NAME

fmigo-master - connect to FMUs served by *fmigo-server* over TCP/IP
fmigo-server - serve FMU over TCP/IP
fmigo-mpi - runs FMUs in an MPI cluster

SYNOPSIS (TCP/IP)

fmigo-master [**−rltcdopfmgwC5NMazZLHDeGRE**] *URLs*
fmigo-server --port PORT *FMUFILENAME*

SYNOPSIS (MPI)

mpiexec -np 1 fmigo-mpi [arguments] : -np 1 fmigo-mpi *FMUFILENAME* [: -np 1 fmigo-mpi *FMUFILENAME* [: ...]]
mpiexec -np (num_fmus+1) fmigo-mpi [arguments] *FMUFILENAMEs*

DESCRIPTION

FmiGo simulates FMUs using the FMILibrary from JModelica.org over TCP/IP, with optional strong coupling between simulation units. Communication over MPI is also possible.

For examples, see the bottom of this document.

TCP/IP

TCP/IP communication is handled by ZeroMQ. Serialization is done via libprotobuf. FmiGo uses a master/server architecture, with a central node (*fmigo-master*) connecting to a number of servers (*fmigo-server*) , each of which serve a single FMU.

URLs

Each URL specifies the address and TCP port of an FMU server. The syntax is:

```
tcp://<address>:<port>
```

Examples:

```
tcp://localhost:3000
tcp://192.168.0.2:3000
```

EXAMPLE

To run two servers in the background on one machine do something like:

```
$ fmigo-server -p 3000 example1.fmu &
$ fmigo-server -p 3001 example2.fmu &
$ fmigo-master tcp://localhost:3000 tcp://localhost:3001
```

MPI

The MPI version of FmiGo uses a structure similar to the TCP/IP version, but with only a single executable. The MPI world consists of a master node (rank 0) and server nodes (rank 1..N). For N FMUs the world size must be N+1. A simple system with two FMUs and a weak connection from FMU0 variable x to FMU1 variable y looks like this:

```
mpiexec -np 1 fmigo-mpi -c 0,x,1,y \
  : -np 1 fmigo-mpi fmu0.fmu \
  : -np 1 fmigo-mpi fmu1.fmu
```

Keep in mind that FMU indices in connections and parameters are still zero based, even if the FMU server nodes begin at rank 1. The MPI ranks do not change the master argument syntax.

If you want multiple copies of one FMU you can increase the -np argument instead of adding more lines to the command line:

```
mpiexec -np 1 fmigo-mpi -c 0,x,1,y \
  : -np 2 fmigo-mpi fmu01.fmu
```

There is also an alternative syntax which produces shorter command lines but require computing the total number of nodes (N FMUs + 1):

```
mpiexec -np 3 fmigo-mpi -c 0,x,1,y fmu0.fmu fmu1.fmu
```

OVERSUBSCRIPTION

When the number of FMUs exceed the total number number of cores in the system, the system will become oversubscribed (number of MPI nodes > number of cores). This will cause CPU use to increase quadratically with the number of FMUs in certain MPI implementations that use spinlocks to increase performance. There are a number of solutions to this problem:

- Switch to a different MPI implementation (from MPICH to OpenMPI for example)
- Let the MPI implementation know how many slots each node has, for example by using a hostfile with OpenMPI with *slots* specified for each host.
- Configure and build your MPI implementation without spinlocks (--with-device=ch3:sock for MPICH)
- Use TCP/IP instead (*fmigo-master* and *fmigo-server*)

OTHER MPI GOTCHAS

On some setups certain MPI features are only available via either *mpiexec* or *mpirun* but not both. One example is Intel's MPI implementation, which as of September 2018 only supports shared memory when run with *mpirun.* In other words, if *mpirun* doesn't work then try switching to *mpiexec* and vice versa. Binding is another example where both OpenMPI and Intel's MPI have different behaviors between the two programs. Which one is appropriate seems to depend; we're had more success with *mpirun.* Some implementations only have *mpiexec* however, such as MS-MPI.

