QuSI-DEM User's Manual

Version 1.0

Scott M. Durski

July 7, 2026

Contents

List of Code Listings

1 Introduction

This user manual for the Quaternion-based Sea Ice Discrete Element Model (QuSI-DEM or QuSI for short) is intended to help users get up and running with the model. It is a compliment to the model description (Durski et al, 2026a) which describes the physics and numerical algorithm, and model evaluation (Durski et al. 2026b) which demonstrates the model sensitivity to various parameters.

This document will step through obtaining the code and building the model, generating initial element distributions, specifying atmospheric and oceanic forcing and boundary conditions, setting up the input script, running the model and analyzing the output of simulations.

2 Obtaining the code

The QuSI-DEM repository is hosted on Bitbucket (bitbucket.org). It contains the source code, sample scripts and input data for several test problems. In order to download QuSI the user should first ensure that 'git' is installed on their system, then clone the repository. For example, at a command line, in a Linux terminal this can be done by first changing to the parent directory of where you would like QuSI installed then typing the following at the command prompt:

Clone Command

```
git clone git@bitbucket.org:durskis/qusi-dem.git
```

This will download the latest release version of the code from the master branch at the Bitbucket repository.

3 Building and running the model

3.1 Prerequisites

There are a number of prerequisites required for QuSI. In the list below are examples of each that have been shown to work.

- a C++ compiler (intel/2024.0)

- a version of Python 3 (Python 3.12)
- blender-mathutils for python (https://github.com/i0n/blender-math)
- python-netcdf
- HDF5-c++ (v.1.14.3)
- boost-c++ library (v.1.89.0)

QuSI has not yet been tested with many variations on these pre-requisites. It is possible that for example, a newer version of a compiler will not be compatible an older version of a library. Creating a docker environment in which to install the model has been shown to be successful.

3.2 Configuring and building

As of this writing, we suggest that QuSI be built by using the makefile in the src directory. The model includes tools in the parent directory (bootstrap.py, config.h, etc.) that ideally would allow the user to use GNU autoconf to determine system attributes and configue the model accordingly. But, these programs have not been tested recently. Before compiling, the user should edit the settings in the src/Makefile to match their requirements and system. In particular, ensure that the paths to library and include file directories are set correctly. For example, the '-I' and '-L' arguments in the makefile in the following lines must point correctly to the HDF library and include files on the user's system.

src/Makefile

```
HDF5_CPPFLAGS = -I/opt/hdf5-hdf5-1_14_3/src/H5FDsubfiling
↪ -I/usr/local/hdf/include
HDF5_FC =
HDF5_FFLAGS = -I/usr/local/hdf/lib -L/usr/local/hdf/lib
↪ -I/usr/local/hdf/include
HDF5_FLIBS =  -lm -ldl -lz -lhdf5_fortran -lhdf5
↪ -lhdf5hl_fortran -lhdf5_hl
HDF5_LDFLAGS = -L/usr/local/hdf/lib
HDF5_LIBS = -lm -ldl -lz  -lhdf5 -lhdf5_hl
```

and that the compiler commands are correct. For example, if using a recent version of the intel compiler one might set CC and CXX as follow

src/Makefile

```
CC = icx
CXX = icpx
```

Additionally, the user may experiment with optimization and other compiler flags on their system by editing the model compiler flags. For example, for code debugging the user may set the following.

src/Makefile

```
CXXFLAGS = -g -O0 -std=gnu++11
#CXXFLAGS = -O3 -std=gnu++11
```

Once the makefile is setup appropriately the model can be built in the src directory by typing 'make' at the command line. Although the model compiles relatively quickly the process can be parallelized to build multiple routines at once by using the '-j' option. For example, to parallelize 4 routines at a time

```
$ make -j 4
```

Once the model is compiled, unless specified otherwise the executable will be located in the $MODEL_ROOT/src directory. Before running the model it will be necessary to set the PYTHON_PATH system variable to ensure that the needed python routines can be located by the executable. In a bash environment this can be done by exporting the variable

```
$ export
↪  PYTHONPATH=/data/user/QuSI/samples/Hopper:/data/QuSI/python
```

Depending on how the model has been built, it may also be necessary to specify the LD_LIBRARY_PATH to point to compiler libraries.

3.3 Running QuSI-DEM

Model execution is typically performed from the command prompt in a terminal window. The model executable requires one command line argument, the name of the python startup script. This script will be discussed in the

next section. To run the model from the command line (assuming the default executable name and startup script names are being used) type

```
$ ${PATH_to_EXECUTABLE}/qusi startup_script.py
```

Where $PATH_TO_EXECUTABE specifies where the executable resides (by default in the $MODEL_ROOT/src directory). If paths to all libraries are set correctly successful execution should result in terminal output such as the following:

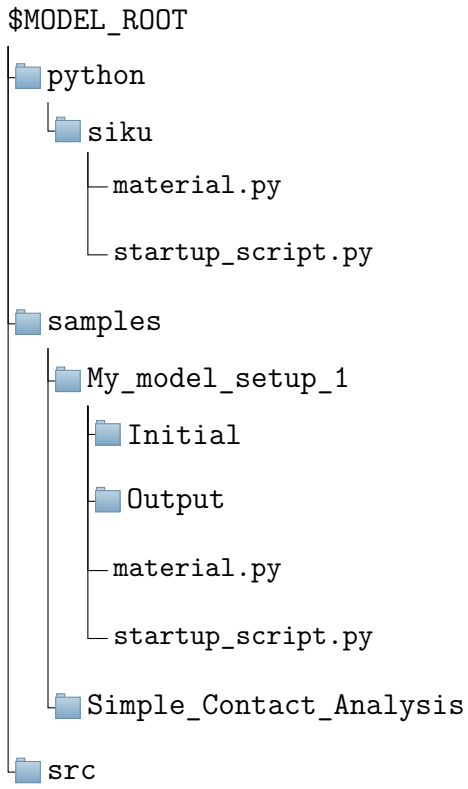
```
$ ../../src/qusi startup_script.py
QuSI 1.0
Quaternion-based Sea Ice discrete element model
Elements: 3
Current: 2000-Feb-09 03:00:02
Current: 2000-Feb-09 03:00:04
Current: 2000-Feb-09 03:00:06
Current: 2000-Feb-09 03:00:08
Current: 2000-Feb-09 03:00:10
...
Current: 2000-Feb-09 04:00:00
DONE!
```

The model outputs the current model time as it progresses through the time integration followed by ”DONE!” when finished. At present, QuSI-DEM only runs in serial, but multiple instances can easily be run simultaneously on a multicore system.

