Georreferenciación de imágenes AVHRR basada en la resolución de la órbita y algoritmos de matching con una imagen de referencia

---

Máster en Geotecnologías Cartográficas
en Ingeniería y Arquitectura

Universidad de Salamanca

Trabajo Fin de Máster

4 de septiembre de 2015

Autor: Víctor Molina García

Tutor: Diego González Aguilera

Este trabajo no habría sido posible sin el soporte económico y académico del Laboratorio de Teledetección de la Universidad de Valladolid, cuyos responsables son el catedrático Dr. José Luis Casanova Roque y la profesora titular Dra. María Julia Sanz Justo, y de la Fundación General de esta misma universidad. A ambos agradezco la ayuda y confianza depositada.

Asimismo, quiero expresar mi más sincera gratitud a mi tutor, Dr. Diego González Aguilera, por su seguimiento y ayuda académica en el campo de la visión computacional y de la correspondencia de imágenes, al profesor Dr. Abel Calle Montes, por la información y ayuda provista en el ámbito de la mecánica orbital, y al investigador Dr. Javier Gutiérrez Fernández, por todas las dudas compartidas aprendiendo LaTeX a lo largo de estos últimos años.

Finalmente, doy las gracias a mis padres, a mi hermana y a Cristina, por haberme aguantado todo este último año.

Valladolid, 4 de septiembre de 2015

Índice general

1. Introducción **7**

1.1. El sensor AVHRR 7

1.2. Georreferenciación de imágenes AVHRR 9

1.3. Objetivos del trabajo 11

2. Marco teórico **13**

2.1. Modelo de elipsoide terrestre 14

2.2. Sistemas de referencia 15

2.2.1. Sistema de referencia terrestre inercial 15

2.2.2. Sistema de referencia terrestre fijo 16

2.2.3. Sistema de referencia terrestre geodésico 19

2.2.4. Sistema de referencia satelital geodésico 21

2.2.5. Sistema de referencia satelital fijo 22

2.3. Modelo orbital 23

2.3.1. Parámetros de Kepler del satélite 23

2.3.2. Obtención de coordenadas ECI del satélite 24

2.3.3. Propagación de los parámetros de Kepler 26

2.4. Modelo de estimación de coordenadas 28

2.4.1. Navegación directa 29

2.4.2. Corrección del desfase del reloj 32

2.4.3. Corrección de la postura del sensor 33

2.5. Modelo de correspondencia de imágenes 36

3. Implementación del modelo **41**
3.1. Flujo de trabajo . . . 42
3.1.1. Determinación de la posición del satélite . . . 42
3.1.2. Estimación inicial de las coordenadas de la imagen . . . 43
3.1.3. Detección de errores de georreferenciación en la imagen . . . 43
3.1.4. Actualización de las coordenadas de la imagen . . . 45
3.2. Librería python-sat . . . 46
3.2.1. Lectura y escritura de efemérides . . . 46
3.2.2. Predicción de la posición del satélite . . . 47
3.2.3. Estimación de las coordenadas de la imagen . . . 47
3.2.4. Actualización de las coordenadas de la imagen . . . 48

4. Aplicaciones y resultados **51**
4.1. Predicción de las coordenadas de MetOp-B . . . 52
4.2. Georreferenciación de imágenes AVHRR . . . 57

5. Conclusiones y líneas futuras **69**

A. Código fuente de python-sat **73**
A.1. Definiciones preliminares . . . 74
A.2. Clase Ephemeris . . . 86
A.3. Clase Orbit . . . 98
A.4. Clase Scene . . . 105
A.5. Clase Matcher . . . 119

Bibliografía **131**

Capítulo 1

Introducción

La correcta georreferenciación de las imágenes de satélite es crucial para poder realizar estudios en los que intervengan series temporales, pues su exactitud determinará la fiabilidad del análisis llevado a cabo. El sensor AVHRR sigue siendo en este sentido un reto después de más de 35 años de servicio.

1.1. El sensor AVHRR

El sensor AVHRR (*Advanced Very High Resolution Radiometer*) es un radiómetro diseñado inicialmente para observación meteorológica (por ejemplo, para la cobertura de nubes), pero que con el paso del tiempo empezó a ser utilizado para otros propósitos no meteorológicos, como la monitorización del índice de vegetación (*Normalized Difference Vegetation Index*, NDVI).

La última versión de este radiómetro, AVHRR/3, cuenta con un total de seis canales con una resolución espacial de 1100 m y una resolución radiométrica de 10 bits, y se encuentra a bordo de los últimos satélites de la familia NOAA (15, 16, 17, 18 y 19) y MetOp (A y B). En cada línea de imagen, este sensor barre aproximadamente 2580 km con un ángulo de barrido de 55.4° a cada lado del nadir.

El AVHRR es un sensor barato de producir, pero a cambio tiene algunos inconvenientes. Uno de los más importantes es la ausencia de *star tracker*, un dispositivo óptico

que permite conocer la posición del satélite observando el fondo de estrellas fijas. En consecuencia, las imágenes AVHRR tienen que ser georreferenciadas siempre en la etapa de posproceso.

Según el grado de procesado de los ficheros capturados del sensor AVHRR, se distinguen tres niveles:

- El Level 0 es el conjunto de datos brutos obtenidos a través de una estación de captura. La estructura de estos ficheros se encuentra detallada en [1].

- El Level 1A está formado por la imagen AVHRR extraída del Level 0, pero sin tener todavía una estructura y formato secuencial en el tiempo para análisis y generación de productos. Es un paso intermedio hacia el siguiente nivel de procesado.

- El Level 1B es el conjunto de datos extraídos en canales, referenciados temporalmente y con metadatos adicionales que indican la calidad de la información, los coeficientes de calibración y los parámetros de georreferenciación [2].

Canal	Longitud de onda central / µm	Ancho de banda / µm	Usos y aplicaciones en productos finales
1	0.630	0.580-0.680	Índice de vegetación / Nubes
2	0.865	0.725-1.000	Índice de vegetación / Nubes
3A	1.610	1.580-1.640	Temperatura de superficie
3B	3.740	3.550-11.300	Incendios forestales
4	10.800	10.300-11.300	Temperatura de superficie / Nubes
5	12.000	11.500-12.500	Temperatura de superficie / Nubes

Cuadro 1.1: Características de los canales del sensor AVHRR/3. Los canales 3A y 3B no funcionan simultáneamente (3A de día, 3B de noche).

1.2. Georreferenciación de imágenes AVHRR

La georreferenciación de imágenes de satélite requiere al menos de dos etapas: primero, calcular con exactitud la posición del satélite en el momento de captura de las líneas de imagen; segundo, proyectar correctamente los píxeles de esta imagen sobre la superficie terrestre.

La solución al problema de la navegación satelital en ausencia de frenado atmosférico fue desarrollada teóricamente por Brouwer en 1959 [3]. Desde entonces, diferentes metodologías se han abordado para resolver el problema de la navegación de los satélites con el sensor AVHRR a bordo. La mayoría de estos métodos se apoyan en al menos un punto de control (por ejemplo, el desarrollado por Brunel y Marsouin [4], o el implementado por Illera [5]). Rosborough propuso en 1994 el planteamiento básico para un modelo automático de navegación en el que se tiene en cuenta la postura del sensor y cómo afecta esta a la formación de la imagen [6]. Este fue la base para el desarrollo de un algoritmo automático de corrección geográfica por Calle [7].

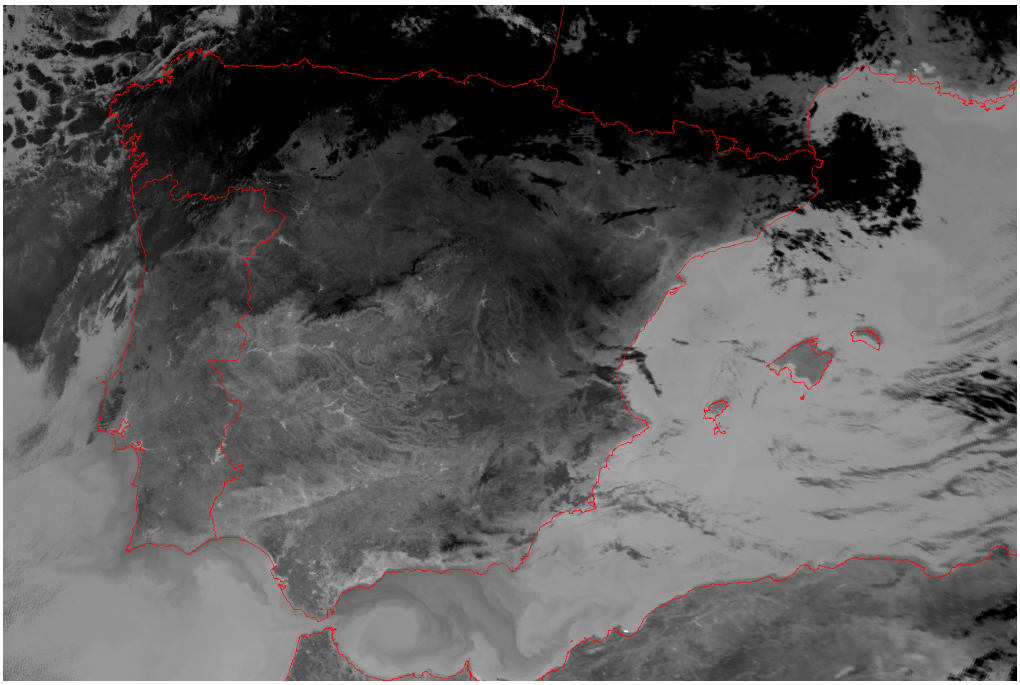

Figura 1.1: Imagen AVHRR (2014/07/08 02:52:09) procesada con iDAP/MacroPro.

Para calcular las órbitas de los satélites es necesario el uso de un modelo físico de la órbita, basado fundamentalmente en las leyes de Kepler, la ley de la gravitación universal de Newton y teoría de perturbaciones. Actualmente, el modelo general de perturbaciones simplificado 4 (*Simplified General Perturbations Number 4*, SGP4) [8], publicado en 1988 [9] en FORTRAN IV, es una de las referencias principales. Este modelo predice el efecto de las perturbaciones de la órbita satelital como consecuencia de la irregularidad de la superficie terrestre, el frenado atmosférico y efectos gravitacionales de cuerpos como la Luna y el Sol. Los parámetros que recibe como variables de entrada y que propaga en el tiempo reciben el nombre de *two-line element set* (TLE), y se encuentran disponibles en Internet, donde son actualizados diariamente. SGP4 tiene un error de aproximadamente 1 km en la época de referencia, y este error crece entre 1 y 3 km por día.

Aunque el código original del modelo SGP4 se encuentra en FORTRAN IV, existen traducciones a otros lenguajes de programación (por ejemplo, C). La librería de Python `pyephem`, desarrollada por Brandon Rhodes [10], es capaz de predecir las órbitas satelitales a partir de un TLE, y utiliza en su núcleo las herramientas XEphem [11], que en última instancia ejecutan el código del modelo SGP4.

Existen varias herramientas de software que actualmente realizan al menos una parte del procesado de las imágenes AVHRR/3 del satélite MetOp-B:

- David Taylor ha desarrollado el software Metop Manager [12], capaz de procesar las imágenes AVHRR de la familia MetOp desde el Level 0 hasta el Level 1B. Es un programa de pago y no es libre: se oferta como un producto final.

