Astrodynamics Software and Science Enabling Toolkit (ASSET) Training

NESC Training – Flight Mechanics Tech. Discipline Team

Astrodynamics and Space Research Laboratory

Presenter: Aaron Houin
PI: Dr. Rohan Sood

The University of Alabama, Tuscaloosa AL

Astrodynamics Software and Science Enabling Toolkit: ASSET

- Funded by NASA under **ROSES-2018 C.29 Astrodynamics Tools**
- Software Goals:
  1. General-purpose optimal control/trajectory design
  2. Allow for large-scale optimization trade studies
  3. Open-source, minimize compatibility issues and dependencies
- Available on GitHub!
  - https://github.com/AlabamaASRL/asset_asrl

ASSET Overview

- Major Components:
  1. Vector Function Auto-Diff System
  2. Non-Linear Optimizer (PSIOPT)
  3. General Purpose Single/Multi-Phase Optimal Control
  4. Spacecraft Trajectory Design Tools
- Implemented in C++
  - Eigen, Intel-MKL, pybind11, fmt
- All components bind to Python for general use
  - Little to no interaction with base C++
- Extensive use of vectorization and parallelization

Public Usage

- ASSET publicly released in December 2022
- Significant usage in academia and industry
  - Multiple peer reviewed articles, conference papers, and PhD Dissertations
- NASA usage includes:
  - Exploration Systems Development Mission Directorate (ESDMD)
    - Artemis 1-5 trajectory analysis
    - Artemis 1 disposal study
    - Human Lander System ballistic lunar transfer (BLT) tool
    - Moon2Mars spacecraft autonomy study (AIMBOT)
  - Space Technology and Mission Directorate (STMD)
    - Solar Cruiser mission
    - Space Force solar sail study
  - Science Mission Directorate (SMD)
    - MoonBEAM MO proposal
    - MoonCAT SMEX proposal
    - IXPE attitude analysis
    - SWIFT HFOS study
    - MMS follow-on concept study

Continuum Space Systems Inc.

Training Overview

- Day 1: Introduction to ASSET's core functionality
  - Vector Functions - "Basic building blocks used in nearly all ASSET operations"
  - ODEs and Integrators - "Defining solution spaces and integrating the dynamics"
  - Phases - "Setting up optimization problems"
  - Optimal Control Problems (OCPs) - "Configuring complex, multi-phase optimization problems"

- Day 2: Using the "Astro Library" for quick astrodynamics modeling
  - Two-Body Problem Example
  - N-Body Problem Example
  - CR3BP Example
  - Ephemeris Pulsing Rotating Example

Training Overview

- Day 1: Introduction to ASSET's core functionality
  - Vector Functions - "Basic building blocks used in nearly all ASSET operations"
  - ODEs and Integrators - "Defining solution spaces and integrating the dynamics"
  - Phases - "Setting up optimization problems"
  - Optimal Control Problems (OCPs) - "Configuring complex, multi-phase optimization problems"

- Day 2: Using the "Astro Library" for quick astrodynamics modeling
  - Two-Body Problem Example
  - N-Body Problem Example
  - CR3BP Example
  - Ephemeris Pulsing Rotating Example

Vector Functions

Arguments (Args)

- `Arguments(n)` declares function taking n arguments and returning them
- **Entry point for defining more complicated vector functions**
  - All vector function definitions begin by declaring arguments
  - **You** must decide what arguments are and track their ordering

$$\mathbf{x} = \begin{bmatrix} x_0 \\ \vdots \\ x_{n-1} \end{bmatrix} \qquad \mathbf{f}(\mathbf{x}) = \mathbf{x}$$

```python
import numpy as np
from asset_asrl import VectorFunctions as vf
from asset_asrl.VectorFunctions import Arguments as Args

X = Args(6)
```

Arguments (Args)

