The Art of Computational Science

The Kali Code

vol. 17

Individual Time Steps: Arbitrary Order Integrators

Piet Hut and Jun Makino

September 13, 2007

Contents

Preface **5**

0.1 xxx 5

1 Introduction **7**

1.1 xxx 7

2 XXX **9**

2.1 xxx 9

3 XXX **43**

3.1 xxx 43

4 XXX **47**

4.1 xxx 47

5 Literature References **51**

Preface

0.1 xxx

We thank xxx, xxx, and xxx for their comments on the manuscript.

Piet Hut and Jun Makino

Chapter 1

Introduction

1.1 xxx

Chapter 2

XXX

2.1 xxx

xxx

```
#!/usr/local/bin/ruby -w

require "nbody.rb"

module Integrator_force_default

  def setup_integrator
    @acc = @pos*0
  end

  def startup_force(wl, era)
    force(wl, era)
  end

  def force(wl, era)
    @acc = era.acc(wl, @pos, @time)
  end
end

module Integrator_pec_mode

include Integrator_force_default

  def integrator_step(wl, era)
    new_point = predict(@next_time)
```

```
    new_point.force(wl, era)
    new_point.correct(self, new_point.time - @time)
    new_point
  end
end

module Integrator_forward

  include Integrator_pec_mode

  def predict_pos_vel(dt)
    [ @pos + @vel*dt,  @vel + @acc*dt ]
  end

  def correct(old, dt)
    @pos, @vel = predict_pos_vel(dt)
  end

  def interpolate_pos_vel(wp, dt)
    predict_pos_vel(dt)
  end
end

module Integrator_forward_plus

  include Integrator_pec_mode

  def predict_pos_vel(dt)
    [ @pos + @vel*dt + (1/2.0)*@acc*dt**2,  @vel + @acc*dt ]
  end

  def correct(old, dt)
  end

  def interpolate_pos_vel(wp, dt)
    predict_pos_vel(dt)
  end
end

module Integrator_protohermite

  include Integrator_pec_mode

  attr_reader :acc

  def predict_pos_vel(dt)
```

```
    [ @pos + @vel*dt + (1/2.0)*@acc*dt**2,  @vel + @acc*dt ]
  end

  def correct(old, dt)
    @vel = old.vel + (1/2.0)*(old.acc + @acc)*dt
    @pos = old.pos + (1/2.0)*(old.vel + @vel)*dt
# same as leapfrog, apart from the an extra term in @pos, proportional to jerk:
#   @pos = old.pos + old.vel*dt + (1/4.0)*(old.acc + @acc)*dt^2
  end

  def interpolate_pos_vel(wp, dt)
    jerk = (wp.acc - @acc) / (wp.time - @time)
    [ @pos + @vel*dt + (1/2.0)*@acc*dt**2,
      @vel + @acc*dt + (1/2.0)*jerk*dt**2 ]
  end
end

module Integrator_leapfrog

  include Integrator_pec_mode

  attr_reader :acc

  def predict_pos_vel(dt)
    [ @pos + @vel*dt + (1/2.0)*@acc*dt**2,  @vel + @acc*dt ]
  end

  def correct(old, dt)
    @pos = old.pos + old.vel*dt + (1/2.0)*old.acc*dt**2
    @vel = old.vel + (1/2.0)*(old.acc + @acc)*dt
  end

  def interpolate_pos_vel(wp, dt)
    jerk = (wp.acc - @acc) / (wp.time - @time)
    [ @pos + @vel*dt + (1/2.0)*@acc*dt**2,
      @vel + @acc*dt + (1/2.0)*jerk*dt**2 ]
  end
end

module Integrator_multistep

  include Integrator_pec_mode

  attr_reader :acc, :acc_0_history, :time_history

  ORDER = 4
```

```
def setup_integrator
  @acc_0_history = []
  @time_history = []
  @acc = [@pos*0]
end

def force(wl, era)
  @acc[0] = era.acc(wl, @pos, @time)
end

def intermediate_point(old_ip, i)
  nil
end

def predict_pos_vel(dt)
  [ @pos + taylor_increment([@vel, *@acc], dt),
    @vel + taylor_increment(@acc, dt)          ]
end

def new_order(order)
  [order + 1, ORDER].min
end

def correct(old, dt)
  order = new_order(old.acc_0_history.size + 1)
  @acc_0_history = [old.acc[0], *old.acc_0_history][0...(order-1)]
  @time_history = [old.time, *old.time_history][0...(order-1)]
  make_taylor(@acc, [@acc[0], *@acc_0_history], [@time, *@time_history])
  @vel = old.vel - taylor_increment(@acc, -dt)
  @pos = old.pos - taylor_increment([@vel, *@acc], -dt)
end

def interpolate_pos_vel(wp, dt)
  if (wp.next_time - @time).abs < (@next_time - wp.time).abs
    predict_pos_vel(dt)
  else
    wp.predict_pos_vel(dt + @time - wp.time)
  end
end

def make_taylor(a, d_0, t)
  dt = []
  t.each do |time|
    dt.push(@time - time)
  end
```

```
    order = t.size
    d = [d_0]
    for k in (1...order)
      d.push([])
      for i in (0...(order-k))
        d[k][i] = (d[k-1][i] - d[k-1][i+1])/(t[i]-t[i+k])
      end
    end
    c = [[1]]
    for k in (1...order)
      c[k] = [0]
      for i in (1...k)
        c[k][i] = c[k-1][i-1] + dt[k-1] * c[k-1][i]
      end
      c[k][k] = 1
    end
    for j in (1...order)
      a[j] = a[0]*0
      for k in (j...order)
        a[j] += c[k][j] * d[k][0]
      end
      (1..j).each{|i| a[j] *= i}
    end
  end

  def taylor_increment(a, dt, number = 1)
    result = a[number-1]
    if number < a.size
      result += (1.0/(number+1))*taylor_increment(a, dt, number+1)
    end
    result*dt
  end
end

module Integrator_hermite

  include Integrator_pec_mode

  attr_reader :acc, :jerk

  def setup_integrator
    @acc = @pos*0
    @jerk = @pos*0
  end

  def force(wl, era)
```

```
    @acc, @jerk = era.acc_and_jerk(wl, @pos, @vel, @time)
  end

  def predict_pos_vel(dt)
    [ @pos + @vel*dt + (1/2.0)*@acc*dt**2 + (1/6.0)*@jerk*dt**3,
      @vel + @acc*dt + (1/2.0)*@jerk*dt**2                      ]
  end

  def correct(old, dt)
    @vel = old.vel + (1/2.0)*(old.acc + @acc)*dt +
                     (1/12.0)*(old.jerk - @jerk)*dt**2
    @pos = old.pos + (1/2.0)*(old.vel + @vel)*dt +
                     (1/12.0)*(old.acc - @acc)*dt**2
  end

  def interpolate_pos_vel(wp, dt)
    tau = wp.time - @time
    snap = (-6*(@acc - wp.acc) - 2*(2*@jerk + wp.jerk)*tau)/tau**2
    crackle = (12*(@acc - wp.acc) + 6*(@jerk + wp.jerk)*tau)/tau**3
    [ @pos + @vel*dt + (1/2.0)*@acc*dt**2 + (1/6.0)*@jerk*dt**3 +
                       (1/24.0)*snap*dt**4 + (1/144.0)*crackle*dt**5,
      @vel + @acc*dt + (1/2.0)*@jerk*dt**2 + (1/6.0)*snap*dt**3 +
                       (1/24.0)*crackle*dt**4                      ]
  end
end

module Integrator_rk4n                  # not partitioned

  include Integrator_force_default

  attr_reader :acc, :jerk
  attr_writer :time

  def setup_integrator
    @acc = @pos*0
    @jerk = @pos*0
  end

  def startup_force(wl, era)
    @acc, @jerk = era.acc_and_jerk(wl, @pos, @vel, @time)
  end

  def force_on_pos_at_time(pos, time, wl, era)
    era.acc(wl, pos, time)
  end
```

```
  def derivative(pos, vel, time, wl, era)
    [ vel, force_on_pos_at_time(pos,time,wl, era) ]
  end

  def integrator_step(wl, era)
    dt = @next_time - @time
    k1 = derivative(@pos,@vel,@time,wl,era)
    k2 = derivative(@pos+0.5*dt*k1[0],@vel+0.5*dt*k1[1],
                    @time+0.5*dt,wl,era)
    k3 = derivative(@pos+0.5*dt*k2[0],@vel+0.5*dt*k2[1],
                    @time+0.5*dt,wl,era)
    k4 = derivative(@pos+dt*k3[0],@vel+dt*k3[1],@time+dt,wl,era)
    new_point = deep_copy
    new_point.pos += (k1[0]+2*k2[0]+2*k3[0]+k4[0])*dt/6
    new_point.vel += (k1[1]+2*k2[1]+2*k3[1]+k4[1])*dt/6
    new_point.time=@next_time