4 Setting up to run a simulation

User input to QuSI-DEM is supplied through python scripts. Ideally, a user should have a working understanding of Python to facilitate model setup. Two python scripts in particular are meant to be modified by the user; material.py and startup_script.py. Examples of these can be found in the `\$MODEL_ROOT/python/siku` directory. By default, the model will use the material.py file from this directory if not supplied in the directory where the code is being executed. But in general, a user should create a working/run directory for their simulations under `\$MODEL_ROOT/samples` and copy the two scripts

into it to customization for their specific purposes. Initial condition and Output files can be organized as subdirectories of the user's working directory to maintain organization



4.1 Setting up properties of different materials in `material.py`

The material.py file describes basic characteristics that will be constant across sets of elements. For example the user may define one set of characteristics to describe sea ice, and another set to describe coastal barrier elements. For each type of material a set of properties should be set in the user's material.py script. The variables that are set in this file and the default values set in the `material.py` example in `\${MODEL_ROOT}/python/siku` are listed in the table below.

A feature of QuSI-DEM that is not fully implemented is the capability to handle multiple thickness categories for a material within a given class, each with distinct characteristics. This is why properties like sigma_t are dimensioned as vectors of length(thickness_intervals) in material.py. But at this

Physical Parameters defined in material.py			
Model variable	Symbol	Description	Default Value
thickness_intervals	H	Thickness (m)	1.0
rho	ρ	Density (kg m^{-3})	910.0
sigma$_\mathrm{t}$	σ_t	Tensile strength (kPa)	12.5
sigma$_\mathrm{c}$	σ_c	Compressive strength (kPa)	125.0
E	E	Elastic modulus (Pa)	1.0×10^9
nu	ν	Poisson's ratio	0.3
mu	μ	Coefficient of internal friction	0.6

Table 1: Table of physical parameters used in idealized experiment test cases

point it is best to consider only a single thickness_interval for a material, and handle elements with differing properties as distinct materials. As in the example snippet from a material.py file below.

material.py

```
@classmethod
def pack_ice(cls):
    mprop = cls.__new__(cls)
    mprop.name = 'pack_ice'
    mprop.thickness_intervals = [1.00]  # m
    mprop.rho = [910.0] * len(mprop.thickness_intervals)
    mprop.sigma_t = [12.5] * len(mprop.thickness_intervals)
    mprop.sigma_c = [125.0] * len(mprop.thickness_intervals)
    mprop.E = 1e9  # Pa
    mprop.nu = 0.3  # 0.3
    mprop.mu = 0.6  # 0.6
    return mprop

@classmethod
def thin_ice(cls):
    mprop = cls.__new__(cls)
    mprop.name = 'thin_ice'
    mprop.thickness_intervals = [0.1]  # m
    mprop.rho = [910.0] * len(mprop.thickness_intervals)
    mprop.sigma_t = [3.25] * len(mprop.thickness_intervals)
    mprop.sigma_c = [32.5] * len(mprop.thickness_intervals)
    mprop.E = 1e9  # Pa
    mprop.nu = 0.3  # 0.3
    mprop.mu = 0.6  # 0.6
    return mprop
```

4.2 Setting up run parameters in `startup_script.py`

is primarily designed with a dynamical core written in c++ for computational efficiency, and a python front-end for accessibility. `startup_script.py` (or however a user chooses to rename it), is the principle interface that a user modifies to setup and monitor model simulations. It is in this script that the user specifies model settings, initial element distributions and model forcing. But it also provides an interface for monitoring run progress or adding pre- or post-processing steps that are performed before or after a simulation. There is significant flexibility in the organization of startup_script.py. But it's essential function is to provide variable values and parameters to the C++

dynamical core. It does this by writing everything from model time step to individual element geometries, to forcing fields, to a python structure named *siku* (afer the original name of the QuSI model.) which is passed to the C++ executable. Here we outline a basic version of the script.

4.2.1 Specifying timestep, start date, I/O frequency

Model time step, start date, and save interval are specified near the top of `startup_script.py`. Times are specified as `datetime` objects, such that they can be specified in microseconds, seconds, minutes etc. at the user's convenience.

startup_script.py

```
# ------------------------------------------------------------
  # date/time settings
  # ------------------------------------------------------------
  hour = datetime.timedelta(minutes=60)
  second = datetime.timedelta(seconds=1)
  minute = datetime.timedelta(minutes=1)
  # Specify simulation start time
  start =  datetime.datetime  ( 2000, 2, 9, 3, 00, 00 )
  siku.time.start = start
  siku.time.finish = start + hour * 1
  siku.time.dt = datetime.timedelta(microseconds=100000)
  siku.time.dts = datetime.timedelta(minutes=10)
  siku.time.save_start = start + minute * 0
  siku.diagnostics.monitor_period = 300000
```

start simulation start date/time (must align with times in wind/current/slope forcing files).

finish simulation end date/time (sets simulation duration).

dt time-stepping interval (forward-Euler) used in the model.

dts time-interval between writes of instantaneous model state to HDF5 files.

save_start time/date of first write of instantaneous model state.