OUTPUT FORMATS

The program can output data in either text based formats or binary .mat files suitable for Matlab or Octave import (compressed and uncompressed). The text formats are CSV and TikZ (space separated). This data is written either to stdout (default) or to a file (*-o*). Matlab output cannot be written to stdout; the program will halt if *-f mat5* or *-f mat5_zlib* are used without *-o* It is also possible to suppress output data entirely, using *-f none*

Whatever the output format, it is important to keep track each FMU's ID in order to make sense of the output. The first FMU (or server URL) listed gets ID 0, the next ID 1 and so on up to N-1.

Finally, if solver fields are to be output (*-E*), their naming scheme derives from the order in which the kinematic couplings are specified, and the number of equations in each kinematic coupling. For each equation the constraint violation and the speed of the violation is reported. For each FMU there are also some number of kinematic connectors, for which forces and/or torques are reported. Mobilities associated with each connector may also be reported. Since these output columns are mostly used for debugging the kinematic solver, further details have been left out since they are subject to change. *The truth is in the code* .

CSV/TIKZ

For CSV and TikZ output, each output row contains output values for one communication point. The first column is the time of the current communication point, after which follows a number of output columns. Output columns are grouped by FMU ID, from lowest (0) to highest (N-1). The output columns are in the same order as listed in the <fmiModelDescription><ModelStructure><Outputs> element in each FMU's modelDescription.xml. Variables not listed in <Outputs> are not written to the output. The columns are named like "fmu%i_%s" in C parlance, where %i is the FMU ID and %s is the name of the variable in question. If FMU ID 1 has an variable listed in <Outputs> called "foo.bar" then it is written with a column name of "fmu1_foo.bar".

The following is some example CSV/TikZ output with rows containing time, sin(time), an increasing integer, a boolean being set equal to the integer and a string of increasing length of alternating a's and double quotation marks (a"a"a"). For all examples, FMU=build/tests/umit-fmus/tests/alltypestest/alltypestest.fmu. You can find this FMU's source code under tests/umit-fmus/tests/alltypestest/.

```
# CSV with header
mpiexec -np 2 fmigo-mpi -t 0.4 -d 0.1 -H $FMU

  #t,fmu0_r_out,fmu0_i_out,fmu0_b_out,fmu0_s_out
  +0.0000000000000000e+00,+0.0000000000000000e+00,0,0,""
  +1.0000000000000001e-01,+9.9833416646828155e-02,1,1,"a"
  +2.0000000000000001e-01,+1.9866933079506122e-01,2,1,"a"""
  +3.0000000000000004e-01,+2.9552020666133960e-01,3,1,"a""a"
  +4.0000000000000002e-01,+3.8941834230865052e-01,4,1,"a""a"""
```

```
# TikZ
mpiexec -np 2 fmigo-mpi -t 0.4 -d 0.1 -f tikz $FMU

  t fmu0_r_out fmu0_i_out fmu0_b_out fmu0_s_out
  +0.0000000000000000e+00 +0.0000000000000000e+00 0 0 ""
  +1.0000000000000001e-01 +9.9833416646828155e-02 1 1 "a"
  +2.0000000000000001e-01 +1.9866933079506122e-01 2 1 "a"""
  +3.0000000000000004e-01 +2.9552020666133960e-01 3 1 "a""a"
  +4.0000000000000002e-01 +3.8941834230865052e-01 4 1 "a""a"""
```

```
# Two copies of the same FMU, CSV with header
mpiexec -np 3 fmigo-mpi -t 0.4 -d 0.1 -H $FMU $FMU

  #t,fmu0_r_out,fmu0_i_out,fmu0_b_out,fmu0_s_out,fmu1_r_out,fmu1_i_out,fmu1_b_out,fmu1_s_out
  +0.0000000000000000e+00,+0.0000000000000000e+00,0,0,"",+0.0000000000000000e+00,0,0,""
  +1.0000000000000001e-01,+9.9833416646828155e-02,1,1,"a",+9.9833416646828155e-02,1,1,"a"
  +2.0000000000000001e-01,+1.9866933079506122e-01,2,1,"a""",+1.9866933079506122e-01,2,1,"a"""
  +3.0000000000000004e-01,+2.9552020666133960e-01,3,1,"a""a",+2.9552020666133960e-01,3,1,"a""a"
  +4.0000000000000002e-01,+3.8941834230865052e-01,4,1,"a""a""",+3.8941834230865052e-01,4,1,"a""a"""
```