    new_point.force(wl,era)
    new_point.estimate_jerk(self)
    new_point
  end

  def estimate_jerk(old)
    @jerk = (old.acc - @acc) / (old.time - @time)
  end

  def predict_pos_vel(dt)
    [ @pos + @vel*dt + (1/2.0)*@acc*dt**2 + (1/6.0)*@jerk*dt**3,
      @vel + @acc*dt + (1/2.0)*@jerk*dt**2                      ]
  end

  def interpolate_pos_vel(wp, dt)
    tau = wp.time - @time
    jerk = (-6*(@vel - wp.vel) - 2*(2*@acc + wp.acc)*tau)/tau**2
    snap = (12*(@vel - wp.vel) + 6*(@acc + wp.acc)*tau)/tau**3
    [ @pos + @vel*dt + (1/2.0)*@acc*dt**2 + (1/6.0)*jerk*dt**3 +
                    (1/24.0)*snap*dt**4,
      @vel + @acc*dt + (1/2.0)*jerk*dt**2 + (1/6.0)*snap*dt**3 ]
  end
end

module Integrator_rk2n                        # not partitioned

  include Integrator_force_default

  attr_reader :acc
  attr_writer :time
```

```
  def setup_integrator
    @acc = @pos*0
  end

  def force_on_pos_at_time(pos,time,wl, era)
    era.acc(wl, pos, time)
  end

  def derivative(pos,vel,time,wl,era)
    [ vel, force_on_pos_at_time(pos,time,wl, era) ]
  end

  def integrator_step(wl, era)
    dt = @next_time - @time
    k1 = derivative(@pos,@vel,@time,wl,era)
    k2 = derivative(@pos+dt*k1[0],@vel+dt*k1[1],@time+dt,wl,era)
    new_point = deep_copy
    new_point.pos += (k1[0]+k2[0])*dt/2
    new_point.vel += (k1[1]+k2[1])*dt/2
    new_point.time=@next_time
    new_point.force(wl,era)
    new_point
  end

  def predict_pos_vel(dt)
    [ @pos + @vel*dt,
      @vel           ]
  end

  def interpolate_pos_vel(wp, dt)
    tau = wp.time - @time
    [ @pos + @vel*dt + (1/2.0)*@acc*dt**2,
      @vel + @acc*dt                      ]
  end
end

module Integrator_rk2n_fsal                  # not partitioned

  include Integrator_force_default

  attr_accessor :acc
  attr_writer :time

  def setup_integrator
    @acc = @pos*0
```

```
  end

  def force_on_pos_at_time(pos,time,wl, era)
    era.acc(wl, pos, time)
  end

  def derivative(pos,vel,time,wl,era)
    [ vel, force_on_pos_at_time(pos,time,wl, era) ]
  end

  def integrator_step(wl, era)
    dt = @next_time - @time
    k1 = [ @vel, @acc ]
    k2 = derivative(@pos+dt*k1[0],@vel+dt*k1[1],@time+dt,wl,era)
    new_point = deep_copy
    new_point.pos += (k1[0]+k2[0])*dt/2
    new_point.vel += (k1[1]+k2[1])*dt/2
    new_point.time=@next_time
    new_point.acc = k2[1]
    new_point
  end

  def predict_pos_vel(dt)
    [ @pos + @vel*dt,
      @vel           ]
  end

  def interpolate_pos_vel(wp, dt)
    tau = wp.time - @time
    [ @pos + @vel*dt + (1/2.0)*@acc*dt**2,
      @vel + @acc*dt                      ]
  end
end

module Integrator_rk4                  # Abramowitz and Stegun's eq. 25.5.22

  include Integrator_force_default

  attr_reader :acc, :jerk
  attr_writer :time

  def setup_integrator
    @acc = @pos*0
    @jerk = @pos*0
  end
```

```
  def startup_force(wl, era)
    @acc, @jerk = era.acc_and_jerk(wl, @pos, @vel, @time)
  end

  def force_on_pos_at_time(pos, time, wl, era)
    era.acc(wl, pos, time)
  end

  def integrator_step(wl, era)
    dt = @next_time - @time
    k1 = @acc
    k2 = force_on_pos_at_time(@pos + 0.5*@vel*dt + 0.125*k1*dt**2,
                              @time + 0.5*dt, wl, era)
    k3 = force_on_pos_at_time(@pos + @vel*dt + 0.5*k2*dt**2,
                              @time + dt, wl, era)
    new_point = deep_copy
    new_point.pos += @vel*dt + (1.0/6)*(k1 + 2*k2)*dt**2
    new_point.vel += (1.0/6)*(k1 + 4*k2 + k3)*dt
    new_point.time = @next_time
    new_point.force(wl,era)
# replace the line above by the line below, to get a third-order FSAL scheme:
#    new_point.acc = k3
    new_point.estimate_jerk(self)
    new_point
  end

  def estimate_jerk(old)
    @jerk = (old.acc - @acc) / (old.time - @time)
  end

  def predict_pos_vel(dt)
    [ @pos + @vel*dt + (1/2.0)*@acc*dt**2 + (1/6.0)*@jerk*dt**3,
      @vel + @acc*dt + (1/2.0)*@jerk*dt**2                      ]
  end

  def interpolate_pos_vel(wp, dt)
    tau = wp.time - @time
    jerk = (-6*(@vel - wp.vel) - 2*(2*@acc + wp.acc)*tau)/tau**2
    snap = (12*(@vel - wp.vel) + 6*(@acc + wp.acc)*tau)/tau**3
    [ @pos + @vel*dt + (1/2.0)*@acc*dt**2 + (1/6.0)*jerk*dt**3 +
                       (1/24.0)*snap*dt**4,
      @vel + @acc*dt + (1/2.0)*jerk*dt**2 + (1/6.0)*snap*dt**3 ]
  end
end

module Integrator_rk3
```

```
include Integrator_force_default

attr_reader :acc, :jerk
attr_writer :time

def setup_integrator
  @acc = @pos*0
  @jerk = @pos*0
end

def startup_force(wl, era)
  @acc, @jerk = era.acc_and_jerk(wl, @pos, @vel, @time)
end

def force_on_pos_at_time(pos, time, wl, era)
  era.acc(wl, pos, time)
end

def integrator_step(wl, era)
  dt = @next_time - @time
  k1 = @acc
  k2 = force_on_pos_at_time(@pos + (2.0/3)*@vel*dt + (2.0/9)*k1*dt**2,
                            @time + (2.0/3)*dt, wl, era)
  new_point = deep_copy
  new_point.pos += @vel*dt + 0.25*(k1 + k2)*dt**2
  new_point.vel += 0.25*(k1 + 3*k2)*dt
  new_point.time = @next_time
  new_point.force(wl,era)
  new_point.estimate_jerk(self)
  new_point
end

def estimate_jerk(old)
  @jerk = (old.acc - @acc) / (old.time - @time)
end

def predict_pos_vel(dt)
  [ @pos + @vel*dt + (1/2.0)*@acc*dt**2 + (1/6.0)*@jerk*dt**3,
    @vel + @acc*dt + (1/2.0)*@jerk*dt**2                      ]
end

def interpolate_pos_vel(wp, dt)
  tau = wp.time - @time
  jerk = (-6*(@vel - wp.vel) - 2*(2*@acc + wp.acc)*tau)/tau**2
  snap = (12*(@vel - wp.vel) + 6*(@acc + wp.acc)*tau)/tau**3
```

```
    [ @pos + @vel*dt + (1/2.0)*@acc*dt**2 + (1/6.0)*jerk*dt**3 +
                       (1/24.0)*snap*dt**4,
      @vel + @acc*dt + (1/2.0)*jerk*dt**2 + (1/6.0)*snap*dt**3 ]
  end
end

module Integrator_cc # NOTE: ONLY WORKS NOW IF ALL BODIES USE THIS METHOD
                     # since gforce is not (yet) implemented for other methods
  include Integrator_force_default

  attr_accessor :acc
  attr_reader :jerk
  attr_writer :time

  def setup_integrator
    @acc = @pos*0
    @jerk = @pos*0
  end

  def startup_force(wl, era)
    @acc, @jerk = era.acc_and_jerk(wl, @pos, @vel, @time)
  end

  def force_on_pos_at_time(pos, time, wl, era)
    era.acc(wl, pos, time)
  end

  def gforce_on_pos_at_time(pos, acc, time, wl, era)
    era.gacc(wl, pos, acc, time)
  end

  def integrator_step(wl, era)
    dt = @next_time - @time
    k1 = @acc
    k2 = force_on_pos_at_time(@pos + 0.5*@vel*dt + (1.0/12)*k1*dt**2,
                              @time + 0.5*dt, wl, era)
    k2 += (1/48.0)*dt*dt*
          gforce_on_pos_at_time(@pos + 0.5*@vel*dt + (1.0/12)*k1*dt**2,
                                k2, @time + 0.5*dt, wl, era)
    new_point = deep_copy
    new_point.pos += @vel*dt + (1/6.0)*(k1 + 2*k2)*dt**2
