Surface Integral Asteroid N-Body Simulator – Documentation

Alex Ho

Contents

1 Introduction

SIANS (Surface Integral Asteroid N-Body Simulator) is a software aimed to solve the full two-body problem, which is commonly done in order to study the dynamics of asteroids. The asteroids can take arbitrary shapes, either through ellipsoidal or with polyhedral shapes. For ellipsoidal shapes, the code is generalized as an N-body code, while for polyhedral shapes it is restricted to a two-body code.

The forces, torques and mutual potential energy are computed through the surface integration method outlined by Conway (2016) (see Sect. 5). The advantage of this method is that it is exact for bodies of ellipsoidal, and does not make any use of power series to expand the potential. Furthermore, this method also is also valid even when the bodies are close to each other.

The forces, torques and potential energies, using the surface integration scheme, can also be calculated without solving running the full simulations. These can be obtained from specified functions (see Sect. 3.6), and can be used to solve your own customised two-body or N-body problem.

1.1 Requirements

SIANS requires the following to run:

- Python3.6 or higher
- NumPy
- C compiler
- Cython
- GNU Scientific Library (GSL)

The following library is optional:

- GNU Multi Precision library (GMP) or Multiple Precision Integers and Rationals library (MPIR) – High precision computations.

SIANS has not been tested on older Python versions ($<$ 3.6). There is therefore no guarantee that SIANS will work on older Python versions.

2 Installation

This section describes how the necessary software can be installed. It is assumed that the reader has a working `Python` 3.X distribution (preferably 3.6 or newer) and NumPy installed.

2.1 Installing SIANS

The SIANS can be found in the github repository. To clone the files, first go to the directory where the program is to be saved and then use git, such as

```
$ cd some/folder/
$ git clone repository
```

The files will then be in `some/folder/SIANS/`. Once the files are downloaded, the `Cython` and `C` files will have to be configured before the program can be used (See Sect. 2.6).

2.2 C compiler

SIANS makes use of both `Cython` and libraries written in `C` to solve the simulations. Having a C-compiler is therefore required.

Linux

For most Linux machines, `gcc` should already be installed in the system. If not, try writing the following in the terminal

```
$ sudo apt-get install gcc
```

Mac OS

Most Mac machines should have a `gcc` compiler installed. If not, homebrew may be used to install gcc

```
$ brew install gcc
```

Note that OpenMP is utilized in SIANS . Having a library that allows their usage may be required. In OSX, this can be done by installing `llvm` and `libomp`

```
$ brew install llvm libomp
```

Windows

For Windows users, it is recommended to use Microsoft Visual Studio Community (MVSC)[1]. Alternatively, the `gcc` compiler can be installed from either from MinGW[2] or Cygwin[3]. However, there may be issues using `Cython` if either the MinGW or Cygwin compilers are used. MVSC is therefore recommended for Windows users.

2.3 Installing Cython

`Cython`[4] is extensively used in the code, both to speed up simple python code as well as acting as a wrapper for certain `C` routines. `Cython` can be installed via pip as

```
$ pip3 install cython
```

There may be issues with `gcc` and `Cython` when compiling the setup file. This may be due to a compatibility issues with `Cython` and `Python` 3.7. An easy workaround is to use an older version of `Cython` , e.g. the 0.29.6 or 0.29.5 version. This can be done by typing in the terminal:

```
$ pip3 install --upgrade cython==0.29.6
```

2.4 Installing GSL

The GNU Scientific Library (GSL)[5] contains a large variety of numerical libraries that are used in the SIANS . The main use of GSL is their numerical integration library, allowing for fast and efficient numerical integrations.

[1] For 2017/2019 version, see https://visualstudio.microsoft.com/vs/community/
[2] http://www.mingw.org/
[3] https://www.cygwin.com/
[4] https://cython.org/
[5] https://www.gnu.org/software/gsl/

2.4.1 Linux

To install GSL on windows, simply write the following in the terminal

```
$ sudo apt-get install libgsl-dev
```

2.4.2 Mac OS

GSL can be installed using homebrew, through

```
$ homebrew install gsl
```

which should install it into the `/usr/local/include/` folder.

2.4.3 Window - MVSC

If MVSC is the C compiler on the system, then GSL can be installed from https://www.bruot.org/hp/libraries/[6] as a `.zip` file. Make sure the GSL library correspond to the MVSC version that is installed on the system. Once the `.zip` file is installed, unpack it to a folder that is easily reachable. This location will be important for the setup of the `Cython` as well as the `C` files. The 32 bit GSL version 2.2 is currently supported for SIANS . For convenience, save the GSL files in the following folder

`C:/GSL/gsl_2_2_msvc2015_32/msvc2015_32/`

If GSL is installed in a different location, the `gsl_libdir` and `gsl_incdir` variables in the `setup.py` file must to be adjusted to match the relevant GSL folder location.

2.5 Installing GMP/MPIR (Optional)

GMP and MPIR allows for high precision computation. These libraries allows one to declare float point variables to an arbitrary precision. This is mainly used for polyhedron potential computation. Because of compatibility, MPIR is used as it can be ported to all OS platforms. Linux users may want to use GMP, as it is supported to almost all Linux distributions. Installation procedures are also described in the documentation of MPIR.

[6]Last accessed August 9. 2021.

2.5.1 Linux

The tar-ball of MPIR can be installed through

```
$ wget http://www.mpir.org/mpir-3.0.0.tar.bz2
```

Unpack the tar-ball

```
$ tar -xvf mpir-3.0.0.tar.bz2
```

Go into the mpir folder

```
$ cd mpir-3.0.0/
$ ./configure
$ make
$ sudo make install
```

The libraries should be saved in `/usr/local/lib`, which is the default library location in the `setup.py` file. Make sure that `/usr/local/lib` is added in the path. This can be done by adding the following line in the .bashrc file

```
export LD_LIBRARY_PATH=/usr/local/lib
```

2.5.2 MAC OS

MPIR can be installed using homebrew, through

```
$ homebrew install mpir
```

2.5.3 Windows

MPIR can be downloaded from their website http://www.mpir.org/. The current build of SIANS uses version 3.0.0 of MPIR. For the purpose of this documentation, the .zip file is downloaded and unpacked at the following directory:

`C:/MPIR/mpir-3.0.0/`

The build version depends on the version of Visual studio installed. For this documentation, Visual studio 2017 community version is used. The installation procedure is as follows

1. Go to the `build.vc15` folder and double click the file named `mpir.sln`.
2. Set the build to "Release" under the system "Win32" and build the `dll_mpir_gc` solution. This installs the required C files for MPIR.
    - Errors may occur where header files are missing. Retarget the solution file and set the Windows SDK version to 10.x.x.
3. Add the compiled libraries to PATH. The relevant folder is

    `C:/MPIR/mpir-3.0.0/build.vc15/dll_mpir_gc/Win32/Release/`

If MPIR is installed in a different location, `mpir_libdir` and `mpl_incdir` variables in the `setup.py` file must be changed accordingly.

2.6 Configure the software

Before the SIANS can be used, the `Cython` and `C` files must be configured. This is done by the `setup.py` file. To configure the files, write the following in the terminal:

```
$ cd SIANS/cymodules
$ python setup.py build_ext --inplace
```

The configuration requires the location of GSL `include/` and `lib/` folders. The default folder locations are set to be consistent with the installation procedure described in this document. If not, the variables `gsl_libdir` and `gsl_incdir` must be manually changed to match the installation location of GSL in the `setup.py` file.

Polyhedron simulations have issues with round-off errors in the potential. This can remedied by increasing the variable precision through MPIR. A separate folder contains the MPIR implementation

```
$ cd SIANS/cymodulesmpir
$ python setup.py build_ext --inplace
```

Similar to GSL, the include and lib folders of MPIR must be properly linked through the `mpir_libdir` and `mpir_incdir` variables in the corresponding `setup.py` file.

3 Basic usage

Once everything is configured, SIANS can be used to run the desired simulations. To run the program, simply write

```
$ python main.py –optional_arguments
```

Optional arguments to `main.py` include

- `-method` (string) – Selects numerical integrator to be used as the solver. Available options are:
  - `RK4`: Standard order 4 Runge-Kutta method.
  - `RK54`: (Default) Embedded Dormand-Prince Runge-Kutta method of order 5(4).
  - `V65`: Embedded Verner Runge-Kutta method of order 6(5).
  - `V76`: Embedded Verner Runge-Kutta method of order 7(6).
  - `V87`: Embedded Verner Runge-Kutta method of order 8(7).
  - `V98`: Embedded Verner Runge-Kutta method of order 9(8).
  - `T98`: Embedded Tsitouras Runge-Kutta method of order 9(8).
  - `F108`: Embedded Feagin Runge-Kutta method of order 10(8).
  - `F1210`: Embedded Feagin Runge-Kutta method of order 12(10).

