VirtualBow v0.7.1

User Manual

Contents

1 Introduction

VirtualBow is a software tool for bow and arrow physics simulation. It allows you to design virtual bow models, simulate their static and dynamic behaviour and visualize the results, providing almost immediate feedback about the bow's predicted performance. The application consists of the following three components:

- **VirtualBow:** The model editor and core of the application. Create, load and save bow models (.bow), edit their various parameters and launch static and dynamic simulations. The model editor takes care of calling the other two components, VirtualBow Solver and VirtualBow Post, although those can be used independently as well.

- **VirtualBow Post:** This is the result viewer. It is used for opening and visualizing the simulation results (.res) produced by the solver. Here you can analyze and evaluate your bow designs.

- **VirtualBow Solver:** The solver performs the actual simulation. It takes a model file (.bow), does the necessary numerical calculations and writes a result file (.res) that can be opened with the result viewer or analyzed by other programs.

The following chapters of this manual will show you how to use the software and its components and explain the various input parameters and simulation results. For help and support, giving feedback or to get the latest version of the software and this manual please visit the project's website at http://www.virtualbow.org/

2 VirtualBow

2.1 Overview

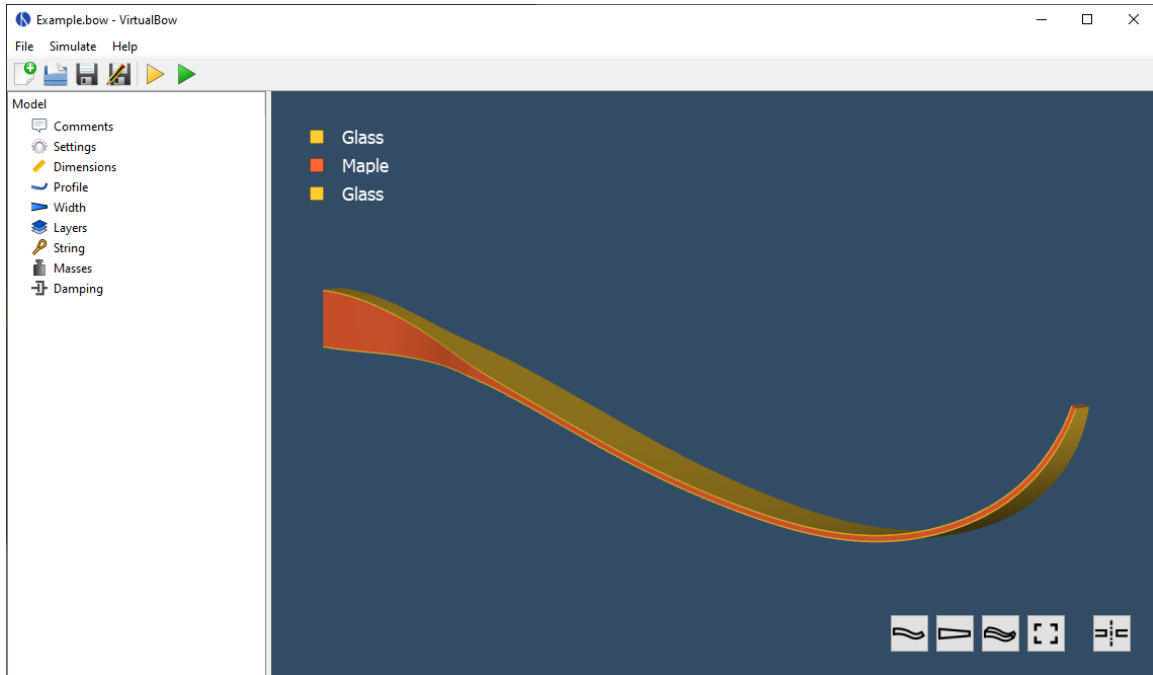This is the main application. Here you can load, edit and save bow models and launch simulations.

Figure 1: Bow Editor

All the parameters of the bow can be accessed via the model tree on the left. Double-click any of the items to edit the respective category. The next sections will cover each of those items in more detail.

The 3D view on the right shows the bow's current geometry resulting from those parameters. Use the mouse to rotate (left button), shift (middle button) and zoom (mouse wheel). More options are available through the buttons on the bottom-right corner.

Use the main menu and/or the toolbar to create, load and save bow models, which are stored as .bow files. Run a static simulation to analyze the bow being drawn or a dynamic simulation to analyze the bow in motion, i.e. when shooting an arrow. The simulation results will be stored next to the model file and opened in VirtualBow Post.

2.2 Parameters

2.2.1 Comments

The comments are meant for documenting the bow model. Any notes about the bow, its paramters or the simulation results can be added here.
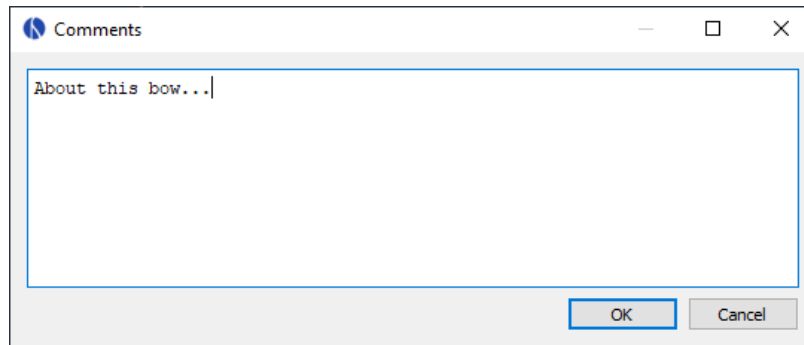