    new_point.time = @next_time
    new_point.force(wl,era)
    k3 = new_point.acc
    new_point.vel += (1/6.0)*(k1 + 4*k2 + k3)*dt
    new_point.estimate_jerk(self)
```

```
    new_point
  end

  def estimate_jerk(old)
    @jerk = (old.acc - @acc) / (old.time - @time)
  end

  def predict_pos_vel(dt)
    [ @pos + @vel*dt + (1/2.0)*@acc*dt**2 + (1/6.0)*@jerk*dt**3,
      @vel + @acc*dt + (1/2.0)*@jerk*dt**2                      ]
  end

  def predict_pos_vel_acc(dt)
    [ @pos + @vel*dt + (1/2.0)*@acc*dt**2 + (1/6.0)*@jerk*dt**3,
      @vel + @acc*dt + (1/2.0)*@jerk*dt**2,
      @acc + @jerk*dt                                           ]
  end

  def interpolate_pos_vel(wp, dt)
    tau = wp.time - @time
    jerk = (-6*(@vel - wp.vel) - 2*(2*@acc + wp.acc)*tau)/tau**2
    snap = (12*(@vel - wp.vel) + 6*(@acc + wp.acc)*tau)/tau**3
    [ @pos + @vel*dt + (1/2.0)*@acc*dt**2 + (1/6.0)*jerk*dt**3 +
                       (1/24.0)*snap*dt**4,
      @vel + @acc*dt + (1/2.0)*jerk*dt**2 + (1/6.0)*snap*dt**3   ]
  end

  def interpolate_pos_vel_acc(wp, dt)
    tau = wp.time - @time
    jerk = (-6*(@vel - wp.vel) - 2*(2*@acc + wp.acc)*tau)/tau**2
    snap = (12*(@vel - wp.vel) + 6*(@acc + wp.acc)*tau)/tau**3
    [ @pos + @vel*dt + (1/2.0)*@acc*dt**2 + (1/6.0)*jerk*dt**3 +
                       (1/24.0)*snap*dt**4,
      @vel + @acc*dt + (1/2.0)*jerk*dt**2 + (1/6.0)*snap*dt**3,
      @acc + jerk*dt + (1/2.0)*snap*dt**2                       ]
  end
end

module Integrator_cco # NOTE: ONLY WORKS NOW IF ALL BODIES USE THIS METHOD
                      # since gforce is not (yet) implemented for other methods
  include Integrator_force_default

  attr_accessor :acc
  attr_reader :jerk
  attr_writer :time
```

```
def setup_integrator
  @acc = @pos*0
  @jerk = @pos*0
end

def startup_force(wl, era)
  @acc, @jerk = era.acc_and_jerk(wl, @pos, @vel, @time)
end

def force_on_pos_at_time(pos, time, wl, era)
  era.acc(wl, pos, time)
end

def gforce_on_pos_at_time(pos, acc, time, wl, era)
  era.gacc(wl, pos, acc, time)
end

def integrator_step(wl, era)
  dt = @next_time - @time
  np = deep_copy
  np.vel += (1/6.0)*np.acc*dt
  np.pos += 0.5*np.vel*dt
  np.acc = np.force_on_pos_at_time(np.pos, @time + 0.5*dt, wl, era)
  np.acc += (1/48.0)*dt*dt*np.gforce_on_pos_at_time(np.pos, np.acc,
                                                    @time + 0.5*dt, wl, era)
  np.vel += (2/3.0)*np.acc*dt
  np.pos += 0.5*np.vel*dt
  np.time = @next_time
  np.force(wl,era)
  np.vel += (1/6.0)*np.acc*dt
  np.estimate_jerk(self)
  np
end

def estimate_jerk(old)
  @jerk = (old.acc - @acc) / (old.time - @time)
end

def predict_pos_vel(dt)
  [ @pos + @vel*dt + (1/2.0)*@acc*dt**2 + (1/6.0)*@jerk*dt**3,
    @vel + @acc*dt + (1/2.0)*@jerk*dt**2                      ]
end

def predict_pos_vel_acc(dt)
  [ @pos + @vel*dt + (1/2.0)*@acc*dt**2 + (1/6.0)*@jerk*dt**3,
    @vel + @acc*dt + (1/2.0)*@jerk*dt**2,
```

```
      @acc + @jerk*dt                                          ]
  end

  def interpolate_pos_vel(wp, dt)
    tau = wp.time - @time
    jerk = (-6*(@vel - wp.vel) - 2*(2*@acc + wp.acc)*tau)/tau**2
    snap = (12*(@vel - wp.vel) + 6*(@acc + wp.acc)*tau)/tau**3
    [ @pos + @vel*dt + (1/2.0)*@acc*dt**2 + (1/6.0)*jerk*dt**3 +
                       (1/24.0)*snap*dt**4,
      @vel + @acc*dt + (1/2.0)*jerk*dt**2 + (1/6.0)*snap*dt**3   ]
  end

  def interpolate_pos_vel_acc(wp, dt)
    tau = wp.time - @time
    jerk = (-6*(@vel - wp.vel) - 2*(2*@acc + wp.acc)*tau)/tau**2
    snap = (12*(@vel - wp.vel) + 6*(@acc + wp.acc)*tau)/tau**3
    [ @pos + @vel*dt + (1/2.0)*@acc*dt**2 + (1/6.0)*jerk*dt**3 +
                       (1/24.0)*snap*dt**4,
      @vel + @acc*dt + (1/2.0)*jerk*dt**2 + (1/6.0)*snap*dt**3,
      @acc + jerk*dt + (1/2.0)*snap*dt**2                        ]
  end
end

class WorldPoint

  ACS_OUTPUT_NAME = "Body"

  MAX_TIMESTEP_INCREMENT_FACTOR = 2

  attr_accessor :pos, :vel, :next_time

  attr_reader :mass, :time,
              :nsteps, :minstep, :maxstep

  def setup(method, dt_param, time)
    extend(eval("Integrator_#{method}"))
    setup_integrator
    setup_admin(dt_param, time)
  end

  def setup_admin(dt_param, time)
    @dt_param = dt_param
    @time = @next_time = time
    @nsteps = 0
    @minstep = VERY_LARGE_NUMBER
    @maxstep = 0
```

```
end

def startup(wl, era, dt_max, init_timescale_factor)
  startup_force(wl, era)
  timescale = era.timescale(wl, self)
  startup_admin(timescale * init_timescale_factor, dt_max)
  true
end

def startup_admin(timescale, dt_max)
  dt = timescale * @dt_param
  dt = dt_max if dt > dt_max
  @next_time = @time + dt
end

def step(wl, era, dt_max)
  new_point = integrator_step(wl,era)
  timescale = era.timescale(wl, new_point)
  new_point.step_admin(@time, timescale, dt_max)
  new_point
end

def step_admin(old_time, timescale, dt_max)
  old_dt = @time - old_time
  @maxstep = old_dt if @maxstep < old_dt
  @minstep = old_dt if @minstep > old_dt
  @nsteps = @nsteps + 1
  new_dt = timescale * @dt_param
  new_dt = dt_max if new_dt > dt_max
  timestep_increment_factor = (new_dt/old_dt).abs
  if timestep_increment_factor > MAX_TIMESTEP_INCREMENT_FACTOR
    new_dt = old_dt * MAX_TIMESTEP_INCREMENT_FACTOR
  end
  @next_time = @time + new_dt
end

def predict(t)
  extrapolate(t)
end

def extrapolate(t)
  wp = deep_copy
  wp.pos, wp.vel = predict_pos_vel(t - @time)
  wp.extrapolate_admin(t)
  wp
end
```

```
def gacc_extrapolate(t)
  wp = deep_copy
  wp.pos, wp.vel, wp.acc = predict_pos_vel_acc(t - @time)
  wp.extrapolate_admin(t)
  wp
end

def extrapolate_admin(t)
  @time = t
end

def interpolate(other, t)
  wp = deep_copy
  wp.pos, wp.vel = interpolate_pos_vel(other, t-@time)
  wp.interpolate_admin(self, other, t)
  wp
end

def gacc_interpolate(other, t)
  wp = deep_copy
  wp.pos, wp.vel, wp.acc = interpolate_pos_vel_acc(other, t-@time)
  wp.interpolate_admin(self, other, t)
  wp
end

def interpolate_admin(wp1, wp2, t)
  @minstep = [wp1.minstep, wp2.minstep].min
  @maxstep = [wp1.maxstep, wp2.maxstep].max
  @nsteps = [wp1.nsteps, wp2.nsteps].max
  @time = t
end

def kinetic_energy
  0.5*@mass*@vel*@vel
end

def potential_energy(body_array)
  p = 0
  body_array.each do |b|
    unless b == self
      r = b.pos - @pos
      p += -@mass*b.mass/sqrt(r*r)
    end
  end
  p
```

```
  end
end

class WorldLine

  attr_accessor  :worldpoint

  def initialize
    @worldpoint = []
  end

  def setup(b, method, dt_param, time)
    @worldpoint[0] = b.to_worldpoint
    @worldpoint[0].setup(method, dt_param, time)
  end

  def startup(era, dt_max, init_timescale_factor)
    @worldpoint[0].startup(self, era, dt_max, init_timescale_factor)
  end