- **Like all vector functions**, it is callable with real arguments
  - Calculate function value, Jacobian, adjoint-Hessian

$$\mathbf{J}(\mathbf{x}) = \frac{\partial \mathbf{f}}{\partial \mathbf{x}} \qquad \mathbf{H}(\mathbf{x}, \boldsymbol{\lambda}) = \left(\frac{\partial \mathbf{f}}{\partial \mathbf{x}^2}\right)^T \cdot \boldsymbol{\lambda}$$

```python
X = Args(6)

print(X)  ## Not numbers!!, its a vector function

## have to evaluate the functions w/ numbers to get results
xvals = np.array([0,1,2,3,4,5])
lvals = np.ones((6))

print("X = ", X(xvals) )

print("Jac X =\n", X.jacobian(xvals) )

print("Hess X =\n", X.adjointhessian(xvals,lvals) )
```

```
<asset.VectorFunctions.Arguments object at 0x0000023A3D1F6570>

X =  [0. 1. 2. 3. 4. 5.]

Jac X =
 [[1. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0.]
 [0. 0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 0. 1.]]

Hess X =
 [[0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0.]]
```

Partitioning Arguments into Elements

$$\mathbf{f}([x_0, \ldots, x_{n-1}]) = x_i$$

- `Arguments` is syntactic sugar, break them down to define useful functions
- Use bracket operator to access elements from the argument set
  - `X[i]` is a scalar function that takes n arguments and returns the ith element
- Can use `X.tolist()` to bust `Arguments` into elements in one line

```
X = Args(6)

x0 = X[0]
x1 = X[1]
x2 = X[2]
x3 = X[3]
x4 = X[4]
x5 = X[5]

## Equivalent to
x0,x1,x2,x3,x4,x5 = X.tolist()
```

```
X = Args(6)

## Scalar function "Elements"
x0 = X[0]
x5 = X[5]

#x42 =X[42]  #throws an error

xvals = np.array([0,1,2,3,4,5])

print("x0 =",x0(xvals))  # prints [0.0]
print("x5 =",x5(xvals))  # prints [5.0]
print("Jac x0 = ", x0.jacobian(xvals))
print("Jac x5 = ", x5.jacobian(xvals))
```

```
x0 = [0.]
x5 = [5.]
Jac x0 =  [1. 0. 0. 0. 0. 0.]
Jac x5 =  [0. 0. 0. 0. 0. 1.]
```

Partitioning Arguments into Vectors

$$\mathbf{f}([\mathbf{x}_0, \ldots, \mathbf{x}_n]) = \mathbf{x}_i$$

- Break into contiguous sub-vectors
  - Use `X.head(size)`,`X.tail(size)`, and `X.segment(start,size)`
  - Or use python syntax, `X[start:size+1]`

```
X = Args(6)

R = X.head(3)
R = X[0:3]     # Same as above

V = X.tail(3)
V = X[3:6]     # same as above

## .segment(start,size)
R = X.segment(0,3) # same as R above
V = X.segment(3,3) # same as V above

N = X.segment(1,4)
N = X[1:5]          #same N as above
```

```
xvals = np.array([0,1,2,3,4,5])

print("R =",R(xvals))
print("V =",V(xvals))
print("N =",N(xvals))
print("Jac N =\n",N.jacobian(xvals))
```

```
R = [0. 1. 2.]
V = [3. 4. 5.]
N = [1. 2. 3. 4.]
Jac N =
 [[0. 1. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0.]
 [0. 0. 0. 0. 1. 0.]]
```

Partitioning Arguments into Vectors

- Break into contiguous sub-vectors
  - Use `X.head(size)`,`X.tail(size)`, and `X.segment(start,size)`
  - Or use python syntax, `X[start:size+1]`
- Can use `X.tolist([(start,size)…])` for one liner

```
xvals = np.array([1,2,3,4,5,6,7,8])

X = Args(8)

R = X.head(3)
V = X.segment(3,3)
t = X[6]
u = X[7]

## Equivalent to the Above
R,V,t,u = X.tolist([ (0,3), (3,3), (6,1),(7,1) ])
```

Partitioning Vectors

- Segments and other vector functions can be partitioned into smaller segments and elements as well

```
X = Args(6)

R = X.head(3)
V = X.tail(3)

r0,r1,r2 = R.tolist()

v0 = V[0]

V12 = V.tail(2)
```

```
print("v0=",v0(xvals))
print("r0=",r0(xvals))
print("V12=",V12(xvals))
```

```
v0= [3.]
r0= [0.]
V12= [4. 5.]
```

