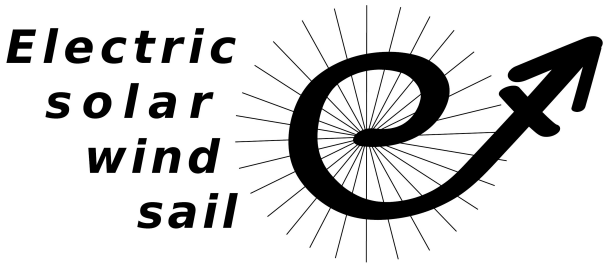

ESAIL D51.1
Description of E-sail dynamic simulator codes

Work Package: **WP 51**

Version: **Version 1.0**

Prepared by:	Finnish Meteorological Institute, Pekka Janhunen
Time:	Helsinki, May 24th, 2013
Coordinating person:	Pekka Janhunen, pekka.janhunen@fmi.fi

(List of participants:)

Participant no.	Participant organisation	Abbrev.	Country
1 (Coordinator)	Finnish Meteorological Institute	FMI	Finland

Table of Contents

1. Applicable documents

AD-1: Press, W.H., Teukolsky, S.A., Vetterling, W.T. And Flannery, B.P., *Numerical Recipes, The art of scientific computing*, 3rd edition, Cambridge, 2007

AD-2: http://www.lua.org

AD-3: http://omniweb.gsfc.nasa.gov

2. Introduction

This document presents two simulation models developed for simulating the dynamical behaviour of the E-sail tether rig. The first described model VESVISION-v2 can simulate the spinning E-sail rig with full physics (tether elasticity, thermal expansion, electric resistivity etc.) and also includes the simulate tether outreeling (change of tether length) and uses a second order numerical solver for the 1-D time-dependent partial differential equations which governs the tethers. The second described model VESVISION-v3 provides a highly accurate 8^{th} order numerical ordinary differential equation solver and a general framework for simulating and arbitrary collection of rigid bodies, point masses and their interaction forces, as well as arbitrary external forces. Both models support OpenMP parallel execution and fast execution was an important design goal. Both models also employ OpenGL based realtime visualisation and user interaction. The entire VESVISION-v2 and the core of VESVISION-v3 is written in C++. In VESVISION-v3 the model is defined flexibly by a Lua [AD-2] script which the user can easily modify or write new ones. In VESVISION-v2, the user can write his E-sail control algorithm in plain C. In that way, VESVISION-v2 can serve as a “flight simulator” for testing various E-sail control algorithms in realistic solar wind conditions.

3. VESVISION-v2

The VESVISION-v2 code (VES=Virtual Electric Sail) models the tether as a 1-D continuous string which has zero stiffness but finite elasticity and thermal expansion. The formulation also includes the relevant mass flow terms at the spacecraft end to allow simulation of tether reeling (changing the tether's length at some speed which is an arbitrary prescribed function of time). The tips of the tethers can contain Remote Units of given mass (modelled as point masses) and the tips can also be connected together by auxiliary tethers (with given mass per length and given elastic properties). There is also a possibility to add “extra” radial tethers with free ends pointing outward from the Remote Units. The extra tethers can also contain their own end masses. The main spacecraft is simulated by a point mass from the point of view of tether dynamics. To study potential precession and tumbling of the main spacecraft, it can also be modelled as a rigid body which responds to the torques coming from the tethers. The approximation made in this case is that the main spacecraft is very small compared to the length of the tethers so that the main spacecraft's angular momentum is negligible in comparison to the angular momentum of the tether rig (the self-consistent main simulation treats the main spacecraft as a point mass, but a solution for a rigid body version of the main spacecraft is computed afterwards without backreaction).

VESVISION-v2 contains models for thermal expansion of the tethers (the effect can be important if an E-sail moves through a planetary shadow so that the tethers undergo rapid temperature variation) and a full electric simulation of the current flowing in the tether and its local voltage, including self-consistently the ohmic voltage drop along the tether (the effect is usually small unless the tether's length approaches 100 km). The electric model contains a potentiometer between each tether and the main spacecraft (for individual control of the voltage of each tether), as well as the electron gun whose current and voltage can be set freely.

As forces acting on the tethers, besides the E-sail force also the gravity gradient force can

be included. The gravity gradient force becomes relevant near a massive body; for example if one decides to deploy the E-sail already in Earth orbit before entering the solar wind.