- Dartcom es una empresa fabricante de estaciones de meteorología y teledetección, y con sus antenas de recepción proporciona el software iDAP/MacroPro [13]. Este programa recibe los ficheros Level 0 de los receptores de Dartcom y los convierte a Level 1B en una ventana geográfica y proyección. No se tiene acceso a los productos intermedios y no es un programa libre, por lo que su uso es rígido. Pese a usar el modelo SGP4 en la predicción orbital, las imágenes AVHRR finales proyectadas presentan distorsión (ver Figura 1.1).

- La Numerical Weather Prediction Satellite Application Facility (NWP SAF) mantiene la librería ATOVS and AVHRR Pre-processing Package (AAPP) [14], cuyo responsable último es Nigel Atkinson. Es capaz de generar el producto Level 1B a partir de los ficheros Level 0. Su uso requiere únicamente un registro gratuito. Con fecha de 27 de julio de 2015, Nigel Atkinson comunicó en el foro de usuarios de AAPP la necesidad de corregir los ángulos de postura para el sensor AVHRR de MetOp-A y MetOp-B, ya que las georreferenciaciones procedentes de AAPP discrepaban con el procesado de EUMETSAT hasta 2.5 km en el nadir y hasta 17 km en los píxeles situados en los extremos de la imagen.[1]

La necesidad de disponer de imágenes AVHRR georreferenciadas en tiempo real (por ejemplo, en la detección de incendios forestales) provoca que para determinadas aplicaciones no sea viable esperar a disponer de los puntos de control provistos en las imágenes de repositorios oficiales, como por ejemplo CLASS NOAA[2]. Asimismo, uno de los principales problemas recurrentes en la automatización de los algoritmos de navegación se encontraba en la búsqueda de puntos de control. No obstante, el desarrollo de las ciencias de la computación y los avances tecnológicos desde comienzos del siglo XXI, así como los nuevos algoritmos diseñados en el ámbito de la visión computacional (SIFT, SURF, FAST, FREAK, etc.), hacen que la generación de estos puntos de anclaje sea actualmente viable sin la necesidad de un operario que realice la tarea de forma manual.

1.3. Objetivos del trabajo

El objetivo del presente trabajo es realizar una primera aproximación a la viabilidad de georreferenciar automáticamente imágenes diurnas del sensor AVHRR/3 a bordo del satélite MetOp-B combinando los resultados clásicos de navegación orbital con puntos de control y los nuevos algoritmos de reconocimiento de rasgos.

[1] Foro de usuarios de AAPP: http://www.nwpsaf.eu/forum/viewtopic.php?f=15&t=231. Consultado el 10 de agosto de 2015.

[2] Enlace web: http://www.class.ngdc.noaa.gov/.

Se ha prescindido del modelo SGP4 y en su lugar se ha desarrollado una librería, `python-sat`, que implementa la teoría orbital presentada fundamentalmente por Rosborough [6] y Calle [7]. La detección de puntos de control se ha basado en la combinación del detector FAST (*Features from Accelerated Segment Test*), el descriptor BRISK (*Binary Robust Invariant Scalable Keypoints*) y una imagen de referencia. Esta librería se ha puesto a prueba con una batería de imágenes obtenidas del repositorio CLASS NOAA, correspondientes a la Península Ibérica capturada en diversas regiones de la imagen en función de la hora de pase del satélite.

Capítulo 2

Marco teórico

En el presente capítulo se muestran las bases teóricas de la implementación que se desarrolla en este trabajo. En esencia, y teniendo en cuenta que el objetivo primordial es proporcionar una buena georreferenciación de imágenes del sensor AVHRR/3 a bordo del satélite MetOp-B, cabe mencionar:

- El modelo geométrico utilizado para describir la Tierra y su superficie, puesto que sobre ella debe proyectarse la imagen obtenida por el sensor.
- Los sistemas de referencia que participan durante todo el proceso, algunos de ellos con referencia en el centro de la Tierra, otros con referencia en el centro del satélite.
- El modelo orbital utilizado para describir el movimiento del satélite y, especialmente, el modelo de propagación de estos parámetros orbitales a partir de ciertos parámetros conocidos en un instante conocido como época.
- El modelo de proyección de los píxeles de la imagen sobre la superficie de la Tierra en función del modelo orbital y los parámetros internos del sensor, el cual permite obtener una primera aproximación de las coordenadas de la imagen.
- El modelo de correspondencia de imágenes que, mediante una imagen de referencia con coordenadas fiables y un proceso de extracción de características, permita la corrección de los errores en la estimación de las coordenadas de la imagen.

2.1. Modelo de elipsoide terrestre

La georreferenciación de imágenes de satélite sobre la superficie terrestre hace necesaria la modelización de la Tierra según un tipo de estructura geométrica con el que muestre cierta semejanza. El esferoide oblato con origen en el punto (0, 0, 0) permite aproximar la superficie terrestre según la siguiente expresión:

$$\frac{x^2 + y^2}{a^2} + \frac{z^2}{b^2} = 1, \tag{2.1}$$

donde (x, y, z) son las coordenadas cartesianas sobre la superficie del esferoide (en el sistema de coordenadas alineado con los ejes principales del esferoide), a es el semieje mayor del esferoide y b es el semieje menor. Asimismo, se define la excentricidad e y el grado de achatamiento f del esferoide como:

$$e = \sqrt{1 - \left(\frac{b}{a}\right)^2}, \qquad f = 1 - \sqrt{1 - e^2}. \tag{2.2}$$

Por otra parte, el propio esferoide terrestre se encuentra en rotación sobre un eje que forma 23 grados y 5 minutos sobre la normal al plano de la eclíptica, y cuya velocidad angular de rotación se denota por ω_{E}.

El modelo de elipsoide utilizado con más frecuencia en la actualidad es el denominado WGS84 (*World Geodetic System 84*), y es el que se utiliza en adelante en este trabajo. Sus parámetros son [15]:

- Un semieje mayor $a_{\mathrm{E}} = 6\,378\,137\,\mathrm{m}$.
- Un semieje menor $b_{\mathrm{E}} = 6\,356\,752.3142\,\mathrm{m}$.
- Un grado de achatamiento $f_{\mathrm{E}} = 1/298.257\,223\,563$.
- Una excentricidad $e_{\mathrm{E}} = 0.081\,819\,19$.
- Una velocidad angular de rotación terrestre $\omega_{\mathrm{E}} = 7.292\,115 \times 10^{-5}\,\mathrm{rad\,s^{-1}}$.
- Un factor gravitacional $\mu = 3.986\,004\,418 \times 10^{14}\,\mathrm{m^3\,s^{-2}}$, obtenido como producto de la constante gravitacional G y la masa de la Tierra M.

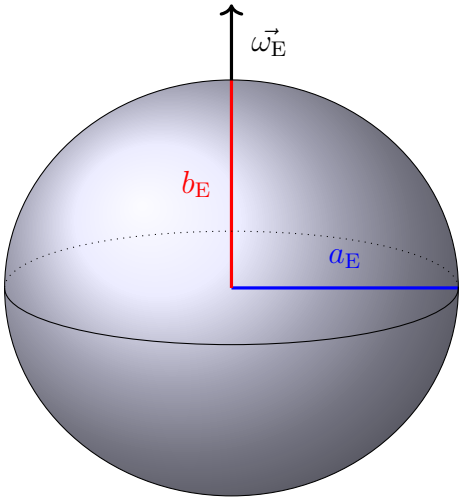

Figura 2.1: Representación del elipsoide terrestre WGS84.

2.2. Sistemas de referencia

La generación de las coordenadas geográficas de los píxeles de una imagen de satélite implica la utilización de una amplia variedad de sistemas de referencia. Por este motivo, en esta sección se describen brevemente las características de todos ellos y sus fórmulas de conversión.

2.2.1. Sistema de referencia terrestre inercial

El sistema de referencia terrestre inercial (*Earth-centered fixed system*, ECI) es el sistema base de coordenadas cartesianas en que se sustenta el resto de sistemas de referencia. Su característica principal es que permanece fijo respecto a las estrellas, por lo que en primera aproximación se puede considerar que el sistema ECI es inercial.

El centro del sistema de referencia ECI es el centro de la Tierra, y sus ejes principales se definen del siguiente modo:

- El eje X se encuentra contenido en el plano ecuatorial y apunta al punto vernal o punto Aries, es decir, el punto de la eclíptica en el que el Sol pasa del hemisferio sur celeste al hemisferio norte.

- El eje Z es perpendicular al eje X y paralelo al eje de rotación terrestre y con el mismo sentido que el vector velocidad angular de rotación terrestre $\vec{\omega}_{\mathrm{E}}$.
- El eje Y se define de manera que la terna de ejes (X, Y, Z) sea un triedro directo.

Un punto genérico en un instante de tiempo t es caracterizado en este sistema de referencia por un vector $\vec{r}_{\mathrm{i}}$ de tres coordenadas, usualmente en metros:

$$\vec{r}_{\mathrm{i}}(t) = [x_{\mathrm{i}}(t), y_{\mathrm{i}}(t), z_{\mathrm{i}}(t)]^{\intercal}.$$

Asimismo, la velocidad de un punto genérico viene dada por el correspondiente vector $\vec{v}_{\mathrm{i}}$ en metros por segundo:

$$\vec{v}_{\mathrm{i}}(t) = \dot{\vec{r}}_{\mathrm{i}}(t) = [\dot{x}_{\mathrm{i}}(t), \dot{y}_{\mathrm{i}}(t), \dot{z}_{\mathrm{i}}(t)]^{\intercal}.$$

El modelo orbital que se detalla en la siguiente sección proporciona las coordenadas del satélite en el sistema ECI.

2.2.2. Sistema de referencia terrestre fijo

El sistema de referencia terrestre solidario (*Earth-centered Earth-fixed*, ECEF) es un sistema de referencia cartesiano no inercial. Al contrario que en el sistema ECI, el eje X del sistema ECEF no es un eje fijo respecto a las estrellas: está contenido dentro del plano ecuatorial en la dirección y sentido que atraviesa el meridiano de Greenwich, por lo que rota solidario con este. El eje Z del sistema ECEF coincide con el del sistema ECI, y el eje Y se define de manera que los ejes (X, Y, Z) del sistema ECEF formen un triedro directo.

Un punto genérico desplazándose en un instante t viene dado en el sistema ECEF por sus correspondientes vectores posición $\vec{r}_{\mathrm{e}}$ y velocidad $\vec{v}_{\mathrm{e}}$:

$$\vec{r}_{\mathrm{e}}(t) = [x_{\mathrm{e}}(t), y_{\mathrm{e}}(t), z_{\mathrm{e}}(t)]^{\intercal}.$$

$$\vec{v}_{\mathrm{e}}(t) = \dot{\vec{r}}_{\mathrm{e}}(t) = [\dot{x}_{\mathrm{e}}(t), \dot{y}_{\mathrm{e}}(t), \dot{z}_{\mathrm{e}}(t)]^{\intercal}.$$

La transformación de coordenadas de un punto $\vec{r}_{\mathrm{i}}(t)$ del sistema ECI al sistema ECEF puede interpretarse, en primera aproximación, como una rotación $\gamma(t)$ de los ejes X e Y del sistema ECI hasta que coincidan con los correspondientes ejes X e Y del sistema ECEF (pues se asume que sus ejes Z coinciden). Este ángulo $\gamma(t)$ es el formado por los dos planos siguientes:

- El plano que contiene al eje Z del sistema ECI y a la semirrecta con origen en el centro de la esfera terrestre y dirigida hacia el punto vernal.
- El plano que contiene el meridiano de Greenwich.

