UA FDL report 99S02B, May 15, 2000.
Prepared for the FAA Technical Center, Atlantic City, NJ
Under Cooperative Agreement 98-G-018, Task E.

Helicopter Rotor Safety Issues

Stage 1 Report

Revision B

Prepared by

Dr. Amnon Katz

Kenneth S. Graham

Kyle Palmer

The Flight Dynamics Laboratory

University of Alabama

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

EXECUTIVE SUMMARY

The University of Alabama has an ongoing effort in modeling of helicopter rotors and the study of rotor dynamics. The starting point of the present project was is the original true blade model, *Bladehelo*, previously developed at the University of Alabama Flight Dynamics Laboratory (UA FDL) and implemented in its real time simulator. *Bladehelo* employs rigorous equations of motion. It uses no prescribed motions and no small angle approximations. The project plan was to use *Bladehelo*, following certain upgrades, to assess the dynamic performance and the functional equivalence of composite blades to metal blades they replace, with a view to establishing certification requirements.

The project is structured in three stages: Stage 1 consists of upgrades to *Bladehelo.* Stage 2 is concerned with validation of the upgraded model, and Stage 3 addresses the structural issues. The present report covers Stage 1 as carried out between 16 May 1998 and 15 May 2000.

Two major modeling upgrades were implemented: flexible blades and a high fidelity wake. Flexible blades are modeled as articulated with adjacent segments joined by spring loaded hinges. Rather than model the segments and hinges specifically, a technique called *structured modeling (SM)* was invoked, which employs generic, model independent code, and allows the details of the model to be defined as input. SM, in turn, is based on the method of *Global Recursive Dynamics* (GRD). Both SM and GRD were developed at the UA FDL prior to the current project. In Stage 1, the pre-existing *Bladehelo* and *SM* codes were merged to create an upgraded *Bladehelo*, which was verified against the previous version and then applied to the modeling of flexible blades.

The wake upgrade replaces the uniform inflow assumed in the pre-existing *Bladehelo* with a dynamic wake generated by a lattice of vortices created by the rotor blades. A trailing vortex is appropriate wherever the circulation about the blade changes, in our computational scheme, this occurs at both ends of each blade element. A shed vortex is appropriate whenever the circulation about the element changes, i.e. every integration step. The wake model creates and propagates the vortices in real time based on the flow field they generate. Limitations of computational throughput dictated that only tip and hub vortices be kept and that vortices be shed at a rate lower than the computational frame rate.

Both upgrades required additional computational resources, especially the wake model. For this purpose, the simulation host at the UA FDL was enhanced with four units of four processors each. The units are designated *Monster0* through *Monster3.* The Monsters are interconnected by SCRAMNet. Multiprocessing proceeds in two stages, spreading to the four processors within each Monster and across the four monsters. Both stages were successfully implemented. Use of the 16 processors yielded a throughput improvement by a factor of 13.4. This allowed each of the upgrades mentioned above, as well as both of them together, to run in real time.

1. PROJECT OVERVIEW

1.1 ROTOR MODELING AT THE UNIVERSITY OF ALABAMA

The University of Alabama Flight Dynamics Laboratory (UA FDL) has been simulating helicopters, and supplying helicopter models to US Army Research Institute, since 1981 (for a recent case, see [1]). The real-time, man-in-the-loop, simulator was developed locally and is continuously being upgraded. The hardware, software, architecture, and math models are all developed in-house by a team of faculty and graduate students.

Since 1994, the UA rotor model has been *Bladehelo* [2] – a true blade model that integrates the flapping motion of individual blades and the shaft rotation. *Bladehelo* employs rigorous equations of motion. There are no prescribed motions, and no small angle approximations. The full non-linear equations of motion are integrated numerically. UA FDL practices physically based simulation. *Bladehelo* is derived from first principles. When the model does not agree with reality, it is not "tuned" and "tweaked" – rather, the discrepancy is studied until its physical source is understood.

This approach led to the discovery of a previously unknown adverse control effect that results from lack of lag relief [3]. In the case of two-bladed rotors, the lessons learned applied only to the method of simulating a rotor. In the case of multi-bladed rigid-in-plane rotors, an adverse control effect is actually there.

The adverse control moment is a special case of the phenomenon of residual shaft bending, also discovered at UA FDL [4]. Some advanced simulators may already represent residual bending correctly, but the phenomenon is not generally known. To first order in disk inclination, residual bending is a pure cross coupling effect. It causes a conventional helicopter to respond slightly to the left of the direction of tip plane tilt. The adverse control effect is a subtle third order effect, which grows faster than first order and becomes significant for large rotor tilt.

Typical real time statistics at the beginning of the present project were as follows: blade motion was computed at 330Hz using fourth order Runge-Kutta integration (equivalent to 1320Hz using fourth order Adams-Bashforth). The body motion was integrated at 80Hz. Transport delay (control to visual) was 70-80ms. *Bladehelo* can also be exercised off line.

Before the current project, *Bladehelo* was subject to two major limitations: 1. rigid blades and 2. uniform inflow. Removal of these limitations is the subject of Stage 1 of the present project.

1.2 FLEXIBLE BLADES

The UA FDL approach to modeling flexible blades is based on representing the rotor as an articulated tree with spring loaded hinges. The treatment of the articulated tree draws on two main elements, both developed locally:

a. A rigorous Global Recursive Dynamics (GRD) of articulated tree structures [5].
b. Structured Modeling (SM).

Tree structures are well suited for recursive computational algorithms. *Local* recursive formulations, where rigid body equations are applied to each part, are intuitively simple. The placement and current motion of each part is derived from the motion of its *ancestors,* while the loads applied are recursively accumulated from its *descendants*.

The value of the local recursive approach is somewhat limited, because forces accumulated and sorted out from complete branches are applied to single parts. In large trees, the mass of these parts tends to be small. Any inaccuracy in sorting out the forces results in the application of large forces to light parts and gives rise to computational instabilities. When the tree is a model for a flexible body, the more detailed the model, the smaller the individual parts, and the more severe the computational instabilities. The local formulation cannot admit dummy parts.

All of these difficulties disappear in the *global* recursive formulation (GRD). The global approach applies the forces and moments accumulated over a branch of the tree to the whole branch, treated as a rigid body with the instantaneous configuration of the branch. It then recursively applies the correction due to motion within the branch. In this formulation, the masses and moments of inertia that appear do not diminish as the model becomes more detailed; dummy parts are possible; conservation laws for the whole system are manifestly satisfied at each recursion step. Correct CG motion is a natural spin-off. The formalism of global recursive dynamics has been developed over the past year prior to the start of the present project [5].

Structured modeling (SM) uses generic recursive code (GRC) with the details of the system of interest represented in a data file. The recursive code computes the inertial effects based on the geometry of the tree. It contains hooks for specific routines that compute loads, e.g., aerodynamic loads and for tables, e.g., tables of aerodynamic coefficients that can be specific to each segment. It is the inertial dynamic effects that follow from the detailed geometry that are most complex. Routines for computing aerodynamic loads on a blade segment can be standard and shared by all segments, with tables of coefficients possibly varying from one segment to another. SM makes it possible to compute any system modeled as a tree virtually without coding and without debugging. Since coding and, in particular, debugging always requires substantial effort, the utility of structured modeling stands out. Structured modeling is superior to schemes for automatic generation of code in that it is not affected by changes and upgrades in compilers, over which the end user has little control.

The concept of structured modeling and the formalism for recursively handling tree-like data structures was developed and demonstrated by the UA FDL in the course of a previous project [6]. The *SM* code that implements GRD had been developed and tested prior to the beginning of the current project (see reference 5, Appendix A). Note that we use SM (in upright characters) to designate the general method of Structured modeling. *SM* (in italics) designates the code implementing SM. The statement of work of the current project called for

a. Merging of the *SM* code into *Bladehelo.*
b. Verification of the new *Bladehelo* against the previous version.
c. Application of the upgraded *Bladehelo* to the modeling of flexible blades.

The above three subtasks, together, form Task 1 of the current project and are described in chapter 2 of this report.

1.3 UNSTEADY AERODYNAMICS (DYNAMICALLY INDUCED WAKE)

Every airborne vehicle, by resting its weight on air, causes a downwash. The vehicle flies through its own downwash and is, in effect, subject to half the downwash velocity. A down-flow is detrimental to performance and requires additional power to overcome, which is known as induced power (also related to induced drag). Induced drag and power are more significant, the slower the aircraft. Since helicopters fly down to zero airspeed, induced drag and power are most significant for a helicopter rotor.

The pre-existing *Bladehelo* models the downwash as a uniform inflow, whose magnitude is determined by conservation of momentum ("momentum theory"). This method of modeling assures a correct value for the average inflow and captures the essence of the physical effect during steady flight. Uniform inflow is the most efficient, and is a design goal. The uniform inflow model is inherently unconservative and underestimates the induced drag and power. When maneuvering, the uniform inflow model responds too quickly and too uniformly. It was pointed out by Prouty [7] that virtually all simulators compute the cross-coupling in transient maneuvers incorrectly. Rosen and Isser [8], [9] suggested that the actual observed effect is caused by distortion of the wake in transient maneuvers. These authors suggested that a wake modeled as a vortex lattice would predict transients correctly and reported that they had found it to be so.

In the current project, the ideas of [8], [9] were adapted to real time simulation. In our version, the vortex lattice is related to a lifting line, rather than lifting surface, model of the blade elements. The lift generated by each blade element is still determined from local airspeed and angle of attack, using tables of lift coefficients. A trailing vortex is generated at the two ends of each blade element. A shed vortex is generated every integration step, when the lift on the blade element is recomputed. A model along these lines was created, coded and exercised off-line. The model accounts for nodes and segments of the vortex lattice. The lattice is created and propagated in real time. True to the *Bladehelo* tradition, motion of the vortex segments is not prescribed, but rather computed based on the flow that the vortex lattice generates. In real time, limitations of computational throughput dictated that only tip and hub vortices be kept and that vortices be shed at a rate lower than the computational frame rate.

This reduced version of the dynamic wake was successfully run in real time, first interacting with the traditional *Bladehelo,* and then interacting with the SM version representing flexible blades. Time available did not allow an assessment of the effect of modeling upgrades on the fidelity with which handling qualities were represented.

Modeling of the dynamic wake makes Task 2 of the present project and is described in chapter 3 of this report.

1.4 ARCHITECTURE AND INTEGRATION.

The pre-existing *Bladehelo* ran on a single Pentium Pro processor running as 16-bit code in a DOS environment. Typically, the blade motion was computed at 330 frames per second, using fourth order Runge-Kutta integration (RK4). Each fourth order Runge-Kutta step is equivalent in terms of computational effort and resulting accuracy to four fourth order Adams-Bashforth (AB4) steps. This computational scheme provided more than adequate precision. It was clear, however, that the introduction of flexible blades, and even more so, of the dynamic wake, would make additional demands on computational resources.

It was expected that a transition to 32-bit computing would increase the speed of the computations. Certain benchmarks suggested that a speedup by a factor of almost three could be achieved. These benchmarks were unconservative in that they did not allow for the communications of the simulation code with visual displays and cockpit controls. It was the issue of communications that was holding up the conversion. The UA FDL had previously developed its own super-efficient interrupt driven communications protocols aimed specifically at real-time simulation. These protocols could not be accommodated in the newer operating systems that supported 32-bit computing.

When 32-bit computing was accomplished (under Windows 95 and Windows NT), it was found that the bottleneck in realizing a speed-up was actually in the 32-bit compilers that were available. The favorable benchmarks had been produced with the Intel optimized compiler that had been packaged with Borland C++ version 5.01. That compiler proved too sensitive and unstable for the *Bladehelo* development work. The regular Borland compilers yielded a speedup of only 25% to 35%. Eventually, the Code-Warrior compiler and an Intel compiler compatible with Microsoft C++ made it possible to realize a speed-up by a factor of nearly two. An additional factor of two was realized from doubling the CPU clock speed, as 400MHz Pentium processors became available

Even, so, it was clear from the beginning that additional computing hardware would be required. After a survey of available systems, we selected four server-type computers running four Pentium II processors each. These four computers, purchased with project funds, were named *Monster0, Monster1, Monster2,* and *Monster3*. The four "monsters" were interconnected by SCRAMNet (also purchased with project funds). The SCRAMNet also connected the four "monsters" to *Janus*, a two-processor computer located by the simulator cab and serving as simulation host. The "monsters" provide a nominal increase in compute power by a factor of 16. This estimate is, of course, unconservative, since it makes no allowance for the overhead involved in multiprocessing. An advantage by a factor of 13.4 was actually realized.

Task 3 of Stage 1 called for the integration of the upgraded *Bladehelo* code, supporting flexible blades, with the dynamic wake model. This inevitably involved the adaptation of *Bladehelo* and wake code to multiprocessing and the creation of the required interfaces. The wake model was actually spread to all 16 processors with each computing the contribution of its share of vortex segments to the induced velocity. The 16 processors produced a computational advantage by a factor of 13.4. The rotor model is less suitable for multiprocessing. Computations for each of

the several blades were allocated to separate processors. With three blades and four processors available, an advantage by a factor of only 1.66 was realized.

Chapter 4 of this report covers the architectural and interface issues addressed above. Additional work on architecture and interfacing, not funded by the present project, was in progress in parallel with Stage 1. The main accomplishments were:

a. The interface of *Bladehelo* with the image generators (Intergraph workstations with original software developed at the UA FDL under a previous contract [10]) was enhanced. The image generators now return terrain elevation and slope, which permits the simulated helicopter to land on elevated surfaces and on sloping surfaces. A large building with a sloping roof was added to the database. Slope landings (up to 11 degrees) can now be practiced.
b. The potentiometers reading pilot controls have been replaced and their wiring redesigned. This eliminated a long-standing problem of noise in control inputs. This problem had previously forced a reduction in control gains, so as not to amplify the noise. With the noise problem eliminated, control sensitivity was increased to the aircraft level.
c. A method was developed to fractalize lines. This was applied to the river bank in the data base with very pleasing results. Previously, the river had looked like a canal.
d. A view of the rotor disk was added in the pilot's visual window.

1.5 THE OVERALL PROJECT

Beyond, Stage 1, the project was designed to assist the FAA in addressing safety and certification issues.

The Rotorcraft Directorate has received many applications from small modifiers (primarily via certification field offices) to replace the all-metal tail rotor blades on older rotorcraft models (such as the Bell Helicopter Textron UH-1H) with “identical” composite designs. Most of these proposed modifications have been terminated economically when it was realized by the applicant that the definition of “identical” was not straightforward. Most of the proposed replacements would have identical external dimensions and would have identical spanwise stiffness (either on a gross basis or on an incremental, span station-by-span station basis).

“Identicality” of external dimensions and of span-wise stiffness of replacement composite blade designs is not hard to determine. The problem of reaching a comprehensive, three dimensional determination of “identicality” for the composite replacement blades is much more difficult. This is so because of considerations such as:

(1) The unique design of each composite laminate;
(2) The chord-wise stiffness effects of the composite replacement;
(3) The three dimensional response of the composite replacement under dynamic loading conditions; and,
(4) Any other consideration that is necessary to make replacement composite blades form, fit, and functionally equivalent (i.e., interchangeable).

The upgrades of Stage 1 were to be verified in Stage 2. This was to involve an instrumented aircraft that would be simulated, permitting comparison of flight data to simulation results. The UA FDL has moved to provide this test aircraft. A helicopter kit – a Revolution Mini500B – was acquired with funds independent of this project. The kit helicopter was constructed by seniors of the Aerospace program at the University of Alabama under supervision of professors. The work started in January 1999. The aircraft was certificated on 6 April 2000 and is, at the time of this writing, undergoing initial flight testing. It is planned that Stage 2 of this project will see the instrumentation of this aircraft, and its use in validating *Bladehelo*.

Following the modeling upgrades of Stage 1, and their validation in Stage 2, Stage 3 is to address the above certification issues. At this writing, Stage 2 and Stage 3 have not been funded.

2. STRUCTURED MODELING AND FLEXIBLE BLADES

2.1 IMPLEMENTING STRUCTURED MODELING (SM) IN *BLADEHELO.*

At the beginning of the present project, *Bladehelo* and *SM* were both in existence as separate codes. The history and general characteristics of *Bladehelo* have been covered in chapter 1. More detailed information is available in references [1], [2]. *Bladehelo* requires the following input files:

a. `Belbatch`, identifies the other input files,
b. `Belclcd,` is a table of aerodynamic coefficients for the blades,
c. `Belelmnt`, defines each blade as a sequence of blade elements,
d. `Groundc`, contains inputs for the ground contact model,
e. `Belinput`, is a general input file.

