Online Appendix to: Design Patterns for Multiphysics Modeling in Fortran 2003 and C++

DAMIAN W. I. ROUSON and HELGI ADALSTEINSSON
Sandia National Laboratories
and
JIM XIA
IBM Corporation

APPENDIX A. FORTRAN 2003 IMPLEMENATION

A.1 Semidiscrete Example

main.f03

```
1  program main
2    use lorenz_module ,only : lorenz,integrate
3    implicit none  ! Prevent implicit typing
4
5    ! This code integrates the Lorenz equations over time using abstractions that
6    ! follow the Semi-Discrete design pattern of Rouson, Adalsteinsson and Xia
7    ! (ACM TOMS 2010).
8
9    type(lorenz)       :: attractor
10   integer            :: step  ! time step counter
11   integer ,parameter :: num_steps=2000,space_dimension=3 ! phase space dimension
12   real    ,parameter :: sigma=10.,rho=28.,beta=8./3.,dt=0.01 ! Lorenz parameters and time step size
13   real    ,parameter ,dimension(space_dimension) &
14                       :: initial_condition=(/1.,1.,1./)
15
16   call attractor%construct(initial_condition,sigma,rho,beta)
17   print *,attractor%output()
18   do step=1,num_steps
19     call integrate(attractor,dt)
20     print *,attractor%output()
21   end do
22 end program
```

lorenz.f03

```
1  module lorenz_module
2    use integrable_model_module ,only : integrable_model,integrate
3    implicit none
4
5    private             ! Hide everything by default
6    public :: integrate ! Expose time integration procedure
7
8    ! This type implements operators required for integration by the above procedure.
9    ! The time derivative function is defined to express the Lorenz equations.
10
11   type ,extends(integrable_model) ,public :: lorenz
12     private
13     real ,dimension(:) ,allocatable :: state   ! solution vector
14     real :: sigma ,rho ,beta                   ! Lorenz parameters
15   contains
16      procedure ,public :: d_dt      => dLorenz_dt      ! time derivative
17      procedure ,public :: add       => add_lorenz      ! add two lorenz objects
18      procedure ,public :: multiply  => multiply_lorenz ! multiply a lorenz object by a real scalar
19      procedure ,public :: assign    => assign_lorenz   ! assign one lorenz object to another
20      procedure ,public :: construct                    ! constructor
21      procedure ,public :: output                       ! accessor: return solution vector
22   end type lorenz
23
24 contains
25
26   subroutine construct(this,initial_state,s,r,b)  ! constructor
```


DOI 10.1145/1644001.1644004 http://doi.acm.org/10.1145/1644001.1644004

```
    class(lorenz)      ,intent(out) :: this
    real ,dimension(:) ,intent(in)  :: initial_state
    real               ,intent(in)  :: s ,r ,b    ! passed values for sigma, rho and beta
    !allocate(this%state(size(initial_state)))
    this%state=initial_state
    this%sigma=s; this%rho=r; this%beta=b
  end subroutine construct

  function output(this) result(coordinates)  ! accessor: return solution vector
    class(lorenz)      ,intent(in)  :: this
    real ,dimension(:) ,allocatable :: coordinates
    !allocate(coordinates(size(this%state)))
    coordinates = this%state
  end function output

  function dLorenz_dt(this) result(dState_dt) ! time derivative: encapsulates Lorenz equations
    class(lorenz)           ,intent(in)  :: this
    class(integrable_model) ,allocatable :: dState_dt
    type(lorenz)            ,allocatable :: delta

    allocate(delta)
    allocate(delta%state(size(this%state)))
    delta%state(1)=this%sigma*( this%state(2) -this%state(1))            ! 1st lorenz equation
    delta%state(2)=this%state(1)*(this%rho-this%state(3))-this%state(2)   ! 2nd lorenz equation
    delta%state(3)=this%state(1)*this%state(2)-this%beta*this%state(3)    ! 3rd lorenz equation
    delta%sigma=0.  ! hold Lorenz parameters constant over time
    delta%rho=0.
    delta%beta=0.
    call move_alloc (delta, dState_dt)
  end function

  function add_Lorenz(lhs,rhs) result(sum) ! add two Lorenz objects
    class(lorenz)           ,intent(in)  :: lhs
    class(integrable_model) ,intent(in)  :: rhs
    class(integrable_model) ,allocatable :: sum
    type(lorenz)            ,allocatable :: local_sum ! obviate need for 'select type(sum)'

    allocate (lorenz :: local_sum)
    !allocate(local_sum%state(size(lhs%state)))
    select type(rhs)
      class is (lorenz)
        local_sum%state = lhs%state + rhs%state
        local_sum%sigma = lhs%sigma + rhs%sigma
        local_sum%rho   = lhs%rho   + rhs%rho
        local_sum%beta  = lhs%beta  + rhs%beta
      class default
        stop 'add_Lorenz: rhs argument type not supported'
    end select
    call move_alloc(local_sum, sum)
  end function

  function multiply_Lorenz(lhs,rhs) result(product) ! multiply a Lorenz object by a real scalar
    class(lorenz) ,intent(in)  :: lhs
    real          ,intent(in)  :: rhs
    class(integrable_model) ,allocatable :: product
    type(lorenz)            ,allocatable :: local_product ! obviate need for 'select type(sum)'

    allocate (local_product)
    local_product%state = lhs%state*rhs
    local_product%sigma = lhs%sigma*rhs
    local_product%rho   = lhs%rho  *rhs
    local_product%beta  = lhs%beta *rhs
    call move_alloc(local_product, product)
  end function

  subroutine assign_lorenz(lhs,rhs)  ! assign one lorenz object to another
    class(lorenz)           ,intent(inout) :: lhs
    class(integrable_model) ,intent(in)    :: rhs
    !if (.not. allocated(lhs%state)) allocate(lhs%state(size(rhs%state)))
    select type(rhs)
      class is (lorenz)
        lhs%state = rhs%state
        lhs%sigma = rhs%sigma
        lhs%rho   = rhs%rho
        lhs%beta  = rhs%beta
      class default
        stop 'assign_lorenz: rhs argument type not supported'
    end select
  end subroutine
end module lorenz_module
```

integrable_model.f03

```
module integrable_model_module
  implicit none       ! Prevent implicit typing
  private             ! Hide everything by default
  public :: integrate ! expose time integration procedure

  ! This stateless type specifies the operators required to support Runge-Kutta time integration,
  ! while deferring the actual implementation of those operators to extensions (children) of this type.

  type ,abstract ,public :: integrable_model
  contains
```

```
    procedure(time_derivative      ) ,deferred :: d_dt     ! time derivative
    procedure( symmetric_operator  ) ,deferred :: add      ! add two integrable_model objects
    procedure( symmetric_assignment) ,deferred :: assign   ! assign one integrable_model to another
    procedure(asymmetric_operator  ) ,deferred :: multiply ! multiply an integrable_model by a real scalar
    generic  :: operator(+) => add      ! Map operators to corresponding proceures
    generic  :: operator(*) => multiply
    generic  :: assignment(=) => assign
  end type integrable_model

  abstract interface
    function time_derivative(this) result(dState_dt)
      import :: integrable_model
      class(integrable_model) ,intent(in)  :: this
      class(integrable_model) ,allocatable :: dState_dt
    end function time_derivative
    function symmetric_operator(lhs,rhs) result(operator_result)
      import :: integrable_model
      class(integrable_model) ,intent(in)  :: lhs,rhs
      class(integrable_model) ,allocatable :: operator_result
    end function symmetric_operator
    function asymmetric_operator(lhs,rhs) result(operator_result)
      import :: integrable_model
      class(integrable_model) ,intent(in)  :: lhs
      class(integrable_model) ,allocatable :: operator_result
      real                    ,intent(in)  :: rhs
    end function asymmetric_operator
    subroutine symmetric_assignment(lhs,rhs)
      import :: integrable_model
      class(integrable_model) ,intent(in)    :: rhs
      class(integrable_model) ,intent(inout) :: lhs
    end subroutine symmetric_assignment
  end interface

contains

  subroutine integrate(model,dt)    ! Explicit Euler time integration
    class(integrable_model) :: model
    real ,intent(in)        :: dt   ! time step size (integration interval)
    model = model + d_dt(model)*dt  ! Explicit Euler formula
  contains
    function d_dt(this) result(dThis_dt) ! support d_dt(arg) time differentiation syntax
      class(integrable_model) ,intent(in)  :: this
      class(integrable_model) ,allocatable :: dThis_dt
      allocate(dThis_dt,source=this)
      dThis_dt = this%d_dt()
    end function
  end subroutine
end module integrable_model_module
```

A.2 Strategy and Surrogate Example

main.f03

```
program main
  use lorenz_module ,only : lorenz
  use timed_lorenz_module ,only : timed_lorenz
  use explicit_euler_module ,only : explicit_euler
  use runge_kutta_2nd_module ,only : runge_kutta_2nd

  ! This code uses the strategy and surrogate patterns described by Rouson, Adalsteinsson and
  ! Xia (ACM TOMS 2010) to solve the Lorenz equations.

  implicit none ! Prevent implicit typing

  type(explicit_euler)  :: lorenz_integrator        ! Time integration strategy
  type(runge_kutta_2nd) :: timed_lorenz_integrator  ! Time integration strategy
  type(lorenz)          :: attractor                ! Lorenz equation/state abstraction
  type(timed_lorenz)    :: timed_attractor          ! Time-stamped Lorenz eq./state abstraction
  integer               :: step                     ! Time step counter
  integer ,parameter    :: num_steps=2000,space_dimension=3
  real    ,parameter    :: sigma=10.,rho=28.,beta=8./3.,dt=0.01 ! Lorenz parameters and step size
  real    ,parameter ,dimension(space_dimension) &
                        :: initial_condition=(/1.,1.,1./)

  call attractor%construct(initial_condition,sigma,rho,beta,lorenz_integrator) !Initialize and choose strategy
  print *,'lorenz attractor:'
  print *,attractor%output()
  do step=1,4*num_steps  ! run explicit Euler at increased resolution for comparison to RK2
    call attractor%integrate(dt/4.)
    print *,attractor%output()
  end do

  call timed_attractor%construct(initial_condition,sigma,rho,beta,timed_lorenz_integrator) !Re-initialize, choose new strategy
  print *,''
  print *,'timed_lorenz attractor:'
  print *,timed_attractor%output()
  do step=1,num_steps
    call timed_attractor%integrate(dt)
    print *,timed_attractor%output()
  end do
end program main
```