Currently *siku.diagnostics.monitor_period* indicates the number of time steps between interpolations of forcing (winds, currents, slopes). Model forcing

is held constant over the *monitor_period*, then updated by interpolation between time records of the forcing file. [*Right now this is set manually, but in a future release two time records will be read in and passed to the C++ program for time interpolation internally at each time step. The overhead of calling the python script to perform the interpolation frequently can be high.*]

4.2.2 Atmospheric and oceanic forcing

The model reads in environmental forcing files through the python interface currently and passes the gridded fields along with the grid to the core C++ code for interpolation to locations of individual elements. The python scripts provided are set up to read all forcing data of a certain type (winds, currents, slopes) at once before the simulation starts.

startup_script.py

```
# ------------------------------------------------------------
#  Wind, current and surface slope initializations (ECM grid example)
# ------------------------------------------------------------
# 10m wind speed
wind_file = 'Siku/DataSets/Forcing/Winds/era5_wind.nc'
siku.uw = wnd.ECMVar(filename=wind_file, var_name='u10', var_t=0)
siku.vw = wnd.ECMVar(filename=wind_file, var_name='v10', var_t=0)
# ocean surface current
current_file = 'Siku/DataSets/Forcing/Currents/ocean_current.nc'
siku.uc = wnd.ECMVar(filename=current_file, var_name='u_oce', var_t=0)
siku.vc = wnd.ECMVar(filename=current_file, var_name='v_oce', var_t=0)
# ocean surface slope
slope_file = 'Siku/DataSets/Forcing/Slopes/ocean_slope.nc'
siku.dz_ln = wnd.ECMVar(filename=slope_file, var_name='xln_slp', var_t=0)
siku.dz_lt = wnd.ECMVar(filename=slope_file, var_name='ylt_slp', var_t=0)
```

Time interpolation is performed via python scripts (`updatewind`, `updatecurrent`, `updateslope` in `startup_script.py`) if specified, at a time interval determined by the setting of `siku.diagnostics.monitor_period`. Currently, the python scripts are designed to read in forcing netcdf files with data on lat-lon grids, where the dimensions and coordinate variable names follow ECMWF conventions. This is specified in `\$MODEL_ROOT/python/siku/wnd.py`. Other formats may be added in this location by the user. The following shows a typical header to a netcdf wind forcing file that can be read by

era5_wind_speed.nc

```
netcdf era5_wind_speed {
dimensions:
        longitude = 559 ;
        latitude = 239 ;
        time = UNLIMITED ; // (29 currently)
variables:
        double longitude(longitude) ;
                longitude:units = "degrees_east" ;
                longitude:long_name = "longitude" ;
        double latitude(latitude) ;
                latitude:units = "degrees_north" ;
                latitude:long_name = "latitude" ;
        int time(time) ;
                time:units = "hours since 1900-01-01
                ↪  00:00:00.0" ;
                time:long_name = "time" ;
                time:calendar = "gregorian" ;
        short u10(time, latitude, longitude) ;
                u10:units = "m s**-1" ;
                u10:long_name = "10 metre U wind component" ;
        short v10(time, latitude, longitude) ;
                v10:units = "m s**-1" ;
                v10:long_name = "10 metre V wind component" ;
```

Note that it is recommended that latitude and longitude be specified in double precision particularly when examining fine scale processes. It is also important that the spatial extent of the forcing data encompasses not only the initial element distribution but also the full area that the farthest floe may drift over the duration of the simulation.

Currently currents and sea surface slopes are read in using the the same function as used for the 10m wind speed (`wnd.ECMVar`) so netcdf files should be structured accordingly (with the appropriate variable names replacing *u10* or *v10* in the call. Ocean currents are assumed to be in units of ms^{-1}. Ocean slopes are assumed to be in units of m/100km.

After the fields are read in with `wnd.ECMVar` the initial time record is determined, and the initial vector fields are read into the siku structure for passing to the dynamical core of the program.

4.2.3 Specifying materials

The next section in the provided template `startup_script.py` gives names to, and passes the materials specified in `material.py` to the *siku* structure. The example below provides for specifying three different types of materials, appending them to *siku* and associating them with identifiers.

startup_script.py

```
# ------------------------------------------------------
# Define material(s)
# ------------------------------------------------------
ice = material.Material()  # default ice characteristics
thin_ice = material.Material.thin_ice()
coast = material.Material.ordinary_coast()

siku.materials.append(ice)  # list of all materials
siku.materials.append(thin_ice)
siku.materials.append(coast)
# table of material names for convenience
matnames = {
    'ice': 0, 'thin_ice': 1, 'ordinary_coast':2
```

Once material types are defined elements can be assigned to the different types as described in the following section.

4.2.4 Initial element distribution

The initial element distribution information is read in from a set of plain text files. Similar to finite element descriptors, one file defines the coordinates of nodes (typically named with the extension '.xyz'), a second describes the connectivity of the nodes into elements (typically named with the extension '.xyzf'). Coordinates of nodes are specified in Cartesian coordinates (x,y,z) on a unit sphere, with the z-axis oriented towards the north pole (rotational axis) of the sphere. The coordinate (.xyz) text file should be formatted with one node per line, with the x, y, and z coordinates separated by spaces or tabs as in the example below (which lists a set of nodes at approximately 64°N, 170°W).