The general input file contains data on the rotor configuration as well as initial conditions and program management options.

The independent *SM* code read as input a small file identifying step size and run duration. The configuration of the articulated tree system was identified in a `*.SM` file. At initialization, this code set up the articulated tree system as a recursive tree of parts (user-defined type `P`). Each part identified its first descendant by a member pointer `parts` and its next sibling by a member pointer `next`. See, for example, figure 2-2, or for more fully developed tree, figure 2-9. This method of constructing a tree with just two pointers per part is original with the UA FDL and was first introduced in reference 6. The pointer to the root of the tree was `LLP` (for "Linked List of Parts"). The dynamics of the system was exercised by the call `accumulate(LLP).` The function `accumulate()`takes a pointer to a part as a parameter. It places the children of the part, determines the forces and moments on the children by calling `accumulate()` for them, and advances each child one time step. In this way, the call to `accumulate()` triggers a sequence of calls, which go down the tree recursively, placing all parts, then up the tree, advancing each by one time step. This was repeated in steps covering a specified time period, and the results stored for later display and analysis. The *SM* code had been validated by running test cases with known results [5].

The operation of `accumulate()` may be summarized by the following pseudo-code:

```
accumulate(P *Part)
{
Item = Part->parts;
while(Item){   CODE A //Place Item relative to Part
      accumulate(Item); // Recursion step.  Determine Item's global properties
      CODE B //Determine the angular acceleration of Item's branch
      CODE C //One integration step (later moved out from this position, see text)
      CODE D //Accumulate Item's contribution to part's global quantities.
      Item=Item->next;} //End of "while(Item)" loop.
```

```
CODE E // call pointed function to determine local loads applied directly to Part
       // and add them to global loads accumulated from children.}
```

Bladehelo, after initialization, goes into an infinite loop, stepping the rotor and body in time, while communicating with the cab and the instrument and visual displays. The top level view of the operation of *Bladehelo* is offered in figure 2-1. The loop shown advances the state of the body one step, each time around. The time step in question is a "large step," or body step, which typically amounts to one sixteenth of a rotor revolution. The rotor forces are integrated by finer steps. Thus, the subroutine `rotorforces()`contains a loop over "small steps," or rotor steps. Typically, each large step is broken into four Runge-Kutta (RK4) small steps or 16 Adams-Bashforth (AB4) steps.

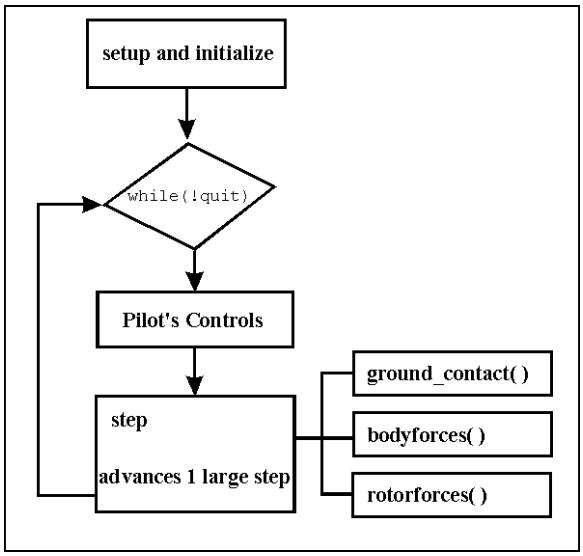


Figure 2-1: TOP LEVEL FLOW OF PRE-EXIXTING *BLADEHELO*

It is here that the initial transition of *Bladehelo* to SM was effected. A merged code called *B0* was created, which computed the rotor motion and resulting forces and moments using *SM*. Everything else in *Bladehelo* was left unchanged. The transition to *B0* was a concise and easy step, and *B0* lent itself well to the task of verification. *SM*, implemented along the lines of reference 5, applies the global approach to the body and computes its motion on the basis of the balance of linear and angular momentum of the whole vehicle. *Bladehelo* applies a local approach to the body and bases its computations on the balance of linear and angular momentum of the body by itself. The study of these issues was deferred. The initial verification was limited to the comparison of rotor forces and moments as computed by the explicit code in bladehelo and the generic code in *SM*.

B0 was created by merging *SM* into *Bladehelo*. The files containing both codes were compiled together. Function `main()` of *SM* was removed, and its initialization functions absorbed in function `main()` of *Bladehelo*. The *SM* input file was renamed `rotor.sm` and added to the list of input files of *Bladehelo*. It was read in and interpreted by the procedure

`Import_Object()`taken over from *SM*. This way, the computer structure containing the articulated tree information was created. The routine `initialize_initsm_settings()` along with the required data members were add to the `P` (part) class. The definition of the rotor was now contained in `rotor.sm`. Inputs such as rotor radius and number of blades were still being read from `belinput`, but no longer used.

The function `Rotorforces()` contained a loop

```
for (k=0; k<small_steps_per_large; k++){}
```

Inside that loop was a call to a function named `Runge_K` which performed the rotor step integration (using the RK4 method). This call was now removed and replaced with the call to `accumulate(LLP)`.

A number of other changes were also necessary:

a. Time steps. The pre-existing *Bladehelo* defined its steps as fractions of the rotor revolution. It used rotor azimuth, rather than time, as the independent variable. SM is generic and knows nothing about rotors and their revolutions. It was necessary to return to time as the independent variable. The input file `Belinput` specifies the number of "large steps" per revolution and the number of "small steps per large". These inputs were retained and used to compute actual time steps based on the nominal rate of rotation of the rotor, which is also specified in `Belinput.`
b. Method of integration. *Bladehelo* used fourth order Runge-Kutta integration (RK4) for the "small steps." This method has the advantage that it is self-starting. On the other hand, it requires values of the derivatives at fictitious points in state space. This presented no problem to *Bladehelo*, because control inputs are held constant during each "large step" in any case. For SM, in its original form, the fictitious points would proliferate recursively. For this reason, RK4 was replaced by fourth order Adams-Bashforth (AB4) as the integration method for the small steps. AB4 is not self-starting, in that it requires data from three previous time steps. AB4 does not achieve its full accuracy until after a few steps have elapsed. In man-in-the-loop simulation, this initialization effect quickly washes away, and is of no concern. The significance of number of steps is somewhat different for RK4 and AB4. Each RK4 step is as labor intensive as four AB4 steps and produces comparable precision. This way, running *Bladehelo* with the step scheme 16x4 (16 "large steps" per revolution and four "small steps" per large step) is as labor intensive as running B0 in the scheme 16x16. The two also proved equally accurate in the verification, see section 2.3.
c. Dynamic routines. *SM* contains generic code that automatically accounts for the inertia of the articulated tree of parts. This is SM's strength, since the inertial computations are quite intricate and would require considerable effort to formulate and code specifically for each model. Parts of the tree are also subject to forces and moments other than inertial. For that purpose, SM supports pointers to dynamic routines that compute those loads and to tables of data to be used in the computation. This is how aerodynamic forces on rotor blades are treated by *SM*. Certain parts of the `Runge-K()` routine, the ones computing aerodynamic loads, were extracted and

reworked into a blade dynamic routine `BH_blade()`. The tables of aerodynamic coefficients in file `belclcd` and the blade element data in file `belelmnt` were taken over as data for `BH_blade()`, which also used global variables describing cyclic and collective control inputs. The dynamic routine `BH_blade()`, together with other dynamic routines, was placed in a source file `dynamic1.cpp`. Pointers to `BH_blade()` and its data were placed in the input file `rotor.sm`. Other dynamic routines provided for application of engine torque to the rotor and reaction torque to the body.

d. Gravity. The original *SM* made no specific provision for gravity. Initially, the weight of the blades was computed explicitly in `BH_blade()`. However, gravity is also generic in nature. The forces and moments due to gravity were eventually absorbed into `accumulate()`.

e. Part identification and data display. The pre-existing *Bladehelo*, when operating off line, displays certain data, including the rotational rate and acceleration of the rotor and the flapping angle of each blade at selected azimuths. *SM* knows nothing of rotors and of blades. In order to create comparable displays for analysis and for the verification, it was necessary to deviate from the generic purity of *SM* and to identify certain parts by function. The following part pointers were introduced:

```
P* hub    = LLP->parts;
P* blade0 = hub->parts;
P* blade1 = blade0->next;
```

`blade0` and `blade1` were used only to capture the flapping angles of the blades. `hub` was used also to capture the azimuth angle as `hub->incidence` and the rotor forces and moments as `hub->g_force` and `hub->g_moment`. (The prefix `g_` stands for "global," namely, including forces and moments acting on the blades).

2.2 VERIFICATION METHOD AND INPUTS

The *B0* code was verified against the pre-existing *Bladehelo*, which we identify in the following as *BH*. We reserve the name *Bladehelo* for the ever-evolving code, which at this writing includes SM capabilities. In this sense, *B0* and *BH* are both versions of *Bladehelo.* In the verification, both *B0* and *BH* represented the same physical system. One code computed its behavior explicitly, the other relied on generic procedures with the system specified as a tree. The results had to agree. Any difference would flag an error or an accuracy problem in at least one of the two codes. *BH* was pre-existing and in use for several years, it was therefore presumed correct and accurate. The test was viewed as the verification of the new *B0* against the veteran *BH*. A deficiency in accuracy can be distinguished from an outright error because it is step dependent. Reducing the step size improves accuracy, but fails to correct errors.

Another point to notice is that neither *BH* nor *B0* is self-starting. The latter is not self-starting because of the AB4 integration scheme that it employs. But both codes are not self-starting because the acceleration of each part, whether computed explicitly or generically, depends on the acceleration of its parent. It takes a few steps before the correct pattern of accelerations is established. If started in an arbitrary state, each code would induce its own initialization

transient. These transients would be different, and the two time histories would never converge. For this reason, any verification work must start in a steady state situation. When set (trimmed) for steady state, the different transients would decay leaving the common steady state. A phase shift, in terms of time, may develop between the two histories. However, when compared as functions of the rotor azimuth, they become the same periodic function.

We start the verification process by comparing steady state conditions. We later go on to compare transient situations. This is done by establishing a steady state first and then disturbing it by a control input.

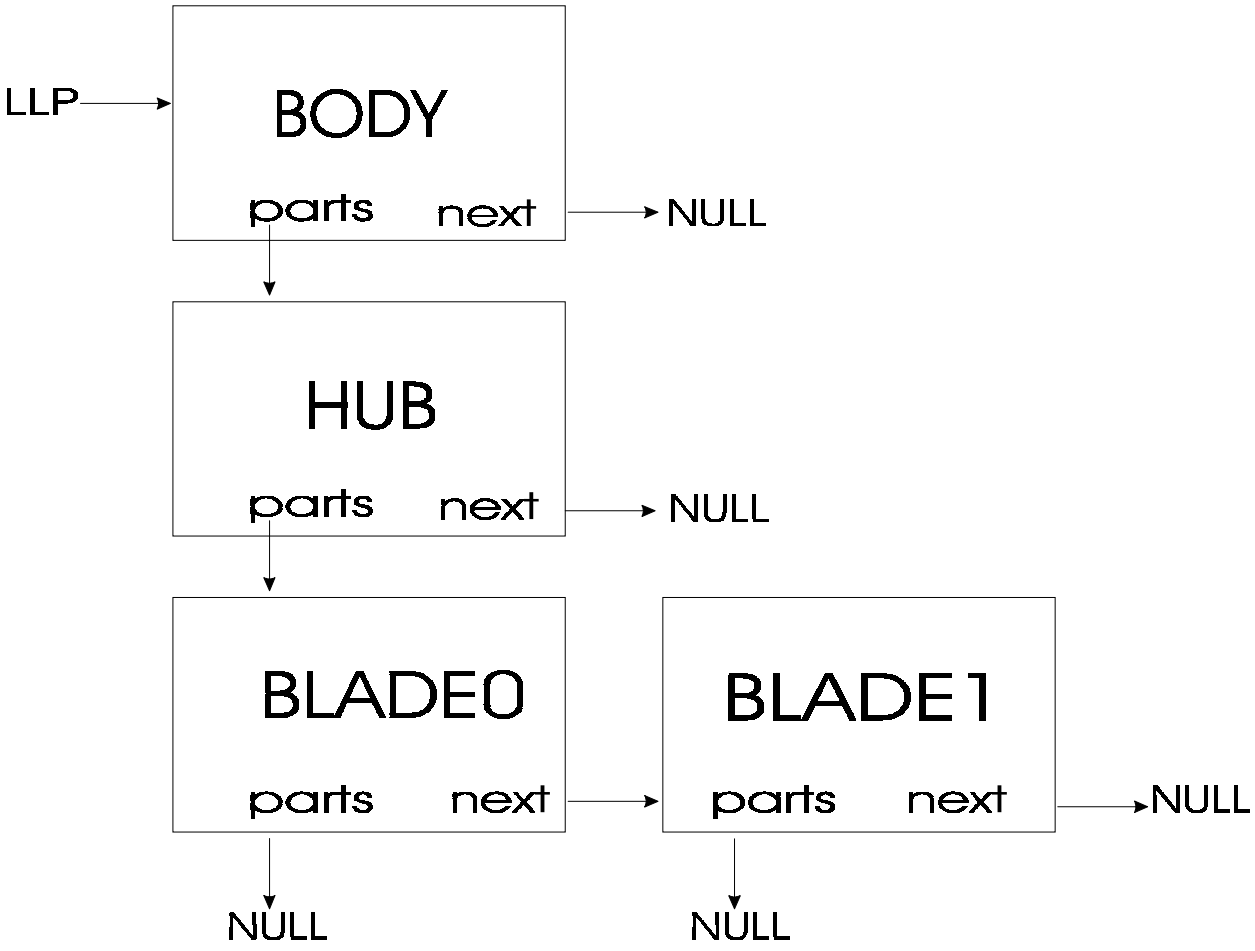


Figure 2-2: Tree Representing Two-Blade Rotor

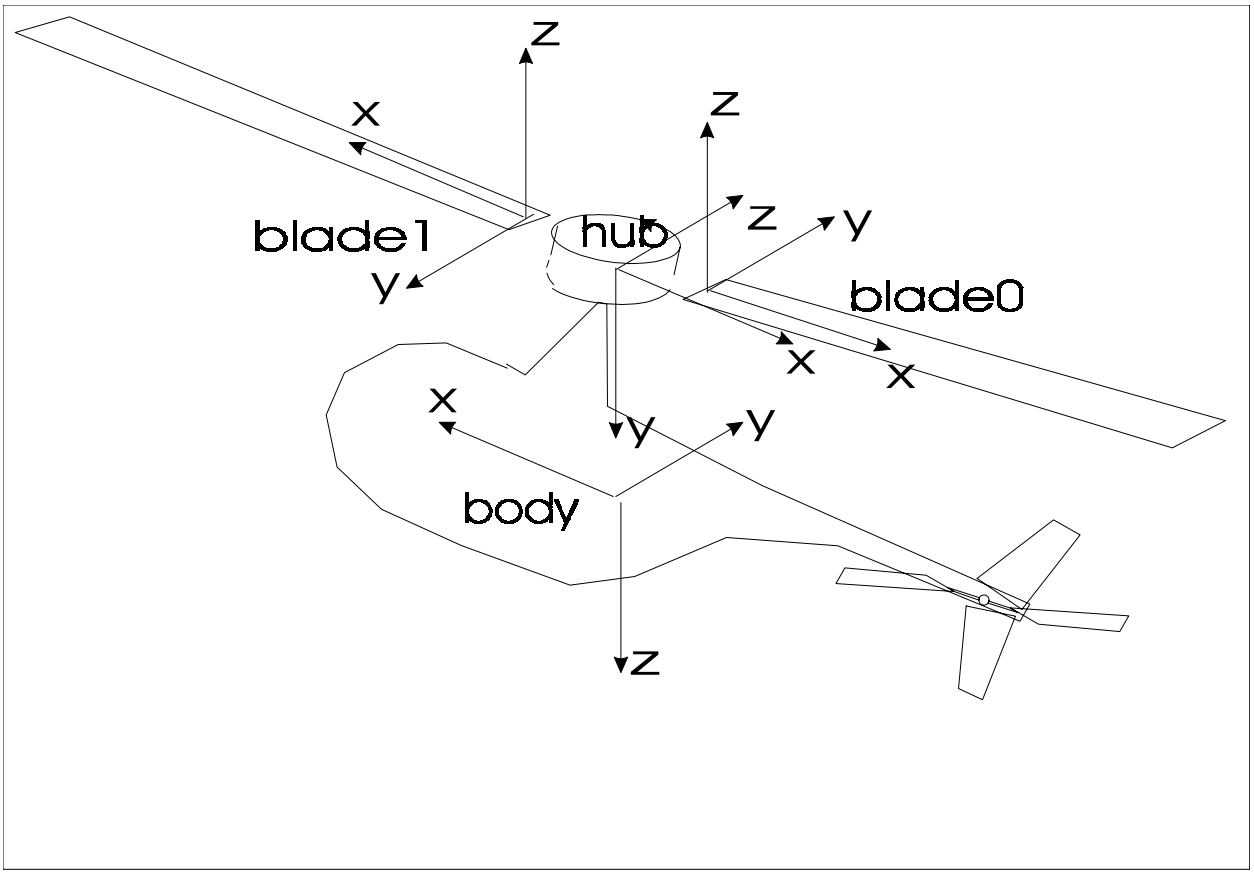


Figure 2-3: Structured Modeling of Two-Blade Helicopter

The verification test required that both codes be fed data representing the same physical system. It had to be within the capabilities of both codes. The system selected was a rotor with two offset flapping blades. The size and physical properties of the blades were inspired by the UH1. The blade data was represented in identical `Belelmnt` and `Belclcd` files. The basic geometry was defined for *BH* in the `Belinput` file. For *B0*, the general arrangement was defined as an articulated tree in the input file `rotor.sm`. The computer tree is shown diagrammatically in figure 2-2. Figure 2-3 shows the physical arrangement together with the reference point and coordinate system of each part. The orientation of the coordinate systems reflects the convention of reference 5 that the *y*-axis of each part be along the hinge connecting it to its parent.