VESVISION-v2 includes the historical satellite-measured solar wind data at 1 minute resolution from NASA's OMNIWeb project. The OMNIWeb 1 min data has been combined from different satellites and covers the time period from 1995 to 2008, i.e. more than one 11-year solar cycle. The OMNIWeb data contains some gaps. The gaps are filled by VESVISION-v2 by an algorithm which uses adjacent data such that the result is smooth and has similar statistical properties as the adjoining real data. The gap filling algorithm enables one to run the simulator for arbitrarily long time (up to 13 years) with realistic solar wind data input. The relevant variables used from OMNIWeb data are the plasma density and the plasma flow velocity vector.

VESVISION-v2 is written in C++, uses OpenGL based 3-D interactive visualisation and also supports parallel execution with OpenMP. VESVISION-v2 implements also an internal application programming interface (API) callable from plain C, intended for the user to write an E-sail control algorithm in C and testing it in the realistic virtual physics environment provided by VESVISION-v2. The API contains simple C-functions for commanding elements such as the tether reel motors, the potentiometers and the electron gun current and voltage. It also contains functions for reading various virtual sensors such as the Remote Unit position sensor (which would be typically based on optical detection from the main spacecraft) and a solar wind density sensor (typically based on a simple omnidirectional electron spectrometer). A different programming language (plain C rather than C++) was selected for the user portion to isolate it very well from the rest of the simulator: the E-sail control routine written by the user can only interact with the underlying simulator by using the restricted set of API C functions. While similar encapsulation could have been achieved by simply using the normal C++ class mechanisms, plain C was selected because it is typically used for programming flight software.

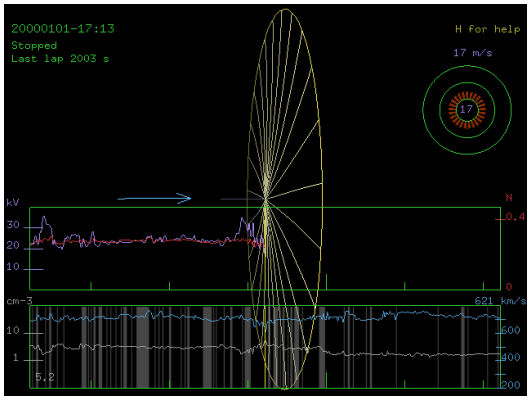

Figure 1: Screen dump from VESVISION-v2

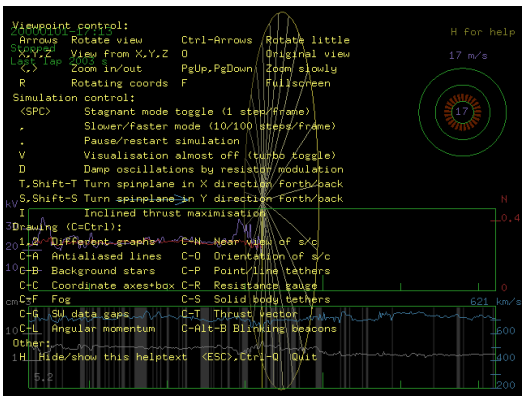

Figure 2: VESVISION-v2 with help texts on

Figures 1 and 2 show examples of VESVISION-v2 screen. The upper left corner shows the time of the solar wind conditions and the time of the most recent tether rig rotation period. The upper right corner shows the gathered delta-v, the instantaneous electron gun voltage and potential drops over each tether potentiometer as a graphical radial bar chart. The bottom panel shows the solar wind data, velocity in blue in linear scale and plasma density

in beige in logarithmic scale. Original data gaps are shown as greyed areas. The panel rolls from right to left as the simulation proceeds, the “now” instant being the vertical yellow line in the middle of the screen. The upper panel shows the thrust history as red curve and the applied electron gun voltage as violet curve. A zoomable and rotatable 3-D plot of the E-sail tether rig is shown in the middle, with a blue arrow showing the instantaneous solar wind direction. Figure 2 is the same as Figure 1 except that the online help texts are showed because the user pressed “H”. The help texts show the key bindings by which the user can interact with the software. When help texts are shown, the rest of the graphics on the screen are slightly dimmed.

Table 1 lists the supported command line options. The same options can also be set through configuration file. Some graphical output related options which can also be set interactively by the user by keypresses are omitted from the list for brevity. Table 2 shows the internal C calls available for a user-written control algorithm. Not all of the available C API calls are used by typical control algorithm and some of them are only used for debugging.