Two_1km_64N_Hex_adjacent.xyz

```
>1  -0.427981040266022   -0.094786590878715   0.898803499861012
>2  -0.428058312421976   -0.094832835655680   0.898761822980901
>3  -0.427902725988590   -0.094834636989228   0.898835718426151
>4  -0.428057271690194   -0.094927125308245   0.898752364688884
>5  -0.427901685679782   -0.094928928442725   0.898826260151047
>6  -0.428134540261654   -0.094973367583156   0.898710673623529
>7  -0.427978959225438   -0.094975171984788   0.898784583293890
>8  -0.428133497758075   -0.095067655299001   0.898701201186127
>9  -0.427977917145483   -0.095069461501956   0.898775110873428
>10 -0.428055187106665   -0.095115702541268   0.898733419831019
```

(Note: In the above example the '>' at the start of each line, indicates line number, but is not text included in the '.xyz' file). The nodes in the model are numbered in the order in which they appear in this file.

The connectivity file (typically appended with '.xyzf') indicates which nodes make up a polygon. Each line of the connectivity file lists the nodes that define one polygonal element. *Nodes must be specified in counterclockwise order*. The following connectivity file for example, corresponds to the coordinate file above, describing two hexagonal elements that share a common edge.

Two_1km_64N_Hex_adjacent.xyzf

```
  5    3    1    2    4    7    5    0
  7    4    6    8    10   9    7    1
```

Each line of the connectivity file describes an element by specifying nodes in any counterclockwise order, with the first node that is repeated at the end of the sequence. This is followed by an integer identifier index that the user may use to describe the type of element. Note that the list of coordinates is only used to define the initial positions of polygon vertices. Elements that are initially connected or (simply abutting with aligned edges) may share the same initial node identifiers. In the example above the vertices that are identical between the two elements are specified by reference to the same nodes (4 and 7) in the coordinate file.

In the example startup_script.py, the elements are read into a structure named *PV* with the `poly_voronoi.PolyVor` function.

startup_script.py

```
PV = poly_voronoi.PolyVor('Initial/Two_1km_64N_Hex_adjacent.xyz',
                          'Initial/Two_1km_64N_Hex_adjacent.xyzf')
```

Initial element velocities (if any) can be read in from a plain text file as well or specified explicitly in startup_script.py. Velocities are specified as longitudinal, latitudinal and vertical (only w=0 currently) components in units of ms^{-1}. Functionality to initialize elements with initial rotation has not been implemented but can be in a similar manner.

4.2.5 Specifying distinct element properties

The identifier index at the end of each line of the element connectivity file (.xyzf) can be used to specify shared properties among subsets of elements. For example some elements may be specified as fixed coastal points, some as thinner ice floes or ice with different failure characteristics. The example `startup_script.py` demonstrates how this information may be specified and passed into the *siku* object (*siku.elements*).

startup_script.py

```
# Initializing element properties that differ depending on how
# they are tagged by the last value on each row of the ".xyzf" file.
for ic,c in enumerate(PV.coords):
    siku.P.update( c )
    if int(PV.seq.bry[ic]) == 0:
      E = element.Element( polygon = siku.P, imat = matnames['ice'] )
      gh = [1.0]
      E.set_gh( gh, ice )
      siku.elements.append( E )
      veli_cmp_str = veli_str[ic].split()
      #siku.elements[ic].velo = (0.00071564,-0.000691596,0.0)
      siku.elements[ic].velo = tuple([float(f) for f in veli_cmp_str])
      siku.elements[ic].failure_rate = 10   # seconds
    if PV.seq.bry[ic] != '0':
      E = element.Element(polygon=siku.P, imat=matnames['ordinary_coast'])
      gh = [1.0]
      E.set_gh( gh, ice )
      siku.elements.append( E )
      siku.elements[ic].flag_state = element.Element.f_static
      #siku.elements[ic].flag_state = element.Element.f_unbondable
      siku.elements[ic].failure_rate = 100
      #siku.elements[ic].flag_state = siku.elements[ic].flag_state +
                                      element.Element.f_unbondable
```

The identifier index from the connectivity file is stored in *PV.seq.bry*. In the example snippet of text above, elements with an index of '0' are specified as

having the material properties of ice with an initial velocity provided from a text file and contact failure_rate set to 10 s. Elements with any index other than zero are classified as having the material properties defined in `material.py` as 'ordinary_coast', are static (immovable),and have a contact failure rate of 100 s. In any case where properties are not specified for an element, those properties will be set to default values.

The element `flag_state` is a property that can carry combinations of attributes. The following is a list of flag states that can be applied for convenience. Typical ice elements that experience body and contact forces are by

flag_state	
flag	Description
f_free	normal model physics (default)
f_static	immovable
f_fastened	bond between element and neighboring coastal element is unbreakable
f_unbondable	unbonded to neighboring elements
f_steady	maintain initial velocity throughout simulation
f_anavel	time-varying velocity specified as function (in c++ code)
f_controlled	override normal physics with controlled movement
f_special	user-defined specification
f_errored	flagged for internal error

default defined as `f_free`. Immovable coastal elements should be labelled as `f_static`. But as mentioned, flag states can be combined, so a coastal element that is specified to be unbonded to neighboring ice elements can be defined as `f_static+f_unbondable`. Other flag settings help with specialized setups, such as having boundary elements that attempt to maintain a constant normal stress (`f_controlled`) or having elements experience a specified deceleration as they move against other elements (`f_anavel`).

4.2.6 Model contact properties section

The next portion of the supplied startup script specifies settings related to element contacts.

startup_script.py

```
# ---------------------- Model contact setttings ------------------------
   siku.settings.contact_method = siku.CONTACT_METHODS['sweep']
   siku.settings.force_model = \
                   siku.CONTACT_FORCE_MODEL['edge_orient']
   # Options currently are face_oriented_shear and COM_oriented_shear.
   siku.settings.force_model_orient = \
                   siku.CONTACT_FORCE_MODEL_ORIENT['COM_oriented_shear']}
```

The `contact_method` describes the algorithm the model uses to determine if elements are in close enough proximity to each other, in order for contact physics to be considered. Currently, the 'sweep' method is the only method that has been well tested, but algorithmically more efficient methods may be added in the future. The model is also set up to allow for different `FORCING_MODEL`s, but currently the 'edge_orient' model is the only one that has been rigorously tested (Durski et al. *submitted* 2026a, Durski and Hutchings,*submitted* 2026b).

