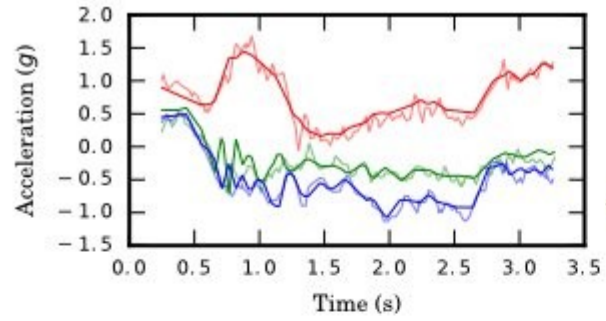
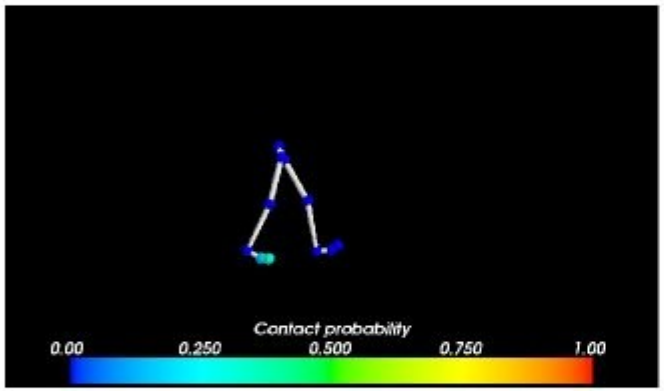
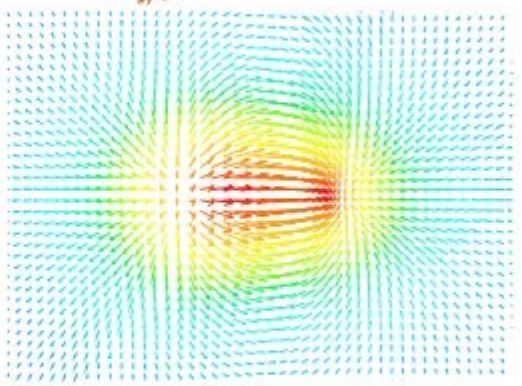
IMUSim: Simulating inertial and magnetic sensor systems in Python







Martin Ling & Alex Young
School of Informatics
University of Edinburgh

10th SciPy Conference, 13th July 2011, Austin, Texas

Overview

IMUSim is a new Python-based simulation package for modelling systems that use **accelerometers, gyroscopes** and/or **magnetometers**.

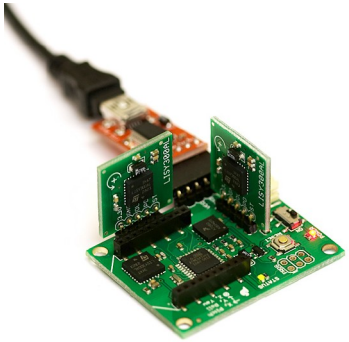
IMU = Inertial Measurement Unit = a device with these sensors.

This talk will explain:

- Background - why we wrote it
- How we implemented it
- How we tested it

Afterwards I'd like to chat about:

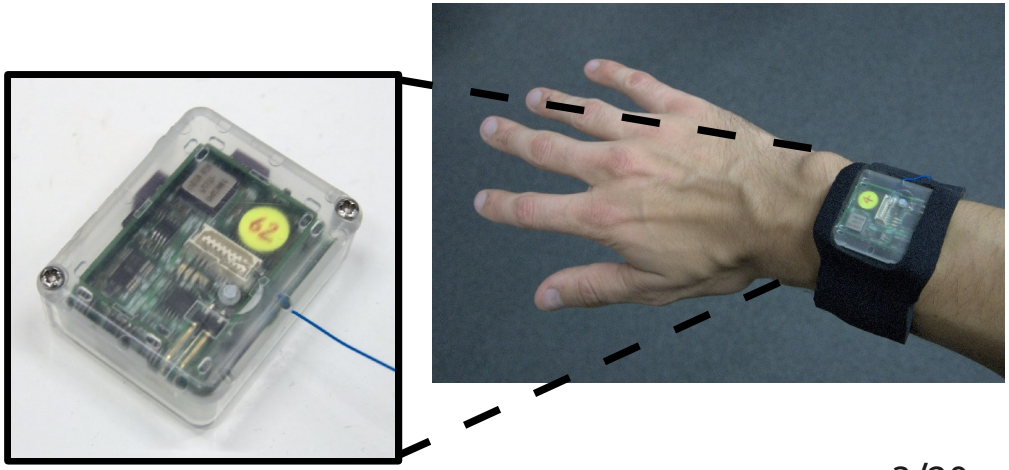
- Opportunities for code reuse
- Ideas for improvement/extension