Figure 2: Comments dialog

2.2.2 Settings

These are numerical settings that can be used to tweak the simulation. Most of the time the default values should be fine though, so if you're reading this manual for the first time you might want to skip this section.
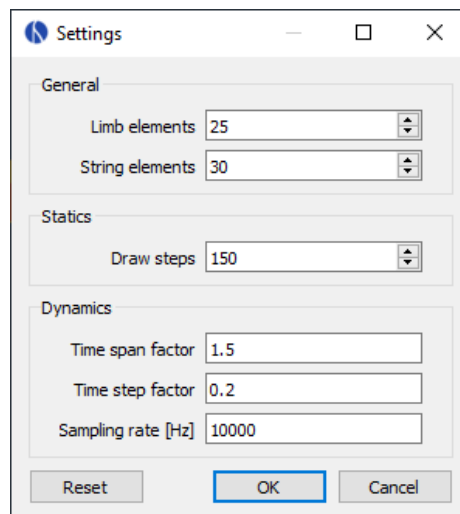

Figure 3: Settings dialog

Since the default settings are meant to be a good general choice, they favor accuracy and reliability over simulation performance. So for specific use cases it might be an advantage to find more efficient settings. Think about running a large number of scripted simulations, for example. On the other hand, even the default settings might sometimes fail with certain bow designs such that different settings have to be used.

General

- **Limb elements:** Number of finite elements that are used to approximate the limb. More elements increase the accuracy but also the computing time.

- **String elements:** Number of finite elements that approximate the string. This number can usually be reduced if the bow has no recurve. In the case of a static analysis with no recurve it can even be set to one without losing any accuracy.
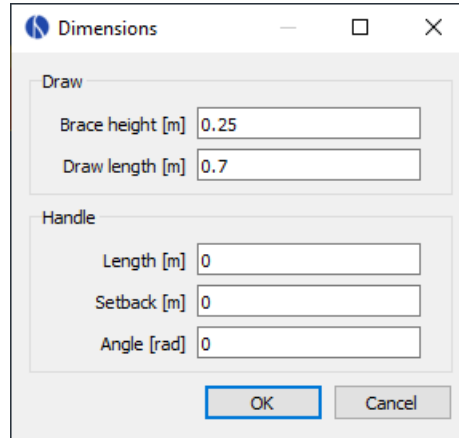
Statics

- **Draw steps:** Number of steps that are performed by the static simulation from brace height to full draw. This determines the resolution of the static results. You can usually decrease this value to speed up the simulation, especially if you're only interested in the dynamic results.

Dynamics

- **Time span factor:** This controls the time period that is simulated. A value of 1 corresponds to the time at which the arrow passes the brace height. The default value is larger than that in order to capture some of the things that occur after the arrow left the bow (maximum dynamic loads on limb and string).

- **Time step factor:** When simulating the dynamics of the bow, the program will repeatedly use the current state of the bow at time t to calculate the next state at time $t + \Delta t$ where Δt is some small timestep. We want this timestep to be as large as possible to keep the computational cost low. But it still has to be small enough to get an accurate and stable solution. The program will estimate this optimal timestep, but to be on the safe side the estimation is multiplied with a reduction factor between 0 and 1 that you can choose here.

- **Sampling rate:** The sampling rate limits the time resolution of the output data. This is done because the dynamic simulation usually produces much finer grained data than is actually useful. Not including all of that in the final output reduces simulation time and memory consumption.

2.2.3 Dimensions

The parameters listed in this dialog define the overall lengths and angles of the bow and its optional stiff middle section. See figure 5 for a visual explanation of those.
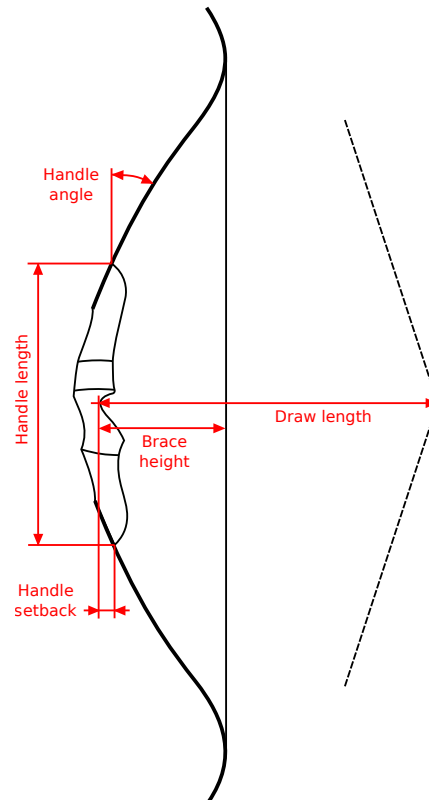
Figure 4: Dimensions dialog

