
        br = branch
        wdiff = diff --color-words

[core]
        editor = vim

[merge]
        summary = true
```

You can edit this file directly or you can use the `git config --global` command:

```
git config --global user.name "Your Name"
git config --global user.email you@yourdomain.example.com
git config --global alias.ci "commit -a"
git config --global alias.co checkout
git config --global alias.st "status -a"
git config --global alias.stat "status -a"
git config --global alias.br branch
git config --global alias.wdiff "diff --color-words"
git config --global core.editor vim
git config --global merge.summary true
```

To set up on another computer, you can copy your `~/.gitconfig` file, or run the commands above.

In detail

user.name and user.email

It is good practice to tell git who you are, for labeling any changes you make to the code. The simplest way to do this is from the command line:

```
git config --global user.name "Your Name"
git config --global user.email you@yourdomain.example.com
```

This will write the settings into your git configuration file, which should now contain a user section with your name and email:

```
[user]
      name = Your Name
      email = you@yourdomain.example.com
```

Of course you'll need to replace `Your Name` and `you@yourdomain.example.com` with your actual name and email address.

Aliases

You might well benefit from some aliases to common commands.

For example, you might well want to be able to shorten `git checkout` to `git co`. Or you may want to alias `git diff --color-words` (which gives a nicely formatted output of the diff) to `git wdiff`

The following `git config --global` commands:

```
git config --global alias.ci "commit -a"
git config --global alias.co checkout
git config --global alias.st "status -a"
git config --global alias.stat "status -a"
git config --global alias.br branch
git config --global alias.wdiff "diff --color-words"
```

will create an `alias` section in your `.gitconfig` file with contents like this:

```
[alias]
        ci = commit -a
        co = checkout
        st = status -a
        stat = status -a
        br = branch
        wdiff = diff --color-words
```

Editor

You may also want to make sure that your editor of choice is used

```
git config --global core.editor vim
```

Merging

To enforce summaries when doing merges (`~/.gitconfig` file again):

```
[merge]
   log = true
```

Or from the command line:

```
git config --global merge.log true
```

Fancy log output

This is a very nice alias to get a fancy log output; it should go in the `alias` section of your `.gitconfig` file:

```
lg = log --graph --pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%cr)
↪%C(bold blue)[%an]%Creset' --abbrev-commit --date=relative
```

You use the alias with:

```
git lg
```

and it gives graph / text output something like this (but with color!):

```
* 6d8e1ee - (HEAD, origin/my-fancy-feature, my-fancy-feature) NF - a fancy file (45␣
↪minutes ago) [Matthew Brett]
*   d304a73 - (origin/placeholder, placeholder) Merge pull request #48 from hhuuggoo/
↪master (2 weeks ago) [Jonathan Terhorst]
|\
| * 4aff2a8 - fixed bug 35, and added a test in test_bugfixes (2 weeks ago) [Hugo]
|/
* a7ff2e5 - Added notes on discussion/proposal made during Data Array Summit. (2␣
↪weeks ago) [Corran Webster]
* 68f6752 - Initial implimentation of AxisIndexer - uses 'index_by' which needs to be␣
↪changed to a call on an Axes object - this is all very sketchy right now. (2 weeks␣
↪ago) [Corr
*   376adbd - Merge pull request #46 from terhorst/master (2 weeks ago) [Jonathan␣
↪Terhorst]
|\
| * b605216 - updated joshu example to current api (3 weeks ago) [Jonathan Terhorst]
| * 2e991e8 - add testing for outer ufunc (3 weeks ago) [Jonathan Terhorst]
| * 7beda5a - prevent axis from throwing an exception if testing equality with non-
↪axis object (3 weeks ago) [Jonathan Terhorst]
| * 65af65e - convert unit testing code to assertions (3 weeks ago) [Jonathan␣
↪Terhorst]
| *   956fbab - Merge remote-tracking branch 'upstream/master' (3 weeks ago)␣
↪[Jonathan Terhorst]
| |\
| |/
```

Thanks to Yury V. Zaytsev for posting it.

Development workflow

You already have your own forked copy of the rpe repository, by following *Making your own copy (fork) of rpe*. You have *Set up your fork*. You have configured git by following *Configure git*. Now you are ready for some real work.

Workflow summary

In what follows we'll refer to the upstream rpe `master` branch, as "trunk".

- Don't use your `master` branch for anything. Consider deleting it.

- When you are starting a new set of changes, fetch any changes from trunk, and start a new *feature branch* from that.
- Make a new branch for each separable set of changes — "one task, one branch" (ipython git workflow).
- Name your branch for the purpose of the changes - e.g. `bugfix-for-issue-14` or `refactor-database-code`.
- If you can possibly avoid it, avoid merging trunk or any other branches into your feature branch while you are working.
- If you do find yourself merging from trunk, consider *Rebasing on trunk*
- Ask on the rpe mailing list if you get stuck.
- Ask for code review!

This way of working helps to keep work well organized, with readable history. This in turn makes it easier for project maintainers (that might be you) to see what you've done, and why you did it.

See linux git workflow and ipython git workflow for some explanation.

Consider deleting your master branch

It may sound strange, but deleting your own `master` branch can help reduce confusion about which branch you are on. See deleting master on github for details.

Update the mirror of trunk

First make sure you have done *Linking your repository to the upstream repo*.

From time to time you should fetch the upstream (trunk) changes from github:

```
git fetch upstream
```

This will pull down any commits you don't have, and set the remote branches to point to the right commit. For example, 'trunk' is the branch referred to by (remote/branchname) `upstream/master` - and if there have been commits since you last checked, `upstream/master` will change after you do the fetch.

Make a new feature branch

When you are ready to make some changes to the code, you should start a new branch. Branches that are for a collection of related edits are often called 'feature branches'.

Making an new branch for each set of related changes will make it easier for someone reviewing your branch to see what you are doing.

Choose an informative name for the branch to remind yourself and the rest of us what the changes in the branch are for. For example `add-ability-to-fly`, or `buxfix-for-issue-42`.