  More details on the numerical methods can be found in Sect. 6.
- `-outfold` (string) – Set as any name. Specifies the location where the figures as well as final solution files are saved. Figures are saved in

  `SIANS/PlotResults/foldExtra/`

  The final output files are saved in

  `SIANS/OutputData/rundate/outfold/`.
- `-input` (string) – Filename (without `.txt`) of the input file containing global variables (e.g. time) as well as the shape parameters. If the argument `-foldExtra` is not used, then the folder name of the saved figures and output files will take the name of the input file.

- `-rundate` (string) – Set as any name or date. Specifies the folder where the Numpy files are saved. Default as `YYYY/MM_DD/` where YYYY, MM and DD represents the year, month and date respectively.
- `-delsol` (boolean) – If set `True`, then all output files in the selected output folder are deleted and the whole simulation is recomputed.
- `-units` (string) – Selects the units used in the simulation. This will rescale the gravitational constant G. Available options are:
  - `None` (string) – (Default) All units are dimensionless, such that $G = 1$.
  - `SI` – Units in m/s (meter, second), such that $G = 6.674 \cdot 10^{-11}$ $m^3 kg^{-1} s^{-2}$.
  - `kmhr` – Units in km/hr (kilometer, hour), such that $G = 6.674 \cdot 10^{-11} \cdot (3600^2/1000^3)$ km^3 kg^{-1} hr^{-2}.
  - `kms` – Units in km/s (kilometer, second), such that $G = 6.674 \cdot 10^{-11} \cdot (1/1000^3)$ km^3 kg^{-1} s^{-2}.

  The units for the initial conditions must be consistent with the `-units` usage. If units of km is used, then initial positions must be in km and not, e.g. meter. See Sect. 3.1.
- `-adaptive` – If set `False`, does not utilize adaptive time stepping for the Runge-Kutta methods. Default `True`.
- `-euler` (boolean) – If `True`, uses euler parameters instead of Tait-Bryan angles when the equations of motion are solved. Input angles are still given by Tait-Bryan angles, but this is converted before the equations of motion are solved. Default `False`
- `-mpir` (boolean) – If `True`, enables mpir to increase the variable precision for the polyhedron potential. Default `False`.
- `-doplot` (boolean) – If `True`, saves some basic plots (orbits and energies) in the relevant output folder. Default `False`
- `-solconv` (boolean) – If `True`, converts the output solution file to a more readable format. Default `False`.

- `-soltype` (string) – The format of the non-Numpy solution file. Can be `txt` (text) or **csv** file. See Sect. 3.2. Default `txt`.

Boolean parameters that take `True` or `False` statements can be written in the following way:

- `True` set as: `true, t, yes, y, 1`
- `False` set as: `false, f, no, n, 0`.

The program does not distinguish between lower and capital letters, so e.g. `True` or `TRUE` can also be used.

3.1 Input file

SIANS reads a *.txt* file as an input file, which contains the global parameters, shape parameters and initial conditions of the bodies. All input parameter files must be saved within the `initial_values/` folder.

Any lines that start with # or ### (i.e. single or triple hash tags) are considered as a comment. Anything other combination of # will cause problems for the file reader. The first 11 non-commented lines, which determines the global parameters used for the main simulation, must be in the following order:

1. `t_end:` (float) – End time of the simulation t_{end}
2. `N:` (integer) – The number of sampling points and computation steps (including time zero). For instance, `N: 5` result in

$$t = [t_0, t_1, t_2, t_3, t_4] \tag{1}$$

   where $t_0 = 0$. `N` must take a value greater than zero. For adaptive time steppers, the number of sampling points will automatically increase if `N` is too small. For non-adaptive time steppers, the time step is

$$\Delta t = \frac{t_{\text{end}}}{N-1}. \tag{2}$$

3. `ntol:` (float) – Error tolerance for the Runge-Kutta methods. Mainly used for adaptive time steppers. If set to 0, uses default values described in their respective sections (See Sect. 6).

4. `iabstol:` (float) – Absolute tolerance for numerical quadrature. This tolerance is used for the integration required to compute the forces and torques. Default value is 10^{-8}.

5. `ireltol:` (float) – Relative tolerance for numerical quadrature. Default value is 10^{-8}.

6. `quadval:` (integer) – This parameter selects the integration rule used for the numerical quadrature routine. Higher value gives a more accurate result, but at the cost of computation speed. Can take values 1 to 6. See https://www.gnu.org/software/gsl/doc/html/integration.html#c.gsl_integration_qag for more details.

7. `hmax:` (float) – The max time step increase used for the adaptive time steppers.

8. `hmin:` (float) – The minimum step size is allowed to take. Note that the step size may become slightly smaller than this given value.

9. `kepler:` (integer) – If set to 0, assumes Cartesian coordinate (position and velocity) as initial condition. If set to 1 or 2, the input data will assume orbital elements as initial condition instead of Cartesian coordinates.

10. `sun:` (integer) – Determines whether the Sun is included in the simulation or not. If set to 1, the Sun is included in the simulation acting with an external gravitational force on the other bodies. The Sun is set to be at the origin of the simulation, and not considered as an additional body. Does not include the Sun if set to 0.

11. `density:` (integer) – If set to 1, values of the masses are considered as densities. If set to 0, the values are standard masses. Units must be consistent with the `-units` call.

Here, the parameter X can take any float/double value (with the exception of `N`, `quadval`, `kepler`, `sun` and `density`). Note that the semicolon `:` must be included behind these global parameters before the value is given.

The shape parameters, as well as the initial conditions, comes after the global parameters. The initial condition can either be with Cartesian coordinates (position and velocity) or Keplerian coordinates (orbital elements). The parameter `kepler` selects this. The units of all parameters must be consistent

with the `-units` usage in Sect. 3. For instance, if `-units kmhr` is enabled, then the semiaxes and positions must also be in kilometers, velocities with units kilometers per horu, and so on.

3.1.1 Cartesian

For `kepler: 0`, Cartesian coordinates are used. The shape parameters and initial conditions must be ordered as follows:

m		
a	b	c
x	y	z
v_x	v_y	v_z
ϕ	θ	ψ
ω_x	ω_y	ω_z

Here, the parameters correspond to

- m – Mass of the body. Must be larger than zero.
- a, b, c – Semiaxes of the ellipsoid, which spans to the x, y, z-axes respectively. If the body is considered as a point mass, *all three* semiaxes for said body must be zero. For polyhedron data, one can replace these with the object filename of the polyhedron.
- x, y, z – Initial positions of the bodies.
- v_x, v_y, v_z – Initial velocities, along the x, y, z-directions, of the bodies.
- ϕ, θ, ψ – Initial rotation angles in degrees. These angles follows the Tait-Bryan angle convention. They correspond to a rotation around the x, y, z-axes of the body respectively[7]. See Sect. 5.3.
- $\omega_x, \omega_y, \omega_z$ – Initial angular velocity of the bodies in the body-fixed frame. Corresponds to an angular speed around the x, y, z-axes of the body respectively.

Every additional row to each parameter correspond to an additional body to be added. At least two rows of initial values must be present in order to have a two-body simulation. The program will not run if the number of rows

[7]For instance, changing ϕ would be a rotation about the x-axis, which correspond to the y and z-axes to change direction, while the x-axis points the same direction.

is inconsistent among all parameters (see Sect. 3.1.7). Furthermore, a new parameter must be indicated with an @ sign (see Sect. 3.1.6).

3.1.2 Keplerian

For `kepler: 1` or `kepler: 2`, orbital parameters are used are used. The orbital parameters are then converted to Cartesian coordinates, which are then used to run the simulation. The conversion procedure is described in Sect. 5.4. For `kepler: 1` the shape parameters and initial conditions must be ordered as follows:

m		
a	b	c
Ω	$\bar{\omega}$	λ
D	e	i
ϕ	θ	ψ
ω_x	ω_y	ω_z
BID		

Here, the parameters correspond to

- m – Mass of the body. Must be larger than zero.
- a, b, c – Semiaxes of the ellipsoid, which spans to the x, y, z-axes respectively. If the body is considered as a point mass, *all three* semiaxes for said body must be zero. For polyhedron data, the line can be replaced with the object filename of the polyhedron.
- D – Orbital period in days.
- e – Eccentricity.
- i – Inclination in degrees.
- Ω – Ascending node in degrees.
- $\bar{\omega}$ – Longitude of the pericenter in degrees.
- λ – Mean longitude in degrees.
- ϕ, θ, ψ – Initial rotation angles in degrees. These angles follows the Tait-Bryan angle convention. Same as the Cartesian case.