Standard Math Operations

- Can Vector Functions using standard math operations
- **Output sizes must be consistent!!**
- LHS is a function that takes the input arguments and performs all dependent operations

```
Res =  [288. 468. 648.]
Jac Res =
 [[   0.     0.     0.    88.    16.     0.]
 [-180.   144.     0.    98.    62.     0.]
 [-216.     0.   144.   144.    36.    36.]]
```

```python
X = Args(6)
R = X.head(3)
V = X.tail(3)

## Multiply Elements/VectorFunctions and python scalars
S = R[0]*V[0]*V[1]*5.0
RtC   = R*2.0
RdC   = R/2.0

## invert scalar functions
inv0 = 1.0/V[0]

### Add/subtract vectors functions of the same size
RpV = R + V
## Sum multiple functions
Vsum = vf.sum([R,V,RdC])

## Add subtract constant vectors
RmC = R - np.array([1.0,1.0,1.0])

## Mutiply/divide vector functions by scalar functions
Rtv0 = R*V[0]
Vdr0 = V/R[0]

N = Rtv0 + Vdr0

v1pv0 = (V[1]+V[0] + 9.0)*2.0

Res = (N*v1pv0)
```

Standard Scalar Math Functions

- Can also apply standard math functions (ex: cos) to any **scalar valued function (output size is 1)**
- Held as free functions in `VectorFunctions(vf)`

```
X = Args(6)

a = vf.sin(X[0])
b = vf.cos(X[1])
c = vf.tan(X[1])

d = vf.cosh((X[1]+X[0])*X[1])

e = vf.arctan2(X[0],X[1]/3.14)

f = X[0]**2   # power operator

g = vf.abs(X[0])

h  = vf.sign(-X[1])
```

Function	Description
`vf.sin(f)`	Returns the sine of an input `Element` or `ScalarFunction`
`vf.cos(f)`	Returns the cosine of an input `Element` or `ScalarFunction`
`vf.tan(f)`	Returns the tangent of an input `Element` or `ScalarFunction`
`vf.arcsin(f)`	Returns the inverse sine of an input `Element` or `ScalarFunction`
`vf.arccos(f)`	Returns the inverse cosine of an input `Element` or `ScalarFunction`
`vf.arctan(f)`	Returns the inverse tangent of an input `Element` or `ScalarFunction`
`vf.sinh(f)`	Returns the hyperbolic sine of an input `Element` or `ScalarFunction`
`vf.cosh(f)`	Returns the hyperbolic cosine of an input `Element` or `ScalarFunction`
`vf.tanh(f)`	Returns the hyperbolic tangent of an input `Element` or `ScalarFunction`
`vf.arcsinh(f)`	Returns the inverse hyperbolic sine of an input `Element` or `ScalarFunction`
`vf.arccosh(f)`	Returns the inverse hyperbolic cosine of an input `Element` or `ScalarFunction`
`vf.arctanh(f)`	Returns the inverse hyperbolic tangent of an input `Element` or `ScalarFunction`
`vf.log(f)`	Returns the natural logarithm of an input `Element` or `ScalarFunction`
`vf.exp(f)`	Returns the exponential function of an input `Element` or `ScalarFunction`
`vf.sqrt(f)`	Returns the square root of an input `Element` or `ScalarFunction`
`vf.sign(f)`	Returns the sign(+1.0,-1.0) of an input `Element` or `ScalarFunction`
`vf.abs(f)`	Returns the absolute value an input `Element` or `ScalarFunction`

Vector Norms and Normalizations

- Apply norms and normalizations to any vector valued function
- More efficient than writing manually
- Access as member functions of the object

```
X = Args(6)

R = X.head(3)
V = X.tail(3)

r   = R.norm()
r   = vf.sqrt(R[0]**2 + R[1]**2 + R[2]**2)  # Same as above but slower

v2 =  V.squared_norm()
v2 = V[0]**2 + V[1]**2 + V[2]**2 # Same as above but slower


Vhat = V.normalized()
Vhat = V/V.norm()        # Same as above but slower

r3 =  R.cubed_norm()
r3 = R.norm()**3

Grav = - R.normalized_power3()  # R/|R|^3
Grav2 = - R/r3          # Same as above but slower
```