Our research

Motion capture of human subjects:

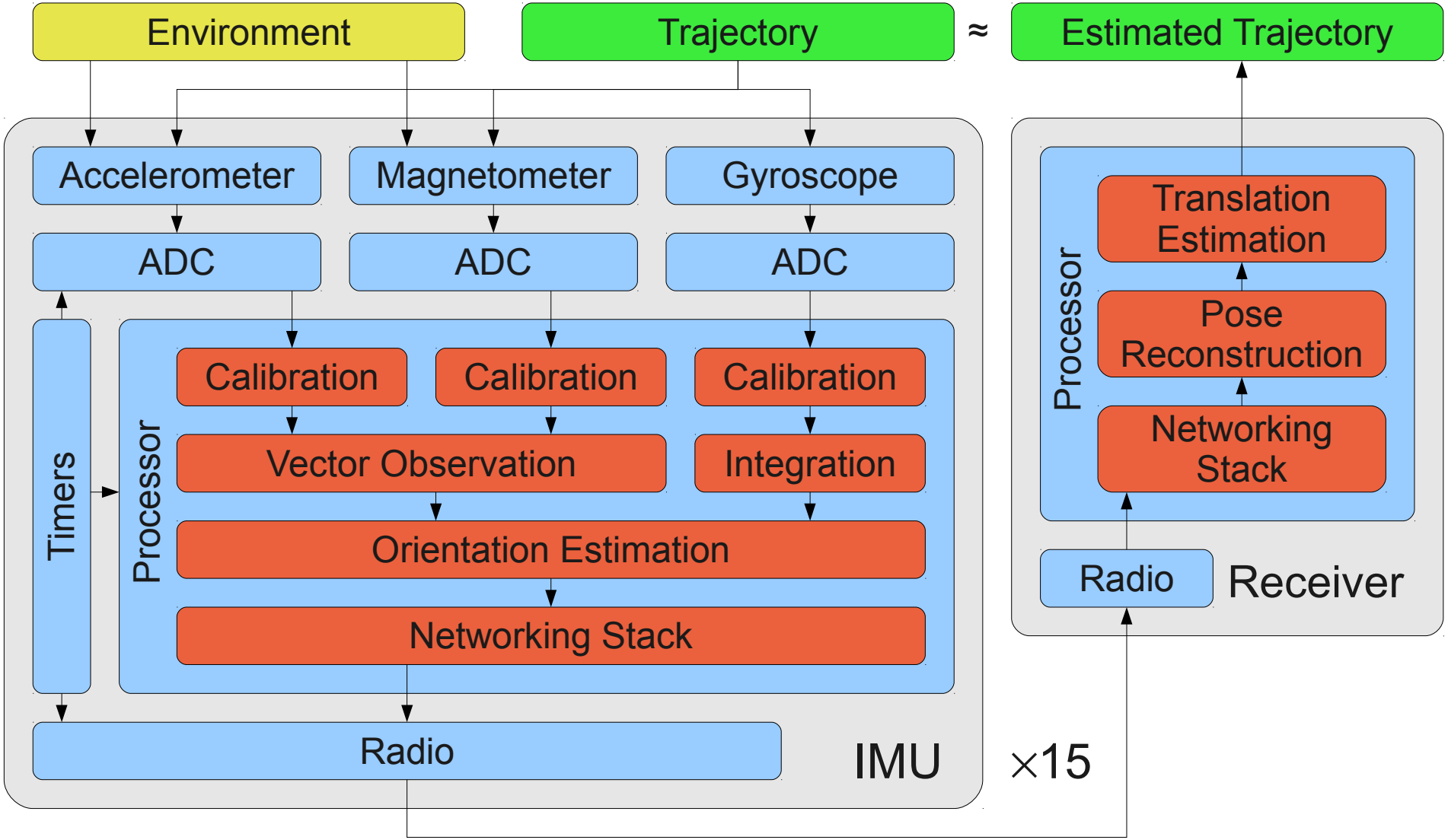
- Conventional method: high speed infrared cameras and optical markers
  - Limited to tracking within small area
  - Problems with marker occlusion
  - Lots of manual post-processing
- Our method: wearable wireless IMUs
  - Unlimited tracking area
  - Constant capture
  - Realtime output



