AST415

Astronomide Sayısal Çözümleme - I

10. Nesne Yönelimli Programlama

Bu derste neler öğreneceksiniz?

- ✔ **Nesne Yönelimli Programlama Nedir?**
  - ✗ Örnek: Doğru Sınıfı
  - ✗ Sınıf Türünün Kontrolü
  - ✗ Örnek Problem 1: Nümerik Türev
  - ✗ Bir Sınıfın İşlevselliğinin Kısıtlanması
  - ✗ Örnek Problem 2: Nümerik İntegrasyon
  - ✗ Örnek Problem 3: Diferansiyel Denklem Çözümü (Opsiyonel)
- ✔ **Dersi Özetleyen Örnekler**
  - ✗ Özet Örnek 1: Programlara Veri Girişi
  - ✗ Özet Örnek 2: Tayfsal Enerji Dağılımı (TED)
  - ✗ Alıştırmalar - I
  - ✗ Alıştırmalar - II

Nesne Yönelimli Programa Nedir?

- ✓ Her ne kadar **nesne yönelimli programlama** (object oriented programming) kavramını farklı programcı ve bilgisayar bilimcileri farklı şekillerde tanımlıyor olsalar da Nesne Yönelimli Programlama'yı sınıf hiyerarşilerine dayalı programcılık paradigması olarak tanımlamak daha doğrudur. **Nesnelerle programlama'** yı da nesne yönelimli programlama kavramı altında ele alanlar olsa dahi, örneğin Python'da her “şey” bir nesne olduğu Için zaten her zaman yapılan nesnelerle programlamadır. Aradaki fark Python'un temel veri türlerinin (int, float, str, list, tuple, dict) dışında, kullanıcı tarafından tanımlanmış türler söz konusu olduğunda nesne yönelimli programlamadan bahsetmekle açıklığa kavuşturulabilir.

  Kalıt (Inheritance) ve Sıradüzen (Hierarchy) Kavramları: Nesne yönelimli programcılıkta temel yaklaşım, birbirleriyle bağlantılı sınıfları bir araya getirerek tek bir birim (aile) şeklinde programlamaktır. Bu yaklaşım, programın detaylarının bir kenara bırakılmasına, daha kolay modifiye edilmesine ve geliştirilmesine yardımcı olur.

- ✓ Sınıflardan oluşan bir aile tıpkı biyolojik bir aile gibi, “ebeveyn” sınıflar ve “çocuk” sınıflar içerir. Böyle bir aileye sınıf sıradüzeni (class hierarchy) adı verilir. Çocuk sınıflar, ebeveyn sınıflarını veri ve metotlarını miras (ya da kalıt) olarak devralır (inheritance), bu veri ve metotları değiştirebilir ve kendi veri ve metotlarını bunlara ekleyebilir. Yani, bir işlevi olan bir sınıfınız var ve yeni bir işleve ihtiyacınız varsa, bu sınıfın bir “çocuğunu” yaratıp, bu işlevi onun yapmasını sağlayabilirsiniz. Böylece hem “ebeveyn” sınıf hala elinizdedir, hem de çocuk sınıf ebeveyn sınıftaki kodlar tekrar etmediğinden daha kısadır ve kolay manipüle edilir.

- ✓ Bir program herhangi bir sınıfın çocuk ya da ebeveyn olduğuna bakmaksızın, tüm bir “aile ağacını” tek bir birim olarak değerlendirir ve bu ailedeki tüm üyelerle aynı şekilde çalışır.

- ✓ “Ebeveyn” sınıf, ana sınıf (base class) ya da üst sınıf (superclass), “çocuk” sınıflar alt sınıf (subclass) ya da türetilmiş sınıf (derived class) olarak adlandırılırlar.

Örnek: Doğru Sınıfı

- ✓ $y = c_0 + c_1 x$ şeklinde tanımlanan doğrular için tanımladığımız bir doğru sınıfını aşağıdaki şekilde kodladığımızı varsayalım (sinif_dogru.py).

```
class Dogru:
    def __init__(self, c0, c1):
        self.c0 = c0
        self.c1 = c1
    def __call__(self, x):
        return self.c0 + self.c1*x
    def tablo(self, L, R, n):
        """L <= x <= R arasinda dogru uzerinde bulunan n noktayi bir
        tablo seklinde listeler"""
        s = ''
        for x in linspace(L, R, n):
            y = self(x)
            s += '%12g %12g\n' % (x, y)
        return s
```

- ✓ Gördüğünüz gibi `__init__` metodu c0 ve c1 öz niteliklerini (attribute) başlatıyor, `__call__` metodu verilen bir x değeri için doğruda karşılık geldiği y değerini hesaplıyor, `tablo` metodu ise belirli bir aralık dahlinde, doğru üzerindeki n adet noktayı ekrana bir tablo şeklinde listeliyor.

Örnek: Doğru Sınıfı
Nesne Yönelimli Programcılık Yaklaşımı

- ✓ Python ve Nesne Yönelimli Programcılık yaklaşımını destekleyen diğer programlama dillerinde `Dogru` sınıfı için yazdığımız kodları tekrar yazmamızı önleyen özel bir kod kurgusu mutlaka bulunur. `Parabol` sınıfını `Dogru` sınıfının kodlarını kalıt (miras, inheritance) alacak şekilde şekilde kodlayabiliriz.

```
class Parabol(Dogru):
```

- ✓ Böylece `Parabol` sınıfı `Dogru` sınıfının tüm kodlarını kalıt alır, bu nedenle `Parabol` sınıfı, üst sınıfı olan `Dogru` sınıfından türetilmiş (derived) bir alt sınıftır (subclass).

- ✓ `Parabol` sınıfı `Dogru` sınıfıyla birebir aynı değildir, ona c2 verisini (attribute) ekler ve `__call__` metodu da `Dogru` sınıfınkininden farklıdır. Ancak, `tablo` metodu olduğu gibi miras alınabilir.

- ✓ Daha önce yazdığımız Parabol sınıfındaki (sinif_parabol.py) `__init__` başlatıcı metodu ve `__call__` metodunu kullanarak Parabol sınıfını nasıl bir alt sınıf olarak kurgulayacağımızı görelim (altsinif_parabol.py).

```
class Parabol(Dogru):
    def __init__(self, c0, c1, c2):
        Dogru.__init__(self, c0, c1)
        self.c2 = c2
    def __call__(self, x):
        return Dogru.__call__(self, x) + self.c2*x**2
```

```
>>> p = Parabol(1, 2, -2)
>>> p1 = p(x=2.5)
>>> print p.tablo(0, 1, 3)

          0           1
        0.5         1.5
          1           1
```

Sınıf Türünün Kontrolü

- ✓ Python'da bir i olgusunun (instance), t türüne (type) ait olup olmadığını kontrol etmek için `isinstance(i,t)` şeklinde bir fonksiyon olduğunu daha önce görmüştük.

```
>>> from altsinif_parabol import *
>>> d = Dogru(-1,1)
>>> isinstance(d,Dogru)
True
>>> isinstance(d,Parabol)
False
```

- ✓ Gördüğünüz gibi bir doğru bir parabol değildir (yani Doğru sınıfının bir olgusu olan d aynı zamanda Parabol sınıfının da bir olgusu değildir.) Tersi doğru mu görelim.

```
>>> p = Parabol(-1,0,10)
>>> isinstance(p,Parabol)
True
>>> isinstance(p,Dogru)
True
```

- ✓ Gördüğünüz gibi bir parabol olgusu aynı zamanda bir doğru olgusudur. Zira onun bütün niteliklerini taşır. Her olgunun __class__ adı verilen özel bir öz niteliği (attribute) daha vardır ve olgunun türünü tutar.

```
>>> p.__class__
<class altsinif_parabol.Parabol at 0x972b05c>
>>> p.__class__ == Parabola
True
>>> p.__class__.__name__
'Parabol'
```

- ✓ Gördüğünüz gibi `p.__class__` bir sınıf nesnesi (ya da sınıf nesnesi tanımı, class definition, class object) iken `p.__class__.__name__` bir metindir.

- ✓ Bu karşılaştırmaya dayanan bir şartlı ifade için her iki şekli de aşağıdaki gibi kullanabilrsiniz. Ancak tavsiye edilen `isinstance(p, Parabol)` fonksiyonunu kullanmanızdır.

```
if p.__class__.__name__ == 'Parabol':
        <ifadeler>
# ya da
if p.__class__ == Parabol:
        <ifadeler>
```

- ✓ Diğer taraftan `issubclass(c1, c2)` ifadesi c1'in c2'nin bir alt sınıfı olup olmadığını kontrol eder.

```
>>> issubclass(Parabol, Dogru)
True
>>> issubclass(Dogru, Parabol)
False
```

- ✓ Bir alt sınıfın üst sınıfları __class__ nesnesinin `__bases__` öz niteliğinde demet (tuple) türünde tutulur. Bu öz niteliğin tüm elemanları birer sınıf nesnesi olacağı için bu sınıfların adını öğrenmek için de sınıf nesnesinin `__name__` öz niteliğini kullanmanız gerekir.

```
>>> p.__class__.__bases__
(<class altsinif_parabol.Dogru at 0x95a7f8c>,)
>>> p.__class__.__bases__[0].__name__
'Dogru'
```

Alternatif Çözüm

- ✓ Parabol sınıfının Dogru sınıfının tüm öz nitelik ve metotlarını miras olarak devralması yerine Dogru sınıfının bir olgusunu öz nitelik olarak kullanmak da alternatif bir çözümdür.

```
class Parabol:
    def __init__(self, c0, c1, c2):
        self.dogru = Dogru(c0, c1) # c0 ve c2 Dogru sinifindan gelsin
        self.c2 = c2
    def __call__(self, x):
        return self.c2*x**2 + self.dogru(x)
```

- ✓ Hangi çözümün tercih edileceği probleme göre değişir. Eğer **”Parabol sınıfı Dogru sınıfının bir nesnesidir”** ifadesi problem açısından da doğal geliyorsa Parabol sınıfının Dogru sınıfı ile ilişki “is-a-relationship” olarak tarif edilir. Eğer **”Parabol sınıfının bir Dogru nesnesine sahip olması gerektiği”** düşünülürse o vakit bu ilişki “has-a-relationship” olarak tarif edilir. Örneğimizde “is-a-relationship” daha doğal görünmektedir; çünkü, doğru parabolün özel bir şeklidir. Ancak programcılıkta bir problemin çok sayıda çözümü olduğunu ve programcının bu çözümler arasında çalışma hızı, kod uzunluğu ve basitliği gibi kriterler bakımından en optimumunu seçmesi gerektiğini de hatırlatmak isterim.