```
# Update the mirror of trunk
git fetch upstream
# Make new feature branch starting at current trunk
git branch my-new-feature upstream/master
git checkout my-new-feature
```

Generally, you will want to keep your feature branches on your public github fork of rpe. To do this, you git push this new branch up to your github repo. Generally (if you followed the instructions in these pages, and by default), git will have a link to your github repo, called `origin`. You push up to your own repo on github with:

```
git push origin my-new-feature
```

In git >= 1.7 you can ensure that the link is correctly set by using the `--set-upstream` option:

```
git push --set-upstream origin my-new-feature
```

From now on git will know that `my-new-feature` is related to the `my-new-feature` branch in the github repo.

The editing workflow

Overview

```
# hack hack
git add my_new_file
git commit -am 'NF - some message'
git push
```

In more detail

1. Make some changes
2. See which files have changed with `git status` (see git status). You'll see a listing like this one:

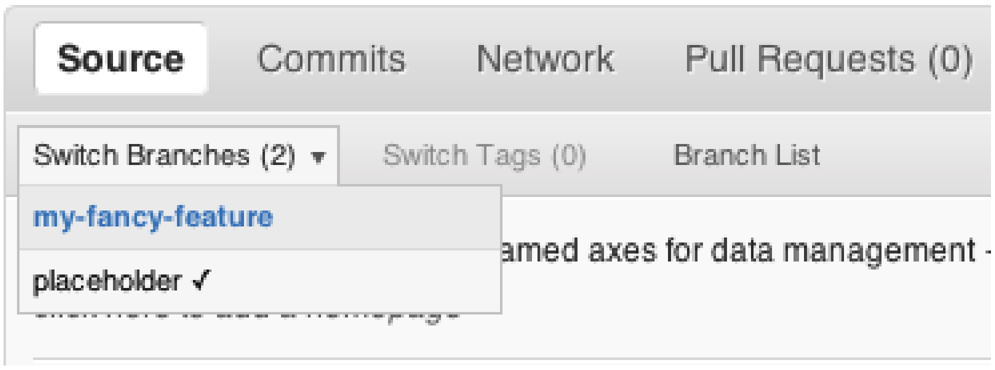
```
# On branch ny-new-feature
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#  modified:   README
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#  INSTALL
no changes added to commit (use "git add" and/or "git commit -a")
```

3. Check what the actual changes are with `git diff` (git diff).
4. Add any new files to version control `git add new_file_name` (see git add).
5. To commit all modified files into the local copy of your repo,, do `git commit -am 'A commit message'`. Note the `-am` options to `commit`. The `m` flag just signals that you're going to type a message on the command line. The `a` flag — you can just take on faith — or see why the -a flag? — and the helpful use-case description in the tangled working copy problem. The git commit manual page might also be useful.
6. To push the changes up to your forked repo on github, do a `git push` (see git push).

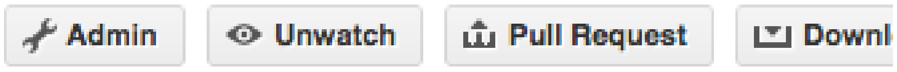
Ask for your changes to be reviewed or merged

When you are ready to ask for someone to review your code and consider a merge:

1. Go to the URL of your forked repo, say `http://github.com/your-user-name/rpe`.
2. Use the 'Switch Branches' dropdown menu near the top left of the page to select the branch with your changes:



3. Click on the 'Pull request' button:



Enter a title for the set of changes, and some explanation of what you've done. Say if there is anything you'd like particular attention for - like a complicated change or some code you are not happy with.

If you don't think your request is ready to be merged, just say so in your pull request message. This is still a good way of getting some preliminary code review.

Some other things you might want to do

Delete a branch on github

```
git checkout master
# delete branch locally
git branch -D my-unwanted-branch
# delete branch on github
git push origin :my-unwanted-branch
```

(Note the colon `:` before `test-branch`. See also: http://github.com/guides/remove-a-remote-branch

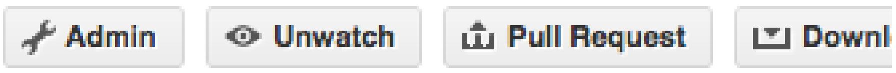
Several people sharing a single repository

If you want to work on some stuff with other people, where you are all committing into the same repository, or even the same branch, then just share it via github.

First fork rpe into your account, as from *Making your own copy (fork) of rpe*.

Then, go to your forked repository github page, say `http://github.com/your-user-name/rpe`

Click on the 'Admin' button, and add anyone else to the repo as a collaborator:



Now all those people can do:

```
git clone git@githhub.com:your-user-name/rpe.git
```

Remember that links starting with `git@` use the ssh protocol and are read-write; links starting with `git://` are read-only.

Your collaborators can then commit directly into that repo with the usual:

```
git commit -am 'ENH - much better code'
git push origin master # pushes directly into your repo
```

Explore your repository

To see a graphical representation of the repository branches and commits:

```
gitk --all
```

To see a linear list of commits for this branch:

```
git log
```

You can also look at the network graph visualizer for your github repo.

Finally the *Fancy log output* `lg` alias will give you a reasonable text-based graph of the repository.

Rebasing on trunk

Let's say you thought of some work you'd like to do. You *Update the mirror of trunk* and *Make a new feature branch* called `cool-feature`. At this stage trunk is at some commit, let's call it E. Now you make some new commits on your `cool-feature` branch, let's call them A, B, C. Maybe your changes take a while, or you come back to them after a while. In the meantime, trunk has progressed from commit E to commit (say) G:

```
      A---B---C cool-feature
     /
D---E---F---G trunk
```

At this stage you consider merging trunk into your feature branch, and you remember that this here page sternly advises you not to do that, because the history will get messy. Most of the time you can just ask for a review, and not worry that trunk has got a little ahead. But sometimes, the changes in trunk might affect your changes, and you need to harmonize them. In this situation you may prefer to do a rebase.

rebase takes your changes (A, B, C) and replays them as if they had been made to the current state of `trunk`. In other words, in this case, it takes the changes represented by A, B, C and replays them on top of G. After the rebase, your history will look like this:

```
              A'--B'--C' cool-feature
             /
D---E---F---G trunk
```

See rebase without tears for more detail.

To do a rebase on trunk:

```
# Update the mirror of trunk
git fetch upstream
# go to the feature branch
git checkout cool-feature
# make a backup in case you mess up
git branch tmp cool-feature
# rebase cool-feature onto trunk
git rebase --onto upstream/master upstream/master cool-feature
```

In this situation, where you are already on branch `cool-feature`, the last command can be written more succinctly as:

```
git rebase upstream/master
```

When all looks good you can delete your backup branch:

```
git branch -D tmp
```

If it doesn't look good you may need to have a look at *Recovering from mess-ups*.

If you have made changes to files that have also changed in trunk, this may generate merge conflicts that you need to resolve - see the git rebase man page for some instructions at the end of the "Description" section. There is some related help on merging in the git user manual - see resolving a merge.

Recovering from mess-ups

Sometimes, you mess up merges or rebases. Luckily, in git it is relatively straightforward to recover from such mistakes.

If you mess up during a rebase:

```
git rebase --abort
```

If you notice you messed up after the rebase:

```
# reset branch back to the saved point
git reset --hard tmp
```

If you forgot to make a backup branch:

```
# look at the reflog of the branch
git reflog show cool-feature

8630830 cool-feature@{0}: commit: BUG: io: close file handles immediately
278dd2a cool-feature@{1}: rebase finished: refs/heads/my-feature-branch onto␣
↪11ee694744f2552d
26aa21a cool-feature@{2}: commit: BUG: lib: make seek_gzip_factory not leak gzip obj
...
```

```
# reset the branch to where it was before the botched rebase
git reset --hard cool-feature@{2}
```

Rewriting commit history

Note: Do this only for your own feature branches.

There's an embarrassing typo in a commit you made? Or perhaps the you made several false starts you would like the posterity not to see.

This can be done via *interactive rebasing*.

Suppose that the commit history looks like this:

```
git log --oneline
eadc391 Fix some remaining bugs
a815645 Modify it so that it works
2dec1ac Fix a few bugs + disable
13d7934 First implementation
6ad92e5 * masked is now an instance of a new object, MaskedConstant
29001ed Add pre-nep for a copule of structured_array_extensions.
...
```

and `6ad92e5` is the last commit in the `cool-feature` branch. Suppose we want to make the following changes:

- Rewrite the commit message for `13d7934` to something more sensible.
- Combine the commits `2dec1ac`, `a815645`, `eadc391` into a single one.

We do as follows:

```
# make a backup of the current state
git branch tmp HEAD
# interactive rebase
git rebase -i 6ad92e5
```

This will open an editor with the following text in it:

```
pick 13d7934 First implementation
pick 2dec1ac Fix a few bugs + disable
pick a815645 Modify it so that it works
pick eadc391 Fix some remaining bugs

# Rebase 6ad92e5..eadc391 onto 6ad92e5
#
# Commands:
#  p, pick = use commit
#  r, reword = use commit, but edit the commit message
#  e, edit = use commit, but stop for amending
#  s, squash = use commit, but meld into previous commit
#  f, fixup = like "squash", but discard this commit's log message
#
# If you remove a line here THAT COMMIT WILL BE LOST.
# However, if you remove everything, the rebase will be aborted.
#
```

To achieve what we want, we will make the following changes to it:

```
r 13d7934 First implementation
pick 2dec1ac Fix a few bugs + disable
f a815645 Modify it so that it works
f eadc391 Fix some remaining bugs
```

This means that (i) we want to edit the commit message for `13d7934`, and (ii) collapse the last three commits into one. Now we save and quit the editor.

Git will then immediately bring up an editor for editing the commit message. After revising it, we get the output:

```
[detached HEAD 721fc64] FOO: First implementation
 2 files changed, 199 insertions(+), 66 deletions(-)
[detached HEAD 0f22701] Fix a few bugs + disable
 1 files changed, 79 insertions(+), 61 deletions(-)
Successfully rebased and updated refs/heads/my-feature-branch.
```

and the history looks now like this:

```
0f22701 Fix a few bugs + disable
721fc64 ENH: Sophisticated feature
6ad92e5 * masked is now an instance of a new object, MaskedConstant
```

If it went wrong, recovery is again possible as explained *above*.

Maintainer workflow

This page is for maintainers — those of us who merge our own or other peoples' changes into the upstream repository.

Being as how you're a maintainer, you are completely on top of the basic stuff in *Development workflow*.

The instructions in *Linking your repository to the upstream repo* add a remote that has read-only access to the upstream repo. Being a maintainer, you've got read-write access.

It's good to have your upstream remote have a scary name, to remind you that it's a read-write remote:

```
git remote add upstream-rw git@github.com:aopp-pred/rpe.git
git fetch upstream-rw
```

Integrating changes

Let's say you have some changes that need to go into trunk (`upstream-rw/master`).

The changes are in some branch that you are currently on. For example, you are looking at someone's changes like this:

```
git remote add someone git://github.com/someone/rpe.git
git fetch someone
git branch cool-feature --track someone/cool-feature
git checkout cool-feature
```

So now you are on the branch with the changes to be incorporated upstream. The rest of this section assumes you are on this branch.

A few commits

If there are only a few commits, consider rebasing to upstream:

```
# Fetch upstream changes
git fetch upstream-rw
# rebase
git rebase upstream-rw/master
```

Remember that, if you do a rebase, and push that, you'll have to close any github pull requests manually, because github will not be able to detect the changes have already been merged.

A long series of commits

If there are a longer series of related commits, consider a merge instead:

```
git fetch upstream-rw
git merge --no-ff upstream-rw/master
```

The merge will be detected by github, and should close any related pull requests automatically.

Note the `--no-ff` above. This forces git to make a merge commit, rather than doing a fast-forward, so that these set of commits branch off trunk then rejoin the main history with a merge, rather than appearing to have been made directly on top of trunk.

Check the history

Now, in either case, you should check that the history is sensible and you have the right commits:

```
git log --oneline --graph
git log -p upstream-rw/master..
```

The first line above just shows the history in a compact way, with a text representation of the history graph. The second line shows the log of commits excluding those that can be reached from trunk (`upstream-rw/master`), and including those that can be reached from current HEAD (implied with the `..` at the end). So, it shows the commits unique to this branch compared to trunk. The `-p` option shows the diff for these commits in patch form.

Push to trunk

```
git push upstream-rw my-new-feature:master
```

This pushes the `my-new-feature` branch in this repository to the `master` branch in the `upstream-rw` repository.

git resources

Tutorials and summaries

- github help has an excellent series of how-to guides.
- learn.github has an excellent series of tutorials

- The pro git book is a good in-depth book on git.
- A git cheat sheet is a page giving summaries of common commands.
- The git user manual
- The git tutorial
- The git community book
- git ready — a nice series of tutorials
- git casts — video snippets giving git how-tos.
- git magic — extended introduction with intermediate detail
- The git parable is an easy read explaining the concepts behind git.
- git foundation expands on the git parable.
- Fernando Perez' git page — Fernando's git page — many links and tips
- A good but technical page on git concepts
- git svn crash course: git for those of us used to subversion

Advanced git workflow

There are many ways of working with git; here are some posts on the rules of thumb that other projects have come up with:

- Linus Torvalds on git management
- Linus Torvalds on linux git workflow . Summary; use the git tools to make the history of your edits as clean as possible; merge from upstream edits as little as possible in branches where you are doing active development.

Manual pages online

You can get these on your own machine with (e.g) `git help push` or (same thing) `git push --help`, but, for convenience, here are the online manual pages for some common commands:

- git add
- git branch
- git checkout
- git clone
- git commit
- git config
- git diff
- git log
- git pull
- git push
- git remote
- git status

Code Generator

A large amount of the code included in the Fortran library is constructed programmatically by a code generator. The code generator is a small library written in Python, that uses templates to generate fortran code, as well as front-ends to generate the required functions, subroutines and interface blocks for all overloaded operators and instrinsic functions.

Structure of the Code Generator

The code generator consists of a small Python library for code generation, 2 configuation files for determining what code is generated, and two driver programs to build the code. The code generator itself resides within the top-level `generator/` directory. At the top level is a Makefile used to run the code generator, the directory containing generated code

Configuration files

The code generator driver programs require input files that define which overloaded operators and intrinsic functions need to be constructed. These files can be found in the `generator/configs/` directory. The file `operators.json` defines overloaded operators and the file `intrinsics.json` defines the overloaded intrinsic functions; the files are in JSON format.

Python components

The python components are located in the subdirectory `generator/python`. In this directory are two Python programs `generate_intrinsics.py` and `generate_operators.py`. These programs take as input a JSON configuration file and output Fortran code files.

Also in this directory is another directory named `rpgen/`, this is the Python library for code generation.

Using the Code Generator

The basic workflow for using the code generator is detailed below, and can be summarised as:

1. Make some change to the code to be generated, either by modifying the JSON configuration files or by modifying templates or even the generator library itself.
2. Run the generator.
3. Integrate the generated source code into the main Fortran library.

Defining the code to be generated

What the code generator produces can be controlled either by configuration files, or modifying the generator itself. Modifying configuration files is covered in *Adding New Operators or Intrinsic Functions*.

Running the generator

The generator can be run from the top-level directory using the Makefile:

```
make -C generator
```

This will produce 4 files in the directory `generator/generated/`:

- `interface_operators.i`: interface blocks for overloaded operators.
- `implementation_operators.f90`: overloaded operator implementations.
- `interface_intrinsics.i`: interface blocks for overloaded intrinsic functions.
- `implementation_intrinsics.f90`: overloaded intrinsic function implementations.

Despite the file extensions, all 4 files contain Fortran code, the extensions are simply part of a naming convention.

Integrating generated code into the Fortran library

Once you have generated new files, you will want to integrate the generated code into the main Fortran library.

Warning: Make sure you carefully check the generated code is correct before proceeding. It is worthwhile looking at both the generator output, and the difference between each output and the existing file in `src/include/`, for example:

```
diff generated/implementation_operators.f90 src/include/implementation_operators.f90
```

This way you can be sure that you have achieved the change you wanted without changing something else you didn't expect to change.

To integrate the generated code all you need to do is copy/move the generated files from `generator/generated/` to `src/include/`:

```
cp generator/generated/* src/include/
```

Once you have done this you should rebuild the library:

```
make clean
make
```

Now you can run the tests and verify that the new code works as expected:

```
cd tests
make test
```

Adding New Operators or Intrinsic Functions

Extending the emulator by adding new overloaded operators or intrinsic functions is fairly straightforward, and usually requires only editing configuration files.

Adding a new intrinsic function

In this example we'll add the GAMMA gfortran intrinsic to the emulator, allowing us to evaluate the Γ function with a reduced precision input.

Overview of existing definitions

We'll need to add a new entry to the `intrinsics.json` file in `generator/configs/` to define this new function, but first let's take a look at some of the existing function definitions:

```
{"intrinsics":
    [

        {"epsilon":
            {
                "return_type": "rpe_var",
                "interface_types": ["1argscalar"]
            }
        },

        {"floor":
            {
                "return_type": "integer",
                "interface_types": ["1argelemental"]
            }
        },

    ]
}
```

Each function is defined as a JSON object, with a name that corresponds to the name of the intrinsic function, and two attributes that determine the return type of the function and the types of interface the function has.

In the above snippet you can see that the function `epsilon` has a return type of `"rpe_var"`, which corresponds to an *`rpe_var`* instance. The function `floor` has a return type of `"integer"`, which just corresponds to a normal Fortran integer type. A complete list of types that can be used in configuartion files can be found in *Type names and variables*.

The interface type is a concept used within the code generator to work out what kind of code it should produce. You can see that the function `epsilon` has interface type `"1argscalar"` which corresponds to a function that takes one scalar as an argument. The function `floor` has interface type `"1argelemental"` which corresponds again to a function that takes one argument, but this time the function is elemental meaning it can take a scalar or an array as input, and operate element-wise on array inputs returning an array output. A function can have more than one interface type if it has multiple interfaces. See *Intrinsic function interface types* for a list of all intrinsic interface types.

Writing the new definition

From the gfortran GAMMA documentation we can see that our new function should accept a single input of a reduced precision number, and return a single output which will also be a reduced precision number. We can also see that the function should be elemental, meaning it can be applied element-wise to an array of input values. Therefore our definition in `intrinsics.json` should look like this:

```
{"gamma":
    {
        "return_type": "rpe_var",
        "interface_types": ["1argelemental"]
    }
}
```

Generating the code

Now that you have created the definition for GAMMA you need to use the generator to actually write the code. The simplest way to do this is by using the Makefile in the `generator/` directory:

```
cd generator/
make
```

This command will generatre a new set of files in the `generated/` subdirectory, and you can inspect these to verify that correct code was written for a GAMMA imnplementation on reduced precision types. First lets look at `interface_intrinsics.i`, it now has these extra lines:

```
PUBLIC :: gamma
INTERFACE gamma
    MODULE PROCEDURE gamma_rpe
END INTERFACE gamma
```

These lines define a public interface for a function `gamma`, with one member function called `gamma_rpe`. Now let's look in the newly generated `implementation_intrinsics.f90` to see the implementation of `gamma_rpe`:

```
!-------------------------------------------------------------------
! Overloaded definitions for 'gamma':
!

ELEMENTAL FUNCTION gamma_rpe (a) RESULT (x)
    TYPE(rpe_var), INTENT(IN) :: a
    TYPE(rpe_var) :: x
    x%sbits = significand_bits(a)
    x = GAMMA(a%val)
END FUNCTION gamma_rpe
```

The generated implementation consists of a single elemental function definition accepting any *`rpe_var`* type and returns an *`rpe_var`* type. The body of the function is simple, it simply sets the nmumber of bits in the significand of the return value to match the input, then calls the normal Fortran `GAMMA` intrinsic with the real value cointained by the reduced precision number as input and stores the result in the output variable `x`. The precision of the return value `x` is reduced by the assignment operation.

To include this code in a build of the library simply follow the instructions in *Integrating generated code into the Fortran library*.

Adding a new operator

The process of adding a new operator proceeds much like *Adding a new intrinsic function*, except with a different configuration file and different JSON attributes. In this example we'll pretend that we don't already have a `**` operator and implement one.

The JSON configuration for operators is the `operators.json` file in `generator/configs/`. An operator definition looks like this:

```
{"<operator-name>":
    {
        "operator": "<operator-symbol>",
        "return_type": "<return-type>",
        "operator_categories": ["<categories>"]
    }
}
```

In this example `<name>` is the name of the operator, in our case this will be `"pow"`; `<operator-symbol` is the symbol used to represent the operator, which in our case will be `"**"`; `<return-type>` is just the type that will be returned by the operator, in this case we want to return a reduced precision value so we will use `"rpe_var"` as the return type. The value supplied for `"operator_categories"` is a list of the categories this operator falls into.

There are only 2 categories available, `"unary"` for unary operators and `"binary"` for binary operators. The list of categories can contain one or both of these values if appropriate, but is our case exponentiation is a binary operator so we'll supply the one value `["binary"]`.

Generating the code for the new operator just requires running the Makefile in `generator/`:

```
cd generator/
make
```

Let's take a look at what was generated in the `generated/` subdirectory, firstly in the `interface_operators.i` file:

```
PUBLIC :: OPERATOR(**)
INTERFACE OPERATOR(**)
    MODULE PROCEDURE pow_rpe_rpe
    MODULE PROCEDURE pow_rpe_integer
    MODULE PROCEDURE pow_rpe_long
    MODULE PROCEDURE pow_rpe_real
    MODULE PROCEDURE pow_rpe_realalt
    MODULE PROCEDURE pow_integer_rpe
    MODULE PROCEDURE pow_long_rpe
    MODULE PROCEDURE pow_real_rpe
    MODULE PROCEDURE pow_realalt_rpe
END INTERFACE OPERATOR(**)
```

This defines a public interface for the `**` operator, which contains 9 member functions. These functions deal with all possible input combinations for the operator. Now let's look at how these operators are defined in the generated `implementation_operators.f90` file, we'll just show a few of the 9 definitions to get a feel for what is generated:

```
!-------------------------------------------------------------------
! Overloaded definitions for (**):
!

ELEMENTAL FUNCTION pow_rpe_rpe (x, y) RESULT (z)
    TYPE(rpe_var), INTENT(IN) :: x
    TYPE(rpe_var), INTENT(IN) :: y
    TYPE(rpe_var) :: z
    z%sbits = MAX(significand_bits(x), significand_bits(y))
    z = x%get_value() ** y%get_value()
END FUNCTION pow_rpe_rpe

...