2.3 VERIFICATION RESULTS

The data described in the next three subsections were taken with the shaft and helicopter body fixed and prevented from moving. The collective and cyclic controls were fixed as indicated resulting in a steady state condition.

2.3.1 CONING

With the shaft vertical and stationary and the cyclic control neutral, the flapping angles of both blades are equal and independent of azimuth. The constant value of these angles is the coning angle of the rotor. Coning data from *B0* is shown in figure 2-4. These data came into perfect agreement with *BH* as soon as the inputs were made consistent.

```
cyclic centered (16*16 steps B0)

Omega= -1942.000  Omegadot=  0.0000045

  Psi:    89.453   179.453   269.453   359.453
beta0:    0.6632    0.6632    0.6632    0.6632
beta1:    0.6632    0.6632    0.6632    0.6632
```

Figure 2-4: Coning data

2.3.2 FLAPPING

Flapping was compared with the shaft still vertical and stationary, but with 1 degree of lateral cyclic control. This results in the tilting of the rotor disk, which is reflected in flapping angles varying periodically with azimuth.

Initially, the flapping angles produced by *B0* were inconsistent: the two blades did not follow identical tracks. The differences were in the last two digits amounting to a few thousandth of a degree (for values roughly varying between plus and minus one degree plus coning angle). The inconsistency was traced to the evaluation of equation (9.16) of reference 5 in `accumulate()`. The equation requires the absolute angular acceleration of the part's parent. This was initially performed in a subroutine, where the parent's data was out of scope. The parent data was backed out as the difference between the part's relative and absolute angular acceleration, which proved not to be accurate enough. The ultimate solution to this problem was to put the computation of (9.16) inline in `accumulate()`, where the parent is visible. This made both blades track consistently in *B0* and brought about good agreement with *BH* (figure 2-5).

Note that *BH* data is captured at exactly 0, 90, 180 and 270 degrees. *B0* data is captured at the time step closest to these values. Both codes employed an identical "governor mechanism" that assured identical and practically the nominal value of the rotor RPM. This was necessary to assure identical results and also provided the side benefit, that *B0* steps closely divided into the rotor revolution. Still, a phase difference of a fraction of a step for the captured data could not be avoided. These slight phase differences are shown in figures 2-4 and 2-5. In figure 2-5, two sets of *B0* data are shown with a different phase deviation. In the first case, the azimuth closest to 90 deg is 89.990 deg, in the second case it is 89.995 deg.

The data is shown as captured from off line computer runs. Note that the azimuth conventions used by *B0* and *BH* are different, as follows:

```
        Rear     Right   Front    Left
BH     -180      -90       0      -270
B0      360       90     180       270
```

The phase differences are responsible for the occasional observed difference between *B0* and *BH* flapping angles. The differences are 1 in the last digit (one ten thousandth of a degree) at the peak of flapping and just under one hundredth of a degree where the curve of flapping angle against azimuth is steepest. The differences are consistent with the slope and the phase difference. The agreement of flapping angles in figure 2-5 is as good as it could be.

```
   B0   1 deg right cyc    16x16  AB4


  Omega= -1941.999  Omegadot= -0.001795867

 Psi:    89.990   179.992   269.990   359.992
beta0:    1.8217    0.8824   -0.2361    0.7033
beta1:   -0.2361    0.7033    1.8217    0.8824


  Omega= -1941.999  Omegadot= -0.0017932613

 Psi:    89.955   179.957   269.955   359.957
beta0:    1.8216    0.8831   -0.2360    0.7027
beta1:   -0.2360    0.7027    1.8216    0.8831

   BH   1 deg right cyc    16x4   RK4

  Omega= -1941.997  Omegadot= 11.508986

 Psi:     0.000   -90.000  -180.000  -270.000
beta0:    0.8822    1.8217    0.7035   -0.2361
beta1:    0.7035   -0.2361    0.8822    1.8217
```

Figure 2-5: Flapping data

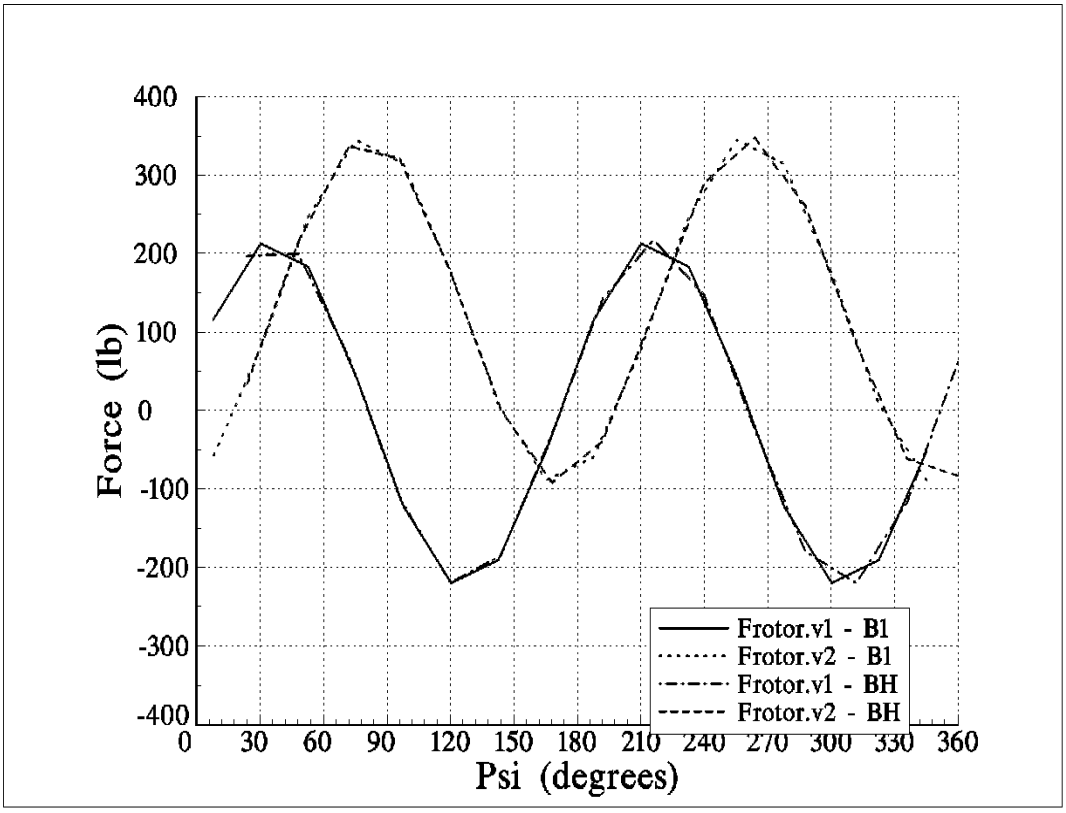
2.3.3 FORCES AND MOMENTS

The next task was to compare the forces and moments transferred from the rotor to the helicopter body. In *B0*, as already remarked, they were captured as `hub->g_force` and `hub->g_moment`. These were plotted against azimuth, with the shaft still fixed (figures 2-6 and 2-7).

Agreement of these curves proved elusive for several weeks. There was a marked disagreement in the moment curves, which proved to be an issue of accuracy rather than of principle. The

correct curve could be obtained with *B0* using smaller time steps. A step scheme of 16x256 produced adequate agreement. But 16x16, which was equivalent to the scheme used by *BH* was not fine enough.

The source of the problem was identified in the integration schemes employed. AB4, as well as RK4, require a consistent set of time derivatives of the variables of state for the nominal time of each step. This is difficult to produce, either by specific code or by generic code, because the angular accelerations of parts depend on the angular accelerations of their parents. The specific code was able to meet this criterion in the case at hand because the flapping hinges and the hub hinge were mutually perpendicular. The same effect was achieved in *B0* by slightly changing the integration scheme of *SM*.



Frotor at 1.00 deg right cyclic

Figure 2-6: Rotor Forces

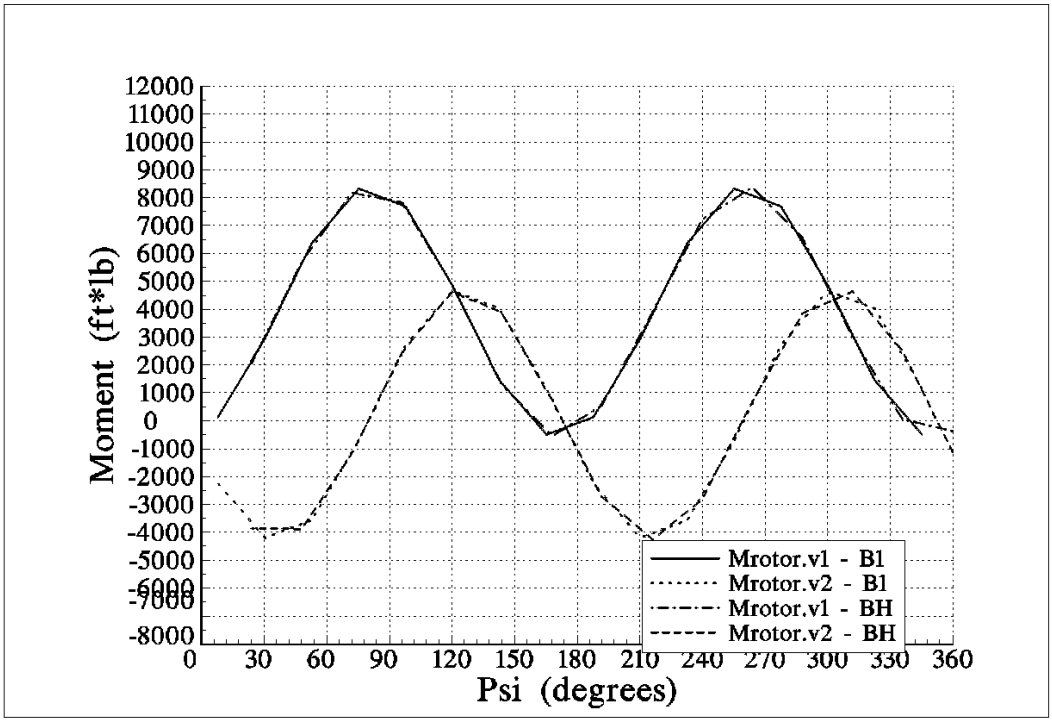


Figure 2-7: Rotor moments

Originally, `accumulate()` advanced each part one step on the recursive way up the tree. Instead, it now determines the angular acceleration of each part on the recursive way up, but does not use it to advance the part. This (for the condition of perpendicular hinges) produces a consistent set of angular accelerations. A separate routine `step(P*)` then goes recursively over the tree and advances each part using the previously produced angular accelerations. The following pseudo-code illustrates the operation of routine `step(P*)`

```
step(P *Part)
{
Item = Part->parts;
while(Item){
      step(Item); // Recursion
      CODE C //One integration step
      Item=Item->next;}//End of loop
```

This integration scheme, identified as "orderly" or as *Process-RI* in Appendix A of reference 5, was added in proof and reflects the experience of the verification described here. The "orderly" procedure produced the good agreement shown in figures 2-6 and 2-7 with a 16x16 step scheme

2.4 VERIFICATION OF FREE HELICOPTER

Only at this point, did *B0* become flyable in real time. Subjective evaluation found no difference between the handling of *B0* and *BH*.

A new code called *BS* was created, which applied the global treatment to the entire helicopter. In *BS*, the forces and moments were taken at the root of the tree (the body) and applied to the entire vehicle using the equations from Section 10 of reference 5. BS and BH were compared by starting both in a state of trimmed hover and then disturbing this state with the same control inputs for both. The results are shown in Figure 2-8. It is seen that the data track together and continue to do so after the control inputs are applied. As is to be expected, the two time histories eventually diverge, and the time they stay together depends on the step size. No step is fine enough to keep the two histories equal indefinitely. These points are illustrated in the figure.

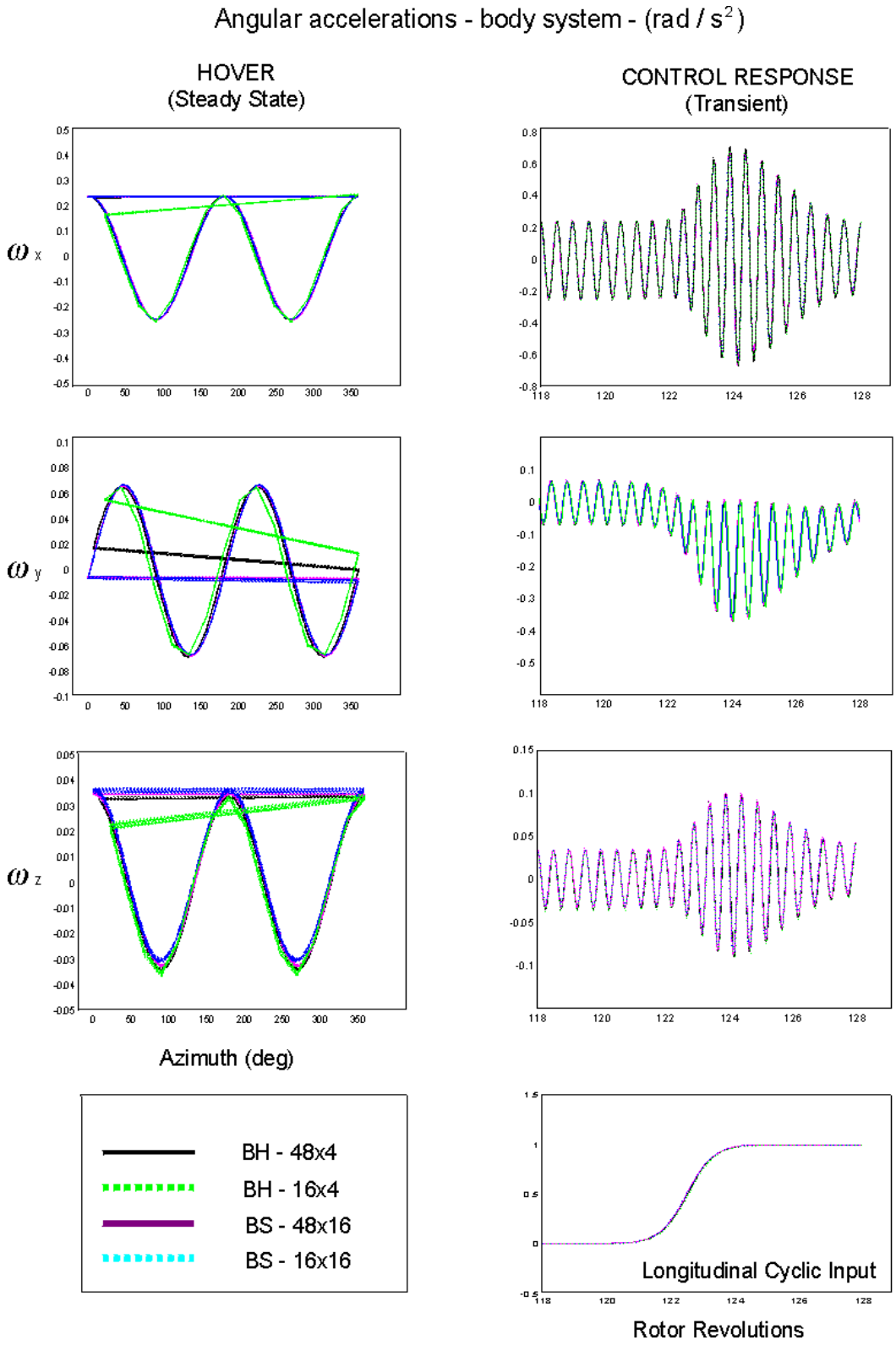


Figure 2-8: Comparison of *BH* and *BS* in stabilized hover and of response to longitudinal cyclic

2.5 THE FEATHERING HINGE

Up to this point, the blades had been modeled as rods with twist, blade pitch, and local incidence being parameters recognized only by the dynamic routine `BH_blade()`. This approach is not adequate for describing flexible blades. The flexing hinges, which connect adjacent blade segments must run along the chord, where the blade is most flexible. When the blade changes pitch as a result of control inputs, the flexing hinges must pitch with it. For this purpose, the blade must be rotated physically about the feathering hinge.