lorenz.f03

```
module lorenz_module
  use strategy_module ,only : strategy ! Abstract time integration strategy
  use integrable_model_module ,only : integrable_model ! Abstract integrand

  implicit none ! Prevent implicit typing
  private       ! Hide everything by default

  public :: integrable_model ! Expose integrable_model

  type ,extends(integrable_model) ,public :: lorenz
    private
    real ,dimension(:) ,allocatable :: state   ! solution vector
    real :: sigma ,rho ,beta                   ! Lorenz parameters
  contains
     procedure ,public :: construct                   ! Constructor: allocate and initialize
     procedure ,public :: d_dt      => dlorenz_dt      ! Time derivative (specifies evolution equations)
     procedure ,public :: add       => add_lorenz      ! Add two instances
     procedure ,public :: multiply  => multiply_lorenz ! Multiply an instance by a real scalar
     procedure ,public :: assign    => assign_lorenz   ! Assign one instance to another
     procedure ,public :: output                       ! Accessor: return state
  end type

contains

  subroutine construct(this,initial_state,s,r,b,this_strategy) ! Constructor: allocate and initialize
    class(lorenz)      ,intent(out) :: this
    real ,dimension(:) ,intent(in)  :: initial_state
    real               ,intent(in)  :: s ,r ,b
    class(strategy)    ,intent(in)  :: this_strategy
   !allocate(this%state(size(initial_state)))
    this%state=initial_state
    this%sigma=s; this%rho=r; this%beta=b
    allocate (this%quadrature, source=this_strategy)
  end subroutine construct

  function dLorenz_dt(this) result(dState_dt) ! Time derivative (specifies evolution equations)
    class(lorenz)           ,intent(in)  :: this
    class(integrable_model) ,allocatable :: dState_dt
    type(lorenz)            ,allocatable :: local_dState_dt ! obviates need for 'select type(dState_dt)'

    allocate(local_dState_dt)
    allocate(local_dState_dt%state(size(this%state)))
    local_dState_dt%state(1) = this%sigma*( this%state(2) -this%state(1))           ! 1st Lorenz equation
    local_dState_dt%state(2) = this%state(1)*(this%rho-this%state(3))-this%state(2) ! 2nd Lorenz equation
    local_dState_dt%state(3) = this%state(1)*this%state(2)-this%beta*this%state(3)  ! 3rd Lorenz equation
    local_dState_dt%sigma    = 0.
    local_dState_dt%rho      = 0.
    local_dState_dt%beta     = 0.
    call move_alloc(local_dState_dt,dState_dt)
  end function

  function add_lorenz(lhs,rhs) result(sum) ! Add two instances
    class(lorenz)           ,intent(in)  :: lhs
    class(integrable_model) ,intent(in)  :: rhs
    class(integrable_model) ,allocatable :: sum
    type(lorenz)            ,allocatable :: local_sum  ! obviates need for 'select type(sum)'

    select type(rhs)
      class is (lorenz)
        allocate(local_sum)
       !allocate(local_sum%state(size(lhs%state)))
        local_sum%state = lhs%state + rhs%state
        local_sum%sigma = lhs%sigma + rhs%sigma
        local_sum%rho   = lhs%rho   + rhs%rho
        local_sum%beta  = lhs%beta  + rhs%beta
      class default
        stop 'assig_lorenz: unsupported class'
    end select
    call move_alloc(local_sum,sum)
  end function

  function multiply_lorenz(lhs,rhs) result(product) ! Multiply an instance by a real scalar
    class(lorenz)           ,intent(in)  :: lhs
    real                    ,intent(in)  :: rhs
    class(integrable_model) ,allocatable :: product
    type(lorenz)            ,allocatable :: local_product ! obviates need for 'select type(product)'

    allocate(local_product)
   !allocate(local_product%state(size(lhs%state)))
    local_product%state = lhs%state* rhs
    local_product%sigma = lhs%sigma* rhs
    local_product%rho   = lhs%rho  * rhs
    local_product%beta  = lhs%beta * rhs
    call move_alloc(local_product,product)
  end function

  subroutine assign_lorenz(lhs,rhs) ! Assign one instance to another
    class(lorenz)           ,intent(inout) :: lhs
    class(integrable_model) ,intent(in)    :: rhs
    select type(rhs)
      class is (lorenz)
       !if (.not. allocated(lhs%state)) allocate(lhs%state(size(rhs%state)))
```

```
        lhs%state = rhs%state
        lhs%sigma = rhs%sigma
        lhs%rho   = rhs%rho
        lhs%beta  = rhs%beta
      class default
        stop 'assig_lorenz: unsupported class'
    end select
  end subroutine

  function output(this) result(coordinates)  ! Accessor: return state
    class(lorenz)      ,intent(in)  :: this
    real ,dimension(:) ,allocatable :: coordinates
   !allocate(coordinates(size(this%state)))
    coordinates = this%state
  end function output
end module lorenz_module
```

strategy.f03

```
module strategy_module
  use surrogate_module ,only : surrogate ! Substitute for integrable_model (avoiding circular references)

  implicit none ! Prevent implicit typing
  private       ! Hide everything by default

  type, abstract ,public :: strategy ! Abstract time integration strategy
  contains
    procedure(integrator_interface), nopass, deferred :: integrate ! Abstract integration procedure
  end type strategy

  abstract interface
    subroutine integrator_interface(this,dt)
      import :: surrogate
      class(surrogate) ,intent(inout) :: this ! integrand
      real             ,intent(in)    :: dt   ! time step size
    end subroutine
  end interface
end module
```

surrogate.f03

```
module surrogate_module
  implicit none ! Prevent implicit typing
  private       ! Hide everything by default (superfluous in this case)

  ! This stateless type serves only for purposes of extension by other types.
  ! In such a role, it can serve as a substitute for the child type when that
  ! type is inaccessible because of Fortran's prohibition against circular references.

  type ,abstract ,public :: surrogate
  end type
end module
```

timed_lorenz.f03

```
module timed_lorenz_module
  use lorenz_module ,only : lorenz,integrable_model ! Parent and grandparent types
  use strategy_module ,only : strategy              ! Time integration strategy

  implicit none              ! Prevent implicit typing
  private                    ! Hide everything by default
  public :: integrable_model ! Expose abstract integrand and integrator

  type ,extends(lorenz) ,public :: timed_lorenz
    private
    real :: time  ! time stamp
  contains
    procedure ,public :: construct                          ! constructor: allocate and initialize state
    procedure ,public :: d_dt      => dTimed_lorenz_dt      ! time derivative (expresses evolution equations)
    procedure ,public :: add       => add_timed_lorenz      ! add two instances
    procedure ,public :: multiply  => multiply_timed_lorenz ! multiply one instance by a real scalar
    procedure ,public :: assign    => assign_timed_lorenz   ! assign one instance to another
    procedure ,public :: output                             ! accessor: return state
  end type timed_lorenz

contains

  subroutine construct(this,initial_state,s,r,b,this_strategy) ! constructor: allocate and initialize state
    class(timed_lorenz) ,intent(out) :: this
    real ,dimension(:)  ,intent(in)  :: initial_state
    real                ,intent(in)  :: s ,r ,b       ! Lorenz parameters: sigma, rho and beta
    class(strategy)     ,intent(in)  :: this_strategy ! time integration algorithm
    call this%lorenz%construct(initial_state,s,r,b,this_strategy)
    this%time   = 0.
  end subroutine

  function dTimed_lorenz_dt(this) result(dState_dt) ! time derivative (expresses evolution equations)
    class(timed_lorenz)     ,intent(in)  :: this
    class(integrable_model) ,allocatable :: dState_dt
    type(timed_lorenz)      ,allocatable :: local_dState_dt ! obviates need for 'select type(dState_dt)'
    allocate(local_dState_dt)
    local_dState_dt%time   = 1.                 ! dt/dt = 1.
    local_dState_dt%lorenz = this%lorenz%d_dt() ! delegate to parent lorenz component
    call move_alloc(local_dState_dt,dState_dt)
  end function
```

```

  function add_timed_lorenz(lhs,rhs) result(sum) ! add two instances
    class(timed_lorenz)     ,intent(in)  :: lhs
    class(integrable_model) ,intent(in)  :: rhs
    class(integrable_model) ,allocatable :: sum
    type(timed_lorenz)      ,allocatable :: local_sum ! obviate need for 'select type(sum)'

    select type(rhs)
      class is (timed_lorenz)
        allocate(local_sum)
        local_sum%time   = lhs%time   + rhs%time
        local_sum%lorenz = lhs%lorenz + rhs%lorenz
      class default
        stop 'add_timed_lorenz: type not supported'
    end select
    call move_alloc(local_sum,sum)
  end function

  function multiply_timed_lorenz(lhs,rhs) result(product) ! multiply one instance by a real scalar
    class(timed_lorenz)     ,intent(in)  :: lhs
    real                    ,intent(in)  :: rhs
    class(integrable_model) ,allocatable :: product
    type(timed_lorenz)      ,allocatable :: local_product  ! obviate need for 'select type(product)'

    allocate(local_product)
    local_product%time   = lhs%time  * rhs
    local_product%lorenz = lhs%lorenz* rhs
    call move_alloc(local_product,product)
  end function

  subroutine assign_timed_lorenz(lhs,rhs) ! assign one instance to another
    class(timed_lorenz)     ,intent(inout) :: lhs
    class(integrable_model) ,intent(in)    :: rhs
    select type(rhs)
      class is (timed_lorenz)
        lhs%time   = rhs%time
        lhs%lorenz = rhs%lorenz
      class default
        stop 'assign_timed_lorenz: type not supported'
    end select
  end subroutine

  function output(this) result(coordinates)  ! return state
    class(timed_lorenz) ,intent(in)  :: this
    real ,dimension(:)  ,allocatable :: coordinates
    coordinates = [ this%time, this%lorenz%output() ]
  end function
end module timed_lorenz_module
```

explicit_euler.f03

```
module explicit_euler_module
  use surrogate_module ,only : surrogate               ! integrable_model parent
  use strategy_module ,only : strategy                 ! time integration strategy
  use integrable_model_module ,only : integrable_model ! abstract integrand

  implicit none ! Prevent implicit typing
  private       ! Hide everything by default

  type, extends(strategy) ,public :: explicit_euler ! 1st-order explicit time integrator
  contains
    procedure, nopass :: integrate
  end type

contains

  subroutine integrate(this,dt) ! Time integrator
    class(surrogate) ,intent(inout) :: this ! integrand
    real             ,intent(in)    :: dt   ! time step size
    select type (this)
      class is (integrable_model)
        this = this + this%d_dt()*dt ! Explicit Euler formula
      class default
        stop 'integrate: unsupported class.'
    end select
  end subroutine
end module
```

runge_kutta_2nd.f03

```
module runge_kutta_2nd_module
  use surrogate_module ,only : surrogate ! integrable_model parent
  use strategy_module ,only : strategy ! parent time integration strategy
  use integrable_model_module ,only : integrable_model ! abstract integrand

  implicit none ! Prevent implicit typing
  private       ! Hide everything by default

  type, extends(strategy) ,public :: runge_kutta_2nd ! 2nd-order Runge-Kutta time integration
  contains
    procedure, nopass :: integrate ! integration procedure
  end type

contains
```

```
15
16    subroutine integrate(this,dt)  ! Time integrator
17      class(surrogate)        ,intent(inout) :: this ! integrand
18      real                    ,intent(in)    :: dt   ! time step size
19      class(integrable_model) ,allocatable   :: this_half ! function evaluation at interval t+dt/2.
20      select type (this)
21        class is (integrable_model)
22          allocate(this_half,source=this)
23          this_half = this + this%d_dt()*(0.5*dt)  ! predictor step
24          this      = this + this_half%d_dt()*dt   ! corrector step
25        class default
26          stop 'integrate: unsupported class'
27      end select
28    end subroutine
29  end module
```

A.3 Puppeteer Example

main.f03

```
1   program main
2     use air_module  ,only : air
3     use cloud_module  ,only :cloud
4     use ground_module  ,only : ground
5     use atmosphere_module ,only : atmosphere,integrate
6     use global_parameters_module ,only : debugging ! print call tree if true
7
8     implicit none ! Prevent implicit typing
9
10    ! This code integrates the Lorenz equations over time using separate abstractions for
11    ! equation and hiding the coupling of those abstractions inside an abstraction that
12    ! follows the Puppeteer design pattern of Rouson, Adalsteinsson and Xia (ACM TOMS 2010).
13
14    type(air)        ,allocatable    :: sky             ! puppet for 1st Lorenz equation
15    type(cloud)      ,allocatable    :: puff            ! puppet for 2nd Lorenz equation
16    type(ground)     ,allocatable    :: earth           ! puppet for 3rd Lorenz equation
17    type(atmosphere)                 :: boundary_layer  ! Puppeteer
18    integer                          :: step            ! time step
19    integer ,parameter               :: num_steps=1000  ! total time steps
20    real    ,parameter               :: x=1.,y=1.,z=1.  ! initial conditions
21    real                             :: t               ! time coordinate
22    real    ,parameter               :: sigma=10.,rho=28.,beta=8./3.,dt=.02 ! Lorenz parameters
23
24    if (debugging) print *,'main: start'
25    allocate (sky, puff, earth)
26    call sky%construct(x,sigma)
27    call puff%construct(y,rho)
28    call earth%construct(z,beta)
29    call boundary_layer%construct(sky,puff,earth) ! transfer allocations into puppeteer
30    t = 0.                                        ! (all puppets are now deallocated)
31    write(*,'(f10.4)',advance='no') t; print *,boundary_layer%state_vector()
32    do step=1,num_steps
33      call integrate(boundary_layer,dt)
34      t = t + dt
35      write(*,'(f10.4)',advance='no') t; print *,boundary_layer%state_vector()
36    end do
37    if (debugging) print *,'main: end'
38  end program main
```

