Vtech CMCC

Contact | Mechanics | Computing | Consulting

P.O. Box 27097
3003 LB Rotterdam
The Netherlands

info@vtech-cmcc.nl

Programmer guide for CONTACT, Getting started with the source code

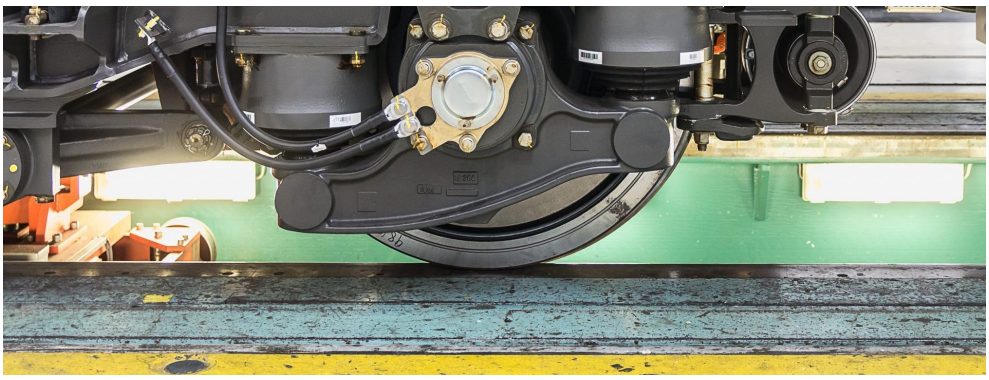
Technical Report 23-01, version 'open-src'

Date

June 16, 2023

Author(s)

Dr.ir. E.A.H. Vollebregt



Chapter 1

Introduction

This document gives a quick introduction to the open-source version of the CONTACT software. A diverse set of aspects is covered, including some general programming advice, some background on Fortran, and discussion of the main aspects of the CONTACT program itself.

This document could be used to start a Wiki, for easier further elaboration.

Chapter 2

Top ten advice for scientific programming

Here's VORtech's top-10 guidelines for scientific programming. Created in 2010, but still fairly up to date.

1. Above all, strive for clarity of the code. Complexity is the worst enemy when your programs get bigger.
2. Use a version management tool.
3. Use good coding standards. Uniformity is more important than making your parts look nice according to your own standards.
4. Automate testing, so that you can test often.
5. Get to know the sensitivity of your calculations: distinguish noise due to round-off from errors in the code.
6. Make a design.
7. Develop incrementally.
8. Create visualization and debugging facilities.
9. Don't optimize until necessary.
10. Use an issue management tool.

Additional practices that are recommended are as follows:

11. Think before you start: what you need, how to do it, how to test if it's working. *Shift left*. The cost of errors increases dramatically the longer they are left in the system.
12. Lean on the compiler to detect programming errors.

13. Don't save on the build system. Errors due to parts not being recompiled or compiled differently are particularly difficult to trace.

14. *Boy scout rule*: leave things better than you found them. Polish, clean up the parts that you studied.

A nice book with lots of advice from Google: [1], and another one collecting wisdom of many experienced programmers: [2].

Chapter 3

Programming on CONTACT

3.1 Development environment

Vollebregt does not use an Integrated Development Environment (IDE) such as Visual Studio. Instead, he works with an editor (GVIM) and makefiles.

- CONTACT is programmed mostly using Fortran, with some bits in C and the CONTACT GUI programmed in Java.
- Vollebregt uses Matlab a lot, some scripting in Perl, and some programming in Python.
- The main compiler used is the one from Intel, with the corresponding facilities of the Intel Math Kernel Library (MKL).
- Separate makefiles are maintained for Windows (using `nmake`) and Linux (GNU `make`), set up in a generic way, with configuration options hidden in include files.

3.2 Makefiles

Makefiles are used to automate compilation. This uses a specific (complex) language of "targets" and "dependencies". Only those targets are compiled (anew) that are "out of date", using the rules and commands defined in the makefile.

We use separate makefiles for Windows (using nmake) and Linux (GNU make). Each makefile can generate multiple outputs for the stand-alone program or the CONTACT library, with or without OpenMP (multithreading), with different options for debugging or naming of symbols (with/without underscores). We also used to create separate versions for 64bit and 32bit platforms, but the latter are no longer used.

3.3 Coding standards

Coding standards help to make code more uniform and enforce practices to make the code easier to understand (best practice 3, page 3).