- $\omega_x, \omega_y, \omega_z$ – Initial angular velocity of the bodies. Same as the Cartesian case.
- BID (binary ID) – Used to determine whether the body is a satellite or not. If set to 0, the body is considered a primary. If set to 1, the body is considered a satellite of the most recent body with BID = 0. See Sect. 3.1.4 for explaination.

For `kepler: 2`, the only change is the orbital period D and mean longitude λ. They are replaced with the semimajor axis a_s (not to be confused with a used as one of the semiaxes of an ellipsoid) and mean anomaly M_0 respectively. The initial conditions are then ordered as:

m		
a	b	c
Ω	$\bar{\omega}$	M_0
a_s	e	i
ϕ	θ	ψ
ω_x	ω_y	ω_z
BID		

Most parameters are the same as in `kepler: 1`. The only changes are:

- a_s - Semimajor axis. The unit must be consistent with the usage of `-units` parameter used to run the program. If kilometer scale is considered, the semimajor axis must be converted to kilometers and not, e.g. meters.
- M_0 - Mean anomaly in degrees

3.1.3 Units

The units of all parameters must be consistent with the `-units` input in the terminal. For instance, if `-units SI` is used, meters (m) and seconds (s) are considered. The initial positions and velocities have units m and m/s respectively. The angular velocities have units of rad/s (radians). If `density: 1`, the density units are kg/m^3. The ellipsoid semiaxes are also given in meters.

If `-units kmhr`, then kilometers (km) and hours (hr) are considered. The initial positions and velocities have units km and km/hr respectively. The

angular velocities have units of rad/hr. If `density: 1`, the density units are kg/km^3. Semiaxes lengths now have units km.

3.1.4 Binary IDs

(Only relevant if `kepler` is set to 1 or 2)

The parameter BID used in the input file determines whether a body is a satellite or not. If set to 0, the body is considered a primary. If set to 1, the body is considered a satellite of the most recent body with BID = 0. For example, consider five bodies A,B,C,D,E with their respective BIDs:

- A - 0
- B - 1
- C - 1
- D - 0
- E - 1

In this case, Body A and D would be considered primary bodies while B, C and E would be satellites. In this scenario, body B and C would be satellites of A while body E would be a satellite of D.

3.1.5 Special symbols

Certain symbols can be used to give a more precise input or to conveniently set the initial conditions.

Multiplication and Division

Multiplications and divisions can be done in the input file. An arbitrary number of multiplications can be used, but the number of divisions is limited to one. For instance, the following numbers will work

- `1.0*3`
- `1*2*3*4*5*6`
- `(2*2*5)/(3*2*2)`

while the following will not work

- `(2*2)/(5*2)/3`

Minus signs and π

The symbol π, as well as minus signs can also be used in the input file. The symbol π must be represented as "pi" in the text file. Example usage of these include

- `2*pi`
- `-2*pi`
- `(-2*pi)/(-3*5)`

Exponents

If large numbers are considered, such as masses of order 10^x, the exponent can be specified in the input file. To do so, simply multiply the number with `1eX`. For instance, the Earth's mass in kilograms can be written as

- `5.972*1e24`.

Negative powers are also understood, e.g.

- `5*1e-3`.

3.1.6 Parameter separation

SIANS will read through the input file line by line to gather data. However, it is unable to separate different parameters unless it is specified by an @ sign. For instance, using `kepler: 0`, the parameters must be separated as:

```
@ Masses
(data)

@ Semiaxes
(data)

@ Initial positions
(data)

@ Initial velocities
(data)

@ Initial angles
(data)
```

```
15
16 @ Initial angular velocities
17 (data)
18
19 @ BID (if kepler = 1 or 2)
20 (data)
```

The labelling after the @ sign is not required, but is included here for illustration purposes. SIANS will terminate if the number of @ separations is inconsistent with the `kepler` choice.

3.1.7 Additional bodies

Additional rows to each input parameter correspond to an additional body added to the simulation. For instance, the following corresponds to the masses and semiaxes of two bodies

```
1 @ Masses
2 2.0
3 2.0
4
5 @ Semiaxes
6 1.0 0.8 0.7
7 0.5 0.3 0.2
```

To add an additional body, simply add an additional line of data for each parameter

```
1 @ Masses
2 2.0
3 2.0
4 3.0
5
6 @ Semiaxes
7 1.0 0.8 0.7
8 0.5 0.3 0.2
9 1.2 1.2 0.8
```

Keep in mind that this must be done for every parameter line, such as the positions, velocities, etc.

3.1.8 Comments

Certain lines may be commented out and SIANS will ignore the data. The comment syntax is similar to Python, in which a hashtag (#) is used to comment out lines. For instance, mass value of 40.0 is ignored

```
@ Masses
2.0
#40.0
2.0

@ Semiaxes
1.0 0.8 0.7
0.5 0.3 0.2
```

The *example.txt* file provides a full example on how the input file is structured. This input file can be used for testing purposes. A separate input file with a different name may also be created, which can be convenient to differentiate different test simulations.

3.2 Output file

The output files are Numpy binary files with the extension `.npz`. These can only be read with the `numpy.load` command, and a function dedicated to reading these solution files is named `read_solution`, which is found in the `readsolution.py` script. The Numpy solution files are mainly used to be analysed with a Python script. See Sect. 3.4.4 for an example.

However, if a standard text file is desired, it can be enabled with the `-solconv` command in the terminal (see Sect. 3). The Numpy solution file will then be converted to a `.csv` file or `.csv` file, which can be read through standard means.

The number of rotation angles will depend on the rotation matrix convention used. For this reason, the `.txt` and `.csv` files will save rotation matrix elements of the bodies while the Numpy solution files save the rotation angles. The rotation matrix conventions used are detailed in Sect. 5.3.

The columns of the `.txt` and `.csv` have the following structure. For a simulation with $N \geq 2$ bodies, the parameters that describes the orbit of body $i = 0, 1, 2, ..., N-1$ are given by the following columns in the `ODE_solution` file

- **Col** 0: Time
- **Col** $3i + 1$: Position x
- **Col** $3i + 2$: Position y
- **Col** $3i + 3$: Position z
- **Col** $3N + 3i + 1$: Velocity x
- **Col** $3N + 3i + 2$: Velocity y
- **Col** $3N + 3i + 3$: Velocity z
- **Col** $6N + 3i + 1$: Angular velocity x
- **Col** $6N + 3i + 2$: Angular velocity y
- **Col** $6N + 3i + 3$: Angular velocity z
- **Col** $9N + 9i + 1$: R_{00} rotation matrix component
- **Col** $9N + 9i + 2$: R_{01} rotation matrix component
- **Col** $9N + 9i + 3$: R_{02} rotation matrix component
- **Col** $9N + 9i + 4$: R_{10} rotation matrix component
- **Col** $9N + 9i + 5$: R_{11} rotation matrix component
- **Col** $9N + 9i + 6$: R_{12} rotation matrix component
- **Col** $9N + 9i + 7$: R_{20} rotation matrix component
- **Col** $9N + 9i + 8$: R_{21} rotation matrix component
- **Col** $9N + 9i + 9$: R_{22} rotation matrix component

The column syntax follows `Python` and `C` array syntaxing. For instance, column 0 (time) corresponds to the first column in the output file. The rotation matrix elements are ordered as

$$R = \begin{bmatrix} R_{00} & R_{01} & R_{02} \\ R_{10} & R_{11} & R_{12} \\ R_{20} & R_{21} & R_{22} \end{bmatrix}. \tag{3}$$

The energies are found in the `Energies` file

- **Col** 0: Potential energy
- **Col** 1: Total linear kinetic energy
- **Col** 2: Total rotational kinetic energy

3.3 Polyhedron shapes

The bodies may take polyhedron shapes. Keep in mind that the polyhedron shapes must have triangular faces. The vertices and face indices must be given through a separate `.obj` file, and can be read as

```
@ Masses
2.0
2.0

@ Semiaxes
1.0 0.8 0.7
polyhedron.obj
```

where `polyhedron.obj` is the polyhedron file. In this example, body 1 is an ellipsoid while body 2 is a polyhedron.

