Modeling with TDD

Part I

Introduction to TDD

Carlos Cruz

What is Test Driven Development (TDD)?

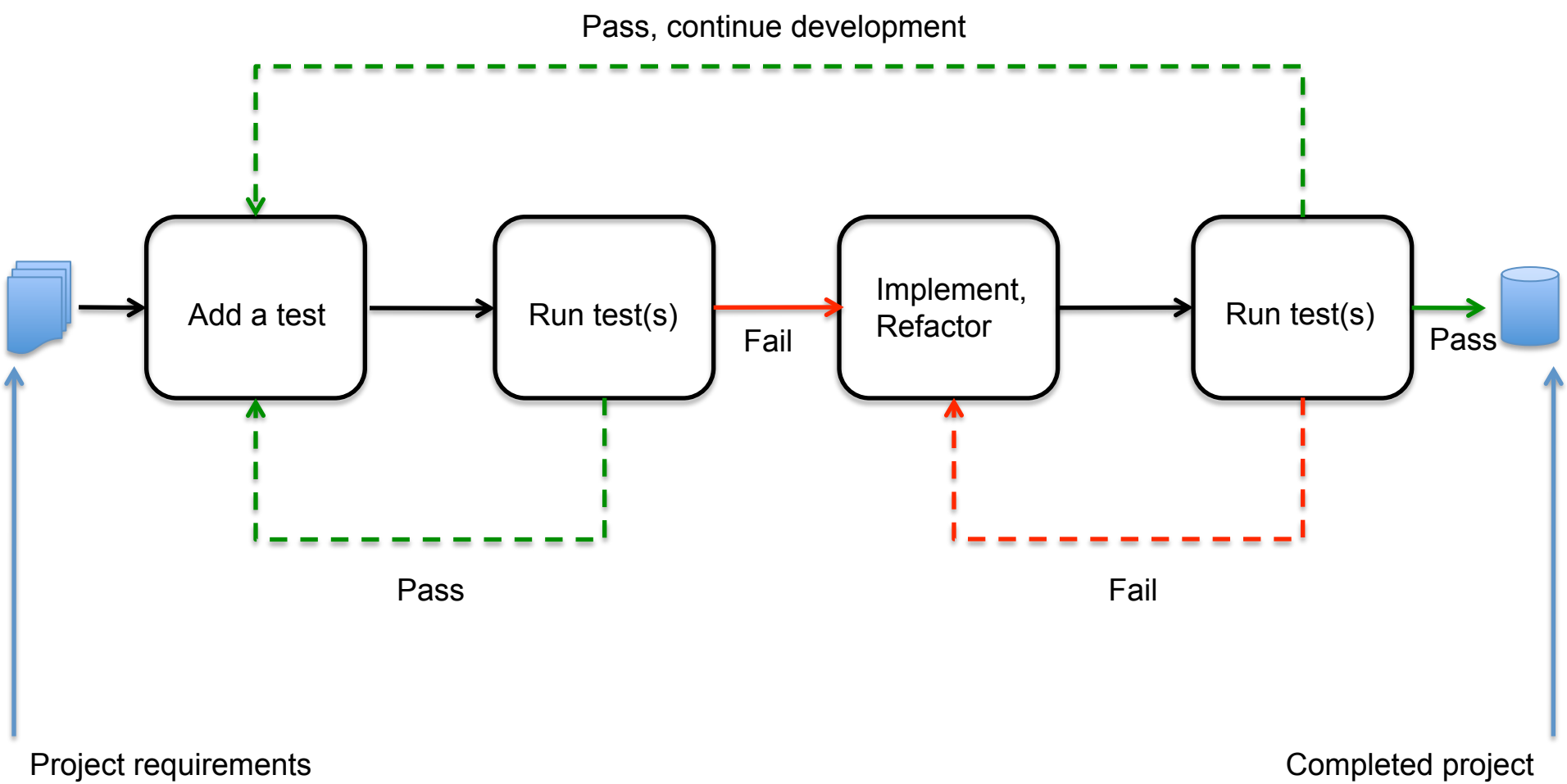
- Recall traditional software development cycle:
  - Design, code, test, debug
- Traditional testing: write a test to check that your code works
- TDD:
  - test, code, re-design

How is it done?

- figure out what you want to do
- Test: write a test to express the idea
  - It should fail
- Implement: write the code
- Test again
  - It should pass

TDD is like the "scientific method"

- figure out what you want to do — Make observation
- Test: write a test to express the idea — Hypothesis
  - It should fail
- Implement: write the code — Experiment
- Test again
  - It should pass
- …


Pass, continue development
Add a test
Run test(s)
Fail
Implement,
Refactor
Run test(s)
Pass
Pass
Fail
Project requirements
Completed project

Requirements

- An underlying assumption of TDD is that you have a unit-testing framework available to you.
- Impossible(?) to do TDD without such tools.