Real values are written in scientific notation with enough decimals to be losslessly transported.

.MAT OUTPUT

For .mat files, the output of each FMU is put in a 1x1 structure array named after the FMU the same way as the column prefixes in CSV/TikZ output (fmu0, fmu1 etc.). Inside each structure array are vectors containing each output variable and corresponding values, each named by the names of the corresponding output variables. Finally, there is a regular array containing the communication points named "t", as a sibling to all the FMU structure arrays. So for the last example listed in the CSV/TikZ subsection, if the .mat file is loaded into a variable d then it will have for example d.fmu0.r_out containing all the values of r_out in

FMU 0. A somewhat trimmed example:

```
$ mpiexec -np 3 fmigo-mpi -t 0.1 -d 0.1 -f mat5 -o out.mat $FMU $FMU
$ octave
octave:1> d = load('out.mat')
d =
  scalar structure containing the fields:
    t =
       0.00000
       0.10000
    fmu0 =
      scalar structure containing the fields:
        r_out =
           0.000000
           0.099833
        i_out =
          0
          1
        b_out =
          0
          1
    fmu1 =
      scalar structure containing the fields:
        r_out = [...]
        i_out = [...]
        b_out = [...]
```

If there is a lot of output data it may be beneficial to use Zlib compressed output, which can be enabled by requesting an output format of *-f mat5_zlib* instead of *-f mat5* . Strings cannot currently be output to .mat files.

FLAGS

−r Realtime mode. Will usleep() in between communication steps if the execution was faster than real time.

−N Make kinematic connectors nonholonomic rather than holonomic. In other words, only lock spatial/angular velocities, not positions or angles.

−Z Start master in paused state. Use ZMQ "unpause" command to start simulation. Requires -z.

−L Solve algebraic loops in initialization mode. Requires GPL and GNU GSL.

−H Print CSV header. TikZ output always has a header.

−E Print solver fields in CSV/TikZ output. This includes things like constraint violations, mobilities and forces. Only sensible when using kinematic constraints, else nothing extra is printed.

−D Always compute numerical directional derivatives, regardless of the value of providesDirectionalDerivatives in all FMUs' modelDescription.xml. This is useful for kinematic coupling if one or more FMUs have a spring-damper type input. Such FMUs would give a much too high mobility estimate, unless the timestep is taken into account somehow. This is exactly what the numerical directional derivative code does, hence this flag.

−e Print some preprocessor variables suitable for "export", to stdout, then quit. This is useful for figuring out at runtime how fmigo was configured. Example cmake -DUSE_GPL=0:

```
$ fmigo-mpi -e
USE_MPI=1
```

```
USE_GPL=0
$ fmigo-master -e
USE_MPI=0
USE_GPL=0
```

Example of passing to export for further script use:

```
$ for e in $(fmigo-mpi -e); do export "$e"; done
$ echo "The value of USE_GPL is $USE_GPL"
The value of USE_GPL is 0
```

OPTIONS

First note that some options specify lists of values. These can either be specified as colon separated lists (like "-w 0:1:0") in one long option, or multiple times for each one ("-w 0 -w 1 -w 0"), or any combination of the two ("-w 0 -w 1:0"). This is convenient when building command lines in script, especially when lengthy strong connection specifications (-C) are involved.