Figure 5: Dimensions of the bow

2.2.4 Profile

The profile curve is the geometric centerline of the bow's limbs in unbraced state. Edit the parameters in the table on the left and view the resulting profile curve on the plot on the right.
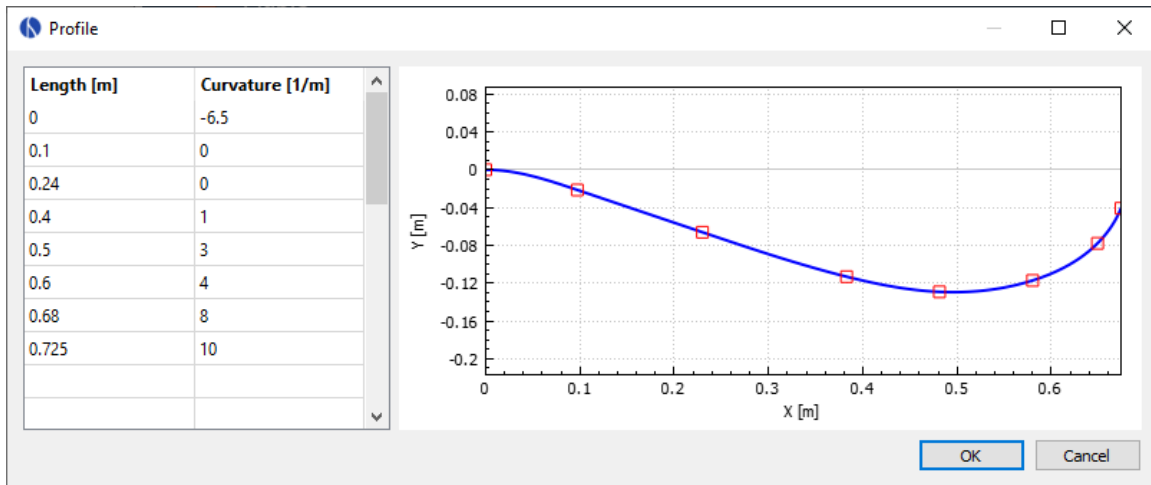

Figure 6: Profile dialog

The profile curve is defined by a series of points where each point has an arc length and a curvature. The arc length is the distance of the point along the curve. Because of that, the last point also defines the total length of the profile curve. The curvature is interpolated linearly between the points, which ensures that there are no jumps in curvature, only smooth transitions.

Note: The profile curve always starts at $(0, 0)$ and with a horizontal angle. Any offsets can be achieved with the parameters under *Dimensions*, see section 2.2.3.

Note: Two points with zero curvature produce a straight line segment while two points with equal and non-zero curvatures produce a circular segment. Two points with different curvatures produce a spiral segment[1] that transitions between the two curvatures.

Note: Mathematically, the curvature κ of a curve is the inverse of its radius of curvature r, so you can calculate the curvature via $\kappa = {}^1\!/_r$ if you know the radius at that point and vice versa.

---

[1]https://en.wikipedia.org/wiki/Euler_spiral

2.2.5 Width

The width dialog is used to define the limb's width along its profile curve. This width is the same for all layers of the bow.
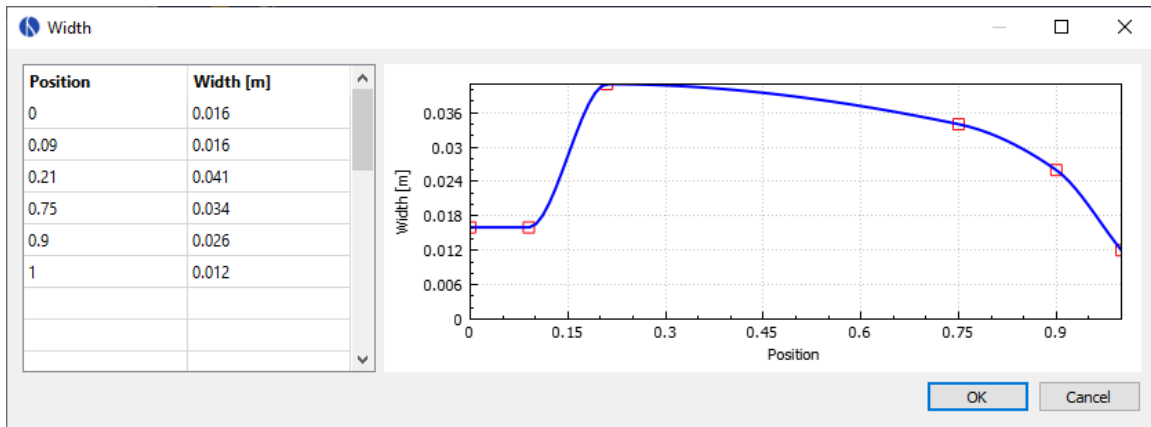
Figure 7: Width dialog

On the table on the left you can specify values for the width at certain relative positions (between 0 and 1) along the limb. This definition of cross section properties relative to the total length of the limb makes it possible to later change the profile curve without having to adjust any cross sections.

The actual width distribution of the limb is constructed as a smooth curve (monotonic cubic spline) passing through the supplied values as shown on the plot on the right.

2.2.6 Layers

With the layer dialog you can create any number of layers and specify their height/thickness and material properties.
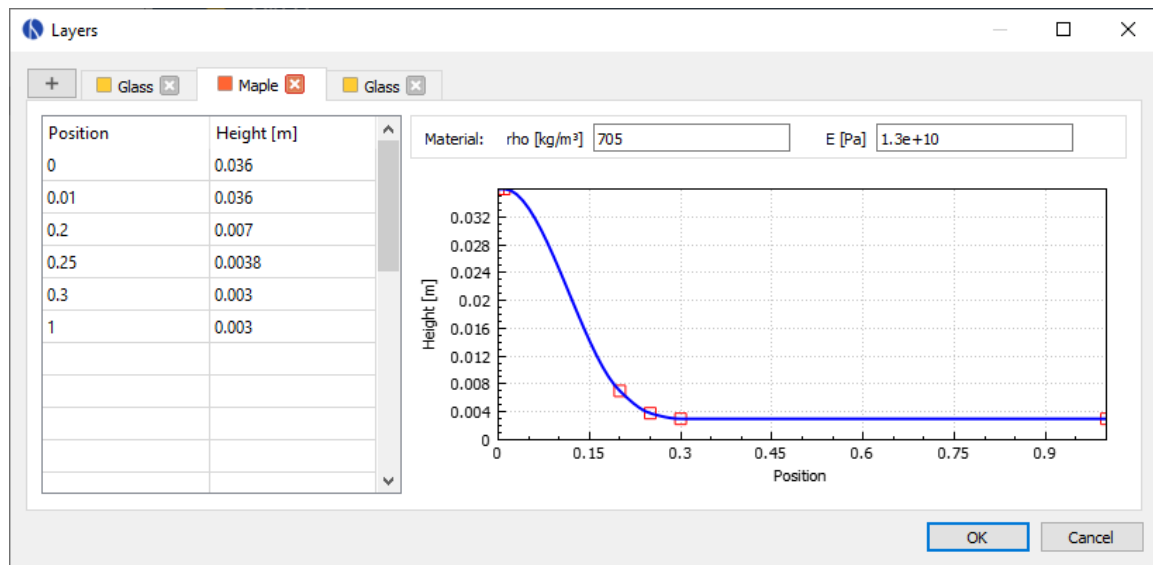
Figure 8: Layer dialog