Testing libraries

- Perl: Test::More
- Python: PyUnit
- Java: jUnit
- C/C++: libtap
- C++: CPPUNIT
- Fortran: pFunit
- Etc…

Testing libraries

- Perl: Test::More
- Python: PyUnit
- Java: jUnit
- C/C++: libtap
- C++: CPPUNIT ⟵ Gtraj
- Fortran: pFunit ⟵ Snowfake

Why TDD?

- Maintenance of software project is (the most) expensive part.
- A significant advantage of TDD is that it enables you to take small steps when writing software.
- So, arguably, with TDD, you can:
  - prevent defects (not find them!)
  - add confidence to written code
  - enhance flexibility, maintainability and reusability of production code
- Beneficial side effect of TDD
  - achieve 100% coverage test – every single line of code is tested (ideally!)

Why TDD?

- Documentation
  - Well-written unit tests provide a working specification of your functional code – as a result unit tests effectively become a significant portion of your technical documentation

Modeling

- refers to the process of generating a model as a conceptual representation of some phenomenon.
- Earth science models
  - Numerical modeling
  - e.g. finite difference solutions of differential equations
- Keep in mind: model results must be analyzed and verified

Earth science modeling

- Earth science deals with complex models
  - interdisciplinary
  - difficult to implement
- Complex models require many data and lots of output
  - standards
- Difficult to learn and use
  - need for adequate documentation/training
- Need for good mechanism to design, build, test and maintain these complex systems.

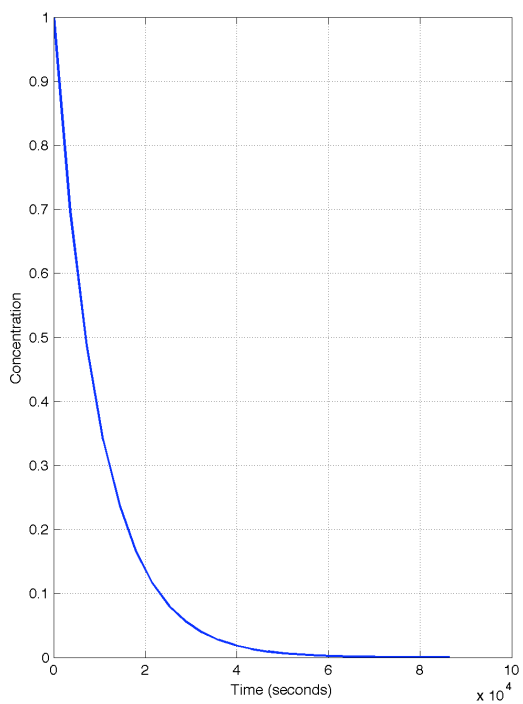
Getting started

- ModelingTDD.tar available on MG:
  - https://modelingguru.nasa.gov/clearspace/docs/DOC-1846

- Integrated sample project contains “decay model” developed with TDD
- Builds/runs on
  - Darwin(requires gfortran compiler)
  - Linux (assumes Intel compiler)

The decay model

$$\frac{dy}{dt} = -\kappa y, \qquad y(0) = y_0$$



$$y = y_0 e^{-\kappa t}$$

The decay model

$$\frac{dy}{dt} = -\kappa y, \qquad y(0) = y_0$$

$$\frac{y^{n+1} - y^n}{\Delta t} = -\kappa y^n$$

The decay model

$$\frac{dy}{dt} = -\kappa y, \qquad y(0) = y_0$$

$$\frac{y^{n+1} - y^n}{\Delta t} = -\kappa y^n$$

$$y^{n+1} = y^n - \Delta t \kappa y^n = y^n \left(1 - \Delta t \kappa\right)$$

Explicit scheme

$$y^{n+1} = y^n\left(1 - \Delta t \kappa\right)$$

- Stability
- Accuracy
  - Truncation error
  - Round off error
- Consistency
- Efficiency

$$\Delta t < \frac{1}{\kappa}$$

Use higher order methods
e.g. 4th order Runge Kutta

Other schemes

$$\frac{dy}{dt} = -\kappa y, \qquad y(0) = y_0$$

$$\frac{y^{n+1} - y^n}{\Delta t} = -\kappa y^{n+1}$$

$$y^{n+1} = \frac{y^n}{1 + \Delta t \kappa}$$

Implicit scheme

Other schemes

$$\frac{dy}{dt} = -\kappa y, \qquad y(0) = y_0$$

$$\frac{y^{n+1} - y^n}{\Delta t} = -\alpha\kappa y^{n+1} - (1-\alpha)\kappa y^n$$

$$y^{n+1} = \frac{y^n\left(1-(1-\alpha)\kappa\Delta t\right)}{1+\alpha\kappa\Delta t}$$