SWangle	0.0	Additional solar wing angle offset (deg)
record_frames	false	Record all drawn frames in pixmap files
max_resistance	1e9	Max. resistance setting of potentiometer (Ohm)
gun_maxcurrent	0.05	Max. current limit of e-gun (A)
gun_maxvoltage	4e4	Max. voltage limit of e-gun (V)
gun_perveance	6e-7	Max. perveance limit of e-gun (A/V^(3/2))
gun_maxpower	400	Max. power limit of e-gun (W)
circular_initial_auxtethers	true	Initialise auxtethers as circular instead of linear
fog	true	Use for effect in rendering
config	“Vesvision.conf”	Name of configuration file
eventfile	“SDF.ves”	Name of keypress events def. script file
tmax	-1	Max. time of simulation (s; <0:infinity)
spinperiod	2500	Spin period (s)
L0init	20e3	Initial tether length (m)
t_forced_spinup	480	Initial period for smooth spin start (s)
rw	1.8e-5	Wire radius (m)
rwauxrel	1.0	Aux. vs. main tether wire radius
rwextrarel	1.0	Extra vs. main tether wire radius
rwstar	1e-3	Effective tether electric radius (m)
extratether_rel_length	0.5	Relative length of extratethers vs. main tethers
rhow	2.7e3	Tether material density (kg/m^3)
rhowauxrel	1.0	Aux. vs. main tether material density
rhowextrarel	1.0	Extra vs. main tether material density
Young	7.2e10	Young modulus of tether material (Pa)
Youngauxrel	1.0	Young modulus of aux. vs. main tether material
Youngextrarel	1.0	Young modulus of extra vs. main tether mat.
endmass	0.3	Remote unit mass at tip of each tether (kg)
endmass1_factor	1.0	Factor by which endmass of node 1 is different
scmass	1e3	Spacecraft body mass (kg)
auxmidmass	0.0	Point mass at middle of each auxtether (kg)
extraendmass	0.15	Extra point mass at tip of each extratether (kg)
endmassmethod_explicit	true	Whether endmass is explicit mass point or tip enhancement of linear density lambda
CFL	0.7	Courant-Friedrichs-Lewy timestep parameter
dt_flightalgo	5.0	Time between calling synchr. flight algorithm (s)
material_damping	5e-4	Dimensionless damping coefficient for numerical stability
hoytether_angle	30	Hoytether diagonal/parallel angle (deg)

Np	30	Number of discretisation points in one tether
Npaux	-1	Number of discr. Points in ont auxtether
auxtethers	false	Whether we have auxtethers or not
exclude_auxtether1	false	Exclude auxtether number 1 (nums start from 0)
exclude_extratether1	false	Exclude extratether number 1
extratethers	false	Whether we have extratethers or not
coulomb_repulsion	false	Take into account Coulomb repulsion of tethers (approximately)
gravitygradient	false	Assume LEO gravity gradient
gravitygradient_alt	5e3	Orbital alt. (km) where grav. grad is calculated
auxtether_lengthcoeff	1.0	Factor by which auxtethers are longer than nominal circle
Nw	70	Number of tethers
realsw	true	Whether to use real solar wind data or not
yyyymmdd	20000101	Solar wind starting date YYYYMMDD
hhmm	0000	Solar wind starting hour and minute
sc_radius	1.0	Spacecraft radius (m)
nSW	7.3e6	Solar wind number density (1/m^3), if realsw=false
vSW	400e3	Solar wind speed (m/s), if realsw=false
gun_rel_energywidth	0.02	Electron gun beam delta-E/E
gun_min_energywidth	50.0	Electron gun delta-E for small energy E
reel_minspeed	-0.1	Min. allowed outlet speed (m/s) of reel (pos.outward)
reel_maxspeed	0.1	Max. allowed outlet speed of reel
reel_maxacc	0.1	Max. allowed reel outlet acceleration (m/s^2)
r_AU	1.0	Solar distance in au
thermal_alpha	0.1	Wire optical absorptivity (one minus albedo)
thermal_epsilon	0.03	Wire infraed emissivity
thermal_expansion	2.31e-5	Wire thermal expansion coefficient