Function	Math Form	Description
`F.norm()`	$\vert\vec{F}\vert$	Returns the euclidean norm of `VectorFunction` or `Segment` `F`
`F.squared_norm()`	$\vert\vec{F}\vert^2$	Returns the square of the euclidean norm of `VectorFunction` or `Segment` `F`
`F.cubed_norm()`	$\vert\vec{F}\vert^3$	Returns the cube of the euclidean norm of `VectorFunction` or `Segment` `F`
`F.inverse_norm()`	$1/\vert\vec{F}\vert$	Returns the inverse of the euclidean norm of `VectorFunction` or `Segment` `F`
`F.inverse_squared_norm()`	$1/\vert\vec{F}\vert^2$	Returns the inverse square of the euclidean norm of `VectorFunction` or `Segment` `F`
`F.inverse_cubed_norm()`	$1/\vert\vec{F}\vert^3$	Returns the inverse cube of the euclidean norm of `VectorFunction` or `Segment` `F`
`F.normalized()`	$\frac{\vec{F}}{\vert\vec{F}\vert}$	Returns the normalized output of `VectorFunction` or `Segment` `F`
`F.normalized_power2()`	$\frac{\vec{F}}{\vert\vec{F}\vert^2}$	Returns the output of `VectorFunction` or `Segment` `F` divided by its euclidean norm squared.
`F.normalized_power3()`	$\frac{\vec{F}}{\vert\vec{F}\vert^3}$	Returns the output of `VectorFunction` or `Segment` `F` divided by its euclidean norm cubed.
`F.normalized_power4()`	$\frac{\vec{F}}{\vert\vec{F}\vert^4}$	Returns the output of `VectorFunction` or `Segment` `F` divided by its euclidean norm to the fourth power.
`F.normalized_power5()`	$\frac{\vec{F}}{\vert\vec{F}\vert^5}$	Returns the output of `VectorFunction` or `Segment` `F` divided by its euclidean norm to the fifth power.

Vector Products

- Take dot,cross, coefficientwise (Hadamard) products between vector valued functions
  - `f1.dot(f2), f1.cross(f2), f1.cwiseProduc(f2)`
  - `vf.dot(f1,f2), vf.cross(f1,f2), vf.cwiseProduct(f1,f2)`
- Mix vector functions and constants as needed

```
R,V,N,K = Args(14).tolist([(0,3),(3,3),(6,4),(10,4)])

C2 = np.array([1.0,1.0])
C3 = np.array([1.0,1.0,2.0])
C4 = np.array([1.0,1.0,2.0,3.0])


dRV = R.dot(V)
dRV = vf.dot(R,V)

dRC = R.dot(C3)
dRC = vf.dot(C3,R)

#dRC = R.dot(C4)  # throws ERROR

RcrossV = R.cross(V)
RcrossV = vf.cross(R,V)
RcrossC3 = vf.cross(R,C3)

RcVcNdC3 = (R.cross(V)).cross(N.head(3)).dot(C3)

#RcrossC4 = vf.cross(R,C4) # throws an error
KpN  = K.cwiseProduct(N)
NpC4 = N.cwiseProduct(C4)
```

Stacking Outputs

- Often we need to concatenate outputs: use `vf.stack(…)`
- Can stack any vector functions as well as floats and vectors

$$\text{stack}\left(\mathbf{f}_1(\mathbf{x}), \ldots, \mathbf{f}_n(\mathbf{x})\right) = \begin{bmatrix} \mathbf{f}_1(\mathbf{x}) \\ \vdots \\ \mathbf{f}_n(\mathbf{x}) \end{bmatrix}$$

```
R,V = Args(6).tolist([(0,3),(3,3)])

Rhat = R.normalized()
Nhat = R.cross(V).normalized()
That = Nhat.cross(Rhat).normalized()


RTN = vf.stack(Rhat,That,Nhat)

## Mix in floats and vectors as needed
Stuff = vf.stack(7.0, Rhat,42.0, np.array([2.71,3.14]) )
```

