# GaitAnalysisToolKit Documentation

***Release 0.2.0dev***

**Jason K. Moore**

December 08, 2014

Contents

Contents:

CHAPTER 1

# Motek Module

Motek Medical sells hardware/software packages which include treadmills with force plate measurement capabilities and motion bases, motion capture systems, visual displays, and other sensors for various measurements. Their software, D-Flow, manages the data streams from the various systems and is responsible for displaying interactive visuals, sounds, and motions to the subject. The `gaitanalysis.motek` module includes classes that eases processing the data collected from typical D-Flow output files, but may have some limitations with respect to the hardware because we only have one system available.

The Human Motion and Control Lab at Cleveland State University has such a system. Our system includes:

- A ForceLink R-Mill which has dual 6 DoF force plates, independent belts for each foot, and lateral and pitch motion capabilities.
- A 10 Camera Motion Analysis motion capture system which includes the Cortex software and hardware for collecting analog and camera data simultaneously.
- Delsys wireless EMG + 3D Accelerometers.
- Motek Medical's D-Flow software and visual display system.

Cortex alone is capable of delivering data from the cameras, force plates, and analog sensors (EMG/Accelerometer), but D-Flow is required to collect data from digital sensors and the treadmill's motion (lateral, pitch, and belts). D-Flow can output multiple types of files which contain the different data.

Our motion capture system's coordinate system is such that the X coordinate points to the right, the Y coordinate points upwards, and the Z coordinate follows from the right-hand-rule, i.e. points backwards with respect to a person walking forward on the treadmill. The camera's coordinate system is aligned to an origin point on treadmill's surface during camera calibration.

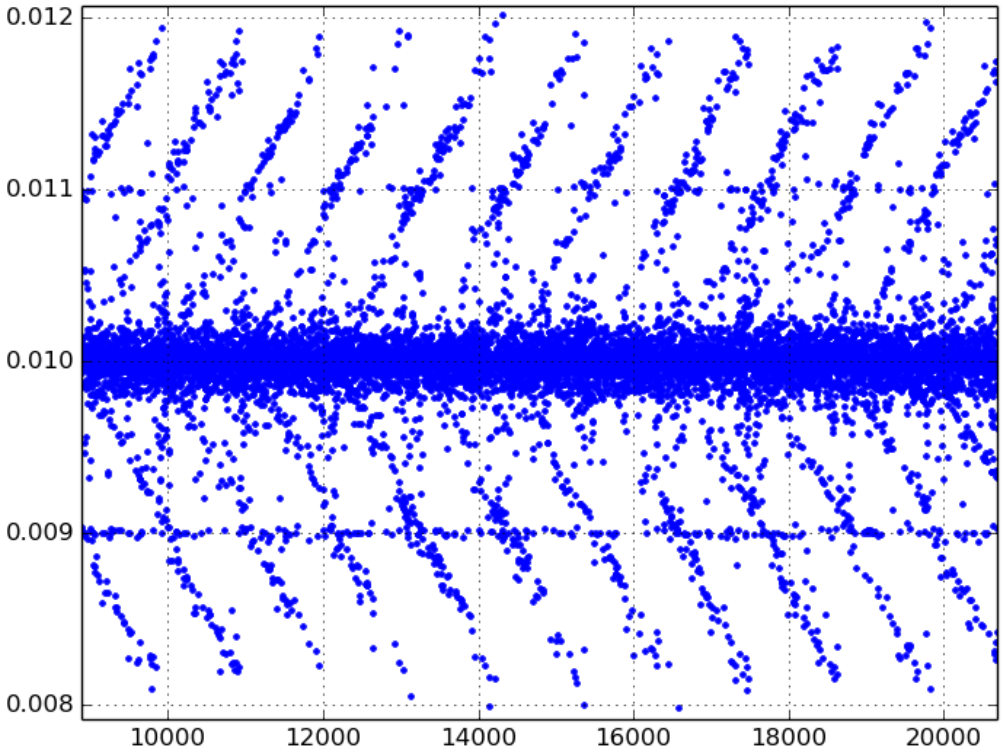
## 1.1 Mocap Module

D-Flow's mocap module has a file tab which allows you to export the time series data collected from Cortex in two different file formats: tab separated values (TSV) and the C3D format (see http://www.c3d.org). The TSV files are approximately twice the size of the C3D files, don't maintain machine precision, and do not allow for meta data storage. But for now, this software only deals with the TSV file format.

The text file output from the mocap module in DFlow is a tab delimited file. The first line is the header and contains a time stamp column, frame number column, marker position columns, force plate force/moment columns, force plate center of pressure columns, other analog columns, and potentially results from the real time Human Body Model which is included with the D-Flow software. These are detailed below. The numerical values of the measurements are provided in decimal floating point notation with 6 decimals of precision, e.g. `123456.123456` [`%1.6f`].

### 1.1.1 Data Column Descriptions

**Time Stamp** The `TimeStamp` column records the D-Flow system time when it receives a "frame" from Cortex in seconds since D-Flow was started. This is approximately at 100 hz (Cortex's sample rate), but has slight variability per sample period, something like +/- 0.002 s or so. This column can be used to synchronize with other D-Flow output files which include a D-Flow time stamp, e.g. the output of the record module. The following figure shows the difference, `.diff()`, of an example D-Flow time stamp, giving the variability in periods at each measurement instance.




**Frame Number** The `FrameNumber` column gives a positive integer to count the frame numbers delivered by Cortex. It seems as though none of the frames are ever dropped but this should be verified.

**Marker Coordinates** The columns that correspond to marker coordinates have one of three suffixes: `.PosX`, `.PosY`, `.PosZ`. The prefix is the marker name which is set by providing a name to the marker in Cortex. There are specific names which are required for D-Flow's Human Body Model's computations. The marker coordinates are given in meters. See below for some additional virtual markers.

**Force Plate Kinetics** There are three forces and three moments recorded by each of the two force plates in Newtons and Newton-Meters, respectively. The prefix for these columns is either `FP1` or `FP2` and represents either force plate 1 (left) or 2 (right). The suffixes are either `.For[XYZ]`, `.Mom[XYZ]` for the forces and moments, respectively. The force plate voltages are sampled at a much higher frequency than the cameras, but delivered at the Cortex camera sample rate, 100 Hz through the D-Flow mocap module. A force/moment calibration matrix stored in Cortex converts the voltages to forces and moments before sending it to D-Flow [1]. Cortex also computes the center of pressure from the forces, moments, and force plate dimensions. These have the same prefixes for the plate number, have the suffix `.Cop[XYZ]`, and are in meters.

[1] Cortex currently does not output anything for the `.MomY` momemt on both of the force plates. So D-Flow records the raw voltages from Cortex and applies the calibration matrix in D-Flow to get correct values using an `.idc` file.

**Analog Channels** Cortex can sample additional analog channels. These columns have headers which take this form `Channel[1-99].Anlg` and the names are fixed to correspond to the channels in the National Instruments DAQ box which samples the analog sensors. The first twelve of these are reserved for the force plate voltage measurements. These correspond to the voltages of the force sensors in the two force plates and are as follows (channels 1-12).

1. F1Y1
2. F1Y2
3. F1Y3
4. F1X1
5. F1X2
6. F1Z1
7. F2Y1
8. F2Y2
9. F2Y3
10. F2X1
11. F2X2
12. F2Z1

Top View of treadmill surface showing location of the Y sensors:

```
----------------------------
|    FP1      |     FP2     |
|             |             |
|          Y2 | Y2          |
|             |             |
|             |             |
| Y1          |          Y3 |    ----> X
|             |             |    |
|             |             |    V
|          Y3 | Y1          |    Z
|             |             |
----------------------------
```

The remaining analog channels are connected to the 16 Delsys EMG/Accelerometers measurements. Each sensor has four signals: EMG, AccX, AccY, and AccZ. The are ordered in the remaining channels as:

13. EMG1
14. ACCX1
15. ACCY1
16. ACCZ1
17. EMG2
18. ACCX2
19. ACCY2
20. ACCZ2
21. etc.

**Note that all of the signals are in volts!**. You must scale them yourself.

---

**Note:** The EMG/Acceleromter channels are 96 milliseconds behind the force plate measurements, according to the DelSys manual [2]. There may be other delays present too that may or may not be taken care of in Cortex or D-Flow. The lag of the EMG/Accelerometers is due to the wireless communication.

---

**Human Body Model** The mocap TSV file can also contain joint angles [degrees], joint moments [Newton-Meters], joint power [Watts], and muscle forces [Newtons] computed by the real time Human Body model. The joint angle headers end in `.Ang`, the joint moments in `.Mom`, the joint power `.Pow`, and the muscle forces are prefixed with `R_` or `L_`. D-Flow also outputs the centor of mass in meters of the person in the `HBM.COM.[XYZ]` columns.

**Segment Positions and Rotations** D-Flow also outputs positional and rotational information about body segments. There are virtual markers with suffixes `.Pos[XYZ]` And there are also segment rotations in degrees. These header labels end in `.Rot[XYZ]`. The definition of the positions and rotations is unclear and it is unclear what they are used for. The following list gives the prefixes:

- `pelvis`
- `thorax`
- `spine`
- `pelvislegs`
- `lfemur`
- `ltibia`
- `lfoot`
- `toes`
- `rfemur`
- `rtibia`
- `rfoot`
- `rtoes`

---

**Todo**

There are probably more of these for the upper body.