Örnek Problem 1: Nümerik Türev

- ✓ Daha önce nümerik türev almak için basit bir yöntem görmüş ve bu yöntemin formülüne dayanan `Turev` isminde bir sınıf da yazmıştık. Kullandığımız formül (1. basamaktan geri türev) her fonksiyonun türevini almak için kullanılabilir ancak bunun için tek yöntem değildir. f'(x)'i hesaplamak için başka formüller de mevcuttur.

$$f'(x) = \frac{f(x+h) - f(x)}{h} + \mathcal{O}(h)$$

1. basamaktan ileri türev

$$f'(x) = \frac{f(x) - f(x-h)}{h} + \mathcal{O}(h)$$

1. basamaktan geri türev

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + \mathcal{O}(h^2)$$

2. basamaktan merkezi türev

$$f'(x) = \frac{4}{3}\frac{f(x+h) - f(x-h)}{2h} - \frac{1}{3}\frac{f(x+2h) - f(x-2h)}{4h} + \mathcal{O}(h^4)$$

4. basamaktan merkezi türev

$$f'(x) = \frac{3}{2}\frac{f(x+h) - f(x-h)}{2h} - \frac{3}{5}\frac{f(x+2h) - f(x-2h)}{4h} + \frac{1}{10}\frac{f(x+3h) - f(x-3h)}{6h} + \mathcal{O}(h^6),$$

6. basamaktan merkezi türev

$$f'(x) = \frac{1}{h}\left(-\frac{1}{6}f(x+2h) + f(x+h) - \frac{1}{2}f(x) - \frac{1}{3}f(x-h)\right) + \mathcal{O}(h^3)$$

3. basamaktan ileri türev

Nümerik Türev Formüllerinin Adaptasyonu

- ✓ Tüm bu formülleri Python'la programlamaya adapte etmek istediğimizde her birini bir sınıf olarak tasarlayabileceğimizi ve bu sınıfların hepsinin aynı öz nitelikleri (f ve h) başlatacağını görürüz. Dolayısıyla bir sınıf hiyerarşisi dahilinde nesne yönelimli programcılıkla soruyu çözmek doğal bir yönelimdir. Başlatıcı için tek bir kod parçası yazıp onu tüm sınıfların kullanmasını sağlayabiliriz. Turev adını verdiğimiz bir üst sınıf (superclass) yaratıp, bu sınıfın __init__ fonksiyonuyla tüm alt sınıfların ihtiyaç duyacağı iki öz niteliği (f ve h) başlatabilriz. Bu durumda diğer tüm sınıflar türevi ne şeklide hesaplıyorlarsa ona uygun bir __call__ metodu yazarak oluşturmamız yeterli olacaktır.

```
class Turev:
    def __init__(self, f, h=1E-9):
        self.f = f
        self.h = float(h)
class Ileri1(Turev):
    def __call__(self, x):
        f, h = self.f, self.h
        return (f(x+h) - f(x))/h
class Geri1(Turev):
    def __call__(self, x):
        f, h = self.f, self.h
        return (f(x) - f(x-h))/h
class Merkezi2(Turev):
    def __call__(self, x):
        f, h = self.f, self.h
        return (f(x+h) - f(x-h))/(2*h)
```

- ✓ Bu basit uyarlama sınıf hiyerarşisi içerisinde nesnel programlamanın basitliğini ve yeteneğini ortaya koymaktadır. Genel strateji, ortak kodu üst sınıf içerisinde kodlayıp, üst sınıftan farklılaşan tüm bileşenleri ise alt sınıflara taşımaktır. Bu durumda geriye kalan üç formülü de kodumuza kolaylıkla ekleyebiliriz (sinif_turev.py).

```
class Merkezi4(Turev):
    def __call__(self, x):
        f, h = self.f, self.h
        return (4./3)*(f(x+h) - f(x-h)) /(2*h) - \
               (1./3)*(f(x+2*h) - f(x-2*h))/(4*h)
class Merkezi6(Turev):
    def __call__(self, x):
        f, h = self.f, self.h
        return (3./2) *(f(x+h) - f(x-h)) /(2*h) - \
               (3./5) *(f(x+2*h) - f(x-2*h))/(4*h) + \
               (1./10)*(f(x+3*h) - f(x-3*h))/(6*h)
class Ileri3(Turev):
    def __call__(self, x):
        f, h = self.f, self.h
        return (-(1./6)*f(x+2*h) + f(x+h) - 0.5*f(x) - \
                (1./3)*f(x-h))/h
```

- Şimdi bu kodu bir test durumu için çalıştıralım ve nasıl çalıştığını anlamaya çalışalım.

```
>>> from sinif_turev import *
>>> from math import pi,sin
>>> mycos = Merkezi4(sin)
>>> print mycos(pi)
-1.00000008274
```

- `Merkezi4(sin)` ifadesi öncelikle f' ve h'i başlatmak üzere Merkezi4 sınıfı içinde başlatıcı metodu (__init__) arar. Ancak bu metod bu sınıfın içinde olmadığı ve bu sınıf `Merkezi4(Turev)` şeklinde tanımlandığı için bu kez üst sınıf olan Turev sınıfının başlatıcı fonksiyonuna yönelir (çünkü `Turev Merkezi4` sınıfının üst sınıflarını veren `Merkezi4.__bases__` listesinde yer alır). Bu şekilde f ve h başlatılır (f, sin; h gönderilmediği için varsayılan olarak 1e-9 değerlerini alır). Böylece `mycos` olgusu oluşmuş olur. `mycos(pi)` ifadesi ile bu olguya doğrudan bir değer gönderildiği için `Merkezi4` sınıfının `__call__` metodu aranır ve bulunduğu için çalıştırılır ve sonuç bu metot tarafından programa döndürülür ve `print` ifadesi de onu ekrana yazdırır. Bu şekilde üst sınıflar içinde de metotların aranması işlemine bilgisyar biliminde dinamik bağlantı (dynamic binding) adı verilir.

- ✓ Bu örnekte standart bir Python fonksiyonu yerine __call__ metodu olan bir nesne de kullanabiliriz. Kodun çalışma şekli aynıdır!

```
class Polinom2:
    def __init__(self, a, b, c):
        self.a, self.b, self.c = a, b, c
    def __call__(self, t):
        return self.a*t**2 + self.b*t + self.c
```

```
>>> f = Polinom2(1, 0, 1)
>>> dfdt = Merkezi4(f)
>>> t = 2
>>> print "f'(%g)=%g" % (t, dfdt(t))
f'(2)=4
```

- ✓ Gördüğünüz gibi `Polinom2` sınıfının bir olgusu olan f, `Polinom2` sınıfının `__call__` metodunun varlığı sayesinde bir fonksiyon gibi `Merkezi4` sınıfına argüman olarak geçirilebilmiştir. Normalde fonksiyonlar sabit isimlere sahip birer yapıyken burada istediğmiz şekilde ismini değiştirip yine de aynı fonksiyona erişebilmiş olduk. Dinamik bağlantı kavramıyla kastedilen işlev budur ve bu, Python programlama dilinde son derece doğal bir şekilde, çoğu zaman farkına bile varmadan kullandığımız bir işlevdir. Aşağıdaki örnek bu kavramı çok iyi anlatmaktadır. Zira f aslında fonksiyonun adı değilken fonksiyon da bir nesne olduğu için ve f herhangi bir nesnenin adı olabildiği için fonk1 (ya da fonk2) fonksiyonunun adının yerine kolaylıkla geçebilmektedir. f denince artık bir fonksiyondan bahsedilmektedir!

```
if girdi == fonk1:
    f = fonk1
elif girdi == fonk2:
    f = fonk2
```

Bir Sınıfın İşlevselliğinin Kısıtlanması

- ✓ Örneğimizde Dogru sınıfının bir alt sınıfı olarak kodlanan Parabol sınıfı, Dogru üst sınıfının (superclass) işlevselliğini genişletmektedir. Miras alma (inheritance) sadece bir sınıfın işlevselliğini genişletmek için değil, kısıtlamak için de kullanılabilir.

```
class Parabol:
    def __init__(self, c0, c1, c2):
        self.c0 = c0
        self.c1 = c1
        Self.c2 = c2
    def __call__(self, x):
        return self.c0 + self.c1*x + self.c2*x**2
    def tablo(self, L, R, n):
        ...
```

- ✓ şeklinde tanımlanmış bir sınıfımız olsun. Dogru sınıfını, Parabol sınıfının bir alt sınıfı olarak tanımlayabilir ve bu şekilde onun işlevselliğini kısıtlayabiliriz.

```
class Dogru(Parabol):
    def __init__(self, c0, c1):
        Parabol.__init__(self, c0, c1, 0)
```

- ✓ `__call__` ve `tablo` metotları Parabol'den miras alınarak aynı şekilde kullanılabilir. Görüldüğü üzere sınıfları hiyerarşik bir şekilde ilişkilendirmenin birden fazla yolu vardır. Bir Doğru sınıfı ile başlayıp Parabol, Kübik polinom ve giderek genel bir polinom şeklinde Doğru sınıfının işlevselliğini genişletmek yerine, genel bir Polinom sınıfı ile başlayıp, tersi yönde Prabol sınıfı onun ilk üç katsayısı hariç tüm katsayılarının sıfır olduğu bir alt sınıfı, Dogru da Parabol sınıfının bir katsayı daha sıfır yapılarak bir alt sınıfı olarak tanımlanabilir ve hiyerarşi bu şekilde de kurulabilir.