  def grow(era, dt_max)
    @worldpoint.push(@worldpoint.last.step(self, era, dt_max))
  end

  def valid_extrapolation?(time)
    unless @worldpoint.last.time <= time and time <= @worldpoint.last.next_time
      raise "#{time} not in [#{@worldpoint.last.time}, #{@worldpoint.last.next_time}]
    end
  end

  def valid_interpolation?(time)
    unless @worldpoint[0].time <= time and time <= @worldpoint.last.time
      raise "#{time} not in [#{@worldpoint[0].time}, #{@worldpoint.last.time}]"
    end
  end

  def acc(pos, t)
    p = take_snapshot_of_worldline(t)
    r = p.pos - pos
    r2 = r*r
    r3 = r2*sqrt(r2)
    p.mass*r/r3
  end

  def gacc(pos, acc, t)
    p = take_gacc_snapshot_of_worldline(t)
```

```
    r = p.pos - pos
    r2 = r*r
    r3 = r2*sqrt(r2)
#    a = p.acc - acc    this is not correct; only works for shared time steps
    a = - acc
    2*(p.mass/r3)*(a - 3*((r*a)/r2)*r)
  end

  def acc_and_jerk(pos, vel, t)
    p = take_snapshot_of_worldline(t)
    r = p.pos - pos
    r2 = r*r
    r3 = r2*sqrt(r2)
    v = p.vel - vel
    [ p.mass*r/r3 , p.mass*(v-3*(r*v/r2)*r)/r3 ]
  end

  def t_at_or_after(t)
    @worldpoint.each do |p|
      return p.time if p.time >= t
    end
    return nil
  end

  def census(t_start, t, t_overshoot)
    n = ([0]*5).to_v
    @worldpoint.each do |p|
      pt = p.time
      if p.nsteps > 0
        case
        when pt < t_start      : n[0] = p.nsteps
        when pt == t_start     : n[1] += 1
        when pt < t            : n[2] += 1
        when pt == t           : n[3] += 1
        when pt <= t_overshoot : n[4] += 1
        end
      end
    end
    n
  end

  def prune(k, t_start, t_end)
    new_worldpoint = []      # protect the original; not yet cleanly modular
    @worldpoint.each do |wp|
      if (wp.nsteps == 0 or t_start < wp.time) and wp.time <= t_end
        if wp.nsteps%k == 0
```

```
        new_worldpoint.push(wp)
      end
    end
  end
  @worldpoint = new_worldpoint
  self
end

def take_snapshot_of_worldline(time)
  if time >= @worldpoint.last.time
    valid_extrapolation?(time)
    @worldpoint.last.extrapolate(time)
  else
    valid_interpolation?(time)
    i = @worldpoint.size
    loop do
      i -= 1
      if @worldpoint[i].time <= time
        return @worldpoint[i].interpolate(@worldpoint[i+1], time)
      end
    end
  end
end

def take_gacc_snapshot_of_worldline(time)
  if time >= @worldpoint.last.time
    valid_extrapolation?(time)
    @worldpoint.last.gacc_extrapolate(time)
  else
    valid_interpolation?(time)
    i = @worldpoint.size
    loop do
      i -= 1
      if @worldpoint[i].time <= time
        return @worldpoint[i].gacc_interpolate(@worldpoint[i+1], time)
      end
    end
  end
end

def next_worldline(time)
  valid_interpolation?(time)
  i = @worldpoint.size
  loop do
    i -= 1
    if @worldpoint[i].time <= time
```

```
        wl = WorldLine.new
        wl.worldpoint = @worldpoint[i...@worldpoint.size]
        return wl
      end
    end
  end
end

class WorldEra

  attr_accessor  :start_time, :end_time, :worldline
  attr_reader :cpu_overrun_flag, :cpu_time_used_in_last_evolve_call

  def initialize
    @worldline = []
    @cpu_overrun_flag = false
  end

  def setup(ss, method, dt_param, dt_era)
    @start_time = ss.time
    @end_time = @start_time + dt_era
    ss.body.each do |b|
      wl = WorldLine.new
      wl.setup(b, method, dt_param, ss.time)
      @worldline.push(wl)
    end
  end

  def startup(dt_max, init_timescale_factor)
    list = @worldline
    while list.size > 0
      new_list = []
      list.each do |wl|
        new_list.push(wl) unless wl.startup(self,dt_max, init_timescale_factor)
      end
      list = new_list
    end
  end

  def evolve(dt_era, dt_max, cpu_time_max, shared_flag)
    @cpu_overrun_flag = false
    cpu_time = Process.times.utime
    while wordline_with_minimum_interpolation.worldpoint.last.time < @end_time
      unless shared_flag
        wordline_with_minimum_extrapolation.grow(self, dt_max)
      else
```

```
      t = wordline_with_minimum_extrapolation.worldpoint.last.next_time
      @worldline.each do |w|
        w.worldpoint.last.next_time = t
        w.grow(self, dt_era)
      end
    end
    if Process.times.utime - cpu_time > cpu_time_max
      @cpu_overrun_flag = true
      return self
    end
  end
  @cpu_time_used_in_last_evolve_call = Process.times.utime - cpu_time
  next_era(dt_era)
end

def acc(wl, pos, t)
  acc = pos*0                           # null vectors of the correct length
  @worldline.each do |w|
    acc += w.acc(pos, t) unless w == wl
  end
  acc
end

def gacc(wl, pos, acc, t)
  gacc = pos*0                          # null vectors of the correct length
  @worldline.each do |w|
    gacc += w.gacc(pos, acc, t) unless w == wl
  end
  gacc
end

def acc_and_jerk(wl, pos, vel, t)
  acc = jerk = pos*0                 # null vectors of the correct length
  @worldline.each do |w|
    unless w == wl
      da, dj = w.acc_and_jerk(pos, vel, t)
      acc += da
      jerk += dj
    end
  end
  [acc, jerk]
end

def snap_and_crackle(wl, wp)
  take_snapshot_except(wl, wp.time).get_snap_and_crackle(wp.pos, wp.vel,
                                                         wp.acc, wp.jerk)
```

```
  end

  def timescale(wl, wp)
    take_snapshot_except(wl, wp.time).collision_time_scale(wp.mass,
                                                            wp.pos, wp.vel)
  end

  def take_snapshot(time)
    take_snapshot_except(nil, time)
  end

  def take_snapshot_except(wl, time)
    ws = WorldSnapshot.new
    ws.time = time
    @worldline.each do |w|
      s = w.take_snapshot_of_worldline(time)
      ws.body.push(s) unless w == wl
    end
    ws
  end

  def report_energy
    take_snapshot(@start_time).total_energy
  end

  def write_diagnostics(t, initial_energy, unscheduled_output = false)
    STDERR.print "  < unscheduled > " if unscheduled_output
    STDERR.print "t = #{sprintf("%g", t)} "
    cen = census(t)
    STDERR.print "(after #{cen[0..2].inject{|n,dn|n+dn}}, "
    STDERR.print "#{cen[3]}, #{cen[4]} steps <,=,> t)\n"
    take_snapshot(t).write_diagnostics(initial_energy)
  end

  def wordline_with_minimum_extrapolation
    t = VERY_LARGE_NUMBER
    wl = nil
    @worldline.each do |w|
      if t > w.worldpoint.last.next_time
        t = w.worldpoint.last.next_time
        wl = w
      end
    end
    wl
  end
```

```
  def wordline_with_minimum_interpolation
    t = VERY_LARGE_NUMBER
    wl = nil
    @worldline.each do |w|
      if t > w.worldpoint.last.time
        t = w.worldpoint.last.time
        wl = w
      end
    end
    wl
  end

  def next_era(dt_era)
    e = WorldEra.new
    e.start_time = @end_time