2.5.1 A PRESCRIBED ROTATION

The initial method of meeting this requirement was to perform a prescribed pitching rotation of the blade in addition to its dynamically integrated flapping motion. This approach created a conflict with the convention of reference 5 which required the flapping hinge to coincide with its *y* axis. A physical change of pitch of the blade moves its *y*-axis. However, the flapping hinge should not be affected. For this reason, in preparation for the modeling of flexible blades, it was decided to decouple the direction of a part's hinge from the *y*-axis of its coordinate system (see Figure 2-9).

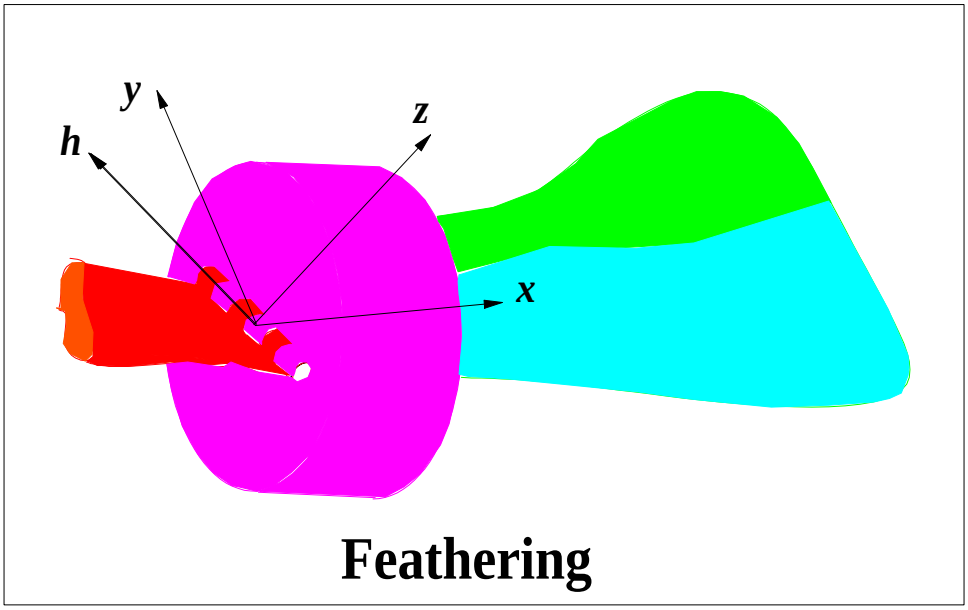


Figure 2-9: Pitch change of flapping blade.

A unit vector `h` (for "hinge") was added as a member of the `P` ("part") class. Equation (9.16) of reference 5 was restated for a general hinge direction as

$$\ddot{\beta} = -\dot{\vec{\omega}}_A^{a\,\mathrm{Pr}(A)} \cdot \vec{h}_A^{\,A} + \frac{1}{\vec{h}_A^{\,AT} J_A^{gA} \vec{h}_A^{\,A}} \left\{ k_{spring}^{A} \beta^{A} + \vec{h}_A^{\,A} \cdot \left[\vec{M}_A^{gA} - m^{gA} \vec{R}_A^{gA} \times \vec{a}_A^{aA} - \dot{J}_A^{gA} \vec{\omega}_A^{aA} - \dot{\vec{K}}_A^{iA} - \Omega_A^{aA} \vec{K}_A^{gA} \right.\right.$$
$$\left.\left. - J_A^{gA} \left(\dot{\vec{\omega}}_A^{aA} - (\dot{\vec{\omega}}_A^{aA} \cdot \vec{h}_A^{\,A}) \vec{h}_A^{\,A} \right) \right] \right\}. \qquad (2.1)$$

This was coded into `accumulate()`. The commanded pitch, resulting from the combined effect of collective and cyclic control, was removed from `BH_blade()`. Instead, a new procedure `Pitch()` was introduced, which resided in `dynamics.cpp` and was called from `rotorforces()`. The effect of `Pitch()` was to change the pitch of the root segment of a blade bodily while, at the same time, returning the flapping hinge to its original direction.

Hinge directions still started by default along the *y*-axis. However, in the case of the root segment of a blade, they are changed as necessary, as the controls are applied.

2.5.2 A PRESCRIBED PART

One obvious limitation of the method of feathering described above is that only one feathering hinge can be accommodated per part, which is not sufficient for a teetering rotor. Also, a flapping part cannot be separated from its feathering extension. For these reasons a different approach to the prescribed feathering was eventually adopted. We returned to the articulated tree with a single hinge between each part and its parent, but that hinge could be either dynamic or prescribed. The incidence at dynamic hinges is integrated in the *SM* manner as described in 2.1 above. The incidence of "prescribed hinges" is prescribed.

A new member, `prescribed,` was added to the `Part` class, which serves the dual role of pointer to the prescribing function and a flag. If `prescribed` is `NULL`, the part is recognized as dynamic and integrated by the GRD method. Otherwise, the integration is bypassed and the prescribing function is called to set the part's incidence. This is illustrated in Figure 2-10 and in the following revised pseudo-codes for `accumulate()` and for `step()`:

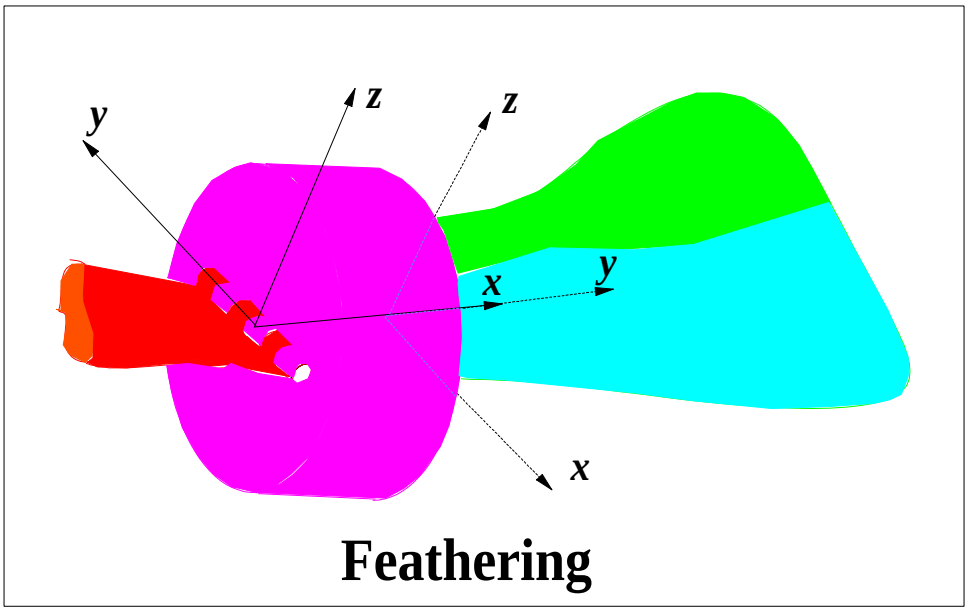


Figure 2-10: A Dynamic flapping hinge and a prescribed feathering hinge.

```
accumulate(P *Part)
{
Item = Part->parts;
while(Item){   CODE A //Place Item relative to Part
     accumulate(Item); // Recursion step.  Determine Item's global properties
     if(!Item->prescribed){
     CODE B }//Determine the angular acceleration of Item's branch
     CODE C //One integration step (later moved out from this position, see text)
     CODE D //Accumulate Item's contribution to part's global quantities.
     Item=Item->next;} //End of "while(Item)" loop.
     CODE E //  call pointed function to determine local loads applied directly to Part
            //  and add them to global loads accumulated from children.}

step(P *Part)
{
Item = Part->parts;
while(Item){
     step(Item); // Recursion
     if(Item->prescribed)(*prescribed)(Item);
     else CODE C //One integration step
     Item=Item->next;}//End of loop
```

The new formalism can represent any combination of dynamic and prescribed hinges. The convention that the *y*-axis be selected along the hinge is again tenable and is respected in Figure 2-11. Nevertheless, it was convenient to maintain the freedom, established in 2.6.1, to have the hinge point in any direction. In the application, the coordinate system of feathering blade segments was selected so that the feathering hinge was along the *x*-axis. This is convenient for uniformity in the `BH_blade` routine that computes aerodynamic loads on these segments as well as other blade segments.

2.5.3 OTHER UPGRADES

In preparation for modeling flexible blades, BS was also upgraded to accept multiple `Belelmnt` and `Belclcd` files, so that the different segments of the same blade could be different.

2.6 STRUCTURED MODELING CAPABILITIES AND LIMITATIONS

The SM formalism has lived up to its promise. It is now possible to define complex models mostly through `*.sm` input text files. Specialized editing tools or graphic interfaces could streamline this process further. Even without these, using an ordinary text editor, the time required for the creation of a new model has been reduced from weeks to hours.

The more complex models that SM easily creates require additional resources to run, especially in real time. Models featuring close-by, parallel hinges are the most demanding in terms of run-time resources. The advance in the speed and capability of microprocessors over the period of the project provided some compensation for this effect. We have been able to run two-segment flexible blades in real time, as reported in Section 2.8 below. Still, by and large, SM makes it easy to create models that are too complex for the UA FDL to run in real time. The data on three segment flexible blade models in 2.8, which can be used to estimate the bending loads on the blades and their fatigue life under various flight regimes, had to be collected off line. The formalism places no limit on the number of segments used to represent a flexible blade, but the throughput of the host computer does.

The Robinson R22 is another example. In the past, *Bladehelo* had been used to represent the R22 with a rotor consisting of two flapping blades hinged at the hub. We felt that this was a fair model of the R22 [3, 4], and the editors of the Journal of Aircraft agreed. SM made it possible to set up a true three hinge arrangement. However, because of the close-by parallel hinges, this model is subject to unstable vibrations of the short center piece in real time. On the other hand, a model of the Hughes 269A (three blades, fully articulated, Figure 2-11) has been run in real time successfully by a single Pentium II processor.

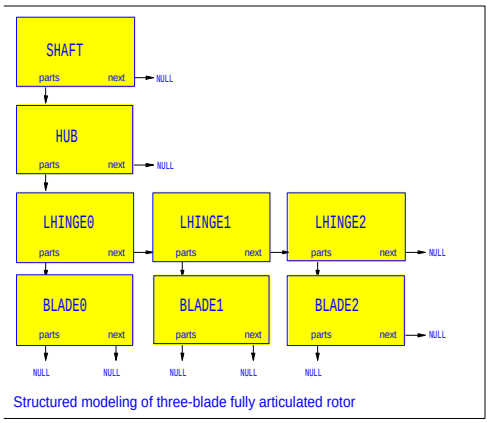


Figure 2-11: Tree representing fully articulated three-blade rotor

SM broke through the coding hurdle. Thanks to it we are now facing the run-time hurdle.

2.6.1 PARALLEL PROCESSING OF ROTOR MODELS

Articulated trees lend themselves to parallel processing of different branches. In our rotor models, branching occurs at the hub and the different blades make separate branches. It is thus possible to implement a limited amount of parallel processing corresponding to the number of blades. This section describes the methods and results.

Parallel processing was initiated by launching separate threads for multiple branches and letting the operating system allocate separate processors to the threads. This was accomplished within the multi-processor "Monster" computers. See Chapter 4 for the details of the architecture.

Separate threads were launched only where branching actually occurred. This was accomplished by introducing the `siblings` flag into the `accumulate()` code:

```
accumulate(P *Part)
{
Item = Part->parts;
siblings = Item->next;
if(siblings){
     while(Item){
          // Launch thread for Item
          Item=Item->next;} //End of "while(Item)" loop.
     // Wait for all threads to return}
else process(Item);
while(Item){
     CODE D //Accumulate Item's contribution to part's global quantities.
     Item=Item->next;} //End of "while(Item)" loop.
CODE E // call pointed function to determine local loads applied directly to Part
          // and add them to global loads accumulated from children.}

process(P *Item){
     CODE A //Place Item relative to its parent
     accumulate(Item); // Recursion step. Determine Item's global properties
     if(!Item->prescribed){
     CODE B }//Determine the angular acceleration of Item's branch
     }
```

Most of the content of the `while(Item)` loop was removed to a separate function `accumulate_item()`, which was made the function of the newly defined thread. If there are siblings, a thread is launched for each and the system then waits for all of the threads to return. If there are no siblings, `accumulate_item()` is called for the only child. The threads used are “persistent threads” created at initialization and reused every time step. See Chapter 4 for more details of the hardware and software architecture.

Note that `accumulate_item()` requires access to the parent of its parameter, `Item`. For this purpose, a `parent` pointer has been added to the `part` class. These pointers are set at initialization when the tree is constructed.

Note also that `CODE D` was not included in `accumulate_item()`, but, rather, is performed later in a second `while(Item)` loop. This is because in `CODE D` the siblings are writing data into their common parent, and this cannot be done safely in parallel.

The performance gains realized from parallel processing were modest. Only part of the work could be done in parallel, and threading overhead took its toll. For a three-blade rotor, the throughput improvement was 66%.

2.7 FLEXIBLE BLADES — INPUTS AND RESULTS

Flexible blades are represented as a sequence of blade segments attached with spring loaded flexing hinges. Figure 2-12 illustrates a two-blade rotor, with each blade broken into two segments as a computer tree. Figure 2-13 shows the corresponding physical system.

With rigid blades, *Bladehelo*, and in particular, the *BS* code, was run with a step scheme of (16×16), which means that there were 16 large steps per revolution and 16 small steps per large step. With a rotor rate of 5.5 revolutions per second, this translates into a frame rate of 88Hz for the body and 1400Hz for the rotor. For rigid blades, this was more than adequate for accuracy, and it could be sustained in real time by a single Pentium processor. Accurate computation of the time histories of flexible blades requires additional resources for the following reasons:

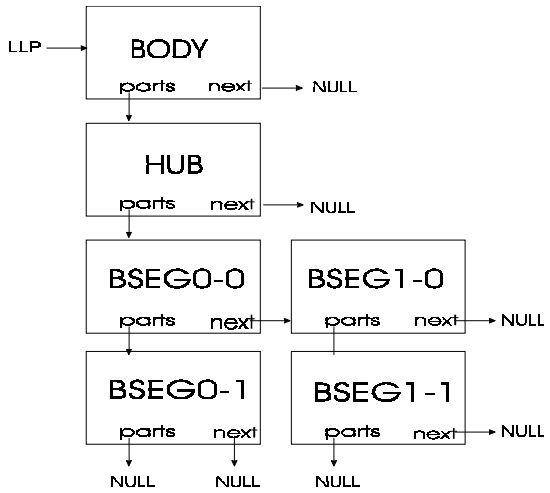


Figure 2-12: Tree representation of helicopter with two two-segment blades

- There are more parts in the tree.
- Aeroelastic modes of higher frequency become relevant, which requires smaller time steps.
- Trees with close parallel hinges are more sensitive (Section 4).