Örnek Devam: Nümerik Türev
Kodun Geliştirilmesi: Türevin Üzerindeki Hatanın Hesabı

- ✓ Sınıf hiyerarşisine dayanan bir kodlama uygulamasının ne kadar güçlü olabileceğni göstermek üzere nümerik türev hesabı yapan programımıızı bir az daha geliştirelim ve ona hesaplanan nümerik türevi gerçek değeri ile (hesaplanabildiğinde) karşılaştıran ve aradaki farkı hata değeri olarak ekrana getiren bir kod parçası ekleyelim. Yapmamız gereken başlatıcı (__init__) metoduna bir öz nitelik ve üst sınıfa hata hesabı yapan bir metod daha eklemek. Bu kodu Turev sınıfına ekleyebileceğimiz gibi onun bir alt sınıfı olan Turev2 sınıfına da ekleyebilir, farklı nümerik türev formüllerinin bu sınıftan kod miras almasını sağlayabiliriz. İkinci yola yakından bakalım:

```
class Turev2(Turev):
    def __init__(self, f, h=1E-9, dfdx_tam=None):
        Turev.__init__(self, f, h)
        self.tam = dfdx_tam
    def hata(self, x):
        if self.tam is not None:
            df_numerik = self(x)
            df_tam = self.tam(x)
        return df_tam - df_numerik
class Ileri1(Turev2):
    def __call__(self, x):
        f, h = self.f, self.h
        return (f(x+h) - f(x))/h
```

- ✓ Türevi farklı formüllerle alan diğer tüm formüllerin tanımlandığı sınıflar da tıpkı yukarıdaki örnek kod parçasında Ileri1 için olduğu gibi Turev2 sınıfından türetilmelidirler ki türevin tam değeri ile formüllere dayalı olarak hesaplanan nümerik değerleri arasındaki fark üzerinden hepsi için birer hata hesabı da yapabiliyor olalım.

```
>>> from sinif_turev2 import *
>>> mycos = Ileri1(sin, dfdx_tam=cos)
>>> print 'Numerik turevin hatasi ', mycos.hata(x=pi)
Numerik turevin hatasi 8.27403709991e-08
```

- ✓ Kodumuzun daha güzel bir testi için farklı türev formüllerinin hangi duyarlılıkta türevi hesapladığını (ne kadar hata verdiğini) ortaya koyan bir test bloğu yazalım. Test bloğumuzun çıktısı farklı h değerleri için farklı türev yöntemlerinin herhangi bir fonksiyon için hesapladığı nümerik türev değerlerini ve hatalarını içeren bir tablo şeklinde olsun (sinif_turev2.py).

```
def tablo(f, x, h_degerleri, yontemler, dfdx=None):
    # Tablo basligi yaz (h ve her metod icin sinif adi
    print ' h ',
    for yontem in yontemler:
        print '%-15s' % yontem.__name__,
    print # yeni satira gec
    for h in h_degerleri:
        print '%10.2E' % h,
        for yontem in yontemler:
            if dfdx is not None: # eger turevin tam degeri varsa hata hesapla
                d = yontem(f, h, dfdx)
                cikti = d.hata(x)
            else: # degeri yaz
                d = yontem(f, h)
                cikti = d(x)
            print '%15.8E' % cikti,
        print # yeni satir

def _test2():
    from math import exp
    def f1(x):
        return exp(-10*x)
    def df1dx(x):
        return -10*exp(-10*x)
    tablo(f1, 0, [2**(-k) for k in range(10)], \
          [Ileri1, Merkezi2, Merkezi4], df1dx)

if __name__ == '__main__':
    _test2()
```

Sadece Fonksiyonlar Kullanarak Alternatif Yaklaşım (Opsiyonel!)

- ✓ Sınıf hiyerarşisiden faydalanmaksızın aynı problemi sadece fonksiyonlardan yararlanarak da çözebilir miydik? Cevap, “neredeyse evet!” 'tir. Sınıf yapısından olduğu gibi f ve h'ı bir başltaıcı ile başlatıp, x'i türev hesabında çağırmak yerine bu kez yazacağımız her fonksiyona f, x ve h'ı birer argüman yapmak durumundayız.

```
def merkezi2_fonk(f, x, h=1.0E-9):
    return (f(x+h) - f(x-h))/(2*h)
```

- ✓ Bu durumda kodun kullanımı önemli ölçüde değişir.

```
>>> from fonksiyon_turev import *
>>> from math import cos,pi,sin
>>> mycos = merkezi2_fonk(sin,pi,1E-6)
>>> print "g'(%g)=%g (tam degeri %g)" % (pi, mycos, cos(pi))
g'(3.14159)=-1 (tam degeri -1)
```

- ✓ Gördüğünüz gibi artık mycos bir nesne değil bir sayıdır (aslında daha önceden “kullanıcı tanımlı bir nesne iken” şimdi bir “float nesnesidir” demek daha doğru!). Sınıf hiyerarşisi dahilideki çözümün güzel tarafı mycos'un argüman olarak x verildiğinde sin(x)'in x noktası için değerini veren tipik bir Python fonksiyonu gibi davranmasıydı. Bu durumda ikinci türev almak da mümkün olacaktı. Zira `mysin = merkezi2(merkezi2(sin))` ifadesi bunun için yeterli olurdu. Oysa fonksiyon olarak kodladığımızda bunu yapamayız. Ayrıca bir nesne ile çalışan diğer matematiksel algoritmaları (interpolasyon, diferansiyel denklem çözümü gibi...) da kullanamayız!

Fonksiyonel Programcılık Örnek Devam: Nümerik Türev

- ✓ Sınıf hiyerarşisinden faydalanarak yaptığımız çözümün önemli bir avantajı __call__ metodu sayesinde x için türev alırken Turev sınıfının bir d olgusuna (instance) x'i doğrudan geçirebilmemizdi. Bu durumda `d(x)` doğrudan x noktasındaki türevi hesaplayan bir ifade oluyordu. Bu fonksiyonaliteye sadece fonksiyonlar kullanılarak da ulaşılabilir şüphesiz!

```
def tureval(f, yontem, h=1.0E-9):
    h = float(h) # tam sayi bolmesinden kacmak icin float donusumu
    if yontem == 'Ileri1':
        def Ileri1(x):
            return (f(x+h) - f(x)) / h
        return Ileri1
    elif yontem == 'Geri1':
        def Geri1(x):
            return (f(x) - f(x-h)) / h
        return Geri1
```

- ✓ Bu durumda program aşağıdaki şekilde çalıştırılabilir.

```
>>> from fonksiyon_turev import *
>>> from math import cos,pi,sin
>>> mycos = tureval(sin, 'Ileri1')
>>> mysin = tureval(mycos, 'Ileri1')
>>> x = pi
>>> print mycos(x), cos(x), mysin(x), -sin(x)
-1.00000008274 -1.0 0.0 -1.22460635382e-16
```

Sadece Bir Sınıf Kullanarak Alternatif Yaklaşım (Opsiyonel!)

- ✓ Her bir yöntemi ayrı bir sınıfta tanımlamak yerine yöntemlerle ilgili temel bilgi bir tabloda tutulup; tek bir sınıfın sadece bir metodunda bu tablodan gerekli bilgiyi alıp kullanır ve türevi hesap edebilir. Herhangi bir yöntem için x_i türev alınacak noktaları, w_i ise bu noktaların ağırlıklarını göstermek üzere nümerik türev için ortak bir şema şu şekilde ifade edilebilir.

$$f'(x) \approx \sum_{i=-r}^{r} w_i f(x_i)$$

- ✓ 2r + 1 adet x_i noktası x etrafında simetrik olup, w_i ağırlıkları ise kullanılan türev yöntemine göre değişir. Örneğin aşağıda merkezi türev yöntemi için ağırlıklar verilmiştir.

$$x_i = x + ih, \quad i = -r, \ldots, r \qquad w_{-1} = -1, \quad w_0 = 0, \quad w_1 = 1$$

- ✓ r = 4 olmak üzere çeşitli yöntemler için ağırlıklar aşağıdaki tabloda verilmiştir.

	$x-4h$	$x-3h$	$x-2h$	$x-h$	x	$x+h$	$x+2h$	$x+3h$	$x+4h$
Merkezi 2	0	0	0	$-\frac{1}{2}$	0	$\frac{1}{2}$	0	0	0
Merkezi 4	0	0	$\frac{1}{12}$	$-\frac{2}{3}$	0	$\frac{2}{3}$	$-\frac{1}{12}$	0	0
Merkezi 6	0	$-\frac{1}{60}$	$\frac{3}{20}$	$-\frac{3}{4}$	0	$\frac{3}{4}$	$-\frac{3}{20}$	$\frac{1}{60}$	0
Merkezi 8	$\frac{1}{280}$	$-\frac{4}{105}$	$\frac{12}{60}$	$-\frac{4}{5}$	0	$\frac{4}{5}$	$-\frac{12}{60}$	$\frac{4}{105}$	$-\frac{1}{280}$
İleri 1	0	0	0	0	1	1	0	0	0
İleri 3	0	0	0	$-\frac{2}{6}$	$-\frac{1}{2}$	1	$-\frac{1}{6}$	0	0
Geri 1	0	0	0	-1	1	0	0	0	0

- ✓ Bu şemayı koda taşıyalım ve kodumuzu çalıştırıalım.

```
class Turev3:
    tablo = {
        ('merkezi', 2):
        [0, 0, 0, -1./2, 0, 1./2, 0, 0, 0],
        ('merkezi', 4):
        [ 0, 0, 1./12, -2./3, 0, 2./3, -1./12, 0, 0],
        ('merkezi', 6):
        [ 0, -1./60, 3./20, -3./4, 0, 3./4, -3./20, 1./60, 0],
        ('merkezi', 8):
        [ 1./280, -4./105, 12./60, -4./5, 0, 4./5, -12./60, 4./105, -1./280],
        ('ileri', 1):
        [0, 0, 0, 0, 1, 1, 0, 0, 0],
        ('ileri', 3):
        [0, 0, 0, -2./6, -1./2, 1, -1./6, 0, 0],
        ('geri', 1):
        [0, 0, 0, -1, 1, 0, 0, 0, 0],
        }
    def __init__(self, f, h=1.0E-9, yontem='merkezi', basamak=2):
        self.f, self.h, self.yontem, self.basamak = f, h, yontem, basamak
        self.agirliklar = array(Turev3.tablo[(yontem, basamak)])
    def __call__(self, x):
        f_degerleri = array([self.f(x+i*self.h) for i in range(-4,5)])
        return dot(self.agirliklar, f_degerleri)/self.h
```

```
>>> from sinif_turev3 import Turev3
>>> from numpy import array,dot
>>> from math import cos,pi,sin
>>> mycos = Turev3(sin,yontem='merkezi',basamak=4)
>>> print "sin'(pi):", mycos(pi)
sin'(pi): -1.00000008274
```