    e.end_time = @end_time + dt_era
    @worldline.each do |wl|
      e.worldline.push(wl.next_worldline(e.start_time))
    end
    e
  end

  def census(t = @end_time)
    tmax = @worldline.map{|w| w.t_at_or_after(t)}.inject{|tt, tm| [tt,tm].max}
    @worldline.map{|w| w.census(@start_time, t, tmax)}.inject{|n, dn| n+dn}
  end

  def prune(k)
    new_worldline = []      # protect the original; not yet cleanly modular
    @worldline.each do |w|
      new_worldline.push(w.prune(k, @start_time, @end_time))
    end
    @worldline = new_worldline
    self
  end
end

module Output

  def diagnostics_and_output(c, at_startup)
    if at_startup
      t_target = @time
    else
      t_target = [@t_end, @era.end_time].min
    end
    output(c, t_target, at_startup)
```

```
    diagnostics(t_target, c.dt_dia)
  end

  def diagnostics(t_target, dt_dia)
    dia_output = false
    while @t_dia <= t_target
      @era.write_diagnostics(@t_dia, @initial_energy)
      @t_dia += dt_dia
      dia_output = true
    end
    dia_output
  end

  def unscheduled_diagnostics(dt_dia)
    t_era = @era.wordline_with_minimum_interpolation.worldpoint.last.time
    unless diagnostics(t_era, dt_dia)
      @era.write_diagnostics(t_era, @initial_energy, true)
    end
  end

  def output(c, t_target, at_startup)
    if (k = c.prune_factor) > 0
      pruned_dump(c, at_startup)
    else
      timed_output(c, t_target, at_startup)
    end
  end

  def pruned_dump(c, at_startup)
    unless at_startup
      @era.clone.prune(c.prune_factor).acs_write($stdout, false, c.precision,
                                                 c.add_indent)
    end
  end

  def timed_output(c, t_target, at_startup)
    while @t_out <= t_target
      if c.output_at_startup_flag or not at_startup
        if c.world_output_flag
          acs_write($stdout, false, c.precision, c.add_indent)
        else
          @era.take_snapshot(@t_out).acs_write($stdout, true,
                                               c.precision, c.add_indent)
        end
      end
      @t_out += c.dt_out
```

```
    end
  end
end

class World

include Output

  def World.admit(file, c)
    object = acs_read([self, WorldSnapshot], file)
    if object.class == self
      object.continue_from_world(c)
      return object
    elsif object.class == WorldSnapshot
      w = World.new
      w.setup(object, c)
      w.startup(c)
      return w
    else
      raise "#{object.class} not recognized"
    end
  end

  def continue_from_world(c)
    diagnostics_and_output(c, true)
    @t_out += c.dt_out
    @t_end += c.dt_end
    @dt_max = c.dt_era * c.dt_max_param
    @new_era = @era.next_era(c.dt_era)
    @old_era, @era = @era, @new_era
  end

  def setup(ss, c)
    @era = WorldEra.new
    @era.setup(ss, c.integration_method, c.dt_param, c.dt_era)
    @dt_max = c.dt_era * c.dt_max_param
    @time = @era.start_time
    @t_dia = @time
    @t_out = @time
    @t_end = @time + c.dt_end
  end

  def startup(c)
    @era.startup(@dt_max, c.init_timescale_factor)
    @initial_energy = @era.report_energy
    diagnostics_and_output(c, true)
```

```
  end

  def evolve(c)
    cpu_time_max = c.cpu_time_max
    while @era.start_time < @t_end
      tmp_era = @era.evolve(c.dt_era, @dt_max, cpu_time_max, c.shared_flag)
      if tmp_era.cpu_overrun_flag
        unscheduled_diagnostics(c.dt_dia)
        cpu_time_max = c.cpu_time_max
      else
        @new_era = tmp_era
        @time = @era.end_time
        if diagnostics_and_output(c, false)
          cpu_time_max = c.cpu_time_max
        else
          cpu_time_max -= @era.cpu_time_used_in_last_evolve_call
        end
        @old_era, @era = @era, @new_era
      end
    end
  end
end

class WorldSnapshot < NBody

  attr_accessor :time

  def initialize
    super
    @time = 0.0
  end

  def get_snap_and_crackle(pos, vel, acc, jerk)
    snap = crackle = pos*0                # null vectors of the correct length
    @body.each do |b|
      r = b.pos - pos
      r2 = r*r
      r3 = r2*sqrt(r2)
      v = b.vel - vel
      a = b.acc - acc
      j = b.jerk - jerk
      c1 = r*v/r2
      c2 = (v*v + r*a)/r2 + c1*c1
      c3 = (3*v*a + r*j)/r2 + c1*(3*c2 - 4*c1*c1)
      d_acc = b.mass*r/r3
      d_jerk = b.mass*v/r3 - 3*c1*d_acc
```

```
    d_snap = b.mass*a/r3 - 6*c1*d_jerk - 3*c2*d_acc
    d_crackle = b.mass*j/r3 - 9*c1*d_snap - 9*c2*d_jerk - 3*c3*d_acc
    snap += d_snap
    crackle += d_crackle
  end
  [snap, crackle]
end

def collision_time_scale(mass, pos, vel)
  time_scale_sq = VERY_LARGE_NUMBER             # square of time scale value
  @body.each do |b|
    r = b.pos - pos
    v = b.vel - vel
    r2 = r*r
    v2 = v*v + 1.0/VERY_LARGE_NUMBER          # always non-zero, for division
    estimate_sq = r2 / v2             # [distance]^2/[velocity]^2 = [time]^2
    if time_scale_sq > estimate_sq
      time_scale_sq = estimate_sq
    end
    a = (mass + b.mass)/r2
    estimate_sq = sqrt(r2)/a          # [distance]/[acceleration] = [time]^2
    if time_scale_sq > estimate_sq
      time_scale_sq = estimate_sq
    end
  end
  sqrt(time_scale_sq)                 # time scale value
end

def kinetic_energy
  e = 0
  @body.each{|b| e += b.kinetic_energy}
  e
end

def potential_energy
  e = 0
  @body.each{|b| e += b.potential_energy(@body)}
  e/2                                 # pairwise potentials were counted twice
end

def total_energy
  kinetic_energy + potential_energy
end

def write_diagnostics(initial_energy)
  e0 = initial_energy
```

```
    ek = kinetic_energy
    ep = potential_energy
    etot = ek + ep
    STDERR.print <<-END
          E_kin = #{sprintf("%.3g", ek)} ,\
     E_pot =  #{sprintf("%.3g", ep)} ,\
      E_tot = #{sprintf("%.3g", etot)}
          E_tot - E_init = #{sprintf("%.3g", etot - e0)}
          (E_tot - E_init) / E_init = #{sprintf("%.3g", (etot - e0)/e0 )}
    END
  end
end

class Body

  def to_worldpoint
    wp = WorldPoint.new
    wp.restore_contents(self)
  end
end

options_text = <<-END

  Description: Individual Time Step, Individual Integration Scheme Code
  Long description:
    This program evolves an N-body code with a fourth-order Hermite Scheme,
    using individual time steps.  Note that the program can be forced to let
    all particles share the same (variable) time step with the option -a.

    This is a test version, for the ACS data format

    (c) 2005, Piet Hut, Jun Makino; see ACS at www.artcompsi.org

    example:
    kali mkplummer.rb -n 4 -s 1 | kali #{$0} -t 1


  Short name:           -g
  Long name:            --integration_method
  Value type:           string
  Default value:        hermite
  Variable name:        integration_method