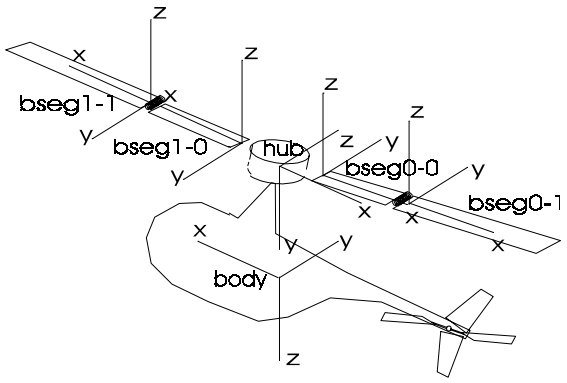


Figure 2-13: Flexible blades represented as two segments with flexing hinge

The two-segment blade features two nearly parallel hinges, the flapping hinge and the flexing hinge. The angle between the two hinges, which is due to control inputs and to blade twist, is small. Two variations were tried. When the flexing hinge was at mid-blade, a step scheme of 16×48 was required to assure accurate results. Upgrade to a faster processor and the conversion of the code to 32 bit operation made it possible to run this triple fast application in real time using a single Pentium processor. With the break at 30% of the span, a step scheme of 16×256 was necessary, which could not be accommodated in real time. Flexible blades with the break at one third of the span or ones modeled with more than two segments produced data off line.

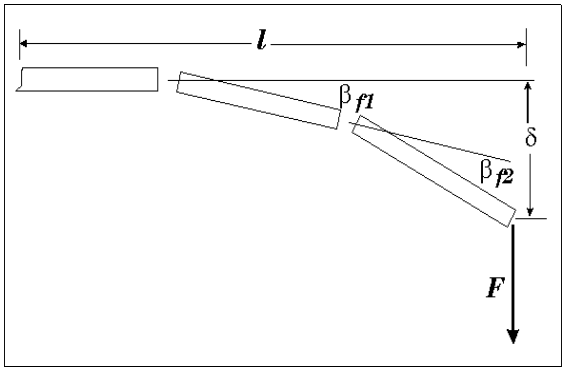


Figure 2-14: Measurement of blade flexibility

The spring on the flexing hinge(s) was selected to fit the compliance as measured on a UH-1H blade. It was found that the ratio of sag at the blade tip to the vertical force applied was

$$\frac{\delta}{F} = 0.190 \text{ in/lb.}, \tag{2.2}$$

See Figure 2-14 for the nomenclature. With a blade length l and a single break at radius κl, a linearized expression for the spring constant (using small angle approximations) is

$$k_{spring} = \frac{M}{\beta_f} = \frac{F(1-\kappa)l}{\delta/[(1-\kappa)l]} = \frac{F(1-\kappa)^2 l^2}{\delta} \tag{2.3}$$

M is the moment at the hinge. For multiple flexing hinges, equation (2.2) changes into

$$\delta = \sum_i \frac{F(1-\kappa_i)^2 l^2}{k_{spring,i}}, \tag{2.4}$$

where the sum is over all the flexing hinges. If a uniform blade is assumed ($k_{spring,i}$ is independent of i), this becomes

$$k_{spring} = \frac{Fl^2}{\delta}\sum_i (1-\kappa_i)^2 . \tag{2.5}$$

This was the equation used to determine the spring constant selected ((2.3) is a special case of (2.5)).

Data of flapping angles and flexing angles was collected off line. Figure 2-15 provides the nomenclature. Table 2-1 offers a comparison of flapping and flexing angles of a rigid blade, one with one flexing hinge at mid-blade, and one with two flexing hinges at 0.3 and 0.6 of the blade span. Data for multi-segment blades is shown with and without spring loading of the flexing hinges.

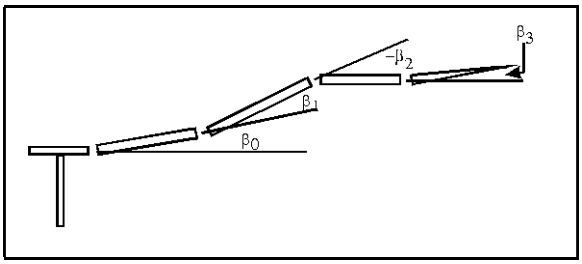


Figure 2-15: Flapping and flexing angles

Table 2-1: Flapping and Flexing angles in OGE hover

Angles (deg)	β_0	β_1	β_2
Rigid blade	0.6618	--	--
2 seg, spring ld	0.6788	-0.0563	--
2 seg, free flex	0.6790	-0.0569	--
3 seg, spring ld	0.6549	0.0632	-0.1359
3 seg, free flex	0.6540	0.0677	-0.1437

It is seen that the flexing angle remains small and is nearly identical with and without spring loading. Elastic stiffness plays only a minor role.

Table 2-2: Flapping and Flexing angles (deg) in hover with 1 deg of cyclic applied

Azimuth (deg)	90	180	270	360
Rigid blade	1.6905	0.7518	-0.3667	0.5721
Seg 1 of 2	1.6965	0.7705	-0.3384	0.5880
Seg 2 of 2	-0.0269	-0.0621	-0.0866	-0.0526
Seg 1 of 3	1.6650	0.7442	-0.3571	0.5647
Seg 2 of 3	0.0865	0.0662	0.0490	0.0681
Seg 3 of 3	-0.1208	-0.1467	-0.1662	-0.1411

Table 2-2 offers flapping and flexing data with the shaft fixed and one degree of right cyclic applied. Table 2-3 offers flapping and flexing angles at 70 knots of forward speed. This data was taken with the helicopter fixed in a 70 knots airflow in trim attitude and with trim controls applied.

Table 2-3: Flapping and Flexing angles (deg) in forward flight at 70 knots

Azimuth (deg)	90	180	270	360
Rigid blade	0.3396	-1.1539	0.8911	2.3173
Seg 1 of 2	0.2744	-1.0894	1.1149	2.3262
Seg 2 of 2	0.0715	-0.2265	-0.5960	-0.0250
Seg 1 of 3	0.2283	-1.0937	1.1820	2.2964

Seg 2 of 3	0.1066	-0.0276	-0.2110	0.0868
Seg 3 of 3	-0.0162	-0.2542	-0.5914	-0.1189

2.8 CONCLUSION

The flexing data collected here is the type of data required for the fatigue analysis of the blade in operation. We have illustrated the ability to produce this data, which the final stage of the project requires. However, some of the data would have to be produced off line or would require additional computational resources.

3. VORTEX LATTICE ROTOR WAKE MODEL.

3.1 WAKE MODEL DESCRIPTION.

The preexisting *Bladehelo* modeled the wake as a uniform inflow based on "momentum theory." The total aerodynamic force created by the rotor was applied to an appropriate mass flow, resulting in a downwash. The rotor was subjected to half of its own downwash.

The new model developed under this project follows the ideas of references 8 and 9, and derives the wake flow from bound, shed, and trailing vortices generated by the rotor. Unlike the references, the vortices are derived using the lifting line rather than the lifting surface scheme. *Bladehelo* models each rotor blade as a sequence of finite blade elements. The aerodynamic force generated by each rotor blade is computed by summing the aerodynamic force generated by the blade elements. The aerodynamic force of an element is determined using the local airspeed at the element (including induced flow) with lookup tables for the lift and drag coefficients. The wake model provides to the rotor model the induced velocity of the air at each blade element's location.

As a rotor blade moves through the air, vorticity is shed in a continuous sheet along the trailing edge. In *Bladehelo*, both the physical blade and its motion have been made discrete. The lift is computed for the elements and assumed uniformly distributed along each of them. These values of lift per unit span are computed only at discrete time steps. The discrete representation of span and of time dictates an approximation of the vortex sheet as a discrete vortex lattice. The vortex lattices produced by the blades, together, form the wake.

Each blade element is treated as a lifting line. The strength of a bound vortex segment is computed using the Kutta-Joukowski theorem

$$\Gamma = \frac{L'}{\rho V}, \tag{3.1}$$

where L' is the lift per unit span for the element, ρ is the air density, and V is the magnitude of the component of air velocity relative to the element that is normal to the lifting line. A trailing vortex is generated at the blade element boundaries and carries the difference of bound vorticity (circulation) about the two adjacent elements. Vortices are shed from each blade element only once a step and carry the difference between the circulation about the element from the last step to the current step. Trailing vortices from the tip and root of the blade carry the full circulation of these elements. These are the strongest trailing vortices. In real time, they are the only ones considered (see section 3.4, below).

The element boundaries at each time step define the initial location of lattice nodes. The nodes move with the flow. Each node moves with the local air velocity, which includes the velocity induced by the combined lattice of all blades. In hover, the lattice produced by each blade forms an essentially helical sheet with the sheets of different blades intertwined. The wake model keeps track of the positions of the vortex lattice's nodes over time. The vortex lines between

nodes are approximated as straight and referred to as "segments." Trailing vortices form "chord-wise segments" and shed vortices form "span-wise segments." The vortex theorem of Helmholtz, states that vorticity is conserved. A free vortex filament maintains a constant strength. The vortex filaments cannot end in the fluid. In the model, vorticity is constant along each segment and is automatically conserved through each node of the lattice. See section 3.2.3 on lattice truncation.

Figure 3-1 illustrates a part of a vortex lattice from one blade.

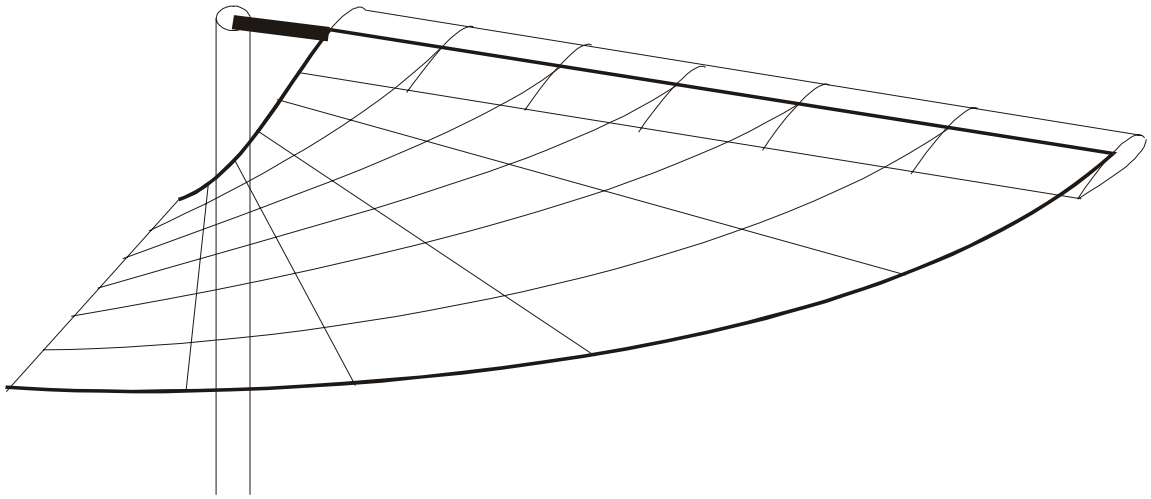


Figure 3-1: Part of the vortex lattice from a single blade.

The vortex lattices are used to determine the velocity of the air at the center of the aerodynamic elements for use in computing the lift and drag. The lattices are also employed to determine the air velocity at each of the lattice nodes so that the positions of the nodes can be propagated forward in time.

If the viscosity of air is neglected, the flow outside vortex lines is irrotational. If it is also assumed incompressible, the equations governing it are linear. The induced flow can then be computed as a superposition of the flow induced by the lattice segments. The flow induced by a vortex filament is given by the Biot-Savart law

$$\vec{v}_i = \int \frac{\Gamma(\vec{r} - \vec{p}) \times \overrightarrow{dr}}{4\pi |\vec{r} - \vec{p}|^3}, \tag{3.2}$$

where Γ is the strength of the filament, $\vec{r}$ is a radius vector to a point on the filament, $\vec{p}$ is the radius vector to the point of interest, and the integration is carried along the filament.

3.2 COMPUTATIONAL DETAILS

3.2.1 VELOCITY INDUCED BY A VORTEX SEGMENT

When applied to a straight filament, equation (3.2) yields

$$v_i = \frac{\Gamma}{4\pi} \frac{\hat{e}_1 \times \hat{e}_2}{|\hat{e}_1 \times \hat{e}_2|} \frac{\hat{s} \cdot (\hat{e}_1 - \hat{e}_2)}{lh}, \tag{3.3}$$

where l is the length of the segment, h is the perpendicular distance from the segment of the point where $\vec{v}_i$ is determined, $\hat{e}_1$ and $\hat{e}_2$ are unit vectors pointing at this point from the two ends of the segment, and $\hat{s}$ is a unit vector along the segment.

In real time, our model employs a less computationally intensive approximation to (3.2), namely

$$\vec{v}_i = \frac{\Gamma l \hat{s} \times \vec{h}}{4\pi \left|\vec{h}\right|^3}. \tag{3.4}$$

In equation (3.4), $\vec{h}$ is a vector from the midpoint of the segment to the position at which the induced velocity is being computed. Substitution of (3.3) for (3.2) would be compatible with our model, because the vortex segments are straight. The use of the further simplified (3.4) is permissible so long as the position at which the induced velocity is being computed is a "long" distance from the segment, compared to the length of the segment. Also, the extra precision of (3.2) may be out of place in the context of the approximation of a discrete lattice. As it turns out, computational restrictions force us to accept fairly long segments. Some off-line computations were performed with (3.3) and compared to the same computations carried out with (3.4). On the whole, agreement appears acceptable.

3.2.2 SMOOTHING

In equation (3.4), as $\vec{h}$ approaches zero, the induced velocity becomes infinite. This is an unrealistic result that arises from the discretization of vortices. At a distance, the discrete segments approximate the effect of the vortex sheets, but at close range the discrete segments introduce large ripples in the flow. The tip and root vortices do exist as discrete vortices, but all other vortices in the lattice are there just as an approximation to a continuous sheet, an approximation that is not valid close to a vortex segment. Initial trials with the wake model using either equation (3.3) or equation (3.4) resulted in instabilities in the solution because some points at which the induced velocity was computed were too near vortex segments. To compensate for this effect, a smoothing mechanism has been incorporated into the model. For computations of the contribution to the induced velocity for a segment, when the point of interest is closer to the segment than a given smoothing radius, R_s, equation (3.4) is replaced by

$$\vec{v}_i = \frac{\Gamma l \hat{s} \times \vec{h}}{4\pi R_s^3}. \tag{3.5}$$

Equation (3.5) gives the average velocity induced by a segment over a sphere centered at the point of interest when the segment midpoint is within the sphere. In the case of equation (3.3), smoothing is accomplished by replacing the perpendicular distance *h* by the smoothing radius R_s. For a rotor with a radius of 24 feet, a smoothing radius of 4 feet was found to give reasonable results. Smoothing radius values as small as 2 feet have been used. In the case of the

tip and root vortices, the smoothing mechanism may be viewed as the effect of a vortex core of finite radius.

3.2.3 LATTICE TRUNCATION

A new set of nodes and vortex segments – span-wise and chord-wise – is added to each lattice at the rotor blade each wake simulation frame. Each such set is referred to as a generation. The generation closest to the blade is the youngest. Theoretically, the vortex segments continue to exist forever. This would imply that, as a simulation continues, the computational requirements for keeping track of the wake would grow indefinitely as more generations are added. The computer program can handle only a finite number of generations. Therefore, after a predefined number of generations have been included in the wake lattices, the oldest generation must be periodically discarded.

Truncation of the vortex lattice is a computational necessity. The magnitude of the velocity induced by the segment is inversely proportional to the square of the distance, and the oldest generation is usually the farthest from the rotor blades. Therefore, discarding the oldest generation should not greatly impact the accuracy of the induced velocity computations at the rotor blades, as long as a sufficient number of generations are retained.

On the other hand, the effect of truncation on the oldest (closest) part of the lattice cannot be ignored. The wake initially contracts and then, in the case of OGE hover, maintains an essentially constant diameter. Truncation leads to the wake flaring back out. In the process, the bottom of the wake becomes unstable and newer generations overtake earlier ones. The instability propagates back up the wake and affects the inflow at the disk
.
It is necessary to compensate for the truncated part of the wake. We do this by introducing a sequence of vortex rings. Each time a revolution's worth of tip vortices is generated by the rotor blades, i.e., each time the rotor completes $1/B$ of a revolution, where B is the number of blades, a revolution's worth of the oldest parts of the wake lattices from each blade are truncated. At the same time, a vortex ring of similar size and strength is introduced at the same location. The ring induces a flow similar to that of the truncated part of the wake. After creation, each ring maintains a constant vortex strength, radius, and orientation and floats with the local flow. In this way, the wake is effectively represented beyond the point of truncation. The flow induced by the rings continues to pull the wake down and prevents it from unraveling.

The number of rings that can be retained is also finite. At some point, the sequence of rings must be truncated. This happens far from the point of truncation of the wake and has no appreciable effect on it. The rings are, by their nature, more robust than the vortex lattice and are not adversely affected by their sequence being cut off.