atmosphere.f03

```
1   module atmosphere_module
2     use air_module ,only : air       ! puppet for 1st Lorenz eq. and corresonding state variable
3     use cloud_module ,only : cloud   ! puppet for 2nd Lorenz eq. and corresonding state variable
4     use ground_module ,only : ground ! puppet for 3rd Lorenz eq. and corresonding state variable
5     use integrable_model_module ,only : integrable_model ,integrate ! parent type and polymorphic time integrator
6     use global_parameters_module ,only : debugging                   ! print call tree if true
7
8     implicit none        ! Prevent implicit typing
9     private              ! Hide everything by default
10    public :: integrate  ! Expose integration procedure from integrable_model_module
11
12    type ,extends(integrable_model) ,public :: atmosphere ! Puppeteer
13      private
14      type(air)    ,allocatable :: air_puppet
15      type(cloud)  ,allocatable :: cloud_puppet
16      type(ground) ,allocatable :: ground_puppet
17    contains
18      procedure ,public :: d_dt                => dAtmosphere_dt           ! time derivative
19      procedure ,public :: dRHS_dV             => dAtmosphereRHS_dState    ! Jacobian contribution (dR/dV)
20      procedure ,public :: state_vector        => atmosphere_state         ! return atmosphere solution vector
21      procedure ,public :: add                 => add_atmosphere           ! add two atmospheres
22      procedure ,public :: subtract            => subtract_atmospheres     ! subtract one atmosphere from another
23      procedure ,public :: multiply            => multiply_atmosphere      ! multiply an atmosphere by a real scalar
24      procedure ,public, pass(rhs) :: inverseTimes => inverseTimesAtmosphere ! abstract Gaussian elimination
25      procedure ,public :: assign              => assign_atmosphere        ! assign one atmosphere to another
26      procedure ,public :: empty_instance      => null_instance            ! create empty atmosphere
27      procedure ,public :: construct                                       ! constructor
28    end type atmosphere
29
30  contains
31
```

```
32  function null_instance(this) result(blank_slate) ! create empty atmosphere
33    class(atmosphere)       ,intent(in)  :: this
34    class(integrable_model) ,allocatable :: blank_slate
35    allocate(atmosphere :: blank_slate)
36  end function
37
38  subroutine construct(this,air_target,cloud_target,ground_target) ! constructor
39    class(atmosphere)         ,intent(out)   :: this
40    type(air)    ,allocatable ,intent(inout) :: air_target
41    type(cloud)  ,allocatable ,intent(inout) :: cloud_target
42    type(ground) ,allocatable ,intent(inout) :: ground_target
43    if (debugging) print *,'    atmosphere%construct(): start'
44    call move_alloc(air_target, this%air_puppet)       ! transfer allocations from puppets to Puppeteer
45    call move_alloc(ground_target, this%ground_puppet)
46    call move_alloc(cloud_target, this%cloud_puppet)
47    if (debugging) print *,'    atmosphere%construct(): end'
48  end subroutine
49
50  function dAtmosphere_dt(this) result(dState_dt) ! time derivative (evolution equations)
51    class(atmosphere)       ,intent(in)  :: this
52    class(integrable_model) ,allocatable :: dState_dt
53    type(atmosphere)        ,allocatable :: delta ! obviates the use of 'select type(this)'
54
55    if (debugging) print *,'    atmosphere%dAtmosphere_dt(): start'
56    allocate(delta)
57    delta%air_puppet    = this%air_puppet%d_dt(   this%cloud_puppet%coordinate())
58    delta%cloud_puppet  = this%cloud_puppet%d_dt( this%air_puppet%coordinate(),this%ground_puppet%coordinate())
59    delta%ground_puppet = this%ground_puppet%d_dt(this%air_puppet%coordinate(),this%cloud_puppet%coordinate())
60    call move_alloc(delta, dState_dt)
61    if (debugging) print *,'    atmosphere%dAtmosphere_dt(): end'
62  end function
63
64  function dAtmosphereRHS_dState(this) result(dRHS_dState) ! atmosphere contribution to Jacobian
65    class(atmosphere)    ,intent(in)  :: this
66    real ,dimension(:,:) ,allocatable ::    dAir_dAir ,   dAir_dCloud ,   dAir_dGround ! Sub-blocks
67    real ,dimension(:,:) ,allocatable ::  dCloud_dAir , dCloud_dCloud , dCloud_dGround ! of dR/dV
68    real ,dimension(:,:) ,allocatable :: dGround_dAir ,dGround_dCloud ,dGround_dGround ! array.
69    real ,dimension(:,:) ,allocatable :: dRHS_dState
70    real ,dimension(:)   ,allocatable :: air_coordinate, cloud_coordinate, ground_coordinate
71    integer                           :: air_eqs,air_vars,cloud_eqs,cloud_vars,ground_eqs,ground_vars,i,j,rows,cols
72    if (debugging) print *,'    atmosphere%dAtmosphereRHS_dState(): start'
73    select type(this)
74       type is (atmosphere) ! Calculate matrices holding partial derivative of puppet evolution equation right-hand
75                            ! sides with respect to the dependent variables of each puppet.
76         air_coordinate  = this%air_puppet%coordinate()
77         cloud_coordinate= this%cloud_puppet%coordinate()
78         ground_coordinate=this%ground_puppet%coordinate()
79
80         dAir_dAir       = this%air_puppet%d_dAir(cloud_coordinate)                    ! Diagonal block submatrix
81         dCloud_dCloud   = this%cloud_puppet%d_dCloud(air_coordinate,ground_coordinate)  ! Diagonal block submatrix
82         dGround_dGround = this%ground_puppet%d_dGround(air_coordinate,cloud_coordinate) ! Diagonal block submatrix
83         air_eqs     = size(dAir_dAir,1)       ! submatrix rows
84         air_vars    = size(dAir_dAir,2)       ! submatrix columns
85         cloud_eqs   = size(dCloud_dCloud,1)    ! submatrix rows
86         cloud_vars  = size(dCloud_dCloud,2)    ! submatrix columns
87         ground_eqs  = size(dGround_dGround,1)  ! submatrix rows
88         ground_vars = size(dGround_dGround,2)  ! submatrix columns
89         dAir_dCloud    = this%air_puppet%d_dy(cloud_coordinate)                                          ! Off-diagonal
90         dAir_dGround   = reshape(source=(/(0.,i=1,air_eqs*ground_vars)/),shape=(/air_eqs,ground_vars/)) ! Off-diagonal
91         dCloud_dAir    = this%cloud_puppet%d_dx(air_coordinate,ground_coordinate)                       ! Off-diagonal
92         dCloud_dGround = this%cloud_puppet%d_dz(air_coordinate,ground_coordinate)                       ! Off-diagonal
93         dGround_dAir   = this%ground_puppet%d_dx(air_coordinate,cloud_coordinate)                       ! Off-diagonal
94         dGround_dCloud = this%ground_puppet%d_dy(air_coordinate,cloud_coordinate)                       ! Off-diagonal
95
96         rows=air_eqs+cloud_eqs+ground_eqs
97         cols=air_vars+cloud_vars+ground_vars
98         allocate(dRHS_dState(rows,cols))
99         dRHS_dState(1:air_eqs, 1:air_vars)                   = dAir_dAir                     ! Begin result assembly
100        dRHS_dState(1:air_eqs, air_vars+1:air_vars+cloud_vars) = dAir_dCloud
101        dRHS_dState(1:air_eqs, air_vars+cloud_vars+1:cols)     = dAir_dGround
102
103        dRHS_dState(air_eqs+1:air_eqs+cloud_eqs, 1:air_vars)                   = dCloud_dAir
104        dRHS_dState(air_eqs+1:air_eqs+cloud_eqs, air_vars+1:air_vars+cloud_vars) = dCloud_dCloud
105        dRHS_dState(air_eqs+1:air_eqs+cloud_eqs, air_vars+cloud_vars+1:cols)     = dCloud_dGround
106
107        dRHS_dState(air_eqs+cloud_eqs+1:rows, 1:air_vars)                   = dGround_dAir
108        dRHS_dState(air_eqs+cloud_eqs+1:rows, air_vars+1:air_vars+cloud_vars) = dGround_dCloud
109        dRHS_dState(air_eqs+cloud_eqs+1:rows, air_vars+cloud_vars+1:cols)     = dGround_dGround ! Finish result assembly
110     end select
111   if (debugging) print *,'    atmosphere%dAtmosphereRHS_dState(): end'
112 end function dAtmosphereRHS_dState
113
114 function atmosphere_state(this) result(phase_space) ! assemble and return solution vector
115   class(atmosphere)  ,intent(in)  :: this
116   real ,dimension(:) ,allocatable :: state
117   real ,dimension(:) ,allocatable :: x,y,z,phase_space
118   integer                         :: x_start,y_start,z_start
119   integer                         :: x_end  ,y_end  ,z_end
120  !if (debugging) print *,'    atmosphere%atmosphere_state(): start' (commented to avoid I/O recursion)
121   x = this%air_puppet%coordinate()   ; x_start=1      ; x_end=x_start+size(x)-1
122   y = this%cloud_puppet%coordinate() ; y_start=x_end+1; y_end=y_start+size(y)-1
123   z = this%ground_puppet%coordinate(); z_start=y_end+1; z_end=z_start+size(z)-1
124   allocate(phase_space(size(x)+size(y)+size(z)))
```

```
125     phase_space(x_start:x_end) = x
126     phase_space(y_start:y_end) = y
127     phase_space(z_start:z_end) = z
128    !if (debugging) print *,'    atmosphere%atmosphere_state(): end' (commented to avoid I/O recursion)
129   end function
130
131   function add_atmosphere(lhs,rhs) result(sum)
132     class(atmosphere)       ,intent(in)  :: lhs
133     class(integrable_model) ,intent(in)  :: rhs
134     class(integrable_model) ,allocatable :: sum
135     type (atmosphere), allocatable :: local_sum ! used to avoid 'select type(sum)'
136
137     if (debugging) print *,'    atmosphere%add_atmosphere(): start'
138     allocate (local_sum)
139     select type(rhs)
140       type is (atmosphere)
141        !allocate(local_sum%air_puppet,local_sum%ground_puppet,local_sum%cloud_puppet)
142         local_sum%air_puppet    = lhs%air_puppet    + rhs%air_puppet
143         local_sum%cloud_puppet  = lhs%cloud_puppet  + rhs%cloud_puppet
144         local_sum%ground_puppet = lhs%ground_puppet + rhs%ground_puppet
145       class default
146         stop 'add_atmosphere: rhs argument type not supported'
147     end select
148     call move_alloc(local_sum, sum)
149     if (debugging) print *,'    atmosphere%add_atmosphere(): end'
150   end function
151
152   function subtract_atmospheres(lhs,rhs) result(difference)
153     class(atmosphere)       ,intent(in)  :: lhs
154     class(integrable_model) ,intent(in)  :: rhs
155     class(integrable_model) ,allocatable :: difference
156     type(atmosphere)        ,allocatable :: local_difference
157
158     if (debugging) print *,'    atmosphere%subtract_atmospheres(): start'
159     allocate(local_difference)
160     select type(rhs)
161       type is (atmosphere)
162        !allocate(local_difference%air_puppet,local_difference%ground_puppet,local_difference%cloud_puppet)
163         local_difference%air_puppet    = lhs%air_puppet    - rhs%air_puppet
164         local_difference%cloud_puppet  = lhs%cloud_puppet  - rhs%cloud_puppet
165         local_difference%ground_puppet = lhs%ground_puppet - rhs%ground_puppet
166       class default
167         stop 'add_atmosphere: rhs argument type not supported'
168     end select
169     call move_alloc(local_difference, difference)
170     if (debugging) print *,'    atmosphere%subtract_atmospheres(): end'
171   end function
172
173   function inverseTimesAtmosphere(lhs,rhs) result(product)  ! Solve linear system Ax=b by Gaussian elimination
174     class(atmosphere)                 ,intent(in) :: rhs
175     class(integrable_model) ,allocatable          :: product
176     real ,dimension(:,:) ,allocatable ,intent(in) :: lhs
177     type(atmosphere)     ,allocatable             :: local_product
178     real ,dimension(:)   ,allocatable             :: x,b
179     real ,dimension(:,:) ,allocatable             :: A
180     real                                          :: factor
181     integer                                       :: row,col,n,p ! p=pivot row/col
182     real ,parameter                               :: pivot_tolerance=1.0E-02
183
184     n=size(lhs,1)
185     b = rhs%state_vector()
186     if (n /= size(lhs,2) .or. n /= size(b)) stop 'integrable_model.f03: ill-posed matrix problem in inverseTimes()'
187    !allocate(A(n,n),b(n))
188     allocate(x(n))
189     A = lhs
190     do p=1,n-1         ! Forward elimination
191       if (abs(A(p,p))<pivot_tolerance) stop 'invert: use an algorithm with pivoting'
192       do row=p+1,n
193         factor=A(row,p)/A(p,p)
194         forall(col=p:n)
195           A(row,col) = A(row,col) - A(p,col)*factor
196         end forall
197         b(row) = b(row) - b(p)*factor
198       end do
199     end do
200     x(n) = b(n)/A(n,n) ! Back substitution
201     do row=n-1,1,-1
202       x(row) = (b(row) - sum(A(row,row+1:n)*x(row+1:n)))/A(row,row)
203     end do
204     allocate(local_product,source=rhs)
205     call local_product%air_puppet%construct(x(1),x(2))
206     call local_product%cloud_puppet%construct(x(3))
207     call local_product%ground_puppet%construct(x(4))
208     call move_alloc(local_product, product)
209   end function
210
211   function multiply_Atmosphere(lhs,rhs) result(product) ! multiply atmosphere object by a real scalar
212     class(atmosphere)       ,intent(in)  :: lhs
213     real                    ,intent(in)  :: rhs
214     class(integrable_model) ,allocatable :: product
215     type(atmosphere)        ,allocatable :: local_product ! used to avoid 'select type(product)'
216
217     if (debugging) print *,'    atmosphere%multiply_Atmosphere(): start'
```

```
    allocate(local_product)
   !allocate(local_product%air_puppet,local_product%ground_puppet,local_product%cloud_puppet)
    local_product%air_puppet    = lhs%air_puppet    * rhs
    local_product%cloud_puppet  = lhs%cloud_puppet  * rhs
    local_product%ground_puppet = lhs%ground_puppet * rhs
    call move_alloc(local_product, product)
    if (debugging) print *,'    atmosphere%multiply_Atmosphere(): end'
  end function

  subroutine assign_atmosphere(lhs,rhs) ! assign one atmosphere object to another
    class(atmosphere)       ,intent(inout) :: lhs
    class(integrable_model) ,intent(in)    :: rhs
    if (debugging) print *,'    atmosphere%assign_atmosphere(): start'
    select type(rhs)
     type is (atmosphere)
      !allocate(lhs%air_puppet,lhs%cloud_puppet,lhs%ground_puppet)
       lhs%air_puppet    = rhs%air_puppet
       lhs%cloud_puppet  = rhs%cloud_puppet
       lhs%ground_puppet = rhs%ground_puppet
     class default
       stop 'assign_atmosphere: rhs argument type not supported'
    end select
    if (debugging) print *,'    atmosphere%assign_atmosphere(): end'
  end subroutine
end module atmosphere_module
```