El ángulo $\gamma(t)$ recibe el nombre de tiempo sidéreo medio de Greenwich (*Greenwich Mean Sidereal Time*, GMST). La fórmula que lo define, en radianes, es:

$$\gamma(t) = \gamma_0(t) + \omega_{\mathrm{E}} \cdot \Delta t_{\mathrm{e}}, \tag{2.3}$$

donde $\gamma_0(t)$ es el tiempo sidéreo medio de Greenwich a las 0 horas del día asociado a t, ω_{E} es la velocidad de rotación terrestre y Δt_{e} es el número de segundos solares desde las 0 horas del día asociado a t.

Debido a que el período de rotación terrestre referido al Sol (período solar, 24 horas) es distinto del período de rotación terrestre referido a las estrellas fijas (período sidéreo, 23 horas, 56 minutos y 4.0916 segundos), el tiempo sidéreo de Greenwich a las 0 horas no es constante, sino que presenta cierta deriva, y esta es modelada a través de la siguiente expresión polinómica:[1]

$$\begin{aligned}\gamma_0(t) = \frac{2\pi \text{ rad}}{86\,400\,\mathrm{s}} \Big[&(24\,110.548\,41\,\mathrm{s}) + (8\,640\,184.812\,866\,\mathrm{s})D \; + \\ &+(0.093\,104\,\mathrm{s})D^2 - (6.2 \times 10^{-6}\,\mathrm{s})D^3\Big],\end{aligned}$$

[1]Fórmula adaptada de la web Navipedia, proyecto impulsado por la Agencia Espacial Europea (*European Space Agency*, ESA) para el desarrollo de un enciclopedia especializada en contenidos sobre sistemas de navegación: http://www.navipedia.net/index.php/CEP_to_ITRF. Consultado el 15 de agosto de 2015.

donde D denota el número de centenares de siglos transcurridos desde el día juliano de referencia J2000 (el 1 de enero de 2000 a las 12:00:00 UT1), es decir:

$$D = \frac{\text{fecha}(t) - \text{J2000}}{36525}.$$

Nótese que como el día juliano de referencia se define a las 12:00:00 UT1, el numerador de D siempre es un número decimal cuya parte fraccionaria es 0.5.

Conocido el valor del GMST para el instante de tiempo t de interés, la conversión de posiciones del sistema ECI al sistema ECEF viene dada por:

$$\begin{bmatrix} x_{\mathrm{e}}(t) \\ y_{\mathrm{e}}(t) \\ z_{\mathrm{e}}(t) \end{bmatrix} = \begin{bmatrix} \cos\gamma(t) & \sin\gamma(t) & 0 \\ -\sin\gamma(t) & \cos\gamma(t) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_{\mathrm{i}}(t) \\ y_{\mathrm{i}}(t) \\ z_{\mathrm{i}}(t) \end{bmatrix} \tag{2.4}$$

La conversión del vector velocidad del sistema ECI al sistema ECEF requiere, no obstante, de una operación previa, consecuencia del carácter no inercial rotatorio del sistema ECEF. Primero, se calcula la velocidad $\vec{v}_{\mathrm{i}}{}'$ vista desde el sistema ECEF como:

$$\vec{v}_{\mathrm{i}}{}' = \vec{v}_{\mathrm{i}} - \vec{\omega}_{\mathrm{E}} \times \vec{r}_i \quad \Longrightarrow \quad \begin{cases} \dot{x}_{\mathrm{i}}' = \dot{x}_{\mathrm{i}} + \omega_{\mathrm{E}} \cdot y_{\mathrm{i}}, \\ \dot{y}_{\mathrm{i}}' = \dot{y}_{\mathrm{i}} - \omega_{\mathrm{E}} \cdot x_{\mathrm{i}}, \\ \dot{z}_{\mathrm{i}}' = \dot{z}_{\mathrm{i}}. \end{cases}$$

Este vector $\vec{v}_{\mathrm{i}}{}'$ es el que debe ser rotado del mismo modo que el vector posición $\vec{r}_{\mathrm{i}}$:

$$\begin{bmatrix} \dot{x}_{\mathrm{e}}(t) \\ \dot{y}_{\mathrm{e}}(t) \\ \dot{z}_{\mathrm{e}}(t) \end{bmatrix} = \begin{bmatrix} \cos\gamma(t) & \sin\gamma(t) & 0 \\ -\sin\gamma(t) & \cos\gamma(t) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \dot{x}_{\mathrm{i}}(t) + \omega_{\mathrm{E}} \cdot y_{\mathrm{i}}(t) \\ \dot{y}_{\mathrm{i}}(t) - \omega_{\mathrm{E}} \cdot x_{\mathrm{i}}(t) \\ \dot{z}_{\mathrm{i}}(t) \end{bmatrix} \tag{2.5}$$

El sistema de referencia terrestre fijo permite en adelante trabajar asumiendo una esfera terrestre fija en la que aparecen los comportamientos asociados a la elección de un sistema no inercial, y sirve de puente entre las coordenadas proporcionadas por los modelos orbitales y las coordenadas latitud-longitud-altura, necesarias para posicionar los puntos con respecto a la superficie terrestre.

2.2.3. Sistema de referencia terrestre geodésico

El sistema de referencia terrestre geodésico (GEO) es el sistema de coordenadas latitud-longitud-altura sobre el elipsoide. Es un sistema de gran interés porque permite posicionar objetos en relación con la superficie del elipsoide terrestre. Asimismo, permite calcular distancias medidas sobre esta superficie. Para un cierto punto P, las coordenadas geodésicas se definen del siguiente modo:

- La latitud φ es el ángulo entre la recta normal al elipsoide que pasa por el punto P y el plano ecuatorial. Su rango es $[-90°, +90°]$, con origen en el Ecuador terrestre y signo positivo hacia el norte.
- La longitud λ es el ángulo que forma el plano que contiene al meridiano de Greenwich con el plano que contiene al meridiano en el que se encuentra el punto P. Su rango es $(-180°, +180°]$, con origen en el meridiano de Greenwich y signo positivo en sentido este.
- La altura h es la distancia entre el punto P y la superficie del elipsoide terrestre en la dirección normal a este. Es positiva si el punto P está por encima del elipsoide.

La relación entre las coordenadas ECEF y las coordenadas GEO solo permite despejar de forma analítica la longitud λ en función de las coordenadas ECEF, puesto que la latitud φ y la altura h se encuentran acopladas [16, pp. 12−14]:

$$\lambda = \arctan2(y_\mathrm{e}, x_\mathrm{e}). \tag{2.6}$$

En el caso de la latitud φ y la altura h se tienen las siguientes expresiones:

$$N = \frac{a_\mathrm{E}}{\sqrt{1 - e_\mathrm{E}^2 \sin^2(\varphi)}}, \tag{2.7}$$

$$h = \frac{p}{\cos\varphi} - N, \tag{2.8}$$

$$\sin\varphi = \frac{z_\mathrm{e}}{N(1 - e_\mathrm{E}^2) + h}, \tag{2.9}$$

$$\varphi = \arctan\left(\frac{z_\mathrm{e} + e_\mathrm{E}^2 \cdot N \cdot \sin\varphi}{p}\right). \tag{2.10}$$

donde a_{E} es el semieje mayor del elipsoide terrestre (en este caso, el modelo WGS84), e_{E} es su excentricidad y $p = \sqrt{x_{\mathrm{e}}^2 + y_{\mathrm{e}}^2}$. Para hallar φ y h debe emplearse un método reiterativo convergente. Para ello:

1. Se inicia el proceso con los valores $N = a_{\mathrm{E}}$ y $h = 0$.
2. Se calcula $\sin\varphi$ y a continuación la latitud φ.
3. La nueva latitud permite calcular nuevos valores de N y de h.
4. Se vuelve al paso 2 y se continúa el bucle hasta que los valores converjan con una tolerancia preestablecida.

La conversión de coordenadas GEO a coordenadas ECEF es más sencilla, ya que la transformación es analítica. Dadas unas coordenadas GEO (φ, λ, h), las coordenadas ECEF correspondientes [6, p. 647] son:

$$x_{\mathrm{e}} = (N + h) \cdot \cos\varphi \cdot \cos\lambda, \tag{2.11}$$

$$y_{\mathrm{e}} = (N + h) \cdot \cos\varphi \cdot \sin\lambda, \tag{2.12}$$

$$z_{\mathrm{e}} = [N(1 - e_{\mathrm{E}}^2) + h] \cdot \sin\varphi, \tag{2.13}$$

donde e_{E} es la excentricidad del elipsoide terrestre y el parámetro N es el mismo que se definió en la conversión de coordenadas ECEF a coordenadas GEO.

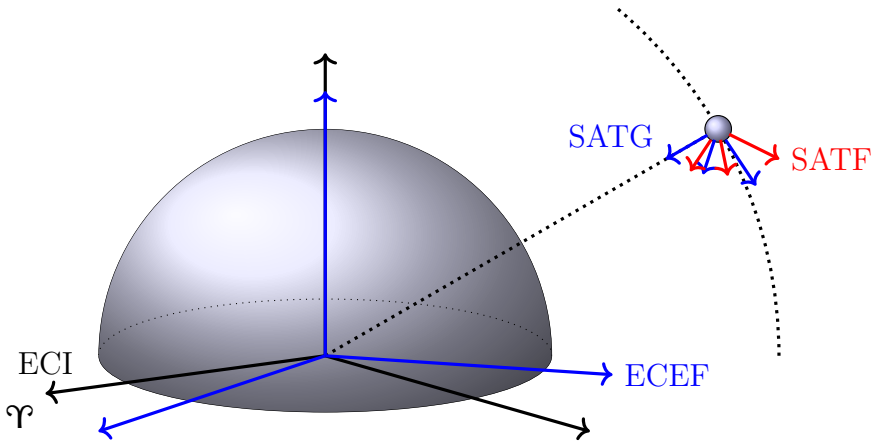

Figura 2.2: Representación de los sistemas ECI, ECEF, SATG y SATF.

2.2.4. Sistema de referencia satelital geodésico

En el marco del posicionamiento satelital resulta de interés disponer de sistemas de referencia con centro en el propio satélite y unos ejes con orientación favorable. Así, el sistema de referencia satelital geodésico (SATG) es un sistema con centro en el satélite y cuyos ejes son tales que:

- El eje X se encuentra dirigido en la dirección perpendicular a la superficie elipsoidal terrestre, con sentido positivo del satélite a la superficie.
- El eje Y es paralelo al vector velocidad $\vec{v}_\mathrm{s}$ del satélite y con sentido positivo en la dirección opuesta al desplazamiento del satélite.
- El eje Z se define de manera que los ejes (X, Y, Z) formen un triedro directo.

De este modo, la orientación esperada de un satélite es tal que la dirección de desplazamiento es la del eje $-$Y, y el plano XZ es aquel en que el sensor a bordo del satélite escanea las líneas de imagen.

