Professional Development Short Course On:

Aerospace Simulations In C++

Instructor:

Dr. Peter Zipfel

ATI Course Schedule:  
http://www.ATIcourses.com/schedule.htm

ATI's Aerospace Simulations In C++:

http://www.aticourses.com/aerospace_simulations_in_C++.htm
Boost Your Skills with On-Site Courses Tailored to Your Needs

The Applied Technology Institute specializes in training programs for technical professionals. Our courses keep you current in the state-of-the-art technology that is essential to keep your company on the cutting edge in today’s highly competitive marketplace. Since 1984, ATI has earned the trust of training departments nationwide, and has presented on-site training at the major Navy, Air Force and NASA centers, and for a large number of contractors. Our training increases effectiveness and productivity. Learn from the proven best.

For a Free On-Site Quote Visit Us At:  http://www.ATIcourses.com/free_onsite_quote.asp

For Our Current Public Course Schedule Go To:  http://www.ATIcourses.com/schedule.htm
Lab2 Classes

... build your world of hierarchies

- CRUISE is built on the Cadac hierarchy
- The classes of CADAC++
- Multiple encapsulated objects
- The Cadac hierarchy
- Abstract base class Cadac
- Round3 inherits from Cadac
- Cruise inherits from Round3
- Target inherits from Round3
- Satellite inherits from Round3
- What you have learned ♣

<table>
<thead>
<tr>
<th>Lab</th>
<th>Topics</th>
<th>Simulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>C++ Essentials QuickStart</td>
<td>CRUISE</td>
</tr>
<tr>
<td>2</td>
<td>Classes Satellite Simulation</td>
<td>CRUISE_2</td>
</tr>
<tr>
<td>3</td>
<td>Modules Target Simulation</td>
<td>CRUISE_3</td>
</tr>
<tr>
<td>4</td>
<td>Tables UAV Uncontrolled</td>
<td>CRUISE_4</td>
</tr>
<tr>
<td>5</td>
<td>Events UAV Controlled</td>
<td>CRUISE_5</td>
</tr>
<tr>
<td>6</td>
<td>Polymorphism UAV Navigation</td>
<td>CRUISE_6</td>
</tr>
<tr>
<td>7</td>
<td>ComBus UAV Homing</td>
<td>CRUISE_7</td>
</tr>
<tr>
<td>8</td>
<td>Architectures UAV Netcentric</td>
<td>CRUISE</td>
</tr>
</tbody>
</table>

SCHEDULE
CRUISE is built on the Cadac hierarchy

- **Cadac** is the abstract base class
  - Characterized by the fact that it contains pure virtual functions

- **Round3** class is derived from **Cadac**
  - Provides the structure for the equations of motion

- **Cruise**, **Target**, and **Satellite** classes are derived from **Round3**
  - These classes declare the vehicle types with names: CRUISE3, TARGET3, and SATELLITE3

- **Modules** are member functions of the derived classes
  - They define the mathematical models of the vehicle components

- **Other classes support the simulation architecture**
## The classes of CADAC++

<table>
<thead>
<tr>
<th>CLASS</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cadac, ...</td>
<td>Hierarchical class structure of vehicles</td>
</tr>
<tr>
<td>Vehicle</td>
<td>Hosting a pointer array of type <code>Cadac</code></td>
</tr>
<tr>
<td>Module</td>
<td>Storing module information</td>
</tr>
<tr>
<td>Variable</td>
<td>Declaring module-variables</td>
</tr>
<tr>
<td>Event</td>
<td>Storing event information</td>
</tr>
<tr>
<td>Packet</td>
<td>Declaring data packets for global communication bus</td>
</tr>
<tr>
<td>Datadeck</td>
<td>Hosting a pointer array of type <code>Table</code></td>
</tr>
<tr>
<td>Table</td>
<td>Storing tabular data</td>
</tr>
<tr>
<td>Matrix</td>
<td>Storing matrix operations</td>
</tr>
<tr>
<td>Document</td>
<td>Storing module-variable definitions</td>
</tr>
</tbody>
</table>
Multiple encapsulated objects

- Each aerospace vehicle is an object created from its hierarchical class structure
  - Data and methods (member functions) are encapsulated
    - Aerodynamic and propulsion data tables
    - Vehicle characteristics are computed in module functions
  - **Cadac** is the abstract base class

- Examples

- **Polymorphism** enables the call of the vehicle objects during run-time
  - **Inheritance** enables a derived object to acquire the properties of the base object
  - **Virtual functions** are overridden until the correct module is executed
The Cadac hierarchy

Abstract base class

Derived class

Derived class

Vehicle names

Target Cruise Satellite

TARGET3 CRUISE3 SATELLITE3

Module-Variable arrays

cruise []
satellite []
target []

Communication bus

combus []

Virtual functions

newton ()
environment ()

aerodynamics ()
propulsion ()
forces ()
newton ()

seeker ()
guidance ()
control ()

intercept ()