---

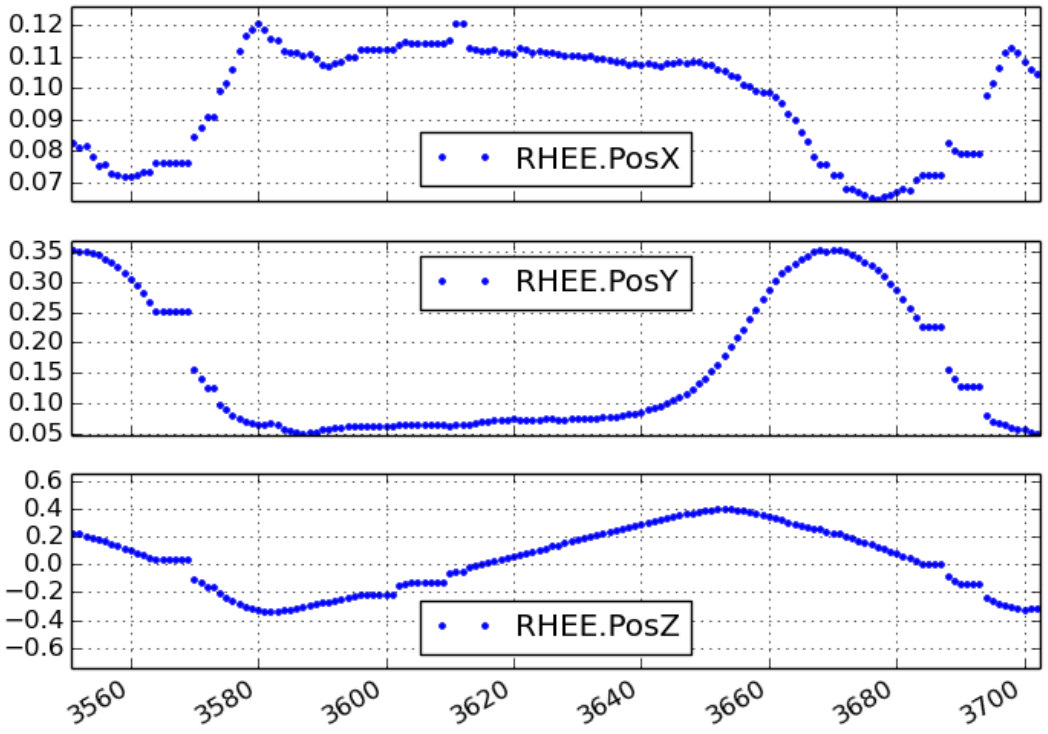
### 1.1.2 Missing Values

D-Flow handles missing values internally to perform well with their real time computations, but there are some important issues to note when dealing with the data outputs from D-Flow with regards to missing values. Depending on how many markers were used, where they were placed, and what analysis is used, different techniques can be used to fill in the gaps.

Firstly, the markers sometimes go missing (i.e. can't been seen by the cameras) which is typical of motion capture systems. Care must be taken that all markers are always captured by the system, but there will always be some missing values. If the data was recorded in a D-Flow version less than 3.16.2rc4 [3], D-Flow records the last non-missing value in all three axes until the marker is visible again when a marker goes missing. The following figure gives an example:

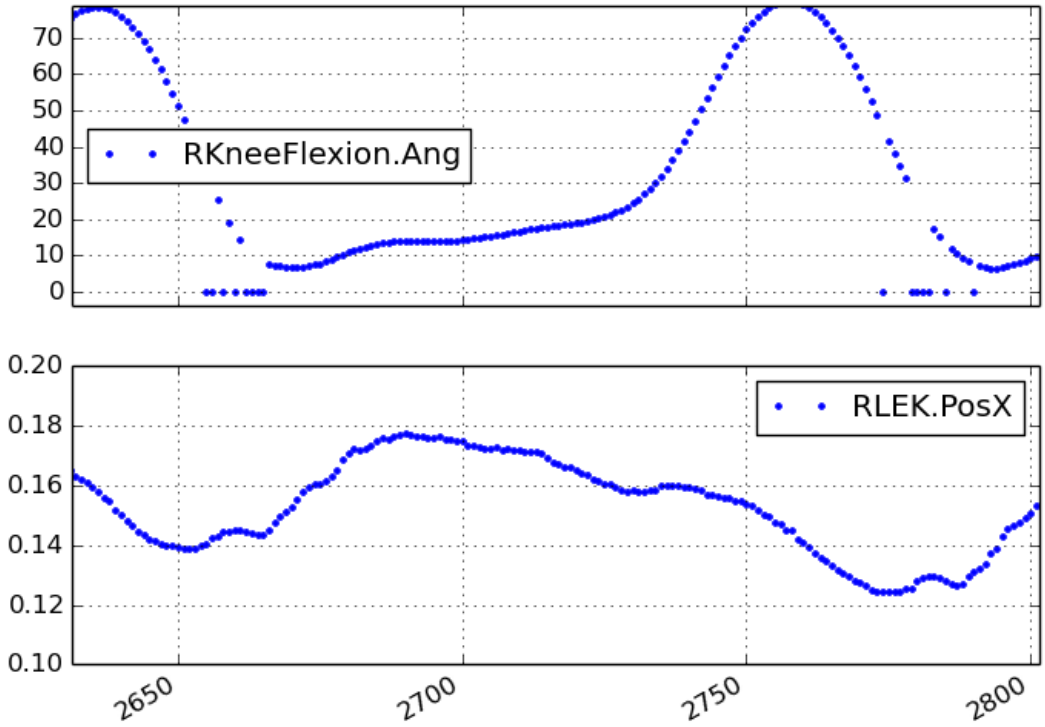
[2] We've done independent measurements that show a ~72 millisecond delay.

[3] We received versions 3.16.1 and then 3.16.2rc4 so I have no idea when the change was introduced between those versions. If this software is used with a version between 3.16.1 and 3.16.2c4, then it may or may not work correctly.




In D-Flow versions greater than or equal to 3.16.2rc4 the missing markers are indicated in the TSV file as either `0.000000` or `-0.000000`, which is the same as has been in the HBM columns in all versions of D-Flow. **The D-Flow version must be provided in the meta data yml file, otherwise it will assume D-Flow is at the latest version.**

The mocap output file can also contain variables computed by the real time implementation of the Human Body Model (HBM). If the HBM computation fails at a D-Flow sample period, strings of zeros, either `0.000000` or `-0.000000`, are inserted for missing values. The following figure shows the resulting HBM output with zeros:




Notice that failed HBM computations don't always correspond to missing markers.

The HBM software only handles zero values for marker coordinates. If markers are zero, then HBM ignores them and tries to compute the inverse dynamics with a reduced set of markers. So if you playback recordings which have missing markers stored as constant values in D-Flow, you will likely get incorrect inverse dynamics.

### 1.1.3 Time Delays

There are time delays between the camera marker data, force plate analog signals, and the wireless EMG/Accelerometers. The documentation for the Delsys wireless system explicity states that there is a 96ms delay in the data with respect to the analog signals that are sampled from the unit which is due to the wireless data transfer. There is also an measurable delay in the camera data with respect to the analog data which seems to hover around 7 ms.

### 1.1.4 Other

Note that the order of the "essential" measurements in the file must be retained if you expect to run the file back into D-Flow for playback. I think the essential measurements are the time stamp, frame number, marker coordinates, and force plate kinetics, and analog channels [4] (maybe because of the IDC file).

[4] The first twelve analog channels may only be required because we use the `.idc` file to work around the fact that the `.MomY` force plate moments are not correctly collected by D-Flow from Cortex.

### 1.1.5 Inertial Compensation

If you accelerate the treadmill there will be forces and moments measured by the force plates that simply come from the inertial effects of the motion. When external loads are applied to the force plates, you must subtract these inertial forces from the measured forces to get correct estimates of the body fixed externally applied forces.

The markers are measured with respect to the camera's inertial reference frame, earth, but the treadmill forces are measured with respect to the treadmill's laterally and rotationally moving reference frame. We need both to be expressed in the same inertial reference frame for ease of future computations.

To deal with this we measure the location of additional markers affixed to the treadmill and the 3D acceleration of the treadmill at 4 points.

Typically, the additional accelerometers are connected to these channels and the arrow on the accelerometers which aligns with the local X axis direction is always pointing forward (i.e. aligned with the negative z direction).

```
# Front left
Channel13.Anlg : EMG
Channel14.Anlg : AccX
Channel15.Anlg : AccY
Channel16.Anlg : AccZ

# Back left
Channel17.Anlg : EMG
Channel18.Anlg : AccX
Channel19.Anlg : AccY
Channel20.Anlg : AccZ

# Front right
Channel21.Anlg : EMG
Channel22.Anlg : AccX
Channel23.Anlg : AccY
Channel24.Anlg : AccZ

# Back right
Channel25.Anlg : EMG
Channel26.Anlg : AccX
Channel27.Anlg : AccY
Channel28.Anlg : AccZ
```

This information will be stored in the meta data file, see below.

Location of of accels and markers should stay the same between unloaded and loaded trials, but position doesn't matter other wise.

## 1.2 Record Module

The record module in D-Flow allows one to sample any signal available in the D-Flow environment at the variable D-Flow sample rate which can vary from 0 to 300 Hz depending on how fast D-Flow is completing it's computations. Any signal that you desire to record, including the ones already provided in the Mocap Module, are available. This is particularly useful for measuring the motions of the treadmill: both belts' speed, lateral motion, and pitching motion. The record module only outputs a tab delimited text file. It includes a `Time` column which records the D-Flow system time in seconds which corresponds to the same time recorded in the `TimeStamp` column in mocap module tsv file. And it additionally records the 6 decimal precision values of other measurements that you include. Finally, the record module is capable of recording the time at which various D-Flow events occur. It does this by inserting commented (#) lines in between the rows when the event occurred. For example an event may look like:

```
#
# EVENT A - COUNT 1
#
```

Where `A` is the event name (fixed by D-Flow, you can't select custom names) and the number after *COUNT* gives an integer count of how many times that event has occurred. D-Flow only seems to allow a total of 6 unique events to be recorded, with names A-F. At the end of the file the total number of event occurrences are counted:

```
# EVENT A occured 1 time
# EVENT B occured 1 time
# EVENT C occured 1 time
# EVENT D occured 1 time
# EVENT E occured 1 time
```

### 1.2.1 Treadmill

The right and left belt speeds can be measured with the record module. You must select a check box in the treadmill module to ensure that the actual speed is recorded and not the desired speed. It does not seem possible to measure the pitch angle nor the lateral position of the treadmill using the record module, it only records the desired (the input) to each.