air.f03

```
module air_module
  use global_parameters_module ,only : debugging ! print call tree information if true

  implicit none ! Prevent implicit typing
  private       ! Hide everything by default

  ! Number of evolution equations/variables exposed to the outside world (via Jacobian sub-block shapes):
  integer ,parameter :: num_eqs=2,num_vars=2

  ! This type tracks the evolution of the first state variable in the Lorenz system
  ! according to the first Lorenz equation.  It also tracks the corresponding paramater (sigma)
  ! according to the differential equation d(sigma)/dt=0. For illustrative purposes,
  ! this implementation exposes the number of state variables (2) to the puppeteer without
  ! providing direct access to them or exposing anything about their layout, storage location
  ! or identifiers (x and sigma).  Their existence is apparent in the rank (2) of the matrix
  ! d_dAir() returns as its diagonal Jacobian element contribution.

  type ,public :: air
    private
    real :: x,sigma ! 1st Lorenz equation solution variable and parameter
  contains
    procedure ,public  :: construct => construct_air  ! constructor: allocate and initialize components
    procedure ,public  :: coordinate=> coordinate_air ! accessor (returns phase-space coordinate)
    procedure ,public  :: d_dt                        ! time derivative (evolution equations)
    procedure ,public  :: d_dAir                      ! contribution to diagonal Jacobian element
    procedure ,public  :: d_dy                        ! contribution to off-diagonal Jacobian element
    procedure ,private :: add_air                     ! add two instances
    procedure ,private :: subtract_air                ! subtract one instance from another
    procedure ,private :: multiply_air                ! multiply an instance by a real scalar
    generic   ,public  :: operator(+)   => add_air    ! map defined operators to corresponding procedures
    generic   ,public  :: operator(-)   => subtract_air
    generic   ,public  :: operator(*)   => multiply_air
  end type air

contains

  subroutine construct_air(this,x_initial,s) ! constructor: allocate and initialize components
    class(air)     ,intent(out) :: this
    real           ,intent(in)  :: x_initial
    real           ,intent(in)  :: s

    if (debugging) print *,'      air%construct_air: start'
    this%x = x_initial
    this%sigma  = s
    if (debugging) print *,'      air%construct_air: end'
  end subroutine

  function coordinate_air(this) result(return_x)  ! accessor (returns phase-space coordinate)
    class(air)          ,intent(in)  :: this
    real  ,dimension(:) ,allocatable :: return_x

   !if (debugging) print *,'      air%coordinate_air: start' (commented to avoid I/O recursion )
   !allocate(return_x(num_vars))
    return_x = [ this%x ,this%sigma ]
   !if (debugging) print *,'      air%coordinate_air: end' (commented to avoid I/O recursion)
  end function

  function d_dt(this,y) result(dx_dt) ! time derivative (evolution equations)
    class(air)         ,intent(in) :: this
    real ,dimension(:) ,intent(in) :: y
    type(air)                      :: dx_dt

    if (debugging) print *,'      air%d_dt: start'
    dx_dt%x=this%sigma*(y(1)-this%x)
    dx_dt%sigma=0.
    if (debugging) print *,'      air%d_dt: end'
```

```
  end function

  function d_dAir(this,y) result(dRHS_dx)  ! contribution to diagonal Jacobian element
    class(air)  ,intent(in)              :: this
    real        ,dimension(:,:) ,allocatable :: dRHS_dx
    real        ,dimension(:) ,allocatable :: y

    if (debugging) print *,'      air%d_dAir: start'
   !allocate(dRHS_dx(num_eqs,num_vars))
   !dRHS_dx = [ d{sigma*(y-x)}/dx   d{sigma*(y-x)}/dsigma ]
   !         [ d{0}/dx             d{0}/dsigma           ]
    if (size(y) /= 1) stop 'd_dAir: invalid y size'
    dRHS_dx = reshape(source=(/-this%sigma,0.,y(1)-this%x,0./),shape=(/num_eqs,num_vars/))
    if (debugging) print *,'      air%d_dAir: end'
  end function

  function d_dy(this,y) result(dRHS_dy) ! contribution to off-diagonal Jacobian element
    class(air)  ,intent(in)              :: this
    real        ,dimension(:,:) ,allocatable :: dRHS_dy
    real        ,dimension(:)   ,allocatable :: y

    if (debugging) print *,'      air%d_dy: start'
    allocate(dRHS_dy(num_eqs,size(y)))
   !dRHS_dy = [ d{sigma*(y(1)-x(1))}/dy(1)  0  ... 0]
   !         [ d{0}/dy(1)                  0  ... 0]
    dRHS_dy = 0.
    dRHS_dy(1,1) = this%sigma
    if (debugging) print *,'      air%d_dy: end'
  end function

  function add_air(lhs,rhs) result(sum) ! add two instances
    class(air) ,intent(in) :: lhs,rhs
    type(air)              :: sum

    if (debugging) print *,'      air%add_air: start'
    sum%x     = lhs%x     + rhs%x
    sum%sigma = lhs%sigma + rhs%sigma
    if (debugging) print *,'      air%add_air: end'
  end function

  function subtract_air(lhs,rhs) result(difference) ! subtract one instance from another
    class(air) ,intent(in) :: lhs,rhs
    type(air)              :: difference

    if (debugging) print *,'      air%subtract_air: start'
    difference%x     = lhs%x     - rhs%x
    difference%sigma = lhs%sigma - rhs%sigma
    if (debugging) print *,'      air%subtract_air: end'
  end function

  function multiply_air(lhs,rhs) result(product) ! multiply an instance by a real scalar
    class(air) ,intent(in) :: lhs
    real       ,intent(in) :: rhs
    type(air)              :: product

    if (debugging) print *,'      air%multiply_air: start'
    product%x     = lhs%x    *rhs
    product%sigma = lhs%sigma*rhs
    if (debugging) print *,'      air%multiply_air: end'
  end function
end module air_module
```

cloud.f03

```
module cloud_module
  use global_parameters_module ,only : debugging ! print call tree if .true.

  implicit none  ! Prevent implicit typing
  private        ! Hide everything by default

  ! This type tracks the evolution of the second state variable in the Lorenz system
  ! according to the second Lorenz equation.  It also tracks the corresponding paramater (rho)
  ! according to the differential equation d(rho)/dt=0. For illustrative purposes,
  ! this implementation does not expose the number of state variables (2) to the puppeteer
  ! because no iteration is required and the need for arithmetic operations on rho is therefore
  ! an internal concern.  The rank of the matrix d_dCloud() returns is thus 1 to reflect the
  ! only variable on which the puppeteer needs to iterate when handling nonlinear couplings in
  ! implicit solvers.

  ! Number of evolution equations/variables exposed to the outside world (via Jacobian sub-block shapes):
  integer ,parameter :: num_eqs=1,num_vars=1

  type ,public :: cloud
    private
    real :: y,rho ! 2nd Lorenz equation solution variable and parameter
  contains
    procedure ,public  :: construct => construct_cloud  ! constructor: allocate and initialize components
    procedure ,public  :: coordinate=> coordinate_cloud ! accessor (returns phase-space coordinate)
    procedure ,public  :: d_dt                          ! time derivative (evolution equations)
    procedure ,public  :: d_dCloud                      ! contribution to diagonal Jacobian element
    procedure ,public  :: d_dx                          ! contribution to off-diagonal Jacobian element
    procedure ,public  :: d_dz                          ! contribution to off-diagonal Jacobian element
    procedure ,private :: add_cloud                     ! add two instances
    procedure ,private :: subtract_cloud                ! subtract one instance from another
```

```
31      procedure ,private :: multiply_cloud               ! multiply an instance by a real scalar
32      generic   ,public  :: operator(+) => add_cloud     ! map defined operators to corresponding procedures
33      generic   ,public  :: operator(-) => subtract_cloud
34      generic   ,public  :: operator(*) => multiply_cloud
35    end type cloud
36
37  contains
38
39    subroutine construct_cloud(this,y_initial,r) ! constructor: allocate and initialize components
40      class(cloud)   ,intent(out) :: this
41      real           ,intent(in)  :: y_initial
42      real ,optional ,intent(in)  :: r
43      if (debugging) print *,'      cloud%construct_cloud: start'
44      this%y = y_initial
45      if (present(r)) this%rho = r
46      if (debugging) print *,'      cloud%construct_cloud: end'
47    end subroutine
48
49    function coordinate_cloud(this) result(return_y)  ! accessor (returns phase-space coordinate)
50      class(cloud)       ,intent(in)  :: this
51      real ,dimension(:) ,allocatable :: return_y
52     !if (debugging) print *,'      cloud%coordinate_cloud: start' (commented to prevent I/O recursion)
53     !allocate(return_y(num_vars))
54      return_y = [this%y]
55     !if (debugging) print *,'      cloud%coordinate_cloud: end' (commented to prevent I/O recursion)
56    end function
57
58    function d_dCloud(this,x_ignored,z_ignored) result(dRHS_dy)  ! contribution to diagonal Jacobian element
59      class(cloud)                       ,intent(in) :: this
60      real ,dimension(:)   ,allocatable ,intent(in) :: x_ignored,z_ignored
61      real ,dimension(:,:) ,allocatable             :: dRHS_dy
62      if (debugging) print *,'      cloud%d_dCloud: start'
63      allocate(dRHS_dy(num_eqs,num_vars))
64     !dRHS_dy(1) = [ d{x(1)*(rho-z(1))-y(1)}/dy(1) ]
65      dRHS_dy(1,1) = -1.
66      if (debugging) print *,'      cloud%d_dCloud: end'
67    end function
68
69    function d_dx(this,x,z) result(dRHS_dx)  ! contribution to off-diagonal Jacobian element
70      class(cloud)                       ,intent(in) :: this
71      real ,dimension(:)   ,allocatable ,intent(in) :: x,z
72      real ,dimension(:,:) ,allocatable             :: dRHS_dx
73      if (debugging) print *,'      cloud%d_dx: start'
74      allocate(dRHS_dx(num_eqs,size(x)))
75     !dRHS_dx = [ d{x(1)*(rho-z(1))-y}/dx(1)  0  ... 0]
76      dRHS_dx = 0.
77      dRHS_dx(1,1) = this%rho-z(1)
78      if (debugging) print *,'      cloud%d_dx: end'
79    end function
80
81    function d_dz(this,x,z) result(dRHS_dz)  ! contribution to off-diagonal Jacobian element
82      class(cloud)                       ,intent(in) :: this
83      real ,dimension(:)   ,allocatable ,intent(in) :: x,z
84      real ,dimension(:,:) ,allocatable             :: dRHS_dz
85      if (debugging) print *,'      cloud%d_dz: start'
86      allocate(dRHS_dz(num_eqs,size(z)))
87     !dRHS_dz = [ d{x(1)*(rho-z(1))-y(1)}/dz(1)  0  ... 0]
88      dRHS_dz = 0.
89      dRHS_dz(1,1) = -x(1)
90      if (debugging) print *,'      cloud%d_dz: end'
91    end function
92
93    function d_dt(this,x,z) result(dy_dt) ! time derivative (evolution equations)
94      class(cloud)       ,intent(in) :: this
95      real ,dimension(:) ,intent(in) :: x,z
96      type(cloud)                    :: dy_dt
97      if (debugging) print *,'      cloud%d_dt_cloud: start'
98      dy_dt%y = x(1)*(this%rho-z(1))-this%y
99      dy_dt%rho = 0.
100     if (debugging) print *,'      cloud%d_dt_cloud: end'
101   end function
102
103   function add_cloud(lhs,rhs) result(sum) ! add two instances
104     class(cloud) ,intent(in) :: lhs,rhs
105     type(cloud)              :: sum
106     if (debugging) print *,'      cloud%add_cloud: start'
107     sum%y   = lhs%y   + rhs%y
108     sum%rho = lhs%rho + rhs%rho
109     if (debugging) print *,'      cloud%add_cloud: end'
110   end function
111
112   function subtract_cloud(lhs,rhs) result(difference) ! subtract one instance from another
113     class(cloud) ,intent(in) :: lhs,rhs
114     type(cloud)              :: difference
115     if (debugging) print *,'      cloud%subtract_cloud: start'
116     difference%y   = lhs%y   - rhs%y
117     difference%rho = lhs%rho - rhs%rho
118     if (debugging) print *,'      cloud%subtract_cloud: end'
119   end function
120
121   function multiply_cloud(lhs,rhs) result(product) ! multiply an instance by a real scalar
122     class(cloud) ,intent(in) :: lhs
123     real         ,intent(in) :: rhs
```

```
    type(cloud)              :: product
    if (debugging) print *,'      cloud%multiply_cloud: start'
    product%y   = lhs%y* rhs
    product%rho = lhs%rho* rhs
    if (debugging) print *,'      cloud%multiply_cloud: end'
  end function
end module cloud_module
```