Modular structure

cruise []
satellite []
target []

newton ()
environment ()
targeting ()

seeker ()
guidance ()
control ()

intercept ()
Abstract base class Cadac

- **Cadac** declares
  - Pointers of module-variable arrays
  - Pure virtual functions for the executive processes
  - Pure virtual functions for the modules

```cpp
class Cadac
{
private:
  char name[CHARN]; //vehicle object name

protected:
  //module-variable array of class 'Round3'
  Variable *round3;
  //module-variable array of class 'Cruise'
  Variable *cruise;
  //module-variable array of class 'Target'
  Variable *target;
  //module-variable array of class 'Satellite'
  Variable *satellite;

public:
  Cadac(){} virtual~Cadac(){};
  //executive functions
  //pure virtual functions
  virtual void sizing_arrays()=0;
  ....
  //module functions
  //... pure virtual functions
};
```

```cpp
class Variable
{
private:
  char name[CHARN];
  char type[CHARN];
  double rval;
  int ival;
  Matrix VEC;
  Matrix MAT;
  char def[CHARL];
  char mod[CHARN];
  char role[CHARN];
  char out[CHARN];

public:
  //methods for initialization
  //methods for reading and loading
};
```
Round3 inherits from Cadac

- Spherical rotating Earth, 3 degrees of freedom
- void Round3::environment(...) 
  - ISO 62 standard atmosphere 
  - Newton’s gravitational acceleration (inverse square law)
- void Round3::newton(...) 
  - Equations of motion are integrated in inertial coordinates 
  - Conversion of input and output from/to geographic coordinates

```cpp
class Round3:public Cadac {
  protected:
  //declaring index arrays which identify module-variables for output
  public:
    Round3();
    virtual~Round3(){};
    //executive functions
    //... pure virtual functions
    //module functions
    //... pure virtual functions
    //virtual functions to be defined in this class
    virtual void def_environment();
    virtual void init_environment(...);
    virtual void environment(...);
    virtual void def_newton();
    virtual void init_newton();
    virtual void newton(...);
};
```
Cruise inherits from Round3

- Bank-to-turn UAV
  ```c++
  void Cruise::aerodynamics();
  Drag polar
  void Cruise::propulsion(...);
  Turbojet
  void Cruise::forces();
  Aerodynamic and propulsive
  void Cruise::control(...);
  Heading, flight path, bank angle, load
  factor, altitude hold
  void Cruise::guidance();
  Line, pro-nav, point, arc
  void Cruise::targeting(...);
  Satellite targeting
  void Cruise::seeker(...);
  Line-of-sight seeker
  void Cruise::intercept(...);
  Waypoint and target miss distance
  ```

```c++
class Cruise:public Round3
{
protected:
  //name of "CRUISE3" vehicle object
  char cruise3_name[CHARL];
  //array pointers
  ...

public:
  Cruise();
  Cruise(...);
  virtual~Cruise();

  //executive functions
  ...

  //module functions
  virtual void aerodynamics();
  virtual void propulsion(...);
  virtual void forces();
  virtual void control(...);
  virtual void guidance();
  virtual void targeting(...);
  virtual void seeker(...);
  virtual void intercept(...);

  //functions of control module
  double control_heading(...);
  double control_flightpath(...);
  double control_bank(...);
  double control_load(...);
  double control_lateral(...);
  double control_altitude(...);

  //functions of guidance module
  Matrix guidance_line();
  Matrix guidance_pronav();
  Matrix guidance_point();
  double guidance_arc();
  double guidance_spline();

  //functions of seeker module
  void seeker_grnd_ranges(...);

  //functions of targeting module
  void targeting_satellite(...);
  void targeting_grnd_ranges(...);
};
```
Target inherits from Round3

- Target over the round, rotating Earth
  - void Target::forces();
    - Compensating for Coriolis and centrifugal accelerations
  - void Target::intercept(...);
    - Setting target status (alive, hit, dead)

```cpp
class Target: public Round3
{
protected:
  // name of "TARGET3" vehicle object
  char target3_name[CHARL];
  // array pointers
  ...
public:
  Target();
  Target(...);
  virtual ~Target();

  // executive functions
  ...
  // module function dummy returns
  virtual void def_aerodynamics(){};
  ...
  // module functions active
  virtual void forces();
  virtual void intercept(...);
};
```
Satellite inherits from Round3

```cpp
class Satellite:public Round3
{
protected:
    //name of "SATELLITE3" vehicle object
char satellite3_name[CHARL];
    //array pointers
...
public:
    Satellite();
    Satellite(...);
    virtual~Satellite();

    //executive functions
...
    //module function dummy returns
    virtual void def_aerodynamics();
    ...
    //module functions active
    virtual void forces();
    virtual void seeker(...);
};
```

- Target over the round, rotating Earth
  - void Satellite::forces();
    - Mass properties and optional thrusting
  - void Satellite::seeker(...);
    - Radar seeker (simplified)
What you have learned

- Class structure of the CADAC++ architecture
- `Cadac` is an abstract base class
- The class hierarchy declares the vehicle objects `Cruise`, `Target`, and `Satellite`
- The vehicle objects are named `CRUISE3`, `TARGET3`, and `SATELLITE3` in the ‘input.asc’ file
- Modules are member functions (methods) of the derived classes
- Use of access specifiers
- Use of constructors and destructors
Lab2  Satellite Simulation