## 1.3 Meta Data

D-Flow does not have any way to store meta data with its output. This is unfortunate because the C3D format has full support for meta data. It is also possible to add meta data into the header of text files, but it is not the cleanest solution. So we've implemented our own method to track this information. The `DFlowData` class has the option to include a meta data file with the other data files that can record arbitrary data about the trial. Things like subject id, subject body segment parameter info, trial description, etc can and should be included. This data will be available for output to the C3D format or other data storage formats and can be used for internal algorithms in further analysis.

The meta data file must conform to the YAML format, which is a common human readable data serialization format. As time progresses the structure of the meta data file will become more standard, but for now there are only a few requirements.

### 1.3.1 Basics

There are some standard meta data that should be collected with every trial.

```
study:
    id: 58
    name: Control Identification
    description: Perturb the subject during walking and running.
subject:
    id: 567
    birthdate: 1982-05-17
    age: 28
    mass: 70
    mass-units: kilogram
    height: 1.82
    height-units: meters
    gender: male/female # for body seg calcs in hbm
trial:
    id: 1
```

```
datetime: 2013-12-03 05:06:00
notes: text to give anomalies
nominal-speed: 5
nominal-speed: m/s
stationary-platform: True/False
pitch: True
sway: True
marker-set: full/lower/NA
dflow-version: 3.16.1
hardware-settings:
   high-performance: True/False
files:
    compensation: ../T002/mocap-module-002.txt
    mocap: mocap-module-001.txt
    record: record-module-001.txt
    cortex: cortex-001.cap
    mox: gait-001.mox
    meta: meta-001.yml
marker-map:
    M5: T10
    M6: STRN
```

---

**Todo**

HBM requires some measurements of the person and that can be found in the HBM tab of the mocap module. We should include those here. ankle width, knee with, cuttoff frequency.

---

**Todo**

We need to store the scaling factors/matrices for the analog signals in the meta data.

---

**study**

**id** Some unique identified for your study.

**name** A string which contains the name of the project.

**description** One or more sentences that give a basic description of the project.

**subject**

**id** A unique identifier for the subject in this trial. This can be a number, a string, etc.

**birth-date** A date formatted string that gives the subjects birthdate.

**age** A integer giving the subjects age in years at the time of the trial. It's better to provide the subject's birthdate so that the age can be computed for the date of the trial.

**mass** A positive real number giving the subjects weight. Note that actual weight on the trial day can likely be computed from the force plate data and that should be used for accuracy purposes.

**mass-units** The full name or standard unit symbol for the mass quantity.

**height** A positive real number giving the subject's height the day of the trial.

**height-units** The full name or standard unit symbol for the height quantity.

**gender** A string describing the gender of the subject.

**trial**

**id** A unique identifier for this trial. The meta file name should also include this identifier.

**datetime** A date formatted string giving the date and/or time of the trial. If you are concerned about the time zone, UTC time is the best to use here.

**notes** A string with a any notes about the trial. The more of this information that can be included in structured tags in the meta.yml file the better. This should be a catch-all otherwise.

**nominal-speed** Most trials have a nominal speed throughout the duration of the trial. This field can be used to denote that. This is primarily for reference as the actual speed can be recorded in D-Flow's record module.

**nominal-speed-units: m/s** The full name or standard unit symbol for the mass quantity.

**stationary-platform** A boolean value, [True|False], that indicates whether the treadmill motion was actuated during the trial. If this flag is false, the DFlowData class will look for compensation data, compensate for the inertial affects to the force plate data, and express the forces and moments in the motion capture reference frame.

**pitch** A boolean value, [True|False], which indicates if the pitch degree of freedom was acutated during the trial.

**sway** A boolean value, [True|False], which indicates if the lateral (sway) degree of freedom was acutated during the trial.

**marker-set** A string that indicates the HBM marker set used during the trial [full||lower|NA].

**dflow-version** This should be a string that matches the version of D-Flow used to record the trial. This is required to deal with changes in D-Flow's output from earlier versions we had.

**cortex-version** This should be a string that matches the version of Cortex used to record the trial.

**hardware-settings** There are tons of settings for the hardware in both D-Flow, Cortex, and the other software in the system. We hope to save the settings from each software with each trial, but for now this field can be used to note the most important ones.

**high-performance** A boolean value that indicates whether the D-Flow high performance setting was on (True) or off (False).

**files** This should be a key value mapping of files associated with this trial. The values should be the path to the file relative to this meta file.

**marker-map** If you want to rename the column headers for markers in the mocap module or record module's TSV files then you can specify the mapping here. For example, if the column headings in the raw data file are `M5.PosX`, `M5.PosY`, and `M5.PosZ` but you want to give the marker an easy to remember name, then the marker map `M5: T10` will set the column headers for that marker to `T10.PosX`, `T10.PosY`, and `T10.PosZ`, respectively. This only works for header names that end in `.Pos[XYZ]`.

### 1.3.2 Analog Channel Names

Since D-Flow doesn't allow you to set the names of the analog channels in the mocap module, the meta data file should include mappings, so that useful measurement names will be available for future use, for example:

```
trial:
    analog-channel-map:
        Channel1.Anlg: F1Y1
        Channel2.Anlg: F1Y2
        Channel3.Anlg: F1Y3
        Channel4.Anlg: F1X1
        Channel5.Anlg: F1X2
        Channel6.Anlg: F1Z1
        Channel7.Anlg: F2Y1
        Channel8.Anlg: F2Y2
        Channel9.Anlg: F2Y3
```

```
        Channel10.Anlg: F2X1
        Channel11.Anlg: F2X2
        Channel12.Anlg: F2Z1
        Channel13.Anlg: Front_Left_EMG
        Channel14.Anlg: Front_Left_AccX
        Channel15.Anlg: Front_Left_AccY
        Channel16.Anlg: Front_Left_AccZ
        Channel17.Anlg: Back_Left_EMG
        Channel18.Anlg: Back_Left_AccX
        Channel19.Anlg: Back_Left_AccY
        Channel20.Anlg: Back_Left_AccZ
        Channel21.Anlg: Front_Right_EMG
        Channel22.Anlg: Front_Right_AccX
        Channel23.Anlg: Front_Right_AccY
        Channel24.Anlg: Front_Right_AccZ
        Channel25.Anlg: Back_Right_EMG
        Channel26.Anlg: Back_Right_AccX
        Channel27.Anlg: Back_Right_AccY
        Channel28.Anlg: Back_Right_AccZ
```

16 accelerometers in order starting at Channel13. EMG, X, Y, Z order

### 1.3.3 Events

D-Flow doesn't allow you to define names to events and auto-names up to 6 events A-F. You can specify an event name map that will be used to automatically segment your data into more memorable names events:

```
trial:
   event:
       A: force plate zeroing begins
       B: walking begins
       C: walking with lateral perturbations begins
```

## 1.4 Usage

The `DFlowData` class is used to post process data collected from the D-Flow mocap and record modules. It does these operations:

1. Loads the meta data file into a Python dictionary if there is one.
2. Loads the mocap and record modules into Pandas `DataFrames`. [5]
3. Shifts the Delsys signals in the mocap module data to accomodate for the wireless time delay, ~96ms.
4. Identifies the missing values in the mocap marker data and replaces with NaN.
5. Returns statistics on how many missing values in the marker time series are present, the max consecutive missing values, etc.
6. Optionally, interpolates the missing marker values and replaces them with interpolated estimates.
7. Compensates the force measurments for the motion of the treadmill base.
   (a) Pulls the compensation file path from meta data.
   (b) Loads the compensation file (only the necessary columns).

[5] Only supports TSV files, we plan to add C3D support for the mocap file.

(c) Identifies the missing markers and interpolates to fill them.

(d) Shifts the Delsys signals to correct time.

(e) Filter the forces, accelerometer, and treadmill markers at 6 hz low pass.

(f) Compute the compensated forces (apply inertial compensation and express in global reference frame)

(g) Replace the force/moment measurements in the mocap data file with the compensated forces/moments.

8. Scales the analog signals to their proper units. [6]
9. Merges the data from the mocap module and record module into one `DataFrame`.
10. Optionally, low pass filter all human related data. (If there wasn't a stationary platform, then these should always be filtered with the same low pass filter as the compensation algorithm used.)
11. Extracts sections of the data based on event names.
12. Writes the cleaned and augmented data to file [7].

### 1.4.1 Python API

The `DFlowData` class gives a simple Python API for working with the D-Flow file outputs.

```
from gaitanalysis.motek import DFlowData

# Initialize the object.
data = DFlowData(mocap_tsv_path='trial_01_mocap.txt',
                 record_tsv_path='trial_01_record.txt',
                 meta_yml_path='trial_01_meta.yml')

# clean_data runs through steps 1 through 8. Many steps are optional
# depending on the optional keyword arguments.
data.clean_data()

# The following command returns a Pandas DataFrame of all the measurements
# for the time period matching the event.
perturbed_walking = data.extract_processed_data(event='walking with perturbation')

# The class in includes writers to write the manipulated data to file, in
# this case a D-Flow compatible text file.
data.write_dflow_tsv('trial_01_clean.txt')
```

### 1.4.2 Command Line

The following command will load the three input files, clean up the data, and write the results to file, which can be loaded back into D-Flow or used in some other application.