ELEMENTAL FUNCTION pow_rpe_real (x, y) RESULT (z)
    TYPE(rpe_var), INTENT(IN) :: x
    REAL(KIND=RPE_REAL_KIND), INTENT(IN) :: y
    TYPE(rpe_var) :: z
    z%sbits = MAX(significand_bits(x), significand_bits(y))
    z = x%get_value() ** y
END FUNCTION pow_rpe_real

...

ELEMENTAL FUNCTION pow_real_rpe (x, y) RESULT (z)
    REAL(KIND=RPE_REAL_KIND), INTENT(IN) :: x
    TYPE(rpe_var), INTENT(IN) :: y
    TYPE(rpe_var) :: z
    z%sbits = MAX(significand_bits(x), significand_bits(y))
    z = x ** y%get_value()
END FUNCTION pow_real_rpe
```

The first definition defines how the `**` operator can be applied to two *rpe_var* instances. It can operate on *rpe_var* types for each argument and returns an *rpe_var* instance. The number of bits in the significand of the result is set to the larger of the number of bits in the significands of the inputs, the calculation is then done in full precision and reduced to the specified precision on assignment to the return value `z`.

The other two functions do something very similar, except they operate on inputs of one reduced precision type and one real number type, the first raising a reduced precision number to the power of a real number, and the second raising a real number to the power of a reduced precision number.

Now that the code for the new operator has been generated and checked it can be included in a build of the library by following the instructions in *Integrating generated code into the Fortran library*.

Reference

Type names and variables

Type name (JSON)	Generator Type variable	Fortran type
`"logical"`	`rpgen.types.LOGICAL`	`LOGICAL`
`"integer"`	`rpgen.types.INTEGER`	`INTEGER`
`"long"`	`rpgen.types.LONG`	`INTEGER(KIND=8)`
`"real"`	`rpgen.types.REAL`	`REAL(KIND=RPE_REAL_KIND)`
`"realalt"`	`rpgen.types.REALALT`	`REAL(KIND=RPE_ALTERNATE_KIND)`
`"rpe_var"`	`rpgen.types.RPE_VAR`	`TYPE(rpe_var)`

Operator categories

Operator category (JSON)	Definition
`"unary"`	A unary operator with one input and one output.
`"binary"`	A binary operator with two inputs and one output.

Intrinsic function interface types

Interface name (JSON)	Definition
`"1argscalar"`	A function with one scalar argument.
`"1argelemental"`	An elemental function with one scalar or array argument.
`"2argelemental"`	An elemental function with two scalar or array arguments.
`"1arrayarg"`	A function with one array argument and a scalalr return value.
`"multiarg"`	An elemental function with multiple arguments.

Making a release

This is a short guide on how to make a new release of the rpe library. There are several stages to making a feature release which are outlined below. Bugfix releases are simpler and don't require making a new branch, the first step is only relevant to feature releases. Making a release requires developer access to the rpe repository.

Cutting a release branch

The first step is to create a branch for a feature release. Each major and minor release version gets its own branch, but bugfixes releases do not (kind of obvious when you think about it but worth saying anyway). Release branches should come off the master branch when you are ready to start preparing the release. Release branches should be named `v{major}.{minor}.x`, so the release branch for version 6.0 would be `v6.0.x` and for version 6.1 it would be `v6.1.x`. The `x` in the version number represents the fact that all bugfix releases will come from the same branch, for example the `v6.1.x` branch will be created for `v6.1.0` but will also be used for `v6.1.1` if it is required, along with any further bugfix releases. The release branch should be pushed to the github organisation (assumed to be the remote named upstream):

```
git push upstream v{major}.{minor}.x
```

Once the release branch has been cut it is a good time to update the `VERSION` file on the **master** branch. You should set it to the major and minor numbers of the anticipated next release after the one you are making (it doesn't matter if plans change later), with the `dev0` suffix (e.g., for next release 6.2 use `v6.2.dev0`). Doing this ensures that people contributing to the master branch know which release their work will likely end up in.

Set version and tagging

The release branch should receive only minor changes (no new features, those should have been committed to the master branch before the release branch was cut). Once you have all the changes required for the release you should set the version number in the file `VERSION` in the repository root. The version for a feature release will be `v{major}.{minor}.0`. Commit the changes to the version file. Push the changes to the release branch (usually via pull request).

Once you have set the version you should make a tag. The tag will be used to indentify the commit where the release was made, and will be used to contruct a release on Github. Tagging should use the following command:

```
git tag -a v{major}.{minor}.{bugfix} -m "Version {major}.{minor}.{bugfix}"
```

You can now push the tag to the upstream:

```
git push --tags upstream
```

You should now reset the version number on the release branch to the next dev version. For example, if you just tagged v6.1.0, you should set the version in the file `VERSION` to `v6.1.dev1`. Commit this change and push it to the release branch (usually via PR).

Making a Github release

Once you have pushed the tag you can go to the Github repository, click the releases icon in the repository summary, and click new release. Type the name of the tag you made into the box for release name and Github will automatically create the release from the existing tag. Write a summary of the release in the text box (a simple changelog is suggested) and create the release. Creating a release will automatically trigger a new build of the documentation on readthedocs and a new Zenodo entry.

Merge-back changes

After you have finalised the release you should merge the release branch back into the master branch. This should be done via a pull request. The way to do it is create a new branch locally, do a non-fast-forward merge of the release branch into it, fixing any merge conflicts, then make a PR from this branch into master:

```
git fetch upstream
git checkout -b mergeback-v{major}.{minor}.x upstream/master
git merge --no-ff upstream v{major}.{minor}.x
# Fix merge conflicts, the file VERSION will conflict, there may be others
```

Merging back should be done after all releases, including bugfix releases

CHAPTER 7

API Reference

API documentation for the Fortran library:

RPE API: `rp_emulator.mod`

Derived types

type `rpe_var`

A type representing a reduced-precision floating-point number. This type stores the number it represents internally.

Type fields

- `%` **`sbits`** *[INTEGER]* :: Number of bits in the significand of the floating-point number
- `%` **`val`** *[REAL,KIND=RPE_REAL_KIND]* :: The real value stored within the instance.

User-callable procedures

function `rpe_literal` (*x*, *n*)

Construct an `rpe_var` instance from a value. Optionally a number of significand bits used to store the value may be given. The value will be truncated before storage. A typical usage of this constructor is to support reduced precision literal values without having to declare extra variables:

```
type(rpe_var) :: a, b, c, d, e

RPE_DEFAULT_SBITS = 10

! variables a, b and c have a 10-bit significands:
a = 5.00
b = 7.23
c = 21.12
```

```
! The literals 7.29e-5, 3.14 and 18.17 have full precision
! (a 23-bit significand):
d = 7.29e-5 * a + 3.14 * b + 18.17 * c  ! d = 406.5

! The literals are replaced by rpe_var instances with 10-bit significands:
e = rpe_literal(7.29e-5) * a + rpe_literal(3.14) * b + rpe_literal(18.17) * c  !␣
↪e = 406.75
```

Parameters

- **x** *[IN]* :: A real or integer value to store in the resulting *rpe_var* instance.
- **n** *[integer,IN,OPTIONAL]* :: An optional number of significand bits used to represent the number, equivalent of setting the `sbits` attribute of an *rpe_var* instance. If not specified then the resulting *rpe_var* will use the default precision specified by *RPE_DEFAULT_SBITS*.

Note: If you use this to create an *rpe_var* instance and then assign the result to another instance, only the value will be copied:

```
! An rpe_var instance with a 13 bit significand:
type(rpe_var) :: a
a%sbits = 13