ground.f03

```
module ground_module
  use global_parameters_module ,only : debugging ! print call tree if true

  implicit none ! Prevent implicit typing
  private       ! Hide everything by default

  ! This type tracks the evolution of the third state variable in the Lorenz system
  ! according to the third Lorenz equation.  It also tracks the corresponding paramater (beta)
  ! according to the differential equation d(beta)/dt=0. For illustrative purposes,
  ! this implementation does not expose the number of state variables (2) to the puppeteer
  ! because no iteration is required and the need for arithmetic operations on beta is therefore
  ! an internal concern.  The rank of the matrix d_dGround() returned is thus 1 to reflect the
  ! only variable on which the puppeteer needs to iterate when handling nonlinear couplings in
  ! implicit solvers.

  ! Number of evolution equations/variables exposed to the outside world (via Jacobian sub-block shapes):
  integer ,parameter :: num_eqs=1,num_vars=1

  type ,public :: ground
    private
    real :: z,beta ! 3rd Lorenz equation solution variable and parameter
  contains
    procedure ,public  :: construct => construct_ground  ! constructor: allocate and initialize components
    procedure ,public  :: coordinate=> coordinate_ground ! accessor (returns phase-space coordinate)
    procedure ,public  :: d_dt                           ! time derivative (evolution equations)
    procedure ,public  :: d_dGround                      ! contribution to diagonal Jacobian element
    procedure ,public  :: d_dx                           ! contribution to off-diagonal Jacobian element
    procedure ,public  :: d_dy                           ! contribution to off-diagonal Jacobian element
    procedure ,private :: add_ground                     ! add two instances
    procedure ,private :: subtract_ground                ! subtract one instance from another
    procedure ,private :: multiply_ground                ! multiply an instance by a real scalar
    generic   ,public  :: operator(+) => add_ground      ! map defined operators to corresponding procedures
    generic   ,public  :: operator(-) => subtract_ground
    generic   ,public  :: operator(*) => multiply_ground
  end type ground

contains

  subroutine construct_ground(this,z_initial,b) ! constructor: allocate and initialize components
    class(ground)  ,intent(out) :: this
    real           ,intent(in)  :: z_initial
    real ,optional ,intent(in)  :: b
    if (debugging) print *,'      ground%construct_ground: start'
    this%z = z_initial
    if (present(b)) this%beta = b
    if (debugging) print *,'      ground%construct_ground: end'
  end subroutine

  function coordinate_ground(this) result(return_z)  ! accessor (returns phase-space coordinate)
    class(ground)       ,intent(in)  :: this
    real ,dimension(:) ,allocatable :: return_z
   !if (debugging) print *,'      ground%coordinate_ground: start' (can cause I/O recursion error)
   !allocate(return_z(num_vars))
    return_z = [ this%z ]
   !if (debugging) print *,'      ground%coordinate_ground: end' (can cause I/O recursion error)
  end function

  function d_dt(this,x,y) result(dz_dt) ! time derivative (evolution equations)
    class(ground)      ,intent(in) :: this
    real ,dimension(:) ,intent(in) :: x,y
    type(ground)                   :: dz_dt
    if (debugging) print *,'      ground%d_dt: start'
    dz_dt%z = x(1)*y(1) - this%beta*this%z
    dz_dt%beta = 0.
    if (debugging) print *,'      ground%d_dt: end'
  end function

  function d_dGround(this,x_ignored,y_ignored) result(dRHS_dz)  ! contribution to diagonal Jacobian element
    class(ground)                    ,intent(in) :: this
    real ,dimension(:)   ,allocatable ,intent(in) :: x_ignored,y_ignored
    real ,dimension(:,:) ,allocatable             :: dRHS_dz
    if (debugging) print *,'      ground%d_dGround: start'
   !dRHS_dz = [ d{x(1)*y(1) - beta*z}/dz(1) ]
    allocate(dRHS_dz(num_eqs,num_vars))
    dRHS_dz(1,1) = -this%beta
    if (debugging) print *,'      ground%d_dGround: end'
  end function

  function d_dx(this,x,y) result(dRHS_dx)  ! contribution to off-diagonal Jacobian element
    class(ground)                    ,intent(in) :: this
    real ,dimension(:)   ,allocatable ,intent(in) :: x,y
    real ,dimension(:,:) ,allocatable             :: dRHS_dx
    if (debugging) print *,'      ground%d_dx: start'
```

```
    allocate(dRHS_dx(num_eqs,size(x)))
   !dRHS_dz = [ d{x(1)*y(1) - beta*z(1)}/dx(1)   0  ...  0 ]
    dRHS_dx=0.
    dRHS_dx(1,1) = y(1)
    if (debugging) print *,'      ground%d_dx: end'
  end function

  function d_dy(this,x,y) result(dRHS_dy)  ! contribution to off-diagonal Jacobian element
    class(ground)                    ,intent(in) :: this
    real ,dimension(:)   ,allocatable ,intent(in) :: x,y
    real ,dimension(:,:) ,allocatable             :: dRHS_dy
    if (debugging) print *,'      ground%d_dy: start'
    allocate(dRHS_dy(num_eqs,size(y)))
   !dRHS_dz = [ d{x(1)*y(1) - beta*z(1)}/dy(1)   0  ...  0 ]
    dRHS_dy = 0.
    dRHS_dy(1,1) = x(1)
    if (debugging) print *,'      ground%d_dy: end'
  end function

  function add_ground(lhs,rhs) result(sum) ! add two instances
    class(ground) ,intent(in) :: lhs,rhs
    type(ground)              :: sum
    if (debugging) print *,'      ground%add_ground: start'
    sum%z    = lhs%z    + rhs%z
    sum%beta = lhs%beta + rhs%beta
    if (debugging) print *,'      ground%add_ground: end'
  end function

  function subtract_ground(lhs,rhs) result(difference) ! subtract one instance from another
    class(ground) ,intent(in) :: lhs,rhs
    type(ground)              :: difference
    if (debugging) print *,'      ground%subtract_ground: start'
    difference%z    = lhs%z    - rhs%z
    difference%beta = lhs%beta - rhs%beta
    if (debugging) print *,'      ground%subtract_ground: end'
  end function

  function multiply_ground(lhs,rhs) result(product) ! multiply an instance by a real scalar
    class(ground) ,intent(in) :: lhs
    real          ,intent(in)  :: rhs
    type(ground)              :: product
    if (debugging) print *,'      ground%multiply_ground: start'
    product%z    = lhs%z   * rhs
    product%beta = lhs%beta* rhs
    if (debugging) print *,'      ground%multiply_ground: end'
  end function
end module ground_module
```

APPENDIX B. C++ IMPLEMENATION

Ref.h

```
#ifndef REF_H_
#define REF_H_

/**
 * A very simple invasive reference counted pointer.
 * In future C++, this should be replaced with shared-ptr
 */

#include <iostream>
#include <unistd.h>

template <typename T>
class Ref {
public:
  typedef Ref<T> Self_t;

  Ref() : ptr_(NULL) {}
  Ref(const Self_t &other) : ptr_(other.ptr_) {
    if(ptr_) ptr_->grab();
  }
  template <typename Other>
  Ref(Ref<Other> other) : ptr_(other.ptr()) {
    if(ptr_) ptr_->grab();
  }
  template <typename Other>
  Ref(Other *other) : ptr_(other) {
    if(ptr_) ptr_->grab();
  }
  ~Ref() {
    if(ptr_) ptr_->release();
  }
  template <typename Other>
  Self_t& operator=(const Ref<Other> &other) {
    if(other.ptr() != ptr_) {
      if(ptr_) ptr_->release();
      ptr_ = other.ptr();
      if(ptr_) ptr_->grab();
    }
  }
  template <typename Other>
```

```
  Self_t& operator=(Other *other) {
    if(ptr_ != other) {
      if(ptr_) ptr_->release();
      ptr_ = other;
      if(ptr_) ptr_->grab();
    }
  }
  T* operator->() {
    return ptr_;
  }
  const T* operator->() const {
    return ptr_;
  }
  T& operator*() {
    return *ptr_;
  }
  const T& operator*() const {
    return ptr_;
  }
  template <typename Other>
  bool operator==(const Ref<Other> &other) const {
    return ptr_ == other.ptr_;
  }
  bool operator==(const T *other) const {
    return ptr_ == other;
  }
  template <typename Other>
  bool operator==(const Other *other) const {
    return ptr_ == other;
  }
  template <typename Other>
  bool operator<(const Ref<Other> &other) const {
    return ptr_ < other.ptr_;
  }
  template <typename Other>
  bool operator<(const Other *other) const {
    return ptr_ < other;
  }
  T* ptr() const { return ptr_; }

private:
  T *ptr_;
};

template <typename Target, typename Source>
Ref<Target> cast(const Ref<Source> &pp) {
  return Ref<Target>(dynamic_cast<Target*>(pp.ptr()));
}

#endif /* ! REF_H_ */
```

RefBase.h

```
#ifndef REFBASE_H_
#define REFBASE_H_

/**
 * Base for reference counted objects.
 */

#include "Ref.h"

class RefBase {
public:
  RefBase() : cnt_(0) {}
  RefBase(const RefBase &) : cnt_(0) {}
  virtual ~RefBase() {}
  void grab() throw() {
    cnt_++;
  }
  void release() throw() {
    cnt_--;
    if(! cnt_) delete this;
  }
private:
  int cnt_;
};

#endif /* ! REFBASE_H_ */
```

fmt.h

```
#ifndef _H_FMT_
#define _H_FMT_

#include "globals.h"
#include <iostream>
#include <iomanip>

// The fmt(...) helper class helps hide the mess that is <iomanip>
struct fmt {
  explicit fmt(real_t value, int width=12, int prec=8) :
    v_(1, value), w_(width), p_(prec)
  {}
```

```
  explicit fmt(crd_t value, int width=12, int prec=8) :
    v_(value), w_(width), p_(prec)
  {}

  const crd_t v_;
  const int w_, p_;
};

inline std::ostream& operator<<(std::ostream &os, const fmt &v) {
  // Store format flags for the stream.
  std::ios_base::fmtflags flags = os.flags();
  // Force our own weird format.
  for(crd_t::const_iterator it = v.v_.begin(); it != v.v_.end(); ++it) {
    os << " " <<std::setw(v.w_) <<std::setprecision(v.p_) <<std::fixed << *it;
  }
  // Restore original format flags.
  os.flags(flags);
  return os;
}

#endif //! _H_FMT_
```

globals.h

```
#ifndef _H_GLOBALS_
#define _H_GLOBALS_

#include "Ref.h"
#include "RefBase.h"
#include <vector>

typedef float real_t;
typedef std::vector<real_t> crd_t;

#endif //!_H_GLOBALS
```