IMU-based motion capture - example

Subject wearing fifteen *Orient-2* wireless IMUs (Young & Ling 2006)

IMU-based motion capture: components



Why we needed a simulator

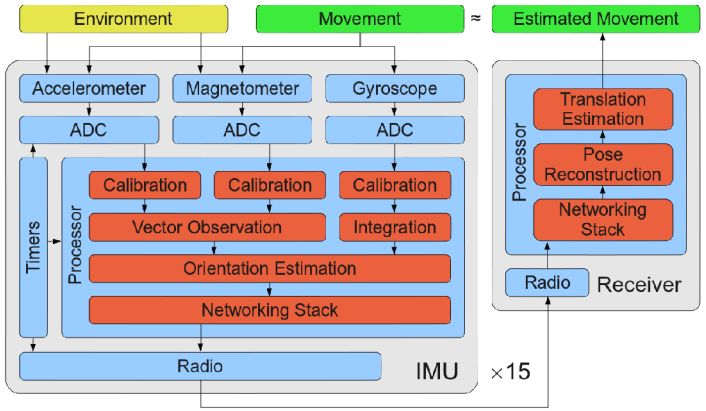
Our goal is to improve accuracy, but:

- Huge amount of work required to develop a complete working system, costly to make changes.
- Impossible to isolate error sources:
  - Noise in some part of the system?
  - Miscalibration?
  - Flawed processing algorithm?
  - Implementation bug?
  - Error in reference measurement?
- Difficult to compare methods and approaches of other systems in a controlled manner.
- Existing simulations:
  - Too simplistic – not realistic tests of methods
  - Too specific – code not reusable even if released

Requirements

For us:

- Simulate complete wireless multi-IMU systems
- Use realistic human motions
- Use realistic environments
- Allow quick interchange of components and methods
- Detailed simulation not required unless relevant to accuracy

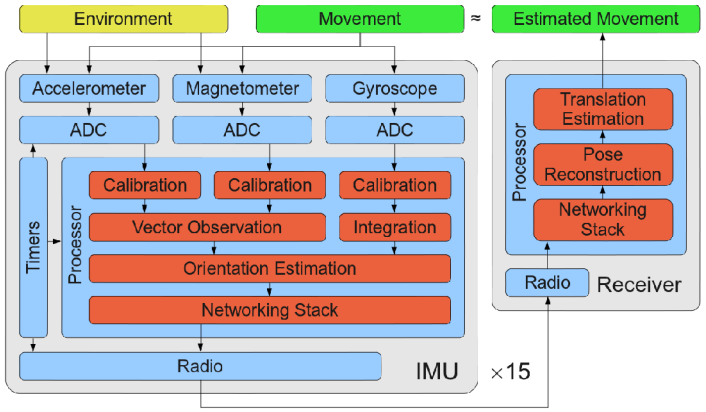


Beyond this:

- Keep code flexible/reusable – not just specific to our needs
- Support simulation of any system using inertial/magnetic sensors
- Release as an open source project for others to extend

Design approach

- Identify the independent, interchangeable elements of the scenario:
  - e.g. sensor, trajectory, environment, vector observation algorithm, etc
- Define an API for models of each element:
  - API defined by abstract classes,
    e.g. `Sensor`, `Trajectory`, `MagneticField`
  - Avoid assumptions about usage or implementation details wherever practical
  - Use semi-abstract classes ('mix-ins') to provide reusable functionality where appropriate
    - e.g. `NoisySensor`, `ContinuousRotationTrajectory`
- Provide models that are driven by real captured data
- No UI - design the API for interactive use
  - Save typing with useful wrappers and sensible defaults
- Use existing library code wherever possible
  - Exceptions: for speed, API consistency, or to avoid rare prerequisites



The IMUSim package

Basic IMU modelling

Models of all factors that affect sensor measurements:

- Trajectory followed by the sensor through space:
  - Position, velocity, acceleration
  - Rotation, angular velocity, angular acceleration
- Environment around the IMU:
  - Magnetic field
  - Gravitational field
- Sensor hardware:
  - Sensitivity, measurement range, bias, etc
  - Noise
- ADC hardware:
  - Range, resolution, linearity, etc
  - More noise
- Timer hardware