The parameters for each new ring are determined as follows:

- The average position of the nodes that comprise the tip vortices for the oldest revolution is taken to be the center of the ring.
- The strength of the ring is the average strength of the tip vortex segments that make up the last revolution.

- The velocity induced by the last revolution's worth of tip vortex segments is computed for the center of the ring. The direction of this vector is taken to be the axis of the ring.
- The velocity induced at the center point by the segments being converted to a ring and the strength of the ring are used to solve for the radius of the new vortex ring.

The flow induced by a ring has been pre-computed and tabulated as a function of the ratio of the distance from the center of the ring to the ring radius and the angle between the relative position vector and the ring axis. To compute the effect of a ring at a point in space, the distance and angle parameters are computed and used with a lookup table to obtain the induced velocity. Figure 3-2 shows the vortex structure of a wake in hover composed of the lattice generated over six revolutions and ten vortex rings.

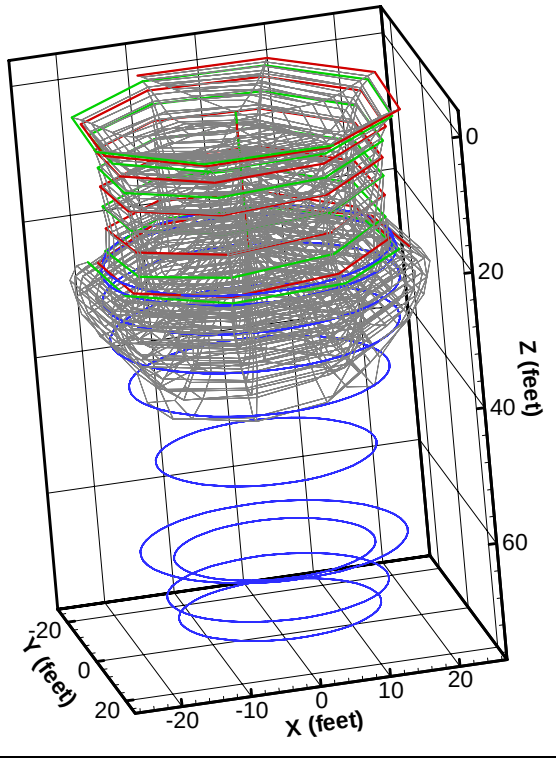


Figure 2: Six-revolution wake in OGE hover

3.3 THE NUMBER OF REVOLUTIONS OF WAKE THAT NEED BE RETAINED

To estimate the number of generations required, the ideal uniform induced velocity for hover was compared to the velocity induced at the hub of the rotor by a number of vortex rings. For the ideal case of uniform disk loading in hover, the vortex strength along the rotor blades is constant. Equation (3.1) shows that this is the case when the lift per unit span is proportional to the velocity. The velocity is proportional to the radial position and so is the lift per unit span.

Let B stand for the number of rotor blades. The non-dimensional thrust coefficient, C_T, can be computed in terms of Γ by integrating the lift per unit span of (3.1) along the rotor blades, each of length R, in the following manner:

$$C_T = \frac{B\int_0^R \rho\Gamma\Omega r dr}{\frac{1}{2}\rho\pi R^4\Omega^2} = \frac{B\Gamma}{\pi R^2\Omega}. \tag{3.6}$$

This equation can be solved for Γ as

$$\Gamma = \frac{C_T\pi R^2\Omega}{B}. \tag{3.7}$$

For the ideal case of constant vortex strength along the blades, all of the vorticity is shed at the tips. Therefore, equation (3.7) also gives the strength of the rotor tip vortices in hover. The tip vortices normally form intertwined helical patterns as they move away from the rotor. For this estimate, however, the tip vortices were modeled as rings. Each ring represents the trailing vortex produced over one revolution of the rotor and has a strength equal to the sum of the strengths of the vortices shed from all the blades, i.e. the strength of the rings is $B\Gamma$.

The radius of the rotor is used for the radius of each ring. In reality, the radii of the shed vortices contract and the air velocity increases as the tip vortices move away from the rotor, but for the current estimate of the conditions near the rotor, these effects will be ignored. The air velocity induced by a vortex ring that is a vertical distance h below the rotor is determined by integrating equation (3.2) around the ring. At the hub, which is centered above the rings, the induced air velocity has only a vertical component, which comes out as

$$v_i = \int_0^{2\pi} \frac{B\Gamma R^2}{4\pi p^3} d\theta = \frac{B\Gamma R^2}{2p^3}, \tag{3.8}$$

where p is the distance from the ring to the hub as shown in figure 3-3. Equation (3.8) can be non-dimensionalized by dividing by the speed of the rotor blade tip, ΩR. After substituting equation (3.6) for Γ, the non-dimensional inflow parameter is given by

$$\lambda_i = \frac{\pi R^3 C_T}{2p^3}. \tag{3.9}$$

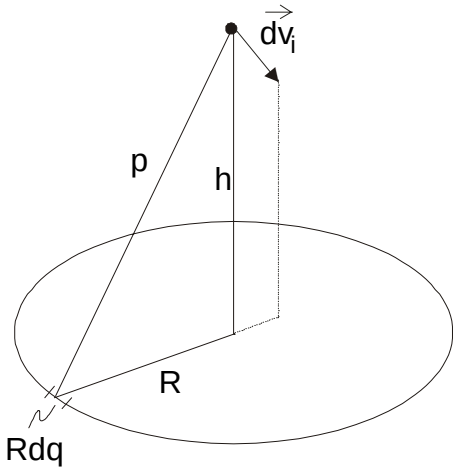


FIGURE 3-3: Velocity induced by a vortex ring

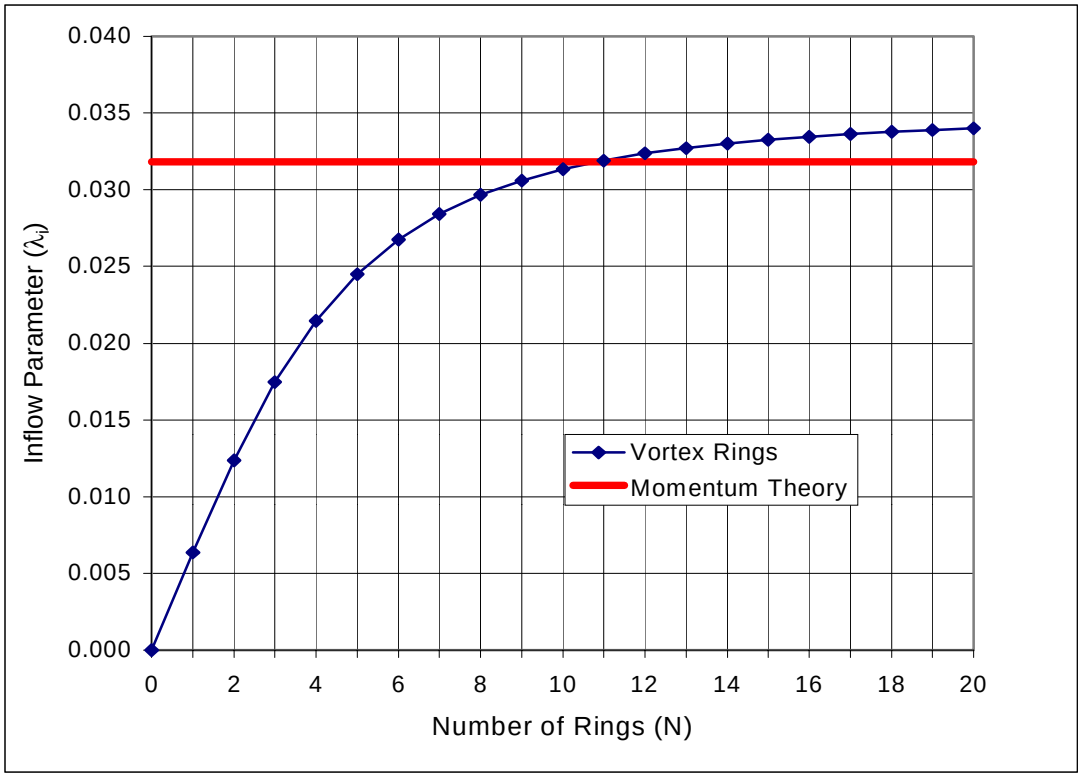


Figure 3-4: Number Of Revolutions Required To Reproduce Induced Velocity In Hover.

The vortex rings move away from the rotor at the speed of the air. Assuming the air velocity remains constant downstream from the rotor, the vertical distance from the hub to the n^{th} ring is given by

$$h_n = v_i n \frac{2\pi}{\Omega} = \lambda_i R 2\pi n \tag{3.10}$$

Replacing p with

$$p_n = R\left(1+\left(\frac{h_n}{R}\right)^2\right)^{1/2}, \tag{3.11}$$

equation (3.9) becomes

$$\lambda_{i_n} = \frac{\pi C_T}{2\left(1+\left(h_n/R\right)^2\right)^{3/2}}. \tag{3.12}$$

The inflow parameter given by actuator disk theory for uniform inflow is

$$\lambda_i = \tfrac{1}{2}\sqrt{C_T} \tag{3.13}$$

Using the inflow parameter from equation (3.12), and dividing by the rotor radius to non-dimensionalize, equation (3.10) becomes

$$\frac{h_n}{R} = \pi n \sqrt{C_T}\,. \tag{3.14}$$

Substituting equation (3.14) into equation (3.5) and summing over N rings gives an estimate for the velocity induced at the rotor hub by the tip vortices generated over N rotor revolutions,

$$\lambda_i = \sum_{n=0}^{N-1} \frac{\pi C_T}{2\left(1+n^2\pi^2 C_T\right)^{3/2}}. \tag{3.15}$$

This equation is plotted versus N in figure 3-4. The solid line is the value for the inflow parameter given by equation (3.13) for ideal uniform inflow. The thrust coefficient used for both plots is for a UH-1H helicopter in hover ($C_T = 4.05\times10^{-3}$). The plot indicates that the uniform inflow value would be achieved with approximately 11 rings, with each ring representing one revolution's worth of tip vortices.

Based on the above estimate, it was decided, as a rule of thumb, that the wake produced by ten rotor revolutions was to be retained. The inaccuracy introduced appears modest, and the very limited computational resources would hardly support more.

3.4 MOMENTUM THEORY STARTUP CORRECTION

When the wake program starts, there is initially no wake and no inflow. When the first few revolutions of wake segments are generated, there is no downwash to move the vortices away from the rotor and to space them from each other. Without a correction, the tip vortices could initially remain too close together triggering instabilities and singularities. With the smoothing

algorithm in force, the effect of each vortex segment on the others could be greatly reduced, preventing the wake from ever spreading out.

These difficulties are avoided by superimposing a uniform inflow correction from the outset. In this way, there is inflow in roughly the correct amount in place immediately, which helps float the trailing and shed vortices away and space them appropriately. As the wake builds up, the vortex lattice produces more and more inflow, and the uniform correction is reduced accordingly. Once the wake is grown to the extent planned, the correction is no longer applied.

The uniform correction is based on the difference between the average of the inflow produced by the vortex lattice, as computed at an appropriately chosen grid of points in the plane of the rotor disk, and the uniform induced velocity that would be required for momentum balance. This average is determined continuously, and the correction is adjusted accordingly. Should the wake be able to account for the entire momentum balance, the uniform correction would automatically reduce to insignificance.

Figure 3-5 traces the history of the wake of a UH1 in OGE hover (including the startup correction) with only tip and hub vortices retained and with eight wake steps per revolution. The figure documents the wake as it builds up and in the steady state that is reached when retaining ten revolutions of wake lattice and ten rings. The figure provides a comparison of the wake model average induced velocity with the value that would be required for momentum balance. It is seen that the difference converges to nearly zero.

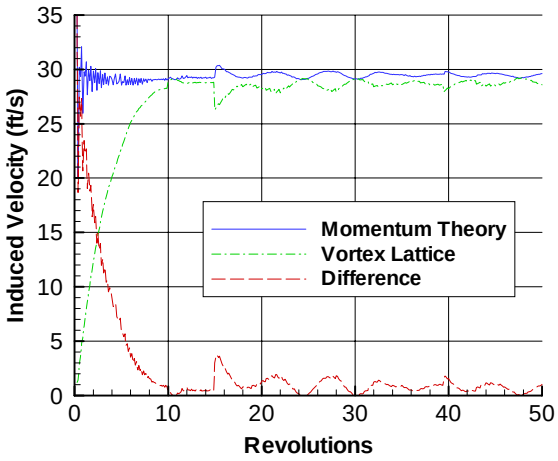


Figure 3-5: Average inflow produced by wake model with ten revolutions and ten rings.

3.5 MODEL SUMMARY.

The wake model is initialized with one set of span-wise vortex segments on each blade. The following algorithm is then employed each wake simulation frame:

- Add a new generation of nodes and vortex segments to the lattices at the rotor blades using equation (3.1) to compute the vortex strengths on the blades.

- Once the wake is "full", each $1/B$ revolutions, replace the oldest revolution's worth of generations with an equivalent vortex ring.
- Compute the induced velocity at the control point of each aerodynamic blade element, at each node of the lattices, and at each ring by summing the contributions from each vortex segment and ring. (A ring is moved by the induced velocity averaged over p points on its circumference).
- Integrate to find the new position of each node in the lattices and each ring using the previously computed induced velocities

3.6 THE REQUIREMENTS OF REAL TIME

Repeated application of equation (3.4) dominates the computational workload of simulating the wake. The time required for one evaluation of (3.4) was estimated by running a loop in which this function was repeatedly called, and then dividing the number of calls by the elapsed time. For a computer with a 400 MHz Pentium II processor, it takes approximately 4.125×10^{-7} seconds to evaluate equation (3.4). The computation of the velocity induced by a ring is different, being based on a table-lookup. However, for simplicity, we assume in the following that the time required by this computation is the same as required to evaluate (3.4).

The number of times this function is called in a wake simulation frame is given by

$$C = (N + pR)(S + R), \tag{3.16}$$

where N is the number of nodes, R is the number of rings, p is the number of points used to compute an average ring velocity, and S is the number of vortex segments in the lattice. If n is the number of nodes per generation per blade, B is the number of rotor blades, and G is the number of generations retained, N is given by

$$N = BGn \tag{3.17}$$

The number of vortex segments is given by

$$S = B(2nG - G - n) = BG(2n-1) - Bn \tag{3.18}$$

Substituting (3.2) and (3.3) into (3.1) gives

$$\begin{aligned} C &= B^2G^2(2n-1)n - B^2Gn^2 \\ &+ RBG[(2p+1)n - p] - pRBn + pR^2 \end{aligned} \tag{3.19}$$

Using typical *Bladehelo* parameters, we set $B=2$ (two blades), and $n=11$ (ten elements per blade). We assume eight wake updates every rotor revolution, which, for ten revolutions retained, gives $G=80$. We also consider ten rings ($R=10$) with four sampling points per ring ($p=4$). With these assumptions, (3.4) will be computed 6,026,400 times each wake step. A single wake frame would spend 2.49 seconds just evaluating equation (3.4). For a UH-1H helicopter, 1/8 of a revolution takes only 0.023 seconds. The wake calculations would be more than 100 times slower than real time.

To allow real-time operation, the number of nodes per generation had to be reduced to only two per rotor blade (Figure 3-7). In this mode, there is only one vortex segment along the blade

instead of one segment per blade element. The strength of the vortex segment is taken to be the average of the strengths of the segments that would have been used for each aerodynamic element. Chord-wise segments are shed only at the blade tips and roots. This is believed to still represent a reasonable model because, in reality, the tip and root vortices are the strongest, they do manifest themselves as discrete vortices, and they have the greatest effect on the induced flow. If only two nodes are used per blade, $C = 174{,}960$, and the time spent on equation (3.4) for one wake frame would be 0.072 seconds. This is still 3.13 times longer than real time.

To achieve real time, parallel processing was employed. The computational burden of the wake model was distributed among sixteen processors. On the face of it, this should have increased throughput by a factor of sixteen. In reality, there is overhead associated with multiprocessing and there are simulation tasks other than evaluating equation (3.4). Real-time operation of the wake model was achieved for a two-bladed rotor with a wake represented only by the tip and root trailing vortices. The lattices were generated over ten revolutions with eight wake steps per revolution, and there were ten vortex rings representing another five previous revolutions.