B.1 Semidiscrete Example

main.cpp

```
#include "lorenz.h"
#include "fmt.h"
#include <iostream>

int main () {
  using namespace std;
  typedef lorenz::ptr_t ptr_t;

  const int   num_steps=2000, space_dimension=3;
  const float sigma=10, rho=28, beta=8.0/3.0, dt=0.01;
  const crd_t initial_condition(space_dimension, 1.0);

  ptr_t       attractor = ptr_t(new lorenz(initial_condition,sigma,rho,beta));
  const crd_t &output   = attractor->output();

  try {
    std::cout << fmt(output, 12, 10) << "\n";

    for (int step = 1; step <= num_steps; ++step) {
      integrate (attractor, dt);
      std::cout << fmt(output, 12, 10) << "\n";
    }
  } catch(std::exception &e) {
    std::cerr << "Error exit following exception of type " << e.what() << "\n";
    return EXIT_FAILURE;
  } catch(...) {
    std::cerr << "Error exit following an unknown exception type\n";
    return EXIT_FAILURE;
  }
  return EXIT_SUCCESS;
}
```

lorenz.h

```
#ifndef __H_LORENZ__
#define __H_LORENZ__ 1

#include "integrable_model.h"

class lorenz : public integrable_model {
public:
  typedef Ref<lorenz> ptr_t;
  lorenz ();
  lorenz (const crd_t, real_t sigma, real_t rho, real_t beta);
  // Default copy and assignment operators are just fine for this type.

public:
  integrable_model::ptr_t   d_dt() const;
  void                      operator+=(integrable_model::ptr_t other);
  integrable_model::ptr_t   operator*(float val) const;

  const crd_t&              output() const;

```

```
20    virtual ~lorenz();
21
22  private:
23    crd_t state_;     // solution vector.
24    float sigma_, rho_, beta_;
25  };
26
27  #endif
```

lorenz.cpp

```
1   #include <iostream>
2   #include <exception>
3
4   #include "lorenz.h"
5
6   using namespace std;
7
8   struct LorenzError : public std::exception {
9     virtual ~LorenzError() throw() {}
10  };
11
12  // default constructor
13  lorenz::lorenz ()
14  {}
15
16  // constructor using each element
17  lorenz::lorenz (const crd_t initial_state, real_t s, real_t r, real_t b) :
18    state_(initial_state), sigma_(s), rho_(r), beta_(b)
19  {}
20
21  const crd_t& lorenz::output() const {
22    return state_;
23  }
24
25  integrable_model::ptr_t lorenz:: d_dt() const
26  {
27    ptr_t result = ptr_t(new lorenz);
28    result->state_.resize(3);
29    result->state_.at(0) = sigma_*(state_.at(1) - state_.at(0));
30    result->state_.at(1) = state_.at(0)*(rho_ - state_.at(2)) - state_.at(1);
31    result->state_.at(2) = state_.at(0)*state_.at(1) - beta_*state_.at(2);
32    return result;
33  }
34
35
36  void lorenz::operator+=(integrable_model::ptr_t rhs) {
37    ptr_t other = cast<lorenz>(rhs);
38    if(other == NULL) {
39      std::cerr << "lorenz::operator+=:  Failed dynamic cast\n";
40      throw LorenzError();
41    }
42    if(other->state_.size() != this->state_.size()) {
43      std::cerr << "lorenz::operator+=:  Non-identical dimensions.\n";
44      throw LorenzError();
45    }
46
47    for(size_t i = 0; i < state_.size(); ++i) {
48      state_.at(i) += other->state_.at(i);
49    }
50  }
51
52  integrable_model::ptr_t lorenz::operator*(real_t rhs) const
53  {
54    ptr_t result = ptr_t(new lorenz(*this));
55    for(size_t i = 0; i < result->state_.size(); ++i) {
56      result->state_.at(i) *= rhs;
57    }
58    return result;
59  }
60
61  lorenz::~lorenz()
62  {}
```

integrable_model.h

```
1   #ifndef __H_INTEGRABLE_MODEL__
2   #define __H_INTEGRABLE_MODEL__ 1
3
4   #include "globals.h"
5
6   class integrable_model : virtual public RefBase {
7   public:
8     typedef Ref<integrable_model> ptr_t;
9
10    virtual ~integrable_model();
11
12    virtual ptr_t d_dt() const = 0;
13    virtual void  operator+=(ptr_t) = 0;
14    virtual ptr_t operator*(real_t) const = 0;
15
16  protected:
17    integrable_model(const integrable_model&);
18    integrable_model();
```

```
};

void integrate (integrable_model::ptr_t, real_t);

#endif
```

integrable_model.cpp

```
#include "integrable_model.h"

typedef integrable_model::ptr_t ptr_t;

integrable_model::integrable_model() : RefBase() {
}

integrable_model::integrable_model(const integrable_model&) : RefBase() {
}

integrable_model::~integrable_model() {
}

void integrate (ptr_t model, real_t dt) {
  *model += *(model->d_dt()) * dt;
}
```

B.2 Strategy Example

main.cpp

```
#include "timed_lorenz.h"
#include "explicit_euler.h"
#include "runge_kutta_2nd.h"
#include "fmt.h"
#include <iostream>

int main () {
  typedef lorenz::ptr_t ptr_t;

  static const int num_steps=10;
  const real_t  sigma=10., rho=28., beta=8./3., dt=0.01;
  crd_t initial_condition(3, 1.0);

  Ref<lorenz> attractor = new lorenz(initial_condition, sigma, rho, beta,
                                     new explicit_euler);
  std::cout << "lorenz attractor:\n"
       << fmt(attractor->coordinate(), 12, 9) << "\n";
  for (int step = 0; step < 4*num_steps; ++step) {
    attractor->integrate(0.25*dt);
    std::cout << fmt(attractor->coordinate(), 12, 9) << "\n";
  }

  Ref<timed_lorenz> timed_attractor
    = new timed_lorenz(initial_condition, sigma, rho, beta,
                       new runge_kutta_2nd);
  std::cout << "\ntimed lorenz attractor:\n"
            << fmt(timed_attractor->coordinate(), 12, 9) << "\n";
  for (int i = 0; i < num_steps; ++i) {
    timed_attractor->integrate(dt);
    std::cout << fmt(timed_attractor->coordinate(), 12, 9) << "\n";
  }

  return 0;
}
```

integrable_model.h

```
#ifndef _H_INTEGRABLE_MODEL_
#define _H_INTEGRABLE_MODEL_

#include "strategy.h"
#include "RefBase.h"
#include "globals.h"

class integrable_model : virtual public RefBase {
public:
  typedef Ref<integrable_model> ptr_t;
  typedef strategy::ptr_t strategy_t;

  integrable_model(strategy_t);
  integrable_model(const integrable_model&);
  virtual ~integrable_model();

  void set_strategy (strategy_t);
  strategy_t get_strategy () const;
  void integrate (real_t);

  virtual ptr_t clone() const = 0;
  virtual ptr_t d_dt() const = 0;
  virtual ptr_t operator+=(ptr_t) = 0;
  virtual ptr_t operator*=(real_t) = 0;

private:
```

```
27    strategy_t quadrature_;
28  };
29
30  #include "model_ops.h"
31
32  #endif //!_H_INTEGRABLE_MODEL_
```

integrable_model.cpp

```
1   #include "integrable_model.h"
2   #include <exception>
3
4   struct integrable_model_error : public std::exception {
5     virtual ~integrable_model_error() throw() {}
6   };
7
8   typedef integrable_model::ptr_t ptr_t;
9   typedef integrable_model::strategy_t strategy_t;
10
11  integrable_model::integrable_model(strategy_t quad) :
12    RefBase(), quadrature_(quad)
13  {}
14
15  integrable_model::integrable_model(const integrable_model& other) :
16    RefBase(), quadrature_(other.quadrature_)
17  {}
18
19  integrable_model::~integrable_model() {
20  }
21
22  void integrable_model::set_strategy (strategy_t quad) {
23    quadrature_ = quad;
24  }
25
26  strategy_t integrable_model::get_strategy () const {
27    return quadrature_;
28  }
29
30  void integrable_model::integrate (real_t dt) {
31    quadrature_->integrate(this, dt);
32  }
```

lorenz.h

```
1   #ifndef _H_LORENZ_
2   #define _H_LORENZ_
3
4   #include "integrable_model.h"
5
6   class lorenz : public integrable_model {
7   public:
8     typedef integrable_model::ptr_t ptr_t;
9     typedef integrable_model::strategy_t strategy_t;
10
11    lorenz(const crd_t&, real_t sigma, real_t rho, real_t beta, strategy_t);
12    virtual ~lorenz();
13
14    virtual ptr_t clone() const;
15    virtual ptr_t d_dt() const;
16    virtual ptr_t operator+=(ptr_t);
17    virtual ptr_t operator*=(real_t);
18
19    void set_coordinate(const crd_t&);
20    const crd_t& coordinate() const;
21    real_t sigma() const;
22    real_t rho() const;
23    real_t beta() const;
24
25  private:
26    crd_t state_;
27    real_t sigma_, rho_, beta_;
28  };
29
30  #endif // !_H_LORENZ_
```

lorenz.cpp

```
1   #include "lorenz.h"
2   #include "fmt.h"
3   #include <exception>
4
5   struct lorenz_error : public std::exception {
6     virtual ~lorenz_error() throw() {}
7   };
8
9   typedef lorenz::ptr_t ptr_t;
10  typedef lorenz::strategy_t strategy_t;
11
12  lorenz::lorenz(const crd_t& ste, real_t s, real_t r, real_t b, strategy_t str) :
13    integrable_model(str), state_(ste), sigma_(s), rho_(r), beta_(b)
14  {}
15
16  lorenz::~lorenz() {
17  }
18
```

```
ptr_t lorenz::clone() const {
  return ptr_t(new lorenz(*this));
}

ptr_t lorenz::d_dt() const {
  crd_t new_state(3);
  new_state.at(0) = sigma_ * (state_.at(1) - state_.at(0));
  new_state.at(1) = state_.at(0) * (rho_ - state_.at(2)) - state_.at(1);
  new_state.at(2) = state_.at(0) * state_.at(1) - beta_ * state_.at(2);
  return ptr_t(new lorenz(new_state, sigma_, rho_, beta_, get_strategy()));
}

ptr_t lorenz::operator+=(ptr_t inval) {
  Ref<lorenz> other = cast<lorenz>(inval);
  if((other == NULL) || (state_.size() != other->state_.size())) {
    std::cerr << "lorenz::operator+=:  Invalid input argument\n";
    throw lorenz_error();
  }
  size_t size = state_.size();
  for(size_t i = 0; i < size; ++i) {
    state_.at(i) += other->state_.at(i);
  }
  return ptr_t(this);
}

ptr_t lorenz::operator*=(real_t val) {
  size_t size = state_.size();
  for(size_t i = 0; i < size; ++i) {
    state_.at(i) *= val;
  }
  return ptr_t(this);
}

void lorenz::set_coordinate(const crd_t& state) {
  state_ = state;
}

const crd_t& lorenz::coordinate() const {
  return state_;
}

real_t lorenz::sigma() const {
  return sigma_;
}

real_t lorenz::rho() const {
  return rho_;
}

real_t lorenz::beta() const {
  return beta_;
}
```

timed_lorenz.h

```
#ifndef _H_TIMED_LORENZ_
#define _H_TIMED_LORENZ_

#include "strategy.h"
#include "lorenz.h"


class timed_lorenz : public lorenz {
public:
  typedef lorenz::ptr_t ptr_t;
  typedef lorenz::strategy_t strategy_t;

  timed_lorenz(const crd_t&, real_t sigma, real_t rho, real_t beta,
               strategy_t, double t_init = 0);
  virtual ~timed_lorenz();
  virtual ptr_t clone() const;
  virtual ptr_t d_dt() const;
  virtual ptr_t operator+=(ptr_t);
  virtual ptr_t operator*=(real_t);

  void   set_time (real_t);
  real_t get_time() const;
private:
  real_t time_;
};

#endif //!_H_TIMED_LORENZ_
```

timed_lorenz.cpp

```
#include "timed_lorenz.h"
#include <exception>

struct timed_lorenz_error : public std::exception {
  virtual ~timed_lorenz_error() throw() {}
};

typedef timed_lorenz::ptr_t ptr_t;
typedef timed_lorenz::strategy_t strategy_t;
```

```
10
11  timed_lorenz::timed_lorenz(const crd_t& ste, real_t s, real_t r, real_t b,
12                             strategy_t strat, double t_init) :
13    lorenz(ste, s, r, b, strat), time_(t_init)
14  {}
15
16  timed_lorenz::~timed_lorenz() {
17  }
18
19  ptr_t timed_lorenz::clone() const {
20    return ptr_t(new timed_lorenz(*this));
21  }
22
23  ptr_t timed_lorenz::d_dt() const {
24    Ref<lorenz> parent = cast<lorenz>(lorenz::d_dt());
25    return ptr_t(new timed_lorenz(parent->coordinate(), parent->sigma(),
26                                  parent->rho(), parent->beta(),
27                                  parent->get_strategy(), 1.0));
28  }
29
30  ptr_t timed_lorenz::operator+=(ptr_t inval) {
31    Ref<timed_lorenz> other = cast<timed_lorenz>(inval);
32    if(other == NULL) {
33      std::cerr << "timed_lorenz::operator+=:  Invalid input type\n";
34      throw timed_lorenz_error();
35    }
36    lorenz::operator+=(other);
37    time_ += other->time_;
38    return ptr_t(this);
39  }
40
41  ptr_t timed_lorenz::operator*=(real_t val) {
42    lorenz::operator*=(val);
43    time_ *= val;
44    return ptr_t(this);
45  }
46
47  void timed_lorenz::set_time (real_t t) {
48    time_ = t;
49  }
50
51  real_t timed_lorenz::get_time() const {
52    return time_;
53  }
```