−l LOGLEVEL

Set FmiGo and FMILibrary log level. Levels are as follows:

0 = nothing (default)
1 = fatal Unrecoverable errors
2 = error Errors that may be not critical for some FMUs
3 = warning Non-critical issues
4 = info Informative messages
5 = verbose Verbose messages
6 = debug Debug messages. Only enabled if FMILibrary is configured with FMILIB_ENABLE_LOG_LEVEL_DEBUG and FmiGo is compiled in Debug mode
7 = all

−G EXECUTION_ORDER_XML

Some systems need information to propagate as if they were using super-dense time. One common example is Gauss-Seidel type stepping, where each FMU is executed one-at-a-time in a serial manner, and only the latest data being used at each step. This is not something the FMI Co-Simulation spec allows since data must be exchanged at matching communication points only. But, since it is required in many cases FmiGo allows it regardless of what the specification says.

This option is for specifying an execution order in an XML format. The format is based on nested execution groups, which can be either serial (<s></s>) or parallel (<p></p>). Each group can contain some FMU IDs (<f></f>) and nest several of the other type of group inside. So a parallel group may contain a bunch of FMU IDs and some serial groups, all of which are executed in parallel. The serial groups in turn may contain FMU IDs and parallel groups which are executed in the order they appear in the XML. The root group is parallel, but may contain a single serial element if so desired. Serial groups must contain at least two elements.

To step all FMUs in parallel, put all <f> in the same <p>:

```
-G "<p><f>0</f><f>1</f><f>2</f></p>"
```

To step all FMUs in series, put all <f> in the same <s> and put the <s> in the root <p>:

```
-G "<p><s><f>0</f><f>1</f><f>2</f></s></p>"
```

More complicated arrangements are of course possible. This example steps two serial groups with two FMUs each in parallel, with a fifth FMU parallel to all of them for good measure:

```
-G "<p> \
      <s> \
        <f>0</f> \
        <f>1</f> \
      </s> \
      <s> \
        <f>2</f> \
        <f>3</f> \
      </s> \
      <f>4</f> \
    </p>"
```

All FMU IDs must occur exactly once in the XML. See FmiGo.xsd (installed under bin/) for more information about the syntax. If no -g or -G is specified, or if there is one or more ModelExchange FMU, then all FMUs are stepped in parallel (Jacobi).

−g SERIAL_EXECUTION_ORDER_LIST

Serial FMU execution order, as a comma-separated list of FMU IDs. This is a simpler legacy way of specifying serial execution order. Example:

```
-g 0,2,1
```

Which means: step FMU0 -> FMU2 -> FMU1. Equivalent to this execution order XML:

```
-G "<p><s><f>0</f><f>2</f><f>1</f></s></p>"
```

Like other lists stepping order lists can also be broken up. The same example can also be written like this:

```
-g 0 -g 2 -g 1
```

The number of entries in -g must match the number of FMUs in the system. If no -g or -G is specified, or if there is one or more ModelExchange FMU, then all FMUs are stepped in parallel (Jacobi).

−c WEAK_CONNECTIONS

Weak connection specification list. Represents which FMU and value reference to connect from and what to connect to. Syntax is

```
-c WCONN1:WCONN2:WCONN3...
```

where the syntax of each WCONNX is one of the following:

```
FMUFROM,VRFROM,FMUTO,VRTO
FMUFROM,NAMEFROM,FMUTO,NAMETO
TYPE,FMUFROM,VRFROM,FMUTO,VRTO
TYPEFROM,FMUFROM,VRFROM,TYPETO,FMUTO,VRTO
TYPEFROM,FMUFROM,VRFROM,TYPETO,FMUTO,VRTO,k,m
FMUFROM,NAMEFROM,FMUTO,NAMETO,k,m
```