Table 1: Command line/config. file options for VESVISION-v2

get_time()	Return spacecraft time in seconds from start of simulation
get_potdrop(w)	Return potential drop (volts) over control resistor of w'th tether
get_scpot()	Return estimate of spacecraft (or electron gun anode) potential (volts) from electron detector
get_tether_tip_direction(w,&ux,&uy,&uz)	Return unit vector to the tip of w'th tether, in Sun-spaceraft coordinates, from the camera system
get_tether_root_direction(w,&ux,&uy,&uz)	Return unit vector along root of w'th tether, in Sun-spacecraft coordinates, from the camera system
get_tension(w)	Return tension of w'th tether (newtons)
get_plasma_density()	Return solar wind density estimate from electron detector (1/m^3)
get_solar_wind(&n,&vx,&vy,&vz)	Return solar wind parameters from ion detector in Sun-spacecraft coordinates and in SI units (1/m^3, m/s)
set_gun_CV(&I,&V)	Set electron gun current (A) and voltage (V). Return 0 on success, 1 if one or both values were too large or too small. The values are set to a closest approximation in that case. The actual values set are returned in I and V.
set_resistance(w,&R)	Set control resistor of w'th tether to R ohms. Return 0 on success, 1 if the value set was too large, 2 if it was too small. The values are set to a closest approximation in those cases. The actual value set is returned in R.
set_reel_speed(w,v,t)	Set w'th tether reel into mode where it constantly reels out tether at speed 'v' (m/s). Negative speed means reeling in. The change from the present reel motion state takes 't' seconds. If 't' is too short or not positive, the motion change is carried out as quickly as the hardware allows. Before, reel mode can be any. During command, mode is 3. After, mode is 1 if speed is 0.0, otherwise 2. Reel modes: (1) Steady, waiting for command, (2) Moving, waiting for command, (3) Executing speed change command, (4) Executing length change command
set_auxreel_speed(w,v,t)	Same as set_reel_speed, but for auxiliary tether reels of the Remote Units. The speed 'v' is multiplied by a correction factor which is the ratio of the initial length of the auxiliary tether versus the main tether. Thus you can pass the same value for speed as you do for set_reel_speed() to obtain isometric expansion of contraction of the tether system.
set_extrareel_speed(w,v,t)	Same as set_reel_speed, but for extrareels of the Remote Units. Same comments apply for the correction factor as in set_auxreel_speed().
change_tether_length(w,dL,t)	Reel out 'dL' metres of tether from w'th reel. Negative 'dL' means reeling in. Before, reel mode is usually 1. If it is not, a set_reel_speed(w,0,0) command is implicitly executed first. During command, mode is 4. After, mode is 1.
change_extratether_length(w,dL,t)	Same as change_tether_length, but for extra tether.
get_reel_speed(w)	Return outletting speed (m/s) of w'th tether reel (positive outward, negative inward)

get_tether_remaining_length(w)	Return estimated length of tether remaining on reel in metres
get_extratether_remaining_length(w)	Return estimated length of extratether remaining on reel in metres
get_tether_outlet_length(w)	Return estimated length of deployed tether in metres
get_extratether_outlet_length(w)	Return estimated length of deployed extratether in metres
get_reel_command_timeremain(w)	If w'th tether reel has an unfinished length or speed change command in execution (i.e., is in mode 3 or 4), return its estimated completion time in seconds, otherwise return 0.0
set_additional_force(dFds)	For algorithm testing only: Set additional (constant) z-directed force per unit length (N/m) for all tethers. The setting is global and remains set until changed by this function. Notice that if and when the tethers are bent, this additional dFds_z has a component along the tether also, unlike the solar wind force which is constructed to be locally perpendicular to the tether. The typical use of this function is to use it with zero electron gun power to simulate an ideally controllable sail with exactly adjustable and constant direction thrust vector. Of course, in the final flight algorithm, this function shouldn't be called.
write_message(msg)	Set msg string visible on screen
damping_mode_requested()	Returns 1 if user has requested "potential damping mode" to be applied by flight algorithm
turning_mode_requested()	Returns spiniplane turning mode that user has requested, if any: -2 for reverse Y-directed spinplane turning mode, -1 for reverse X-directed spinplane turning mode, 0 for no turning mode, +1 for X-directed spinplane turning mode, +2 for Y-directed spinplane turning mode
inclined_thrust_maximisation_mode_requested()	Returns 1 if user has requested "inclined thrust maximisation mode" to be applied by flight algorithm
tether0_cut()	Returns 1 if tether 0 has been cut by user definition at this time
set_thrust_scalar(w, F)	Set Remote Unit thruster thrust scalar for w'th tether to value F newtons
set_thrust_vector(w,Fx,Fy,Fz)	Set Remote Unit thruster thrust vector for w'th tether to value F newtons

Table 2: Internal control algorithm C API of VESVISION-v2

4. VESVISION-v3

VESVISION-v3 addresses the following two shortcomings of VESVISION-v2: (1) the order of accuracy of the underlying differential equation solver and (2) applicability to possibly interesting non-traditional E-sail configurations which do not necessarily consist of a single main spacecraft with a number of radial tethers and thus cannot be simulated with VESVISION-v2. When addressing these additional needs, not all features of

VESVISION-v2 were found feasible to retain, however. Hence VESVISION-v3 cannot replace VESVISION-v2 in all tasks. Hence the domains of applicability of the two versions are distinct, but with a wide overlap.