explicit_euler.h

```
1   #ifndef _H_EXPLICIT_EULER_
2   #define _H_EXPLICIT_EULER_
3
4   #include "strategy.h"
5   #include "integrable_model.h"
6
7   class explicit_euler : public strategy
8   {
9   public:
10    virtual ~explicit_euler();
11    virtual void integrate (model_t this_obj, real_t dt) const;
12  };
13
14  #endif //!_H_EXPLICIT_EULER_
```

explicit_euler.cpp

```
1   #include "explicit_euler.h"
2   #include "integrable_model.h"
3   #include <exception>
4
5   explicit_euler::~explicit_euler()
6   {}
7
8   void explicit_euler::integrate (model_t this_obj, real_t dt) const {
9     *this_obj += this_obj->d_dt() * dt;
10  }
```

runge_kutta_2nd.h

```
1   #ifndef _H_RUNGE_KUTTA_2ND_
2   #define _H_RUNGE_KUTTA_2ND_
3
4   #include "strategy.h"
5   #include "integrable_model.h"
6
7   class runge_kutta_2nd : public strategy {
8   public:
9     virtual ~runge_kutta_2nd();
10    virtual void integrate(model_t this_obj, real_t dt) const;
11  };
12
13  #endif
```

runge_kutta_2nd.cpp

```
1   #include <iostream>
2   #include <exception>
```

```

#include "integrable_model.h"
#include "runge_kutta_2nd.h"

using namespace std;

runge_kutta_2nd::~runge_kutta_2nd() {
}

void runge_kutta_2nd::integrate (model_t this_obj, real_t dt) const {
  model_t this_half = this_obj + this_obj->d_dt() * (0.5*dt); // predictor
  *this_obj += this_half->d_dt() * dt; // corrector
}
```

strategy.h

```
#ifndef _H_STRATEGY_
#define _H_STRATEGY_

#include "globals.h"
#include "RefBase.h"

class integrable_model;
class strategy : public RefBase {
public:
  typedef Ref<strategy> ptr_t;
  typedef Ref<integrable_model> model_t;

  virtual ~strategy() {}
  virtual void integrate (model_t this_obj, real_t dt) const = 0;
};

#endif //!_H_STRATEGY_
```

model_ops.h

```
#ifndef _H_MODEL_OPS_
#define _H_MODEL_OPS_

inline integrable_model::ptr_t
operator+(integrable_model::ptr_t a, integrable_model::ptr_t b)
{
  integrable_model::ptr_t tmp = a->clone();
  *tmp += b;
  return tmp;
}

inline integrable_model::ptr_t operator*(integrable_model::ptr_t a, real_t b) {
  integrable_model::ptr_t tmp = a->clone();
  *tmp *= b;
  return tmp;
}

#endif //! _H_MODEL_OPS_
```

B.3 Puppeteer Example

main.cpp

```
#include "atmosphere.h"
#include "fmt.h"

typedef integrable_model::ptr_t ptr_t;

int main() {
  const int num_steps=1000;
  const real_t x=1., y=1., z=1., sigma=10., rho=28, beta=8./3., dt=0.02;

  air sky(x, sigma);
  cloud puff(y, rho);
  ground earth(z, beta);
  ptr_t boundary_layer = ptr_t(new atmosphere(sky, puff, earth));

  real_t t = 0.;
  std::cout << fmt(t,5,2) << " "
            << fmt(boundary_layer->state_vector()) << "\n";
  for(int step = 1; step <= num_steps; ++step) {
    integrate(boundary_layer, dt);
    t += dt;
    std::cout << fmt(t,5,2) << " "
              << fmt(boundary_layer->state_vector()) << "\n";
  }
}
```

integrable_model.h

```
#ifndef _H_INTEGRABLE_MODEL_
#define _H_INTEGRABLE_MODEL_

#include "mat.h"
#include "RefBase.h"

class integrable_model : virtual public RefBase {
```

```
public:
  typedef Ref<integrable_model> ptr_t;

  integrable_model();
  integrable_model(const integrable_model&);
  virtual ~integrable_model();

  // The following methods do dynamic allocation (yuck).
  virtual ptr_t d_dt() const = 0;
  virtual void  dRHS_dV(mat_t &result) const  = 0;
  virtual ptr_t clone() const = 0;
  virtual ptr_t  inverse_times(const mat_t&) const = 0;
  virtual crd_t state_vector() const = 0;

  // The following methods are destructive updates.
   virtual ptr_t operator+=(ptr_t) = 0;
  virtual ptr_t operator-=(ptr_t) = 0;
  virtual ptr_t operator*=(real_t) = 0;
};

// Integration routine.
void integrate(integrable_model::ptr_t state, double dt);

#endif //!_H_INTEGRABLE_MODEL_
```

integrable_model.cpp

```
#include "integrable_model.h"
#include "model_ops.h"

integrable_model::integrable_model() : RefBase() {
}

integrable_model::integrable_model(const integrable_model&) : RefBase() {
}

integrable_model::~integrable_model() {
}

void integrate(ptr_t state, double dt) {
  static const int num_iterations = 5;
  ptr_t initial = state->clone();
  mat_t identity, dRHS_dState;
  for(int iteration = 0; iteration < num_iterations; ++iteration) {
    state->dRHS_dV(dRHS_dState);
    identity.identity(dRHS_dState.rows());
    mat_t jacobian = identity - (0.5*dt) * dRHS_dState;
    ptr_t residual= state-(initial+((initial->d_dt()+state->d_dt())*(0.5*dt)));
    ptr_t scratch = residual->inverse_times(jacobian);
    *state -= scratch;
  }
}
```

atmosphere.h

```
#ifndef _H_ATMOSPHERE_
#define _H_ATMOSPHERE_

#include "integrable_model.h"
#include "air.h"
#include "cloud.h"
#include "ground.h"

class atmosphere : public integrable_model {
public:
  typedef integrable_model::ptr_t ptr_t;
  typedef Ref<atmosphere> self_t;

  atmosphere(const air&, const cloud&, const ground&);
  virtual ~atmosphere();

  // The following methods do dynamic allocation (yuck).
  virtual ptr_t d_dt() const;
  virtual void  dRHS_dV(mat_t&) const;
  virtual ptr_t clone() const;
  virtual ptr_t  inverse_times(const mat_t&) const;
  virtual crd_t state_vector() const;

  // The following methods are destructive updates.
  virtual ptr_t operator+=(ptr_t);
  virtual ptr_t operator-=(ptr_t);
  virtual ptr_t operator*=(real_t);

private:
  air air_;
  cloud cloud_;
  ground ground_;
};

#endif //!_H_ATMOSPHERE_
```

atmosphere.cpp

```
#include "atmosphere.h"
```

```
#include <exception>
#include <cmath>

struct atmosphere_error : public std::exception {
  virtual ~atmosphere_error() throw() {}
};

typedef atmosphere::ptr_t ptr_t;

atmosphere::atmosphere(const air &a, const cloud &c, const ground &g) :
  air_(a), cloud_(c), ground_(g)
{}

atmosphere::~atmosphere() {
}

ptr_t atmosphere::d_dt() const {
  return
    ptr_t(new atmosphere(air_.d_dt(cloud_.coordinate()),
                         cloud_.d_dt(air_.coordinate(),ground_.coordinate()),
                         ground_.d_dt(air_.coordinate(),cloud_.coordinate())));
}

void atmosphere::dRHS_dV(mat_t &result) const {
  // Figure out the required dimensions.
  dim_t adim = air_.dimensions();
  dim_t cdim = cloud_.dimensions();
  dim_t gdim = ground_.dimensions();
  // Resize the result matrix.
  result.clear_resize(adim.eqs  + cdim.eqs  + gdim.eqs,
                      adim.vars + cdim.vars + gdim.vars);
  if(result.rows() != result.cols()) {
    std::cerr << "atmosphere::dRHS_dV:  Ill-formed problem:  total of "
              << result.rows() << " equations and " << result.cols()
              << " variables\n";
    throw atmosphere_error();
  }
  // dAir/dAir
  result.set_submat(0, 0, air_.d_dAir(cloud_.coordinate()));
  // dAir/dCloud
  result.set_submat(0, adim.vars, air_.d_dy(cloud_.coordinate()));
  // dAir/dGround is all zero -- skipping that one.
  // dCloud/dAir
  result.set_submat(adim.eqs, 0,
                    cloud_.d_dx(air_.coordinate(),ground_.coordinate()));
  // dCloud/dCloud
  result.set_submat(adim.eqs, adim.vars,
                    cloud_.d_dCloud(air_.coordinate(),ground_.coordinate()));
  // dCloud/dGround
  result.set_submat(adim.eqs, adim.vars+cdim.vars,
                    cloud_.d_dz(air_.coordinate(),ground_.coordinate()));
  // dGround/dAir
  result.set_submat(adim.eqs+cdim.eqs, 0,
                    ground_.d_dx(air_.coordinate(),cloud_.coordinate()));
  // dGround/dCloud
  result.set_submat(adim.eqs+cdim.eqs, adim.vars,
                    ground_.d_dy(air_.coordinate(),cloud_.coordinate()));
  // dGround/dGround
  result.set_submat(adim.eqs+cdim.eqs, adim.vars+cdim.vars,
                    ground_.d_dGround(air_.coordinate(),cloud_.coordinate()));
}

ptr_t atmosphere::clone() const {
  return ptr_t(new atmosphere(*this));
}

crd_t atmosphere::state_vector() const {
  const crd_t &cc = cloud_.coordinate(), &gc = ground_.coordinate();
  crd_t state_space = air_.coordinate();
  state_space.insert(state_space.end(), cc.begin(), cc.end());
  state_space.insert(state_space.end(), gc.begin(), gc.end());
  return state_space;
}


ptr_t atmosphere::inverse_times(const mat_t &lhs) const {
  static const real_t pivot_tolerance = 1e-2;

  const int n = lhs.rows();
  crd_t b = this->state_vector();
  if((n != lhs.cols()) || (size_t(n) != b.size())) {
    std::cerr <<"integrable_model::inverse_times:  ill-posed matrix problem\n";
    throw atmosphere_error();
  }
  crd_t x(n);
  mat_t A(lhs);
  for(int p = 0; p < n-1; ++p) { // forward elimination
    if(fabs(A(p,p)) < pivot_tolerance) {
      std::cerr << "integrable_model::inverse_times:  "
                << "use an algorithm with pivoting\n";
      throw atmosphere_error();
    }
    for(int row = p+1; row < n; ++row) {
```

```
      real_t factor = A(row,p) / A(p,p);
      for(int col = p; col < n; ++col) {
        A(row,col) = A(row,col) - A(p,col)*factor;
      }
      b.at(row) = b.at(row)- b.at(p)*factor;
    }
  }
  x.at(n-1) = b.at(n-1) / A(n-1,n-1); // back substitution
  for(int row = n-1; row >= 0; --row) {
    real_t the_sum = 0;
    for(int col = row+1; col < n; ++col) {
      the_sum += A(row,col) * x.at(col);
    }
    x.at(row) = (b.at(row) - the_sum) / A(row,row);
  }
  return ptr_t(new atmosphere(air(x.at(0), x.at(1)),
                              cloud(x.at(2), cloud_.rho()),
                              ground(x.at(3), ground_.beta())));
}

ptr_t atmosphere::operator+=(ptr_t other) {
  self_t added = cast<atmosphere>(other);
  if(other == NULL) {
    std::cerr << "atmosphere::operator+=:  Invalid input type\n";
    throw atmosphere_error();
  }
  air_    += added->air_;
  cloud_  += added->cloud_;
  ground_ += added->ground_;
  return ptr_t(this);
}

ptr_t atmosphere::operator-=(ptr_t other) {
  self_t subbed = cast<atmosphere>(other);
  if(other == NULL) {
    std::cerr << "atmosphere::operator-=:  Invalid input type\n";
    throw atmosphere_error();
  }
  air_    -= subbed->air_;
  cloud_  -= subbed->cloud_;
  ground_ -= subbed->ground_;
  return ptr_t(this);
}

ptr_t atmosphere::operator*=(real_t value) {
  air_    *= value;
  cloud_  *= value;
  ground_ *= value;
  return ptr_t(this);
}
```