Matrix Operations

- Limited support for matrix operations
- Interpret output of vector function as matrix
  - `vf.ColMatrix, vf.RowMatrix`
- Supported Operations
  - Addition/Subtraction
  - Inversion/Transposition
  - Matrix-Matrix and Matrix-Vector multiply

```
R,V,U = Args(9).tolist([(0,3),(3,3),(6,3)])

## Three orthonormal basis vectors
Rhat = R.normalized()
Nhat = R.cross(V).normalized()
That = Nhat.cross(Rhat).normalized()

RTNcoeffs = vf.stack(Rhat,That,Nhat)

RTNmatC = vf.ColMatrix(RTNcoeffs,3,3)  # Interpret as col major 3x3  matrix
RTNmatR = vf.RowMatrix(RTNcoeffs,3,3)  # Interpret as row major 3x3  matrix

M2 = RTNmatC*RTNmatR # Multiply matrices together result is column major

U1 = RTNmatC*U       # Multiply on the right by a VectorFunction of size (3x1)
U2 = RTNmatR*U
U3 = M2*U

Zero = RTNmatR.inverse()*U -RTNmatC*U
Identity = RTNmatC*RTNmatC.transpose()

RTNmatC + RTNmatC
```

Conditional Operations

- Limited support for conditional operations using: `vf.ifelse`
  - Write conditions between scalar functions and constants (<,>)
  - Use condition to control output with vf.ifelse
    - Both branches must be same size
  - Combine multiple conditions with bitwise operators (|,&)
- Be wary using these inside of the optimizer
  - Not differentiable at the switch

```python
x0,x1,x2 = Args(3).tolist()

condition = x0<1.0

output_if_true = x1*2
output_if_false = x1+x2

func = vf.ifelse(condition,output_if_true,output_if_false)

print(func([0,  2,3]))  # prints [4.0]
print(func([1.5,2,3]))  # prints [5.0]

combo_condition = (x0<1.0)|(x0>x1)
func = vf.ifelse(combo_condition,output_if_true,output_if_false)

Fine = vf.ifelse(condition,vf.stack(x1,x2),vf.stack(x2,x1))
## Not the same size !!
#Error = vf.ifelse(condition,vf.stack(x1,x2),output_if_false)
```

Interpolated Data

- Use `vf.InterpTable1D` to incorporate numerical data

$$\{t_0, \dots, t_n\}, \{\mathbf{f}_0, \dots, \mathbf{f}_n\} \rightarrow \mathbf{f}(t) \in [t_0, t_n]$$

- Can be used inside of vector functions
- Use Cases:
  - Ephemeris planet positions
  - Aerodynamic coefficients

```python
ts = np.linspace(0,2*np.pi,1000)
kind = 'cubic' # or 'linear'

VecDat = np.array([ [np.sin(t),np.cos(t)] for t in ts])

Tab = vf.InterpTable1D(ts,VecDat,axis=0,kind=kind)
#Or if data is transposed
Tab = vf.InterpTable1D(ts,VecDat.T,axis=1,kind=kind)

#Or if data is a list of arrays with time included as one the elements
VecList = [ [np.sin(t), np.cos(t), t] for t in ts]
Tab = vf.InterpTable1D(VecList,tvar=2,kind=kind)

## If data is scalar
ScalDat = [np.sin(t) for t in ts]
STab =vf.InterpTable1D(ts,ScalDat,kind=kind)


print("Tab(np.pi/2.0)=",Tab(np.pi/2.0)) #prints [1,.0]
print("STab(np.pi/2.0)=",STab(np.pi/2.0)) # prints [1.0]

## Use as Vector Function
x,V,t = Args(4).tolist([(0,1),(1,2),(3,1)])

f1 = STab(t) + x  # STab(t) is an asset scalar function
f2 = Tab(t) + V   # Tab(t) is an asset vector function
```

```
Tab(np.pi/2.0)= [ 1.00000000e+00 -3.35768873e-14]
STab(np.pi/2.0)= [1.]
```