VESVISION-v3 models an arbitrary collection of rigid bodies and point masses which interact with arbitrary force fields and are influenced by arbitrary external forces. For each point mass the modelled degrees of freedom are position and velocity. For rigid bodies, also the attitude of the body described by a unit quaternion and the angular momentum are included. The collection of bodies yields a large system of ordinary differential equations (ODEs) which is solved by a highly accurate 8th order Runge-Kutta method described in AD-1. According to our experience with this integrator and its built-in error estimation we consider that its truncation error is small enough to be considered insignificant for the simulation task at hand. Thus the main approximation in VESVISION-v3 is the replacement of the continuous tether system by a finite set of discrete bodies (rigid bodies and/or point masses), not the routine which integrates said discrete body equations.

The general-purpose modelling core of VESVISION-v3 is written with C++ and similarly to VESVISION-v2 it implements interactive realtime OpenGL visualisation as well as supports OpenMP parallelisation. In VESVISION-v3, however, the user must define his mechanical model not with command line options, but flexibly with Lua scripting language [AD-2]. In this way, the model definition is well isolated from the simulator core and the user has complete freedom in how to set up his discretised model of a mechanical system. Technically, it would be possible to use VESVISION-v3 for simulating mechanical models which are quite unrelated to the E-sail.

Table 3 lists the Lua commands which are available to the user for defining the mechanical model and controlling it. Table 4 describes functions that the user may define in the Lua script to implement online control of the model. Tables 5 and 6 show listings of a minimal two-body model and a somewhat more complicated LEO tether model which creates two point masses (spacecraft and tether end mass) connected by a massless tether and computes magnetic Lorentz force and gravity gradient forces acting on such single-tether system in LEO. Our production-scale Lua scripts which implement E-sail models with auxtethers have typically 500-800 lines. Figures and show screen dumps of VESVISION-v3 run with a Lua model that has created a 12-tether auxtethered E-sail, tethers being modelled by chains of point masses connected by massless springs with given rest length. The violet box goes from (-1 km,-1 km) to (+1 km,+1km); it is drawn just to visualise the scale. The tether length is 2.4 km.

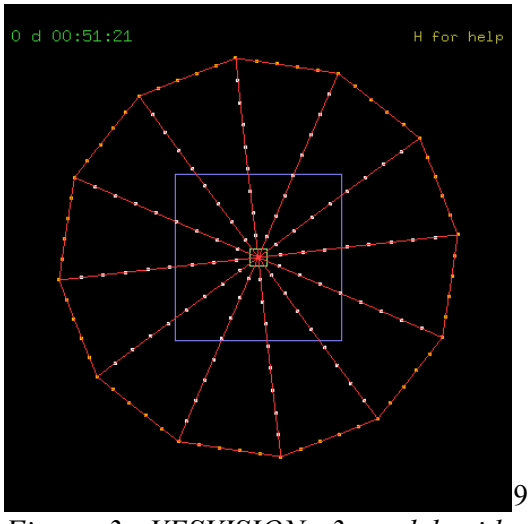

Figure 3: VESVISION-v3 model with 12 auxiliary tethers.