Click the plus button on the top left to add layers or delete them by closing their tab. Rearrange the order of the layers by dragging the tabs. Double-click on a tab to rename a layer.

The table on the left sets the height distribution of the layer. It works the same way as the limb's width: You specify a number of values at different relative positions, which the program uses to create an interpolating curve. (In this case also a monotonic cubic spline.)

The layer's material is specified by the following two constants,

- **rho:** Density (Mass per unit volume)

- **E:** Elastic modulus (Measure for the stiffness of a material)

For synthetic materials like e.g. fiber-reinforced composites you can often find those numbers in a datasheet provided by the manufacturer. Natural materials like wood are more difficult, because their properties can vary quite a bit. You can find average numbers on the internet, for example at `http://www.wood-database.com`. Those are probably good enough as a first reference. However, in order to be really sure about a specific material there is no other way than to test it. One possibility is a simple bending test as shown in Appendix C.

2.2.7 String

Here you can define the mechanical properties of the string by providing data for the string material and the number of strands being used.
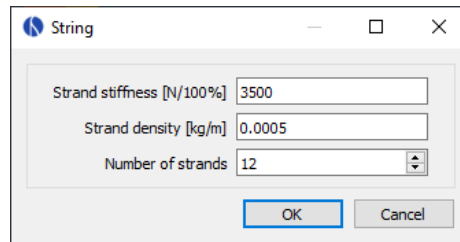

Figure 9: String dialog

- **Strand density:** Linear density of the strands (mass per unit length)
- **Strand stiffness:** Stiffness of the strands against elongation (force per unit strain)
- **Number of strands:** Total number of strands in the string

Note: The stiffness of the string material can be an important parameter in dynamic analysis. The static results however aren't affected very much by it as long as the value is high enough to prevent significant elongation.

Note: The linear density of a string material can be easily determined with a kitchen scale (weight divided by length). The stiffness however is much more difficult to obtain. Manufacturers of string materials usually don't publish those numbers. Table 1 shows the results of tensile tests for three common bow string materials. They were done by the German Institutes for Textile and Fiber Research[1] in July 2018.

Material	Density [kg/m]	Breaking strength [N]	Elongation at break [%]	Stiffness (linearized) [$N/100\%$]
Dacron B50	370e-6	180	8.5	2118
Fastflight Plus	176e-6	318	2.9	10966
BCY 452X	192e-6	309	2.5	12360

Table 1: Test results for some common bowstring materials, the stiffness was estimated from breaking strength and elongation

---

[1] https://www.ditf.de/en/index/ditf.html

2.2.8 Masses

This dialog is used to define various individual masses of the bow. Only the mass of the arrow is actually required, the other ones are optional and may also be set to zero.
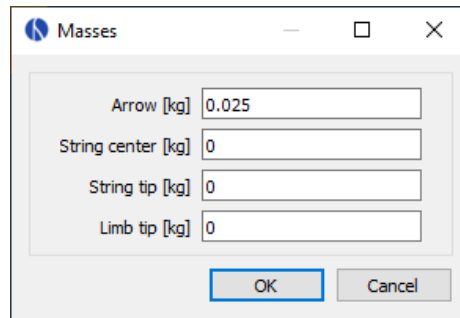
Figure 10: Masses dialog

- **Arrow:** Mass of the arrow
- **String center:** Additional mass at the string center (serving, nocking point)
- **String tip:** Additional mass at the ends of the string (serving, silencers)
- **Limb tip:** Additional mass at the limb tip (nocks, overlays)

2.2.9 Damping

This dialog allows setting a damping ratio for the limbs and string, respectively. Those parameters can be used to adjust for the energy a bow loses by dissipation, for example due to internal friction/hysteresis of the materials.
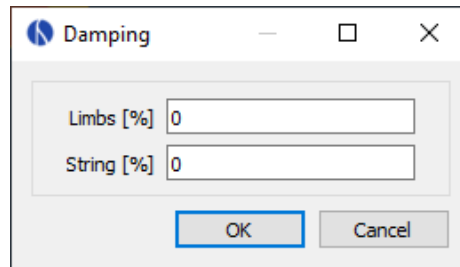
Figure 11: Damping dialog

Note: The damping ratio characterizes how quickly oscillations in a system decay over time. A system with a damping ratio of 0% is undamped, it doesn't dissipate any energy and just keeps going with a constant amplitude. The higher the damping ratio the faster the amplitudes decay over time, losing energy with each oscillation. Once the damping ratio reaches 100% the system no longer oscillates at all (no overshoot), this is called critical damping.
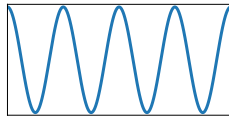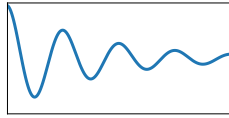
Damping ratio	Amplitude
0%	
10%	
100%	

Table 2: Oscillations with increasing damping ratio

The damping ratios for a bow's limbs and string are mostly empirical and there isn't yet any practical experience with those parameters in VirtualBow. Realistic values are probably in the range of 0 - 20% though.