Hybrid scheme

```fortran
program testDriver
   use pFUnit
   implicit none

   type (t_TestSuite) :: suite
   type (t_TestResult) :: result
   type (t_Report)     :: rprt
   character(len=100) :: summary_statement

   call pFUnit_init()

! Build suite from test procedures:
   suite = TestSuite('Tests of decay model')

! Add tests here

! Run the tests and accumulate the results in "result"
   result = newTestResult(mode=MODE_USE_STDOUT)
   call Run(suite, result)
   summary_statement=Summary(result)
   print*,trim(summary_statement)

   call clean(result)
   call clean(suite)

end program testDriver
```

```fortran
module testIntegrators_mod
   use pFunit
implicit none
   private
   public testEuler

contains

   subroutine testEuler
   implicit none

! test condition that if k=0, yf=y0

end subroutine testEuler

end module testIntegrators_mod
```

```fortran
program testDriver
  use pFUnit
  use testIntegrators_mod
  implicit none

  type (t_TestSuite) :: suite
  type (t_TestResult) :: result
  type (t_Report)     :: rprt
  character(len=100) :: summary_statement

  call pFUnit_init()

! Build suite from test procedures:
  suite = TestSuite('Tests of decay model')

! Add tests here
  call add(suite, TestCase1Step('testEuler', testEuler))

! Run the tests and accumulate the results in "result"
  result = newTestResult(mode=MODE_USE_STDOUT)
  call Run(suite, result)
  summary_statement=Summary(result)
  print*,trim(summary_statement)

  call clean(result)
  call clean(suite)

end program testDriver
```

```fortran
module testIntegrators_mod
  use pFunit
  use Integrators_mod
  implicit none
  private
  public testEuler
  integer, parameter :: dp = selected_real_kind(14) ! 64 bit precision

contains

  subroutine testEuler
  implicit none

  ! y is the solution to the decay equation
  real(kind=dp), allocatable, dimension(:) :: y
  real(kind=dp) :: y0       ! initial condition
  real(kind=dp) :: timeStep ! in seconds
  real(kind=dp) :: kappa    ! in seconds^-1
  integer :: numSteps
  integer :: i

! test condition that if k=0, yf=y0

  timeStep = 3600
  kappa = 0.0
  numSteps = 86400/timeStep
  allocate(y(numSteps))

  y0 = 1.0
  call Euler(y, y0, timeStep, numSteps, kappa)
  call assertEqual(y0, y(numSteps), tolerance=1e-8)

  deallocate(y)

  end subroutine testEuler

end module testIntegrators_mod
```

```fortran
Module Integrators_mod

  implicit none
  private
  public Euler

contains

  subroutine Euler

  end subroutine Euler

end Module Integrators_mod
```

```fortran
module testIntegrators_mod
  use pFunit
  use Integrators_mod
  implicit none
  private
  public testEuler
  integer, parameter :: dp = selected_real_kind(14) ! 64 bit precision

contains

  subroutine testEuler
  implicit none

  ! y is the solution to the decay equation
  real(kind=dp), allocatable, dimension(:) :: y
  real(kind=dp) :: y0       ! initial condition
  real(kind=dp) :: timeStep ! in seconds
  real(kind=dp) :: kappa    ! in seconds^-1
  integer :: numSteps
  integer :: i

! test condition that if k=0, yf=y0

  timeStep = 3600
  kappa = 0.0
  numSteps = 86400/timeStep
  allocate(y(numSteps))

  y0 = 1.0
  call Euler(y, y0, timeStep, numSteps, kappa)
  call assertEqual(y0, y(numSteps), tolerance=1e-8)

  deallocate(y)

  end subroutine testEuler

end module testIntegrators_mod
```

```fortran
Module Integrators_mod

  implicit none
  private
  public Euler

  integer, parameter :: dp = selected_real_kind(14)

contains

  subroutine Euler(yf, y0, timeStep, numSteps, kappa)
  real(kind=dp), intent(out) :: yf(:)
  real(kind=dp), intent(in)  :: timeStep
  real(kind=dp), intent(in)  :: y0
  real(kind=dp), intent(in)  :: kappa
  integer, intent(in)        :: numSteps

  integer :: i
  real :: time
  real :: yy0

  yy0 = y0
  do i = 1,numSteps
    time = time + timeStep
    yf(i) = yy0*(1.0 - timeStep*kappa)