air.h

```
#ifndef _H_AIR_
#define _H_AIR_

#include "mat.h"

class air {
public:
  air(real_t x, real_t sigma);

  const crd_t& coordinate() const;
  air d_dt(const crd_t&) const;
  mat_t d_dAir(const crd_t&) const;
  mat_t d_dy(const crd_t&) const;
  air& operator+=(const air&);
  air& operator-=(const air&);
  air& operator*=(real_t);

  inline dim_t dimensions() const { return dim_t(dim_, dim_); }

private:
  static const int dim_;
  crd_t x_;  // sigma is stored at x_[1]
};

#endif //!_H_AIR_
```

air.cpp

```
#include "air.h"

const int air::dim_ = 2;

air::air(real_t x, real_t sigma) {
  x_.push_back(x);
  x_.push_back(sigma);
}

const crd_t& air::coordinate() const {
  return x_;
}
```

```
air air::d_dt(const crd_t &y) const {
  return air((x_.at(1) * (y.at(0) - x_.at(0))), 0);
}

mat_t air::d_dAir(const crd_t& y) const {
  mat_t result(dim_, dim_);
  result(0, 0) = -x_.at(1);
  result(0, 1) = y.at(0) - x_.at(0);
  return result;
}

mat_t air::d_dy(const crd_t &y) const {
  mat_t result(dim_, y.size());
  result(0, 0) = x_.at(1);
  return result;
}

air& air::operator+=(const air &other) {
  x_.at(0) += other.x_.at(0);
  x_.at(1)   += other.x_.at(1);
  return *this;
}

air& air::operator-=(const air &other) {
  x_.at(0) -= other.x_.at(0);
  x_.at(1)   -= other.x_.at(1);
  return *this;
}

air& air::operator*=(real_t value) {
  x_.at(0) *= value;
  x_.at(1) *= value;
  return *this;
}
```

cloud.h

```
#ifndef _H_CLOUD_
#define _H_CLOUD_

#include "mat.h"

class cloud {
public:
  cloud(real_t y, real_t rho);

  const crd_t& coordinate() const;
  real_t rho() const;
  cloud d_dt(const crd_t&, const crd_t&) const;
  mat_t d_dCloud(const crd_t&, const crd_t&) const;
  mat_t d_dx(const crd_t&, const crd_t&) const;
  mat_t d_dz(const crd_t&, const crd_t&) const;
  cloud& operator+=(const cloud&);
  cloud& operator-=(const cloud&);
  cloud& operator*=(real_t);

  inline dim_t dimensions() const { return dim_t(dim_, dim_); }

private:
  static const int dim_;
  crd_t y_;
  real_t rho_;
};

#endif //!_H_CLOUD_
```

cloud.cpp

```
#include "cloud.h"

const int cloud::dim_ = 1;

cloud::cloud(real_t y, real_t rho) :
  y_(1, y), rho_(rho)
{}

const crd_t& cloud::coordinate() const {
  return y_;
}

real_t cloud::rho() const {
  return rho_;
}

cloud cloud::d_dt(const crd_t &x, const crd_t &z) const {
  return cloud((x.at(0) * (rho_ - z.at(0)) - y_.at(0)), 0);
}

mat_t cloud::d_dCloud(const crd_t&, const crd_t&) const {
  mat_t result(dim_, dim_);
  result(0, 0) = -1;
  return result;
}

```

```
mat_t cloud::d_dx(const crd_t &x, const crd_t &z) const {
  mat_t result(dim_, x.size());
  result(0, 0) = rho_ - z.at(0);
  return result;
}

mat_t cloud::d_dz(const crd_t &x, const crd_t &z) const {
  mat_t result(dim_, z.size());
  result(0, 0) = -x.at(0);
  return result;
}

cloud& cloud::operator+=(const cloud &other) {
  y_.at(0) += other.y_.at(0);
  rho_ += other.rho_;
  return *this;
}
cloud& cloud::operator-=(const cloud &other) {
  y_.at(0) -= other.y_.at(0);
  rho_ -= other.rho_;
  return *this;
}

cloud& cloud::operator*=(real_t value) {
  y_.at(0) *= value;
  rho_ *= value;
  return *this;
}
```

ground.h

```
#ifndef _H_GROUND_
#define _H_GROUND_

#include "mat.h"

class ground {
public:
  ground(real_t y, real_t rho);

  const crd_t& coordinate() const;
  real_t beta() const;
  ground d_dt(const crd_t&, const crd_t&) const;
  mat_t d_dGround(const crd_t&, const crd_t&) const;
  mat_t d_dx(const crd_t&, const crd_t&) const;
  mat_t d_dy(const crd_t&, const crd_t&) const;
  ground& operator+=(const ground&);
  ground& operator-=(const ground&);
  ground& operator*=(real_t);

  inline dim_t dimensions() const { return dim_t(dim_, dim_); }

private:
  static const int dim_;
  crd_t z_;
  real_t beta_;
};

#endif //!_H_GROUND_
```

ground.cpp

```
#include "ground.h"

const int ground::dim_ = 1;

ground::ground(real_t z, real_t beta) :
  z_(1, z), beta_(beta)
{}

const crd_t& ground::coordinate() const {
  return z_;
}

real_t ground::beta() const {
  return beta_;
}

ground ground::d_dt(const crd_t& x, const crd_t& y) const {
  return ground((x.at(0) * y.at(0) - beta_ * z_.at(0)), 0);
}

mat_t ground::d_dGround(const crd_t&, const crd_t&) const {
  mat_t result(dim_, dim_);
  result(0, 0) = -beta_;
  return result;
}

mat_t ground::d_dx(const crd_t &x, const crd_t &y) const {
  mat_t result(dim_, x.size());
  result(0, 0) = y.at(0);
  return result;
}

```

```
33 mat_t ground::d_dy(const crd_t &x, const crd_t &y) const {
34   mat_t result(dim_, y.size());
35   result(0, 0) = x.at(0);
36   return result;
37 }
38
39 ground& ground::operator+=(const ground &other) {
40   z_.at(0) += other.z_.at(0);
41   beta_ += other.beta_;
42   return *this;
43 }
44
45 ground& ground::operator-=(const ground& other) {
46   z_.at(0) -= other.z_.at(0);
47   beta_ -= other.beta_;
48   return *this;
49 }
50
51 ground& ground::operator*=(real_t value) {
52   z_.at(0) *= value;
53   beta_ *= value;
54   return *this;
55 }
```

mat.h

```
1  #ifndef _H_MAT_
2  #define _H_MAT_
3
4  #include "globals.h"
5  #include <iostream>
6  #include <iomanip>
7
8  class mat_t {
9  public:
10   typedef crd_t::value_type value_type;
11   typedef crd_t::reference  reference;
12
13   mat_t();
14   mat_t(int rows, int cols);
15   void clear();
16   void resize(int rows, int cols);
17   void clear_resize(int rows, int cols, value_type value = 0);
18   void identity(int rows);
19   int rows() const;
20   int cols() const;
21   value_type operator()(int r, int c) const;
22   reference operator()(int r, int c);
23   void set_submat(int r, int c, const mat_t &other);
24   mat_t& operator-=(const mat_t&);
25   mat_t& operator*=(real_t);
26
27 private:
28   int r_, c_;
29   crd_t data_;
30 };
31
32 inline mat_t operator-(const mat_t &a, const mat_t &b) {
33   mat_t retval(a);
34   retval -= b;
35   return retval;
36 }
37
38 inline mat_t operator*(real_t value, const mat_t &matrix) {
39   mat_t retval(matrix);
40   retval *= value;
41   return retval;
42 }
43
44 struct dim_t {
45   const int eqs;
46   const int vars;
47
48   dim_t(int eqcnt, int varcnt) :
49     eqs(eqcnt), vars(varcnt)
50   {}
51 };
52
53 inline std::ostream& operator<<(std::ostream &os, const mat_t &mat) {
54   std::ios_base::fmtflags flags = os.flags();
55   for(int r = 0; r < mat.rows(); ++r) {
56     os << "[";
57     for(int c = 0; c < mat.cols(); ++c) {
58       os << " " <<std::setw(12) <<std::setprecision(8) <<std::fixed <<mat(r,c);
59     }
60     os << "]\n";
61   }
62   os.flags(flags);
63   return os;
64 }
65
66 #endif // !_H_MAT_
```

mat.cpp

```
#include "mat.h"
#include <exception>
#include <iostream>

struct matrix_error : public std::exception {
  virtual ~matrix_error() throw() {}
};

mat_t::mat_t() :
  r_(0), c_(0)
{}

mat_t::mat_t(int rows, int cols) {
  this->resize(rows, cols);
}

void mat_t::clear() {
  r_ = c_ = 0;
  data_.clear();
}

void mat_t::resize(int rows, int cols) {
  if(rows < 0 || cols < 0) {
    std::cerr << "mat_t::resize:  Rows and columns must be >= 0.\n";
    throw matrix_error();
  }
  if(! data_.empty()) {
    // Copy data over.
  }
  else {
    // common case.
    data_.resize(rows*cols);
    r_ = rows;
    c_ = cols;
  }
}

void mat_t::clear_resize(int rows, int cols, value_type value) {
  if(rows < 0 || cols < 0) {
    std::cerr << "mat_t::clear_resize:  Rows and columns must be >= 0\n";
    throw matrix_error();
  }
  data_.resize(rows*cols);
  r_ = rows;
  c_ = cols;
  std::fill(data_.begin(), data_.end(), value);
}

void mat_t::identity(int size) {
  this->clear_resize(size, size, 0);
  for(int i = 0; i < size; ++i) {
    this->operator()(i,i) = 1;
  }
}

int mat_t::rows() const {
  return r_;
}

int mat_t::cols() const {
  return c_;
}

mat_t::value_type mat_t::operator()(int r, int c) const {
  if(r < 0 || r >= r_ || c < 0 || c >= c_) {
    std::cerr << "mat_t::operator():  Invalid index (" << r << ", " << c
              << ").  Bounds are (" << r_ << ", " << c << ")\n";
    throw matrix_error();
  }
  return data_.at(c*r_ + r);
}

mat_t::reference mat_t::operator()(int r, int c) {
  if(r < 0 || r >= r_ || c < 0 || c >= c_) {
    std::cerr << "mat_t::operator():  Invalid index (" << r << ", " << c
              << ").  Bounds are (" << r_ << ", " << c << ")\n";
    throw matrix_error();
  }
  return data_.at(c*r_ + r);
}

void mat_t::set_submat(int startrow, int startcol, const mat_t &other) {
  for(int r = 0; r < other.rows(); ++r) {
    for(int c = 0; c < other.cols(); ++c) {
      this->operator()(r+startrow, c+startcol) = other(r,c);
    }
  }
}

mat_t& mat_t::operator-=(const mat_t &other) {
  if(this->rows() != other.rows() || this->cols() != other.cols()) {
    std::cerr << "mat_t::operator-=:  Matrices must be of identical size.\n";
```

```
93      throw matrix_error();
94    }
95    const size_t size = data_.size();
96    for(size_t i = 0; i < size; ++i)
97      data_[i] -= other.data_[i];
98    return *this;
99  }
100
101 mat_t& mat_t::operator*=(real_t value) {
102   for(crd_t::iterator it = data_.begin(); it != data_.end(); ++it)
103     *it *= value;
104   return *this;
105 }
```

model_ops.h

```
1   #ifndef _H_INTEGRABLE_MODEL_OPS
2   #define _H_INTEGRABLE_MODEL_OPS
3
4   #include "integrable_model.h"
5   #include <exception>
6
7   struct model_ops_exception : public std::exception {
8     virtual ~model_ops_exception() throw() {}
9   };
10
11  typedef Ref<integrable_model> ptr_t;
12
13  inline ptr_t operator+(ptr_t a, ptr_t b) {
14    if(a == NULL || b == NULL) {
15      std::cerr << "ptr_t + ptr_t:  Neither pointer must be NULL\n";
16      throw model_ops_exception();
17    }
18    ptr_t c = a->clone();
19    *c += b;
20    return c;
21  }
22
23  inline ptr_t operator-(ptr_t a, ptr_t b) {
24    if(a == NULL || b == NULL) {
25      std::cerr << "ptr_t - ptr_t:  Neither pointer must be NULL\n";
26      throw model_ops_exception();
27    }
28    ptr_t c = a->clone();
29    *c -= b;
30    return c;
31  }
32
33  inline ptr_t operator*(ptr_t a, real_t b) {
34    if(a == NULL) {
35      std::cerr << "ptr_t * real_t:  Pointer must not be NULL\n";
36      throw model_ops_exception();
37    }
38    ptr_t c = a->clone();
39    *c *= b;
40    return c;
41  }
42
43  inline ptr_t operator*(real_t b, ptr_t a) {
44    return a*b;
45  }
46
47  #endif //!_H_INTEGRABLE_MODEL_OPS
```