3 VirtualBow Post

3.1 Overview

This part of the application is used to open and visualize simulation results that are stored in .res files. It is launched automatically by the model viewer when a simulation has finished, but it can also be used as a standalone application.
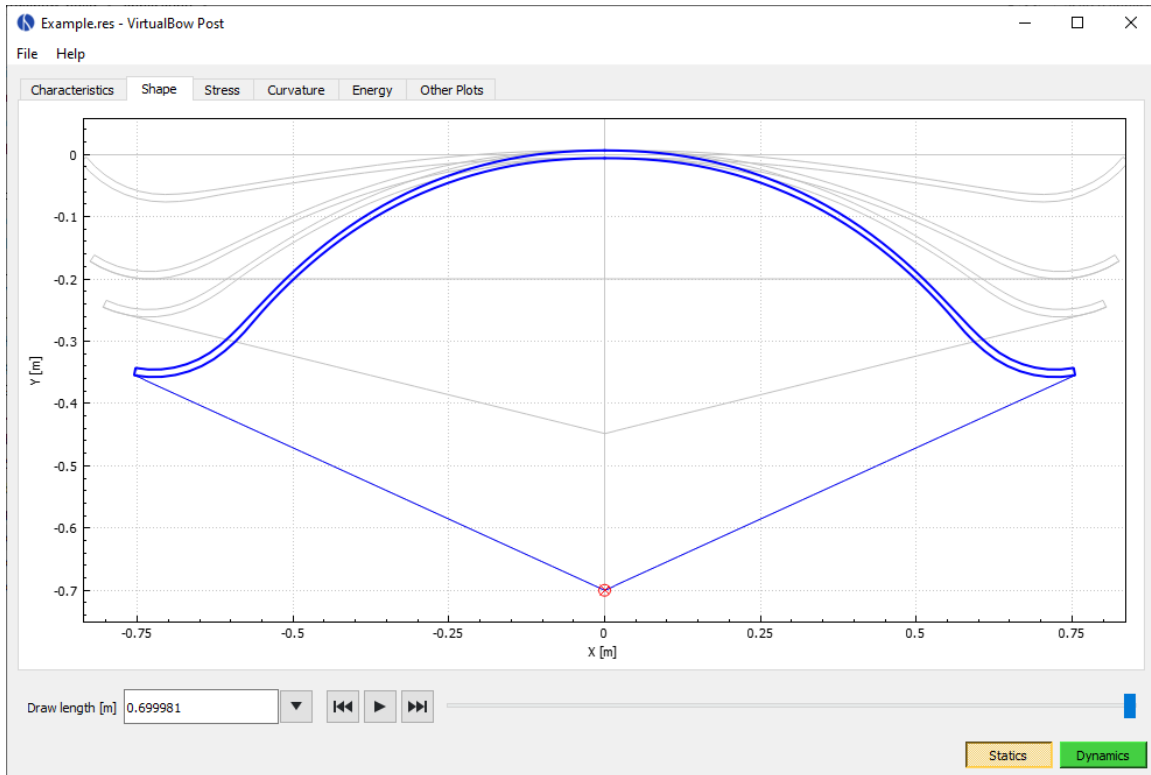
Figure 12: Simulation results

After loading a result file you can use the buttons on the bottom right corner of the window to switch between static and dynamic results, if available. The results themselves are organized in different tabs, which will be explained in the next sections.

Below the tabs there is a slider where you can change the current state of the bow that is being viewed, either by draw length in the static case or by time in the dynamic case. This value applies to all of the tabs. Drag the slider to a specific position or use the play buttons to show the simulation results as a continuous animation. The input field and the dropdown menu next to it allow you to jump directly to special points in the results, for example where certain forces or stresses reach their maximum or the arrow leaves the string.

3.2 Results

3.2.1 Characteristics

This tab shows all results that can be expressed as a single number. Some of them differ between statics and dynamics.

Statics

- **Final draw force:** Draw force of the bow at full draw

- **Drawing work:** Total work done by drawing the bow. This work is stored in the bow as potential elastic energy and later partially transfered to the arrow, depending on the bow's efficiency. It is equal to the area under the force-draw curve of the bow (See also *Other Plots*).

- **Storage ratio:** This is an indicator of how much energy the bow stores and is defined as

$$\text{storage ratio} = \frac{\text{drawing work}}{1/2 \cdot \text{draw force} \cdot (\text{draw length} - \text{brace height})},$$

  which is the amount of energy stored by the bow's draw curve in relation to a linear draw curve with the same final draw force.

- **Limb mass:** Total mass of a single bow limb, including the additional tip mass

- **String mass:** Total mass of the bow string, including additional masses

- **String length:** Length of the string, determined by the desired brace height

Dynamics

- **Final arrow velocity:** Velocity of the arrow when departing from the bow

- **Degree of efficiency:** The bow's degree of efficiency, i.e. useful energy output (kinetic energy of the arrow) divided by energy input (static drawing work)

- **Kinetic energy arrow:** Kinetic energy of the arrow on departure from the string

- **Kinetic energy limbs:** Kinetic energy of the limbs on arrow departure

- **Kinetic energy string:** Kinetic energy of the string on arrow departure

Common

- **Maximum absolute stresses:** Maximum stress by absolute value for each layer

- **Maximum absolute forces:** Maximum forces by absolute value. Entries are the draw force, the string force and the grip support force, i.e. the force that is required to hold the bow handle at its position.