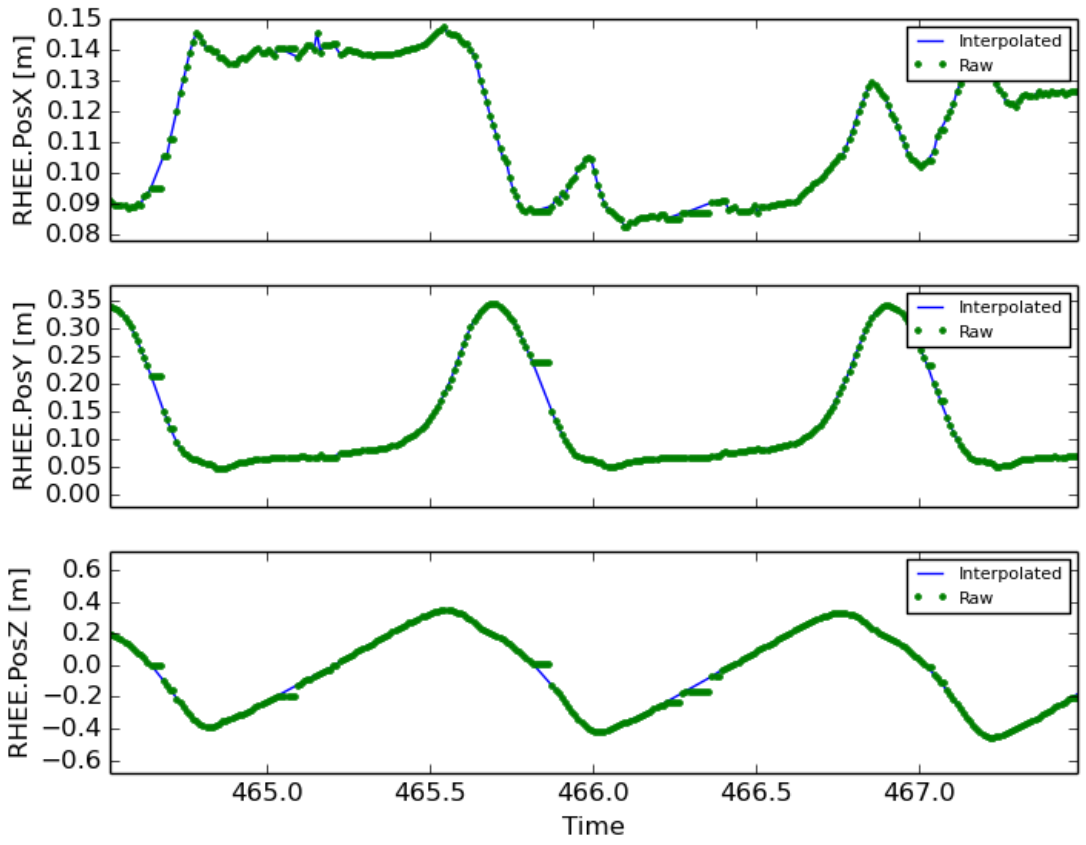
```
dflowdata -m trial_01_mocap.txt -r trial_01_record.txt -y trial_01_meta.yml trial_01_clean.txt
```

### 1.4.3 Examples

This shows how to compare the raw marker data with the new interpolated data, in this case a simple linear interpolation.

[6] Not implemented yet, scaling factors should be stored in meta data?.
[7] Only outputs to tsv.

```
import pandas
import maplotlib.pyplot as plt

data = DFlowData('mocap-module-01.txt', 'record-module-01.txt')
data.clean_data()

unclean = pandas.read_csv('mocap-module-01.txt', delimiter='\t')

fig, axes = plt.subplots(3, 1, sharex=True)

for i, label in enumerate(['RHEE.PosX', 'RHEE.PosY', 'RHEE.PosZ']):

    axes[i].plot(data.data['TimeStamp'], data.data[label],
                 unclean['TimeStamp'], unclean[label], '.')

    axes[i].set_ylabel(label + ' [m]')

    axes[i].legend(['Interpolated', 'Raw'], fontsize=8)

axes[2].set_xlabel('Time')

fig.show()
```



CHAPTER 2

# Gait Module

The gaitanalysis.gait module provides tools to process and analyze with typical data collected during the measurement of human locomotion (gait). In general, the three dimensional coordinates throughout time of a set of markers which are attached to anatomical features on the human are tracked. Secondly, various analog signals are recorded. In particular, voltages which are proportional to the applied forces and moments on one or two force plates, voltages from electromyography (EMG) measurements, and/or accelerometers, etc. All of these measurements are stored as discrete samples in time.

CHAPTER 3

# gaitanalysis Package

The following documentation provides both the pubic and private API for the modules included in the gaitanalysis package. We will try our best to follow semantic versioning with respect to the public API. The private member functions, starts with _, are subject to change during development and will not be held to semantic versioning. Use at your own risk.

## 3.1 motek Module

**class** gaitanalysis.motek.**DFlowData**(*mocap_tsv_path=None*, *record_tsv_path=None*, *meta_yml_path=None*)

Bases: object

A class to store and manipulate the data outputs from Motek Medical's D-Flow software.

**_analog_column_labels**(*labels*)

Returns a list of analog channel column labels and the indices of the labels.

**Parameters** **labels** : list of strings

This should be a superset of column labels, some of which may be human body model results, and should include the default analog channel labels output from the D-Flow mocap model, i.e. "Channel[1-99].Anlg".

**Returns** **analog_labels** : list of strings

The labels of analog channels in the order found in *labels*.

**analog_indices** : list of integers

The indices of the analog columns with respect to the indices of *labels*.

**emg_column_labels** : list of strings

The labels of emg channels in the order found in *labels*

**accel_column_labels** : list of strings

The labels of accelerometer channels in the order found in *labels*

**_c = 'Z'**

**_calibrate_accel_data**(*data_frame*, *y1=0*, *y2=-9.81*)

Two-point calibration of accelerometer signals. Converts from voltage to meters/second^2

**Parameters** **data_frame** : pandas.DataFrame

Accelerometer data in volts to be calibrated

**y1** : float, optional

**y2** : float, optional

**Returns** **data_frame** : pandas.DataFrame

Calibrated accelerometer data in m/s^2

#### Notes

A calibration file must be specified in the meta file and its structure is as follows: There must be a column for each accelerometer signal to be calibrated, so three columns per sensor. There must be three rows of accelerometer readings. The first row is the reading when the sensors are placed with z-axis pointing straight up. The second row is the reading when the x-axis is pointing straight up. The third row is the reading when the y-axis is pointing straight up. (xyz) —– (001) (100) (010)

**_clean_compensation_data**(*data_frame*)

Returns a the data frame with Delsys signals shifted and all signals low pass filtered.

**Parameters** **data_frame** : pandas.DataFrame

This data frame should contain only the columns needed for the compensation calculations: accelerometers and forces/moments.

**Returns** **data_frame** : pandas.DataFrame

The cleaned compensation data.

**_compensate**(*mocap_data_frame*, *markers_missing=False*)

Returns the data frame with the forces compensated.

**Parameters** **mocap_data_frame** : pandas.DataFrame

A data frame that contains the force plate forces and moments to be compensated, along with the measurements of the markers and accelerometers that were attached to the treadmill.

**markers_missing** : pandas.DataFrame

If the treadmill markers have missing markers, this should be true so that they are fixed before the compensation.

**Returns** mocap

#### Notes

This method does the following:

1. Pulls the compensation file path from meta data.
2. Loads the compensation file (only the necessary columns).
3. Identifies the missing markers in the compensation file and interpolates to fill them.
4. Shifts the Delsys signals to correct time.
5. Filter the forces, accelerometer, and treadmill markers at 6 hz low pass.
6. **Compute the compensated forces (apply inertial compensation and express** in global reference frame)
7. **Replace the force/moment measurements in the mocap data file with the** compensated forces/moments.

**_compensate_forces**(*calibration_data_frame*, *data_frame*)
Computes the forces and moments which are due to the lateral and pitching motions of the treadmill and subtracts them from the measured forces and moments based on linear acceleration measurements of the treadmill.

**_compensation_needed**()
Returns true if the meta data includes:

'trial: stationary-platform: False'

**_end = 'Z'**

**_express**(*data_frame*, *rotation_matrix*)
Returns a new data frame in which the marker, force, moment, and center of pressure vectors are expressed in a different reference frame using the provided rotation matrix.

**Parameters** **data_frame** : pandas.DataFrame

A data frame which contains columns for the DFlow marker, force, moment, and center of pressure outputs.

**rotation_matrix** : array_like, shape(3, 3)

A rotation matrix which will be premultiplied to all vectors in the provided data frame.

**Returns** **rotated_data_frame** : pandas.DataFrame

A copy of the provided data frame in which all vectors are expressed in a new reference frame.

#### Notes

If v_df is a vector expressed in data frame's original coordinate system and v_new is the same vector expressed in the desired coordinate system, then:

v_new = rotation_matrix * v_df

This function does nothing with the segment columns. As it stands it is not clear what the *.Rot[XYZ]* columns are so all segment columns are left alone.

This function also does not currently deal with any human body model columns, e.g. the center of mass.

**_express_in_isb_standard_coordinates**(*data_frame*)
Returns a new data frame in which the marker, force, moment, and center of pressure vectors are expressed in the the ISB standard coordinate system given in [Wu1995]. This referene frame has the X axis aligned in the directin of travel, the Y axis vertical and opposite of gravity, and the Z axis to the right, following the right hand rule.

**Parameters** **data_frame** : pandas.DataFrame

A data frame which contains columns for the DFlow marker, force, moment, and center of pressure outputs. The coordinate system for the vectors must be the Cortex default system (X to the right, Y up, Z backwards).

**Returns** **rotated_data_frame** : pandas.DataFrame

A copy of the provided data frame in which all vectors are expressed in the ISB standard coordinate system.

#### Notes

See *DflowData._express* for more details.

#### References

[Wu1995]

**_extract_events_from_record_file**()
Returns a dictionary of events and times. The event names will be the default A-F which is output by D-Flow unless you specify unique names in the meta data file. If there are no events in the record file, this will return nothing.

**_force_column_labels**(*without_center_of_pressure=False*)
Returns a list of force column labels.

**Parameters** **without_center_of_pressure: boolean, optional, default=False**

If true, the center of pressure labels will not be included in the list.

**Returns** **labels** : list of strings

A list of the force plate related signals.

**_generate_cortex_time_stamp**(*data_frame*)
Returns the data frame with a new index based on the constant sample rate from Cortex.

**_hbm_column_labels**(*labels*)
Returns a list of human body model column labels, the indices of the labels, and the indices of the non-hbm labels in relation to the rest of the header.

**Parameters** **labels** : list of strings

This should be a superset of column labels, some of which may be human body model results.

**Returns** **hbm_labels** : list of strings

The labels of columns of HBM data time series in the order found in *labels*.

**hbm_indices** : list of integers

The indices of the HBM columns with respect to the indices of *labels*.