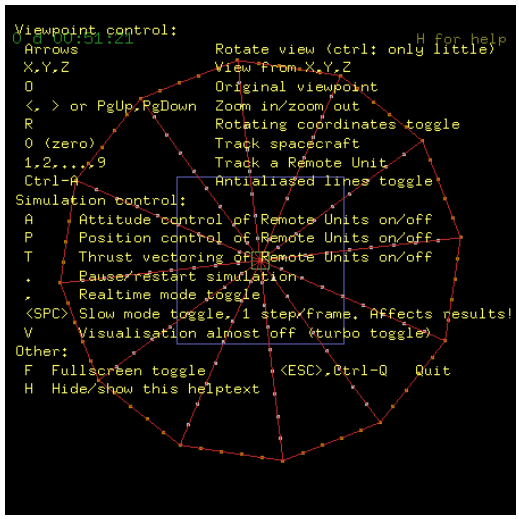

Figure 4: Same as Fig.3 But with online help texts

vesSetWindowTitle(“title”)	Set title of GLUT window to given string, default is "Virtual Electric Sail".
vesSetParams({param1=val 1,...})	Set numeric global parametres, supported ones are: eps_rel: Relative epsilon for ODE integrator, default 1e-7 eps_abs:Absolute epsilon for ODE integrator, default 1e-10 dt:Timestep, default 0.5 s tmax: Time when stop simulation, default -1 (never stop) rcamera: Distance of camera from origin, default 100 m gxx: Gravity field gx derivative with respect to x gyy: Gravity field gy derivative with respect to y gzz: Gravity field gz derivative with respect to z solarwind_startepoch planetGM: G*M of planet centred at origin (0,0,0)
obj=vesCreateBody(type, {prop1=val1,...})	Create new object, supported types are "box", "cylinder" and "pointmass". Returns a handle that can be passed to vesDefineInteractionForce. Supported object properties: mass(scalar): Mass of object (kg) radius(scalar): Radius of cylinder object (m) height(scalar): Height of cylinder object (m) boxsize(3-vector): Depth,width and height of box object (m) rCM(3-vector): Initial object's geometric centroid position (m)(equal to centre of mass CM except for prisms, hence name) v(3-vector): Initial velocity of object's CM (m/s) omega(3-vector): Initial angular velocity of object (1/s) rotateangle(scalar): Object's attitude is initially rotated (rad) rotateaxis(3-vector): Direction around which rotateangle rotation is done rotateangle2(scalar): Possible second rotation parameter (rad) rotataeaxis2(3-vector): Possible second rotation axis colour(3-vector): Object's RGB colour used for visualisation (0..1)
data=vesGetBodyData(obj)	Return table of data of given body (the argument must be previously returned by vesCreateBody). Currently for point masses, 'data' contains 7 named fields: type: "pointmass", "cylinder" or "box" mass: mass of particle (kg) x: x-coordinate position of body's centre of mass (m) y: y-coordinate position of body's centre of mass (m) z: z-coordinate position of body's centre of mass (m) vx: velocity x component of body's centre of mass (m/s) vy: velocity y component of body's centre of mass (m/s) vz: velocity z component of body's centre of mass (m/s) In case of a rigid body, in addition seven additional fields are stored in the table. The quaternion defining the attitude: qs: s-component of the attitude quaternion (cos(alpha/2)) qx:x-component of the attitude quaternion (sin(alpha/2)*nx) qy:y-component of the attitude quaternion (sin(alpha/2)*ny) qz:z-component of the attitude quaternion (sin(alpha/2)*nz) as well as the angular momentum: Lx: x-component of angular momentum (Nms) Ly: y-component of angular momentum (Nms)

	Lz: z-component of angular momentum (Nms)
vesRedefineBodyData(obj,data)	Set new data values for given object. The table 'data' can contain named entries x,y,z,vx,vy,vz. For rigid bodies, the additional entries qs,qx,qy,qz,Lx,Ly,Lz can be given as well. Missing entries are not set (they retain their old values). Additional named entries are silently ignored.
f=vesDefineInteractionForce(objA,objB,rA,rB,{prop1=val1,...})	Define inter-object force between objects A and B which must be handles previously returned by vesCreateBody. The forces affect points rA (given in A's object coordinates) and rB (given in B's object coordinates). The force is a central force along the line connecting rA and rB, defined by a spring constant (F = -k*r) and possible hysteretic and/or viscous damping coefficient. A force handle is returned that can be passed to vesRedefineInteractionForce later. Supported properties: springconst: Spring constant k, default 1e-3 N/m r0: Length offset, F = -k*max(r-r0,0), default 0.0 m r0dot: Time derivative of r0 (true r0(t)=r0+r0dot*t), default 0.0 m/s rel_lossmodulus: Loss modulus relative to spring constant, default 0.0 dampconst: Viscous damping constant D, F = -D*v, default 0.0 Ns/m longest_rateindependent_period: Maximum oscillation period for which hystereic occurs, default 600 s
vesRedefineInteractionForce(f,{prop1=val1,...})	Redefine inter-object force, f must be previously returned by vesDefineInteractionForce. Supported properties are the same as for vesDefineInteractionForce.
val=vesGetForceValue(f)	Get instantaous value of the interaction force f where f is a handle previously returned by vesDefineInteractionForce. Attractive force is returned as positive and repulsive as negative.
vesSetExternalForce(obj,F)	Sets external force (3-vector) for the given object
vesAddExternalForce(obj,F)	Adds to external force (3-vector) for the given object (i.e., same as vesSetExternalForce, but adds to the old value instead of overwriting it).
dv=vesGetDeltav()	Returns accumulated delta-v of the centre of mass (3-vector, m/s)
vesSetInfoString(s)	Sets "info" string written at the middle top of screen
rho,v=vesGetSolarWind(t)	Returns solar wind density (scalar, 1/m3) and velocity (3-vector, m/s) corresponding to given time (seconds from start of simulation)
vesDumpState(fn)	Dump everything to the given restartable CDF state file
vesRestoreState(fn)	Restore state from previously dumped CDF file