  Description:          Choice of integration method
  Long description:
    This option chooses the integration method.  The user is expected to
    provide a string with the name of the method, for example "leapfrog",
```

```
  "hermite".


Short name:           -c
Long name:            --step_size_control
Value type:           float
Default value:        0.01
Variable name:        dt_param
Description:          Determines the time step size
Long description:
  This option sets the step size control parameter dt_param << 1.  Before
  each new time step, we first calculate the time scale t_scale on which
  changes are expected to happen, such as close encounters or significant
  changes in velocity.  The new time step is then given as the product
  t_scale * dt_param << t_scale.


Short name:           -f
Long name:            --init_timescale_factor
Value type:           float
Default value:        0.01
Variable name:        init_timescale_factor
Description:          Initial timescale factor
Long description:
  This option allows the user to determine how extra small the initial
  timesteps are, for all particles.  In order to allow a safe startup
  for high-order multistep methods, all particles are forced to start
  their integration with a time scale that is significantly smaller
  than what they normally would be, by a factor "init_timescale_factor".


Short name:           -e
Long name:            --era_length
Value type:           float
Default value:        0.0078125
Variable name:        dt_era
Description:          Duration of an era
Long description:
  This option sets the time interval between begin and end of an era,
  which is the period in time that contains a bundle of world lines,
  all of which are guaranteed to extend beyond the era boundaries with
  by at least one world point in either direction.  In other words, each
  world line has an earliest world point before the beginning of the era,
  and a latest world point past the end of the era.  This guarantees
  accurate interpolation at each time within an era.
```

```
Short name:           -m
Long name:            --max_timestep_param
Value type:           float
Default value:        1
Variable name:        dt_max_param
Description:          Maximum time step (units dt_era)
Long description:
  This option sets an upper limit to the size dt of a time step,
  as the product of the duration of an era and this parameter:
  dt <= dt_max = dt_era * dt_max_param .


Short name:           -d
Long name:            --diagnostics_interval
Value type:           float
Default value:        1
Variable name:        dt_dia
Description:          Diagnostics output interval
Long description:
  The time interval between successive diagnostics output.
  The diagnostics include the kinetic and potential energy,
  and the absolute and relative drift of total energy, since
  the beginning of the integration.
      These diagnostics appear on the standard error stream.


Short name:           -o
Long name:            --output_interval
Value type:           float
Default value:        1
Variable name:        dt_out
Description:          Snapshot output interval
Long description:
  This option sets the time interval between output of a snapshot
  of the whole N-body system, which which will appear on the
  standard output channel.

  The snapshot contains the mass, position, and velocity values
  for all particles in an N-body system, in ACS format


Short name:           -y
Long name:            --pruned_dump
Value type:           int
Default value:        0
```

```
Variable name:        prune_factor
Description:          Prune Factor
Long description:
  If this option is invoked with a positive argument k = 1, then
  the full information for a particle is printed as soon as it
  makes a step.  If the prune factor is set to a value k > 1,
  the information is printed only for 1 out of every k steps.
  The output appears in ACS format on the standard output channel.
  It is guaranteed that for each particle the full information will
  be printed before the first step and after the last step.
  The resulting stream of outputs contains information for different
  particles at different times, but within each worldline, the
  world points are time ordered.

  If this option is not invoked, or if it is invoked with the default
  value k = 0, no such action will be undertaken.  This option, when
  invoked with k > 0, overrides the normal output options (a specified
  value for the normal output interval will be ignored).


Short name:           -t
Long name:            --time_period
Value type:           float
Default value:        10
Variable name:        dt_end
Print name:           t
Description:          Duration of the integration
Long description:
  This option sets the duration t of the integration, the time period
  after which the integration will halt.  If the initial snapshot is
  marked to be at time t_init, the integration will halt at time
  t_final = t_init + t.


Short name:           -u
Long name:            --cpu_time_max
Value type:           int
Default value:        60
Variable name:        cpu_time_max
Description:          Max cputime diagnost. interval
Long description:
  This option sets the maximum cpu time interval between diagnostics output,
  in seconds.


Short name:           -i
```

```
  Long name:            --init_out
  Value type:           bool
  Variable name:        output_at_startup_flag
  Description:          Output the initial snapshot
  Long description:
    If this flag is set to true, the initial snapshot will be output
    on the standard output channel, before integration is started.


  Short name:           -r
  Long name:            --world_output
  Value type:           bool
  Variable name:        world_output_flag
  Description:          World output format, instead of snapshot
  Long description:
    If this flag is set to true, each output will take the form of a
    full world dump, instead of a snapshot (the default).  Reading in
    such an world again will allow a fully accurate restart of the
    integration,  since no information is lost in the process of writing
    out and reading in, in terms of world format.


  Short name:           -a
  Long name:            --shared_timesteps
  Value type:           bool
  Variable name:        shared_flag
  Description:          All particles share the same time step
  Long description:
    If this flag is set to true, all particles will march in lock step,
    all sharing the same time step.


  END

clop = parse_command_line(options_text)

World.admit($stdin, clop).evolve(clop)
```