Suggested Organization

- Place implementation of vector function inside of python function
  - Pass constants in
  - Return final function out

```python
def FunctionImpl(a,b,c):
    x0,x1,x2 = Args(3).tolist()
    eq1 = x0 +a - x1
    eq2 = x2*b + x1*c
    return vf.stack(eq1,eq2)

func = FunctionImpl(1,2,3)

print(func([1,1,1]))  # prints [1,5]
```

Function Composition

- Existing Vector Functions can and should be reused inside other Vector Functions
  - Instantiate and use call operator to “evaluate” at new arguments

```python
def RTNBasis():
    R,V = Args(6).tolist([(0,3),(3,3)])
    Rhat = R.normalized()
    Nhat = R.cross(V).normalized()
    That = Nhat.cross(R).normalized()
    return vf.stack(Rhat,That,Nhat)
```

```python
def RTNTransform():

    X = Args(9)

    RV,U = X.tolist([(0,6),(6,3)])

    R,V = RV.tolist([(0,3),(3,3)])

    RTNBasisFunc = RTNBasis() # Instantiate function object

    RTNcoeffs = RTNBasisFunc(RV)  ### Call Function at new  arguments
    RTNcoeffs = RTNBasisFunc(R,V) # Same effect as original
    RTNcoeffs = RTNBasisFunc(vf.stack(R,V))

    RTNmat = vf.RowMatrix(RTNcoeffs,3,3)

    U_RTN = RTNmat*U

    return U_RTN
```

Practice: Angle Between 2 Vectors

- Write a vector function to calculate the angle between two 3D vectors

$$\mathbf{x} = \begin{bmatrix} \mathbf{v}_1 \\ \mathbf{v}_2 \end{bmatrix} \qquad f(\mathbf{x}) = \cos^{-1}(\hat{\mathbf{v}}_1 \cdot \hat{\mathbf{v}}_2)$$

Test Point:
[1,0,0,1.5,1.5,0]

Result:
0.78539816 rads

Practice: Angle Between 2 Vectors

- Write a vector function to calculate the angle between two 3D vectors

$$\mathbf{x} = \begin{bmatrix} \mathbf{v}_1 \\ \mathbf{v}_2 \end{bmatrix} \qquad f(\mathbf{x}) = \cos^{-1}(\hat{\mathbf{v}}_1 \cdot \hat{\mathbf{v}}_2)$$

Test Point:
[1,0,0,1.5,1.5,0]

Result:
0.78539816 rads

```
def AngleBetween():
    X = Args(6)
    V1hat = X.head(3).normalized()
    V2hat = X.tail(3).normalized()
    cosang = V1hat.dot(V2hat)
    return vf.arccos(cosang)
```

Practice: CR3BP EOMs

- Write a vector function to calculate equations of motion of the Circular Restricted 3-Body Problem

$$\mathbf{x} = [r_x, r_y, r_z, v_x, v_y, v_z] = [\mathbf{r}, \mathbf{v}]$$

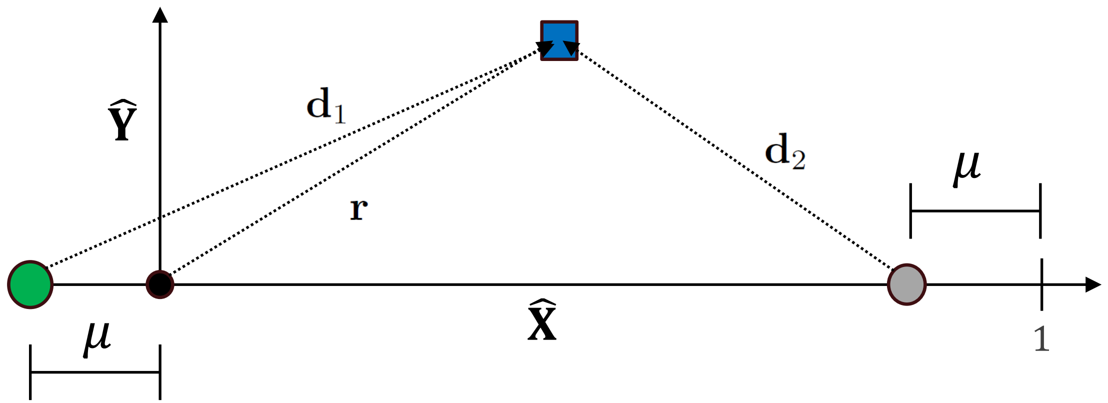
$$\mathbf{f}(\mathbf{x}) = \begin{bmatrix} \mathbf{v} \\ -\frac{1-\mu}{d_1^3}\mathbf{d}_1 - \frac{\mu}{d_2^3}\mathbf{d}_2 + \mathbf{q} \end{bmatrix}$$