… is the simplest simulation

- Features of satellite simulation
- Class hierarchy of satellite simulation
- Equations of motions are derived from Newton’s Law
- Inertial and Earth coordinate systems
- Earth and geographic coordinate systems
- Geographic and flight path coordinate system
- Geographic initialization
- Solving the equations of motion
- Circular orbiting satellite over the equator
- What you have learned ♠

<table>
<thead>
<tr>
<th>Lab</th>
<th>Topics</th>
<th>Simulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>C++ Essentials QuickStart</td>
<td>CRUISE</td>
</tr>
<tr>
<td>2</td>
<td>Classes Satellite Simulation</td>
<td>CRUISE_2</td>
</tr>
<tr>
<td>3</td>
<td>Modules Target Simulation</td>
<td>CRUISE_3</td>
</tr>
<tr>
<td>4</td>
<td>Tables UAV Uncontrolled</td>
<td>CRUISE_4</td>
</tr>
<tr>
<td>5</td>
<td>Events UAV Controlled</td>
<td>CRUISE_5</td>
</tr>
<tr>
<td>6</td>
<td>Polymorphism UAV Navigation</td>
<td>CRUISE_6</td>
</tr>
<tr>
<td>7</td>
<td>ComBus UAV Homing</td>
<td>CRUISE_7</td>
</tr>
<tr>
<td>8</td>
<td>Architectures UAV Netcentric</td>
<td>CRUISE</td>
</tr>
</tbody>
</table>

SCHEDULE
Features of satellite simulation

- 3 DoF, round, rotating Earth;
- Coordinates: longitude, latitude, altitude
- Newton’s equations solved in inertial coordinates
- Conversion from geographic to inertial coordinates and vice versa
- Propulsion is optional
Class hierarchy of satellite simulation

Abstract base class

Derived class

Derived class

Virtual functions

Round3

Cadac

Module-
Variable arrays

Satellite

forces ()

forces ()

forces ()

Modular structure

environment ( )

environment ( )

environment ( )

newton ( )

newton ( )

newton ( )
Equations of motions are derived from Newton’s Law

Newton’s Law

$$mD^{I}v^{I}_{B} = f_{a,p}^{I} + mg$$

Relationship of inertial and geographic velocities

$$v^{I}_{B} = v^{E}_{B} + \Omega^{EI} s_{BI}$$

Matrix equations programmed in ‘newton’ module

$$m\left[ dv_{B}^{I} / dt \right]^{I} = \overline{T}^{G} \left( \overline{T}^{VG} \left[ f_{a,p}^{I} \right]^{V} + m \left[ g^{G} \right] \right)$$

$$\left[ ds_{BI} / dt \right]^{I} = \left[ v_{B}^{I} \right]^{I}$$


I: Inertial coordinates
G: Geographic coordinates
V: Flight path coordinates

Zipfel, p. 261 *
Inertial and Earth coordinate systems

- Inertial coordinate system
  - $1^I$ axis is aligned with the Vernal Equinox
  - $3^I$ axis is aligned with the spin axes of the Earth

- Angle $\Xi$ is the hour angle

- The transformation matrix of Earth wrt inertial coordinates is

$$[T]^{EI} = \begin{bmatrix}
\cos \Xi & \sin \Xi & 0 \\
-\sin \Xi & \cos \Xi & 0 \\
0 & 0 & 1
\end{bmatrix}$$
Earth and geographic coordinate systems

- **Earth coordinate system**
  - $1^E$ axis is at the intersection of the Greenwich Meridian and the equator
  - $3^E$ axis is aligned with the spin axis of the Earth

- **Geographic coordinate system**
  - $1^G$ axis points to the north pole
  - $3^G$ axis point to the center of the Earth
  - $l$ longitude
  - $\lambda$ latitude

- **TM sequences through two intermediate coordinate systems $X$ and $Y$**

\[
[T]^{GE} = [T(180^\circ)]^{GY} [T(90^\circ - \lambda)]^{YX} [T(l)]^{XE}
\]

- **The TM of geographic wrt Earth coordinates is**

\[
[T]^{GE} = \begin{bmatrix}
-\sin \lambda \cos l & -\sin \lambda \sin l & \cos \lambda \\
-\sin l & \cos l & 0 \\
-\cos \lambda \cos l & -\cos \lambda \sin l & -\sin \lambda
\end{bmatrix}
\]
Geographic and flight path coordinate system

- Geographic coordinate system
  - \(1^G\) axis points to the north pole
  - \(3^G\) axis point to the center of the Earth

- Flight path coordinate system
  - \(1^V\) axis parallel and in the direction of the velocity vector
  - \(2^V\) axis is horizontal and points to the right
  - \(3^V\) axis point “down”

- \(\chi\) heading angle (chi)
- \(\gamma\) flight-path angle

- The TM of flight path wrt geographic coordinates is

\[
[T]^{VG} = [T(\gamma)]^{VX} [T(\chi)]^{XG}
\]

\[
[T]^{VG} = \begin{bmatrix}
\cos \gamma & 0 & -\sin \gamma \\
0 & 1 & 0 \\
\sin \gamma & 0 & \cos \gamma
\end{bmatrix}
\begin{bmatrix}
\cos \chi & \sin \chi & 0 \\
-\sin \chi & \cos \chi & 0 \\
0 & 0 & 1
\end{bmatrix}
\]

\[
= \begin{bmatrix}
\cos \gamma \cos \chi & \cos \gamma \sin \chi & -\sin \gamma \\
-\sin \chi & \cos \chi & 0 \\
\sin \gamma \cos \chi & \sin \gamma \sin \chi & \cos \gamma
\end{bmatrix}
\]