La conversión entre vectores ECEF y vectores SATG se realiza a través de una matriz de rotación formada por los tres vectores unitarios ($\vec{w}_x$, $\vec{w}_y$, $\vec{w}_z$) del sistema de referencia SATG. Dado un satélite con posición $\vec{r}_\mathrm{s}$ y velocidad $\vec{v}_\mathrm{s}$ en el sistema ECEF de vectores unitarios ($\vec{u}_x$, $\vec{u}_y$, $\vec{u}_z$), los vectores unitarios del sistema SATG son [6, p. 648]:

$$\begin{aligned}
\vec{w}_x &= -\cos\varphi_\mathrm{s}\cdot\cos\lambda_\mathrm{s}\,\vec{u}_x - \cos\varphi_\mathrm{s}\cdot\sin\lambda_\mathrm{s}\,\vec{u}_y - \sin\varphi_\mathrm{s}\,\vec{u}_z,\\
\vec{w}_z &= (\vec{v}_\mathrm{s}\times\vec{w}_x)/|\vec{v}_\mathrm{s}\times\vec{w}_x|,\\
\vec{w}_y &= \vec{w}_z\times\vec{w}_x.
\end{aligned}$$

Por lo tanto, la conversión de un vector $\vec{r}_\mathrm{e}$ del sistema ECEF en un vector $\vec{r}_\mathrm{sg}$ del sistema SATG se realiza a través de la matriz de rotación T:

$$\vec{r}_\mathrm{sg} = T\,\vec{r}_\mathrm{e}, \tag{2.14}$$

donde T es una matriz 3×3 cuyas filas son las componentes de los tres vectores unitarios del sistema SATG:

$$T = [\vec{w}_x \;\; \vec{w}_y \;\; \vec{w}_z]^\intercal\,. \tag{2.15}$$

2.2.5. Sistema de referencia satelital fijo

El sistema de referencia descrito en la subsección previa es una situación ideal de la postura real del satélite y el sensor que se encuentran orbitando. Aunque el objetivo de los sistemas estabilizadores de los satélites es mantener los ejes principales del satélite en la misma disposición que los ejes del sistema de referencia satelital geodésico, existen pequeñas discrepancias fácilmente modelables. El sistema solidario con los ejes principales del satélite es el sistema de referencia satelital fijo (SATF).

La conversión de un vector $\vec{r}_{\mathrm{sg}}$ del sistema SATG en un vector $\vec{r}_{\mathrm{sf}}$ del sistema SATF se realiza a través de tres sucesivas rotaciones: la primera sobre el eje X SATG (*yaw*, ξ_{y}); a continuación se sucede una rotación sobre el nuevo eje Y (*roll*, ξ_{r}); finalmente, se realiza una rotación sobre el nuevo eje Z (*pitch*, ξ_{p}). En definitiva:

$$\vec{r}_{\mathrm{sf}} = A\,\vec{r}_{\mathrm{sg}}, \tag{2.16}$$

donde A es una matriz 3×3 producto de tres matrices de rotación:

$$A = \begin{bmatrix} \cos\xi_{\mathrm{p}} & \sin\xi_{\mathrm{p}} & 0 \\ -\sin\xi_{\mathrm{p}} & \cos\xi_{\mathrm{p}} & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos\xi_{\mathrm{r}} & 0 & -\sin\xi_{\mathrm{r}} \\ 0 & 1 & 0 \\ \sin\xi_{\mathrm{r}} & 0 & \cos\xi_{\mathrm{r}} \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\xi_{\mathrm{y}} & \sin\xi_{\mathrm{y}} \\ 0 & -\sin\xi_{\mathrm{y}} & \cos\xi_{\mathrm{y}} \end{bmatrix}. \tag{2.17}$$

Estas tres correcciones en postura son pequeñas, es decir, sobre los ángulos ξ_{y}, ξ_{r} y ξ_{p} puede realizarse la siguiente aproximación para ángulos muy pequeños:

$$\sin\theta \approx \theta, \qquad \cos\theta \approx 1, \qquad \forall\,\theta \ll 0.02\,\mathrm{rad}.$$

Esta aproximación simplifica considerablemente la matriz de rotación A:

$$A = \begin{bmatrix} 1 & \xi_{\mathrm{p}} & -\xi_{\mathrm{r}} \\ -\xi_{\mathrm{p}} & 1 & \xi_{\mathrm{y}} \\ \xi_{\mathrm{r}} & -\xi_{\mathrm{y}} & 1 \end{bmatrix}. \tag{2.18}$$

Nótese que la matriz A es antisimétrica y su inversa coincide con su traspuesta:

$$A^{\intercal} = -A, \qquad A^{-1} = A^{\intercal}. \tag{2.19}$$

2.3. Modelo orbital

El modelo orbital se basa en las leyes de Kepler [17, cap. 11] y asume que la órbita del satélite es elíptica. Esta órbita queda caracterizada por los parámetros keplerianos, que permiten conocer las coordenadas ECI del satélite y, en consecuencia, en cualquiera de los sistemas de referencia descritos en la sección anterior. En ausencia de parámetros keplerianos instantáneos, es posible obtenerlos de forma indirecta por propagación lineal temporal de los parámetros de un instante de tiempo próximo al de computación, siempre que la diferencia temporal no sea excesiva (menos de 3 días [7, p. 2]).

2.3.1. Parámetros de Kepler del satélite

Los parámetros de Kepler permiten describir la órbita elíptica de un satélite y su posición en el espacio. Estos son:

- El instante de tiempo t para el que se definen los parámetros de Kepler.
- La excentricidad de la órbita elíptica e.
- La anomalía media M, que es el ángulo que formaría el satélite con el eje de la elipse si su órbita fuera circular uniforme con radio el semieje mayor de la órbita.
- El movimiento medio n o promedio de la velocidad angular orbital del satélite.
- La longitud del nodo ascendente Ω, que es el ángulo, medido en el plano ecuatorial, entre el eje X del sistema ECI y la dirección hacia el nodo ascendente en la línea de los nodos (es decir, la línea intersección del plano orbital y el plano ecuatorial).
- El argumento del perigeo ω, que es el ángulo, medido en el plano orbital, entre la línea de los nodos y la línea de los ápsides (es decir, la línea que pasa por el centro de la órbita y por su perigeo).
- La inclinación de la órbita i, o sea, el ángulo que forman los planos orbital y ecuatorial terrestre. En satélites heliosíncronos es generalmente próxima a 99 grados.

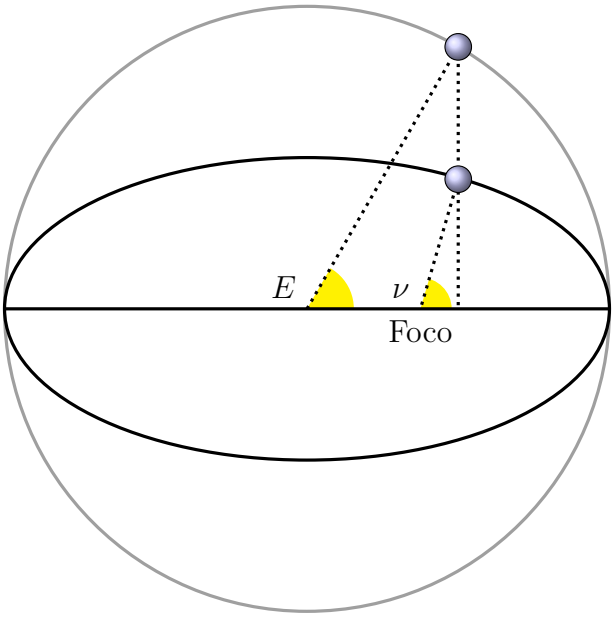

Figura 2.3: Representación del problema de Kepler.

De estos siete parámetros, el primero es la referencia temporal del resto de variables, los tres siguientes (e, M, n) permiten localizar el satélite en la órbita, y los tres últimos (Ω, ω, i) orientan la elipse respecto a una posición terrestre [7, p. 2].

2.3.2. Obtención de coordenadas ECI del satélite

Los parámetros de Kepler permiten calcular otros parámetros derivados y que son de interés en la resolución de la posición del satélite:

- El semieje mayor a de la órbita elíptica, que se obtiene por aplicación de la tercera ley de Kepler:

$$a = (\mu/n^2)^{1/3}. \tag{2.20}$$

- La anomalía excéntrica E es el ángulo, medido en el plano orbital, entre el eje de la elipse y la proyección del satélite sobre una circunferencia con centro el de la órbita y radio el semieje mayor a. Se obtiene a partir de la anomalía media M y la excentricidad de la órbita e mediante la ecuación de Kepler:

$$M = E - e \sin E. \tag{2.21}$$

Su resolución debe ser recursiva, tomando como valor inicial $E_0 = M$ y calculando el valor k-ésimo como $E_k = M + e \sin E_{k-1}$ hasta que el proceso converja con una tolerancia preestablecida.

- La anomalía verdadera ν es el ángulo, medido en el plano orbital, entre la línea de los ápsides y la dirección definida por el centro de la Tierra y el satélite. Se obtiene en función de la excentricidad y la anomalía excéntrica:

$$\nu = 2 \arctan \left[\sqrt{\frac{1+e}{1-e}} \cdot \tan \left(\frac{E}{2} \right) \right]. \tag{2.22}$$

Una vez que se determinan los parámetros de Kepler derivados, se dispone de toda la información necesaria para calcular las coordenadas del satélite. La distancia r_i del satélite se escribe en función del semieje mayor de la órbita a, la excentricidad e y la anomalía verdadera ν como:

$$r_\mathrm{i} = \frac{a(1-e^2)}{1 + e\cos\nu} = \frac{a \cdot p}{1 + e\cos\nu}.$$

Con esta distancia y los parámetros de Kepler, las coordenadas ECI del satélite son:

$$x_\mathrm{i} = r_\mathrm{i} \left[\cos\Omega \cos(\omega + \nu) - \sin\Omega \sin(\omega + \nu) \cos i \right], \tag{2.23}$$

$$y_\mathrm{i} = r_\mathrm{i} \left[\sin\Omega \cos(\omega + \nu) + \cos\Omega \sin(\omega + \nu) \cos i \right], \tag{2.24}$$

$$z_\mathrm{i} = r_\mathrm{i} \left[\sin(\omega + \nu) \sin i \right], \tag{2.25}$$

y, además, las componentes de la velocidad ECI del satélite en ese mismo instante son:

$$\dot{x}_\mathrm{i} = \frac{x_\mathrm{i}\, l\, e}{r_\mathrm{i}\, p} \sin\nu - \frac{l}{r_\mathrm{s}} \left[\cos\Omega \sin(\omega + \nu) + \sin\Omega \cos(\omega + \nu) \cos i \right], \tag{2.26}$$

$$\dot{y}_\mathrm{i} = \frac{y_\mathrm{i}\, l\, e}{r_\mathrm{i}\, p} \sin\nu - \frac{l}{r_\mathrm{s}} \left[\sin\Omega \sin(\omega + \nu) - \cos\Omega \cos(\omega + \nu) \cos i \right], \tag{2.27}$$

$$\dot{z}_\mathrm{i} = \frac{z_\mathrm{i}\, l\, e}{r_\mathrm{i}\, p} \sin\nu + \frac{l}{r_\mathrm{s}} \cos(\omega + \nu) \sin i, \tag{2.28}$$

donde $p = (1 - e^2)$ y $l = \sqrt{\mu p}$. Conocidas la posición ECI $\vec{r}_\mathrm{i}$ y la velocidad ECI $\vec{v}_\mathrm{i}$, el problema de localización del satélite queda resuelto, ya que se dispone de las expresiones de transformación a los sistemas ECEF y GEO.

2.3.3. Propagación de los parámetros de Kepler

Desde el punto de vista práctico, los parámetros keplerianos de la órbita del satélite pueden obtenerse mediante un modelo de propagación basado en los parámetros de un instante de referencia, denominado época, t_0.