Additional functionality

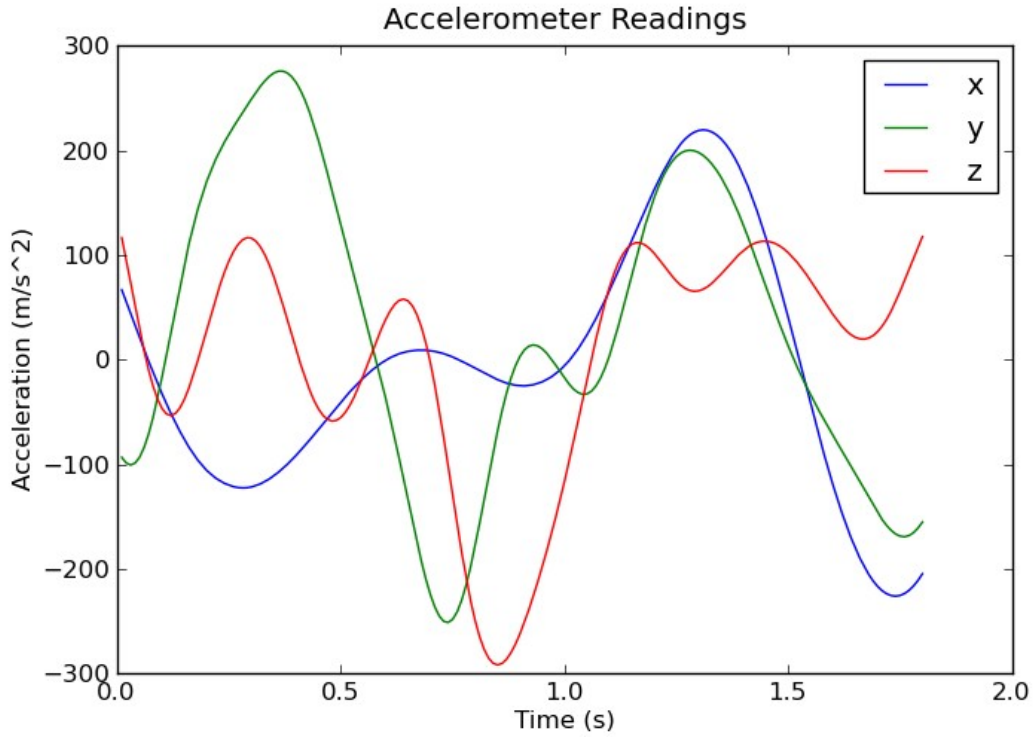
- Radio, network stack and channel model support for modelling wireless multi-IMU systems
- Implementations of existing processing algorithms for inertial and magnetic sensor data:
  - Sensor calibration
  - Vector observation
  - Orientation estimation
  - Posture reconstruction
  - Translation estimation
- General purpose mathematical utilities useful for implementing models and processing algorithms, e.g. Kalman filter implementations
- 2D and animated 3D visualisation tools.

Approx 7,000 lines of Python using NumPy, SciPy, SimPy, Matplotlib, Mayavi and Cython

Usage example #1: ideal accelerometer on random trajectory

```
# Import all public symbols from IMUSim
from imusim.all import *

# Create a new simulation
sim = Simulation()

# Create a randomly defined trajectory
trajectory = RandomTrajectory()

# Create an instance of an ideal IMU
imu = IdealIMU(simulation=sim,
     trajectory=trajectory)

# Define a sampling period
dt = 0.01

# Set up a behaviour that runs on the
# simulated IMU
behaviour = BasicIMUBehaviour(platform=imu,
     samplingPeriod=dt)

# Set the time inside the simulation
sim.time = trajectory.startTime

# Run the simulation till the desired
# end time
sim.run(trajectory.endTime)
```