TYPE is a single character specifying the value type on the connection. If TYPE is absent then the connection is assumed to be of type real, unless NAMEs are specified (more on this further down). Possible types:

r - Real
i - Integer
b - Boolean
s - String

FMUFROM and FMUTO are the indexes of the FMUs to read values from and to respectively. VRFROM and VRTO are the corresponding value references. Example:

```
-c 0,0,1,0:r,0,1,1,1:i,0,1,1,2
```

This means: connect real values FMU0 (value reference 0) to FMU1 (vr 0) and FMU0 (vr 1) to FMU1 (vr 1), and connect integer value FMU0 (vr 1) to FMU1 (vr 2). This is a little hard to read though, so breaking connection lists with multiple "-c"'s is recommended. Weak connections can also be specified in any order, so the following specifies the exact same set of connections:

```
-c i,0,1,1,2 -c 0,1,1,1 -c r,0,0,1,0
```

It is possible to specify types on either side of the connection, by which type conversion may be performed. For example:

```
-c r,0,1,i,1,0 -c b,0,2,r,1,2
```

Means: connect FMU0 VR1 to FMU1 VR0, truncating each real to an integer. Connect FMU0 VR2 to FMU1 VR2, converting false to 0 and true to 1. More information on type conversion is given at the end of this section.

NAMEFROM and NAMETO are alternatives to VRFROM and VRTO. They allow you to specify connections (and infer TYPE) by NAME rather than value reference. It is required that the FMU has exactly one connection with any given name. Example:

```
-c 0,x_out,1,x_in
```

Connects variable x_out in FMU0 to variable x_in in FMU1. Each name must have at least one non-numeric character to count as such, or the argument parser won't be able to tell what's what.

Finally k (slope) and m (intercept) are used for transforming values linearly. They can be any real value. This can be combined with data type conversion, for instance taking fixed point integers from one FMU, converting them to real and scaling by 1.0/65536 before passing the resulting scaled real values to the other FMU. This would look something like this:

```
-c i,0,1,r,1,2,1.5259e-05,0
-c 0,some_integer,1,some_real,1.5259e-05,0
```

Before transformation every type is converted to real. This conversion follows C conventions. Strings or enums may not be converted, in any direction. The converted real values (x) are scaled and offset by k and m respectively:

```
y = k*x + m
```

The resulting values (y) are then converted to the target type. For reals no further conversion is required. For integers this means truncation. For booleans, abs(y) > 0.5 is considered true.

Default is no connections.

−p PARAMS

Parameter specification list. Specifies parameters to send to FMUs during initialization. Format is:

```
-p PARAM1:PARAM2:PARAM3...
```

where each PARAMX is one of the following:

```
FMU,VR,VALUE
TYPE,FMU,VR,VALUE
FMU,NAME,VALUE
```

Comma, colon and backslash characters in VALUE must be escaped with backslash. A typical use case is paths on Windows, which might look something like this in bash syntax:

```
-p 's,0,0,C\:\\foo bar\\woo.tx'
```

Note that extra escaping may be necessary in order to pass through your shell properly. If using double quotes in bash:

```
-p "s,0,0,C\\:\\\\foo bar\\\\woo.tx"
```

No characters other than comma, colon and backslash may be escaped. Having a single trailing backslash in an option is an error ("C\:\\foo\").

If TYPE is not specified then real values are assumed, just like with weak connections. If NAME is non-numeric then the value reference and type is looked up by name. Example:

```
-p b,0,0,true:r,0,0,0:s,0,0,hello
```

which means set FMU0 boolean VR0 to true, FMU0 real VR0 to zero and FMU0 string VR0 to "hello". Note that despite identical value references these entries refer to different parameters since VRs apply with respect to a base type. See -c option for a list of possible types (i, r, s, b). Parameters can be specified in any order, and like all lists they can be broken up, so the following specifies the exact same set of parameters:

```
-p s,0,0,hello -p b,0,0,true -p 0,0,0
```

Finally, some examples using names:

```
-p 0,some_boolean,true
-p 1,some_integer,123
-p 2,participant1,Alice:2,participant2,Bob
```

Default is no parameters.