Cada uno de los valores instantáneos de estos parámetros keplerianos es combinación de un término medio (el comportamiento a largo plazo) y de un término osculatorio (el comportamiento a corto plazo) [6, pp. 646−647], pero en este trabajo se utiliza una simplificación en la que solo se tienen en cuenta los términos medios con propagación temporal de primer orden:

$$\begin{aligned} t &= t_0 + \Delta t, & \Omega &= \Omega_0 + \dot{\Omega}\,\Delta t, \\ e &= e_0, & \omega &= \omega_0 + \dot{\omega}\,\Delta t, \\ M &= M_0 + n\,\Delta t, & i &= i_0, \\ n &= n_0 + \dot{n}\,\Delta t, & a &= (\mu/n^2)^{1/3}, \end{aligned}$$

donde $\Delta t = t - t_0$ es el intervalo de tiempo entre la época y el instante de estudio.

Las épocas para los satélites, con sus correspondientes parámetros, son distribuidas en forma de boletines con alta periodicidad a través de Internet. Uno de los boletines más populares son los NORAD *two-line element sets* (TLE) proporcionados por CelesTrak[2] y que son las variables de entrada del modelo orbital SGP4. Un TLE está formado por una estructura fija de tres líneas: la primera de 24 caracteres y las dos restantes de 69 caracteres, tal como aparece a continuación [18]:

```
AAAAAAAAAAAAAAAAAAAAAAA
1 BBBBBC DDDDDEEE FFFFF.FFFFFFFF -.GGGGGGGG -HHHHH-H -IIIII-I 0 JJJJK
2 BBBBB LLL.LLLL MMM.MMMM NNNNNNN OOO.OOOO PPP.PPPP QQ.QQQQQQQQRRRRRS
```

La primera línea contiene el nombre del satélite. El resto de términos corresponde a los siguientes parámetros:

[2]Enlace web: https://celestrak.com/

- `BBBBB`: número del satélite.
- `C`: clasificación del satélite.
- `DDDDD`: año y fecha de lanzamiento.
- `EEE`: pieza de lanzamiento.
- `FFFFF.FFFFFFFF`: año y día juliano decimal de la época.
- `-.GGGGGGGG`: un medio de la primera derivada del movimiento medio en revoluciones por día al cuadrado.
- `-HHHHH-H`: un sexto de la segunda derivada del movimiento medio en revoluciones por día al cubo (se asume el punto decimal).
- `-IIIII-I`: término de radiación BSTAR (se asume el punto decimal).
- `JJJ`: número de TLE (se incrementa en una unidad cada vez que se genera un nuevo TLE).
- `K`: *checksum* módulo 10 de la línea 2.
- `LLL.LLLL`: inclinación en grados.
- `MMM.MMMM`: longitud del nodo ascendente en grados.
- `NNNNNNN`: excentricidad de la órbita (se asume el punto decimal).
- `OOO.OOOO`: argumento del perigeo en grados.
- `PPP.PPPP`: anomalía media en grados.
- `QQ.QQQQQQQQ`: movimiento medio en revoluciones por día.
- `RRRRR`: número de revolución en la época.
- `S`: *checksum* módulo 10 de la línea 3.

Por ejemplo, el TLE de MetOp-B del 1 de marzo de 2015 tiene la siguiente estructura:

```
METOP-B
1 38771U 12049A   15060.09038281  .00000104  00000-0  67552-4 0  9999
2 38771  98.7116 121.5278 0001914 123.9555 292.2003 14.21473887127060
```

Las primeras derivadas de la longitud del nodo ascendente y del argumento del perigeo se obtienen a partir de estos parámetros keplerianos [19]:

$$\dot{\Omega} = -2\pi \cdot \frac{3\, n\, J_2}{\mu\, p^2} \left(\frac{1}{2} \cos i \right), \tag{2.29}$$

$$\dot{\omega} = -2\pi \cdot \frac{3\, n\, J_2}{\mu\, p^2} \left(\frac{5}{4} \sin^2 i - 1 \right), \tag{2.30}$$

donde $J_2 = 1.7555 \times 10^{25}\,\mathrm{m}^5\,\mathrm{s}^{-2}$ es el coeficiente del segundo término zonal, que está relacionado con el carácter oblato de la Tierra. Con estas dos últimas expresiones ya se dispone de todas las variables necesarias para propagar a un tiempo t los parámetros keplerianos de una época t_0.

2.4. Modelo de estimación de coordenadas

La estimación de las coordenadas de una imagen de satélite puede realizarse desde dos puntos de vista diferentes. Por una parte, se encuentra la navegación directa, es decir, a partir de los parámetros internos del sensor y de los parámetros orbitales del satélite se asignan a cada píxel las coordenadas geográficas del punto de intersección de la superficie terrestre con la recta que pasa por el punto de vista equivalente del sensor y el centro del píxel. Por otra parte, se encuentra la navegación indirecta, que sigue un razonamiento opuesto: partiendo de una malla de coordenadas geográficas se estima cuál es el píxel más próximo en la imagen para cada una de esas posiciones.

En este trabajo se opta por el proceso de navegación directa, puesto que las imágenes originales AVHRR/3 obtenidas de los servidores CLASS-NOAA cuentan en su interior con una batería de puntos de control para la imagen sin proyectar.

2.4.1. Navegación directa

El proceso de navegación directa requiere saber en cada píxel de la imagen el instante de tiempo t en que se tomó la medida y el ángulo de escaneo δ del sensor. El primero proporciona en última instancia las coordenadas ECEF del satélite en el momento de lectura del píxel; el segundo indica la orientación del sensor al mirar hacia la Tierra.

El ángulo de escaneo δ se mide en el sistema de referencia SATF y se asume que el proceso de escaneado tiene lugar en el plano XZ del sistema SATF. El ángulo δ se toma positivo en el sentido positivo del eje Z del sistema SATF, que corresponde con la mitad izquierda de la imagen observada en la dirección de movimiento del satélite.

Se denomina vector de observación $\vec{d}$ al vector que se encuentra sobre la línea de observación del sensor AVHRR/3 con origen en el satélite y final sobre la superficie terrestre. Las componentes de este vector, en el sistema SATF, son:

$$\vec{d}_{\mathrm{sf}} = d\,\vec{u}_{\mathrm{sf}} = d \begin{bmatrix} \cos\delta \\ 0 \\ \sin\delta \end{bmatrix}, \tag{2.31}$$

donde d es la magnitud del vector de observación y $\vec{u}_{\mathrm{sf}}$ es el vector unitario de observación en el sistema SATF. Este vector $\vec{d}_{\mathrm{sf}}$ necesita ser expresado en el sistema ECEF para que se encuentre en el mismo sistema de referencia que las coordenadas de posición del satélite. Para ello, se aplican las matrices de rotación traspuestas de A (conversión de SATF a SATG) y de T (conversión de SATG a ECEF) al vector unitario $\vec{u}_{\mathrm{sf}}$:

$$\vec{u}_{\mathrm{e}} = T^{\intercal}\,A^{\intercal}\,\vec{u}_{\mathrm{sf}} = T^{\intercal}\,A^{\intercal} \begin{bmatrix} \cos\delta \\ 0 \\ \sin\delta \end{bmatrix}. \tag{2.32}$$

De este modo, $\vec{u}_{\mathrm{e}}$ es el vector unitario de observación en el sistema ECEF. Como este vector solamente ha sufrido transformaciones sin cambio de escala, el módulo del vector de observación es el mismo en los sistemas SATF y ECEF:

$$\vec{d}_{\mathrm{e}} = d\,\vec{u}_{\mathrm{e}}. \tag{2.33}$$

En el caso de que los ángulos de postura sean desconocidos, se asume que la matriz de rotación A es la identidad (los sistemas SATF y SATG coinciden) y los ángulos de postura tendrán que ser extraídos mediante un posprocesado de la imagen.

Sea $\vec{r}_e$ el vector posición del satélite en el sistema ECEF en el instante de captura y $\vec{d}_e$ el vector de observación para un determinado píxel. El vector posición $\vec{R}_e$ del punto capturado en la superficie terrestre para ese píxel cumple la condición de coplanaridad:

$$\vec{R}_e = \vec{r}_e + \vec{d}_e = \vec{r}_e + d\,\vec{u}_e. \tag{2.34}$$

En esta ecuación, los vectores $\vec{r}_e$ y $\vec{u}_e$ son conocidos, y el vector $\vec{R}_e$ y la magnitud d son las incógnitas a resolver. Para despejar la magnitud d, es necesario aplicar la ley de los cosenos sobre el triángulo que forman los vectores $\vec{R}_e$, $\vec{r}_e$ y $\vec{d}_e$ (Figura 2.4):

$$R_e^2 = r_e^2 + d_e^2 - 2r_e d_e \cos\alpha = r_e^2 + d_e^2 - 2r_e d_e(-\cos\beta) = r_e^2 + d_e^2 + 2(r_e \cdot \vec{u}_e)d_e,$$

donde se ha tenido en cuenta que $\beta = \pi - \alpha$. Si se traslada r_e^2 al miembro de la izquierda y se intercambian los dos lados de ecuación:

$$d_e^2 + 2(r_e \cdot \vec{u}_e)d_e = R_e^2 - r_e^2.$$

A continuación, se suma $(r_e \cdot \vec{u}_e)^2$ en los dos miembros:

$$d_e^2 + 2(r_e \cdot \vec{u}_e)d_e + (r_e \cdot \vec{u}_e)^2 = R_e^2 - r_e^2 + (r_e \cdot \vec{u}_e)^2.$$

El lado izquierdo de la ecuación es el desarrollo de un binomio de Newton, luego:

$$[d_e + (r_e \cdot \vec{u}_e)]^2 = R_e^2 - r_e^2 + (r_e \cdot \vec{u}_e)^2.$$

Finalmente, tomando raíces y despejando d_e se llega a la expresión deseada [6, p. 649]:

$$d_e = -(r_e \cdot \vec{u}_e) + \sqrt{R_e^2 - r_e^2 + (r_e \cdot \vec{u}_e)^2}. \tag{2.35}$$

Esta expresión depende de la magnitud R del vector posición del punto sobre la superficie terrestre. Sin embargo, un método recursivo convergente formado por las ecuaciones (2.34) y (2.35) puede ser implementado con facilidad, tomando como valor inicial de R el semieje mayor terrestre $R = a_E$ y procediendo del siguiente modo:

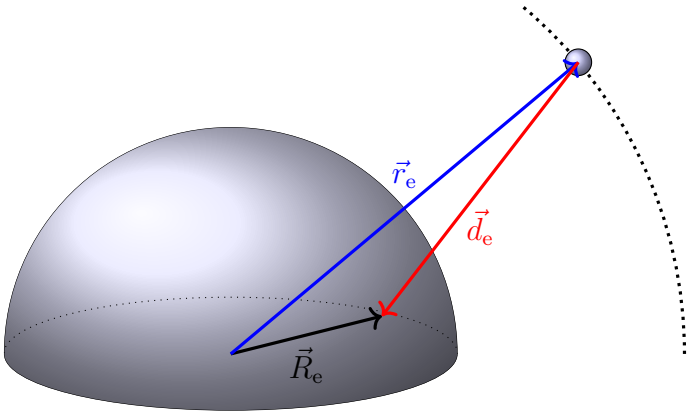

Figura 2.4: Condición de coplanaridad de los vectores $\vec{R}_e$, $\vec{r}_e$ y $\vec{d}_e$.

1. Determinar la posición y velocidad ECEF del satélite para el instante de tiempo en que se capturó el píxel en estudio.

2. Calcular la matriz de transformación T del sistema ECEF al sistema SATG.

3. Calcular la matriz de transformación A del sistema SATG al sistema SATF si se conocen los ángulos de postura del sensor; en caso contrario, asumir que A es la matriz identidad.