**non_hbm_indices** : list of integers

The indices of the non-HBM columns with respect to the indices of *labels*.

**static _header_labels**(*path_to_file*, *delimiter='t'*)
Returns a list of labels from the header, i.e. the first line of a delimited text file.

**Parameters** **path_to_file** : string

Path to the delimited text file with a header on the first line.

**delimiter** : string, optional, default=' '

The delimiter used in the file.

**Returns** **header_labels** : list of strings

A list of the headers in order as included from the file.

**_identify_missing_markers**(*data_frame*)
Returns the data frame in which all marker columns have had constant marker values replaced with NaN.

**Parameters** **data_frame** : pandas.DataFrame, size(n, m)

A data frame which contains columns of marker position time histories. The marker time histories may contain periods of constant values.

**Returns** **data_frame** : pandas.DataFrame, size(n, m)

The same data frame which was supplied expect that constant values in the marker columns have been replaced with NaN.

#### Notes

D-Flow replaces missing marker values with the last available measurement in time. This method is used to properly replace them with a unique idnetifier, NaN. If two adjacent measurements in time were actually the same value, then this method will replace the subsequent ones with NaNs, and is not correct, but the likelihood of this happening is low.

**_load_compensation_data**()
Returns a data frame which includes the treadmill forces/moments, and the accelerometer signals as time series with respect to the D-Flow time stamp.

**_load_mocap_data**(*ignore_hbm=False*, *id_na=False*)
Returns a data frame generated from the mocap TSV file.

**Parameters** **ignore_hbm** : boolean, optional, default=False

If true, the columns associated with D-Flow's real time human body model computations will not be loaded.

**id_na** : boolean, optional, default=False

If true the marker and/or HBM columns will be loaded with all '0.000000' and '-0.000000' strings in the HBM columns replaced with numpy.NaN. This is dependent on the D-Flow version as, versions <= 3.16.1 stored missing markers as the previous valid value.

**Returns** **data_frame** : pandas.DataFrame

**_load_record_data**()
Returns a data frame containing the data from the record module.

**_marker_column_labels**(*labels*)
Returns a list of column labels that correpsond to markers, i.e. ones that end in '.PosX', '.PosY', or '.PosZ', given a master list.

**Parameters** **labels** : list of strings

This should be a superset of column labels, some of which may be marker column labels.

**Returns** **marker_labels** : list of strings

The labels of columns of marker time series in the order found in *labels*.

**_merge_mocap_record**()
Returns a data frame that is a merger of the mocap and record data, if needed.

**_missing_markers_are_zeros**()
Returns True if the mocap module marker missing values are represented as strings of zeros, '-0.000000' and '0.000000' and False if missing markers are represented as the previous valid value. This depends on the D-Flow version being present in the trial meta data. If it isn't then it is assumed that missing markers are represented as strings of zeros, i.e. latest D-Flow behavior.

**_mocap_column_labels**()
Returns a list of strings containing the motion capture file's column labels. The list is in the same order as in the mocap tsv file.

**_orient_accelerometers**(*data_frame*)

**Parameters** **mocap_data_frame** : pandas.DataFrame

DataFrame containing accelerometer signals to be placed in treadmill reference frame.

**Returns** **mocap_data_frame** : pandas.DataFrame

DataFrame containing accelerometer signals in treadmill reference frame.

**_parse_meta_data_file**()
Returns a dictionary containing the meta data stored in the optional meta data file.

**_relabel_analog_columns**(*data_frame*)
Relabels analog channels in data frame to names defined in the yml meta file. Channels not specified in the meta file are keep their original names. self.analog_column_labels, self.emg_column_labels, and self.accel_column_labels are updated with the new names.

**Parameters** **data_frame** : pandas.DataFrame, size(n, m)

**Returns** **data_frame** : pandas.DataFrame, size(n, m

The same data frame with columns relabeled.

**_relabel_markers**(*data_frame*)
Returns the data frame with the columns renamed to reflect the provided mapping and updates the marker column and mocap column label attributes to reflect the new names. If there is no marker map in the meta data the data frame is returned unmodified.

**Parameters** **data_frame** : pandas.DataFrame

A data frame with column names which match the keys in the meta data:trial:marker-map.

**_resample_record_data**(*data_frame*)
Resamples the raw data from the record file at the sample rate of the mocap file.

**_segment = 'rtoes'**

**_shift_delsys_signals**(*data_frame*, *time_col='TimeStamp'*)
Returns a data frame in which the Delsys columns are linearly interpolated (and extrapolated) at the time they were actually measured.

**_store_compensation_data_path**()
Stores the path to the compensation data file.

#### Notes

The meta data yaml file must include a relative file path to a mocap file that contains time series data appropriate for computing the force inertial and rotational compensations. The yaml declaration should look like this example:

**files:** mocap: mocap-378.txt record: record-378.txt meta: meta-378.yml compensation: ../path/to/mocap/file.txt

**_suffix = '.RotZ'**

**_suffix_beg = '.Cop'**

**analog_channel_regex = '^Channel[0-9]+\\.Anlg$'**

**c = 'Z'**

**clean_data**(*ignore_hbm=False*, *id_na=True*, *interpolate=True*, *interpolation_order=1*)

Returns the processed, "cleaned", data.

**Parameters** **ignore_hbm** : boolean, optional, default=False

HBM columns will not be loaded from the mocap model data TSV file. This can save some load time if you are not using that data.

**id_na** : boolean, optional, default=True

Identifies any missing values in the marker and/or HBM data and replaces the with np.NaN.

**interpolate** : boolean, optional, default=True

If true, the missing values in the markers and/or HBM columns will be interpolated. Note that if force compensation is needed, the markers on the treadmill will always have the missing values interpolated regardless of this flag. This argument is ignored if `id_na` is False, as there are no identified missing marker values available to interpolate over.

**interpolation_order** : integer, optional, default=1

The spline interpolation order (between 1 and 5). See scipy.interpolate.InterpolatedUnivariateSpline.

#### Notes

1.Loads the mocap and record modules into Pandas `DataFrame`s.

2.Relabels the columns headers to more meaningful names if this is specified in the meta data.

3.Shifts the Delsys signals in the mocap module data to accomodate for the wireless time delay. The value of the delay is stored in `DflowData.delsys_time_delay`.

4.Identifies the missing values in the mocap marker data and replaces them with `numpy.nan`.

5.Optionally, interpolates the missing marker and HBM values and replaces them with interpolated estimates.

6.Compensates the force measurments for the motion of the treadmill base, if needed.

(a)Pulls the compensation mocap file path from meta data.

(b)Loads the compensation mocap file (only the necessary columns).

(c)Identifies the missing markers and interpolates to fill them.

(d)Shifts the Delsys signals to correct time.

(e)Filter the forces, accelerometer, and treadmill markers with a 6 hz low pass 2nd order Butterworth filter.

(f)Computes the compensated forces by subtracting the inertial forces and expressing the forces in the camera reference frame.

(g)Replaces the force/moment measurements in the mocap data file with the compensated forces/moments.

8.Merges the data from the mocap module and record module into one data frame.

**constant_marker_tolerance = 1e-16**

**cortex_sample_rate = 100**

**delsys_time_delay = 0.096**

**dflow_segments = ['pelvis', 'thorax', 'spine', 'pelvislegs', 'lfemur', 'ltibia', 'lfoot', 'toes', 'rfemur', 'rtibia', 'rfoot', 'rt**

**extract_processed_data**(*event=None*, *index_col=None*, *isb_coordinates=False*)

Returns the processed data in a data frame. If an event name is provided, then a data frame with only that event is returned.

**Parameters** **event** : string, optional, default=None

A name of a detected event. Must be a valid key in self.events. This will be either the D-Flow auto-named events (A, B, C, D, E, F) or the names specified in the meta data file.

**index_col** : string, optional, default=None

A name of a column in the data frame. If provided the the column will be removed from the data frame and used as the index. This is useful for assigning one of the time columns as the index.

**isb_coordinates** : boolean, optional, default=False

If True, the marker, force, moment, and center of pressure vectors will be expressed in the ISB standard coordinate system instead of the Cortex default coordinate system.

**Returns** **data_frame** : pandas.DataFrame

The processed data.

**force_plate_names = ['FP1', 'FP2']**

**force_plate_regex = '^FP[12]\\.[For|Mom|Cop][XYZ]$'**

**force_plate_suffix = ['.ForX', '.MomX', '.CopX', '.ForY', '.MomY', '.CopY', '.ForZ', '.MomZ', '.CopZ']**

**hbm_column_regexes = ['^\\s?[LR]_.*', '.*\\.Mom$', '.*\\.Ang$', '.*\\.Pow$', '.*\\.COM.[XYZ]$']**

**hbm_na = ['0.000000', '-0.000000']**

**low_pass_cutoff = 6.0**

**marker_coordinate_regex = '.*\\.Pos[XYZ]$'**

**marker_coordinate_suffixes = ['.PosX', '.PosY', '.PosZ']**

**missing_value_statistics**(*data_frame*)

Returns a report of missing values in the data frame.

**rotation_suffixes = ['.RotX', '.RotY', '.RotZ']**

**segment_labels = ['pelvis.PosX', 'pelvis.PosY', 'pelvis.PosZ', 'pelvis.RotX', 'pelvis.RotY', 'pelvis.RotZ', 'thorax.PosX**

**treadmill_markers = ['ROT_REF.PosX', 'ROT_REF.PosY', 'ROT_REF.PosZ', 'ROT_C1.PosX', 'ROT_C1.PosY', 'R**

**write_dflow_tsv**(*filename*, *na_rep='NA'*)

**class** gaitanalysis.motek.**MissingMarkerIdentifier**(*data_frame*)

Bases: object

**_c = 'Z'**

**constant_marker_tolerance = 1e-16**

**identify**(*columns=None*)

Returns the data frame in which all or the specified columns have had constant values replaced with NaN.

**Returns** **data_frame** : pandas.DataFrame, size(n, m)

The same data frame which was supplied with constant values replaced with NaN.

**columns** : list of strings, optional, default=None

The specific list of columns in the data frame that should be analyzed. This is typically a list of all marker columns.