Zipfel, p. 80
Geographic initialization

- Geographic initialization converted to inertial position and velocity

Implemented in: `void Round3::init_newton()

Initialization
\[ \| \mathbf{v}_B^E \|, \chi, \gamma, l, \lambda, h \]

Conversion

\[
\begin{align*}
[T]_{GE} &= \text{cadigd}(l, \lambda) \\
|\mathbf{s}_{BI}\rangle^E &= |\mathbf{T}_{EG}| |\mathbf{s}_{BI}\rangle^G \\
|\mathbf{v}_B^E\rangle^G &= \text{cart\_from\_pol}(|\mathbf{v}_B^E\rangle^E, \chi, \gamma) \\
[T]_{IE} &= \mathbf{E} \\
[T]_{IG} &= |\mathbf{T}_{IE}| [T]_{EG}^G \\
|\mathbf{v}_B^I\rangle^I &= |\mathbf{T}_{IG}^G| |\mathbf{v}_B^E\rangle^G + |\mathbf{\Omega}_{EI}| |\mathbf{s}_{BI}\rangle^I \\
[T]_{IG} &= \text{matBr}(\chi, \gamma) \\
[T]_{GV} &= |\mathbf{T}|_{IG}^G
\end{align*}
\]

\[ |\mathbf{v}_B^E| \quad \text{Geographic speed} \]
\[ \chi \quad \text{Heading angle} \]
\[ \gamma \quad \text{Flight path angle} \]
\[ l \quad \text{Longitude} \]
\[ \lambda \quad \text{Latitude} \]
\[ h \quad \text{Altitude} \]

\[ |\mathbf{s}_{BI}\rangle^I \quad \text{Inertial position} \]
\[ |\mathbf{v}_B^I\rangle^I \quad \text{Inertial velocity} \]
\[ [T]_{GV}^G \quad \text{T.M. of geographic wrt flight path coor.} \]
\[ [T]_{IG}^G \quad \text{T.M. of inertial wrt geographic coor.} \]
Solving the equations of motion

- Integrating inertial states and converting them to geographic output

**Implemented in:** void Round3::newton(double int_step)

---

**D1 Newton**

**Dynamic Equations**

\[
\begin{align*}
[\frac{dv_f}{dt}]^I &= [T]^{IG} ([T]^{GV} [f_{sp}]^V + [g]^G) \\
[ds_g / dt]^V &= [v_f]^I \\
[sv] &= \text{vfrom}([sv])
\end{align*}
\]

**Kinematic Equations**

\[
\begin{align*}
[\frac{s_{BI}}{dt}]^I &= [T]^{EI} [s_{BI}]^I \\
(i, \lambda, h) &= \text{cadspf}([s_{BI}]^E) \\
[T]^{GE} &= \text{cadige}(i, \lambda) \\
[T]^{GI} &= [T]^{GE} [T]^{EI} \\
[v_B^E]^{IG} &= [T]^{GI} ([v_B^I]^{I} - [\Omega^{EI}]^{I} [s_{BI}]^{I}) \\
([v_B^E], \chi, \gamma) &= \text{pol_from_cart}([v_B^E]^{IG}) \\
[T]^{IG} &= \text{mat2tr}(\chi, \gamma) \\
[T]^{GV} &= [T]^{IG}
\end{align*}
\]

**Output**

\[
\begin{align*}
|v_B^E| \\
\chi \\
\gamma \\
l \\
\lambda \\
h
\end{align*}
\]

\[f_{sp}]^V \text{ Propulsive specific force}
Circular orbiting satellite over the equator

**Input**

- `alt 300000` //Vehicle altitude - m module newton
- `psivgx 90` //Vehicle heading angle - deg module newton
- `thtvgx 0` //Vehicle flight path angle - deg module newton
- `dvbe 7243.2` //Vehicle speed - m/s module newton

**Satellite inertial speed**

\[ |v_B^f| = \sqrt{\frac{GM_\odot}{R_\odot + alt}} \]

**Earth circumferential speed at equator**

\[ |\Delta v| = (R_\odot + alt)\omega^{EI} \]
What you have learned

• The core of the simulation, i.e., the ‘newton’ module, simulates satellite orbits

• Key coordinate systems
  – Inertial, Earth, geographic and flight path angle

• Newton’s equations of motion are integrated in inertial coordinates

• Input/output conversion are from geographic to inertial coordinates and back ♠
Lab2 HandsOn2

... let’s start with a simple class structure and simulation

- **HandsOn2_1** Duplicate the satellite trajectory
- **HandsOn2_2** Add the module ‘seeker’ to the ‘Satellite’ class
- **HandsOn2_3** Find the constructor and destructor of the ‘Satellite’ class
- **HandsOn2_4** Raise the satellite orbit by thrusting
HandsOn2_1  Duplicate the satellite trajectory