The `.obj` file must be in the following form

```
v x1 y1 z1
v x2 y2 z2
v x3 y3 z3
...
v xN yN zN
f d1 d2 d3
f d1 d2 d4
...
```

Here, the three numbers after `v` are the coordinates of the vertices, while the three numbers after `f` are the vertex indices making up the triangular face.

We remind the reader that the software is limited to a two-body problem for any number of polyhedral shapes. For instance two ellipsoids and a polyhedron will not work for the software.

3.4 Examples

3.4.1 Saving in a specific folder

We want to run the simulation and save our results in the folder called `TestRun/`. This can be done by running the program as

```
$ python main.py --outfold TestRun
```

The solution files ares saved in

`SIANS/Outputdata/YYYY/MM_DD/TestRun/`

The data output folder can be specified to a specific name instead of the rundate, such as

```
$ python main.py --outfold TestRun --rundate Specificfolder
```

The solution files are then saved to

`SIANS/Outputdata/Specificfolder/TestRun/`

Using this can be convenient in order to keep track of complete simulations instead of test simulations.

If some basic plots are desired from the simulation, which includes plots of the orbits and energies, then it can be enabled by

```
$ python main.py --outfold TestRun --doplot t
```

The figures are saved in the `SIANS/PlotResults/TestRun/` folder.

3.4.2 Higher order method and overwriting old files

We now want to run the simulation using a higher order method, such as Verner's order 9(8) method. Similar to the previous example, we want to save this in the folder `TestRun/`. However, because we already have the simulation saved in the `TestRun/` folder, the program will read the final Numpy files and no new results will be obtained. We therefore have to remove the old Numpy files in order to recompute everything. This can be done as

```
$ python main.py --outfold TestRun --method V98 --delsol t
```

If we instead want to use the Feagin's order 12(10) method, we can instead write

```
$ python main.py -outfold TestRun -method F1210 -delsol t
```

3.4.3 Specifying input file

SIANS reads input files with global parameters, shape parameters and initial conditions. Consider an input file named `example.txt`. We now want to save this in a different folder `TestFolder2/` using the V98 method. Because this folder is brand new, no Numpy files needs to be deleted[8]. The SIANS can be run as

```
$ python main.py -foldExtra TestRun -method V98 -input example
```

3.4.4 Reading solution file

The Numpy solution files can be read with the function `read_solution`, and the energies from `read_params`, both found in the module `readsolution.py`.

```
import readsolution
sol,elparams,masses = readsolution.read_solution(arguments*)
U, Ek, Erot, Etot = readsolution.read_params(arguments*)
```

Function arguments for `read_solution` and `read_params` are described in Sect. 4.1 and Sect. 4.2 respectively.

Let us now consider a simple test simulation

```
$ python main.py -foldExtra TestRun -method RK54 -input example
  -rundate Examplefold
```

The output data is saved in

`SIANS/Outputdata/Examplefold/TestRun/`

This can be read in this example python code

[8]The argument `-delsol True` may still be used, but since no solution files exist, the program will run as a brand new simulation.

```
import readsolution

sol,elparams,masses = readsolution.read_solution("TestRun",
    rundate="Examplefold")
```

The parameter `sol` is a class instance that contains the computed values of position, velocity, angles and angular velocity of the bodies (more details below). `elparams` contains the ellipsoid semiaxes and `masses` the masses of the bodies. If the output data is saved in a different location, e.g.

`some/folder/SIANS/Outputdata/Examplefold/TestRun/`

then it must be specified as

```
import readsolution
New_save_folder = "some/folder/SIANS/Outputdata/"
sol,elparams,masses = readsolution.read_solution("TestRun",
    rundate="Examplefold", foldloc=New_save_folder)
```

The variables of `sol` can be unpacked as

```
time = sol.t
N_bodies = sol.N_bodies # Number of bodies in system
vx = sol.vx   # Velocity x-component
vy = sol.vy   # Velocity y-component
vz = sol.vz   # Velocity z-component
x = sol.x     # x-position
y = sol.y     # y-position
z = sol.z     # z-position
omegax = sol.omegax   # Angular velocity x-component
omegay = sol.omegay   # Angular velocity y-component
omegaz = sol.omegaz   # Angular velocity z-component
# If using Tait-Bryan angles
phi = sol.phi     # Rotation angle wrt. x-axis
theta = sol.theta # Rotation angle wrt. y-axis
psi = sol.psi     # Rotation angle wrt. z-axis
# If using Euler parameters
e0 = sol.e0
e1 = sol.e1
e2 = sol.e2
e3 = sol.e3
```

These parameters include the computed variables, for all time steps and for

all bodies. For instance, if one wants the x-positions of body 1 and N[9], one can write

```
x_body1 = x[0]
x_bodyN = x[N_bodies-1]
```

If the solution files are saved in a readable format (achieved when the option `-solconv` is set to `True`), then the files are found in same folder as the Numpy solution files.

3.4.5 Multi-solution files

The software has a hard-coded limit on the number of time steps that is saved into the solution file. If the number of time steps exceeds this limit, the current parameters will be dumped into a solution file, and the variables are reset. When the variables are reset, the initial conditions are equal to the final time step before the variable reset. The only thing that changes is the number of solution files and their format. Multiple solution files are only given in the Numpy binary format, and not `.txt` or `.csv` formats.

To read these multi-solution files, one can use the `ReadMultiSolutions` class from the `readsolution.py` file

```
import readsolution
sol = readsolution.ReadMultiSolutions(arguments*)
```

Arguments used are outlined in Sect. 4.3. The following time step limits are imposed. These limits can be changed in the `ODESolvers.pyx` file.

```
if NBodies <= 4
  Max steps = 500000
if NBodies > 4 and NBodies <= 15:
  Max steps = 100000
If NBodies > 15:
  Max steps = 50000
```

Let us consider the following test simulation for a two-body problem, using the Tait-Bryan rotation matrix convention.

```
$ python main.py --foldExtra TestRun --method RK54 --input bigexample
  --rundate Examplefold
```

[9]Here, we assume that body 1 is first body in the simulation, therefore index 0 in `Python`.

Assume that the number of time steps far exceeds the imposed limits above, then multiple `Solutions.npz` and `EnergiesSol.npz` files are saved in the folder `SIANS/Outputdata/Examplefold/TestRun/`. The solution file $i = 1$ files can be read by

```
import readsolution
i = 1 # or a different number
N_bodies = 2
use_eulerP = 0  # Set 1 if Euler parameters are used.
sol = readsolution.ReadMultiSolutions("TestRun", i, N_bodies,
    use_eulerP, rundate=Examplefold)
```

The class instance `sol` now contains the parameters of the first 500000[10] time steps. If $i = 2$, then it is the parameters of the next 500000 time steps, and so on.

3.5 Executing from a python script

The file `exampleruns.py` contains various examples on how to execute SIANS from a python script. This can be a convenient approach if a large list of simulations are to be run through a for loop or through parallel processing.

3.6 Obtaining forces, torques and mutual potential energy

The output files do not save the forces or torques computed during the simulations. Obtaining the forces or torques bodies of arbitrary shapes at any time step or configuration can be achieved by using the `getforce` function found in the `other_functions_cy.py` file. These functions can be called from `Python`, and can be used to calculate the gravitational forces for your own defined systems. The forces and torques computed from this function are based on the surface integration scheme (see Sec 5.1), and is applicable for bodies of arbitrary shapes.

The function can be imported as

```
import cymodules.other_functions_cy as OFcy
# Or if mpir is enabled
# import cymodulesmpir.other_functions_cy as OFcy
Fx, Fy, Fz = OFcy.getforce(arguments*)
```

[10] Or a different limit if this is changed

The arguments of the `getforce` function are listed in Sect. 4.4.

For similar reasons, one can also obtain the mutual potential energy, using the surface integration method, in a similar manner. This is found in the `getpotential` function in the `other_functions_cy.py` file.

```
import cymodules.other_functions_cy as OFcy
# Or if mpir is enabled
# import cymodulesmpir.other_functions_cy as OFcy
U = OFcy.getpotential(arguments*)
```

The arguments of the `getpotential` function are listed in Sect. 4.5.

Issues with the surface integration arises when large numbers are involved. This is remedied by scaling the mass units down to unity, which is automatically taken care of in the function itself when the masses and densities are provided as input argument. See Sect. 5.5.

Note: For polyhedral shapes, because the surface integration scheme is performed in the body-fixed frame of the integrated body, the vertices of each polyhedra must be located so that the centroids of the respective polyhedra are located at the origin.

The function `obtain_force_U` in the file `exampleruns.py` contains various examples on how the forces can be obtained through this function.