4. Determinar el vector unitario de observación $\vec{u}_{sf}$ para el píxel en estudio y transformarlo al sistema ECEF, $\vec{u}_e$.

5. Para un cierto valor de R (si aún no se tiene ningún valor, tomar $R = a_E$), computar la magnitud d del vector de observación con la ecuación (2.35).

6. Determinar el vector posición $\vec{R}$ utilizando la ecuación (2.34).

7. Calcular el radio R_E del elipsoide de referencia en la latitud φ asociada al vector posición $\vec{R}$ calculado previamente, según la expresión siguiente:

$$R_E(\varphi) = \sqrt{\frac{(a_E^2 \cos\varphi)^2 + (b_E^2 \sin\varphi)^2}{(a_E \cos\varphi)^2 + (b_E \sin\varphi)^2}}. \tag{2.36}$$

8. Asignar a la magnitud R el valor R_E y volver al paso 5 hasta que el proceso converja con una tolerancia preestablecida.

2.4.2. Corrección del desfase del reloj

La georreferenciación por navegación directa de imágenes del sensor AVHRR puede llevar asociados distintos errores de registrado que deben ser corregidos. Uno de los problemas más comunes es el desfase del reloj a bordo de los sensores AVHRR, imprecisión confirmada por el Departamento de Navegación de NESDIS/NOAA en el año 2000 [7, p. 8]. Dado que cada línea de imagen se captura en 167 ms, un error de 1 s en el reloj es un error de desplazamiento de 6 km en la imagen sobre la superficie terrestre.

La corrección del desfase debe realizarse con ayuda de uno o varios puntos de control. Supóngase un punto de control de vector posición $\vec{R}'_{\mathrm{e}}$ en el sistema ECEF asociado a un píxel de vector posición calculado $\vec{R}'_{\mathrm{e}}$. Si el desplazamiento del píxel respecto a sus coordenadas ideales es pequeño, el error de posición en la dirección orbital se halla como proyección de la diferencia de vectores posición sobre el vector $-\vec{w}_y$ del sistema SATG:

$$\delta r = -\delta\vec{R}_{\mathrm{e}} \cdot \vec{w}_y = -(\vec{R}_{\mathrm{e}} - \vec{R}'_{\mathrm{e}}) \cdot \vec{w}_y. \tag{2.37}$$

Asimismo, si la velocidad del satélite en el sistema ECEF es $\vec{v}_{\mathrm{e}}$, su proyección v en la dirección de desplazamiento de la órbita es:

$$v = -\vec{v}_{\mathrm{e}} \cdot \vec{w}_y, \tag{2.38}$$

y asumiendo un desplazamiento uniforme en ese lapso de tiempo, se puede obtener una estimación del desfase δt asociado al error en posición del punto de control al calcularlo con el modelo:

$$v \cdot \delta t = \delta r \qquad \Longrightarrow \qquad \delta t = \frac{\delta r}{v}. \tag{2.39}$$

El desfase temporal δt es positivo si el reloj del satélite está adelantado, y es negativo si está atrasado. En el caso de disponer de diferentes puntos de control, puede plantearse la misma expresión en forma de sistema de ecuaciones sobredimensionado:

$$[v_k] \cdot \delta t = [\delta r_k] \qquad \Longrightarrow \qquad G \cdot \delta t = L. \tag{2.40}$$

La solución óptima para reducir el error cuadrático medio es:

$$\delta t = (G^{\intercal} G)^{-1} G^{\intercal} L. \tag{2.41}$$

2.4.3. Corrección de la postura del sensor

Después de la corrección de desfase del sensor, cabe la posibilidad de que algunos píxeles continúen sin estar georreferenciados correctamente en relación con los puntos de control disponibles, cuyas coordenadas se consideran exactas. Este error remanente puede tener su origen en que los sistemas de referencia SATG y SATF no estén alineados, es decir, que los ángulos de postura (ξ_r, ξ_p, ξ_y) no sean nulos.

Rosborough [6, pp. 651−652] propone para este problema un modelo de corrección de postura en el que asume que los errores de orientación son constantes para un cierto intervalo de tiempo. Brunel y Marsouin [4] concluyen que esta aproximación es cierta para los intervalos de captura de una imagen AVHRR, dado que no llegan a sobrepasar los 11 minutos.

Al igual que en la sección anterior, se asume la existencia de un punto de control de vector posición $\vec{R}'_\mathrm{e}$ en coordenadas ECEF que se corresponde con un píxel de la imagen de vector posición asociado $\vec{R}_\mathrm{e}$. La diferencia entre estos vectores origina que los vectores de observación asociados (el predicho, $\vec{d}_\mathrm{e}$, y el verdadero, $\vec{d}'_\mathrm{e}$) sean diferentes:

$$\vec{d}_\mathrm{e} = \vec{R}_\mathrm{e} - \vec{r}, \qquad \vec{d}'_\mathrm{e} = \vec{R}'_\mathrm{e} - \vec{r}_\mathrm{e} = \vec{d}_\mathrm{e} - \delta\vec{R}_\mathrm{e}, \tag{2.42}$$

donde $\vec{r}_\mathrm{e}$ es el mismo vector posición del satélite para ambos casos. El siguiente paso inmediato es normalizar el vector de observación verdadero para tener su vector unitario:

$$\vec{u}'_\mathrm{e} = \frac{\vec{d}'_\mathrm{e}}{|\vec{d}'_\mathrm{e}|}. \tag{2.43}$$

A continuación, se rota este $\vec{u}'_\mathrm{e}$ al sistema SATG a través de la matriz de rotación T, y se obtiene el vector unitario de observación en el sistema SATG, $\vec{u}'_\mathrm{sg}$:

$$\vec{u}'_\mathrm{sg} = T\,\vec{u}'_\mathrm{e}. \tag{2.44}$$

Finalmente, se rota $\vec{u}'_\mathrm{sg}$ al sistema SATF a través de la matriz de rotación A, cuyos elementos son desconocidos y están relacionados con los ángulos de postura:

$$\vec{u}'_\mathrm{sf} = A\,\vec{u}'_\mathrm{sg}. \tag{2.45}$$

Escribiendo de forma explícita la ecuación (2.45), se tiene:

$$\begin{bmatrix} \cos\delta \\ 0 \\ \sin\delta \end{bmatrix} = \begin{bmatrix} 1 & \xi_{\mathrm{p}} & -\xi_{\mathrm{r}} \\ -\xi_{\mathrm{p}} & 1 & \xi_{\mathrm{y}} \\ \xi_{\mathrm{r}} & -\xi_{\mathrm{y}} & 1 \end{bmatrix} \cdot \begin{bmatrix} (u_x)'_{\mathrm{sg}} \\ (u_y)'_{\mathrm{sg}} \\ (u_z)'_{\mathrm{sg}} \end{bmatrix}. \tag{2.46}$$

La relación (2.45) también puede escribirse en sentido opuesto, del sistema SATF al sistema SATG, como:

$$\vec{u}'_{\mathrm{sg}} = A^{\intercal}\, \vec{u}'_{\mathrm{sf}}, \tag{2.47}$$

que de forma explícita queda como:

$$\begin{bmatrix} (u_x)'_{\mathrm{sg}} \\ (u_y)'_{\mathrm{sg}} \\ (u_z)'_{\mathrm{sg}} \end{bmatrix} = \begin{bmatrix} 1 & -\xi_{\mathrm{p}} & \xi_{\mathrm{r}} \\ \xi_{\mathrm{p}} & 1 & -\xi_{\mathrm{y}} \\ -\xi_{\mathrm{r}} & \xi_{\mathrm{y}} & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos\delta \\ 0 \\ \sin\delta \end{bmatrix}. \tag{2.48}$$

Reorganizando esta ecuación matricial, se pueden despejar todos los ángulos de postura en un vector columna. Primero, se desarrollan las tres ecuaciones contenidas dentro de la ecuación matricial:

$$\begin{aligned} (u_x)'_{\mathrm{sg}} &= \cos\delta + \xi_{\mathrm{r}} \sin\delta, \\ (u_y)'_{\mathrm{sg}} &= \xi_{\mathrm{p}} \cos\delta - \xi_{\mathrm{y}} \sin\delta, \\ (u_z)'_{\mathrm{sg}} &= -\xi_{\mathrm{r}} \cos\delta + \sin\delta. \end{aligned} \quad \Longrightarrow \quad \begin{aligned} (u_x)'_{\mathrm{sg}} - \cos\delta &= \sin\delta \cdot \xi_{\mathrm{r}} \\ (u_y)'_{\mathrm{sg}} &= +\cos\delta \cdot \xi_{\mathrm{p}} - \sin\delta \cdot \xi_{\mathrm{y}}, \\ (u_z)'_{\mathrm{sg}} - \sin\delta &= -\cos\delta \cdot \xi_{\mathrm{r}}. \end{aligned}$$

A continuación, se reescribe la ecuación matricial:

$$\begin{bmatrix} (u_x)'_{\mathrm{sg}} - \cos\delta \\ (u_y)'_{\mathrm{sg}} \\ (u_z)'_{\mathrm{sg}} - \sin\delta \end{bmatrix} = \begin{bmatrix} \sin\delta & 0 & 0 \\ 0 & \cos\delta & -\sin\delta \\ -\cos\delta & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} \xi_{\mathrm{r}} \\ \xi_{\mathrm{p}} \\ \xi_{\mathrm{y}} \end{bmatrix}. \tag{2.49}$$

Este sistema de ecuaciones no presenta solución única si únicamente se dispone de un punto de control, puesto que puede observarse que el determinante de la matriz del sistema es siempre nulo. La obtención de los ángulos de giro puede, entonces, abordarse a través de dos vías:

- Usar m puntos de control ($m > 1$) y resolver el sistema sobredimensionado (2.49) minimizando el error cuadrático medio. La solución de este problema es:

$$\xi_{\mathrm{r}} = h_2/m, \tag{2.50}$$

$$\xi_{\mathrm{p}} = (g_3 h_1 - g_2 h_3)/(g_1 g_3 - g_2^2), \tag{2.51}$$

$$\xi_{\mathrm{y}} = (g_1 h_3 - g_2 h_1)/(g_1 g_3 - g_2^2), \tag{2.52}$$

  donde los coeficientes g_1, g_2, g_3, h_1, h_2 y h_3 se definen como sigue:

$$g_1 = +\sum_{k=1}^{m} \sin^2 \delta_k, \qquad h_1 = -\sum_{k=1}^{m} (u_y)'_{\mathrm{sg}_k} \sin \delta_k,$$

$$g_2 = -\sum_{k=1}^{m} \sin \delta_k \cos \delta_k, \qquad h_2 = +\sum_{k=1}^{m} [(u_x)'_{\mathrm{sg}_k} \sin \delta_k - (u_z)'_{\mathrm{sg}_k} \cos \delta_k]$$

$$g_3 = +\sum_{k=1}^{m} \cos^2 \delta_k, \qquad h_3 = +\sum_{k=1}^{m} (u_y)'_{\mathrm{sg}_k} \cos \delta_k.$$

- Usar un único punto de control y establecer alguna restricción adicional. Primero, se resta la primera ecuación de (2.49) multiplicada por $\sin \delta$ y la tercera ecuación de (2.49) multiplicada por $\cos \delta$. La segunda ecuación solamente es necesario expandirla:

$$\xi_{\mathrm{r}} = (u_x)'_{\mathrm{sg}} \sin \delta - (u_z)'_{\mathrm{sg}} \cos \delta, \tag{2.53}$$

$$\xi_{\mathrm{p}} \cos \delta - \xi_{\mathrm{y}} \sin \delta = (u_y)'_{\mathrm{sg}}. \tag{2.54}$$