1. Load the CRUISE_2 project and run the test case with input file ‘input2_1.asc’

2. Plot the trajectory with the KPLOT-GLOBE program and thus duplicate the circular satellite trajectory over the equator, as shown in Slide 10 of ‘Lab2 Satellite Simulation’

3. What is the deviation in percent from the starting altitude?

4. Reduce and increase the integration step size by an order of magnitude
   - How does the integration step size effect the accuracy? ♣
HandsOn2_2  Add the module ‘seeker’ to the ‘Satellite’ class

- For future usage, the ‘Satellite’ class needs a ‘seeker’ module. Provide two skeleton member functions ‘def_seeker()’ and ‘seeker(...)’ as place holders in the file ‘satellite_modules.cpp’

1. In file ‘satellite_modules.cpp’ add the skeleton of ‘def_seeker()’ and ‘seeker(...)’ following the example of functions ‘def_forces()’ and ‘forces()’, without code inside the function scopes

2. Declare the two new functions in the class ‘Satellite’ of the header file ‘class_hierarchy.hpp’

3. Check your modifications by recompiling ♣
HandsOn2_3 Find the constructor and destructor of the ‘Satellite’ class

1. Locate the constructor of the ‘Satellite’ class
   
   What is the purpose of the parameters?
   
   What memory allocations occur?

2. Locate the destructor of the ‘Satellite’ class

   What action does the destructor perform? ♣
HandsOn2_4  Raise the satellite orbit by thrusting

- In the ‘force’ module a constant thrust force can be applied to the satellite
  1. Augment the input file ‘input2_1.asc’ with the satellite thrust variable
  2. Determine what thrust is required to increase the satellite altitude by 100 km after one orbit
  3. Make a plot with KPLOT-GLOBE ♣
Lab2 Result2

... let’s start with a simple class structure and simulation

• **Result2 1** Duplicate the satellite trajectory
• **Result2 2** Add the module ‘seeker’ to the Satellite class
• **Result2 3** Find the constructor and destructor of the ‘Satellite’ class
• **Result2 4** Raise the satellite orbit by thrusting
Result2_1  Duplicate the satellite trajectory

1 & 2 Self explanatory

3. The deviation form the starting altitude is 0.33%

4. Integration step size
   - increased by order of magnitude the error is 3.3%
   - decreased by order of magnitude the error is 0.031%
Result2_2  Add the module ‘seeker’ to the Satellite class

- For the implementation of the two member functions ‘def_seeker()’ and ‘seeker(…)’, look into the file ‘satellite_modules.cpp’ and ‘class_hierarchy.hpp’ of the CRUISE simulation.
Result2_3 Find the constructor and destructor of the ‘Satellite’ class

1. The constructor is located in file ‘class_functions.cpp’

   The parameter list consists of the pointer to the module array and the size of the module array

   Dynamic memory allocation is made for

   Module variable array ‘satellite’

   Combus index arrays ‘round3_com_ind’ and ‘com_satellite3’

2. The destructor is located in file ‘class_functions.cpp’

   It releases the dynamic memory allocated of the ‘Satellite’ members

Result2_4  Raise the satellite orbit by thrusting

Required thrust to raise satellite by 100km

\[ \text{sat}_\text{thrust} = 1.03 \text{ N} \]
Lab3  Modules

... model the vehicle components

- Modules form the core of the simulation
- Module-variables interface between modules
- Three functions compose the ‘newton’ module
- Modules are called every integration cycle
- Module-variables are of type Variable
- Example code of module ‘newton’
- How to build a module
- Matrix utility operations
- Matrix class and methods
- More examples of Matrix methods
- Utility functions of class Matrix
- What you have learned  ♣

<table>
<thead>
<tr>
<th>Lab</th>
<th>Topics</th>
<th>Simulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>C++ Essentials QuickStart</td>
<td>CRUISE</td>
</tr>
<tr>
<td>2</td>
<td>Classes</td>
<td>CRUISE_2</td>
</tr>
<tr>
<td></td>
<td>Satellite Simulation</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Modules</td>
<td>CRUISE_3</td>
</tr>
<tr>
<td></td>
<td>Target Simulation</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Tables</td>
<td>CRUISE_4</td>
</tr>
<tr>
<td></td>
<td>UAV Uncontrolled</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Events</td>
<td>CRUISE_5</td>
</tr>
<tr>
<td></td>
<td>UAV Controlled</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Polymorphism</td>
<td>CRUISE_6</td>
</tr>
<tr>
<td></td>
<td>UAV Navigation</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>ComBus</td>
<td>CRUISE_7</td>
</tr>
<tr>
<td></td>
<td>UAV Homing</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Architectures</td>
<td>CRUISE</td>
</tr>
<tr>
<td></td>
<td>UAV Netcentric</td>
<td></td>
</tr>
</tbody>
</table>

SCHEDULE
Modules form the core of the simulation

- Modules are public functions (methods) of the classes Round3, Cruise, Satellite and Target.
- The `structure Module` declares the type of the elements in the ‘module_list’ array.
- The module list and calling sequence is established in the ‘input.asc’ file.
- Coding of the modules is greatly simplified by matrix equations.
- The `class Matrix` enables the straightforward writing of matrix equations.
Module-variables interface between modules