4 Function documentations

This section contains information on the arguments for certain functions that may be of use after the software has completed the simulations. Note that variable type `float` and `int` correspond to 64 bit float point precision and 32 bit integer precision respectively. The `str` format is a string (text). Boolean types (`bool`) only accepts `True` or `False` input.

4.1 read_solution

This function reads the final solution files. The function is located in the `readsolution.py` file.

```
Required input:
-- foldname (str):
   Name of the folder where the solution file is saved, equal
    to the -outfold input argument in the terminal described
    in Sect. 3.1.

Optional arguments:
-- rundate (str):
   The date folder, specified by the -rundate input command in
     the terminal described in Sect. 3.1.

-- simplify (bool):
   If True (default), sorts the solution class instance in a
    convenient format.
   False is only used during the simulation run before energy
    computation.

-- foldloc (str):
   Specifies the location where OutputFolder/ is located.

-- extension (str):
   Solution files may have appended text to the filename. This
     argument specifies the text appended to the ODE_Solution.
    npz file.

-- skipread (bool):
   If True, does not read the solution file and program
    proceeds as normal. The returned solution class instance
    is empty.
   Defaults to False
```

```
Output:
-- sol (class instance):
   The solution class instance that contains the positions,
    velocities, angular velocities and rotation angles.
-- elparams (float, array):
   An Nx3 array, for N bodies, that contains the semiaxes of
    the bodies.
-- masses (float, array):
   An array that contains the masses of the bodies
```

4.2 read_params

This function is mainly used to read the energy files. The function is located in the `readsolution.py` file.

```
Required input:
-- foldname (str):
   Name of the folder where the solution file is saved, equal
    to the -outfold input argument in the terminal described
    in Sect. 3.1.

Optional arguments:
-- rundate (str):
   The date folder, specified by the -rundate input command in
     the terminal described in Sect. 3.1.

-- foldloc (str):
   Specifies the location where OutputFolder/ is located.
Output:
-- U (float, array):
   Potential energy computed by the surface integration method
    . See Sect. 5.1.
-- Ek (float, array):
   The total linear kinetic energy of the system.
-- Erot (float, array):
   The total rotational kinetic energy of the system.
-- Etot (float, array):
   The total energy of the system.
```

4.3 ReadMultiSolutions

This function reads solution files that are separated into multiple parts, in order to prevent memory overflow. The function is located in the `readsolution.py`

file.

```
Required input:
-- foldname (str):
   Name of the folder where the solution file is saved, equal
    to the -outfold input argument in the terminal described
    in Sect. 3.1.

-- index (int):
   The i'th solution file that is split. For instance, if
    there are following files:
   Solutions1.npz
   Solutions2.npz
   Solutions3.npz
   Then index=2 choses Solutions2.npz.

-- N_bodies (int):
   Specifies the number of bodies in the simulation

-- use_eulerP (int):
   If set to 1, indicates that Euler Parameters are used to
    describe the rotation matrices. If set to 0, uses Tait-
    Bryan angles. See Sect. 5.3.

Optional arguments:
-- rundate (str):
   The date folder, specified by the -rundate input command in
     the terminal described in Sect. 3.1.

-- simplify (bool):
   If True (default), sorts the solution class instance in a
    convenient format.
   False is only used during the simulation run before energy
    computation.

-- foldloc (str):
   Specifies the location where OutputFolder/ is located.

-- mergeall (bool):
   If True, merges every single split solution file to the
    class instance.
   Not recommended for very large simulations.
   Default False

Output:
-- sol (class instance):
```

```
  The solution class instance that contains the positions,
   velocities, angular velocities and rotation angles.
```

4.4 getforce

This function is used to obtain the forces or torques for some arbitrary configuration. The function is located in the `other_functions_cy.pyx` file.

Note: The force and torque vectors are given in the body-fixed frame of the integrated body (body 1). To obtain these vector quantities in the inertial frame, apply the rotation matrix on the resulting force/torque vector. To obtain the torque on the second body, (body 2), simply switch the parameters around.

```
Required input:
-- pos (float, array):
  Position of the two bodies given in order:
  [x1, y1, z1, x2, y2, z2]

-- angles (float, array):
  Rotation angles of the bodies given in order:
  [phi1, theta1, psi1, phi2, theta2, psi2]
  Rotation angles follow the z-y-x convention (Tait-Bryan)
  Must not be Euler parameters/Quarternions

-- massdens (float, array):
  Masses and densities of the bodies [m1, m2, rho1, rho2]

-- semiaxes (float, array):
  Ellipsoid semiaxes given as [a1, b1, c1, a2, b2, c2]
  Semiaxes are ignored if the body is a polyhedron
  Semiaxes must have shapes
  a > b > c
  or
  a = b > c
  or
  a = b < c
  else treated as a point mass.

-- nfaces (int, array):
  The number of faces of the polyhedron [n1, n2].
  Treated as ellipsoid if n = 0
-- nverts (int, array):
  Number of vertices of each body [N1, N2].
```

```
-- vertices (float, array):
   The vertices of the polyhedron, ordered as:
   [vx1_1, vy1_1, vz1_1,
    vx1_2, vy1_2, vz1_2,
    ...,
    vx1_N1, vy1_N1, vz1_N1,
    vx2_1, vx2_1, vz2_1,
    vx2_2, vx2_2, vz2_2,
    ...,
    vx2_N2, vy2_N2, vx2_N2]
   That is, vertices of body 1 in order [x,y,z,x,y,z,...],
    then following for body 2.
   Number of vertices must be equal 3*verts
   Coordinates of the vertices must be so that the centroid is
     at the origin.

-- face_ids (int, array):
   Vertex indices that make up each face of the polyhedron on
    each body.
   Structure similar to to vertices

-- FM_check (int):
   If set to 1, computes the forces.
   If set to 2, computes the torques.

-- G_grav (float):
   The gravitational constant. Units must be consistent with
    input positions.

Optional arguments:
-- eabs (float):
   Absolute error limit for integration. See iabstol in Sec
    3.1
-- erel (float):
   Relative error limit for integration. See ireltol in Sec
    3.1
-- qval (int):
   Integration key. See quadval in Sect. 3.1.
-- roundoff (int):
   If set to 1, force values smaller than 10^(-16) are rounded
     down to zero.

Output:
-- Fx (float):
```

```
  Force/Torque along x-direction
-- Fy (float):
  Force/Torque along y-direction
-- Fz (float):
  Force/Torque along z-direction
```

4.5 getpotential

This function is used to obtain the mutual potential energy for some arbitrary configuration. The function is located in the `other_functions_cy.pyx` file.

```
Required input:
-- pos (float, array):
  Position of the two bodies given in order:
  [x1, y1, z1, x2, y2, z2]

-- angles (float, array):
  Rotation angles of the bodies given in order:
  [phi1, theta1, psi1, phi2, theta2, psi2]
  Rotation angles follow the z-y-x convention (Tait-Bryan)
  Must not be Euler parameters/Quarternions

-- massdens (float, array):
  Masses and densities of the bodies [m1, m2, rho1, rho2]

-- semiaxes (float, array):
  Ellipsoid semiaxes given as [a1, b1, c1, a2, b2, c2]
  Semiaxes are ignored if the body is a polyhedron
  Semiaxes must have shapes
  a > b > c
  or
  a = b > c
  or
  a = b < c
  else treated as a point mass.

-- nfaces (int, array):
  The number of faces of the polyhedron [n1, n2].
  Treated as ellipsoid if n = 0
-- nverts (int, array):
  Number of vertices of each body [N1, N2].

-- vertices (float, array):
  The vertices of the polyhedron, ordered as:
  [vx1_1, vy1_1, vz1_1,
```

```
   vx1_2, vy1_2, vz1_2,
   ...,
   vx1_N1, vy1_N1, vz1_N1,
   vx2_1, vx2_1, vz2_1,
   vx2_2, vx2_2, vz2_2,
   ...,
   vx2_N2, vy2_N2, vx2_N2]
  That is, vertices of body 1 in order [x,y,z,x,y,z,...],
   then following for body 2.
  Number of vertices must be equal 3*verts
  Coordinates of the vertices must be so that the centroid is
    at the origin.

-- face_ids (int, array):
  Vertex indices that make up each face of the polyhedron on
   each body.
  Structure similar to to vertices

-- G_grav (float):
  The gravitational constant. Units must be consistent with
   input positions.

Optional arguments:
-- eabs (float):
  Absolute error limit for integration. See iabstol in Sec
   3.1
-- erel (float):
  Relative error limit for integration. See ireltol in Sec
   3.1
-- qval (int):
  Integration key. See quadval in Sect. 3.1.

Output:
-- U (float):
  The mutual potential energy
```