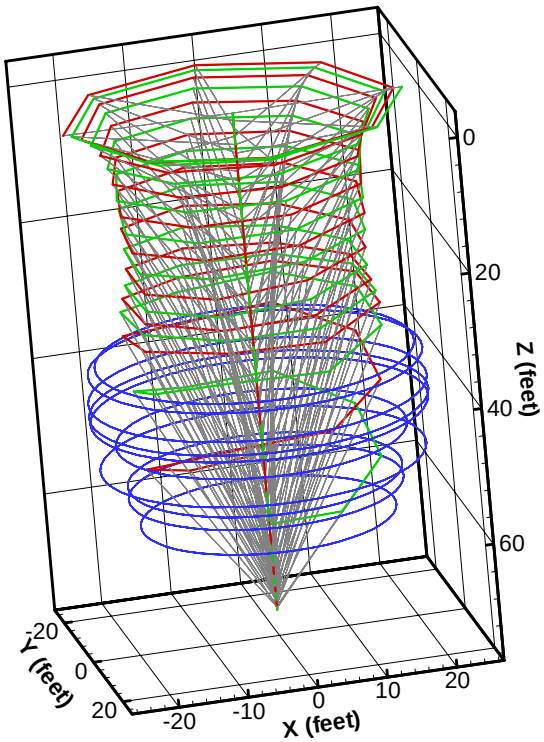


FIGURE 3-7: Ten-revolution real-time wake in OGE hover (consisting of only tip and root trailing vortices and corresponding shed vortices).

3.7 EFFECT OF WAKE MODEL ON FLIGHT CHARACTERISTICS.

Figure 3-8 shows the response of *Bladehelo* to a forward cyclic step input with the vortex lattice wake model. A history of pitch and roll rate is plotted against time. The flight condition is OGE hover. The helicopter is trimmed, during which time the wake builds up and stabilizes. The

helicopter is then released. It remains nearly stationary until three seconds later when a forward cyclic input is applied.

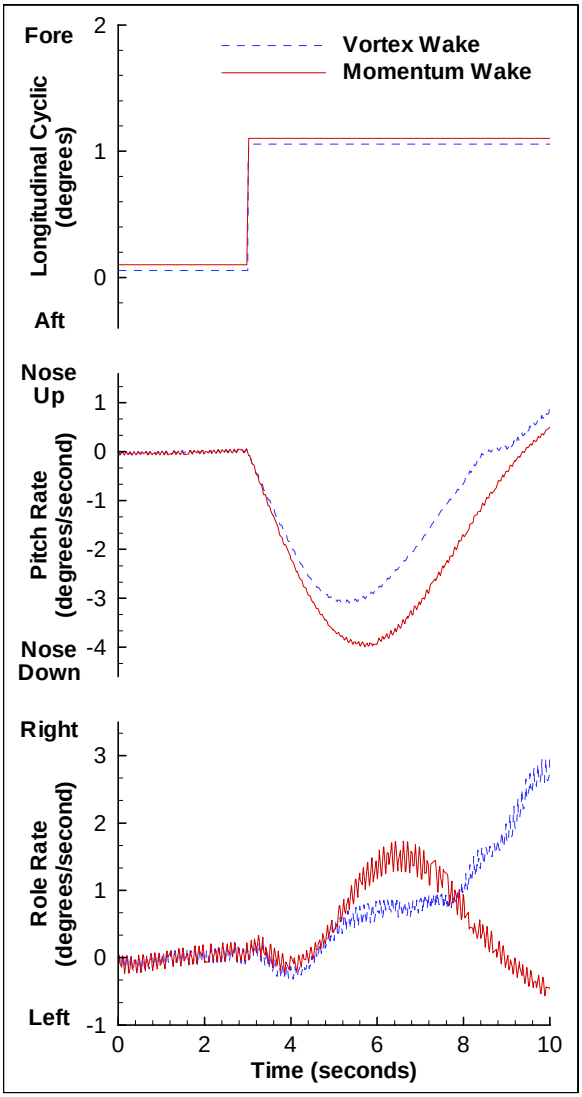


Figure 3-8: Comparison of control response with vortex lattice wake and with uniform inflow.

For comparison, a history based on the preexisting uniform inflow model is also shown. In both cases, the helicopter responds by pitching down and rolling right, however, the details vary. Initially, as the rotor is tilted relative to the shaft, both models roll slightly to the left. This is the effect of "residual bending" [4]. This effect is driven by torque and is not affected by the wake model. Next, as the shaft starts to tilt forward, the tip of each blade sees an increased angle of attack in the front. This causes the rotor to tilt to the right. With uniform inflow, this continues so long as the shaft pitches down. With the wake model, the Rosen-Isser effect of compressed wake in the front [9] eventually takes effect and limits the rate of roll to the right. As the shaft pitches back up, the model with uniform inflow arrests the right roll. But the simulation with the

wake model, possibly driven by an inverse Rosen-Isser effect, accelerates its right roll. This preliminary data does not conform to the specific expectations of references [1] and [3]. Further investigation, as well as validation against test data for the helicopter modeled, is in order.

3.8 CONCLUSIONS

A wake model based on keeping track of the vorticity generated by the rotor has been formulated and coded. The model computes the induced velocity generated by this vortex structure and uses it to propagate the vortex structure itself as well as to produce the induced velocity to be output for use by the rotor model.

This model has been run in real time in interaction with *Bladehelo,* the University of Alabama's helicopter simulation code. The ability to run this extremely computationally intensive wake model in real time was made possible by the combination of several factors:

- Restriction to the simplest case, where only tip and root trailing vortices and the corresponding shed vortices were tracked.
- Simple formulation based on lifting line formalism for the blades.
- Efficient C++ coding.
- Advances in microprocessor technology concurrent with this effort.
- Parallel processing.

Issues that had to be addressed in the course of the development included:

- Methods of smoothing the local effects of modeling a continuous vortex sheet as a lattice.
- Stability of the wake - especially in hover after the wake has almost fully contracted.
- Methods of truncation that have negligible effects at the rotor disk. (The use of rings is the current solution.)

These issues merit continued research.

In real time, *Bladehelo,* equipped with this wake model, flew and handled well, as far as cursory subjective evaluation could probe. However, on the ground, the wake model induced a form of ground resonance.

Serious validation work, as planned for Stage 2 of the project is called for. Data collected in real time as well as off line, must be compared to flight data before the full value, as well as the deficiencies of this wake model, if any, can be determined

4. ARCHITECTURE.

4.1 SIMULATION HOST HARDWARE UPGRADE.

At the start of the current project, in May 1998, the host computer at the UA FDL was named *Archimedes* - a PC with a 200 MHz Pentium Pro microprocessor running MS DOS as the operating system. The host simulation code utilized the microprocessor in 16 bit mode. The simulation frame rate for the body equations was 85 Hz, and the frame rate for the rotor blade dynamics was greater than 300 Hz, using RK4 (fourth order Runge-Kutta) integration, which is the equivalent of 1200 Hz using the AB4 (Fourth Order Adams Bashforth) integration scheme. These rates provided excellent accuracy for the dynamics models that were then in use. However, it was clear that the structured modeling based rotor simulation would require considerably higher frame rates as well as more throughput per frame. Even more so, the vortex lattice wake model requires extensive computational resources. For initial development, *Archimedes* was replaced by *Hamilton* - a 400 MHz Pentium II based PC. This hardware upgrade doubled the performance. *Hamilton* remains on line as the simulation host for the DOS based versions of *Bladehelo.*

4.2 COMPILERS

4.2.1 BORLAND COMPILERS.

To increase the performance further, the simulation code was converted to be a Windows application so that it could utilize the processor in 32 bit mode. Access to hardware devices is accomplished in a much different fashion for the Win32 Application Programmer's Interface (API) than for DOS. Initial benchmarks were obtained with only the model and the main simulation loop converted. The serial communications and the reading of the analog to digital converter, used for the flight controls, were left out. The code was compiled with an Intel created optimizing compiler bundled with the Borland C++ 5.01 compiler. Operating in this fashion, the 32 bit version of the simulation was approximately 2.8 times faster than the 16 bit version. When the serial communications and the reading of the analog flight controls were added to the Win32 version, the speedup over the 16 bit version dropped to a factor of 2.4.

As the program became more complex, however, use of the Intel optimizing compiler became more difficult. Minor changes in the program would cause the compiler to crash unpredictably or to flag previously accepted syntactically correct code as an error. Unexplained changes would be required in unrelated portions of the program to make compilation possible. The amount of time it took to successfully compile the simulation after each minor change made the optimizing compiler impractical. In addition, the results obtained with code compiled by the optimizing compiler at times varied from the results obtained with the Borland C++ 5.01 compiler. For these reasons, the Intel compiler was abandoned and the standard Borland C++ compilers, versions 5.01 and 5.02, were used. These compilers provided only about 30% speed improvement over the DOS version.

4.2.2 CODE WARRIOR.

The most recent version of the Borland C++ compiler was several years old. An effort was made to identify a more modern compiler that could produce executables with better performance. The first one tried was the Metroworks CodeWarrior C++ 4.0 compiler. With optimizations turned on, the resulting executable ran just under 70% faster than the 32 bit Borland version. The compiler was very stable, even with optimizations turned on. The C++ syntax supported very closely follows the ANSI C++ standard, making it more convenient for programming. Compilation times with optimization on, however, are very long. A full compile of the simulation host program could take up to 40 minutes. Use of the compiler for development work was more practical with optimizations turned off. In this mode, a complete compile takes 5-10 minutes. Compile times were kept down by only compiling with optimization for the final versions and through the use of project files that only rebuild the files where a source code change has occurred.

4.2.3 INTEL VTUNE ENVIRONMENT / MICROSOFT VISUAL C++.

An updated version of the Intel optimizing C++ compiler is available directly from Intel as part of their VTune Performance Enhancement Environment. It is designed to take maximum advantage of Intel microprocessors up to a Pentium III processor. As part of an educational institution support program, Intel donated a copy of the latest version, 4.0, of the VTune package to the UA FDL. The Intel compiler recognizes the same C++ syntax as Microsoft Visual C++ version 6.0 and can be used as a plug-in for the Visual C++ developer's studio. Microsoft C++ has some deviations from the ANSI C++ standard, which necessitated modifications to the FDL source code before it could be compiled with either the Microsoft or the Intel compiler.

Compile time using the Intel compiler is significantly longer than for the Microsoft compiler, although it is shorter than for the Code Warrior package with optimizations on. Compilation of *Bladehelo* from scratch requires about 25 minutes. To avoid the long compile times during development, the Microsoft compiler was employed until the conversion of the code was complete. The executable resulting from the Microsoft compiler ran slower than the executable produced by the Borland C++ compiler. Once the incompatibilities were resolved, the Intel compiler was employed. The resulting executable runs about 70% faster than the Borland compiled program, and is, therefore, about twice as fast as the 16 bit DOS version.

Because the compile times are significantly shorter, and the resulting program is slightly faster, the Intel compiler is currently the preferred tool. The VTune package also includes performance analysis tools that can be used to improve computational efficiency

4.3 DISTRIBUTED HOST PROCESSING.

It was anticipated that, even with the added speed provided by 32 bit operation, structured modeling (chapter 2), and in particular the vortex lattice wake (chapter 3), would require substantial added hardware resources to achieve real-time operation. Selecting and acquiring the new hardware, as well as installing and integrating it was an important part of the present project. A system of multiple networked computers, each with multiple CPUs, was selected. It consists of four computers that we named *Monster0, Monster1, Monster2,* and *Monster3*. Each

"monster" is a Dell 6300 PowerEdge server with four Intel 400 MHz Pentium II processors. The "monsters" are interfaced with a SCRAMNet 150 megabit fiber optic ring network (see section 4.3.2). An additional computer named *Janus*, which was not obtained with project funds, was also added to the network. *Janus* is a Dell Precision workstation with two Intel 450 MHz Pentium II processors. The rotor model was converted to run on a monster with separate processors computing the history of different blades. The wake computation task has been converted to run on up to all sixteen processors of the four "monsters" and interface with the helicopter running on *Janus*.

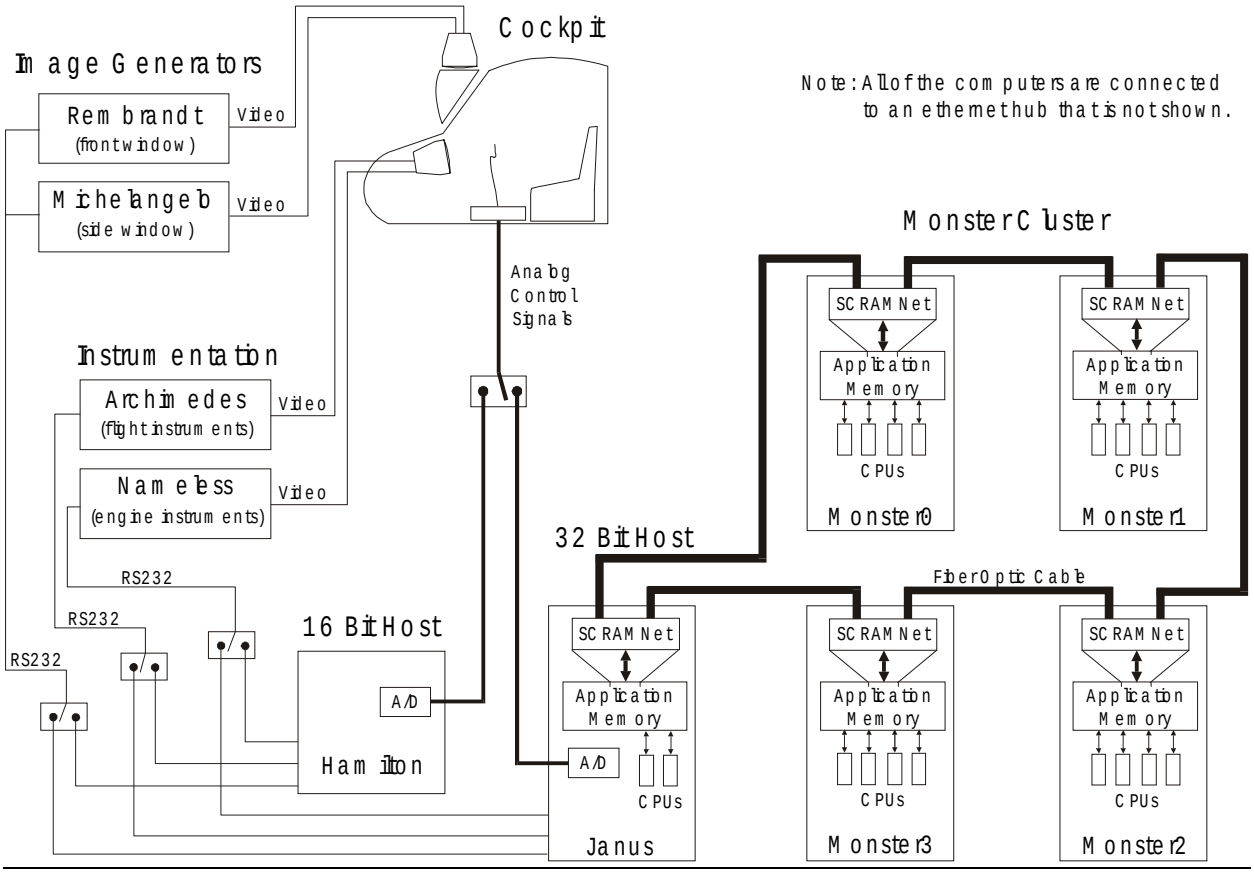


Figure 4-1: Simulator Architecture

4.3.1 SOFTWARE ARCHITECTURE FOR MULTI PROCESSING

When adapting a model for multiprocessing, it is necessary to identify independent tasks that can be performed simultaneously. Computations that depend on the results of previous calculations must be executed in the proper order and contention for shared system resources must be avoided. Memory is the principal shared resource in the system used for the wake simulation. It must be ensured that no individual memory location is written to by more than one processor at the same time, and that no processor reads a memory location at the same time that another processor is writing to it.

Parallel processing is performed on two levels. First, the computations are divided among multiple computers (i.e., the four “monsters”). On a second level, the tasks required of each computer are distributed among its processors (four per monster).

4.3.2 PARALLEL PROCESSING APPLIED TO THE ROTOR MODEL

In the case of Structured Modeling of a helicopter rotor, it was difficult to find operations that lent themselves well to parallel processing. The only parallelism that suggested itself was performing the computations pertaining to different blades in parallel. Since the number of blades rarely exceeds four, there was no point utilizing more than the four processors inside a single "Monster" computer. This last task is achieved by launching separate threads and relying on the operating system to assign them to the available processors.

Section 2.6.1 describes how the Structured Modeling code was organized to launch separate threads for separate branches of the tree when branching actually occurs. This achieved limited parallel processing for the rotor model. The performance gains were quite modest. A three bladed fully articulated rotor (representing the Hughes 269A) achieved a performance gain by a factor of 1.6 when split into three threads.

