Report file - Problem Set #7

Matteo Dell'Acqua
GitHub: MatteoDellAcqua6121

October 28, 2024

Abstract

This is the report for the problem set #7. Since the problem set is composed of two exercises, we divide the report into two sections, one for each problem. The scripts (labelled as ps_7.'problem number') and the raw file of the images are in this directory.

1 Problem 1

1.1 Formulation of the problem

We are asked to code a function that computes the position of the L_1 Lagrange point, where a satellite orbits a heavy celestial body M in synchrony with a much lighter one m ($M \gg m \gg$ satellite mass).

Theorem 1 *The distance r from the centre of the heavy object to the satellite satisfies:*

$$\frac{GM}{r^2} - \frac{Gm}{(R-r)^2} = \omega^2 r \tag{1}$$

with $\omega^2 = GM/R^3$.

In order for the satellite to move in synchrony with the light body, its acceleration (LHS) needs to match the centripetal force (RHS) for a circular motion with the same angular velocity ω of that of the light body. This is found by the same matching procedure:

$$\omega^2 R = \frac{GM}{R^2} \tag{2}$$

Dividing eq. (1) by GM/R^2 we obtain:

Theorem 2 *The position of L_1 can satisfies:*

$$\frac{1}{r'^2} - \frac{m'}{(1-r')^2} = r' \tag{3}$$

where we defined $m' = m/M$ and $r' = r/R$.

The bulk of the problem consists in finding the root $\overline{r'}$ of eq. (3), or equivalently:

$$f(r') = (1-r')^2 - m'r'^2 - (1-r')^2 r'^3 \tag{4}$$

1.2 Computational methods

In order to find the root, we first need to determine an interval in which it is via *bracketing*: we start with a "reasonable" interval and then increase it symmetrically until the function has opposite sign at the extrema (and thus a root in the interval). In our case, we know the L_1 point is between the two celestial bodies in question, i.e. $\overline{r'} \in [0, 1]$ in natural units, and thus we start with the interval $[0.4, 0.6]$:

```
    #import the bracket and newton functions from the class jupiter notebook
def bracket(func,m):
    a = 0.4
    b = 0.6
    maxab = 1.e+7
    while(b - a < maxab):
        d = b - a
        a = a - 0.1 * d
        b = b + 0.1 * d
        if(func(a,m) * func(b,m) < 0.):
            return(a, b)
    return(a, b)
```

The *Newton method* uses the knowledge of the derivative of the function whose root we are trying to find, in order to improve the efficiency:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}, \qquad \epsilon_{i+i} \simeq -\frac{\epsilon_i^2 f''(\overline{x})}{2f(\overline{x})} \tag{5}$$

Thus we define the function and its derivative (computed via Wolphram Alpha):

```
    def func(x,m):
        return((1.-x)**2-m*x**2-x**3*(1.-x)**2)

    def dfunc(x,m):
        return(-2. - 2.*(-1. + m)* x - 3.* x**2 + 8. *x**3 - 5. *x**4)
```

and the Newton method is implemented as follows:

```
 def newton_raphson(xst,m):
    tol = 1.e-10
    maxiter = 100
    x = xst
    for i in np.arange(maxiter):
        delta = - func(x,m) / dfunc(x,m)
        x = x + delta
        if(np.abs(delta) < tol):
            return(x)
```

Finally, we need to remember that the result is written in natural units, and we multiply by R:

```
    def L1(m,R):
        (a, b) = bracket(func,m)
        z= newton_raphson(0.5*(a+b),m)*R
        return z
```

where we started the Newton method from the middle of the resulting bracketing.

1.3 Results

We are able to find the position of the L_1 point for the following celestial systems:

$$\text{Earth and Moon}: \quad 3.263 \cdot 10^8 m \tag{6}$$
$$\text{Sun and Earth}: \quad 1.472 \cdot 10^{11} m \tag{7}$$
$$\text{Sun and Jupiter-mass planet at the distance of the Earth}: \quad 1.388 \cdot 10^{11} m \tag{8}$$

2 Problem 2

2.1 Formulation of the problem

In this problem we are going to minimize the function:

$$y = (x - 0.3)^2 \exp(x) \tag{9}$$

using Brent's 1D method.