The `force_model_orient` specified whether normal and shear orientation at a joint are determined by the orientation of the edge between the elements or by the orientation of the centers of mass (COMs) of the two elements relative to each other. This is discussed in (Durski et al. *submitted* 2026a) and sensitivity to this parameter is presented in Durski and Hutchings,*submitted* 2026b).

4.2.7 Global model parameters

A variety of scalar constant parameters can be passed to the model through the startup python script under the `phys_consts` label. These parameters set global values for variables that are constant across elements and contacts (Future releases may provide further customization by allowing the user to modify some of these parameters on an element or contact basis). Parameters can be specified in any order or omitted if using default values.

startup_script.py

```
# ----------------------- Global model parameters ----------------
# contact failure parameters
siku.settings.phys_consts['crushed_stiffness'] = 0.25 # crushed ice relative stiffness - 'residual strength'
siku.settings.phys_consts['dest_threshold'] = 0.1  # sets max weakening (typically less than crushed_stiffness)

# Drag, friction and viscosity parameters
siku.settings.phys_consts['wind_drag_coef'] = 0.003  # 0.003 -
siku.settings.phys_consts['current_drag_coef'] = 0.1  # 0.1
siku.settings.phys_consts['wind_angle'] = 0  # wind turning angle (degrees, positive clockwise))
siku.settings.phys_consts['current_angle' ] =0 # ocean current turning angle (degrees)
siku.settings.phys_consts['fric_dissp'] = 0.0 # fric. diss. coeff (proportional to normal compr.force - COLLISION)
siku.settings.phys_consts['etha'] = 0.0 # -//- viscosity  0.0051
siku.settings.phys_consts['elas_collision_damping'] = 0.0  # damping of elastic normal force in COLLISION contact

# Rotation
siku.settings.phys_consts['periodRot'] = 86400  # planetary period of rotation in seconds (large for negligible rot)

# ---------------------- control element parameters  ---------------
siku.settings.phys_consts['nudge_fac'] = 0 # a nudging rate factor for controlled elements
siku.settings.phys_consts['control_limit_factor'] = 0.0  # % of max stress to drive contact to on control element
```

All parameters set here have default values as specified in the table below. Various parameters can be set to very large or small values to effectively

Global model parameters		
parameter	default	Description
Contact failure parameters		
crushed_stiffness ($\mathrm{D_c^o}$)	0.2	fraction of initial strength at which contact transitions from JOINT to COLLISION
dest_threshold ($\mathrm{D_t}$)	0.1	fraction of initial strength at which normal stress remains constant with further compressive displacement (typically less than crushed$_\mathrm{s}$tiffness)
Drag, friction and viscosity parameters		
wind_drag_coef ($\rho_\mathrm{a}\mathrm{C_d^a}$)	0.003 (kg m^{-3})	wind drag $\times$ air density
current_drag_coef ($\rho_\mathrm{o}\mathrm{C_d^o}$)	0.1 (kg m^{-3})	current drag $\times$ water density
wind_angle	0 ($+^\circ$clockwise)	wind-ice turning angle
current_angle	0 ($+^\circ$clockwise)	current-ice turning angle
fric_dissp	0.3	coefficient of frictional dissipation (multiplied by normal compressive force, used in COLLISION contact only)
etha	0.0 (kg s^{-1}m^{-1})	viscous dissipation proportional to relative velocity/rotation of elements
elas_collision_damping	0.0	damping of elastic normal force in COLLISION contact
Rotation		
periodRot	8640000 s^{-1}	planetary period of rotation in seconds (large for negligible rot)
Control element parameters		
nudge_fac	0 s^{-1}	a nudging rate factor for controlled elements
control_limit_factor	1.0	fraction of max stress to drive contact to on control element

eliminate the associated term from the model physics. For example, by default, the planetary rotation rate is set to 100 days^{-1}, which largely excludes the influence of the Coriolis force.

4.2.8 Specifying output names and paths

Finally, the user needs to provide a path to the save directory and an output file prefix in `startup_script.py`. This will determine where HDF5 output files are written from the C++ code.

startup_script.py

```
##  =============  Specify the output file prefix  ======================
    siku.save_dir = '/data/sdurski/Siku/Runs/Simple_Contact_Analysis'
    siku.output_file_prefix: str='Simple_slope_test_3Hex_Rot_1_chk'
##  ============== Save specifics of the model setup used for this run ==
    THIS_file: str = os.path.join(os.getcwd(),'startup_script.py')
    MATRL_file: str =  os.path.join(os.getcwd(), 'material.py')
    Setup_file: str = os.path.join(siku.save_dir+'/Setups', \
    siku.output_file_prefix + '_setup.py')
    Material_file: str = os.path.join(siku.save_dir+'/Setups', \ siku.output_file_prefix + '_material.py')
    copyfile(THIS_file, Setup_file)
    copyfile(MATRL_file, Material_file)
```

Snapshots of the model state will be saved in `\${save_dir}/Output`, given the prefix specified and numbered sequentially starting from zero. In the given example the startup script itself and the `material.py` file are also written to a subdirectory of `\${save_dir}` labeled `\Setups`. These setup-file saves are given the same `output_file_prefix` as the model output so the user can easily associate simulation output with the setup. The example is also given in the startup script of saving a tarred zipped version of the source directory associated with each run. (This is useful to track changes if actively modifying the C++ source code between simulations but has limitations.). The user may feel free to rearrange the output structure through different specifications of paths. For example, the user may want to define distinct output directories for each simulation.