- The screen width is set to 110 characters. No line should be longer than that.
- Indentation (tabs) goes with 3 characters.
- Using `implicit none` all of the time (best practice 12, page 3).
- Use lots of white space, to facilitate identifying the separate constituents of compound statements.
- Avoid uppercase characters, except for constants.
- Use underscores in variable names with different parts.
- Type names start with `t_`, e.g. `t_grid`, defined in the module `m_grids.f90`. The prefix `m_` is used for modules, `p_` for pointers.
- Subroutines specific for a derived type start with the type it is used for, e.g. `grid_copy`.
- Relational operators `.lt.`, `.le.`, `.eq.`, etc. are used instead of the alternative forms <=, <, ==. This is a subjective, personal preference of Vollebregt, imposed on others for the uniformity of the code.

3.4 Conditional compilation

Conditional compilation is used to make different executables from the same source code.

- Platform dependence, e.g.

```
#if defined _WIN32 || defined _WIN64
   integer,          parameter :: platform = plat_win
   character(len=1), parameter :: path_sep = '\\'
#else
   integer,          parameter :: platform = plat_lnx
   character(len=1), parameter :: path_sep = '/'
#endif
```

- Compiling with one or another compiler:

```
#ifdef PLATF_ppc
   icount = IARGC()
#else
   icount = command_argument_count()
#endif
```

- Compiling with or without OpenMP (multithreading):

```
#ifdef _OPENMP
   iparll = omp_in_parallel()
#else
   iparll = 0
#endif
```

- Compiling with or without certain features:

```
#ifdef WITH_MKLFFT
   use mkl_dfti
#endif
```

Some of the variables used are set automatically by the compiler (`_WIN32`, `_OPENMP`), others are set using compiler options (`-DWITH_MKLFFT`).

3.5 (Semi-)automated testing

Scripts are used to run a series of tests, one by one, and pull up a diff of the new and reference output (best practice 4, page 3). Especially `cleanup.pl`, `run_tests.pl`, and `set_refout.pl` in the `examples` folder (programmed in Perl).

It is desired to reduce the number of scripting languages used, reprogramming Perl and shell scripts in Python.

We go through the differences manually, judging whether these are acceptable or not. We often find slight differences in the numerical outputs due to round-off errors. Differences between Windows and Linux or just between two runs of the compiler after a code modification. This round-off error cannot be circumvented. It takes some training to judge quickly where this occurs (best practice 5, page 3).

It is desired to automate testing.

3.6 Version identification

CONTACT prints the revision number from Subversion in its startup message:

```
-------------------------------------------------------------------------
|  CONTACT - detailed investigation of 3D frictional contact problems   |
|                                                                       |
|  Version: trunk(64), $Revision: 2116 $, $Date:: 2022-01-14$           |
|  Copyright Vtech CMCC, all rights reserved.                           |
|  (Subversion revision 2116M, 14-Jan-2022 17:00)                       |
-------------------------------------------------------------------------
```

This facilitates the comparison of different outputs, e.g. reference output from one version and new output from a modified version. If differences need be explained, we can then use bisection to identify the version where those differences can be seen for the first time.

This needs to be extended for the open source version using Git for version control.

3.7 Modern Fortran

The first version of CONTACT was developed using Fortran IV. Then came Fortran77 and Fortran90. Today we use some features of Fortran 2003 or 2008.

A nice feature is to have multiple subroutines used under a generic name. An example is the `write_log` function from `m_print_output`. This can be called with one argument, a literal character string, or two arguments, an array of strings and the number of lines it contains:

```
public write_log
interface write_log
   module procedure write_log_str
   module procedure write_log_msg
end interface write_log

subroutine write_log_str(str)
   !--purpose: Write a single string to the log output streams
   implicit none
   !--subroutine arguments
   character(len=*), intent(in) :: str
end subroutine write_log_str

subroutine write_log_msg(nlines, msg)
   !--purpose: Write an array of strings to the log output streams
   implicit none
   !--subroutine arguments
   integer,          intent(in) :: nlines
   character(len=*), intent(in) :: msg(nlines)
end subroutine write_log_msg
```

If the compiler then encounters a call like `write_log('some message')`, it will try to match the actual arguments (one literal string) with each of the options. The example matches with `write_log_str`, which is then the actual subroutine that will be used.

See also operator overloading in Section 4.4.

3.8 Pointers and associate

Pointers are slightly different in Fortran than in other languages like C or C++. Whereas assignment '=' changes the pointer value (memory address) in C/C++, in Fortran this changes the variable that is pointed to (memory contents). In Fortran, the assignment '=>' is used to change the pointer value (memory address).