4.5.1 Polyhedron vertex/face id ordering

The vertex and face id ordering of the functions `getforce` and `getpotential` follow a specific form. Consider a polyhedron given by the `polyhedron1.obj` file

```
v x1 y1 z1
v x2 y2 z2
...
```

```
v xN yN zN
f d1 d2 d3
f d1 d2 d4
...
```

To obtain the force between an ellipsoid and a polyhedron, the `vertices` and `face_ids` arrays must be ordered as

```
vertices = np.array([x1, y1, z1, x2, y2, z2, ..., xN, yN, zN
    ], dtype=np.float64)
face_ids = np.array([d1, d2, d3, d1, d2, d3, ...], dtype=np.
    int32)
```

Consider a second polyhedron given by the `polyhedron2.obj` file

```
v k1 m1 n1
v k2 m2 n2
...
v kN mN nN
f e1 e2 e3
f e1 e2 e4
...
```

If we are to integrate over `polyhedron1`, the `vertices` and `face_ids` arrays must be ordered as

```
vertices = np.array(
      [x1, y1, z1, x2, y2, z2, ..., xN, yN, zN,
       k1, m1, n1, k2, m2, n2, ..., kN, mN, zN],
       dtype=np.float64)
face_ids = np.array(
      [d1, d2, d3, d1, d2, d3, ...,
       e1, e2, e3, e1, e2, e4, ...],
       dtype=np.int32)
```

If we instead are to integrate over `polyhedron2`, the ordering vertices and face ids is reversed.

5 Mathematical framework

5.1 Equations of motion

The force and torque applied on a non-spherical object is computed through the surface integration procedure outlined by Conway (2016). Consider two non-spherical bodies A and B. The force on body A in the gravitational field of B is computed as

$$\mathbf{F}_A = \rho_A \iint_{S_A} \Phi_B(\mathbf{r}')\mathbf{n}dS \tag{4}$$

and the torque computed as

$$\mathbf{M}_A = -\rho_A \iint_{S} \Phi_B(\mathbf{r})\mathbf{n} \times \mathbf{r}dS. \tag{5}$$

The surface integrals are calculated using the QAG quadrature algorithm implemented by the GNU Scientific Library (Galassi et al., 2002). The equations of motion are solved as an initial value problem though a Runge-Kutta method (see Sect. 6).

The energies of the system are computed after the equations of motion are solved. The kinetic energies are solved through standard procedures, while the potential energy is calculated as

$$U = \frac{\rho_A}{3} \iint_{S_A} \left(\mathbf{r}\Phi_B(\mathbf{r}) - \frac{1}{2} |\mathbf{r}|^2 \mathbf{g}_B(\mathbf{r}) \right) \cdot \mathbf{n}dS. \tag{6}$$

A more detailed description on the method and how things are computed is outlined Ho et al. (2021).

5.2 Gravitational potential

The gravitational potential Φ depends on the shape of the body. For an ellipsoid, the gravitational potential is outlined in MacMillan (1930), while for a polyhedron, the formulation is based on the work of Conway (2015).

5.3 Rotational Motion

The rotational motion of the bodies can be described through rotation angles. The convention used in the program is the Tait-Bryan angles (ϕ, θ, ψ), and the corresponding rotation matrix is given by

$$\mathcal{R}_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi \\ 0 & \sin\phi & \cos\phi \end{bmatrix} \tag{7}$$

$$\mathcal{R}_y = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix} \tag{8}$$

$$\mathcal{R}_z = \begin{bmatrix} \cos\psi & -\sin\psi & 0 \\ \sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{9}$$

$$\mathcal{R}(\phi,\theta,\psi) = \mathcal{R}_z\mathcal{R}_y\mathcal{R}_x. \tag{10}$$

The rotation angles (ϕ, θ, ψ) change over time, and the following kinematic equations relate the time rate of change of these angles to the angular velocity of the body

$$\dot{\phi} = \hat{\omega}_x + (\hat{\omega}_y \sin\phi + \hat{\omega}_z \cos\phi)\tan\theta \tag{11}$$

$$\dot{\theta} = \hat{\omega}_y \cos\phi - \hat{\omega}_z \sin\phi \tag{12}$$

$$\dot{\psi} = (\hat{\omega}_y \sin\phi + \hat{\omega}_z \cos\phi)\sec\theta. \tag{13}$$

It should, however, be noted that there is a singularity associated to Eqs. (11) and (13) when $\theta = n\pi/2$, for any non-zero integer n. To avoid this singularity, one can use Euler parameters (e_0, e_1, e_2, e_3) to determine the rotational motion. he rotation matrix for Euler parameter convention is given by

$$\mathcal{R} = 2\begin{bmatrix} e_0^2 + e_1^2 - \frac{1}{2} & e_1e_2 - e_0e_3 & e_1e_3 + e_0e_2 \\ e_1e_2 + e_0e_3 & e_0^2 + e_2^2 - \frac{1}{2} & e_2e_3 - e_0e_1 \\ e_1e_3 - e_0e_2 & e_2e_3 + e_0e_1 & e_0^2 + e_3^2 - \frac{1}{2} \end{bmatrix}. \tag{14}$$

The Euler parameters have the following constraint:

$$e_0^2 + e_1^2 + e_2^2 + e_3^2 - 1 = 0. \tag{15}$$

The Euler parameters are time stepped through

$$\dot{e}_0 = \frac{1}{2}\left(-e_1\hat{\omega}_x - e_2\hat{\omega}_y - e_3\hat{\omega}_z\right) \tag{16}$$

$$\dot{e}_1 = \frac{1}{2}\left(e_0\hat{\omega}_x - e_3\hat{\omega}_y + e_2\hat{\omega}_z\right) \tag{17}$$

$$\dot{e}_2 = \frac{1}{2}\left(e_3\hat{\omega}_x + e_0\hat{\omega}_y - e_1\hat{\omega}_z\right) \tag{18}$$

$$\dot{e}_3 = \frac{1}{2}\left(-e_2\hat{\omega}_x + e_1\hat{\omega}_y + e_0\hat{\omega}_z\right). \tag{19}$$

Because of the torque, the rotation angles and the angular velocity will vary over time. The computed torques in Eq. (5) are used to determine how the angular velocity changes over time, using the equations of motion in each body in their body-fixed frame, which is given as

$$I\dot{\hat{\boldsymbol{\omega}}} + (\hat{\boldsymbol{\omega}} \times I\hat{\boldsymbol{\omega}}) = \hat{\mathbf{M}}. \tag{20}$$

where I is the inertia tensor of the body, $\hat{\mathbf{M}}$ the torque in the body-fixed frame and $\hat{\boldsymbol{\omega}}$ the angular velocity in the body-fixed frame. For bodies where the non-diagonal components of the inertia tensor are zero (such as an ellipsoid), the equations become

$$I_{11}\dot{\hat{\omega}}_x + (I_{33} - I_{22})\hat{\omega}_y\hat{\omega}_z = \hat{M}_x \tag{21}$$

$$I_{22}\dot{\hat{\omega}}_y + (I_{11} - I_{33})\hat{\omega}_x\hat{\omega}_z = \hat{M}_y \tag{22}$$

$$I_{33}\dot{\hat{\omega}}_z + (I_{22} - I_{11})\hat{\omega}_x\hat{\omega}_y = \hat{M}_z. \tag{23}$$

5.3.1 Frame of reference

All variables, except for the angular velocities, are given in the inertial frame. The rotation matrix $\mathcal{R}$ of some body A therefore projects a vector in the reference frame of A back to the inertial frame.

For a two-body problem, it is common to express the positions $\mathbf{r}_s$ and velocities $\mathbf{v}_s$ of the secondary in the reference frame of the primary. This can be achieved by

$$\mathbf{r}_s^{[p]} = \mathcal{R}_p^T(\mathbf{r}_s - \mathbf{r}_p) \tag{24}$$

$$\mathbf{v}_s^{[p]} = \mathcal{R}_p^T(\mathbf{v}_s - \mathbf{v}_p) \tag{25}$$

where the subscript p and s correspond to the parameters of the primary and secondary. The $^{[p]}$ indicates the vector in the reference frame of the primary and $\mathcal{R}_p^T$ is the transpose of the rotation matrix of the primary.