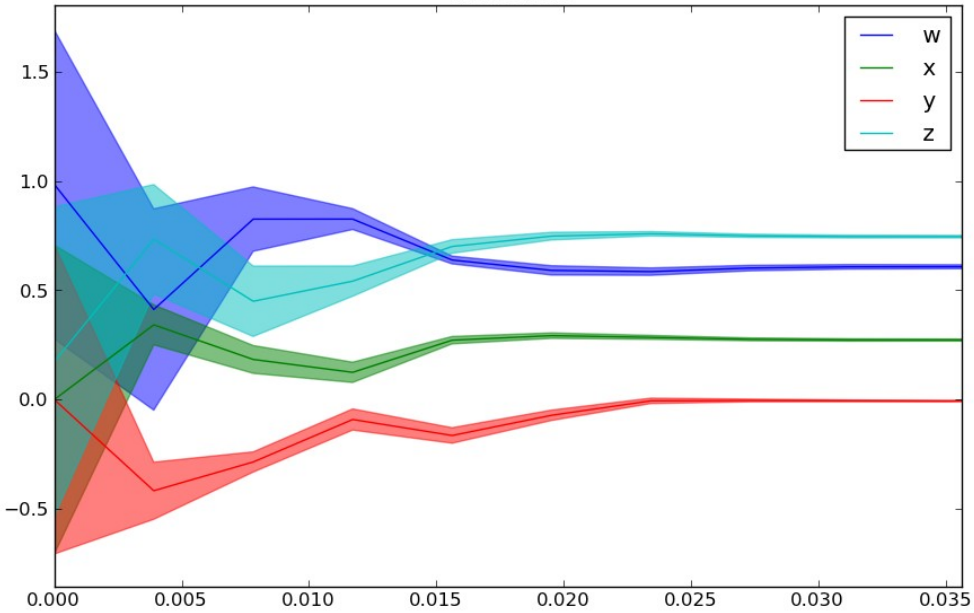
```
# Plot accelerometer measurements
plot(imu.accelerometer.rawMeasurements)
title("Accelerometer Readings")
xlabel("Time (s)")
ylabel("Acceleration (m/s^2)")
legend()
```



Data types: Time series

New `TimeSeries` class to represent time series data:

```
>>> imu.accelerometer.rawMeasurements.timestamps
array([ 0.01, 0.02, ..., 1.79, 1.8 ])

>>> imu.accelerometer.rawMeasurements.values
array([[  66.705814  , ..., -204.6486176 ],
       [ -93.40026896, ..., -155.16993659],
       [ 116.56420017, ...,  117.56964057]]
```



- Data may be scalars, vectors or quaternions
- May have associated variances or covariances
- Data points may be added sequentially:

  ```
  timeSeries.add(time, value, variance=None)
  ```

- Array versions of data constructed on demand
- Simplifies passing related arrays around, avoids stupid mistakes
- Augmented version of `plot()` supports plotting `TimeSeries` directly:
  - Automatic labels
  - Automatic display of uncertainty
- Very general-purpose idea – would be good to combine ideas with similar classes elsewhere

Data types: Quaternions

Quaternions are an extension of complex numbers, useful to represent 3D rotations.

We wrote a new quaternion math implementation in Cython:

- May be the fastest and most complete implementation available for Python now
- Supports efficient operations with arrays of quaternion values
- Please reuse it!

```
# Operations with single quaternions

>>> q1 = Quaternion(0, 1, 0, 0)

>>> q1.toMatrix()
matrix([[  1.,  0.,  0.],
        [  0., -1.,  0.],
        [  0.,  0., -1.]])

>>> q2 = Quaternion.fromEuler((45, 10, 30), order='zyx')

>>> q1 * q2
Quaternion(-0.2059911, 0.8976356, -0.3473967, 0.176446)

>>> q2.rotateVector(vector(1,2,3))
array([[ 0.97407942],
       [ 1.30224882],
       [ 3.36976517]])
```

```
# A TimeSeries may have quaternion values, stored as
# a QuaternionArray which wraps an Nx4 NumPy array.

>>> trajectory.rotationKeyFrames.values
QuaternionArray(
    array([[-0.04667, -0.82763,  0.29852, -0.47300],
           [-0.10730, -0.81727,  0.33822, -0.45402],
           ...,
           [ 0.40666, -0.04250,  0.80062,  0.43796],
           [ 0.42667, -0.01498,  0.82309,  0.37449]]))

>>> trajectory.rotationKeyFrames.values[1]
Quaternion(-0.10730, -0.81727,  0.33822, -0.45402)

# A QuaternionArray may be used in math expressions
# for efficient operations over arrays of quaternions.
```

Model API example: trajectories

- The measurements of an IMU depend on its trajectory. A trajectory model must provide:

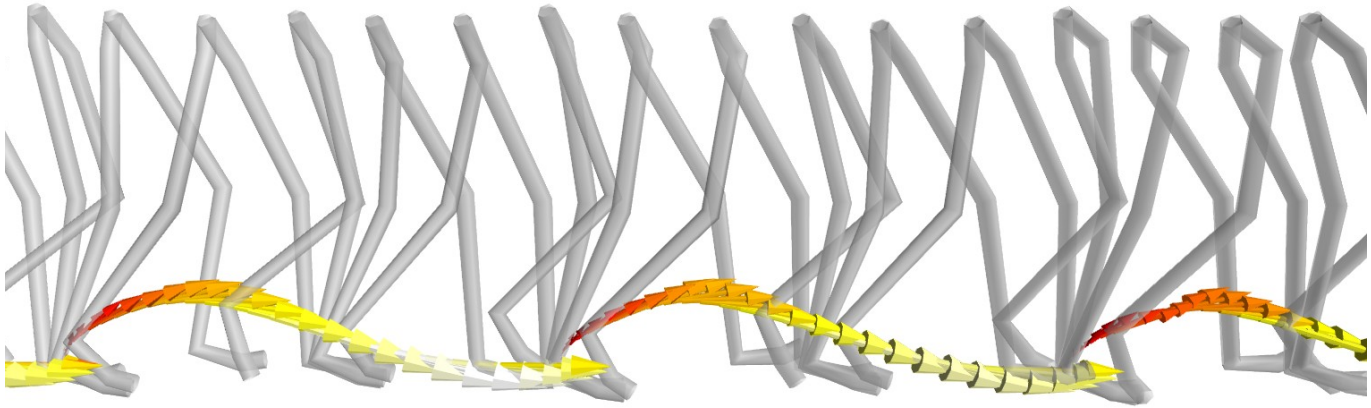
```
# Position in m
>>> trajectory.position(t)
array([[-10.36337587],
       [ 4.63926506],
       [ -0.17801693]])

# Linear velocity in m/s
>>> trajectory.velocity(t)
array([[ 30.79525389],
       [-20.9180481 ],
       [ 2.68236355]])

# Linear acceleration in m/s^2
>>> trajectory.acceleration(t)
array([[ 178.30674569],
       [ -15.11472827],
       [ 15.54901256]])
```

```
# Rotation as a quaternion
>>> trajectory.rotation(t)
Quaternion(-0.046679, -0.82763, 0.29852, -0.47300)

# Rotational velocity in rad/s
>>> trajectory.rotationalVelocity(t)
array([[-2.97192064],
       [ 2.97060751],
       [-7.32688967]])

# Rotational acceleration in rad/s^2
>>> trajectory.rotationalAcceleration(t)
array([[ -8.46813312],
       [ 19.43475152],
       [-31.28760834]])

# all in the global co-ordinate frame.
```

- The trajectory may be fully defined in advance, or evolve as the simulation progresses, e.g. to simulate the effect of a control system.

Interpolating from motion capture data

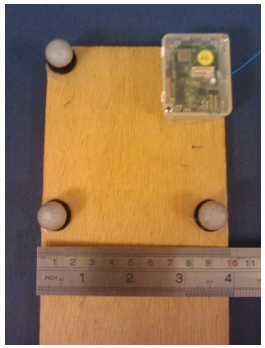
- Defining realistic trajectory models directly is difficult, especially for e.g. humans
- To allow simulations using realistic trajectories, we create continuous-time, differentiable trajectories from motion capture data, accounting for rigid body kinematics.



- This requires:
  - Cubic spline fitting of position data – done using the `splrep/splev` functions from `scipy.interpolate`, including appropriate smoothing to account for measurement noise.
  - Equivalent spline fitting of rotation data, which is less straightforward:
    - SLERP/SQUAD are not C^2-continuous, i.e. cannot recover an angular acceleration.
    - We provide an implementation of the quaternion B-spline algorithm by Kim et al. 1995

Magnetic field interpolation

- Real environments have significant magnetic field distortions
- These affect the ability to find headings accurately
- Important to simulate in realistic fields