Table 3: Lua callable functions of VESVISION-v3

vesPeriodicTask(t)	If this function is defined by the user's Lua script, it is called at dt intervals before the ODE integrator. The argument t is the global time in seconds. Typically vesPeriodicTask could for example make calls to vesRedefineInteractionForce to modify the behaviour of the simulation.
key_was_pressed= vesKeypressHandler(key,t)	If this function is defined by the user's Lua script, it is called whenever a key is pressed on the graphics window. The first argument 'key' is the pressed key as a one-length string and the second argument t is the global time in seconds. The function must return a single Boolean value which is true if the routine recognised the key and false otherwise. The default control keys of vesvision are checked only if the return value was false, so the vesKeypressHandler takes precedence over vesvision's default key bindings.

Table 4: Special optional user-defineable functions in VESVISION-v3

```
mass1 = vesCreateBody("box",{mass=0.001,boxsize={0.1,0.1,0.1}})
mass2 = vesCreateBody("pointmass",{mass=1e-3,rCM={0.1,0,0}})
vesDefineInteractionForce(mass1,mass2,{0,0,0},{0,0,0},{springconst=1e-
4,dampconst=0.03e-4})
```

Table 5: A minimal Lua script to implement a two-mass system in VESVISION-v3

```
tether = {}
-----------------------------------
R_E = 6371.2e3
GM_E = 5.9723e24*6.6742e-11
alt = 700e3
dt = 10.0
m1 = 3.0
m2 = 1.0
spinperiod = 900*8
tether.Young = 0.1 * 70e9
tether.rel_lossmodulus = 0.02
tether.rwbase = 37.5e-6
tether.len = 1e3
diptilt = 11*(math.pi/180)
-----------------------------------
function CheckNum(x)
  if type(x) ~= "number" then
        print(debug.traceback("*** CheckNum: table value is not numeric",2))
        error("")
  end
  return x
end
-----------------------------------
vesSetParams({eps_rel=1e-5,eps_abs=1e-
6,dt=dt,rcamera=CheckNum(15*R_E),planetGM=CheckNum(GM_E)})
vorbit = math.sqrt(GM_E/(R_E+alt)) -- initial orbital speed of CM
L1 = tether.len*m2/(m1+m2)
L2 = tether.len*m1/(m1+m2)
omega = 2*math.pi/spinperiod
tension = m1*L1*omega^2
print(string.format("Tether tension = %g cN",100*tension))
v1 = L1*omega
v2 = L2*omega
mass1=vesCreateBody("pointmass",{mass=CheckNum(m1),rCM={0,0,-(R_E+alt)-
L1},v={vorbit-v1,0,0}})
mass2=vesCreateBody("pointmass",{mass=CheckNum(m2),rCM={0,0,-(R_E+alt)
+L2},v={vorbit+v2,0,0}})
tether.springconst = tether.Young*(math.pi*tether.rwbase^2)/tether.len
forceparams =
  {springconst=CheckNum(tether.springconst),
       rel_lossmodulus=CheckNum(tether.rel_lossmodulus),
       r0=CheckNum(tether.len)}
tether.force = vesDefineInteractionForce(mass1,mass2,{0,0,0},{0,0,0},forceparams)
fp = io.open("leotether.dat","w")
fp:write("# t x y z tension\n")
-----------------------------------
function Bmodel(r,t)
  local dipmom = -8e22
  local omegaE = 2*math.pi/(24*3600.0)
  local Mz = dipmom*math.cos(diptilt)
  local Mxy = dipmom*math.sin(diptilt)
  local M = Vector.new({Mxy*math.cos(omegaE*t),Mxy*math.sin(omegaE*t),Mz})
  local rmagn2 = r[1]^2 + r[2]^2 + r[3]^2
  local rmagn = math.sqrt(rmagn2)
```

```
  local ru = Normalise(r)
  return (1e-7/(rmagn2*rmagn))*((3*DotProduct(M,ru))*ru - M)
end
---------------------------------
function vesPeriodicTask(t)
  -- m1 is electron emitter
  -- local coordinates, origin is CM of body pair: m1 is at -L1, m2 is at +L2
  -- current is I(x) = I0*(L2-x)/(L1+L2)
  -- force per length is I(x) x B = I(x)*(u x B) where u is unit vector along tether, pointing
from m1 to m2
  -- force on tether is F=integrate(dx*I(x),x=-L1..L2)*(uxB) = I0*(uxB)*(1/2)*(L1+L2)
        -- torque M = integrate(rxdF) = u x integrate(dx*x*(uxB)*I(x)) =
(ux(uxB))*I0*(1/6)*(L2-2*L1)*(L1+L2)
  --         = (1/3)*(L2-2*L1)*(uxF)
  --         = ((u.B)u-B)*I0*(1/6)*(L2-2*L1)*(L1+L2)
  -- F1=(1-s)*F, F2=s*F, 0<=s<=1, find s from correct torque
  -- torque M2 = u*L2 x F2 = L2*s*(uxF)
  -- torque M1 = (-u*L1) x F1 = -L1*(1-s)*(uxF)
  -- demand M1+M2=M: L2*s-L1*(1-s) = (1/3)*(L2-2*L1) ==> s=1/3
  -- ==> F1=(2/3)*F, F2=(1/3)*F
  local I0 = 30e-3
  local data1 = vesGetBodyData(mass1)
  local data2 = vesGetBodyData(mass2)
  local r1 = Vector.new({data1.x,data1.y,data1.z})
  local r2 = Vector.new({data2.x,data2.y,data2.z})
  local v1 = Vector.new({data1.vx,data1.vy,data1.vz})
  local v2 = Vector.new({data2.vx,data2.vy,data2.vz})
  local rmid = (m1*r1+m2*r2)/(m1+m2)
  local vmid = (m1*v1+m2*v2)/(m1+m2)
  local B = Bmodel(rmid,t)
  local u = Normalise(r2-r1)
  local F = (I0*0.5*tether.len)*CrossProduct(u,B)
  if (DotProduct(F,vmid) < 0) then
        local F1 = (2.0/3.0)*F
        local F2 = (1.0/3.0)*F
        vesSetExternalForce(mass1,F1)
        vesSetExternalForce(mass2,F2)
  else
        vesSetExternalForce(mass1,{0,0,0})
        vesSetExternalForce(mass2,{0,0,0})
  end
  fp:write(string.format(
"%g %g %g %g %g\n",t,rmid[1],rmid[2],rmid[3],vesGetForceValue(tether.force)))
end
```