5 Generating initial element distributions

The methods used to generate grids are currently implemented in MATLAB but could be translated into other programming languages such as Python relatively easily. A set of MATLAB scripts for working with is available at bitbucket (https://bitbucket.org/durskis/matlab_scripts_for_siku_bit/src/master/). The element distribution tools take advantage of built-in MATLAB functions for two-dimensional polygons and finite element mesh generation and refinement. Initial element distributions are specified in a Cartesian plane

to facilitate this. For a regional domain, an appropriate local map projection (e.g., an ESRI projection) can be used to map coastal boundaries onto the plane. Once the initial element distribution is generated, the inverse of the projection can be used to project all vertices onto the sphere. While this can lead to some distortion to the shape of the polygons, since vertices of adjacent polygons are collocated, they will map to the same location on the sphere and not introduce erroneous initial contact forces.

5.1 Generating a setup with a few elements

A basic example of generating a small set of hexagonal elements is provided in the `\Model\_Setup` subdirectory in the file `Basic\_element\_generation.m`. Polygons are generated on a mapped two-dimensional plane. To do this and associate the plane with coordinates on the sphere, we use an ESRI map projection that is appropriate for the region of interest. Elements are defined in the map projection coordinates and then mapped back to the sphere as in the code snippet below. (The projection chosen in this example has coordinates in units of meters and is rotated relative to lat-lon.)

Basic_element_generation.m

```
lat0=64;
lon0=-167.5;
% choose an appropriate map projection based on the latitude and longitude
proj_esri_code = 3571; % esri code for a metric projection near Bering Strait
prj1 = projcrs(proj_esri_code);

% Specify one element cenered at x0,y0
[x0,y0] = projfwd(prj1,lat0,lon0);

% Determine element vertices in the projection coordinates
% Hexagonal element Area = 3 * sqrt(3)/2*s^2
% So for a 1 km^2 area  s = 2/(3*sqrt(3))^{1/2} = 0.6204
slen = 620.4; % in meters   for 1 km ^2
% V will contain a list of vertex coordinates in the projection coordinate
% R will list the IDs of vertices associated with each element (connectivity)
V=zeros(6,2);
R{1}=[];
for ip = 1:6
    V(ip,1) = x0 + slen*cos(ip/6*2*pi);
    V(ip,2) = y0 + slen*sin(ip/6*2*pi);
    R{1} = [R{1} ip];
end
...
% eliminate duplicate (co-located) vertices, and replace vertex IDs in
% element list where necessary.
[V_f, R_f]=element_clean_and_reorder(V,R);

% specify labels to identify elements to be treated specially
Eside = zeros([length(R_f) 1]);

% ensure that the ordering of vertices in R_f is counter-clockwise oriented.
R_fc=make_polygons_ccw(V_f, R_f);

% perform inverse projection onto the surface of a unit sphere,
% then write vertices to file ('.xyz) in 3-D Cartesian coordinates (x,y,z)
% and list of verticex IDs associated with each element to connectivity file ('.xyzf').
write_element_polygons(file_prefix,V_f,R_f,Eside,prj1);
```

5.2 Generating a mesh that consists of many elements

For more complex situations, where one wants to fill a region with random elements with aligned edges, a mostly automated procedure has been developed using custom scripts and tools within MATLAB. Automating the procedure facilitates creating multiple randomized distributions of elements for a given domain relatively quickly to produce ensemble sets. This can be helpful in addressing issues of sensitivity of simulation outcomes to initial element edge orientations.

The procedure we use to generate the initial element distributions in com-

plex domains generally follows steps as displayed graphically in Figure 1. Though most of these steps are automated, they are described here so the user understands what is going on 'behind the curtains'.

1. Specify a polygon shaped domain (D_o) with any number of sides. Figure 1 a). Using a map projection at this step to specify coordinates simplifies the final step of mapping element to a sphere. A slightly expanded version of this domain D^+ is created for the next step
2. A triangulation of D^+ is generated using MATLAB mesh generating tools (Figure 1 b)
3. The positions of the vertices of the triangles are randomly displaced to reduce uniformity. (Without this step most elements will appear as aligned regular hexagons.
4. The 'perturbed' triangulation is tesselated. Polygons are formed by drawing lines that connect midpoints of the sides of the triangles. (Figure 1 c)
5. The tesselation is clipped along the boundary of the actual domain (D_o).
6. Element clean up is performed to limit the number of very small elements or elements with extreme aspect rations. (These tend to appear particularly along the clipped boundary.)
7. Boundary elements are specified (Figure 1 d). The vertices of these polygons are aligned with those of interior elements that touch the boundary of D_o.
8. Vertices in the x-y projection are mapped onto the unit sphere using the inverse map projection. (Figure 1 e).

5.2.1 Uniform resolution mesh

To generate a mesh of elements at roughly uniform resolution, the function `Model_setup\\Generate_uniform_elements_in_a_polygon.m` can be used.

Basic_element_generation.m

```
Generate_uniform_elements_in_polygon(Coast0,prj,Hscale,noise_fac,seed_num, file_prefix);
```

The inputs to this function are:

- *Coast0* the domain boundary (D_o)as a polygon in a map projection. An N$\times$2 array where N is the number of vertices (or sides) of the polygon.
- *prj* a MATLAB map projected coordinate reference system object. (e.g. `prj1=projcrs(3571)`)
- *Hscale* Element horizontal length scale (in the units of the map projection)
- *noise_fac* Random noise (fraction of average element edge length). A value of 1 would preturb the x-y position of a triangle vertex by as much as a distance of *Hscale*.
- *seed_num* A random seed (specified as a non-negative integer. This sets the randomization. Multiple similarly randomized element distributions can be generated for the same domain by varying this number. Reseting it to the same value as used previously should generate an identical randomization as that case.
- *file_prefix* The prefix for vertices (.xyz), element (.xyzf) and matlab (.mat) files along with figures generated of the generated element mesh.