! Construct an rpe_type instance with a 15-bit significand and assign to a,
! the value will be truncated to 13 bits and the variable a will continue
! to store only 13 significand bits.
a = rpe_literal(1.23456789, 15)
```

---

subroutine `apply_truncation`(*rpe*) *[elemental]*

Apply the required truncation to the value stored within an *rpe_var* instance. Operates on the input in-place, modifying its value. The truncation is determined by the `sbits` attribute of the *rpe_var* instance, if this is not set then the value of *RPE_DEFAULT_SBITS*.

Parameters **rpe** [rpe_var,*INOUT*] :: The *rpe_var* instance to alter the precision of.

function `significand_bits`(*x*) *[elemental]*

Determine the number of significand bits being used by the input types. For inputs that are *rpe_var* instances this function returns the number of significand bits in use by the reduced-precision number. For real numbers it will return either 23 for single-precision inputs or 52 for double-precision inputs. For all other input types the result will be zero.

Parameters **x** *[IN]* :: Any Fortran type.

Variables

`RPE_ACTIVE` *[LOGICAL,default=.TRUE.]*

Logical value determining whether emulation is on or off. If set to `.FALSE.` then calls to *apply_truncation()* will have no effect and all operations will be carried out at full precision.

`RPE_DEFAULT_SBITS` *[INTEGER,default=52]*

The default number of bits used in the significand of an *rpe_var* instance when not explicitly specified. This takes effect internally when determining precision levels, but does not bind an *rpe_var* instance to a particular precision level (doesn't set `rpe_var%sbits`).

`RPE_IEEE_HALF` *[LOGICAL,default=.FALSE.]*
Logical value determining if IEEE half-precision emulation is turned on. If set to `.TRUE.` and a 10-bit significand is being emulated the emulator will additionally impose range constraints when applying truncation:

- Values that overflow IEEE half-precision will lead to real overflows with a corresponding floating-point overflow exception.
- Values out of the lower range of IEEE half-precision will be denormalised.

This option only affects the emulation when emulating a 10-bit significand.

Parameters

`RPE_DOUBLE_KIND` *[INTEGER]*
The kind number for double precision real types.

`RPE_SINGLE_KIND` *[INTEGER]*
The kind number for single precision real types.

`RPE_REAL_KIND` *[INTEGER]*
The kind number of the real-values held by reduced precision types. This is a reference to *`RPE_DOUBLE_KIND`*, but could be changed (in source) to be *`RPE_SINGLE_KIND`*.

`RPE_ALTERNATE_KIND` *[INTEGER]*
The kind number of an alternate type of real-value. This is a reference to *`RPE_SINGLE_KIND`*, but can be changed (in source) if the value referenced by *`RPE_REAL_KIND`* is changed.

CHAPTER 8

Downloads

Source code downloads for released versions, see the *Changelog* for details on each version.

Filename Released	Summary
`rpe-5.0.0.tar.gz` 20 Sep 2016	version 5.0.0 source
`rpe-4.1.1.tar.gz` 24 Aug 2016	version 4.1.1 source
`rpe-4.1.0.tar.gz` 22 Aug 2016	version 4.1.0 source
`rpe-4.0.0.tar.gz` 05 Jan 2016	version 4.0.0 source
`rpe-3.1.1.tar.gz` 04 Nov 2015	version 3.1.1 source
`rpe-3.0.1.tar.gz` 04 Nov 2015	version 3.0.1 source
`rpe-2.0.1.tar.gz` 04 Nov 2015	version 2.0.1 source
`rpe-1.0.1.tar.gz` 04 Nov 2015	version 1.0.1 source

Changelog

v5.0.x

Release v5.0.0

Date 20 September 2016

- Change summary

    - The default value for `RPE_DEFAULT_SBITS` has changed to 52, previously it was 23. This change means that by default all `rpe_var` types behave like double precision `real` types.
    - The upper limit for the value of an IEEE half-precision number has been corrected to 65504. This value was previously too small (32768) resulting in an over-conservative representation of IEEE half-precision values. This change only affects code running with the `RPE_IEEE_HALF` option turned on.
    - The `huge` intrinsic has been reimplemented for `rpe_var` types. It now returns the largest value with an 11-bit exponent and the number of significand bits in its input. It also behaves correctly when using a 10-bit significand and `RPE_IEEE_HALF = .true.`, returning the largest number with a 5-bit exponent and 10-bit significand (65504).

- Incompatibilities
    - The rounding mode has changed to be compliant with IEEE 754. The new mode will likely give different (but more realistic) results. The new behaviour is the same as that obtained in v4.1 with the option `RPE_IEEE_ROUNDING = .true.`.
    - The option `RPE_IEEE_ROUNDING` has been removed, the new rounding behaviour is equivalent to `RPE_IEEE_ROUNDING = .true.`. There is no option to change to the behaviour of `RPE_IEEE_ROUNDING = .false.`.

v4.1.x

Release v4.1.1

Date 24 August 2016

Add deprecation notices to API documentation. There are no changes to the source code and no need to upgrade from v4.1.0.

Release v4.1.0

Date 22 August 2016

Adds an IEEE 754 compliant rounding scheme and new functionality for explicit handling of literal floating-point values.

- Features
    - A new (currently opt-in) IEEE 754 compliant rounding mode, activated by setting the module variable `RPE_IEEE_ROUNDING = .true.`. This option is provided to help manage a transition to IEEE 754 compliant rounding in a future release. Eventually this option will be removed and IEEE 754 compliant rounding will become the default and only rounding mode.
    - A new helper function `rpe_literal` is provided to help write correct reduced-precision code that contains numeric literals.
- Deprecations
    - The current rounding mode (round to nearest) is deprecated. Future releases will use the IEEE 754 compliant rounding mode. Users should set `RPE_IEEE_ROUNDING = .true.` to get the new rounding behaviour. We recommend the IEEE 754 rounding mode to ensure best results.

v4.0.x

Release v4.0.0

Date 5 January 2016

This release is a major overhaul of the code with the aim of making it more portable and reliable.

- Features
  - Compatible with Intel Fortran compilers. The code may work with other compilers too, but only Intel and GNU are tested currently.
- Incompatibilities: This release is not compatible with version 3 or below.
  - Removed the `rpe_shadow` derived type.
  - Removed abstract base class `rpe_type`, the only user type is now `rpe_var`.
  - Removed getter and setter methods, the value of an `rpe_var` instance is now accessed directly using the `%val` attribute.

v3.1.x

Release v3.1.1

Date 3 November 2015

- License code under the Apache 2.0 license.

Release v3.1.0

Date 16 October 2015

- Support for IEEE half-precision emulation via the `RPE_IEEE_HALF` module variable.

v3.0.x

Release v3.0.1

Date 4 November 2015

- License code under the Apache 2.0 license.

Release v3.0.1

Date 21 August 2015

- Features
  - Support for different precision levels in different variables:
    * You can set the `%sbits` attribute of any `rpe_type` instance to define the number of significand bits used by that variable.
  - A set of unit tests is included to help us ensure the emulator core is robust.
  - HTML documentation is included with the source (requires Sphinx to build).
- Incompatibilities
  - This version is incompatible with the v2.0.x series.
  - The public API is now smaller, including only the required parts of the library.
  - Module variables renamed: `RPE_BITS` -> `RPE_DEFAULT_SBITS`
  - Reduction of precision subroutine renamed: `reduce_precision` -> `apply_truncation`
  - Internal differences to support mixed precision may cause different results to previous versions.

v2.0.x

Release 2.0.1

Date 3 November 2015

- License code under the Apache 2.0 license.

Release 2.0.0

Date 29 July 2015

- Features
  - Reduce precision on assignment:
    - The precision of the value held within an `rpe_type` instance is reduced whenever a value is assigned, meaning an `rpe_var` instance cannot ever store a full precision value, and an `rpe_shadow` type will always store a reduced precision value when it has been assigned to directly (but one can of course assign a full precision value to the variable it is shadowing and have that value retained).
    - Explicit calls to `reduce_precision` are no longer required in any overloaded operators or intrinsic routines, as the reduction of precision will be performed implicitly on assignment of the result.
- Incompatibilities
  - The change from explicit reduction of precision within overloaded operators and intrinsics will likely cause the emulator to return different results than the v1.0.x series.

v1.0.x

Release v1.0.1

Date 4 November 2015

- License code under the Apache 2.0 license.

Release 1.0.0

Date 28 July 2015

- Features
  - Initial version used operationally for experiments.

Reduced Precision Emulator

This software provides a Fortran library for emulating the effect of low-precision real numbers. It is intended to be used as an experimental tool to understand the effect of reduced precision within numerical simulations.

Citation

An article describing the emulator is published in Geoscientific Model Development and can be cited as:

> Dawson, A. and P. D. Düben (2017): rpe v5: An emulator for reduced floating-point precision in large numerical simulations. *Geosci. Model Dev.*, 10, 2221-2230, doi: 10.5194/gmd-10-2221-2017.

The software itself has a doi for all released versions from v5.0.0, produced automatically using Zenodo. The record for the most recent release is here.

Index

A

apply_truncation() (fortran subroutine), **44**

R

RPE_ACTIVE (fortran variable), **44**
RPE_ALTERNATE_KIND (fortran variable), **45**
RPE_DEFAULT_SBITS (fortran variable), **44**
RPE_DOUBLE_KIND (fortran variable), **45**
RPE_IEEE_HALF (fortran variable), **44**
rpe_literal() (fortran function), **43**
RPE_REAL_KIND (fortran variable), **45**
RPE_SINGLE_KIND (fortran variable), **45**
rpe_var (fortran type), **43**

S

significand_bits() (fortran function), **44**