2.2 Computational methods

Brent's method is a combination of the parabolic and golden ratio ones.

Parabolic method: it takes a clever route of fitting a parabola to $x(f)$ and update x to be the *exact* root of the *approximating* parabola:

$$x = b + \frac{P}{Q} \tag{10}$$
$$R = \frac{f(b)}{f(c)} \tag{11}$$
$$S = \frac{f(b)}{f(a)} \tag{12}$$
$$T = \frac{f(a)}{f(c)} \tag{13}$$
$$\mathrm{P} = S\left[T(R-T)(c-b) - (1-R)(b-a)\right] \tag{14}$$
$$Q = (T-1)(R-1)(S-1) \tag{15}$$

The code is taken from the class Jupiter notebook:

```
def parabolic_step(func=None, a=None, b=None, c=None):
    """returns the minimum of the function as approximated by a parabola"""
    fa = func(a)
    fb = func(b)
    fc = func(c)
    denom = (b - a) * (fb - fc) - (b -c) * (fb - fa)
    numer = (b - a)**2 * (fb - fc) - (b -c)**2 * (fb - fa)
```

```
    # If singular, just return b
    if(np.abs(denom) < 1.e-15):
        x = b
    else:
        x = b - 0.5 * numer / denom
    return(x)
```

Golden ratio: it updates the bracketing by keeping the fractional length the same (it turns out to be equal to the *golden ratio*) and such that the conditions:

$$f(b) < f(a), \quad f(b) < f(c) \tag{16}$$

are always satisfied.

The code is a slight modification of the one from the class Jupiter notebook, in order to implement a single step and returning the whole tuple of the updated intervale:

```
    def golden_step(func=None, astart=None, bstart=None, cstart=None, tol=1.e-5):
    gsection = (3. - np.sqrt(5)) / 2
    a = astart
    b = bstart
    c = cstart
    # Split the larger interval
    if((b - a) > (c - b)):
        x = b
        b = b - gsection * (b - a)
    else:
        x = b + gsection * (c - b)
    fb = func(b)
    fx = func(x)
    if(fb < fx):
        return (a,b,x)
    else:
        return (b,x,c)
```

Brent's method: it uses parabolic approximations, but it keeps track of a bracketing interval, and under certain conditions, it reverts to a golden section search.

These conditions are:

- The parabolic step falls outside the bracketing interval
- The parabolic step is greater than the step before the last. The neat observation is that when $Q << 1$ the function `parabolic_step` simply does not update the bracket!

```
def brent(f,astart,bstart,cstart, tol=1.e-5, maxiter=10000):
    a = astart
    b = bstart
    c = cstart
    bold = b + 2. * tol
    niter = 0
    while((np.abs(bold - b) > tol) & (niter < maxiter)):
```

```
        bold = b
        #compute the parabolic step
        b = parabolic_step(func=func, a=a, b=b, c=c)
        if(a< b < bold):
            c = bold
        elif(bold<b<c):
            a = bold
        #use the golden step for anomalous cases: either b outside of the interval or Q<<1
        #(remember that in this case, the parabolic_step function just returns bold)
        else:
            (a,b,c)=golden(func=func, a=a, b=b, c=c)
        niter = niter + 1
    return(b)
```

Once again, before starting we implement a bracketing (on the derivative of y since we are looking for a critical point).

2.3 Results

We obtain:

$$z = 0.299998 \tag{17}$$

which matches with the result of the implementation of `scipy.optimize.brent`:

$$z_{\text{SciPy}} = 0.300000, \quad \delta_z := z - z_{\text{SciPy}} = -2.439 \cdot 10^{-6}, \quad \delta_z/z = -8.129 \cdot 10^{-6} \tag{18}$$