Pointers in Fortran behave much like nick-names for the names of other variables, e.g.

```
type(t_profile), pointer :: my_rail

my_rail => wtd%trk%rai
   ...
```

This allows `my_rail` to be used as shorthand for `wtd%trk%rai` and behave like any other variable of type `t_profile`. The associate clause achieves mostly the same without the variable declaration:

```
associate( my_rail => wtd%trk%rai )
   ...
end associate
```

Pointers are used only minimally in CONTACT, because of the risk they have for allocating memory and forgetting about deallocation. Associate is being used more and more throughout the code.

3.9 Lazy implementation

The modules/objects used in CONTACT are expanded using a strategy called "lazy implementation". Only those parts are implemented that are actually needed or foreseen to be needed.

An example concerns the construction of a rotation matrix from Euler angles (`t_rotmat` in `m_markers`). Four functions are defined for this:

```
function rotmat_pure_roll
function rotmat_pure_yaw
function rotmat_pure_pitch
function rotmat_roll_yaw
```

In principle, additional variants could be defined for `roll_pitch`, `yaw_pitch`, and `roll_yaw_-pitch`. There has been no need for these functions (yet), so they have not (yet) been implemented.

A typical error message in a generic function would be "this functionality (variant, combination) is not yet supported".

Adding all variants that logically fit together creates ballast: additional time for compilation, and additional maintenance work.

Chapter 4

Internal structure of CONTACT

4.1 Stacking of modules

Figure 4.1 shows how different modules in CONTACT build on each other (best practice 6, page 3).

- The makefiles are at the bottom, together with facilities provided by SoftwareKey (licensing mechanism) and Intel (MKL).
- Right above this we find facilities for printing, as needed to integrate the CONTACT output in the output streams of software packages like SIMPACK or RecurDyn.
- Going upwards we find generic facilities (timing, input, licensing, BLAS), quite generic facilities oriented towards CONTACT (markers, grids, grid functions, interpolation), and more specific facilities (friction, influence coefficients, w/r profiles).
- Next comes the aggregation in `m_hierarch_data`. This provides the data-type `t_probdata` that groups all the elements used for a single "contact problem" in module 3.
- The layer above `m_hierarch_data` shows the components used in the stand-alone program in module 3: I/O, preparations, solving, and specific computations.
- The layer `m_wrprof_data` provides the data-type `t_ws_track` that groups all the elements used for a single "w/r contact problem" in module 1. This uses many aspects of `m_hierarch_data`, and adds specific elements for the analysis of wheel/rail contact analysis.
- The layer above `m_wrprof_data` shows the additional components used for solving w/r contact problems.
- The top layer adds the main program and driver routines, and the additional data and wrapper functions for the CONTACT library version.