  De este modo, el ángulo ξ_{r} ya se encuentra despejado. La restricción debe imponerse, pues, a los ángulos ξ_{p} y ξ_{y}. Una consideración aceptable es hallar los valores que minimicen ambos ángulos. Para ello, se despeja ξ_{p} en función de ξ_{y}:

$$\xi_{\mathrm{p}} = \frac{(u_y)'_{\mathrm{sg}} + \xi_{\mathrm{y}} \sin \delta}{\cos \delta}. \tag{2.55}$$

  A continuación, se define la función $f(\xi_{\mathrm{y}})$ siguiente:

$$f(\xi_{\mathrm{y}}) = \xi_{\mathrm{y}}^2 + \xi_{\mathrm{p}}^2 = \xi_{\mathrm{y}}^2 + \frac{[(u_y)'_{\mathrm{sg}} + \xi_{\mathrm{y}} \sin \delta]^2}{\cos^2 \delta}, \tag{2.56}$$

  y se exige que su primera derivada con respecto a ξ_{y} sea nula:

$$f'(\xi_{\mathrm{y}}) = 0. \tag{2.57}$$

Se calcula la derivada en cuestión:

$$f'(\xi_{\mathrm{y}}) = 2\xi_{\mathrm{y}} + \frac{2[(u_y)'_{\mathrm{sg}} + \xi_{\mathrm{y}} \sin\delta] \cdot \sin\delta}{\cos^2\delta} =$$
$$= \frac{2}{\cos^2\delta}\left[\xi_{\mathrm{y}}\cos^2\delta + (u_y)'_{\mathrm{sg}}\sin\delta + \xi_{\mathrm{y}}\sin^2\delta\right] = \frac{2}{\cos^2\delta}\left[\xi_{\mathrm{y}} + (u_y)'_{\mathrm{sg}}\sin\delta\right] = 0.$$

El mínimo se encuentra, siempre que $\cos\delta \neq 0$, en:

$$\xi_{\mathrm{y}} = -(u_y)'_{\mathrm{sg}}\sin\delta.$$

Finalmente, esta expresión de ξ_{y} es introducida en (2.55) para hallar ξ_{p}:

$$\xi_{\mathrm{p}} = \frac{(u_y)'_{\mathrm{sg}} - (u_y)'_{\mathrm{sg}}\sin^2\delta}{\cos\delta} = (u_y)'_{\mathrm{sg}}\frac{1-\sin^2\delta}{\cos\delta} = (u_y)'_{\mathrm{sg}}\frac{\cos^2\delta}{\cos\delta} = (u_y)'_{\mathrm{sg}}\cos\delta.$$

En definitiva, si solamente se dispone de un punto de control, se propone la solución siguiente para los ángulos de postura:

$$\xi_{\mathrm{r}} = (u_x)'_{\mathrm{sg}}\sin\delta - (u_z)'_{\mathrm{sg}}\cos\delta, \tag{2.58}$$
$$\xi_{\mathrm{p}} = (u_y)'_{\mathrm{sg}}\cos\delta, \tag{2.59}$$
$$\xi_{\mathrm{y}} = -(u_y)'_{\mathrm{sg}}\sin\delta. \tag{2.60}$$

2.5. Modelo de correspondencia de imágenes

En la sección previa, los procedimientos de corrección geográfica de imágenes AVHRR, tanto en el cálculo del desfase del reloj como en los ángulos de postura, necesitaban puntos de control que sirvieran de anclaje geográfico a la imagen.

La búsqueda de puntos de control parte de una imagen de referencia en la que existen ciertas regiones características con coordenadas bien definidas; esos puntos son buscados en la imagen objetivo y, a partir de las diferencias encontradas entre las coordenadas calculadas y las coordenadas ideales, se estiman los errores que deben corregirse. Esta búsqueda de puntos de control puede ser realizada de forma manual, y existe software comercial para ello (por ejemplo, Exelis ENVI[3]), pero la tarea es tediosa y no automatizada, situación no conveniente para sistemas de producción en tiempo real.

[3]Enlace web: http://www.exelisvis.com/ProductsServices/ENVIProducts/ENVI.aspx.

La alternativa a esta descripción es la automatización de la búsqueda de puntos de control, es decir, la utilización de algoritmos que determinen regiones características en una imagen de referencia y a su vez sean capaces de encontrarlas en imágenes objetivo, con independencia de la existencia de transformaciones afines o proyectivas entre ambas.

El procedimiento ordinario de correspondencia de imágenes, con independencia del algoritmo empleado, suele dividirse en tres etapas [20, p. 1]:

1. La búsqueda de puntos característicos, es decir, la presencia de ciertos puntos con unos rasgos singulares que los diferencian considerablemente en el conjunto de la imagen. Ejemplos de puntos característicos propios de imágenes de satélite son los correspondientes a cabos y golfos a lo largo de la línea de costa o los pertenecientes a lagos interiores.

2. La descripción de los puntos característicos, o sea, la especificación del punto característico a través de una función matemática que, dentro de lo posible, sea invariante frente a transformaciones afines y proyectivas, a cambios de iluminación y a la inclusión de ruido en la imagen.

3. La correspondencia de los puntos característicos en imágenes diferentes, mediante la definición de una función distancia que determine la semejanza entre las descripciones de los puntos característicos de distintas imágenes.

La primera decisión es, por lo tanto, elegir los detectores y descriptores empleados en la detección y la correspondencia de puntos característicos en imágenes de satélite. Después, es necesario elegir la función distancia que comparará las descripciones de los puntos característicos (por ejemplo, la norma L2, la norma L1 o la norma de Hamming), aunque a veces esta elección está sujeta a la propia elección del descriptor.

Los algoritmos SIFT (*Scale-Invariant Feature Transform*, 1999) y SURF (*Speeded Up Robust Features*, 2006) son detectores-descriptores que han demostrado ser efectivos en la correspondencia de imágenes: SIFT obtiene mejores resultados pero es computacionalmente más costoso, mientras que SURF busca compromiso entre tasa de corresponden-

cias y eficiencia computacional [21]. Ambos responden bien a la aplicación de transformaciones de diverso tipo sobre las imágenes, si bien el descriptor de SIFT es débil frente a cambios de iluminación [20]. Asimismo, SIFT solamente es invariante frente a cuatro de los seis parámetros que caracterizan una transformación afín, por lo que Morel y Yu desarrollaron en 2011 un algoritmo derivado de SIFT que sí es invariante a estos seis parámetros [22], y que recibe el nombre de ASIFT (*Affine SIFT*).

El hecho de que los algoritmos anteriores se encuentren bajo patentes en determinadas aplicaciones provoca que sean descartados, ya que su uso interfiere con la licencia libre de este trabajo. En su lugar, se opta por alternativas que en la fecha de publicación de este trabajo son libres:

- Se utiliza el detector de esquinas FAST (*Features from Accelerated Segment Test*), desarrollado por Rosten y Drummond [23], que es altamente eficiente en términos de computación. Dado un determinado píxel P de intensidad I_P, el detector toma un círculo de 16 píxeles alrededor de P y a continuación se toma un entorno de m píxeles contiguos a este círculo (generalmente 12). Si los m píxeles del entorno son más intensos que $I_\mathrm{P} + I_\mathrm{t}$ o bien son más oscuros que $I_\mathrm{P} - I_\mathrm{t}$, donde I_t es un umbral preestablecido por el usuario, se dice entonces que P es una esquina.

- Se emplea el descriptor del algoritmo BRISK (*Binary Robust Invariant Scalable Keypoints*). La descripción de los puntos característicos se realiza con una cadena binaria resultado de concatenar los resultados de tests sencillos de comparación de intensidad en la vecindad del punto.

- Dado el carácter binario del descriptor BRISK, la correspondencia de puntos característicos se realiza mediante la distancia de Hamming, al igual que en el descriptor BRIEF (*Binary Robust Independent Elementary Features*) [24]. Esta distancia se calcula simplemente como el número de bits diferentes entre dos descripciones, por lo que se reduce a la aplicación de operaciones XOR, que son muy eficientes en tiempo de computación.

Un análisis subjetivo de los canales disponibles en las imágenes diurnas del sensor AVHRR/3 muestra que el rango dinámico del canal 2 ($0.725-1.0\,\mu m$) es adecuado para la extracción de esquinas; por lo tanto, esta es la banda empleada en la correspondencia de imágenes.

Cabe mencionar que el detector de esquinas FAST necesita un filtrado de los puntos característicos antes de generar su descripción, ya que en las imágenes satelitales encuentra un gran número de esquinas correspondientes a nubes, y estos puntos no aportan información de valor al proceso de corrección geométrica de las imágenes AVHRR/3; de hecho, la ausencia de este filtrado puede causar correspondencias erróneas que desencadenen en un fallo completo de la cadena de corrección.

La detección de las principales nubes en las imágenes AVHRR es sencilla usando el canal térmico 5 ($11.5-12.5\,\mu m$): una simple comprobación por umbrales en las cuentas digitales (DN, *Digital Number*) permite crear una máscara de nubes básica. La condición de nube en este trabajo se obtuvo por ensayo-error con varias imágenes AVHRR/3:

$$\mathrm{DN}_5(l, c) > 500. \tag{2.61}$$

Para garantizar la ausencia de nubes durante el proceso descriptivo, la máscara de nubes generada es extendida binariamente con un buffer de 20 píxeles.

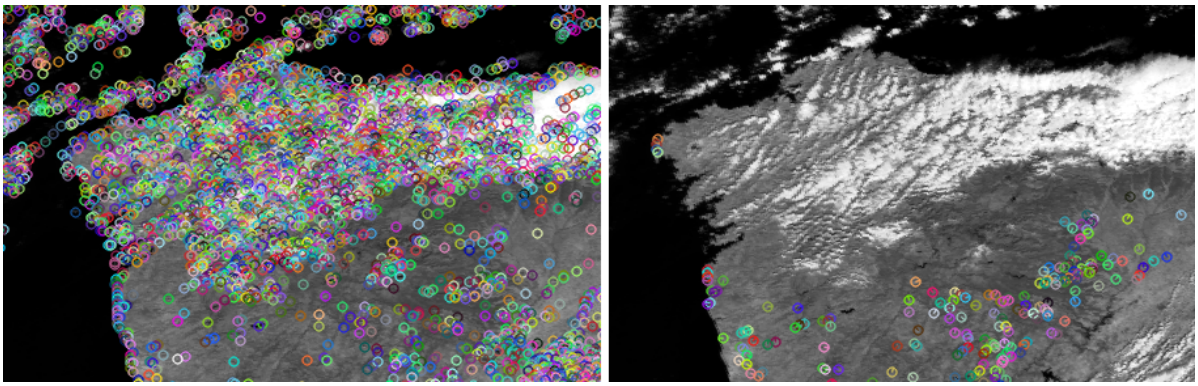

Figura 2.5: Comparación del detector de esquinas FAST antes y después de filtrar las nubes (izquierda y derecha, respectivamente).

Capítulo 3

Implementación del modelo

Una vez descrito el marco teórico del presente trabajo, el siguiente paso es materializarlo mediante su implementación y aplicación sobre imágenes reales.