Örnek Problem 2: Nümerik İntegral

- ✓ Nümreik türev almanın pek çok yöntemi (ve dolayısı ile pek çok formülü) olduğu gibi nümerik integral almanın da pek çok yöntemi bulunmaktadır. Sınıf hiyerarşisine bir dayalı nesne yönelimli programlama çözümü bu yöntemlere de tıpkı nümerik türev problemine uygulandığı gibi uygulanabilir.

$$\int_a^b f(x)dx \approx \sum_{i=0}^{n-1} w_i f(x_i)$$

Tüm nümerik integrasyon yöntemleri w_i ağırlıkları; x_i integralin alındığı aralıktaki noktaları göstermek üzere yandaki formülle uygulanabilir.

$$x_i = a + \frac{h}{2} + ih, \quad w_i = h, \quad h = \frac{b-a}{n}, \quad i = 0, \ldots, n-1$$

Orta Nokta Yöntemi

$$x_i = a + ih, \quad h = \frac{b-a}{n-1}, \quad i = 0, \ldots, n-1$$

$$w_0 = w_{n-1} = \frac{h}{2}, \; w_i = h, \quad i = 1, \ldots, n-2$$

Yamuk (Trapezoid) Yöntemi

$$h = 2\frac{b-a}{n-1}, \quad w_0 = w_{n-1} = \frac{h}{6},$$
$$w_i = \frac{h}{3} \text{ for } i = 2, 4, \ldots, n-3,$$
$$w_i = \frac{2h}{3} \text{ for } i = 1, 3, 5, \ldots, n-2$$

Simpson Yöntemi

(x_i noktaları yamuk yöntemindekiyle aynıdır)

$$x_i = a + (i + \frac{1}{2})h - \frac{1}{\sqrt{3}}\frac{h}{2} \quad \text{for } i = 0, 2, 4, \ldots, n-2$$
$$x_i = a + (i + \frac{1}{2})h + \frac{1}{\sqrt{3}}\frac{h}{2} \quad \text{for } i = 1, 3, 5, \ldots, n-1$$

Legendre (Gauss) Yöntemi

$$h = 2(b-a)/n$$
$$w_i = h/2, \; i = 0, \ldots, n-1$$

Nümerik İntegral Formüllerinin Adaptasyonu

- ✓ Tüm bu formülleri Python'la programlamaya adapte etmek istediğimizde x_i, w_i değerlerini birer NumPy dizisine toplayıp, f(x)'i de vektörleştirilmiş (NumPy dizileri üzerinde işlem yapabilen) bir fonksiyon olarak $f(x_i)$ kodlamanın iyi bir çözüm olduğunu görebiiriz. Yine her bir formülü aşağıdaki yapıda bir sınıfın içerisine adapte etmemiz mümkündür.

```
class BirIntegralYontemi:
    def __init__(self, a, b, n):
        # [a,b] araliginda n nokta ve agirliklari hesapla
    def integral(self, f):
        s = 0
        for i in range(len(self.agirliklar)):
            s += self.agirliklar[i]*f(self.noktalar[i])
        return s
```

- ✓ Sınıf yapısını bu şekilde tasarladığımızda integral metodunun her bir integral yöntemi için ortak olacağını görmek son derece kolaydır. Dolayısıyla bu metod, üst sınıfın bir kodu olmalı ve her bir yöntem için yazılacak alt sınıflar tarafından kalıt alınabilmelidir. O nedenle bu metodu herhangi bir integral yöntemi için yazacağımız üstteki gibi bir sınıftan çıkarıp bir üst sınıfa (superclass) alalaım.

```
class Integral:
    def __init__(self, a, b, n):
        self.a, self.b, self.n = a, b, n
        self.noktalar, self.agirliklar = self.metot_olustur()
    def metot_olustur(self):
        raise NotImplementedError('%s sinifinda boyle bir kural yok' % \
        self.__class__.__name__)
    def integralal(self, f):
        s = 0
        for i in range(len(self.agirliklar)):
            s += self.agirliklar[i]*f(self.noktalar[i])
        return s
```

- ✓ Görüldüğü gibi `__init__` başlatıcı metodu (constructor) a, b ve n özniteliklerini başlatıyor. Şimdi tüm bu alt sınıflar bu kodu kalıt (inheritance) alabilir. x_i noktaları ve w_i ağırlıklarını birer dizi (ya da liste) olarak hesap etme işi `metot_olustur` metodunda gerçekleştiriliyor. Ancak bu liste ve dizinin içeriğini her bir integrasyon yöntemi ayrı bir formülle, ilgili alt sınıfta dolduracağı için bu işin gerçekleşmediği durumda `NotImplementedError` (bu metot adapte edilmedi hatası) üretiliyor. Bu `metot_olustur` metodu ne zamanki herhangi bir integrasyon yöntemi bir alt sınıfta bir nümerik integral alma yönteminin formülünü adapte ediliyor, onun tarafından devre dışı bırakılmış oluyor. Bu şekilde `metot_olustur` metodunu herhangi bir nümerik integral formülü ile integrali hesaplayan herhangi bir alt sınıfta yazmayı unutursak bu yöntemin adapte edilmediğine dair hata mesajı başlatıcı metot içerisinde üretilmiş olunur. Hata aynı zamanda yöntemin tanımlandığı alt sınıfın adı da verildiği için hangi yöntemin adapte edilemediğini göstermesi bakımından da akıllı bir şekilde yaratılyor gördüğünüz gibi...

- ✓ Bir metodun bu şekilde kod içerisinde tekrar oluşturulması ya da duruma göre daha önce aynı isimle oluşturulmuş bir metodun yerini alması bilgisayar biliminde polimorfizm olarak adlandırılır. Bu şekilde kodlanan metotlara da polimorfik metotlar denir. Genel pratik, tıpkı bizim örneğimizde olduğu gibi, polimorfik bir metodu bir üst sınıf içerisinde oluşturup, duruma göre bir alt sınıfta yeniden tanımlamaktır (overloading).

- ✓ `integralal` metodundaki kod, herhangi bir yöntemle integral alacak tüm alt sınıflar tarafından kalıt alınabilir. Bu kod NumPy dizilerinin yanı sıra herhangi bir x – w türü (liste, demet) için de çalışabilir. Ama bu kodun üst sınıfa NumPy'ın `dot` fonksiyonundan yararlanmak üzere bir de NumPy dizileri ile çalışan versiyonunu aşağıdaki şekilde yazabiliriz.

```
def vektor_ntegralal(self, f):
        return dot(self.agirliklar,f(self.noktalar[i]))
```

- ✓ Şimdi bir de herhangi bir integrasyon yöntemi (orta nokta yöntemi) ile nümerik integral hesaplayan bir alt sınıfı koldayalım Yapmamız gereken yöntemin formülünü `metot_olustur()` metoduna adapte etmek.

```
class OrtaNokta(Integral):
    def metot_olustur(self):
        a, b, n = self.a, self.b, self.n # kisa yazim icin tekrar donusum
        h = (b-a)/float(n)
        x = linspace(a + 0.5*h, b - 0.5*h, n)
        w = zeros(len(x)) + h
        return x, w
```

- ✓ Gördüğünüz gibi bu metot işlemlerini vektörize bir şekilde NumPy dizleri üzerinden gerçekleştiriyor ve sonuçta da bir NumPy dizisi döndürüyor. Diğer yöntemleri de adapte etmeden önce kodumuzun nasıl çalışacağını görmek için bir _test() bloğu yazalım.

```
def _test():
    def f(x): return x*x
    on = OrtaNokta(0, 2, 101)
    print on.integralal(f)
```

- ✓ Kod gördüğünüz gibi önce OrtaNokta sınıfından on adında bir olgu oluşturuyor. OrtaNokta sınıfının başlatıcı (__init__) metodu olmadığı için bu metot üst sınıf olan Integral 'den kalıt alınıyor. Ancak buradaki self on olgusuna karşılık geldiği dolayısı ile OrtaNokta 'nın bir olgusu olduğu için self.metot_olustur ile kastedilen OrtaNokta sınıfındaki metot_olustur sınıfıdır. Aynı isimle üst sınıf olan Integral 'de de bir sınıf bulunuyor olsa da çalışan bu OrtaNokta metodundaki metot_olustur metodu olur. En sonda çalıştırılan (çağrılan) integralal metodu ise üst sınıfta yer almaktadır ve bütün alt sınıflar tarafından kalıt alınabileceği için bir alt sınıf olan OrtaNokta'nın on olgusu tarafından da kalıt alınmıştır ve integrali hesaplar.