Table 6: An exemplary VESVISION-v3 Lua script for modelling a satellite in LEO which has deployed a massless tether with an end mass and which is affected by Lorentz and gravity gradient forces. Use of customised Lua 3-vector utility class and simple text file output are also demonstrated. Lines starting by double minus signs are comments.

5. Conclusions

Two numerical models, VESVISION-v2 and VESVISION-v3 have been created and are in active everyday use for simulating the dynamical behaviour of E-sails and related systems. Functionality of the two versions is partly overlapping and partly complementary. Version v2 is more E-sail specific and (possibly) less accurate in its numerical implementation while version v3 is more generic (can simulate and arbitrary collection of point masses and rigid bodies interacting by arbitrary forces) and its integrator is very accurate, but the code only simulates the mechanical behaviour of the tethers, not their self-consistent electrodynamics (unless the user writes such routines himself in Lua). Most of our E-sail models have been run using both tools using a variety of different approximations (for example, tethers have been modelled as chains of point masses as well as chains of rigid bars in version 3) and the results have been found similar. Version 2 is usually faster which has enables longer duration simulations than version 3.

It is a natural question to ask if it would be feasible to have only one tool which combines the benefits of both versions i.e. provides flexibility, very accurate integration and full E-sail specific physics models. The main nontrivial challenge in providing such tool would be to have a modelling framework of the tethers' self-consistent electrodynamics (including self-consistent computation of the voltage along the tether which takes into account ohmic potential drop) which is compatible with the high-order ODE solver. This challenge arises because a natural framework for formulating tether electrodynamics is a partial differential equation while the high-accuracy mechanical framework of version 3 uses ordinary differential equation, ODE. Also, there is the tradeoff of speed and accuracy. Version 2 is sometimes significantly faster than version 3 and as such remains valuable. Our recommendation for now is to continue using both models. In our opinion, this also increases reliability of the analysis because the two models have been developed independently and are using different mathematical modelling approaches.