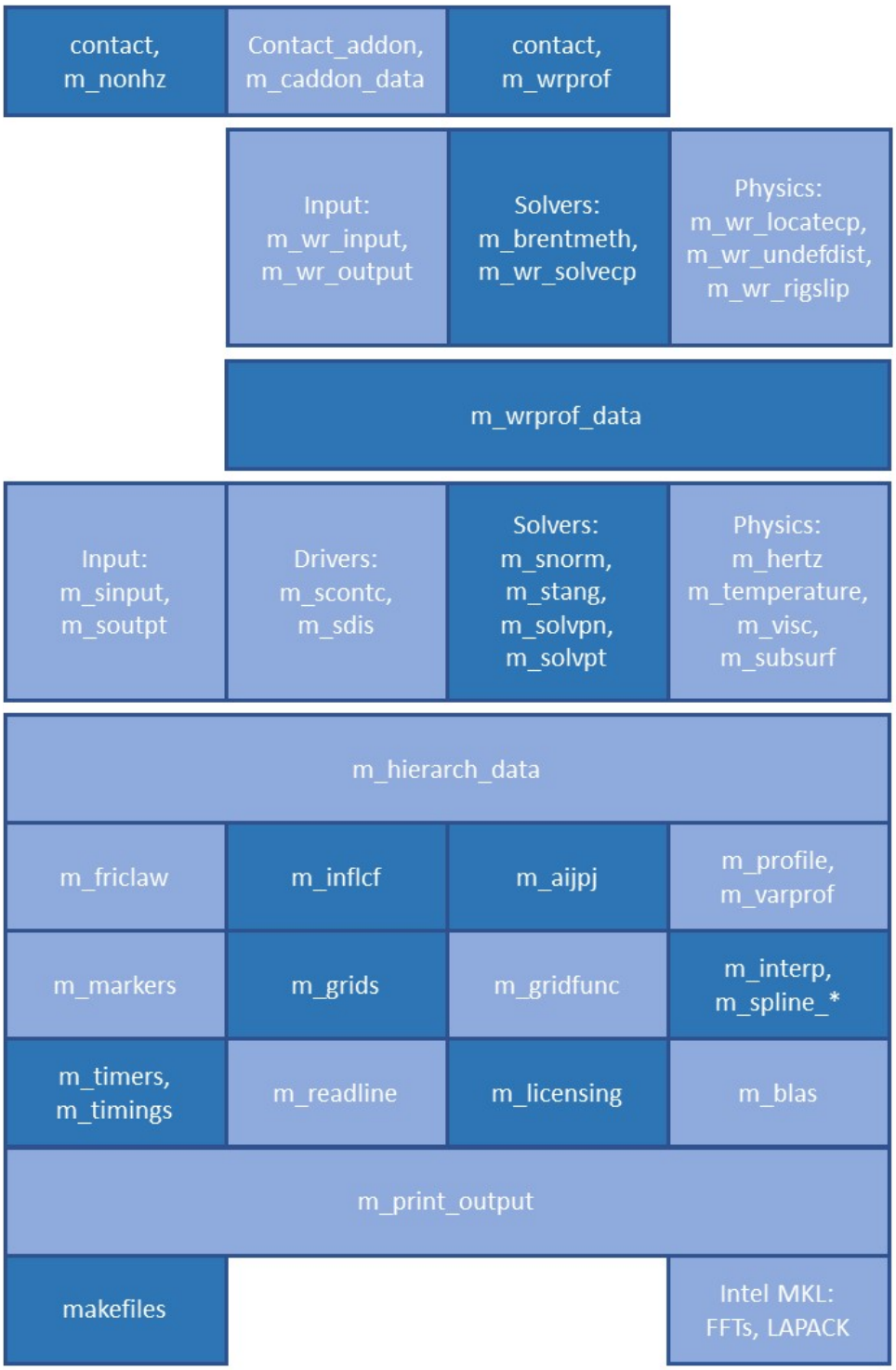


Figure 4.1: *Stacking of the Fortran90 modules used in CONTACT.*

4.2 Scope of modules

Modules are used to group functionality that belongs together and separate from other functionality that is less related. "High cohesion, loose coupling" (best practice 6, page 3).

Some modules are centered around a derived type. An example of this is `m_friclaw`, that hides the many facets of different friction laws from the rest of the program. This module includes the input from the `inp`-file and output to the `out`-file specific to friction. This uses ideas from object orientation.

Other modules just combine related computations. Examples are `m_solvpt` for different solution algorithms for the tangential problem, and `m_sdis` for different preparations for the actual solver routines.

4.3 Hierarchical data-structure

The implementation of CONTACT revolves around "hierarchical data-structures" as defined in `m_hierarch_data.f90` and `m_wrprof_data.f90`.

The main object in contact is a "contact problem". In module 1, this is encoded in the type `t_ws_track`, in module 3 it is `t_probdata`. These types group together all the inputs, configuration, intermediate results, and outputs of a single contact problem.

In module 3 these data comprise

```
! aggregate of all data for a CONTACT calculation in module 3:

type :: t_probdata
   type(t_metadata) :: meta   ! meta-data describing the calculation
   type(t_scaling)  :: scl    ! scaling factors for the CONTACT library
   type(t_ic)       :: ic     ! integer control digits
   type(t_material) :: mater  ! material-description of the bodies
   type(t_potcon)   :: potcon ! description of potential contact area
   type(t_grid)     :: cgrid  ! main discretisation grid for CONTACT
   type(t_hertz)    :: hertz  ! Hertzian problem-description
   type(t_geomet)   :: geom   ! geometry-description of the bodies
   type(t_friclaw)  :: fric   ! input parameters of friction law used
   type(t_kincns)   :: kin    ! kinematic description of the problem
   type(t_influe)   :: influ  ! combined influence coefficients
   type(t_solvers)  :: solv   ! variables related to solution algorithms
   type(t_output)   :: outpt1 ! solution and derived quantities
   type(t_subsurf)  :: subs   ! data of subsurface stress calculation
end type t_probdata
```

A number of these items are re-used in the contact problem central to module 1. Additional data

used there are

```
   ! aggregate data for the half wheelset on track/roller combination:

   type :: t_ws_track
      type(t_discret)  :: discr   ! parameters used for potential contact
                                  !     areas and discretization
      type(t_vec)      :: ftrk, ttrk, fws, tws, xavg, tavg
      real(kind=8)     :: dfz_dzws ! sensitivity of fz_tr to z_ws

      type(t_wheelset) :: ws      ! wheel-set data, including wheel profile
      type(t_trackdata):: trk     ! track/roller data, including rail profile
      integer          :: numcps  ! number of contact problems
      type(p_cpatch)   :: allcps(MAX_NUM_CPS) ! pointers to the data for
                                  !     all contact problems of ws on trk
   end type t_ws_track
```

Module 1 uses an array of "contact patches", where each contact patch eventually contains the data for a "contact problem" as defined in module 3.