3.2.2 Shape

Shows the shape of the limb and string as well as the position of the arrow at different stages of either the draw (statics) or the shot (dynamics). The little red circle symbolizes the back end of the arrow. You can use the slider at the bottom to change the current state of the bow or to view all states as a continuous animation.

3.2.3 Stress

Shows the stress distribution along the limb for each one of the layers, evaluated at the back and belly sides.

3.2.4 Curvature

Shows the curvature of the limb along its length. This is not the total curvature but rather the difference to the unbraced shape.

3.2.5 Energy

This plot shows how the total amount of energy in the bow develops during the simulation and how it is distributed between components (limbs, string arrow) and type of energy (potential/elastic or kinetic). In the dynamic case it shows how the initial potential energy of the limbs is transferred to the arrow and other components of the bow and how much unused energy stays in the bow after the departure of the arrow.

3.2.6 Other Plots

Here you can combine arbitrary simulation results and plot them together, e.g. things like the draw curve of the bow or the velocity of the arrow.

4 VirtualBow Solver

4.1 Command Line Interface

The VirtualBow solver can be used from the command line to launch simulations in batch mode, without opening the GUI. The executables for the three components of VirtualBow are named

```
virtualbow-gui          // Main program
virtualbow-slv          // Solver
virtualbow-post         // Result viewer
```

The solver command takes a model file as the input and produces a result file as the output. The type of simulation (static or dynamic) as well as other options can be set. The detailed usage as shown by the `--help` option is:

```
> virtualbow-slv --help
Usage: virtualbow-slv [options] input output

Options:
  -?, -h, --help  Displays this help.
  -v, --version   Displays version information.
  -s, --static    Run a static simulation.
  -d, --dynamic   Run a dynamic simulation.
  -p, --progress  Print simulation progress.


Arguments:
  input           Model file (.bow)
  output          Result file (.res)
```

Note: To use the command line interfaces on Windows you have to either specify the complete path to the respective executable or add the installation directory to your PATH environment variable. There is an option to do this automatically during installation.

Note: On MacOS, the VirtualBow executables are hidden inside the application bundle. They can be accessed by their full path though, or their location can be temporarily added to the PATH environment variable with the command

```
export PATH = $PATH:/Applications/VirtualBow.app/Contents/MacOS
```

Put this line into your `.bash_profile` to make the change persistent.

4.2   Scripting

The main use of the command line interface is the ability to write programs that automatically launch simulations and analyze their results. This is especially useful when a large number of simulations has to be done, for example when performing design optimizations or exploring the influence of certain model parameters on the simulation results. Launching simulations from the command line is only one part of this. Scripts that interoperate with VirtualBow also have to be able to read and write the model and result files that constitute the input and output of the solver.

Note: Since VirtualBow is still far from finished, the contents of both of these files are subject to change. The goal is to keep compatibility with older model files as much as possible, such that newer versions of VirtualBow can still open older files. This might not always be possible though. Any scripts that work with those files should therefore expect breaking changes in the future, which means that they will have to be updated in order to keep working with newer versions of VirtualBow.

Model Files

The model files have a .bow extension, but are in fact just plain text JSON[1] files. JSON is a very common human readable format for storing data in the form of objects, arrays, strings, numbers and more in a hierarchical way. As the format is text based, it is possible to just open the model files with a text editor and have a look at contents. For a detailed specification of the model files, see appendix A.1.

Result Files

The result files use the binary MessagePack[2] format, which is more space efficient than JSON but very similar in the kind of data it can represent. For a detailed specification of the result files, see appendix A.2.

Examples

As a consequence of the model and result file formats, VirtualBow can interoperate with any programming language that supports JSON and MessagePack. Both are very common formats and many programming languages have either built-in support or external libraries available. Examples for several languages that are commonly used in scientific computing can be found in appendix B.

---

[1]http://json.org
[2]http://msgpack.org

A File Formats

A.1 Model Files (.bow)

Field	Type	Unit	Description
version	`string`	–	VirtualBow version
comment	`string`	–	User comment
settings			
n_limb_elements	`integer`	–	Number of limb elements
n_string_elements	`integer`	–	Number of string elements
n_draw_steps	`integer`	–	Number of steps for the static simulation
time_span_factor	`double`	–	Factor for modifying total simulation time
time_step_factor	`double`	–	Factor for modifying simulation time steps
sampling_rate	`double`	Hz	Time resolution for the dynamic output
dimensions			
brace_height	`double`	m	Brace height
draw_length	`double`	m	Draw length
handle_length	`double`	m	Handle length
handle_setback	`double`	m	Handle setback
handle_angle	`double`	m	Handle angle
profile	`double[][]`	m, 1/m	Table with arc length and curvature
width	`double[][]`	–, m	Table with position and width
layers			
{			
name	`string`	–	Name of the layer
height	`double[][]`	–, m	Table with positions and heights
rho	`double`	kg/m^3	Density of the layer material
E	`double`	Pa	Elastic modulus of the layer material
}			
{ ... }			
string			
strand_stiffness	`double`	N	Stiffness of the string material
strand_density	`double`	kg/m	Density of the string material
n_strands	`integer`	–	Number of strands
masses			
arrow	`double`	kg	Mass of the arrow
string_center	`double`	kg	Additional mass at string center
string_tip	`double`	kg	Additional mass at string tips
limb_tip	`double`	kg	Additional mass at limb tips
damping			
damping_ratio_limbs	`double`	–	Damping ratio of the limbs
damping_ratio_string	`double`	–	Damping ratio of the string