- ✓ Şimdi de yamuk yöntemi (trapezoid) için aynı şekilde vektorize bir alt sınıf kodlayalım.

```
class YamukYontemi(Integral):
    def metot_olustur(self):
        x = linspace(self.a, self.b, self.n)
        h = (self.b-self.a)/float(self.n - 1)
        w = zeros(len(x)) + h
        w[0] /= 2
        w[-1] /= 2
        return x, w
```

- ✓ Simpson yönteminin adaptasyonu biraz daha uzun bir kod parçası gerektirir zira formülü biraz daha komplikedir.

```
class Simpson(Integral):
    def metot_olustur(self):
        if self.n % 2 != 1:
            print 'n=%d tek olmak zorundadir, olmayinca 1 eklenir' % self.n
            self.n += 1
        x = linspace(self.a, self.b, self.n)
        h = (self.b-self.a)/float(self.n - 1)*2
        w = zeros(len(x)) + h
        w[0:self.n:2] = h*1./3
        w[1:self.n-1:2] = h*2./3
        w[0] /= 2
        w[-1] /= 2
        return x, w
```

- ✓ Şimdi de Gauss-Legendre yöntemini kodumuza adapte edelim. Bu kez dilimler kullanmak yerine biraz daha komplike bir iş olduğu için sıradan bir for döngüsünü tercih ettik. Ama aynı şeyi yine de dilimlerle yapabilirdik (deneyiniz!).

```
class GaussLegendre(Integral):
    def metot_olustur(self):
        if self.n % 2 != 0:
            print 'n=%d cift olmak zorundadir, olmayinca 1 cikarilir' % self.n
            self.n -= 1
        naralik = int(self.n/2.0)
        x = zeros(self.n)
        h = (self.b-self.a)/float(naralik)
        sqrt3 = 1./sqrt(3)
        for i in range(naralik):
            x[2*i] = self.a + (i+0.5)*h - 0.5*sqrt3*h
            x[2*i+1] = self.a + (i+0.5)*h + 0.5*sqrt3*h
        W = zeros(len(x)) + h/2.0
        return x, w
```

- ✓ Son olarak kodumuzu çalıştıralım ve farklı nümerik integrasyon metotlarının nasıl sonuç verdiğini görelim.

```
def _test2():
    def f(x): return x + 2
    a = 2; b = 3; n = 4
    for yontem in OrtaNokta, YamukYontemi, Simpson, GaussLegendre:
        y = yontem(a, b, n)
        print y.__class__.__name__, y.integralal(f)
```

```
>>>
OrtaNokta 4.5
YamukYontemi 4.5
n=4 tek olmak zorundadir, olmayinca 1 eklenir
Simpson 4.5
GaussLegendre 4.5
>>>
```

Örnek 3: Diferansiyel Denklem Çözümü (Opsiyonel!)

- ✓ Diferansiyel denklemleri temelde ikiye ayırmak mümkündür: 1) Skaler adi diferansiyel denklemler (tek bir diferansiyel denklem içerenler) 2) Adi diferansiyel denklem sistemleri

$$\frac{du}{dt} = f(u,t), \quad u(0) = u_0$$

Skaler adi diferansiyel denklem

$$\frac{du^{(i)}}{dt} = f(u^{(0)}, u^{(1)}, \ldots, u^{(n-1)}, t)$$

Adi diferansiyel denklem sistemi

$$u^{(i)}(0) = u_0^{(i)}, \quad i = 0, \ldots, n-1$$

Başlangıç koşulları

- ✓ u^0, u^1, ... , $u^{(n-1)}$ fonksiyonlarını bir vektörde başlangıç koşulları $u_0 = (u_0^{(0)}, u_0^{(1)}, \ldots, u_0^{(n-1)})$ başka bir vektörde toplamak iyi bir fikirdir. Bu iki vektörü bir adi diferansiyle denklem sistemi için NumPy dizileri ile tanımlayabileceğimiz için aslında yazacağımız kodu hem tek denklem içeren skaler denklemler hem de denklem sistemleri için kullanabiliriz.

- ✓ Çözmemiz gereken denklem sistemlerine bir örnek bir yayın ucuna asılı kütle ile oluşturulan bir sistemi tanımlayan denklemlerdir.

$$mu'' + \beta u' + ku = F(t), \quad u(0) = u_0, \; u'(0) = 0$$

$$u^{(0)}(t) = u(t), \quad u^{(1)}(t) = u'(t)$$

olmak üzere bu ikinci derece denklemin çözümü birinci dereceden iki fonksiyonla verilebilir. Buradaki bilinmeyenler $u^{(0)}(t) = u(t)$ (konum) ve $u^{(1)}(t) = u'(t)$ (hız) 'dır.

- ✓ Bu iki bilinmeyen aşağıdaki şekilde ifade edilebilir.

$$\frac{d}{dt}u^{(0)}(t) = u^{(1)}(t),$$
$$\frac{d}{dt}u^{(1)}(t) = m^{-1}(F(t) - \beta u^{(1)} - k u^{(0)})$$

- ✓ Bu tür sistemleri u'(t) = f(u,t) şeklinde ifade etmek yaygın bir pratiktir. Bu durumda u bir vektör olduğu için, f de bir vektör olur.

$$u(t) = (u^{(0)}(t), u^{(1)}(t))$$
$$f(t,u) = (u^{(1)}, m^{-1}(F(t) - \beta u^{(1)} - k u^{(0)}))$$

Çözüm İçin Nümerik Yöntemler

- ✓ Diferansiyel denklem sistemlerinin çözümü için önerilen yöntemler u fonksiyonuna t_k (k = 1, 2, ...) eşit aralıklı (t_k = k Δt) zaman dilimleri için u_k yaklaştırmasını hesaplarlar.

$$u_{k+1} = u_k + \Delta t\, f(u_k, t_k)$$ İleri Euler Yöntemi

$$u_{k+1} = u_{k-1} + 2\Delta t f(u_k, t_k)$$ Orta Nokta Yöntemi

$$K_1 = \Delta t\, f(u_k, t_k),$$
$$K_2 = \Delta t\, f(u_k + \frac{1}{2}K_1, t_k + \frac{1}{2}\Delta t)$$

olmak üzere $u_{k+1} = u_k + K_2$ İkinci Dereceden Runge – Kutta Yöntemi

$$K_1 = \Delta t\, f(u_k, t_k),$$
$$K_2 = \Delta t\, f(u_k + \frac{1}{2}K_1, t_k + \frac{1}{2}\Delta t),$$
$$K_3 = \Delta t\, f(u_k + \frac{1}{2}K_2, t_k + \frac{1}{2}\Delta t),$$
$$K_4 = \Delta t\, f(u_k + K3, t_k + \Delta t)\,.$$

olmak üzere $u_{k+1} = u_k + \frac{1}{6}(K_1 + 2K_2 + 2K_3 + K_4)$ Dördüncü Dereceden Runge – Kutta Yöntemi

$$u_{k+1} = u_k + \Delta t\, f(u_{k+1}, t_{k+1})$$ Geri Euler Yöntemi

Nümerik Çözüm Formüllerinin Adaptasyonu

- ✓ *Üst sınıf (superclass):* Diferansiye denklemlerin çözümü için üst sınıf olarak `DiferansiyelDenklemCozucu` üst sınıfını öncelikli olarak kodlayalım. Bu sınıf diğer sınıfların ihtiyaç duyacağı tüm kodu sağlamalı, alt sınıflar bu sınıftaki kodları kalıt alabilmelidir. Bunun için 1) t anı için u(t) çözümünü tutmalı, 2) karşılık gelen t anını tumalı, 3) f(u,t) fonksiyonunu çağrılabilir bir Python nesnesi olarak tutumalı 4) dt özniteliğinde Δt zaman aralıklarını tutmalı 5) Hangi zaman aralığında olduğunun belirlenmesi için aralık sayısı k'yi k isimlli bir öznitelikte tutmalı, 6) u_0 başlangıç koşulu başlatmalı, 7) Aralıktaki tüm zaman basamakları için bir döngü çalıştırmalıdır.

- ✓ İyi bir çözüm için döngüyü kurmak üzere bir `dongu` metodu ve çözümü formüle uygun olarak bir adım ilerletmek içinde bir `ileri` metodu koyacağız. Ancak bu ikinci metod şimdilik boş olacak, zira bu metodu her bir çözüm yöntemi ayrı bir formülle tekrar kuracak.

- ✓ Üst sınıfa ilişkin tüm kod bu derse ilişkin diğer örneklerle birlikte sinif_diferansiyeldenklem.py dosyasında verilmiştir. İleri Euler metodu ile çözüm için gereken alt sınıfı da aşağıda örnek olarak görüyoruz.

```
class IleriEuler(DiferansiyelDenklemCozucu):
    def ileri(self):
        u, dt, f, k, t = self.u, self.dt, self.f, self.k, self.t[-1]
        uyeni = u[k] + dt*f(u[k], t)
    return uyeni
```

- ✓ ! Önemli Not: Yukarıdaki gibi özellikle formülleri basit gösterebilmek ve kodun okunurluğunu arttırmak üzere self 'in özniteliklerini yerel değişkenlere atadığımız vakit, bunun sadece özniteliklerin değerlerini okumak için yapıldığını unutmayınız. Bu şekilde özniteliklerini değerlerini değiştirmek mümkün değildir. Örneğin k += 1, sadece yerel k'nın değerini değiştirir; sınıf boyunca self 'in özniteliği vasfıyla kullanabileceğiniz k'yı self.k += 1 ile arttırabilirsiniz!

- ✓ Diğer tüm nümerik çözüm yöntemlerinin adaptasyonu sinif_diferansiyeldenklem.py dosyasında bulunabilir.