---

xxxx

yyyy

zzzz

Chapter 3

XXX

3.1 xxx

In `module Integrator_hermite`:

```
  def force(wl, era)

p "hermite_force" if $STARTUP
    @acc, @jerk = era.acc_and_jerk(wl, @pos, @vel, @time)
  end
```

and in `module Integrator_force_default`

```
  def startup_force(wl, era)

$STARTUP = true
p "entering startup_force"
    force(wl, era)
p "exiting startup_force"
$STARTUP = false
  end

  def force(wl, era)
p "normal_force" if $STARTUP
    @acc = era.acc(wl, @pos, @time)
  end
```

This shows that what we do here is safe:

```
<kamuy|indiv_timesteps_3> kali mkplummer.rb -n 5 -s 1 | kali world3.rb -t 1
==> Plummer's Model Builder <==
Number of particles: N = 5
pseudorandom number seed given: 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
             actual seed used: 1
==> Individual Time Step, Individual Integration Scheme Code <==
Choice of integration method: integration_method = hermite
Determines the time step size: dt_param = 0.01
Initial timescale factor: init_timescale_factor = 0.01
Duration of an era: dt_era = 0.01
Maximum time step (units dt_era): dt_max_param = 1.0
Diagnostics output interval: dt_dia = 1.0
Snapshot output interval: dt_out = 1.0
Asynchronous output interval: async_output_interval = 0
Duration of the integration: t = 1.0
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
"entering startup_force"
"hermite_force"
"exiting startup_force"
"entering startup_force"
"hermite_force"
"exiting startup_force"
"entering startup_force"
"hermite_force"
"exiting startup_force"
"entering startup_force"
"hermite_force"
"exiting startup_force"
"entering startup_force"
"hermite_force"
"exiting startup_force"
at time t = 0 (from interpolation after 0 steps to time 0):
    E_kin = 0.376 ,     E_pot =  -0.351 ,      E_tot = 0.0255
       E_tot - E_init = 0
        (E_tot - E_init) / E_init = 0
at time t = 1 (from interpolation after 2300 steps to time 1):
    E_kin = 0.295 ,     E_pot =  -0.269 ,      E_tot = 0.0255
       E_tot - E_init = 1.04e-10
        (E_tot - E_init) / E_init = 4.08e-09
ACS
  NBody
    Array body
      Body body[0]
```

```
        Vector acc
            -3.3926697902326147e-01  -3.5469282500536559e-01   3.3887816489405043e-01
        Float dt_param
             1.0000000000000000e-02
 ....
```