−C STRONG_CONNECTIONS

Strong coupling specification. Syntax is

```
-C SCONN1:SCONN2:SCONN3...
```

where SCONNX has the following syntax:

```
SCONNX=TYPE,FMU0,FMU1,[PARAMS]
```

FMU0 and FMU1 are the two sides of the strong coupling. It is also possible to have strong connections involving more than two FMUs, see "multiway" further down. PARAMS depend on TYPE, and TYPE is the type of connection:

[ball|lock]:

PARAMS=pos0,acc0,force0,quat0,angAcc0,torque0,pos1,acc1,force1,quat1,angAcc1,torque1

where posX/accX/forceX/angAccX/torqueX are VR triplets (X,Y,Z) and quatX are VR quadruplets (X,Y,Z,W), giving a total of (3+3+3+4+3+3) x 2 = 38 value references.

The difference between "ball" and "lock" is that lock tries to lock the orientation of both connectors (ball only cares about position).

shaft:

PARAMS=shaftAngle0,angularVelocity0,angularAcceleration0,torque0,shaftAngle1,angularVelocity1,angularAcceleration1,torque1

The connection tries to keep both shaftAngles equal.

Examples:

```
-C shaft,0,1,20,19,14,17,20,19,14,17
```

Meaning: Connect a shaft between FMU0 and FMU1, with VRs shaftAngle=20, angularVelocity=19, angularAcceleration=14 and torque=17 on both sides

```
-C lock,0,1,\
    0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,\
    0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18
```

Meaning: Create a lock constraints between FMU0 and FMU1 w ith VRs pos={0,1,2}, acc={3,4,5}, force={6,7,8}, quat={9,10,11,12}, angAcc={13,14,15} and torque={16,17,18} on both sides.

Note that like all lists you can concatenate the strong connection specifications with colon characters, but the result is hardly readable:

```
-C shaft,0,1,\
    20,19,14,17,20,19,14,17:\
  lock,0,1,\
    0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,\
    0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18:\
  ball,1,2,\
    0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,\
    0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18
```

Just like weak connections, variable references can be resolved by name. So the following is also OK:

```
-C shaft,0,1,\
  theta1,omega1,alpha1,tau1,\
  angle2,angularVelocity2,angularAcceleration2,torque2
```

It is also possible to specify multi-way shaft-like constraints using the "multiway" strong connection, which is useful for things like differential gears. Such contraints involve N FMUs (N>=2) and an extra weight in each connector specification. The syntax is:

```
   SCONNX=multiway,N,FMU0,FMU1 ... FMU(N-1),[shaftAngle,angularVelocity,angularAccel-
eration,torque,weight]xN
```

The following example will constrain the average angle and velocity of the connectors on FMU 1 and 2 to be equal to the angle and velocity on FMU 0:

```
   -C multiway,3,0,1,2,\
      phi,omega,alpha,tau,-1,\
      omega,alpha,tau,0.5,\
      omega,alpha,tau,0.5
```

The -1, 0.5 and 0.5 are then the weights.

Default is no strong connections. Specifying strong connections is incompatible with using the Gauss-Seidel stepper (-m gs).

−w VISIBILITIES

Visibility specification list. Specifies which FMUs should show their simulator application windows. Syntax is

```
   -w VIS0:VIS1:VIS2...
```

where VISX is 1 or 0 depending on whether FMUX's window should be shown or not. Example:

```
   -w 0:1:1:0
```

which means: show simulator windows for FMU1 and FMU2, but not FMU0 or FMU3. Note that multiple "-w"'s can be used to break up a list:

```
   -w 0 -w 1 -w 1 -w 0
```

Unlike -p, -c and -C order is significant for visibility lists. The following is *not* equivalent to the previous example:

```
   -w 1 -w 1 -w 0 -w 0
```

By default no windows are shown (batch mode).