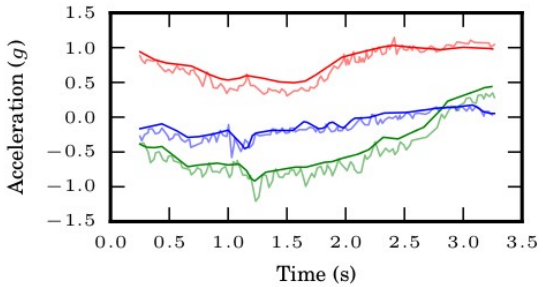
- Sampling a real field in a 3D grid is laborious
- Much quicker to sweep an IMU around while tracking it optically
- Simulation using this data requires $\mathbb{R}^3 \rightarrow \mathbb{R}^3$ interpolation with non-uniform input points
- Can be implemented using Natural Neighbour Interpolation, based on a 3D Delaunay triangulation
- We use a wrapper for the C implementation of this method by Ross Hemsley

http://code.google.com/p/interpolate3d

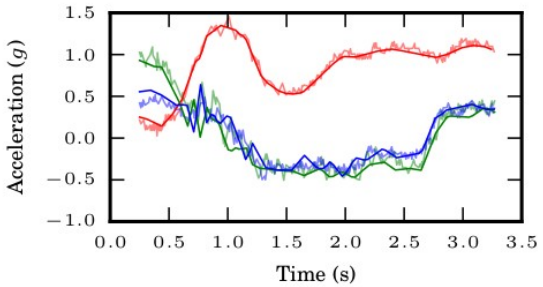
Testing the simulator

- We test the simulator by comparing its simulated sensor measurements with real ones
  - Synchronised optical motion capture & IMU logging
  - Magnetic field mapping of capture area
  - Construct simulation from capture & mapping
  - Compare logged IMU data to simulated results
- Data from these experiments is shipped as part of the test suite – final test is against reality
  - Suite also includes unit tests (30k+ test cases)
  - Designed for use with `nosetests`
  - Coverage analysis to find untested paths
    - Doing this for Cython code is a current issue