Diferansiyel Denklem Çözümü Uygulamalar
1. u' = u

- ✓ Kodumuzu bazı diferansiyel denklemler için test etmeye en basit diferansiyel denklem uygulamasıyla başlayalım. Bu denklemi İleri Euler yöntemi ile çözmek üzere yazacağımız kod temelde aşağıdaki gibi olacaktır.

```
from sinif_diferansiyeldenklem import *
from matplotlib import pyplot as plt
from numpy import linspace
def f(u, t):
    return u

T = 3
N = 100
yontem = IleriEuler(f)
yontem.baslangic_kosulu(1.0)
U, t = yontem.cozum(linspace(0,T,N+1))
plt.plot(t,u)
plt.plot(show)
```

- ✓ Runge-Kutta yönteminin (4. basamak) zamandaki artışın büyük değerleri için (büyük Δt, küçük N) dahi ne kadar etkin bir yöntem olduğunu kolaylıkla gösterebiliriz. Kodun tamamını (biraz daha komplike haliyle) uyg1_exp.py dosyasında bulabilirsiniz.

```
N = 10 # kucuk N buyuk adim araligi demektir!
for cozum_yontemi in IleriEuler, RungeKutta4:
    yontem = cozum_yontemi(f)
    yontem.baslangic_kosulu(1.0)
    u, t = yontem.cozum(linspace(0,T,N+1))
    plt.plot(t, u)
    plt.legend('%s' % cozum_yontemi.__name__)
    plt.hold('on')
plt.show()
```

Diferansiyel Denklem Çözümü Uygulamalar
Yatay Atış

- ✓ Kodumuzu bu kez yatay atış problemini çözmek üzere kullanalım. Yatay atış problemi aşağıdaki denklemlerle anlatılabilir. (x,y) cismin sırasıyla yatay ve düşey konumunu, g yer çekimi ivmesini göstermektedir.

$$\frac{d^2x}{dt^2} = 0 \qquad \frac{d^2y}{dt^2} = -g$$

- ✓ Bu ikinci dereceden diferansiyel denklemleri, birinci derece diferansiyel denklemlere dönüştürerek yazabiliriz. Ancak bu kez 4 denkleme ihtiyacımız olur.

$$\frac{dx}{dt} = v_x \qquad \frac{dv_x}{dt} = 0 \qquad \frac{dy}{dt} = v_y \qquad \frac{dv_y}{dt} = -g$$

- ✓ Başlangıç koşulları da aşağıdaki gibidir. V_0 cismin ilk hızı, θ ise atış sırasında yatayla yapılan açıdır.

$$x(0) = 0 \qquad v_x(0) = v_0 \cos\theta \qquad y(0) = y_0 \qquad v_y(0) = v_0 \sin\theta$$

- ✓ Diferansiyel denklemlerimizin sağ tarafını döndüren bir fonksiyon aşağıdaki gibi yazılabilir.

```
def f(u, t):
    x, vx, y, vy = u
    g = 9.81
    return [vx, 0, vy, -g]
```

- ✓ Problemi çözmek için yazacağımız ana kod aşağıdaki gibi olabilir.

```
from math import pi
from numpy import linspace,cos,sin
from sinif_diferansiyeldenklem import *
from matplotlib import pyplot as plt
v0 = 5
theta = 80*pi/180
u0 = [0, v0*cos(theta), 0, v0*sin(theta)]
T = 1.2
N = 120 # dt = 0.01
t_noktalari = linspace(0,T,N)
yontem = IleriEuler(f)
yontem.baslangic_kosulu(0.)
u, t = yontem.cozum(t_noktalari)
```

- ✓ Problemin çözülmesiyle her bir elemanı bir dizi (x, vx, y, vy) olan 4 elemanlı bir dizi elde edilir. Örneğin zamanla x'in (yatay konum) nasıl değiştiğini öğrenmek istiyorsanız x dizisini bu diziden çekmeniz gerekir.

```
x_degerleri = u[:,0]
plt.plot(t, x_degerleri)
```

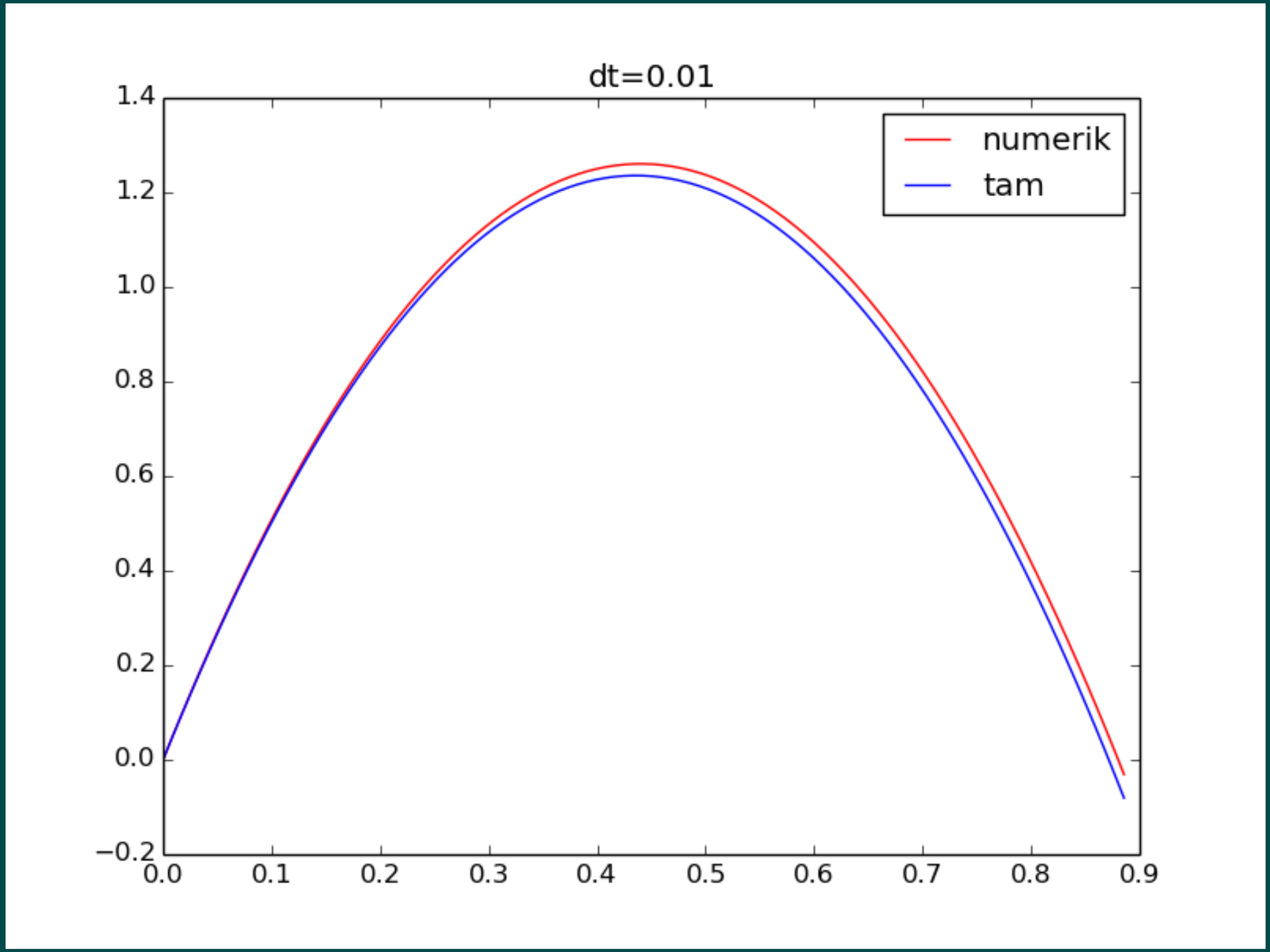
- ✓ Eğer yatay atışın grafiğini çizmek istiyorsanız cismin t anı için sadece yatay konumunu (x) değil aynı zamanda düşey konumunu (y) da bilmeli ve x'e karşılıkk y'yi çizdirmelisiniz.

```
x_degerleri = u[:,0]
y_degerleri = u[:,2]
plt.plot(x_degerleri, y_degerleri)
```

- ✓ Problemin tam (analitik) çözümü vardır ve nümerik çözümün başarısını test etmek için kolaylıkla kullanılabilir.

```
def tam(x):
    g = 9.81; y0 = u0[2]
    return x*tan(theta) - g*x**2/(2*v0**2)*1/(cos(theta))**2 + y0
plt.plot(x_degerleri, y_degerleri, "r-",
x_degerleri, tam(x_degerleri), "b-",
legend=("numerik", "tam"),
title="dt=%g" % dt)
```

- ✓ Problemin çözümünü uyg2_yatayatis.py dosyasında bulabilirsiniz. dt = 0.01 zaman aralığı için İleri Euler yöntemiyle çözüm aşağıdaki grafikte verilmiştir. Mavi gerçek çözümü, kırmızı ise nümerik yaklaşımı göstermektedir.



Dersi Özetleyen Örnek: Programlara Veri Girişi

- ✓ Programlara veri pek çok yoldan gelebilir: komut satırından kullanıcıyla interaktif olarak, dosya aracılığıyla, bir girdi formu (input form) aracılığıyla ya da bir Grafik Arayüz (GUI) aracılığıyla. Dolayısıyla her bir okuma yöntemini bir alt sınıfta kodlamak ve ortak kodu da üst sınıfa almak iyi bir strateji olarak görünmektedir. Programlarınızda ana programa birkaç satır ekleyerek bu kodu kolaylıkla adapte edebilir, farklı noktalardan veri girişini sağlayabilrsiniz.

- ✓ Programımız [a, n] aralığındaki x değerleri için f(x) değerini ekrana getirececk bir ana program örneği üzerinden kurgulayalım. Bu problem sadece örnek olarak seçilmiştir, sınıf hiyerarşisi yapısı başka bir probleme de kolaylıkla adapte edilebliir.

```
cikti_dosyasi = open(filename, 'w')
from numpy import linspace
for x in linspace(a, b, n):
    cikti_dosyasi.write('%12g %12g\n' % (x, f(x)))
    cikti_dosyasi.close()
```

- ✓ Amacımız a, b, n, dosya adı ve f (fonksiyon) değişkenlerine girdi sağlamaktır. f fonksiyonunu kullanıcı girdisinden oluşturmak üzere StringFunction fonksiyonunu kullanacağız.

```
from scitools.StringFunction import StringFunction
from numpy import linspace
f = StringFunction(formul)
```

- ✓ a, b, n, dosya adı ve f (fonksiyon) değişkenlerine girdi sağlamak üzere iyi bir yöntem bir sözlük oluşturup, bu sözlüğün içeriğini kullanıcıdan komut satırı, girdi dosyası ve/veya grafik arayüz araclığı ile gönderilecek veri ile güncellemek iyi bir fikirdir.

```
p = dict(formul='x+1', a=0, b=1, n=2, dosya_adi='veri.dat')
```

- ✓ Alt sınıfların yapacağı iş, adlarında verilene uygun olarak bu 5 değişkeni (a, b, n, dosya_adi, f) “beslemek” olacaktır.

```
girdi = Altsinif_adi(p):
a, b, n, dosya_adi, formul = girdi.veri_cek()
```