5.2.2 Spatially varying element size

Element size can be adjusted to focus simulation cost on regions of interest or resolve small-scale processes in a portion of a much larger domain. This is useful, for example, for ice-coast interaction, where details of the coastal or landfast ice edge shape may be critical, but allowing larger-scale stress propagation through the regional or ocean-wide ice pack is also vital. This involves only minor modification to the procedure outlined above.

1. Additional parameters must be passed to the MATLAB finite element mesh generator routines. See the MATLAB `generateMesh` function for details on the 'Hgrad' and 'Hedge' parameters to specify along which edges resolution should be concentrated and the gradient with which it should vary away from those edges.
2. The inflated domain D^+ must be expanded more beyond D_o where resolution is to be low than where it is high. In general D^+ should extend

beyond the domain by an amount equal or greater to the local element scale.

(Currently a routine is not supplied that generates refined meshes fully automatically.)

Manual adjustments Sometimes the procedures presented above create element distributions that are problematic. This can manifests as cascading cracking under limited strain early in a simulation, or indivual elements rapidly accellerating from those that surround them. Usually this is due to elements that are very small relative to the average element area, or have a very short side relative to the average side length. Due to their small dimensions (lower mass) such elements will accellerate more rapidly upon bond failure and may require running with a much smaller time step then would be otherwise necessary.

If early in a simulation there are signs of unrealistic early fracturing under small strains, or the model blows up with a reasonably set time step, it can be useful to examine the initial element distribution for problematic elements. In doing so it helps to identify where failure is originating. If the model crashes, the model will likely report a specific element index 'W' (angular velocity) becoming NaN. It is likely that an unusually small or misproproportioned element is in the vicinity of this element. (Note that element IDs begin at '0' not '1' within .) If the model produces premature cracking, but does not blow up, it is useful to save model output at high enough frequency to identify where the cracking originates, in order to try to identify problematic elements.

The MATLAB `Model_setup` and `Processing_tools` have a number of scripts that help in identifying where in a domain an element ID is, or what element IDs are in a particular part of the domain. Elements can be sorted based on their distance from a given element, and plotted with element labels for a small subset of the elements in the domain. MATLAB scripts are also provided that allow the user to manually select element IDs to merge, if one finds that a problematic element can be merged with a neighbor and still result in a convex polygon. In some cases this is not possible but by moving one or two vertices shared by a set of elements a set of more reasonably sized convex polygons can be achieved. This task is left in the user's hands.

6 Analysis of model output

As explained in 4.2.1 the time-interval for saving output and the save start-time are specified in the model run parameters file. 'Snapshots' of model fields are written to files starting at the save start time and again every save time-interval. (*Currently there is no functionality to output time-averaged fields.*) A new output file is created at each record interaval, named according the the output file prefix specified in the run parameters file appended with a number running sequentially starting at 0001.

6.1 Format of model output

Output files are written in HDf5 format. They contain most of the model variables that may be of interest in directory tree-like structures. One convenient way to view the contents of an output file is with a utility such as HDFView (https://www.hdfgroup.org/download-hdfview/). Below is an example of model output as viewed in HDFView The output file contains various structures such as 'Contacts', 'Elements', 'Wind'. Each one contains fields or sub structures with related model fields or parameters. In Figure 2, the 'Contacts' structure is being displayed. It shows for example that Contact 369, is between elements 74 ('i1') and 4623 ('i2'). Vertex 3 of the first element ('v11') was in contact with Vertex 1 of the second element ('v22'), that the contact has weakened to 0.1 of the original strength ('durability'). The contact type is 'Collision' ('type'=1) and the previously existing bond failed due to shear under compression ('failure_type'=7). (More information is also available in this structure, but not displayed such as the normal and shear stress and the energy of the contact).

It is advisable for the user to become familiar with the variables in this file for analysis of QuSI output by examining the QuSI C++ code for variables of the same name. In the future attribute information will be added to the fields in the HDF5 file to describe the contents and units of each variable.

A variety of MATLAB routines are available for extracting contents of the HDF files into convenient structures for plotting and analysis. Most variables of interest can be accessed directly in matlab with the `h5read` command. One particularly relevant field that is needed for plotting element distributions spatially, that needs to be derived from the model output fields is the element vertex positions (in x,y or lat-lon). The functions in the matlab_tools di-

rectory that start with `read_element_*` process the information stored in the element quaternion to derive the position of each element vertex correctly. The MATLAB code snipet below demonstrates a sequence of function calls that would load Element and Contact information from all time records of 'Experiment_A' in the '/Output' directory.

Basic_element_generation.m

```
file_i.dir ='./Output';
file_i.prefix = 'Experiment_A';
% Collect file information
file_i=Siku_element_h5_file_collect(file_i);
% Collect element information
Elem=read_element_wfaces_h5(file_i);
% Collect contact information
Cd=read_contact_data_h5(file_i);
```

The Elem and Cd structures extracted in the calls above contain time series information collected over all the output files for all elements and contacts in the domain. This includes time series of such things as position, velocity, normal and shear stresses. A variety of MATLAB functions are available in the Processing_tools subdirectory that can be customized to extract more or less information from the HDF5 files for analysis.