#### Notes

D-Flow replaces missing marker values with the last available measurement in time. This method is used to properly replace them with a unique identifier, NaN. If two adjacent measurements in time were actually the same value, then this method will replace the subsequent ones with NaNs, and is not correct, but the likelihood of this happening is low.

**marker_coordinate_suffixes = ['.PosX', '.PosY', '.PosZ']**

**statistics**()

Returns a data frame containing the number of missing samples and maximum number of consecutive missing samples for each column.

gaitanalysis.motek.**low_pass_filter**(*data_frame*, *columns*, *cutoff*, *sample_rate*, ***kwargs*)

Returns the data frame with indicated columns filtered with a low pass second order forward/backward Butterworth filter.

**Parameters** **data_frame** : pandas.DataFrame

A data frame with time series columns.

**columns** : sequence of strings

The columns that should be filtered.

**cutoff** : float

The low pass cutoff frequency in Hz.

**sample_rate** : float

The sample rate of the time series in Hz.

**kwargs** : key value pairs

Any addition keyword arguments to pass to dtk.process.butterworth.

**Returns** **data_frame** : pandas.DataFrame

The same data frame which was passed in with the specified columns replaced by filtered versions.

gaitanalysis.motek.**markers_for_2D_inverse_dynamics**(*marker_set='lower'*)

Returns lists of markers from the D-Flow human body model marker protocol(lower or full), that should be used with leg2d.m.

**Parameters** **marker_set** : string, optional, default='lower'

Specify either 'lower' or 'full' depending on which marker set you used.

**Returns** **left_marker_coords** : list of strings, len(12)

The X and Y coordinates for the 6 left markers.

**right_marker_coords** : list of strings, len(12)

The X and Y coordinates for the 6 right markers.

**left_forces** : list of strings, len(3)

The X and Y ground reaction forces and the Z ground reaction moment of the left leg.

**right_forces** : list of strings, len(3)

The X and Y ground reaction forces and the Z ground reaction moment of the left leg.

#### Notes

The returned marker labels correspond to the ISB standard coordinate system for gait, with X in the direction of travel, Y opposite to gravity, and Z to the subject's right.

D-Flow/Cortex output data (and marker labels) in a different coordinate system and follow this conversion:

- The D-Flow X unit vector is equal to the ISB Z unit vector.
- The D-Flow Y unit vector is equal to the ISB Y unit vector.
- The D-FLow Z unit vector is equal to the ISB -X unit vector.

So it is up to the user to ensure that the marker and force data that corresponds to the returned labels is expressed in the ISB coordinate frame before passing it into `leg2d.m`. `DFlowData` has methods that can express the data in the correct coordinate system on output.

The forces and moments must also be normalized by body mass before using with `leg2d.m`.

`gaitanalysis.motek.`**`spline_interpolate_over_missing`**(*data_frame*, *abscissa_column*, *order=1*, *columns=None*)

Returns the data frame with all missing values replaced by some interpolated or extrapolated values derived from a spline.

**Parameters** **data_frame** : pandas.DataFrame

A data frame which contains a column for the abscissa and other columns which may or may not have missing values, i.e. NaN.

**abscissa_column** : string

The column name which represents the abscissa.

**order** : integer, optional, default=1

The order of the spline. Can be 1 through 5 for linear through quintic splines. The default is a linear spline. See documentation for scipy.interpolate.InterpolatedUnivariateSpline.

**columns** : list of strings, optional, default=None

If only a particular set of columns need interpolation, they can be specified here.

**Returns** **data_frame** : pandas.DataFrame

The same data frame passed in with all NaNs in the specified columns replaced with interpolated or extrapolated values.

## 3.2 gait Module

**class** gaitanalysis.gait.**GaitData**(*data*)
Bases: object

A class to store typical gait data.

**attrs_to_store = ['data', 'gait_cycles', 'gait_cycle_stats', 'strikes', 'offs']**

**grf_landmarks**(*right_vertical_signal_col_name*, *left_vertical_signal_col_name*, *method='force'*, *do_plot=False*, *min_time=None*, *max_time=None*, ***kwargs*)
Returns the times at which heel strikes and toe offs happen in the raw data.

**Parameters** **right_vertical_signal_col_name** : string

The name of the column in the raw data frame which corresponds to the right foot vertical ground reaction force.

**left_vertical_signal_col_name** : string

The name of the column in the raw data frame which corresponds to the left foot vertical ground reaction force.

**method: string {force|accel}**

Whether to use force plate data or accelerometer data to calculate landmarks

**Returns** **right_strikes** : np.array

All indices at which right_grfy is non-zero and it was 0 at the preceding time index.

**left_strikes** : np.array

Same as above, but for the left foot.

**right_offs** : np.array

All indices at which left_grfy is 0 and it was non-zero at the preceding time index.

**left_offs** : np.array

Same as above, but for the left foot.

#### Notes

This is a simple wrapper to gait_landmarks_from_grf and supports all the optional keyword arguments that it does.

**inverse_dynamics_2d**(*left_leg_markers*, *right_leg_markers*, *left_leg_forces*, *right_leg_forces*, *body_mass*, *low_pass_cutoff*)
Computes the hip, knee, and ankle angles, angular rates, joint moments, and joint forces and adds them as columns to the data frame.

**Parameters** **left_leg_markers** : list of strings, len(12)

The names of the columns that give the X and Y marker coordinates for six markers.

**right_leg_markers** : list of strings, len(12)

The names of the columns that give the X and Y marker coordinates for six markers.

**left_leg_forces** : list of strings, len(3)

The names of the columns of the ground reaction forces and moments (Fx, Fy, Mz).

**right_leg_forces** : list of strings, len(3)

The names of the columns of the ground reaction forces and moments (Fx, Fy, Mz).

**body_mass** : float

The mass, in kilograms, of the subject.

**low_pass_cutoff** : float

The cutoff frequency in hertz.

**Returns** **data_frame** : pandas.DataFrame

The main data frame now with columns for the new variables. Note that the force coordinates labels (X, Y) are relative to the coordinate system described herein.

#### Notes

This computation assumes the following coordinate system:

```
Y
 ^ _ o _
 |   |   ---> v
 |  /              -----> x
```

where X is forward (direction of walking) and Y is up.

Make sure the sign conventions of the columns you pass in are correct!

**The markers should be in the following order:**

1. Shoulder
2. Greater trochanter
3. Lateral epicondyle of knee
4. Lateral malleolus
5. Heel (placed at same height as marker 6)
6. Head of 5th metatarsal

The underlying function low pass filters the data before computing the inverse dynamics. You should pass in unfiltered data.

**load**(*filename*)
Loads data from disk via HDF5 (PyTables).

**Parameters** **filename** : string

Path to an HDF5 file.

**plot_gait_cycles**(**col_names*, ***kwargs*)
Plots the time histories of each gait cycle.

**Parameters** **col_names** : string

A variable number of strings naming the columns to plot.

**mean** : boolean, optional

If true the mean and standard deviation of the cycles will be plotted.

**kwargs** : key value pairs

Any extra kwargs to pass to the matplotlib plot command.

**plot_landmarks**(*col_names*, *side*, *event='both'*, *index=0*, *window=None*, *num_cycles_to_plot=None*, *curve_kwargs=None*, *heel_kwargs=None*, *toe_kwargs=None*)

Creates a plot of the desired signal(s) with the gait event times overlaid on top of the signal.

**Parameters** **col_names** : sequence of strings

A variable number of strings naming the columns to plot.

**side** : string, {right|left}

Whether to plot the gait landmarks from the right or left leg.

**event** : string, {heelstrikes|toeoffs|both|none}

Which gait landmarks to plot.

**index** : integer, optional, default=0

The index of the first time sample in the plot. This is useful if you want to plot the cycles starting at an arbitrary point in time in the data.

**window** : integer, optional, default=None

The number of time samples to plot. This is useful when a trial has many cycles and you only want to view some of them in the plot.

**num_cycles_to_plot** : integer, optional, default=None

This is an alternative way to specify the window. If this is provided, the window argment is ignored and the window is estimated by the desired number of cycles.

**curve_kwargs** : dictionary, optional

Valid matplotlib kwargs that will be used for the signal curves.

**heel_kwargs** : dictionary, optional

Valid matplotlib kwargs that will be used for the heel-strike lines.

**toe_kwargs** : dictionary, optional

Valid matplotlib kwargs that will be used for the toe-off lines.

**Returns** **axes** : matplotlib.Axes

The list of axes for the subplots or a single axes if only one column was supplied. Same as *matplotlib.pyplot.subplots* returns.

#### Notes

The *index*, *window* and *num_cycles_to_plot* arguments do not simply set the x limit to bound the data of interest, they do not plot any data outside the desired range (and is thus faster).

**save**(*filename*)

Saves data to disk via HDF5 (PyTables).

**Parameters** **filename** : string

Path to an HDF5 file.

**split_at**(*side*, *section='both'*, *num_samples=None*, *belt_speed_column=None*)

Forms a pandas.Panel which has an item for each cycle. The index of each cycle data frame will be a percentage of gait cycle.

**Parameters** **side** : string {right|left}

Split with respect to the right or left side heel strikes and/or toe-offs.

**section** : string {both|stance|swing}

Whether to split around the stance phase, swing phase, or both.

**num_samples** : integer, optional

If provided, the time series in each gait cycle will be interpolated at values evenly spaced at num_sample in time across the gait cycle. If None, the maximum number of possible samples per gait cycle will be used.

**belt_speed_column** : string, optional

The column name corresponding to the belt speed on the corresponding side.

**Returns** **gait_cycles** : pandas.Panel

A panel where each item is a gait cycle. Each cycle has the same number of time samples and the index is set to the percent of the gait cycle.

**time_derivative**(*col_names*, *new_col_names=None*)

Numerically differentiates the specified columns with respect to the time index and adds the new columns to *self.data*.

**Parameters** **col_names** : list of strings

The column names for the time series which should be numerically time differentiated.

**new_col_names** : list of strings, optional

The desired new column name(s) for the time differentiated series. If None, then a default name of *Time derivative of <origin column name>* will be used.