where:

$$\mathbf{d}_1 = \mathbf{r} - [-\mu, 0, 0]$$

$$\mathbf{d}_2 = \mathbf{r} - [1-\mu, 0, 0]$$

$$\mathbf{q} = [2v_y + r_x, -2v_x + r_y, 0]$$



Test Point (mu = 0.01215):
[.5,.5,0,.25,.25,0]

Result:
[0.25, 0.25, 0., -0.3623761, -1.36485121, 0.]

Practice: CR3BP EOMs

- Write a vector function to calculate equations of motion of the Circular Restricted 3-Body Problem

$$\mathbf{x} = [r_x, r_y, r_z, v_x, v_y, v_z] = [\mathbf{r}, \mathbf{v}]$$

$$\mathbf{f}(\mathbf{x}) = \begin{bmatrix} \mathbf{v} \\ -\frac{1-\mu}{d_1^3}\mathbf{d}_1 - \frac{\mu}{d_2^3}\mathbf{d}_2 + \mathbf{q} \end{bmatrix}$$

where:

$$\mathbf{d}_1 = \mathbf{r} - [-\mu, 0, 0]$$

$$\mathbf{d}_2 = \mathbf{r} - [1-\mu, 0, 0]$$

$$\mathbf{q} = [2v_y + r_x, -2v_x + r_y, 0]$$

```python
def CR3BPEOMs(mu):

    X = Args(6)

    ## X=[r,v]
    R = X.head(3)
    V = X.segment(3,3)

    # P1,P2 positions
    P1loc = np.array([-mu,0,0])
    P2loc = np.array([1.0-mu,0,0])

    # Relative Position Vectors
    D1 = R-P1loc
    D2 = R-P2loc

    # Planar elements of r,v
    rx = R[0]
    ry = R[1]
    vx = V[0]
    vy = V[1]

    # Gravity terms
    G1 = (mu-1)*D1.normalized_power3()
    G2 = -mu*D2.normalized_power3()

    # Rotating Frame Terms,pad 0 to bottom of result for sum
    Q = vf.stack(2*vy+rx,-2*vx+ry,0.0)

    # Total Acceleration
    Acc = vf.sum([G1,G2,Q])

    EOMs = vf.stack(V,Acc)

    return EOMs
```