- CADAC’s modular structure mirrors the hardware components of an aerospace vehicle
  - A module is a model of a vehicle component
    - Examples: aerodynamics, propulsion, actuator, guidance, control,…
  - Each module consists of at least two functions and possibly four:
    - `def_module()`, `init_module()`, `module()`, `term_module()`
- The calling order of the module is controlled by the input file
- Data between modules is transferred by module-variables
  - Module-variables, being the only allowed interface, are strictly controlled
  - Each vehicle object reserves protected arrays for its module-variables
  - There is a one-to-one relationship between the module-variable name and the array location
  - The file ‘doc.asc’ documents all module-variables
- Module-variables can only be of type `int`, `double`, 3x1 vector, and 3x3 matrix
- Inside a module
  - Module-variables are localized as input from other modules
  - Computations create new module-variables
  - These are loaded onto the object’s array for output
Three functions compose the ‘newton’ module

```cpp
void Round3::def_newton()
Definition of module-variables used in this module

void Round3::init_newton()
Initial calculations

void Round3::newton(double int_step)
Integrations and time-phased calculations
```

Integration loop
Modules are called every integration cycle

Calling sequence established in
`input.asc`

```
MODULES
  environment  def,init,exec
  aerodynamics def,exec
  propulsion   def,init,exec
  forces       def,exec
  newton       def,init,exec
  targeting    def,exec
  seeker       def,exec
  guidance     def,exec
  control      def,exec
  intercept    def,exec
END
```

```
struct Module
{
  string name;
  string definition;
  string initialization;
  string execution;
  string termination;
};
```

```
void order_modules(fstream &input,int &num,Module *module_list)
```

```
//Every integration step, module is called in `execute(…)`
if(module_list[j].name=="newton")
  vehicle_list[i]->newton(int_step);
```

```
//Module is initialized in `main()`
if((module_list[j].name=="newton")&&(module_list[j].initialization=="init"))
  vehicle_list[i]->init_newton();
```

```
//Vehicle’s constructor calls module to define module-variables
if((module_list[j].name=="newton")&&(module_list[j].definition=="def"))
  def_newton();
```

```
//Loading modules into `module_list`
```
Module-variables are of type \texttt{Variable}

\begin{verbatim}
Defining module-variables in \texttt{void Round3::def_newton()}

\texttt{round3[19].init("lonx",0,"Vehicle longitude - deg","newton","
round3[20].init("latx",0,"Vehicle latitude - deg","newton","
round3[21].init("alt",0,"Vehicle altitude - m","newton","out";

Documenting module-variables in \texttt{doc.asc}

\begin{verbatim}
<table>
<thead>
<tr>
<th>LOC</th>
<th>NAME</th>
<th>DEFINITION</th>
<th>MODULE</th>
<th>PURPOSE</th>
<th>OUTPUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td>lonx</td>
<td>Vehicle longitude - deg</td>
<td>newton</td>
<td>diag</td>
<td>scrn,plot,com</td>
</tr>
<tr>
<td>20</td>
<td>latx</td>
<td>Vehicle latitude - deg</td>
<td>newton</td>
<td>diag</td>
<td>scrn,plot,com</td>
</tr>
<tr>
<td>21</td>
<td>alt</td>
<td>Vehicle altitude - m</td>
<td>newton</td>
<td>out</td>
<td>scrn,plot,com</td>
</tr>
</tbody>
</table>
\end{verbatim}

Using module-variables

\begin{verbatim}
void Round3::newton(double int_step)
{
    //localizing module-variables
    //input from other modules
    double time=round3[0].real();
    Matrix FSPV=round3[10].vec();
    ...
    //--------------------------
    ...
    //loading module-variables
    //state variables
    round3[35].gets_vec(SBII);
    round3[36].gets_vec(VBII);
    ...
}
\end{verbatim}
\end{verbatim}
Example code of module ‘newton’

```cpp
void Round3::def_newton()
{
    //Definition of module-variables
    round3[19].init("lonx",0,"Vehicle longitude-deg","newton","diag","scrn,plot,com");
    round3[20].init("latx",0,"Vehicle latitude-deg","newton","diag","scrn,plot,com");
    round3[21].init("alt",0,"Vehicle altitude-m","newton","out","scrn,plot,com");
    ...
}

void Round3::init_newton()
{
    //localizing module-variables
    //input from other modules
    double time=round3[0].real();
    Matrix FSPV=round3[10].vec();
    double grav=round3[11].real();
    Matrix WEII=round3[27].mat();
    //state variables
    Matrix SBII=round3[35].vec();
    Matrix VBII=round3[36].vec();
    Matrix ABII=round3[37].vec();
    ...

    //building gravitational vector in geographic coordinates
    GRAV.assign_loc(2,0,grav);

    //integrating inertial state variables
    ABII_NEW=TIG*((TGV*FSPV)+GRAV);
    VBII_NEW=integrate(ABII_NEW,ABII,VBII,int_step);
    SBII=integrate(VBII_NEW,VBII,SBII,int_step);
    ABII=ABII_NEW;VBII=VBII_NEW;
    ...
}

void Round3::newton(double int_step)
{
    //loading module-variables
    //state variables
    round3[35].gets_vec(SBII);
    round3[36].gets_vec(VBII);
    round3[37].gets_vec(ABII);
    ...
}
```