4.3.3 PARALLEL PROCESSING APPLIED TO THE WAKE MODEL

The case of the wake model is quite different and much better suited for multiprocessing. The wake computation that consumes the most processing time, computing the induced velocity at every lattice node, lends itself very well to parallel processing. For a given configuration of the wake lattices, equation (3.4) can be applied to multiple nodes simultaneously. The computation for each node is completely independent from the other nodes. Also, for a node, the induced velocity is computed by summing the contributions from every vortex segment using equation (3.4). Since the contribution from each segment is independent, the summation can be performed in parts over subsets of the segments simultaneously by multiple processes. The results obtained by each process can then be summed to get the total velocity.

Parallel use of all sixteen processor, i.e., all four "Monsters" is indicated. One instance of the wake program, a process, is run on each computer that is dedicated to the wake simulation. The computers are connected with a SCRAMNet 150 megabit fiber optic ring network. SCRAMNet is a commercial product designed by the Systran Corporation with real-time simulation in mind. Each computer in a SCRAMNet network has a SCRAMNet expansion card installed. Each card has onboard random access memory (RAM), which can be mapped into an application’s address space as shared memory. Write operations to shared memory by one of the computers causes the SCRAMNet card to transmit the data to all of the other cards in the network. Programs running on the other computers can then read the new values from their copy of the shared memory. The amount of shared memory in the installation used for the work reported here was 1 megabyte. The networking is transparent to the programs running on the separate computers. The SCRAMNet memory is accessed in the same manner as if all of the programs were running on the same computer. SCRAMNet requires no networking protocols and achieves extremely fast response and impressive throughput. Almost all networking overhead for the simulation program is eliminated.

Even so, the fact that the physical memory resides on an expansion card in each machine cannot be ignored. Because the contents of the memory on a SCRAMNet card can be modified independently of the operations of the processors in the computer, the contents cannot be stored in a RAM cache. Any calculation requiring data stored in SCRAMNet memory will cause that data to be read into the processor through the PCI bus directly from the SCRAMNet card. This will happen each time the data is needed, even if the same piece of data is read repeatedly. Reading data across the PCI bus is much slower than reading information from a RAM cache.

The calculations performed for the wake model are extremely repetitive. Storing all of the wake model data in the SCRAMNet shared memory, instead of system memory, causes a severe loss of computational efficiency. In preliminary trials of the four computer, sixteen processor configuration, with all of the wake data stored in SCRAMNet, the wake model ran as much as ten times slower than on a single processor using only system memory. To avoid this pitfall, only the data that must be shared was placed in SCRAMNet memory. Also, data in SCRAMNet memory that is needed repeatedly is copied into system memory before use.

For the wake simulation, the blade element positions and the lift per unit span are stored in SCRAMNet memory, as are flags used to synchronize the wake processes. Each process maintains a complete copy of the node data in its own system memory. The strengths of all of the vortex segments and the connectivity information are also stored in system memory by every wake process. One vector value per lattice node is maintained in the shared SCRAMNet memory and used to communicate intermediate values of the induced velocity at the node between the processes.

For the purpose of computing the contributions of multiple lattice segments to the induced velocity at each point in parallel, the lattice nodes are logically partitioned into a number of zones equal to the number of processes. Each zone contains an equal (or as equal as possible) number of nodes. Responsibility for a subset of the vortex segments is also assigned to each process. The number of segments assigned to a process is proportional to the speed of the computer running the process. The work done by a process is proportional to the number of nodes and to the number of segments. Computations will be done for the same number of nodes by each process. Therefore, by adjusting the number of segments assigned to a process, the loads can be balanced to accommodate computers with different speeds in the same network.

Each wake simulation step, the induced velocity must be computed at each lattice node so that the node's position can be numerically propagated forward in time. Each wake process begins work on a different node zone. For each node in that zone, the process computes the contribution to the induced velocity by the segments assigned to it. All of the computations are performed using data stored in system memory. The result for each node is copied to SCRAMNet memory. After all of the processes are finished working on their starting zones, the processes switch zones. The contributions to induced velocity are computed for each node in the new zones and the results are added to the values previously stored in SCRAMNet memory. This continues in a round robin fashion until each process has treated all of the nodes. At this point, the total induced velocity at each node is stored in SCRAMNet memory.

For each node in the zone with which it started, each process uses the induced velocities stored in shared memory and the node positions stored in its system memory to propagate the node positions forward in time. The new node positions are then written back to SCRAMNet memory. After all of the processes have updated the node positions for their zone, each process copies the new positions for the other zones from SCRAMNet memory to its system memory. Each wake process now has all of the new node positions and is prepared to begin the cycle over. On a lower level, each of the computers has multiple processors, in our case: four. To utilize the multiple processors, the wake program has been designed to use the multi-threading capabilities of the Windows NT 4 operating system. A computational thread represents a task that can be independently scheduled by the operating system and executed by a processor simultaneously with other threads being executed by other processors. All of the threads of a single process have access to the process' global data and to the heap, but each thread has its own stack area for local variables and subroutine calls. At any instant, Windows NT will schedule one thread per processor, if the threads are available to be executed.

To take advantage of multiple threads, each process further subdivides each node zone into as many groups as there are threads. When a process is working on a zone, each thread performs the computations for a different portion of that zone.

To assess the computational advantage gained by parallel processing, and the loss to overhead, the wake program was timed running in three modes. The first mode was with the wake computations performed by a single processor, with all of the wake data stored in system memory. In the second configuration, the wake computations were performed by four processors in one "monster" computer using multi-threading. Again, all data was stored in system memory. For this test, the wake computations were performed 3.98 times faster. The loss to overhead was, therefore, negligible. Finally, the wake computations were split among all four "monster" computers with each computer using four processors. SCRAMNet memory was utilized for the data that had to be shared between computers, as previously described. This configuration performed the computations 13.40 times faster than for one processor. The difference between this factor and the factor of 16 that corresponds to the number of processors employed is attributed to the overhead required to coordinate between the four computers and the losses involved in passing data across SCRAMNet. For each of theses three tests, reasonable dummy rotor data was generated by the master wake process. Therefore, these factors do not include any time required to interface to the helicopter simulation program, *Bladehelo*.

4.4 WAKE PROGRAM INTERFACE.

The rotor wake model is a self-contained program. The interface to the wake model is through SCRAMNet shared memory. A program with a rotor model that uses the wake model accesses it in three phases: a startup phase, an update phase that is used during a simulation run, and a shutdown phase.

4.4.1. STARTUP PHASE

During the startup phase, the rotor program places certain data structures describing the rotor system into SCRAMNet memory. The first structure, called `WakeInterfaceData` by the

wake program, contains parameters describing the rotor system as a whole. Only one of these structures is created in shared memory. The contents of this structure are detailed in table 4-1.

The wake program views the rotor system as a series of aerodynamic elements. A `WakeElementData` structure (table 4-2) contains information about a single aerodynamic element. The rotor program places into shared memory one instance of `WakeElementData` for each aerodynamic element of each rotor blade. These structures must immediately follow the `WakeInterfaceData` structure in memory and be arranged in the same order as the elements are arranged along each rotor blade, beginning with each blade's root element and ending with that blade's tip element. An array of structures, each with three double precision elements that represent the vector components of the inertial position of the tip of each blade, is also placed in shared memory immediately following the `WakeElementData` structures. The order of the vector blade tip positions in the array must correspond to the blade order in the preceding data.

The rotor code must also allocate two SCRAMNet interrupts to be used as flags for synchronization. One interrupt must be allocated in a sending mode and the other in a receiving mode.

TABLE 4-1. WAKEINTERFACEDATA STRUCTURE CONTENTS

Variable Name	Bytes, Int/Float	Description
`num_blades`	4, I	number of rotor blades in the rotor system
`num_elements_per_blade`	4, I	number of aerodynamic elements per blade
`num_wake_steps_per_revolution`	4, I	number of wake steps done per rotor revolution
`rotor_radius`	8, F	nominal radius of the rotor disc (ft)
`air_density`	8, F	density of the air (slug/ft^3)
`time_step`	8, F	time length of one wake step (sec)
`Mode`	4, F	program mode (OPERATE, RESET, etc.)
`step_counter`	4, I	number of wake passes
`Simulated_elapsed_time`	8, F	elapsed time in the simulation (sec)
`hub_position_inertial`	3× 8, F	inertial position vector of the rotor hub (ft)
`inertial_to_azimuth`	9× 8, F	inertial to azimuth passive rotation matrix
`uniform_induced_velocity`	3× 8, F	momentum theory induced velocity (ft/sec)
`inertial_to_shaft`	9x 8, F	inertial to rotor shaft passive rotation matrix

A C++ class, `SCRAMNet`, has been created to facilitate interactions with the SCRAMNet hardware, including allocation of memory and interrupts. It is possible to arrange the structures in memory without the use of the `SCRAMNet` class, however, coordination between the two programs becomes more complicated. The following code fragment, which employs the

SCRAMNet class, allocates a block of data in SCRAMNet to be used as the shared memory interface between the rotor program and the wake program:

```
volatile void* scramnet_block_ptr =
SCRAMNet::allocateMemory(
  sizeof(WakeInterfaceData) +
  num_elements_per_blade *num_blades * sizeof(WakeElementData) +
  num_blades * sizeof(Vector),
 wake_scramnet_block_name, true, 0);
```

The first parameter in this subroutine call represents the size of the memory block to be allocated, in bytes. In this example, num_blades is the number of rotor blades in the rotor system, and num_elements_per_blade is the number of aerodynamic elements per rotor blade. Vector is a data type that is comprised of three double precision floating point values. The second-to-last parameter is a Boolean value indicating whether the block of data should be initialized before it is returned. The last parameter is the value with which the block should be filled if it is to be initialized. wake_scramnet_block_name is a variable of type string that specifies the name of the block of data. This name is stored in shared memory with the data block and is used by the two programs to uniquely specify the block of SCRAMNet memory. When the wake program begins, it waits until a block of memory with the correct name is allocated in shared memory. The following code fragments can be used to allocate the SCRAMNet interrupts:

```
wake_start_pass_interrupt =
SCRAMNet::allocateInterrupt(string("START:")
                                    +wake_scramnet_block_name);

wake_pass_done_interrupt =
SCRAMNet::allocateInterrupt(string("DONE:")
                                 + wake_scramnet_block_name,
                               &wake_pass_done_event);
```

TABLE 4-2. WAKEELEMENTDATA STRUCTURE CONTENTS

Variable Name	Bytes, Int/Float	Description
length	8, F	length of element (ft)
in_plane_air_speed	8, F	magnitude of the air relative velocity of the control point minus the span-wise component (ft/sec)
lift_per_unit_span	8, F	lift force per unit span at the control point (lb/ft)
Position	3× 8, F	inertial position of the root end of the element (ft)
control_point_position	3× 8, F	inertial position of the control point (ft)
control_point_air_velocity	3× 8, F	absolute velocity of the air at the control point (ft)

`wake_start_pass_interrupt` and `wake_pass_done_interrupt` are both variables of type `SCRAMNet::InterruptHandle`.

In the setup phase, after the data block and the interrupts have been allocated, the rotor code initializes those values stored in shared memory that remain constant during execution of the wake and rotor models. In the `WakeInterfaceData` structure, these values are the number of aerodynamic elements per blade, the nominal rotor disc radius, and the number of blades. Although the interface to the wake program has been written to allow the air density and the wake time step to change during simulation, these values are also currently initialized in the setup phase and remain unchanged during normal operation. The mode, an enumerated type, is also set in this phase to the value STARTUP (which currently has the numeric value 5). The element lengths are assumed constant and, therefore, they must be initialized in the instances of `WakeElementData` during the setup phase.

The wake program cannot begin its initialization until this configuration data has been set by the rotor program. The value for the number of blades is used as a flag for the rotor program to signal the wake program that the appropriate data has been initialized in shared memory. Therefore, the variable for the number of blades must not be changed from zero to its actual value until all of the other configuration data has been set.

Having initialized the shared memory data, the rotor program waits on the event that the wake program uses to signal completion of a wake pass. Using the event from the example above, this could be accomplished using the following code:

```
wake_pass_done_event.waitForEvent(max_milliseconds_to_wait);
```

where `max_milliseconds_to_wait` is an integer value indicating an appropriate timeout value in milliseconds. The function returns true if the event is signaled or becomes signaled before the timeout expires. It returns false otherwise. The rotor program should reset the event to a non-signaled state after the wait function returns. This is accomplished by the call:
`wake_pass_done_event.reset();`

The wake program triggers the SCRAMNet interrupt that sets this event when it completes its initialization and is ready to begin simulation looping. When this interrupt is received and the event is reset by the rotor program, the wake setup phase is complete.

4.4.2. UPDATE PHASE

The update phase is typically controlled by the rotor program, while it is in a loop, simulating the rotor in operation. During the update phase, the rotor program updates shared memory data that changes on a frame-by-frame basis, except for the air velocity at the control point, in the `WakeElementData` structure. After updating the data, the rotor program signals the wake program to perform a pass. This is done by sending a SCRAMNet interrupt. Using the interrupt handle from the example above:

```
SCRAMNet::sendInterrupt(wake_start_pass_interrupt);
```

After receiving the interrupt calling for the start of a pass, the wake program updates in shared memory the absolute velocity of the air in inertial coordinates at each blade element control point. The wake then sends the interrupt indicating its pass is complete. The rotor code is waiting for this interrupt, and resets it after it is received. The data returned by the wake model can be left in SCRAMNet memory and accessed as needed, or it can be immediately copied to a more convenient location.

The update phase should be performed in the same manner to accomplish a reset of the wake program, except that the mode variable should be set to RESET (4), as opposed to OPERATE (1) which is the setting while the rotor program is in the simulation loop

.

4.4.3. SHUTDOWN PHASE

The shutdown phase exists merely to indicate to the wake program that the rotor program no longer requires its services. The rotor program sets the mode variable to the value SHUTDOWN (6) and sends the SCRAMNet interrupt that prompts the wake to begin a pass. Upon receipt of the interrupt, the wake program terminates.

The rotor wake model is a self-contained program. It interfaces the helicopter simulation through a structured interface that is defined in this section. The rotor model specifies to the wake the initial positions of vortex nodes, the strength of the bound vortices and the control points where the induced velocity is required. The wake model keeps track of the vortex lattice and propagates it in time. It returns to the rotor model the induced velocities at the control points. The wake model does not know or care about the type of rotor model that is employed. To illustrate, the same wake model program ran successfully with *Bladehelo* versions BH (rigid blades) and BS (flexible blades).

REFERENCES

1. Katz, A., “Dynamic Simulation Models for the UH1-H and TH-67 Helicopters,” UA FDL report 97S04A, Prepared for Anacapa Sciences, Inc. under Subcontract ASI-97-DHH-1058-13 from US Army Research Institute, Ft. Rucker, AL., October 30, 1997.
2. Graham, K., and Katz, A., “A Blade Element Model in a Low Cost Helicopter Simulator,” AIAA Paper 94-3436-CP, Proceedings of the AIAA Flight Simulation Technologies Conference, Scottsdale, Arizona, August 1-3, 1994, pp 287-293.
3. Katz, A. and Graham, K. “Control Issues for Rigid-in-Plane Helicopter Rotors,” Journal of Aircraft vol. 33, #2, pp. 311-315, March-April 1996.
4. Katz, A., “Residual Shaft Bending by Helicopter Rotors,” Journal of Aircraft vol. 34, #5, pp. 688-690, September-October 1997.
5. Katz, A. and Waits, T. “Global Recursive Dynamics of Articulated Tree Structures,” Computational Mechanics vol. 23 #4 pp. 288-298, May 1999
6. A. Katz, J. Van Pelt, T. Waits and W. Liu, “Simulator Construction Set (SCS) Phase 1”, UA FDL report 95S01B prepared for Loral Western Development Labs, San Jose, CA under Subcontract SO-242825-A, item 05, January 8, 1996.
7. Prouty, R. W., "The Case of the Cross-Coupling Mystery," Rotor & Wing, June 1994 pp. 48-49
8. Rosen, A. and Isser, A. “A New Model of Rotor Dynamics During Pitch and Roll of a Hovering Helicopter,” Journal of the American Helicopter Society vol. 40, #3, pp.17-28, July 1995.
9. Rosen, A. and Isser, A. “A Model of the Unsteady Aerodynamics of a Hovering Helicopter Rotor that includes Variations of the Wake Geometry,” Journal of the American Helicopter Society vol. 40**,** #3, pp. 6-16, July 1995.
10. Sharma, M., Graham, K., and Huckaba, J., "PC Based Visual Image Generator For Simulation," UA FDL Report 97S03B, October 30, 1997.