Practice: Two-Body Low-Thrust EOMS with RTN Thrust

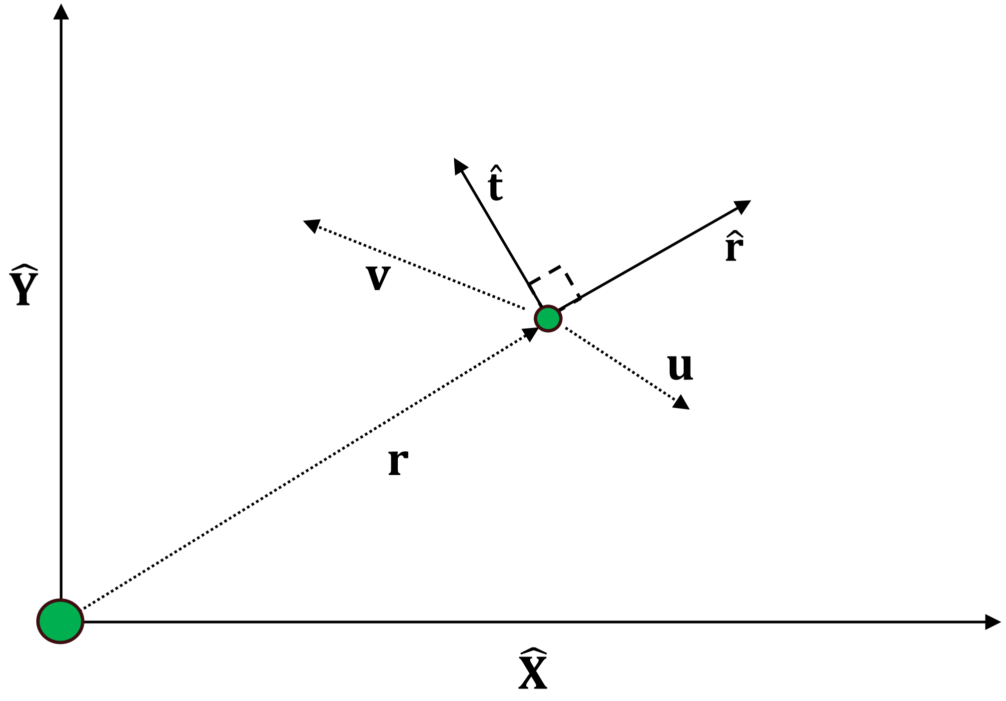
- Write a vector function to calculate equations of motion of a simple low-thrust spacecraft
  - Two-Body gravity
  - Throttle vector is in the RTN frame

$$\mathbf{x} = [r_x, r_y, r_z, v_x, v_y, v_z, u_r, u_t, u_n] = [\mathbf{r}, \mathbf{v}, \mathbf{u}]$$

$$\mathbf{f}(\mathbf{x}) = \begin{bmatrix} \mathbf{v} \\ -\frac{\mu}{|\mathbf{r}|^3}\mathbf{r} + \alpha \mathbf{M}\mathbf{u} \end{bmatrix}$$

where:

$$\mathbf{M} = [\hat{\mathbf{r}}; \hat{\mathbf{t}}; \hat{\mathbf{n}}]$$
$$\mathbf{n} = \mathbf{r} \times \mathbf{v}$$
$$\mathbf{t} = \mathbf{n} \times \mathbf{r}$$



Test (mu=1, alpha = 0.02):
[.9, 0, 0, .25, .25, 0, .7071, .7071, 0]
Result:
[0.25, 0.25, 0., -1.2204259, 0.014142, 0.]

Practice: Two-Body Low-Thrust EOMS with RTN Thrust

- Write a vector function to calculate equations of motion of a simple low-thrust spacecraft
  - Two-Body gravity
  - Throttle vector is in the RTN frame

$$\mathbf{x} = [r_x, r_y, r_z, v_x, v_y, v_z, u_r, u_t, u_n] = [\mathbf{r}, \mathbf{v}, \mathbf{u}]$$

$$\mathbf{f}(\mathbf{x}) = \begin{bmatrix} \mathbf{v} \\ -\frac{\mu}{|\mathbf{r}|^3}\mathbf{r} + \alpha\mathbf{M}\mathbf{u} \end{bmatrix}$$

where:

$$\mathbf{M} = [\hat{\mathbf{r}}; \hat{\mathbf{t}}; \hat{\mathbf{n}}]$$
$$\mathbf{n} = \mathbf{r} \times \mathbf{v}$$
$$\mathbf{t} = \mathbf{n} \times \mathbf{r}$$

Test (mu=1, alpha = 0.02):
[.9, 0, 0, .25, .25, 0, .7071, .7071, 0]
Result:
[0.25, 0.25, 0., -1.2204259, 0.014142, 0.]

```
def TBEOMs(mu,alpha):
    X = Args(9)

    R = X.head(3)
    V = X.segment(3,3)
    U = X.tail(3)

    RTNBasisFunc = RTNBasis()

    RTNCoeffs = RTNBasisFunc(R,V)

    M = vf.ColMatrix(RTNCoeffs,3,3)

    Acc = -mu*R.normalized_power3() + alpha*(M*U)

    EOMs = vf.stack(V,Acc)

    return EOMs
```

Conclusion

- See tutorial online for more in-depth details