5.4 Kepler to Cartesian

If orbital elements are used as initial condition of the simulation, they will be converted to Cartesian coordinates in order to run the simulation. The general procedure is to use the equations (Curtis, 2013)

$$\mathbf{r} = [Q]_{xX}\{\mathbf{r}\}_X \tag{26}$$

$$\mathbf{v} = [Q]_{xX}\{\mathbf{v}\}_X \tag{27}$$

where

$$\{\mathbf{r}\}_X = \frac{h^2}{\mu}\frac{1}{1+e\cos M}\begin{bmatrix}\cos M\\ \sin M\\ 0\end{bmatrix} \tag{28}$$

$$\{\mathbf{v}\}_X = \frac{\mu}{h}\begin{bmatrix}-\sin M\\ e+\cos M\\ 0\end{bmatrix} \tag{29}$$

and

$$[Q]_{xX} = \begin{bmatrix}\cos\Omega\cos\bar{\omega} - \sin\Omega\sin\bar{\omega}\cos i & -\cos\Omega\sin\bar{\omega} - \sin\Omega\cos\bar{\omega}\cos i & \sin\Omega\sin i\\ \sin\Omega\cos\hat{\omega} + \cos\Omega\sin\hat{\omega}\cos i & -\sin\Omega\sin\hat{\omega} + \cos\Omega\cos\hat{\omega}\cos i & -\cos\Omega\sin i\\ \sin\hat{\omega}\sin i & \cos\hat{\omega}\sin i & \cos i\end{bmatrix}. \tag{30}$$

Here, μ is the gravitational parameter, h the specific angular momentum, e the eccentricity, M the true anomaly, Ω the ascending node, $\hat{\omega}$ the longitude of the pericenter and i the inclination. The specific angular momentum is computed by

$$h = \sqrt{a\mu(1-e^2)} \tag{31}$$

where a is the semimajor axis, which relate to the orbital period D as

$$a^3 = \frac{\mu D^2}{(2\pi)^2}. \tag{32}$$

The true anomaly can be computed from the mean anomaly M_0 as

$$M = M_0 + \left(2e - \frac{e^3}{4}\right)\sin M_0 + \frac{5}{4}e^2\sin 2M_0 + \frac{13}{12}e^3\sin 3M_0 + O(e^4). \tag{33}$$

The mean anomaly relates to the mean longitude λ and the longitude of the pericenter $\hat{\omega}$ as

$$M_0 = \lambda - \hat{\omega}. \tag{34}$$

5.5 Mass scaling

The surface integration procedure may have difficulties integrating function values that are too large, as the integrator is unable to reach the desired accuracy. This is an issue as a large number is compared with a small one (the desired accuracy). As a consequence, all variables that contains the mass will be scaled with a mass scaling factor M_* before the equations of motion are solved. The value of M_*, regardless of units used (see `-units` argument in Sect. 3), is calculated as the power factor of the maximum mass among the bodies in the simulation. In short, it is given as

$$M_* = 10^{x_s} \tag{35}$$

$$x_s = \mathrm{Max}\left\{\mathrm{floor}\left[\log_{10}\left(|M|\right)\right]\right\} \tag{36}$$

where M is the array containing the masses of all bodies in the simulation, floor rounds down the values to nearest integer and Max the maximum value in that array. The variables that are scaled as follows:

- Masses: New units are M_*.
- Densities: New units are M_* m^{-3}.
- Gravitational constant: New unit is m^3 M_*^{-1} s^{-2}.

The units of meter (m) and seconds (s) will also change based on the unit selection before the simulation starts. As a result of new unit scales, the computed forces and torques will also be scaled as

$$F = M_* \text{ m s}^-2$$

$$M = M_* \text{ m}^2 \text{ s}^-2.$$

However, the computed positions, velocities, angular velocities and angles do not depend on the masses, as the masses cancels. In reality, the mass scaling only affects the output forces and torques, but not the final dynamics of the bodies.

5.5.1 Mass scaling example

Consider a two-body simulation, where standard SI units (meters and seconds) are used, and the masses of body A and B are

$$m_A = 1.36 \cdot 10^{11} \text{ kg} \tag{37}$$
$$m_B = 9.31 \cdot 10^{9} \text{ kg}. \tag{38}$$

The mass scale is then

$$M_* = 10^{11} \text{ kg} \tag{39}$$

and the masses rescaled as

$$m_A = 1.36 M_* \tag{40}$$
$$m_B = 9.31 \cdot 10^{-2} M_*. \tag{41}$$

The gravitational constant is then

$$G_g = 6.67408 \frac{\text{m}^3}{M_* \text{s}^2}. \tag{42}$$

5.6 Collision detection

The program first checks whether the sphere surrounding two bodies are intersecting. For an ellipsoid, the radius of the sphere is equal to the largest semiaxis of the ellipsoid. If the spheres intersect, a second algorithm is used to determine if the ellipsoids have truly intersected. The algorithm follows the work of Alfano and Greer (2003). Intersection occurs when one of the following is satisfied:

- One of the four eigenvalues is complex.
- At least one pair of eigenvectors is equal within 14 decimal digits.

For a polyhedron, the radius is determined by the distance from the vertex with the largest separation from the centroid. However, an equivalent algorithm of detecting polyhedron intersection is not implemented. The polyhedron is considered to have collided with a different body when the separation between the two bodies is smaller than the radius of the polyhedron.

6 Numerical methods

This section briefly describes the Runge-Kutta methods implemented in the software. The embedded Runge-Kutta methods are implemented through standard means, where the coefficients are given by their respective Butcher tableaus (see e.g. Hairer et al., 1993).

6.1 Standard 4th order Runge-Kutta

This is the classic 4th order Runge-Kutta method that is commonly found in the literature. It does not make use of adaptive time stepping.

6.2 Dormand-Prince Runge-Kutta 5(4) method

The order 5(4) method by Dormand and Prince (1980) is the default ODE solver a number of ODE solving libraries (e.g. `integrate.solve_ivp` from `Scipy` and `ode45` in `Matlab`).

In our implementation, the adaptive step size is determined by (Hairer et al., 1993) The new step size h_n is then

$$h_{n+1} = h_n \min(h_{max}, \max(h_{min}, fac \cdot (1/\epsilon)^{1/(q+1)})) \tag{43}$$

where h_{min} is the minimum step size increase is allowed to take, h_{max} the maximum step size increase, fac a safety factor and $q = \min(p, \hat{p})$. The variables h_{min} and h_{max} are given by the `hmin` and `hmax` in the input file (see Sect. 3.1). The variable ϵ is the local error

$$\epsilon = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(\frac{y_{n+1,i} - \hat{y}_{n+1,i}}{sc_i}\right)^2} \tag{44}$$

where

$$sc_i = atol + \max(|y_{ni}|, |y_{n+1,i}|)rtol \tag{45}$$

Here, $atol, rtol$ are the absolute and relative tolerances respectively, which is (or can be) given as user inputs. The default values of these for the DP54 method is $atol = rtol = 10^{-8}$.

6.3 Verner 6(5) (efficient) method

An implementation of the 9-stage order 6(5) Runge-Kutta method due to J.H Verner is also available (V65). A description of the 'robust' method is given in Verner (2010). We will, however, use the 'efficient' method. The Butcher Tableau coefficients are taken from Verner's webpage http://people.math.sfu.ca/~jverner/, which also includes table coefficients of higher orders.

The error estimation for this method is the same as the DP54 method described in Sect. 6.2. However, the tolerances *atol* and *rtol* should take very small values. For the V65 method, the default values are $atol = rtol = 10^{-9}$. The safety factor is given as

$$fac = (0.25)^{1/(q+1)}. \tag{46}$$

6.4 Verner 7(6) (efficient) method

The 9-stage Verner method of order 7(6) is also included. Like the 6(5) method, we use the efficient coefficients taken from http://people.math.sfu.ca/~jverner/[11]. The default tolerance values for this method is $atol = rtol = 10^{-10}$. The safety factor is given by Eq. (46).

6.5 Verner 8(7) (efficient) method

The 13-stage Verner method of order 8(7) is included in SIANS which also uses the efficient coefficients. The default tolerance values are $atol = rtol = 10^{-12}$ with the safety factor given in Eq. (46).

6.6 Verner 9(8) (efficient) method

Finally, we have also implemented the 16-stage Verner method of order 9(8). The efficient coefficients are again used. The default tolerance values are $atol = rtol = 10^{-13}$ with the safety factor, again, given in Eq. (46).

[11] Last accessed August 9. 2021.