−d TIMESTEP

Timestep size. Default is 0.1.

−f OUTFORMAT

Output file format. Can be "csv" (default), "tikz" or "none". Use -H to print header with CSV output. TikZ output always uses a header.

−o OUTFILE

Result output file. Default is STDOUT.

−t ENDTIME

End simulation time in seconds. Default is 1.0.

−S MAX_SAMPLES

Maximum number of data samples collected during the simulation. Negative value indicates that all data should be collected. Defaults to -1.

−M COMPLIANCE
Set compliance for kinematic solver (real value, default = 0.0).

−R RELAXATION
Set relaxation time for the kinematic stepper. This is units of the time step. For a a value of less than "2", the constraint violations decrease by a factor of 1/(1+4*relaxation) per step. Above "2" the decrease slows down. TO BE CONTINUED.
area = u

−a ARGSFILENAME
Add extra arguments parsed from file with given name, or stdin if filename is -. This is useful for large systems where the total size of the connection specification exceeds the operating system's limit for program arguments (2 KiB of Windows). The arguments in the file may be separated by anything std::ifstream::operator>>(std::string) considers a white space (space, newline, tab etc.). The parsed tokens effectively replace the "-a ARGSFILENAME" in the list of arguments. Recursive files are not allowed - if the argument file itself contains a "-a" token then the program stops. Example:

```
fmigo-master -t 100 -a args -p 0,1,123
```

Contents of file args:

```
-C shaft,0,1,0,1,2,3,0,1,2,3
-C shaft,1,2,6,7,8,9,0,1,2,3
-c 2,1,0,6
```

Resulting equivalent command line:

```
fmigo-master -t 100 -C shaft,0,1,0,1,2,3,0,1,2,3 -C shaft,1,2,6,7,8,9,0,1,2,3 -c 2,1,0,6 -p 0,1,123
```

stdin example producing the same command line (bash style here-document):

```
fmigo-master -t 100 -a - -p 0,1,123 << EOF
-C shaft,0,1,0,1,2,3,0,1,2,3
-C shaft,1,2,6,7,8,9,0,1,2,3
-c 2,1,0,6
EOF
```

−z command_port[:results_port]
Set up ZMQ command (REQ/REP) and optional results (PUSH/PULL) ports. Allows controlling master and PULLing results over ZMQ. If results_port is given then -f none is assumed, unless -f csv or -f tikz is specified after -z.

Messages are serialized using protobuf. For more information, see src/master/control.proto.

EXAMPLES

To run an FMU simulation from time 0 to 5 with timestep 0.01:

```
fmigo-master -t 5 -d 0.01 tcp://localhost:3000
```

To simulate two FMUs connected from the first output of the first FMU to the first input of the second:

```
fmigo-master -c 0,0,1,0 tcp://localhost:3000 tcp://localhost:3001
```

Simulating four strongly coupled spring systems for 100 s at 100 Hz and writing the result to a CSV file:

```
  fmigo-master -t 100 -d 0.01 \
    -p 0,3,0 -p 0,0,0:0,6,1 -p 0,9,2 -p 1,0,1:1,6,2 -p 1,3,2 -p 1,9,2 -p 2,0,2:2,6,3 -p 2,3,2 -p 2,9,2 -p
3,0,3:3,6,4 \
    -C shaft,0,1,6,7,8,10,0,1,2,4 \
    -C shaft,1,2,6,7,8,10,0,1,2,4 \
    -C shaft,2,3,6,7,8,10,0,1,2,4 \
    tcp://localhost:3000 tcp://localhost:3001 tcp://localhost:3002 tcp://localhost:3003 > results/output-
N4-h0.01.csv
```

ABOUT

The app was built by Stefan Hedman at UMIT Research Lab 2013. Large parts were rewritten by Tomas Härdin at UMIT Research Lab 2014 - 2018.