A.2 Result Files (.res)

P: Number of limb nodes, Q: Number of string nodes, R: Number of layer nodes

Field	Type	Unit	Description
setup			
string_length	double	m	Length of the string
string_mass	double	kg	Mass of the string, including additional masses
limb_mass	double	kg	Mass of one limb, including additional masses
limb_properties			
length	double[P]	m	Arc lengths of the limb nodes (unbraced)
angle	double[P]	rad	Orientation angles of the limb nodes (unbraced)
x_pos	double[P]	m	X coordinates of the limb nodes (unbraced)
y_pos	double[P]	m	Y coordinates of the limb nodes (unbraced)
width	double[P]	m	Cross section width
height	double[P]	m	Cross section height (total)
rhoA	double[P]	kg/m	Linear density of the cross sections
Cee	double[P]	N	Longitudinal stiffness of the cross sections
Ckk	double[P]	Nm^2	Bending stiffness of the cross sections
Cek	double[P]	Nm	Coupling between bending and elongation
layers			
{			
length	double	m	Arc lengths of the layer nodes
He_back	double[R][P]	N/m^2	Stress evaluation matrix[1] (back)
Hk_back	double[R][P]	N/m	Stress evaluation matrix[1] (back)
He_belly	double[R][P]	N/m^2	Stress evaluation matrix[1] (belly)
Hk_belly	double[R][P]	N/m	Stress evaluation matrix[1] (belly)
}			
{ ... }			
statics			
final_draw_force	double	N	Final draw force
drawing_work	double	J	Drawing work
storage_ratio	double	–	Storage ratio
max_string_force_index	integer	–	Simulation state with maximum absolute string force
max_grip_force_index	integer	–	Simulation state with maximum absolute grip force
max_draw_force_index	integer	–	Simulation state with maximum absolute draw force
max_stress_value	double[]	Pa	Maximum absolute stresses for each layer
max_stress_index	integer[][]	–	Simulation state and layer node for each stress value
states			
{ ... }			Sequence of bow states (see table below)
dynamics			
final_pos_arrow	double	m	Position of the arrow at departure
final_vel_arrow	double	m/s	Final velocity of the arrow
final_e_pot_limbs	double	J	Final potential energy of the limbs
final_e_kin_limbs	double	J	Final kinetic energy of the limbs
final_e_pot_string	double	J	Final potential energy of the string
final_e_kin_string	double	J	Final kinetic energy of the string
final_e_kin_arrow	double	J	Final kinetic energy of the arrow
efficiency	double	–	Degree of efficiency
max_string_force_index	integer	–	Simulation state with maximum absolute string force
max_grip_force_index	integer	–	Simulation state with maximum absolute grip force
arrow_departure_index	integer	–	Simulation state at which the arrow leaves the bow
max_stress_value	double[]	Pa	Maximum absolute stresses for each layer
max_stress_index	integer[][]	–	Simulation state and layer node for each stress value
states			
{ ... }			Sequence of bow states (see table below)

`N`: Number of simulation steps
`P`: Number of limb nodes
`Q`: Number of string nodes

Field	Type	Unit	Description
states			
time	`double[N]`	s	Time
draw_length	`double[N]`	m	Draw length
draw_force	`double[N]`	N	Draw force
string_force	`double[N]`	N	String force (total)
strand_force	`double[N]`	N	String force (strand)
grip_force	`double[N]`	N	Grip force
pos_arrow	`double[N]`	m	Arrow position
vel_arrow	`double[N]`	m/s	Arrow velocity
acc_arrow	`double[N]`	m/s^2	Arrow acceleration
x_pos_limb	`double[N][P]`	m	X coordinates of the limb nodes
y_pos_limb	`double[N][P]`	m	Y coordinates of the limb nodes
angle_limb	`double[N][P]`	rad	Rotation angles of the limb nodes
epsilon	`double[N][P]`	–	Longitudinal strain at the limb nodes
kappa	`double[N][P]`	m	Bending curvature at the limb nodes
x_pos_string	`double[N][Q]`	m	X coordinates of the string nodes
y_pos_string	`double[N][Q]`	m	Y coordinates of the string nodes
e_pot_limbs	`double[N]`	J	Potential energy of the limbs
e_kin_limbs	`double[N]`	J	Kinetic energy of the limbs
e_pot_string	`double[N]`	J	Potential energy of the string
e_kin_string	`double[N]`	J	Kinetic energy of the string
e_kin_arrow	`double[N]`	J	Kinetic energy of the arrow

[1]**Note:** For space efficiency reasons, the stresses for each layer aren't stored directly in the result files. Instead they can be calculated as needed by multiplying the layer's stress evaluation matrices with the strain and curvature of the limb at the given bow state,

$$\mathtt{sigma\_back} = \mathtt{He\_back} \cdot \mathtt{epsilon} + \mathtt{Hk\_back} \cdot \mathtt{kappa}$$
$$\mathtt{sigma\_belly} = \mathtt{He\_belly} \cdot \mathtt{epsilon} + \mathtt{Hk\_belly} \cdot \mathtt{kappa}$$

The result is a vector of stresses corresponding to the nodes of the layer.