- ✓ Öncelikle `VeriOku` adında, parametrelerin bulunduğu sözlüğü bir öznitelik olarak başlatacak, sözlükten her bir parametrenin değerini tek tek çekebilecek bir `verial` metodu ve tüm veriler birlikte gönderildiğinde hepsini çekip ilgili parametrelere atayack bir `vericek` metoduna sahip bir sınıf kodlayalım.

```
class VeriOku:

    def __init__(self, parametreler):
        self.p = parametreler

    def verial(self, parametre_adi):
        return self.parametreler[parametre_adi]

    def vericek(self):
        return [self.p[ad] for ad in sorted(self.p)]

    def __str__(self):
        import pprint
        return pprint.pformat(self,p)
```

- ✓ Kodun bu yapısıyla parametreler sözlüğü sıralandığı (sorted) için bu kodu çağıracak programın parametreleri alfabetik sıraya göre çekeceği unutulmamalıdır.

```
a, b, dosya_adi, formul, n = girdi.girdiyi_oku()
```

- ✓ Veriyi kullanıcıdan interaktif olarak alma: Kullanıcıdan veri almann en kolay yolu bunu komut satırından kullanıcı girişiyle yapmaktır. Bunu aşağıdaki şekilde yapabiliriz.

```
class InteraktifGiris(VeriOku):
    def __init__(self, parametreler):
        VeriOku.__init__(self, parametreler)
        self._kullanici_mesaji
    def _kullanici_mesaji(self):
        for ad in self.p:
            self.p[ad] = eval(raw_input("Parametre adi " + ad + ": "))
```

- ✓ Bu kod yapısında eval fonksiyonunu kullanmak ciddi bir güçlük yaratacak. Zira, girdi bir metin olduğu zaman (örneğin dosya adı veri.dat), eval(veri.dat) bir hataya neden olacak; çünkü, “dat” uzantısı `eval` fonksiyonu tarafından “veri” nesnesinin bir metodu ya da özniteliği olarak yorumlanacaktır. Bunun yerine `scitools.misc` paketinden `str2obj` metodunu kullanmayı tercih edeceğiz. Bu _kullanici_mesaji fonksiyonundaki ifadeyi aşağıdaki şekilde değiştirmemizi gerektirir.

```
self.p[ad] = str2obj(raw_input("Parametre adi " + ad + ": "))
```

- ✓ Dosyadan okuma: Veriyi “parametre_adi = deger” ikilileri şeklinde bir dosyaya girip okunmasını da sağlayabiliriz. Örnek bir girdi dosyasının yapısı aşağıdaki gibi olacaktır.

```
formul = sin(x) + cos(x)
dosya_adi = veri.dat
a = 0
b = 1
```

- ✓ Bu noktada karşılaşabileceğimiz muhtemel bir problem dosya adını programa nasıl sağlayacağımız olacaktır. Bunun kolay bir yolu programı çalştırırken veri dosyasını ona girdi verecek şekilde bir yönlendirmeyle çalıştırmaktır.

```
ozbasturk@thau:~$ program.py < veri.dat
```

- ✓ Dosyayı satır satır okuyup, “=” işaretinden satırları ayırıp (split), işaretin sol tarafını parametre adı, sağ tarafının değeri olarak belirleyecek ve gereksiz boşlukları da atacak bir koda ihtiyacımız olacak.

```
class DosyadanOku(VeriOku):
    def __init__(self, parametreler):
        VeriOku.__init__(self, parametreler)
        self._dosyaoku
    def _dosyaoku(self, dosya=sys.stdin):
        for satir in dosya:
            if '=' in satir:
                ad, deger = satir.split('=')
                self.p[ad.strip()] = str2obj(deger.strip())
```

- ✓ Kodu bu şekilde yazmanın bir faydası kullanıcı dosya adını komut satırından istendiği şekilde sağlamazsa, interaktif oturumun başlaması ve kullanıcının bir önceki alt sınıfla okunabilecek parametre_adi = deger ikililerini sağlayabilmesidir. Programcı interaktif bir oturumdan `Ctrl + D` ile çıkabilir.
- ✓ Komut satırından okuma: Kullanıcı parametreleri komut satırından --secenek deger ikilileri ile programı çalıştırırken de sağlayabilir. Bunun için daha önce de gördüğümüz getopt fonksiyonunu kullanmalıyız.

```
ozbasturk@thau:~$ program.py --a 1 --b 10 --n 101 --formul "sin(x) + cos(x)"
```

```
class KomutSatirindanOku(VeriOku):
    def __init__(self, parametreler):
        self.sys_argv = sys.argv[1:]
        VeriOku.__init__(self, parametreler)
        self._komut_satirindan_oku()
    def _komut_satirindan_oku(self):
        # getopt icin opsiyonlari hazirla
        secenek_isimleri = [ad + "=" for name in self.p]
        try:
            secenekler,degerler = getopt.getopt(self.sys_argv, '', secenek_isimleri)
        except getopt.GetoptError, e:
            print 'Komut satirindan girilen secenekte hata:\n', e
            sys.exit(1)
        for secenek, deger in secenekler:
            for ad in self.p:
                if secenek == "--" + ad:
                    self.p[ad] = str2obj(deger)
```

- ✓ Kullanıcının komut satırından tüm parametreleri değil de sadece bazılarını sağlayabileceği esnek bir yapı kurmaya da ihtiyaç duyabiliriz.

```
a, b, n = girdi.verial('a', 'b', 'n')     # 3 degisken
n = girdi.verial('n')      # 1 degisken
```

- ✓ Bu esneklik `verial` fonksiyonunun aşağıdaki şekilde değiştirilmesiyle sağlanabilir.

```
class VeriOku:
    ...
    def verial(self, *parametre_adlari):
        if len(parametre_adlari) == 1:
            return self.p[parametre_adlari[0]]
        else:
            return [self.p[ad] for ad in parametre_adlari]
```

- ✓ Şimdi sınıf hiyerarşisine dayanan bu kodun nasıl çalıştığını bir ana program yazarak görelim.

```
p = dict(formul='x+1', a=0, b=1, n=2, dosya_adi='veri.dat')
from sinif_verioku import *
kullanici_girisi = eval(sys.argv[1])
del sys.argv[1] # getopt fonksiyonun dogru calismasi icin program adini siliyoruz
girdi = kullanici_girisi(p)
a, b, dosya_adi, formul, n = girdi.vericek()
print girdi
```

- ✓ Veri okumak için yazdığımız sınıflar sinif_verioku.py dosyasında kodu çalıştırmak için hazırladığımız ana programı demo_verioku.py dosyalarında bulabilirsiniz.

Dersi Özetleyen Örnek 2: Taysal Enerji Dağılımı (TED)
Spectral Energy Distribution (SED)

- ✓ Doğada kara cisme en benzeyen cisim yıldızlardır (belki yıldızların merkezi bölgeleri demek daha doğru). Bu nedenle kara cisim ışınımını tanımlayan yasalar en iyi yıldızlara uygulanır. Sıcaklığı T olan bir kara cismin λ dalgaboyunda yaptığı ışınım miktarının hesabı uzun süre astrofizikçileri meşgul etmiş, 1900 yılında Max Planck'ın modern fiziğin ve kuantum teorisinin temellerini atan önerisiyle tam bir çözüme kavuşturulabilmiştir. Öncesinde Wihelm Wien tarafından 1896 yılında kısa dalgaboylarında geçerli, Rayleigh-Jeans tarafından ise uzun dalgaboylarında geçerli iki yaklaşım önerilmiştir.

- ✓ Sınıf hiyerarşisi dahlinde bu üç yöntemi adım adım kodlayarak sıcaklığı T olan bir kara cismin λ dalgaboyunda yaptığı ışınım miktarını W sr^{-1} m^3 biriminde hesab eden bir program geliştirelim. Sonrasında kodumuzuı geniş bir dalgaboyu aralığı dahlinde enerji miktarlarını her üç yöntemle de hesap eden ve birbirleri ile bir grafik üzerinde karşılaştıran bir ana program dahilinde test edelim.

$$I(\lambda, T) = \frac{2hc^2}{\lambda^5} e^{-\frac{hc}{\lambda k T}}$$

Wien Yaklaşımı

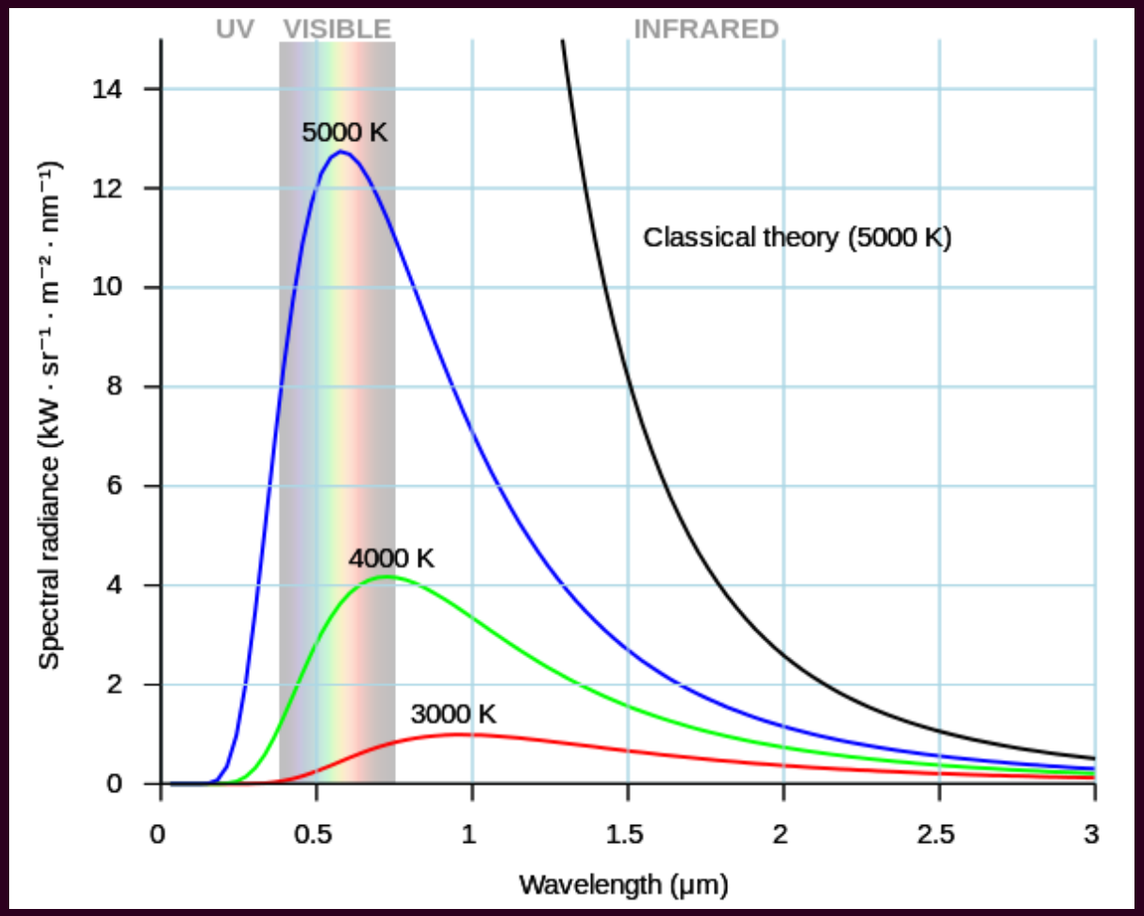
$$B_\lambda(T) = \frac{2ck_\mathrm{B}T}{\lambda^4}$$