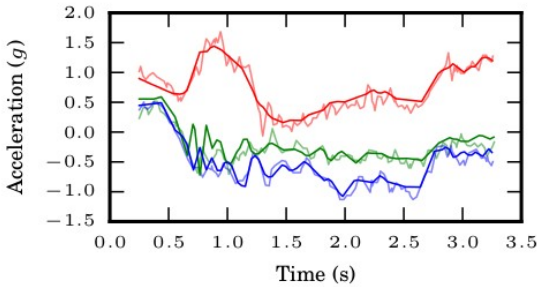
Comparing against reality



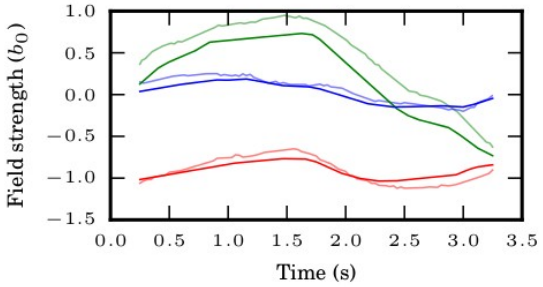
(a) Femur Accelerometer



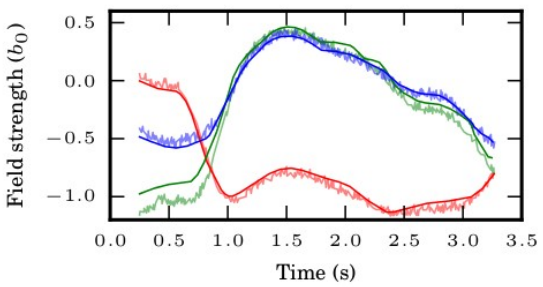
(b) Tibia Accelerometer



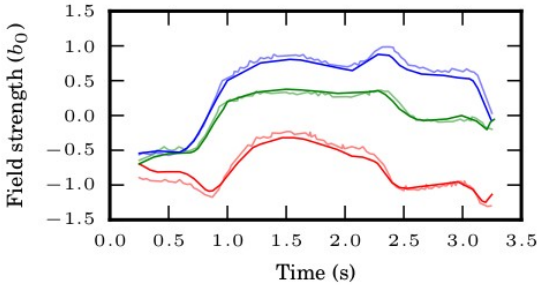
(c) Foot Accelerometer



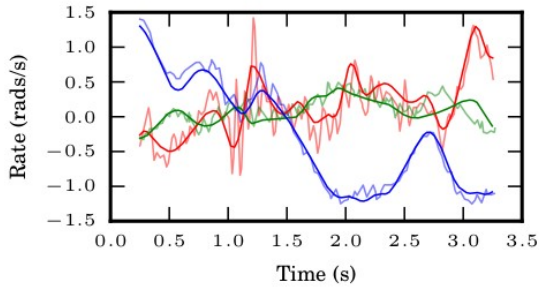
(d) Femur Magnetometer



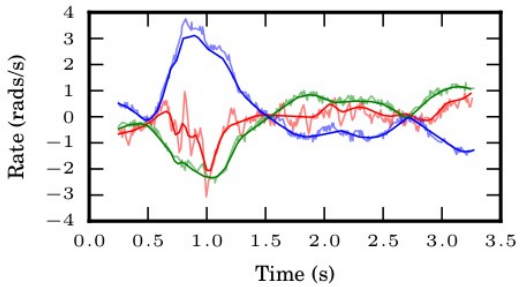
(e) Tibia Magnetometer



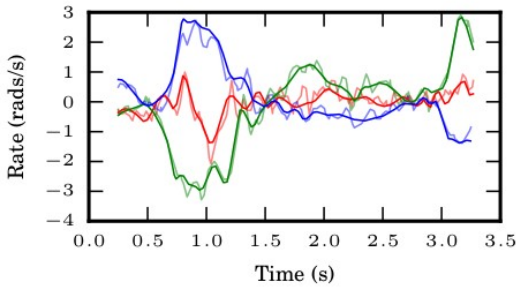
(f) Foot Magnetometer



(g) Femur Gyroscope



(h) Tibia Gyroscope



(i) Foot Gyroscope

Usage example #2: realistic simulation of an IMU on the foot of a walking human

```
# Import all public symbols from IMUSim
from imusim.all import *

# Define a sampling period
dt = 0.01

# Create an instance of a realistic IMU model
imu = Orient3IMU()

# Create a magnetic field model from sampled data
magField = InterpolatedVectorField(
    loadtxt('positions.txt'),
    loadtxt('values.txt'))

# Create an environment using this field
env = Environment(magneticField=magField)

# Define a procedure for calibrating an IMU
# in our target environment
calibrator = ScaleAndOffsetCalibrator(
    environment=env, samples=1000,
    samplingPeriod=dt, rotationalVelocity=20)

# Calibrate the IMU
cal = calibrator.calibrate(imu)

# Import motion capture data of a human
sampledBody = loadBVHFile('walk.bvh',
    CM_TO_M_CONVERSION)

# Convert to continuous time trajectories
splinedBody = SplinedBodyModel(sampledBody)

# Create a new simulation
sim = Simulation(environment=env)

# Assign the IMU to the simulation
imu.simulation = sim

# Attach the IMU to the subject's right foot
imu.trajectory = splinedBody.getJoint('rfoot')

# Set the starting time of the simulation
sim.time = splinedModel.startTime

# Set up the behaviour to run on the IMU
BasicIMUBehaviour(platform=imu,
    samplingPeriod=dt, calibration=cal,
    initialTime=sim.time)

# Run the simulation
sim.run(splinedModel.endTime)
```

Further information

- IMUSim tutorial
  - Introduces enough usage to allow realistic simulations
  - Assumes programming & domain knowledge but no Python experience
- API reference
  - Generated by Epydoc from docstrings
- Mailing list
- Published papers
  - A. D. Young, M. J. Ling and D. K. Arvind. "IMUSim: A Simulation Environment for Inertial Sensing Algorithm Design and Evaluation", in *Proceedings of the 10th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN 2011)*, pp. 199-210. ACM, 2011.
  - Paper to appear in SciPy 2011 proceedings
- Source code (GPLv3)

http://www.imusim.org/

Contributions & Conclusions

Contributions

- A flexible simulation framework for modelling any system including inertial or magnetic sensors
- Quaternion math library
- Time series data utilities
- C^2-continuous quaternion interpolation
- Vector field interpolation
- Reusable Kalman filter and Unscented Kalman Filter implementations

Conclusions

- Python made this an easy job, completed as a side project by two researchers over a few months
- Ease of development encouraged us to make it as flexible and reusable as possible – this was very little extra effort
- Wide range of potential use areas: robotics, aerospace, healthcare...
- Many opportunities for further integration – particularly with physics simulations