### Definition of module-variables in array ‘round3[ ]’, a member of class ‘Round3’
- **name, value, definition, module, type, out to: screen, plot file, combus**

### Getting module-variables from ‘round3[ ]’ array
- **Getting module-variables from ‘round3[ ]’ array**

### Computations
- **Computations**

### Loading module-variables onto ‘round3[ ]’ array
- **Loading module-variables onto ‘round3[ ]’ array**
How to build a module

1. Start with sample module as template
2. Reserve contiguous slots in the module-array
3. In ‘def_module’ categorize module-variables as
   "data" input data values
   "init" initialized in init_module
   "state" state variables for integration
   "out" output to other modules
   "diag" diagnostic, not used by other modules
   "save" value saved for next cycle
4. In ‘module’ localize module-variables by category
   //input data
   //restore saved values
   //state variables
   //input from other modules
5. Insert computations
6. In ‘module’ load module-variables by category
   //state variables
   //saving values
   //output to other modules
   //diagnostics
7. Incorporate new module into simulation frame

---

**Definition module** is called once, before ‘input.asc’
Defines all module-variables
Directs variables to output streams

**Initialization module** is called once, after ‘input.asc’
Executes initial calculations

**Module** is called once every integration step
Localizes module-variables
Executes computations every integration step
Integrates state variables
Calls other subordinate functions
Calls utility subroutines
Performs table look-up
Loads module-variables

**Watch out**
Initialize all local variables
Don’t call other module’s subordinate functions

---

Example: Newton Module

```c++
#define GRAV 9.81
#define PI 3.14159

//Definition of newton module-variables
//Member function of class 'Round3'
//Module-variable locations are assigned
// to round3[17-39]
void Round3::def_newton()
{
    //Definition of module-variables
    round3[19].init("lonx",0,"Vehicle longitude-
    round3[20].init("latx",0,"Vehicle latitude-deg
    round3[21].init("alt",0,"Vehicle altitude-m","

    //localizing module-variables
    round3[19].real();
    round3[10].vec();
    round3[11].real();
    round3[27].mat();

    //state variables
    round3[35].vec();
    round3[36].vec();
    round3[37].vec();

    //building gravitational vector in geographic
    GRAV.assign_loc(2,0,grav);

    //integrating inertial state variables
    ABII_NEW=TIG* (TGV*FSPV)+GRAV;
    VBII_NEW=integrate(ABII_NEW,ABIII,VBII,int_step);
    SBII=integrate(VBII_NEW,VBIII,SBII,int_step);
    ABIII=ABII_NEW;
    VBIII=VBII_NEW;

    //loading module-variables
    round3[35].gets_vec(SBII);
    round3[36].gets_vec(VBII);
    round3[37].gets_vec(ABII);
    ...
}
```
Matrix utility operations

- Source code should be programmed in matrices as much as possible
  - Compact code
  - Avoids errors
- Requirements of flight simulations
  - Mostly 3x1 and 3x3 matrices, some of higher dimensions (Kalman filters)
  - Elements of matrices are of type `double`
- Class `Matrix` instantiate a pointer to the matrix `*pbody` and initializes the elements to zero
- Examples from module `Target::forces()`

```cpp
//Coriolis acceleration in V-coordinates
WEIG=TGE*WEII*TEG;
CORIO_V=TVG*WEIG*VBEG*2;
//centrifugal acceleration in V-coordinates
CENTR_V=TVG*WEIG*WEIG*TGI*SBII;
```

- Special features
  - All matrix manipulations are carried out in pointer arithmetic
  - Creating a matrix returns `*this` of the initialized matrix
  - Usage of `copy constructor` for the matrix assignment operator
  - Overloaded offset operator `[]`
Matrix class and methods

- **Matrix class is global in scope**
- **Private methods declare size and pointer of array**
- **Public methods are the constructors/destructor and utility functions**

```cpp
class Matrix
{
private:
    // number of rows
    int num_row;
    // number of columns
    int num_col;
    // total number of elements
    int num_elem;
    // pointer to array
    double *pbody;
public:
    // default constructors
    Matrix();
    // overloaded constructor
    Matrix(int row_size, int col_size);
    // copy constructor
    Matrix(const Matrix &MAT);
    // destructor
    ~Matrix();

    // Absolute value of vector
    // Example: value = VEC.absolute();
    double absolute()
    { double ret=0;
      for(int i=0; i<num_elem; i++)
          ret += (*(pbody+i))*(*(pbody+i));
      ret = sqrt(ret);
      return ret;
    }
};
```
More examples of Matrix methods

double Matrix::get_loc(const int &r, const int &c)
{
    if((r<num_row)&&(c<num_col))
        return *(pbody+r*num_col+c);
    else
        {cout<<"*** Error: invalid matrix loc. 'Matrix::get_loc()' *** ";exit(1);} 
}

Matrix & Matrix::identity()
{
    if (num_row==num_col)
    {
        for(int r=0;r<num_row;r++)
            *(pbody+r*num_row+r)=1;
    }
    else
        {cout<<"*** Error: matrix not square 'Matrix::identiy()' *** ";exit(1);} 
        return *this;
}
Utility functions of class Matrix