7 Example

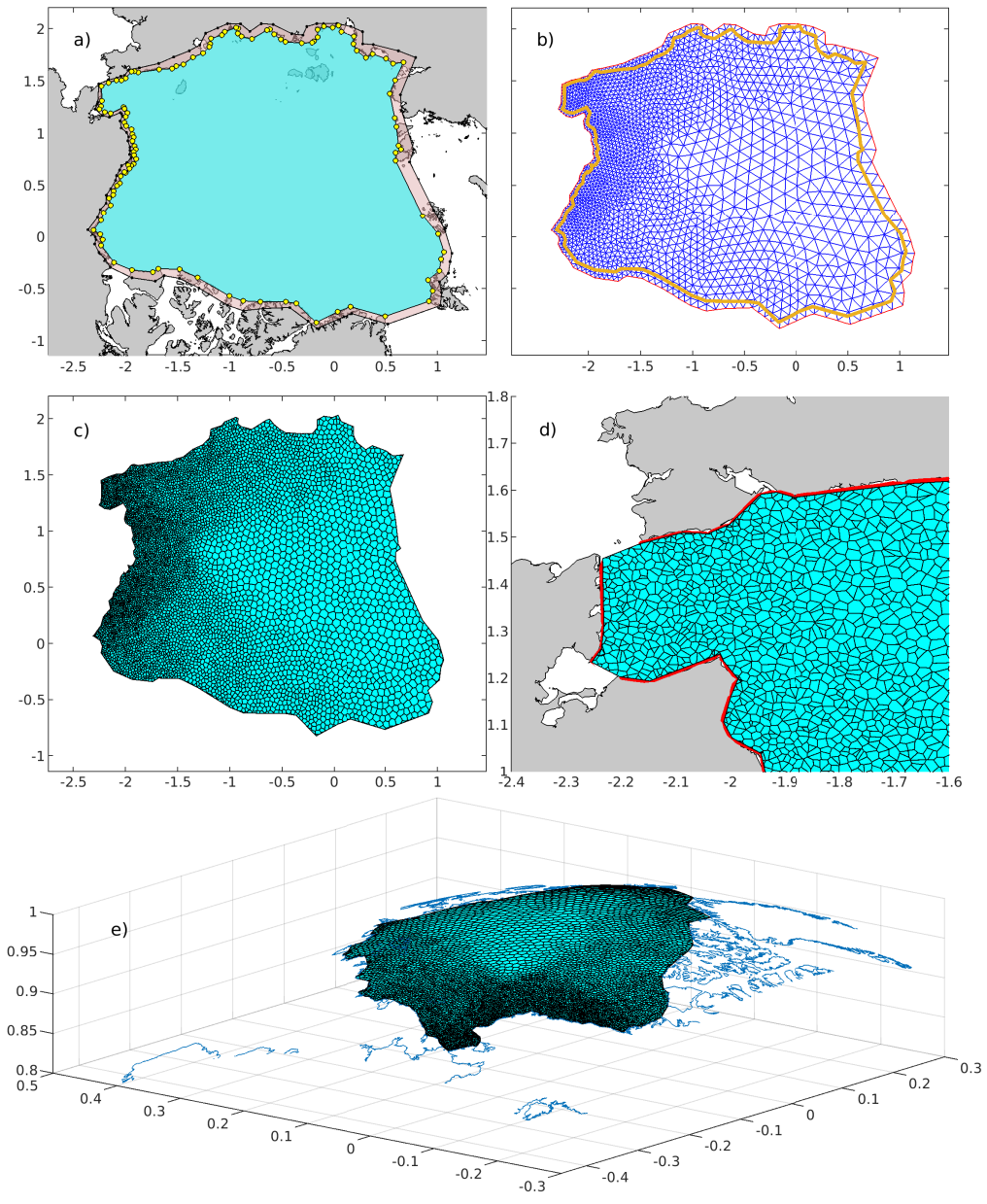


Figure 1: Process for generating the initial discrete element distribution for a coarse resolution Arctic Ocean setup

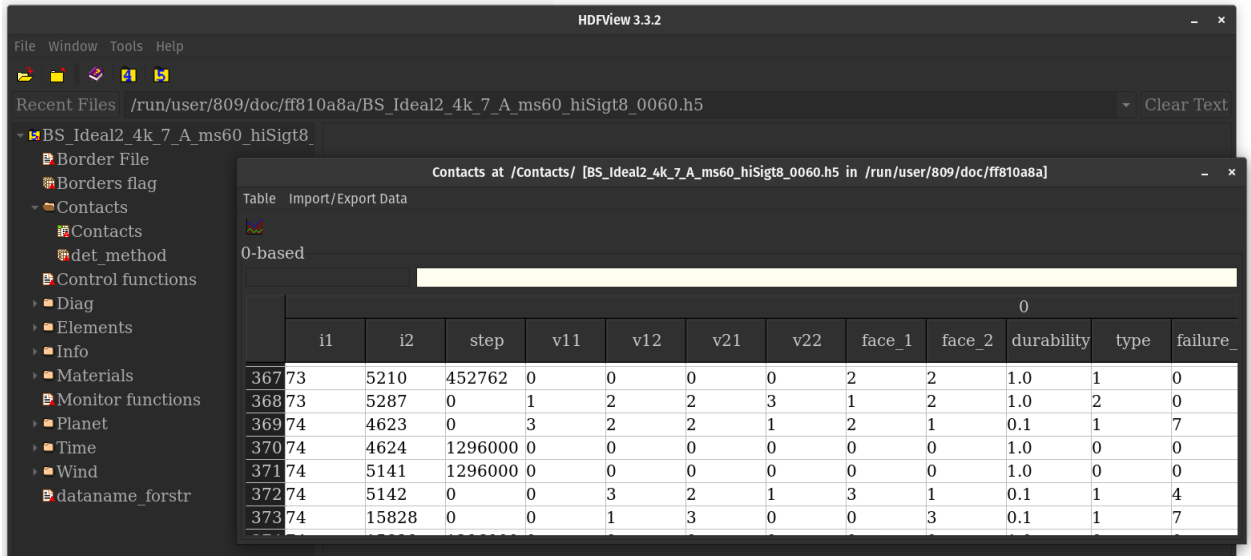


Figure 2: Example of model HDF5 file output as viewed with HDFView