**tpose**(*data_frame*)

Computes the mass of the subject. Computes to orientation of accelerometers on a subject during quiet standing relative to treadmill Y-axis

gaitanalysis.gait.**find_constant_speed**(*time*, *speed*, *plot=False*, *filter_cutoff=1.0*)

Returns the indice at which the treadmill speed becomes constant and the time series when the treadmill speed is constant.

**Parameters** **time** : array_like, shape(n,)

A monotonically increasing array.

**speed** : array_like, shape(n,)

A speed array, one sample for each time. Should ramp up and then stablize at a speed.

**plot** : boolean, optional

If true a plot will be displayed with the results.

**filter_cutoff** : float, optional

The filter cutoff frequency for filtering the speed in Hertz.

**Returns** **indice** : integer

The indice at which the speed is consider constant thereafter.

**new_time** : ndarray, shape(n-indice,)

The new time array for the constant speed section.

gaitanalysis.gait.**gait_landmarks_from_accel**(*time*, *right_accel*, *left_accel*, *threshold=0.33*, ***kwargs*)

Obtain right and left foot strikes from the time series data of accelerometers placed on the heel.

**Parameters** **time** : array_like, shape(n,)

A monotonically increasing time array.

**right_accel** : array_like, shape(n,)

The vertical component of accel data for the right foot.

**left_accel** : str, shape(n,)

Same as above, but for the left foot.

**threshold** : float, between 0 and 1

Increase if heelstrikes/toe-offs are falsly detected

**Returns** **right_foot_strikes** : np.array

All times at which a right foot heelstrike is determined

**left_foot_strikes** : np.array

Same as above, but for the left foot.

**right_toe_offs** : np.array

All times at which a right foot toeoff is determined

**left_toe_offs** : np.array

Same as above, but for the left foot.

gaitanalysis.gait.**gait_landmarks_from_grf**(*time*, *right_grf*, *left_grf*, *threshold=1e-05*, *filter_frequency=None*, ***kwargs*)

Obtain gait landmarks (right and left foot strike & toe-off) from ground reaction force (GRF) time series data.

**Parameters** **time** : array_like, shape(n,)

A monotonically increasing time array.

**right_grf** : array_like, shape(n,)

The vertical component of GRF data for the right leg.

**left_grf** : str, shape(n,)

Same as above, but for the left leg.

**threshold** : float, optional

Below this value, the force is considered to be zero (and the corresponding foot is not touching the ground).

**filter_frequency** : float, optional, default=None

If a filter frequency is provided, in Hz, the right and left ground reaction forces will be filtered with a 2nd order low pass filter before the landmarks are identified. This method assumes that there is a constant (or close to constant) sample rate.

**Returns** **right_foot_strikes** : np.array

All times at which right_grfy is non-zero and it was 0 at the preceding time index.

**left_foot_strikes** : np.array

Same as above, but for the left foot.

**right_toe_offs** : np.array

All times at which left_grfy is 0 and it was non-zero at the preceding time index.

**left_toe_offs** : np.array

Same as above, but for the left foot.

#### Notes

Source modifed from:

https://github.com/fitze/epimysium/blob/master/epimysium/postprocessing.py

gaitanalysis.gait.**interpolate**(*data_frame*, *time*)

Returns a data frame with a index based on the provided time array and linear interpolation.

**Parameters** **data_frame** : pandas.DataFrame

A data frame with time series columns. The index should be in same units as the provided time array.

**time** : array_like, shape(n,)

A monotonically increasing array of time in seconds at which the data frame should be interpolated at.

**Returns** **interpolated_data_frame** : pandas.DataFrame

The data frame with an index matching *time_vector* and interpolated values based on *data_frame*.

gaitanalysis.gait.**plot_gait_cycles**(*gait_cycles*, **col_names*, ***kwargs*)

Plots the time histories from each gait cycle on one graph.

**Parameters** **gait_cycles** : pandas.Panel

A panel of gait cycles. Each item should be a cycle DataFrame with time histories of variables. The index should be the percent gait cycle.

**col_names** : string

A variable number of strings naming the columns to plot.

**mean** : boolean, optional, default=False

If true the mean and standard deviation of the gait cycles will be plotted instead of the individual lines.

**kwargs** : key value pairs

Any extra kwargs to pass to the matplotlib plot command.

## 3.3 controlid Module

**class** gaitanalysis.controlid.**SimpleControlSolver**(*data*, *sensors*, *controls*, *validation_data=None*)

Bases: object

This assumes a simple linear control structure at each time instance in a gait cycle.

The measured joint torques equal some limit cycle joint torque plus a matrix of gains multiplied by the error in the sensors and the nominal value of the sensors.

m_measured(t) = m_nominal + K(t) [s_nominal(t) - s(t)] = m*(t) - K(t) s(t)

This class solves for the time dependent gains and the "commanded" controls using a simple linear least squares.

**compute_estimated_controls**(*gain_matrices*, *nominal_controls*)

Returns the predicted values of the controls and the contributions to the controls given gains, K(t), and nominal controls, m*(t), for each point in the gait cycle.

**Parameters** **gain_matrices** : ndarray, shape(n, q, p)

The estimated gain matrices for each time step.

**control_vectors** : ndarray, shape(n, q)

The nominal control vector plus the gains multiplied by the reference sensors at each time step.

**Returns** **panel** : pandas.Panel, shape(m, n, q)

There is one data frame to correspond to each gait cycle in self.validation_data. Each data frame has columns of time series which store m(t), m*(t), and the individual components due to K(t) * se(t).

#### Notes

m(t) = m0(t) + K(t) * [ s0(t) - s(t) ] = m0(t) + K(t) * se(t) m(t) = m*(t) - K(t) * s(t)

This function returns m(t), m0(t), m*(t) for each control and K(t) * [s0(t) - s(t)] for each sensor affecting each control. Where s0(t) is estimated by taking the mean with respect to the gait cycles.

**controls**

**deconstruct_solution**(*x*, *covariance*)

Returns the gain matrices, K(t), and m*(t) for each time step in the gait cycle given the solution vector and the covariance matrix of the solution.

m(t) = m*(t) - K(t) s(t)

**Parameters** **x** : array_like, shape(n * q * (p + 1),)

The solution matrix containing the gains and the commanded controls.

**covariance** : array_like, shape(n * q * (p + 1), n * q * (p + 1))

The covariance of x with respect to the variance in the fit.

**Returns** **gain_matrices** : ndarray, shape(n, q, p)

The gain matrices at each time step, K(t).

**control_vectors** : ndarray, shape(n, q)

The nominal control vector plus the gains multiplied by the reference sensors at each time step.

**gain_matrices_variance** : ndarray, shape(n, q, p)

The variance of the found gains (covariance is neglected).

**control_vectors_variance** : ndarray, shape(n, q)

The variance of the found commanded controls (covariance is neglected).

#### Notes

**x looks like:**

**[k11(0), k12(0), ..., kqp(0), m1*(0), ..., mq*(0), ...,** k11(n), k12(0), ..., kqp(n), m1*(n), ..., mq*(n)]

If there is a gain omission matrix then nan's are substituted for all gains that were set to zero.

**form_a_b**()

Returns the A matrix and the b vector for the linear least squares fit.

**Returns** **A** : ndarray, shape(n * q, n * q * (p + 1))

The A matrix which is sparse and contains the sensor measurements and ones.

**b** : ndarray, shape(n * q,)

The b vector which constaints the measured controls.

#### Notes

In the simplest fashion, you can put:

```
m(t) = m*(t) - K * s(t)
```

into the form:

```
Ax = b
```

with:

```
b = m(t)
A = [-s(t) 1]
x = [K(t) m*(t)]^T

[-s(t) 1] * [K(t) m*(t)]^T = m(t)
```

**form_control_vectors**()

Returns an array of control vectors for each cycle and each time step in the identification data.

**Returns** **control_vectors** : ndarray, shape(m, n, q)

The sensor vector form the i'th cycle and the j'th time step will look like [control_0, ..., control_(q-1)].

**form_sensor_vectors**()

Returns an array of sensor vectors for each cycle and each time step in the identification data.

**Returns** **sensor_vectors** : ndarray, shape(m, n, p)

The sensor vector form the i'th cycle and the j'th time step will look like [sensor_0, ..., sensor_(p-1)].

**gain_inclusion_matrix**

**identification_data**

**least_squares**(*A*, *b*, *ignore_cov=False*)

Returns the solution to the linear least squares and the covariance matrix of the solution.

**Parameters** **A** : array_like, shape(n, m)

The coefficient matrix of Ax = b.

**b** : array_like, shape(n,)

The right hand side of Ax = b.

**ignore_cov: boolean, optional, default=False**

The covariance computation for a very large A matrix can be extremely slow. If this is set to True, then the computation is skipped and the covariance of the identified parameters is set to zero.

**Returns** **x** : ndarray, shape(m,)

The best fit solution.

**variance** : float

The variance of the fit.

**covariance** : ndarray, shape(m, m)

The covariance of the solution.

**plot_control_contributions**(*estimated_panel*, *max_num_gait_cycles=4*)

Plots two graphs for each control and each gait cycle showing contributions from the linear portions. The first set of graphs shows the first few gait cycles and the contributions to the control moments. The second set of graph shows the mean contributions to the control moment over all gait cycles.

**Parameters** **panel** : pandas.Panel, shape(m, n, q)

There is one data frame to correspond to each gait cycle. Each data frame has columns of time series which store m(t), m*(t), and the individual components due to K(t) * se(t).

**plot_estimated_vs_measure_controls**(*estimated_panel*, *variance*)

Plots a figure for each control where the measured control is shown compared to the estimated along with a plot of the error.

**Parameters** **estimated_panel** : pandas.Panel

A panel where each item is a gait cycle.

**variance** : float

The variance of the fit.

**Returns** **axes** : array of matplotlib.axes.Axes, shape(q,)

The plot axes.

**plot_gains**(*gains*, *gain_variance*, *y_scale_function=None*)

Plots the identified gains versus percentage of the gait cycle.