A central object in CONTACT is a "contact patch". This is defined primarily using a "potential contact area" and a corresponding "CONTACT grid" of $m_x \cdot m_y$ rectangular elements of size $\delta x \times \delta y$. Module 1 identifies and defines contact patches, module 3 performs the solution.

4.4 Module m_markers, operator overloading

Module `m_markers` provides the derived types (classes?) `t_vec`, `t_rotmat`, and `t_marker`. The main ideas behind these are explained in the paper [3].

A fourth type that is defined is `t_coordsys`. The idea behind this is to provide automated checking of compatibility of markers used in computations. This hasn't been worked out.

The module uses operator overloading to define standard operations like addition of vectors, multiplication of a rotation matrix times vector, or the product of two rotation matrices, e.g.

```
public operator (+)
interface operator (+)
   procedure vec_add
end interface operator (+)

function vec_add(v1, v2)
   type(t_vec), intent(in) :: v1, v2  !--inputs
   type(t_vec)             :: vec_add !--result value
end function vec_add
```

If the compiler then encounters the addition of two vectors, it will call the function as follows:

```
ftot = vec( 0d0, 0d0, 0d0 )
do icp = 1, numcps
   cp   => wtd%allcps(icp)%cp
   ftot = ftot + cp%ftrk   ! --> ftot = vec_add(ftot, cp%ftrk)
enddo
```

4.5 Module m_ptrarray

Module `m_ptrarray` provides a number of generic facilities, esp. `reallocate_arr`, that permit changing size of arrays during program execution.

4.6 Module m_grids

Module `m_grids` defines the derived type `t_grid` with corresponding functions. The main ideas behind this are presented in [3].

There are actually three different types of grids that are squeezed into a single derived type `t_grid`: curves, used to represent profiles, curvilinear grids, used to represent wheel and rail surfaces, and uniform grids, used for the potential contact area. Benefits of combining into one type are that this avoids repetition (translating a profile, translating a surface), and that this facilitates conversion from one type to another (profile to surface).

The module does not aim to provide a fully generic grid data-type. Only those grids are supported that are needed by CONTACT. The operations provided came about by the process of "lazy implementation" (§3.9). More functions can be added as needed in CONTACT.

For profiles, the grid used may be complemented with a spline approximation (§4.7).

4.7 Interpolations: m_interp, m_spline_*

Interpolations are essential to CONTACT's analysis of the wheel/rail contact geometry problem. These are used in two different forms:

1. subroutines for 2d bilinear interpolation on curvilinear grids, provided in `m_interp`,
2. subroutines related to splines, for profile smoothing and interpolation.

A lot of code has been developed for working with splines. This started with "PP-splines" to represent plane curves (profiles), adding smoothing, adding "B-spline curves", and finally adding "B-spline surfaces". These are typically stored together with a grid definition. The original PP-form

is implemented in `m_spline_def`, `make`, `get`; extensions for 1D/2D B-splines are implemented in `m_bspline_def`, `make`, `get`.

The main ideas related to splines are presented in [3]. The arrays used are stored along with the original profile data in the `t_grid` data structure. Crucial subroutines are `make_smoothing_spline_section`, `grid_spline_eval`, and `solve_cubic_segm`. Many other routines are built on top of these cornerstones.

The functionality for splines is still under construction. Needs completion of the functionality and extension of the documentation.

The main work-horse for 2D interpolation is subroutine `interp_wgt_surf2unif`, establishing the mapping between a curved surface and a uniform mesh in an efficient way. A number of variants are built on top of this, for interpolation of scalar data or 3-vectors, packed in an array, surface (grid), or grid-function.

Bibliography

[1] T. Winters, T. Manshreck, and H. Wright. *Software Engineering at Google; Lessons Learned from Programming over Time*. O'Reilly Media Inc., Sebastopol, CA, 2020.

[2] K. Henney. *97 Things Every Programmer Should Know: Collective Wisdom from the Experts*. O'Reilly and Associates, Boston, MA, 2010.

[3] E.A.H. Vollebregt. Detailed wheel/rail geometry processing using the planar contact approach. *Vehicle System Dynamics*, 60(4):1253–1291, 2022. Open access.