6.7 Tsitouras 9(8) method

The Tsitouras method of oder 9(8) (Tsitouras, 2001) is also implemented. The local error for this method is computed as

$$\epsilon = \|y_{n+1} - \hat{y}_{n+1}\| \tag{47}$$

where $\|,\|$ is the norm. The step size control algorithm is then given as

$$h_{n+1} = 0.9h_n \left(\frac{\text{TOL}}{\epsilon}\right)^{1/p} \tag{48}$$

where $\text{TOL} = a = rtol$ is the user specified tolerance, with a default value of $TOL = 10^{-13}$.

6.8 Feagin 10(8) method

The Feagin method of order 10(8) (Feagin, 2007) has also been implemented. The coefficients are taken from https://sce.uhcl.edu/rungekutta/[12]. The local error is computed as

$$\epsilon = h\epsilon_{fac}\epsilon_l \tag{49}$$

where

$$\epsilon_{fac} = \frac{1}{360} \tag{50}$$

$$\epsilon_l = \|k_2 - k_{16}\|. \tag{51}$$

The step size is then

$$h_{n+1} = h_n \min(h_{max}, \max(h_{min}, fac \cdot (TOL/\epsilon)^{1/(q+1)})). \tag{52}$$

For this method, TOL is defaulted at 10^{-12}.

6.9 Feagin 12(10) method

The Feagin method of order 12(10) (Feagin, 2012) is a 25 stage method. The local error is computed in the same way as the 10(8) method. The main

[12]Last accessed August 9. 2021.

difference is the value of ϵ_{fac} and ϵ_l, which are given as

$$\epsilon_{fac} = \frac{49}{640} \tag{53}$$

$$\epsilon_l = \|k_2 - k_{24}\|. \tag{54}$$

The step size is identical to that of Eq. (52), and default $TOL = 10^{-12}$.

7 List of files

The following directories contain the following files:

7.1 Main directory

- `main.py` - The main program that is called to run the whole software.
- `readinput.py` - Reads the input files with initial conditions and integration parameters.
- `readsolution.py` - Saves and reads the results of the simulations.
- `other_functions.py` - Contains some helper functions for the main software.
- `exampleruns.py` - Contains some examples on how the software can be executed through a `Python` script.
- `helperfuncs.py` - Contains some functions that are used when the software is executed through a `Python` script.

7.2 cymodules (with or without mpir)

The `C` (`.c`) files have corresponding `.h` files and the `Cython` (`.pyx`) files have corresponding `.pxd` files.

- `commonCfuncs.c` - Contains some helper functions used among the `C` files.
- `diffeqsolve.c` - Functions used to solve the equations of motion.
- `ODESolvers.pyx` - The `Cython` wrapper of the Runge-Kutta solver. Calls the `C` functions to solve the equations of motion and returns to `main.py` with a class instance.
- `odetableaus.pyx` - The Butcher tableaus for the Runge-Kutta solvers.
- `other_functions_cy.pxd` - Contains some helper functions used for the software, including the calculation of the energies.
- `potentials.c` - The gravitational potentials used for the surface integration method are found here.

- `SurfaceIntegrals.c` - The surface integration method for the force, torque and mutual potential energy are found here.

7.3 initial_values

This directory contains all the initial parameters used for the simulations. These files must be in `.txt` format.

- `example.txt` - An example input file.

8 Known issues

Here is a list of known issues with SIANS that may arise in the simulation, with some possible workarounds.

8.1 Roundoff error from GSL

The numerical integrator (quadrature) from GSL may raise an error indicating:

```
1 ERROR: roundoff error prevents tolerance from being achieved
2 Default GSL error handler invoked.
3 Aborted (core dumped)
```

As indicated by the error message, this is because the computed error is unable to reach the desired tolerance values. This can be solved by either using a lower `quadval` value in the input file or the tolerance values `iabstol` and `ireltol` must be larger (i.e. less accurate integration schemes).

This error may also indicate that the system is unstable, e.g. the bodies overlap.

The problem may also arise if the parameter values are inconsistent with the unit usage. For instance, `-units` (option) uses kilometers and hours, but the initial velocity is given in meters per second.

8.2 Maximum number of subdivisions reached from GSL

The numerical integrator (quadrature) from GSL may raise an error indicating:

```
1 gsl: ..\..\integration\qag.c:257:
2 ERROR: maximum number of subdivisions reached
3 Default GSL error handler invoked.
```

The most common cause is due to the missing argument `-units` (option) that is required to determine the physical dimensions used. By default, SIANS uses dimensionless quantities. If the input data considers data that are not dimensionless, then the dimensions must be specified.

However, this error message may also indicate that the system is not stable or non-physical behaviour occurs. This may issue may stem from the initial conditions that is used for the simulation. For instance, the bodies may be intersecting.

The current implementation of the surface integration limits to 2000 subinterval ranges. This can be increased by changing the number 2000 to any other number in the following lines

```
gsl_integration_workspace * w =
    gsl_integration_workspace_alloc(2000);
gsl_integration_qag(&F, 0, 1, itol[0], itol[1], 2000, itol
    [2], w, &result, &error);
gsl_integration_workspace_free(w);
```

or for the ellipsoid approach

```
gsl_integration_workspace * w =
    gsl_integration_workspace_alloc(2000);
gsl_integration_qag(&F, -c_A, c_A, itol[0], itol[1], 2000,
    itol[2], w, &result, &error);
gsl_integration_workspace_free(w);
```

These lines are found in the `SurfaceIntegrals.c` modules.

8.3 Step size and Euler Parameters

The integration scheme may also diverge if the step sizes of the Runge-Kutta solver is too large. This causes issues with time stepping of the Euler parameters, in which the constraint equation of Eq. (15) is no longer satisfied and the sum of $e_0^2 + e_1^2 + e_2^2 + e_3^2$ becomes far larger than one. This generally occurs when adaptive time stepping is disabled, and can be remedied by reducing the time step or using an adaptive time stepper.

8.4 Inaccurate force calculations

The surface integration scheme suffer from round-off errors during the surface integral process, and can result in some force components being inaccurate. For instance, if two bodies of equal shapes are separated only along the x-direction, there should only be a force along the x-direction. However, the results from the surface integrals may result in some small force components

along the y and/or z-directions. It is therefore advised for the user to be mindful of these small errors.

References

S. Alfano and M. L. Greer. Determining if two solid ellipsoids intersect. *Journal of guidance, control, and dynamics*, 26(1):106–110, 2003.

J. T. Conway. Vector potentials for the gravitational interaction of extended bodies and laminas with analytical solutions for two disks. *Celestial Mechanics and Dynamical Astronomy*, 125:161–194, June 2016. doi: 10.1007/s10569-016-9679-y.

J. T. Conway. Analytical solution from vector potentials for the gravitational field of a general polyhedron. *Celestial Mechanics and Dynamical Astronomy*, 121(1):17–38, Jan 2015. doi: 10.1007/s10569-014-9588-x.

H. D. Curtis. *Orbital mechanics for engineering students*. Butterworth-Heinemann, 2013.

J. R. Dormand and P. J. Prince. A family of embedded runge-kutta formulae. *Journal of computational and applied mathematics*, 6(1):19–26, 1980.

T. Feagin. A tenth-order runge-kutta method with error estimate. In *Proceedings of the IAENG Conference on Scientific Computing*, 2007.

T. Feagin. High-order explicit runge-kutta methods using m-symmetry. *Neural, Parallel & Scientific Computations*, 20, 09 2012.

M. Galassi et al. *GNU scientific library*. Network Theory Limited, 2002.

E. Hairer, S. P. Nørsett, and G. Wanner. *Solving Ordinary Differential Equations I (2Nd Revised. Ed.): Nonstiff Problems*. Springer-Verlag, Berlin, Heidelberg, 1993. ISBN 0-387-56670-8.

A. Ho, M. Wold, J. T. Conway, and M. Poursina. Extended two-body problem for rotating rigid bodies. *Celestial Mechanics and Dynamical Astronomy*, 133(8):35, August 2021. doi: 10.1007/s10569-021-10034-8.

E. Jones, T. Oliphant, P. Peterson, et al. SciPy: Open source scientific tools for Python, 2001–. URL http://www.scipy.org/. [Online; accessed ¡today¿].

W. MacMillan. *The Theory of the Potential*. (MacMillan: Theoretical Mechanics). McGraw-Hill Book Company, Incorporated, 1930.

C. Tsitouras. Optimized explicit runge–kutta pair of orders 9 (8). *Applied numerical mathematics*, 38(1-2):123–134, 2001.

J. H. Verner. Numerically optimal runge–kutta pairs with interpolants. *Numerical Algorithms*, 53(2):383–396, Mar 2010. ISSN 1572-9265. doi: 10.1007/s11075-009-9290-3. URL https://doi.org/10.1007/s11075-009-9290-3.