**Parameters** **gain_matrix** : ndarray, shape(n, q, p)

The estimated gain matrices for each time step.

**gain_variance** : ndarray, shape(n, q, p)

The variance of the estimated gain matrices for each time step.

**y_scale_function** : function, optional, default=None

A function that returns the portion of a control and sensor label that can be used for scaling the y axes.

**Returns** **axes** : ndarray of matplotlib.axis, shape(q, p)

**sensors**

**solve**(*sparse_a=False*, *gain_inclusion_matrix=None*, *ignore_cov=False*)
Returns the estimated gains and sensor limit cycles along with their variance.

**Parameters** **sparse_a** : boolean, optional, default=False

If true a sparse A matrix will be used along with a sparse linear least squares solver.

**gain_inclusion_matrix** : boolean array_like, shape(q, p)

A matrix which is the same shape as the identified gain matrices which has False in place of gains that should be assumed to be zero and True for gains that should be identified.

**ignore_cov: boolean, optional, default=False**

The covariance computation for a very large A matrix can be extremely slow. If this is set to True, then the computation is skipped and the covariance of the identified parameters is set to zero.

**Returns** **gain_matrices** : ndarray, shape(n, q, p)

The estimated gain matrices for each time step.

**control_vectors** : ndarray, shape(n, q)

The nominal control vector plus the gains multiplied by the reference sensors at each time step.

**variance** : float

The variance in the fitted curve.

**gain_matrices_variance** : ndarray, shape(n, q, p)

The variance of the found gains (covariance is neglected).

**control_vectors_variance** : ndarray, shape(n, q)

The variance of the found commanded controls (covariance is neglected).

**estimated_controls** : pandas.Panel

**validation_data**

CHAPTER 4

# Introduction

This is a collection of tools that are helpful for gait analysis. Some are specific to the needs of the Human Motion and Control Lab at Cleveland State University but other portions may have potential for general use. It is relatively modular so you can use what you want. It is primarily structured as a Python distribution but the Octave files are also accessible independently.

CHAPTER 5

# Python Packages

The main Python package is `gaitanalysis` and it contains five modules listed below. `oct2py` is used to call Octave routines in the Python code where needed.

**`gait.py`** General tools for working with gait data such as gait landmark identification and 2D inverse dynamics. The main class is `GaitData`.

**`controlid.py`** Tools for identifying control mechanisms in human locomotion.

**`markers.py`** Routines for processing marker data.

**`motek.py`** Tools for processing and cleaning data from Motek Medical's products, e.g. the D-Flow software outputs.

**`utils.py`** Helper functions for the other modules.

Each module has a corresponding test module in `gaitanalysis/tests` sub-package which contain unit tests for the classes and functions in the respective module.

CHAPTER 6

# Octave Libraries

Several Octave routines are included in the `gaitanalysis/octave` directory.

**`2d_inverse_dynamics`** Implements joint angle and moment computations of a 2D lower body human.

**`inertial_compensation`** Compensates force plate forces and moments for inertial effects and re-expresses the forces and moments in the camera reference frame.

**`mmat`** Fast matrix multiplication.

**`soder`** Computes the rigid body orientation and location of a group of markers.

**`time_delay`** Deals with the analog signal time delays.

CHAPTER 7

# Installation

You will need Python 2.7 and setuptools to install the packages. Its best to install the dependencies first (NumPy, SciPy, matplotlib, Pandas, PyTables). The SciPy Stack instructions are helpful for this: http://www.scipy.org/stackspec.html.

Supported versions:

- python >= 2.7
- numpy >= 1.6.1
- scipy >= 0.9.0
- matplotlib >= 1.1.0
- tables >= 2.3.1
- pandas >= 0.12.0
- pyyaml >= 3.10
- DynamicistToolKit >= 0.3.5
- oct2py >= 1.2.0
- octave >= 3.8.1

We recommend installing Anaconda for users in our lab to get all of the dependencies.

We also utilize Octave code, so an install of Octave with is also required. See http://octave.sourceforge.net/index.html for installation instructions.

You can install using pip (or easy_install). Pip will theoretically [1] get the dependencies for you (or at least check if you have them):

```
$ pip install https://github.com/csu-hmc/GaitAnalysisToolKit/zipball/master
```

Or download the source with your preferred method and install manually.

Using Git:

```
$ git clone git@github.com:csu-hmc/GaitAnalysisToolKit.git
$ cd GaitAnalysisToolKit
```

Or wget:

```
$ wget https://github.com/csu-hmc/GaitAnalysisToolKit/archive/master.zip
$ unzip master.zip
$ cd GaitAnalysisToolKit-master
```

[1] You will need all build dependencies and also note that matplotlib doesn't play nice with pip.

Then for basic installation:

```
$ python setup.py install
```

Or install for development purposes:

```
$ python setup.py develop
```

## 7.1 Dependencies

It is recommended to install the software dependencies as follows:

Octave can be installed from your package manager or from a downloadable binary, for example on Debian based Linux:

```
$ sudo apt-get install octave
```

For oct2py to work, calling Octave from the command line should work after Octave is installed. For example,

```
$ octave
GNU Octave, version 3.8.1
Copyright (C) 2014 John W. Eaton and others.
This is free software; see the source code for copying conditions.
There is ABSOLUTELY NO WARRANTY; not even for MERCHANTABILITY or
FITNESS FOR A PARTICULAR PURPOSE.  For details, type 'warranty'.

Octave was configured for "x86_64-pc-linux-gnu".

Additional information about Octave is available at http://www.octave.org.

Please contribute if you find this software useful.
For more information, visit http://www.octave.org/get-involved.html

Read http://www.octave.org/bugs.html to learn how to submit bug reports.
For information about changes from previous versions, type 'news'.

octave:1>
```

The core dependencies can be installed with conda in a conda environment:

```
$ conda create -n gait python=2.7 pip numpy scipy matplotlib pytables pandas pyyaml nose sphinx
$ source activate gait
```

And the dependencies which do not have conda packages can be installed into the environment with pip:

```
(gait)$ pip install DynamicistToolKit oct2py
```

CHAPTER 8

# Tests

When in the repository directory, run the tests with nose:

```
$ nosetests
```

CHAPTER 9

# Vagrant

A vagrant file and provisioning script are included to test the code on both a Ubuntu 12.04 and Ubuntu 13.10 box. To load the box and run the tests simply type:

```
$ cd vagrant
$ vagrant up
```

See `VagrantFile` and the `*bootstrap.sh` files to see what's going on.

CHAPTER 10

# Documentation

The documentation is hosted at ReadTheDocs:

http://gait-analysis-toolkit.readthedocs.org

You can build the documentation (currently sparse) if you have Sphinx and numpydoc:

```
$ cd docs
$ make html
$ firefox _build/html/index.html
```

CHAPTER 11

# Contributing

The recommended procedure for contributing code to this repository is detailed here. It is the standard method of contributing to Github based repositories (https://help.github.com/articles/fork-a-repo).

If you have don't have access rights to this repository then you should fork the repository on Github using the Github UI and clone the fork that you just made to your machine:

```
git clone git@github.com:<your-username>/GaitAnalysisToolKit.git
```

Change into the directory:

```
cd GaitAnalysisToolKit
```

Now, setup a remote called `upstream` that points to the main repository so that you can keep your local repository up-to-date:

```
git remote add upstream git@github.com:csu-hmc/GaitAnalysisToolKit.git
```

Now you have a remote called `origin` (the default) which points to **your** Github account's copy and a remote called `upstream` that points to the main repository on the csu-hmc organization Github account.

It's best to keep your local master branch up-to-date with the upstream master branch and then branch locally to create new features. To update your local master branch simply:

```
git checkout master
git pull upstream master
```

If you have access rights to the main repository simply, clone it and don't worry about making a fork on your Github account:

```
git clone git@github.com:csu-hmc/GaitAnalysisToolKit.git
```

Change into the directory:

```
cd GaitAnalysisToolKit
```

Now, to contribute a change to the repository you should create a new branch off of the local master branch:

```
git checkout -b my-branch
```

Now make changes to the software and be sure to always include tests! Make sure all tests pass on your machine with:

```
nosetests
```

Once tests pass, add any new files you created:

```
git add my_new_file.py
```

Now commit your changes:

```
git commit -am "Added an amazing new feature."
```

Push your commits to a mirrored branch on the Github repository that you cloned:

```
git push origin my-branch
```

Now visit the repository on Github (either yours or the main one) and you should see a "compare and pull button" to make a pull request against the main repository. Github and Travis-CI will check for merge conflicts and run the tests again on a cloud machine. You can ask others to review your code at this point and if all is well, press the "merge" button on the pull request. Finally, delete the branches on your local machine and on your Github repo:

```
git branch -d my-branch && git push origin :my-branch
```

## 11.1 Git Notes

- The master branch on main repository on Github should always pass all tests and we should strive to keep it in a stable state. It is best to not merge contributions into master unless tests are passing, and preferably if someone else approved your code.
- In general, do not commit changes to your local master branch, always pull in the latest changes from the master branch with `git pull upstream master` then checkout a new branch for your changes. This way you keep your local master branch up-to-date with the main master branch on Github.
- In general, do not push changes to the main repo master branch directly, use branches and push the branches up with a pull request.
- In general, do not commit binary files, files generated from source, or large data files to the repository. See https://help.github.com/articles/working-with-large-files for some reasons.

CHAPTER 12

# Release Notes

## 12.1 0.1.2

- Fixed bug preventing GaitData.plot_grf_landmarks from working.
- Removed inverse_data.mat from the source distribution.

## 12.2 0.1.1

- Fixed installation issue where the octave and data files were not included in the installation directory.

## 12.3 0.1.0

- Initial release
- Copied the walk module from DynamicistToolKit @ eecaebd31940179fe25e99a68c91b75d8b8f191f

CHAPTER 13

# Indices and tables

- *genindex*
- *modindex*
- *search*

[Wu1995] Wu G. and Cavanagh, P. R., 1995, “ISB recommendations for standardization in the reporting of kinematic data”, J. Biomechanics, Vol 28, No 10.

## g

## Symbols



## A



## C

## S



## T



## V



## W