Member functions

```c++
void print();
double absolute();
Matrix adjoint();
void assign_loc(...);
Matrix & build_mat33(...);
Matrix & cart_from_pol(...);
Matrix col_vec(const int &col);
double determinant();
Matrix diamat_vec();
Matrix diavec_mat();
void dimension(int row, int col);
Matrix ellipse();
int get_cols();
int get_index(const int &row, const int &col);
double get_loc(const int &r, const int &c);
double * get_pbody();
int get_rows();
Matrix & identity();
Matrix inverse();
Matrix mat33_vec9();
Matrix & ones();
bool operator!=(const Matrix &B);
Matrix operator*(const double &b);
Matrix operator*(const Matrix &B);
Matrix & operator*=(const double &b);
Matrix & operator*=(const Matrix &B);
Matrix operator+(const double &b);
Matrix operator+(const Matrix &B);
Matrix & operator+=(const double &b);
Matrix & operator+=(const Matrix &B);
Matrix operator-(const double &b);
Matrix operator-(const Matrix &B);
Matrix & operator-=(const double &b);
Matrix & operator-=(const Matrix &B);
Matrix & operator=(const Matrix &B);
bool operator==(const Matrix &B);
double & operator[](const int &r);
Matrix operator~();
```

Overloaded operators

```c++
bool operator!=(const Matrix &B);
Matrix operator*(const double &b);
Matrix operator*(const Matrix &B);
Matrix & operator*=(const double &b);
Matrix & operator*=(const Matrix &B);
Matrix operator+(const double &b);
Matrix operator+(const Matrix &B);
Matrix & operator+=(const double &b);
Matrix & operator+=(const Matrix &B);
Matrix operator-(const double &b);
Matrix operator-(const Matrix &B);
Matrix & operator-=(const double &b);
Matrix & operator-=(const Matrix &B);
Matrix & operator=(const Matrix &B);
bool operator==(const Matrix &B);
double & operator[](const int &r);
Matrix operator~();
```
What you have learned

- Modules mirror vehicle components
- Module loading and calling sequence is established in the input file
- Communication between modules is only by module-variables
- The value of module-variables is stored in protected arrays of the vehicle classes
- Computations in modules are carried out in matrix form, where applicable
- Matrix manipulations use methods of the class Matrix ♣

SCHEDULE
Lab3 Target Simulation

... simulates vehicles moving on the ground

- Features of target simulation
- Class hierarchy of target simulation
- Equations of motions are compensated
- Vehicles on the Earth’s surface experience apparent forces
- Calculating the acceleration compensation
- Straight target trajectory
- What you have learned ♣

<table>
<thead>
<tr>
<th>Lab</th>
<th>Topics</th>
<th>Simulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>C++ Essentials QuickStart</td>
<td>CRUISE</td>
</tr>
<tr>
<td>2</td>
<td>Classes Satellite Simulation</td>
<td>CRUISE_2</td>
</tr>
<tr>
<td>3</td>
<td>Modules Target Simulation</td>
<td>CRUISE_3</td>
</tr>
<tr>
<td>4</td>
<td>Tables UAV Uncontrolled</td>
<td>CRUISE_4</td>
</tr>
<tr>
<td>5</td>
<td>Events UAV Controlled</td>
<td>CRUISE_5</td>
</tr>
<tr>
<td>6</td>
<td>Polymorphism UAV Navigation</td>
<td>CRUISE_6</td>
</tr>
<tr>
<td>7</td>
<td>ComBus UAV Homing</td>
<td>CRUISE_7</td>
</tr>
<tr>
<td>8</td>
<td>Architectures UAV Netcentric</td>
<td>CRUISE</td>
</tr>
</tbody>
</table>
Features of target simulation

• Targets move on the surface of the round, rotating Earth
  – Fixed
  – Translating
  – Accelerating

• Coordinates: longitude, latitude, altitude

• Newton’s equations are compensated with apparent forces ♣
Class hierarchy of target simulation

Abstract base class

Derived class

Derived class

Virtual functions

newton()
environment()

forces()
intercept()

Module-Variable arrays

environment() forces() newton() intercept()

Modular structure

TARGET3
Equations of motions are compensated

- Newton’s Law governs the motion of the target
- Compensating forces are applied to constrain the target to the surface of the Earth

Newton’s Law with compensation

\[ ma_{\text{comp}} = mD^I v^I_B = ma_{\text{comp}} + mg \]

Matrix equations programmed in ‘newton’ module

\[
[dv^I_B / dt]^I = [T]^{GI} ([T]^{VG} [a_{\text{comp}}]^V + [g]^G)
\]

\[ I \quad \text{Inertial coordinates} \]
\[ G \quad \text{Geographic coordinates} \]
\[ V \quad \text{Flight path coordinates} \]
You have enjoyed ATI's preview of

Aerospace Simulations In C++

Please post your comments and questions to our blog:

http://www.aticourses.com/wordpress-2.7/weblog1/

Sign-up for ATI's monthly Course Schedule Updates:

http://www.aticourses.com/email_signup_page.html