B Scripting Examples

These examples show how to use VirtualBow with several common scientific programming languages. All of these examples perform the same series of basic tasks:

1. Load, modify and save a model file

2. Run a static simulation with the model

3. Load the result file and evaluate the maximum stress of the first layer at full draw

B.1 Python

Python requires two external packages: msgpack[1] for reading the result files and NumPy[2] for evaluating the stresses. They can be installed via `pip install msgpack numpy`.

```python
import json              # Loading and saving model files
import msgpack           # Loading result files
import numpy as np       # Evaluating stresses
import subprocess        # Runnig the simulation

# Load model file
with open("input.bow", "r") as file:
    input = json.load(file)

# Modify model data
input["string"]["n_strands"] += 1

# Save model file
with open("input.bow", "w") as file:
    json.dump(input, file, indent=2)

# Run a static simulation
subprocess.call(["virtualbow-slv", "--static", "input.bow", "output.res"])

# Load the result file
with open("output.res", "rb") as file:
    output = msgpack.unpack(file, raw=False)

# Evaluate stresses
He_back = np.array(output["setup"]["limb_properties"]["layers"][0]["He_back"])
Hk_back = np.array(output["setup"]["limb_properties"]["layers"][0]["Hk_back"])

epsilon = np.array(output["statics"]["states"]["epsilon"][-1])
kappa   = np.array(output["statics"]["states"]["kappa"][-1])
sigma_back = He_back.dot(epsilon) + Hk_back.dot(kappa)

print(sigma_back.max())
```

---

[1] https://pypi.org/project/msgpack/
[2] https://pypi.org/project/numpy/

B.2 Matlab

This example uses the JSONLab[1] library, which can read and write both JSON and MessagePack files.

Note: Support for MessagePack in JSONLab is still fairly new. The current version 1.9.8 - beta (at the time of writing) contains a bug that prevents it from loading files with empty arrays. This can be the case with VirtualBow result files. The problem has been fixed in the latest development version[2], so we have to use that until the next release comes out.

```matlab
% Load model file
input = loadjson('input.bow');

% Modify model data
input.string.n_strands = input.string.n_strands + 1;

% Save model file
savejson('', input, 'input.bow');

% Run a static simulation
system('virtualbow-slv --static input.bow output.res');

% Load the result file
output = loadmsgpack('output.res');

% Evaluate stresses
He_back = output.setup.limb_properties.layers{1}.He_back;
Hk_back = output.setup.limb_properties.layers{1}.Hk_back;

epsilon = output.statics.states.epsilon(:,end);
kappa   = output.statics.states.kappa(:,end);
sigma = He_back*epsilon + Hk_back*kappa;

disp(max(sigma));
```

---

[1] https://de.mathworks.com/matlabcentral/fileexchange/33381
[2] https://github.com/fangq/jsonlab

B.3 Julia

Two external packages are used here, JSON[1] for loading model files and MsgPack[2] for loading result files. They can be installed with `julia> import Pkg; Pkg.add("JSON"); Pkg.add("MsgPack")`.

```julia
using JSON        # Loading and saving model files
using MsgPack     # Loading result files

# Load model file
stream = open("input.bow", "r")
input = JSON.parse(stream)
close(stream)

# Modify model data
input["string"]["n_strands"] += 1

# Save model file
stream = open("input.bow", "w")
JSON.print(stream, input, 2)
close(stream)

# Run a static simulation
run(`virtualbow-slv --static input.bow output.res`)

# Load the result file
stream = open("output.res", "r")
output = unpack(stream)
close(stream)

# Evaluate stresses
He_back = hcat(output["setup"]["limb_properties"]["layers"][1]["He_back"]... )
Hk_back = hcat(output["setup"]["limb_properties"]["layers"][1]["Hk_back"]... )

epsilon = output["statics"]["states"]["epsilon"][end]
kappa   = output["statics"]["states"]["kappa"][end]
sigma = He_back*epsilon + Hk_back*kappa

println(max(sigma...))
```

---

[1]https://juliahub.com/ui/Packages/JSON/uf6oy/0.21.0
[2]https://juliahub.com/ui/Packages/MsgPack/oDgLV/1.1.0

C Bending Test

An easy way to determine the elastic modulus of a material is a bending test. It can be done without any special equipment. Figure 13 shows the setup. A test piece with length l is clamped on one side and subjected to a vertical force F at its free end. The deflection s due to this load is measured.
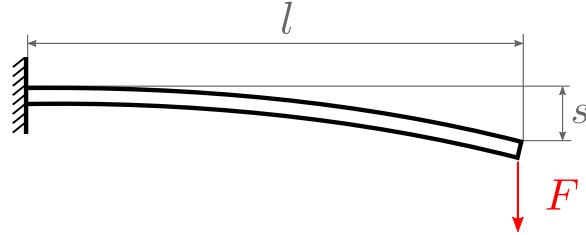

Figure 13: Experimental setup

The elastic modulus can then be calculated depending on the cross section geometry using the equations in Table 3.

Test geometry	Elastic Modulus
	$E = \dfrac{4}{wh^3}\dfrac{Fl^3}{s}$
	$E = \dfrac{12\,\ln(h_l\,l) + 6}{w\,(h_l - h_0)^3}\dfrac{Fl^3}{s}$

Table 3: Elastic modulus for different test geometries

Note: The following practical considerations might help with choosing a good test setup,

- The precision of the cross sections is very important, especially the height.

- The equations above hold for slender beams and small deflections. The test setup should be chosen accordingly. As a rule of thumb: $h, s < l/15$.

- A simple way to apply a defined force is to hang a mass m onto the beam tip and use $F = m \cdot g$, with $g = 9.81\,\mathrm{m/s^2}$.

- If there is some small initial deflection due to gravity, then s is simply the difference in deflection after application of the force.