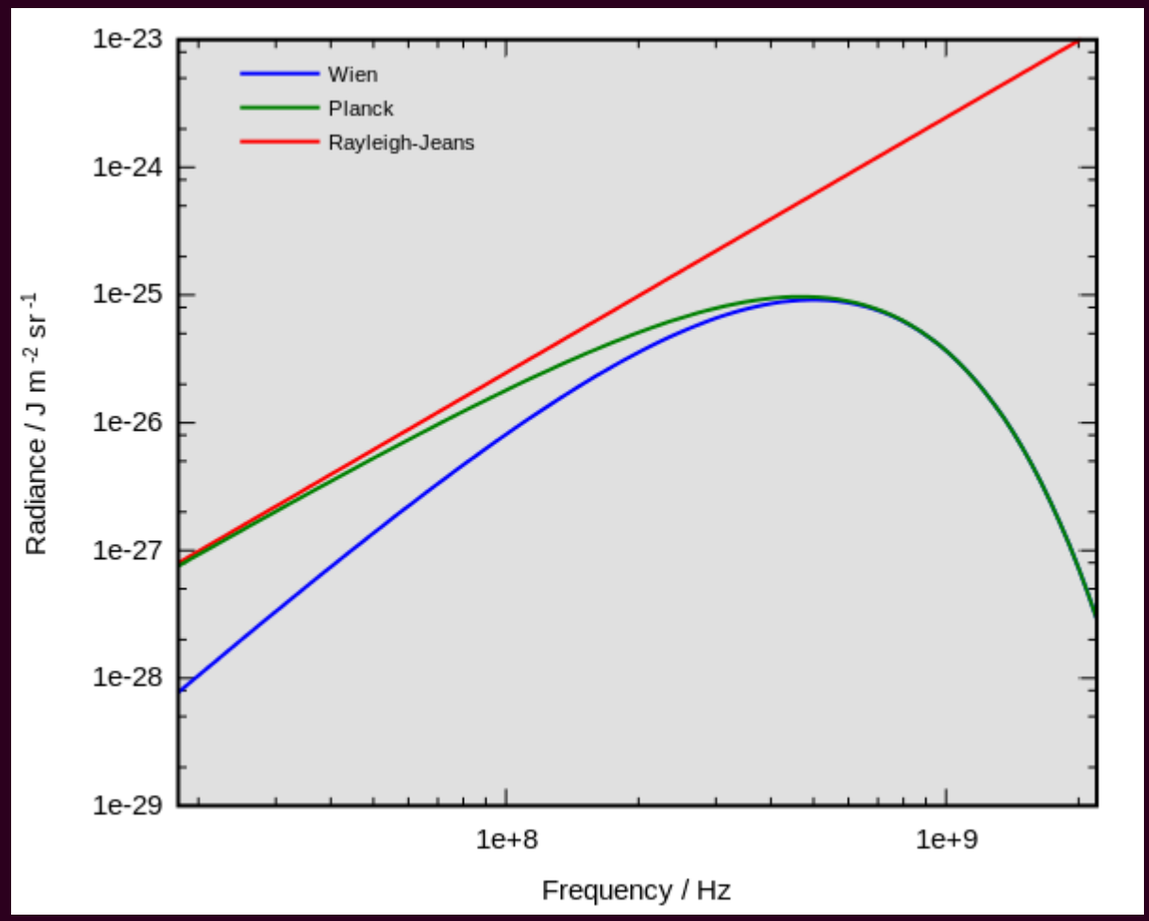
Rayleigh-Jeans Yaklaşımı

$$B_\lambda(\lambda, T) = \frac{2hc^2}{\lambda^5} \frac{1}{e^{\frac{hc}{\lambda k_\mathrm{B} T}} - 1}.$$

Planck Yasası

Tayfsal Enerji Dağılımı (TED)
Sonuçların Karşılaştırılması





Alıştırmalar - I

1. Sıcaklığı (T), yarıçapı (R) ve uzaklığı (d) bilinen bir yıldızın aşağıda listelenen parametrelerini hesaplayan `YildizPar` isimli bir sınıf kodlayınız.

1.a. Sınıfınızın sıcaklk (T), yarıçap ve uzaklık (d) parametrelerinin yanı sıra bir de gerekli sabitleri (Güneş'in ilgili parametreleri gibi) başatan `__init__` başlatıcı metodu olmalıdır.

1.b. Yıldızın toplam ışınım gücünü (lüminosite) hesap eden bir `L metodu` olmalıdır.

1.c. Yıldızın mutlak bolometrik parlaklığını hesaplayan `Mbol` metodu olmalıdır.

1.d. Uzaklık modülünü hesaplayan `uzaklik_modulu` metodu olmalıdır.

1.e. Yıldızın görünen bolometrik parlaklığını hesaplayan `mbol` metodu olmalıdır.

1.f. Yıldızın görünen görsel parlaklığını hesaplayan `mV` metodu olmalıdır.

1.g. Yıldızdan Dünya'ya ulaşan akıyı hesaplayan `gorunen_aki` metodu olmalıdır.

1.h. Yıldızın maksimum ışınım yaptığı dalgaboyunu hesaplayan `wien_kayma_yasasi metodu` olmalıdır.

1.i. Sınıfın ve fonksiyonlarının ne yaptığnı, hangi öznitelik ve metotları içerdiğini anlatan bir iç dokümantasyonu olmalıdır.

1.j. Sınıfınızın hangi parametreleri alıp, sonuçta hangi parametreleri hesapladığını anlatan bir __str__ metodu olmalıdır.

Yazdığınız sınıfı test etmek üzere bir `_test()` fonskiyonu yazınız (aynı kodun içine) ve sınıfınızı Dschubba (T = 28000 K, R = 5.16×10^{11} m, d = 180 pc) ve Güneş yıldızları için test ediniz.

2. `YildizPar` sınıfını `TayfsalEnerjiDagilimi` sınıfının bir alt sınıfı olarak kodlayabilir misiniz? Bunun getireceği avantajları değerlendiriniz ve avantajlı olacağını düşünüyorsanız sınıfı bu şekilde kodlayınız.

Gerekli sabitler: $\sigma = 5.670373 \times 10^{-5}$ erg cm^{-2} s^{-1} K^{-4}, M_V (Güneş) = $4^m.83$, M_{bol}(Güneş) = $4^m.75$, $L_{güneş} = 3.826 \times 10^{33}$ erg / s

Alıştırmalar - II

3. Dalgalar fiziğinin en önemli filkirlerinden biri her kompleks dalga formunun kosinüs ve sinüs fonksiyonlarının harmoniklerinin toplamı olarak ifade edilebilmesidir.

$$f(x) = a_0 + a_1 \cos x + a_2 \cos 2x + a_3 \cos 3x + a_4 \cos 4x + ... + b_1 \sin x + b_2 \sin 2x + b_3 \sin 3x + b_4 \sin 4x + ...$$

Bu şekilde oluşturulan serilere Fourier serileri adı verilir.

`FourierSerisi` adında ve aşağıdaki özellikleri sağllayan bir sınıf kodlayanız.

3.a. Sınıfınızın $a = [a_0, a_1, a_2, a_3, ..., a_{n-1}]$ ve $b = [b_0, b_1, b_2, b_3, ..., b_{n-1}]$ harmonik terimlerinin katsayılarını birer NumPy dizisi olarak başlatan bir `__init__(self, a, b, n)` metodu bulunmalıdır (n: terim sayısı).

3.b. Sınıfınızın herhangi bir x için serinin değerini hesaplayan bir `__call__(self, x)` metodu olmaldır.

3.c. Sınıfınızın verilen katsayilar için oluşan Fourrier serisini ekrana güzel bir şekilde yazdıran bir `__str__(self)` metodu olmaldır.

Test: Bu kodu kullanarak aşağıdaki dalga formunu oluşturan bir `_test()` fonksiyonu yazınız.

$$\Psi = 2 / (n+1) (\sin x - \sin 3x + \sin 5x - \sin 7x + ... -/+ \sin nx) = 2 / (n + 1) \Sigma^{n}_{k=1} (-1)^{(k-1)/2} \sin kx;\ k : \text{tek tam sayı}$$

Baştaki 2 / (n + 1) terimi dalga formunun şeklini değiştirmeyecek sadece maksimum değeri 1 olacak şekilde ölçeklendirecektir.

x = [0, pi] aralığında (radyan cinsinden) n = 5 nokta için Ψ dalga formunun grafiğini çiziniz. Daha sonra n'i sırasıyla 11, 21 ve 41'e çıkararak grafik çizimini tekrarlaynız ve dalga formunun nasıl değiştiğini gözleyiniz.