    yy0 = yf(i)
  end do

  end subroutine Euler

end Module Integrators_mod
```

```fortran
module testIntegrators_mod
  use pFunit
  use Integrators_mod
  implicit none
  private
  public testEuler
  integer, parameter :: dp = selected_real_kind(14) ! 64 bit precision

contains

  subroutine testEuler
  implicit none

  ! y is the solution to the decay equation
  real(kind=dp), allocatable, dimension(:) :: y
  real(kind=dp) :: y0       ! initial condition
  real(kind=dp) :: timeStep ! in seconds
  real(kind=dp) :: kappa    ! in seconds^-1
  integer :: numSteps
  integer :: i

! test condition that if k=0, yf=y0

  timeStep = 3600
  kappa = 0.0
  numSteps = 86400/timeStep
  allocate(y(numSteps))

  y0 = 1.0
  call Euler(y, y0, timeStep, numSteps, kappa)
  call assertEqual(y0, y(numSteps), tolerance=1e-8)

  deallocate(y)

  end subroutine testEuler

end module testIntegrators_mod
```

```fortran
Module Integrators_mod

  use Parameters_mod, only: kappa, dp
  implicit none
  private
  public Euler

contains

  subroutine Euler(yf, y0, timeStep, numSteps)
  real(kind=dp), intent(out) :: yf(:)
  real(kind=dp), intent(in)  :: timeStep
  real(kind=dp), intent(in)  :: y0
  integer, intent(in)        :: numSteps

  integer :: i
  real :: time
  real :: yy0

  yy0 = y0
  do i = 1,numSteps
    time = time + timeStep
    yf(i) = yy0*(1.0 - timeStep*kappa)
    yy0 = yf(i)
  end do

  end subroutine Euler

end Module Integrators_mod
```

```fortran
module testIntegrators_mod
  use pFunit
  use Integrators_mod
  use Parameters_mod, only: initialize, dp
  implicit none
  private
  public testEuler

contains

  subroutine testEuler
  implicit none

  ! y is the solution to the decay equation
  real(kind=dp), allocatable, dimension(:) :: y
  real(kind=dp) :: y0       ! initial condition
  real(kind=dp) :: timeStep ! in seconds
  real(kind=dp) :: kappa    ! in seconds^-1
  integer :: numSteps
  integer :: i

! test condition that if k=0, yf=y0

  timeStep = 3600
  kappa = 0.0
  call initialize(numSteps, timeStep, kappa)
  allocate(y(numSteps))

  y0 = 1.0
  call Euler(y, y0, timeStep, numSteps)
  call assertEqual(y0, y(numSteps), tolerance=1e-8)

  deallocate(y)

  end subroutine testEuler
end module testIntegrators_mod
```

```fortran
module Parameters_mod

  implicit none

  private
  public initialize
  public dp, kappa

  integer, parameter :: dp = selected_real_kind(14)
  real(kind=dp), parameter :: SECONDS_PER_DAY = 86400
  real(kind=dp) :: kappa ! s-1

contains

   subroutine initialize(numSteps, timeStep, k)
   integer, intent(out)      :: numSteps
   real(kind=dp), intent(in) :: timeStep
   real(kind=dp), intent(in) :: k

   numSteps = SECONDS_PER_DAY/timeStep
   kappa = k

   end subroutine initialize

end module Parameters_mod
```

```fortran
module testIntegrators_mod
   use pFunit
   use Integrators_mod
   use Parameters_mod, only: initialize, dp
   implicit none
   private
   public testEuler

contains

   subroutine testEuler
   implicit none

   ! y is the solution to the decay equation
   real(kind=dp), allocatable, dimension(:) :: y
   real(kind=dp) :: y0       ! initial condition
   real(kind=dp) :: timeStep ! in seconds
   real(kind=dp) :: kappa    ! in seconds^-1
   integer :: numSteps
   integer :: i

! test condition that if k=0, yf=y0

   timeStep = 3600
   kappa = 0.0
   call initialize(numSteps, timeStep, kappa)
   allocate(y(numSteps))

   y0 = 1.0
   call Euler(y, y0, timeStep, numSteps)
   call assertEqual(y0, y(numSteps), tolerance=1e-8)

   deallocate(y)

   end subroutine testEuler
end module testIntegrators_mod
```

To be continued…

- Feel free to go over/refactor sample code
- Try to implement 4th order Runge-Kutta
- Next time:
  - Introduction to some real problems in Earth Science
- Start discussing the shallow water model

End of part I