Chapter 4

XXX

4.1 xxx

```
|gravity> kali mkplummer.rb -n 3 -s 1 | kali nbody_set_id.rb > tmp.in
==> Plummer's Model Builder <==
Number of particles: N = 3
pseudorandom number seed given: 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
==> Takes an N-body system, and numbers all particles <==
The value of @body_id for the first particle: n = 1
Floating point precision: precision = 16
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
             actual seed used: 1
```

```
|gravity> kali world3.rb -t 0.1 -d 0.1 -o 0.1 < tmp.in > tmp.h
==> Individual Time Step, Individual Integration Scheme Code <==
Choice of integration method: integration_method = hermite
Determines the time step size: dt_param = 0.01
Initial timescale factor: init_timescale_factor = 0.01
Duration of an era: dt_era = 0.0078125
```

```
Maximum time step (units dt_era): dt_max_param = 1.0
Diagnostics output interval: dt_dia = 0.1
Snapshot output interval: dt_out = 0.1
Prune Factor: prune_factor = 0
Duration of the integration: t = 0.1
Max cputime diagnost. interval: cpu_time_max = 60
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
t = 0 (after 0, 0, 0 steps <,=,> t)
          E_kin = 0.25 ,     E_pot =  -0.5 ,      E_tot = -0.25
          E_tot - E_init = 0
          (E_tot - E_init) / E_init = -0
t = 0.1 (after 101, 0, 3 steps <,=,> t)
          E_kin = 0.303 ,     E_pot =  -0.553 ,      E_tot = -0.25
          E_tot - E_init = -1.43e-10
          (E_tot - E_init) / E_init = 5.72e-10
```

```
|gravity> kali world3.rb -t 0.1 -d 0.1 -g leapfrog -c 0.0001 -o 0.1 < tmp.in > tmp.l
==> Individual Time Step, Individual Integration Scheme Code <==
Choice of integration method: integration_method = leapfrog
Determines the time step size: dt_param = 0.0001
Initial timescale factor: init_timescale_factor = 0.01
Duration of an era: dt_era = 0.0078125
Maximum time step (units dt_era): dt_max_param = 1.0
Diagnostics output interval: dt_dia = 0.1
Snapshot output interval: dt_out = 0.1
Prune Factor: prune_factor = 0
Duration of the integration: t = 0.1
Max cputime diagnost. interval: cpu_time_max = 60
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
t = 0 (after 0, 0, 0 steps <,=,> t)
          E_kin = 0.25 ,     E_pot =  -0.5 ,      E_tot = -0.25
          E_tot - E_init = 0
          (E_tot - E_init) / E_init = -0
t = 0.1 (after 8503, 0, 3 steps <,=,> t)
          E_kin = 0.303 ,     E_pot =  -0.553 ,      E_tot = -0.25
          E_tot - E_init = 2.04e-10
          (E_tot - E_init) / E_init = -8.18e-10
```

```
|gravity> kali world3.rb -t 0.1 -d 0.1 -g multistep -o 0.1 < tmp.in > tmp.m
==> Individual Time Step, Individual Integration Scheme Code <==
Choice of integration method: integration_method = multistep
Determines the time step size: dt_param = 0.01
Initial timescale factor: init_timescale_factor = 0.01
Duration of an era: dt_era = 0.0078125
Maximum time step (units dt_era): dt_max_param = 1.0
Diagnostics output interval: dt_dia = 0.1
Snapshot output interval: dt_out = 0.1
Prune Factor: prune_factor = 0
Duration of the integration: t = 0.1
Max cputime diagnost. interval: cpu_time_max = 60
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
t = 0 (after 0, 0, 0 steps <,=,> t)
          E_kin = 0.25 ,     E_pot =  -0.5 ,      E_tot = -0.25
          E_tot - E_init = 0
          (E_tot - E_init) / E_init = -0
t = 0.1 (after 101, 0, 3 steps <,=,> t)
          E_kin = 0.303 ,     E_pot =  -0.553 ,      E_tot = -0.25
          E_tot - E_init = 5.85e-10
          (E_tot - E_init) / E_init = -2.34e-09
```

```
|gravity> kali world3.rb -t 0.1 -d 0.1 -g rk4 -o 0.1 < tmp.in > tmp.r
==> Individual Time Step, Individual Integration Scheme Code <==
Choice of integration method: integration_method = rk4
Determines the time step size: dt_param = 0.01
Initial timescale factor: init_timescale_factor = 0.01
Duration of an era: dt_era = 0.0078125
Maximum time step (units dt_era): dt_max_param = 1.0
Diagnostics output interval: dt_dia = 0.1
Snapshot output interval: dt_out = 0.1
Prune Factor: prune_factor = 0
Duration of the integration: t = 0.1
Max cputime diagnost. interval: cpu_time_max = 60
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
```

```
Incremental indentation: add_indent = 2
t = 0 (after 0, 0, 0 steps <,=,> t)
          E_kin = 0.25 ,     E_pot =  -0.5 ,      E_tot = -0.25
          E_tot - E_init = 0
          (E_tot - E_init) / E_init = -0
t = 0.1 (after 101, 0, 3 steps <,=,> t)
          E_kin = 0.303 ,     E_pot =  -0.553 ,      E_tot = -0.25
          E_tot - E_init = -1.97e-11
          (E_tot - E_init) / E_init = 7.86e-11
```

```
|gravity> cat tmp.h tmp.l | kali nbody_diff.rb
==> 6N-dimensional phase space distance between two N-body systems <==
Floating point precision: precision = 2
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
6N-dim. phase space dist. for two 3-body systems:   1.6577260953383540e-09
```

```
|gravity> cat tmp.h tmp.m | kali nbody_diff.rb
==> 6N-dimensional phase space distance between two N-body systems <==
Floating point precision: precision = 2
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
6N-dim. phase space dist. for two 3-body systems:   6.0710519740368357e-09
```

```
|gravity> cat tmp.h tmp.r | kali nbody_diff.rb
==> 6N-dimensional phase space distance between two N-body systems <==
Floating point precision: precision = 2
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
6N-dim. phase space dist. for two 3-body systems:   7.7645385253603229e-10
```

nil nil nil nil nil nil

Chapter 5

Literature References

[to be provided]