En este capítulo se describe la implementación del marco teórico, es decir, el flujo de trabajo que se sigue desde la imagen original hasta su corrección geométrica final. La materialización de este flujo de trabajo se ha llevado a cabo mediante el desarrollo de una librería en Python, `python-sat`, que se apoya en varios módulos externos de Python con gran potencia en el ámbito de la cartografía y la teledetección:

- NumPy, la librería de manipulación de arrays [25].
- SciPy, la librería que contiene funciones de interés en cálculo científico, como interpoladores, funciones de convolución o manipulación de imágenes binarias [26].
- GDAL/OGR, la herramienta de edición ráster y vectorial más popular en el ámbito de código abierto.
- OpenCV, el módulo de herramientas de visión computacional, como son por ejemplo los detectores y descriptores en el marco de la correspondencia de imágenes.

Todo el código que forma la librería `python-sat` se encuentra liberado bajo la licencia general pública de GNU (GNU GPL v3.0) [27], y se encuentra disponible en el repositorio de GitHub https://github.com/molinav/python-sat.

3.1. Flujo de trabajo

En el flujo de trabajo para la corrección geométrica de imágenes AVHRR se pueden distinguir varias etapas:

1. La determinación de la posición del satélite en el momento de captura de la imagen.
2. La estimación inicial de las coordenadas de los píxeles de la imagen a partir de la geometría del problema.
3. La detección automática de puntos de control en la imagen a partir de una imagen de referencia.
4. La actualización de la estimación inicial realizada en la georreferenciación de la imagen de satélite.

3.1.1. Determinación de la posición del satélite

El primer paso que debe seguirse al corregir una imagen AVHRR es conocer cuándo fue tomada cada una de las líneas escaneadas. El formato L1B de estas imágenes es manipulable a través de los controladores de GDAL, que permiten acceder a los metadatos dentro de estos ficheros. Entre ellos se encuentran `START` y `END`, que almacenan las fechas y horas de captura, con precisión de milisegundos, de la primera y la última línea de la imagen del sensor AVHRR.

Una vez que se conocen estos dos parámetros, se puede calcular el tiempo necesario para capturar cada línea de la imagen, que deberá ser 166.667 ms, es decir, 6 líneas por segundo. Con estos datos es posible determinar la fecha y hora de captura de cada una de las m líneas de imagen como:

$$t_l = t_{\text{start}} + (166.667\,\text{ms}) \cdot l, \qquad l = 0, 1, \ldots, m. \tag{3.1}$$

El conocimiento del instante t_l asociado a la línea l-ésima permite hallar la posición del satélite en el momento de captura de dicha línea si existe una época de referencia t_0

para la cual se conocen los parámetros keplerianos. Esta época se obtiene de los NORAD TLE, disponibles en la red con periodicidad diaria. Una vez leído el TLE, la propagación orbital descrita en la subsección 2.3.3 conduce a los valores de los vectores posición $(\vec{r}_{\mathrm{e}})_l$ del satélite.

3.1.2. Estimación inicial de las coordenadas de la imagen

La estimación inicial de las coordenadas de la imagen fue descrita teóricamente en la subsección 2.4.1. Para cada línea l, existen 2048 píxeles cuyo índice de columna se denota por c y cuyo ángulo de vista δ_c es [7]:

$$\delta_c = (1024.5 - c) \cdot \delta_0, \qquad c = 0, 1, \ldots, 2047, \tag{3.2}$$

donde δ_0 es el paso angular del sensor a lo largo de los píxeles de una línea. Recuérdese que δ_c es positivo hacia el lado izquierdo de la imagen en el sentido de desplazamiento del satélite. La apertura máxima es aproximadamente 55.4 grados medida desde la vertical, por lo que una primera estimación de δ_0 es:

$$\delta_0 \approx \frac{55.4^\circ}{1024} = 0.054102^\circ. \tag{3.3}$$

Este valor δ_0 se asume constante. Del mismo modo, todos los píxeles de una misma línea l comparten el vector posición del satélite $(\vec{r}_{\mathrm{e}})_l$ y se asume que la matriz de rotación A es la identidad. Conocidos δ_c y $(\vec{r}_{\mathrm{e}})_l$, la ecuación (2.35) es resoluble para cada píxel (l, c) de la imagen y de ella se obtienen sus vectores posición $(\vec{R}_{\mathrm{e}})_{lc}$ en el sistema ECEF.

El último paso en este bloque es convertir las coordenadas de los píxeles del sistema ECEF al sistema GEO. Una buena resolución de este bloque debe conducir a coordenadas GEO con altura muy próxima a cero, puesto que el modelo proyecta los píxeles sobre la superficie del esferoide terrestre.

3.1.3. Detección de errores de georreferenciación en la imagen

Una vez alcanzado este punto, es posible proyectar la imagen AVHRR original a una ventana en coordenadas latitud-longitud mediante remuestreo por vecino más próximo

con un espaciado de píxel no inferior a la resolución espacial del sensor. En estas condiciones, la imagen proyectada no debe presentar deformaciones apreciables (este hecho es visualizable, por ejemplo, en las formas de las tierras emergentes).

No obstante, un análisis exhaustivo permite observar dos problemas. El primero de ellos es un desplazamiento en la dirección de desplazamiento del satélite (no superior a 30 km), que es modelable como un desfase del reloj a bordo (*drift*); el segundo de ellos, más sutil, es un desplazamiento en postura.

La corrección de estos errores debe hacerse mediante un método de correspondencia con una imagen de referencia cuyas coordenadas sean confiables. El objetivo es detectar puntos homólogos entre la imagen de referencia y la imagen objetivo, los cuales, empero, presentarán diferentes coordenadas geográficas $(\varphi_{lk},\, \lambda_{lk})$ y $(\varphi'_{lk},\, \lambda'_{lk})$. El procedimiento es análogo para los errores de reloj y de postura:

1. Hallar puntos característicos en las imágenes de referencia y objetivo mediante el detector FAST.

2. Filtrar los puntos característicos de manera que se elimine el mayor número de esquinas detectadas en regiones nubosas.

3. Describir los puntos característicos que superen el filtro de nubes con BRISK.

4. Calcular la distancia de Hamming entre los puntos característicos de la imagen de referencia y los de la imagen objetivo, a fin de detectar puntos comunes.

5. Recolectar las coordenadas GEO de los puntos comunes, convertirlas al sistema ECEF y computar los vectores diferencia de posición $(\delta\vec{R}_{\mathrm{e}})_{lk} = (\vec{R}_{\mathrm{e}})_{lk} - (\vec{R}'_{\mathrm{e}})_{lk}$.

   - En el caso de querer corregir el desfase del reloj δt, computar la diferencia de posición en la dirección orbital y hallar δt con la ecuación (2.41).
   - En el caso de querer corregir la postura del sensor, resolver la ecuación (2.49) según haya uno o más puntos de control, para obtener ξ_{r}, ξ_{p} y ξ_{y}. Comprobar que el rango de valores obtenido para estos ángulos es aceptable.

3.1.4. Actualización de las coordenadas de la imagen

Una vez que se han detectado los errores de desplazamiento y postura, el siguiente paso es actualizar la información orbital y del sensor, es decir:

- Recalcular los tiempos de captura de las líneas restando el desfase δt obtenido:

$$t_l = t_{\text{start}} - \delta t + (166.667\,\text{ms}) \cdot l, \qquad l = 0, 1, \ldots, m. \tag{3.4}$$

- Recalcular la matriz de rotación A a partir de su definición en (2.17).

Una vez realizadas estas modificaciones, deben repetirse todos los pasos correspondientes a las subsecciones 3.1.1, 3.1.2 y 3.1.3 (usando los nuevos t_l y A). Llegados a este punto, la imagen AVHRR debe encontrarse georreferenciada con una precisión aceptable (en el entorno de $1-2\,\text{km}$).

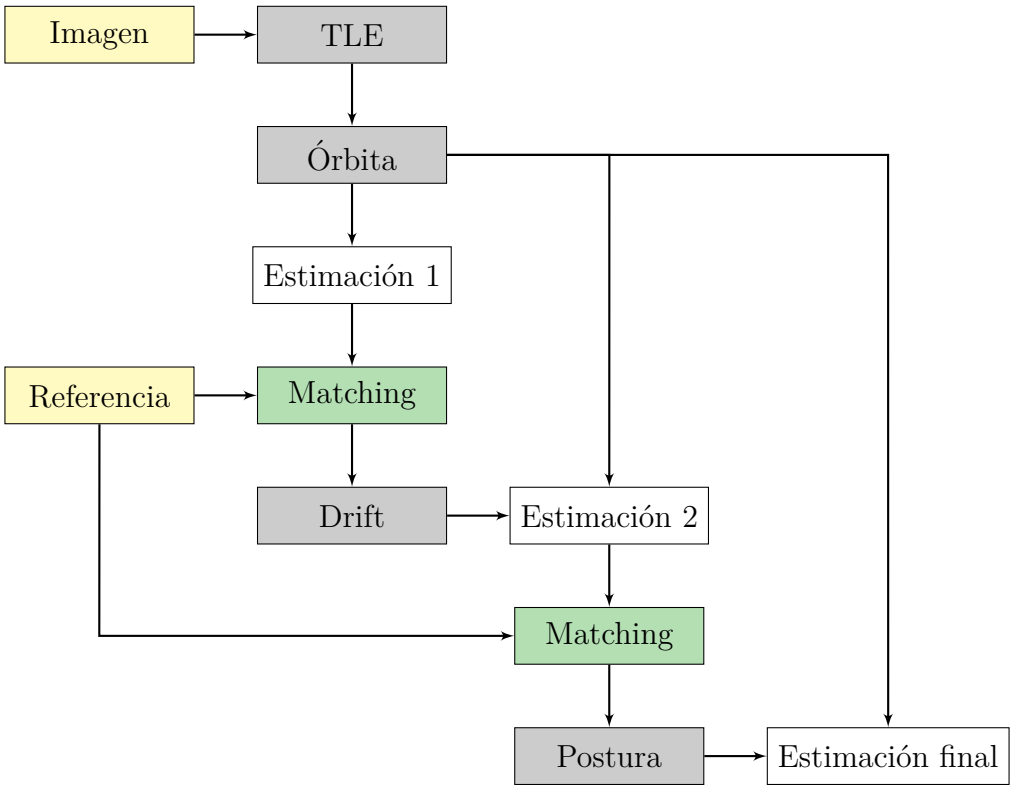

Figura 3.1: Diagrama de flujo del proceso de corrección geométrica de imágenes AVHRR.

3.2. Librería python-sat

El flujo de trabajo se implementó en una librería de Python: `python-sat`. Su estructura está orientada a objetos, y cuenta con diversas clases en las que se dividen las etapas del proceso de geocorrección de imágenes AVHRR. El código fuente de la librería y su descripción completa se encuentra disponible en el apéndice A.

3.2.1. Lectura y escritura de efemérides

El primer paso en la geocorrección de imágenes AVHRR es la lectura de las efemérides de la órbita elipsoidal del satélite. Para ello se define la clase `Ephemeris` (ver A.2), que permite la lectura de TLE a partir de tres líneas de texto y, asimismo, permite exportar instancias de esta clase a la estructura de texto original.

Las propiedades de `Ephemeris` son los parámetros de Kepler en la época de referencia leída, y los métodos de lectura y escritura son `from_tle` y `to_tle`. Un ejemplo de su uso se muestra a continuación:

```
# Ejemplo de lectura y escritura de efemerides.
################################################################################

from six import text_type
from sat.Ephemeris import Ephemeris

# Cadenas de texto para el TLE de MetOp-B del dia 22 de marzo de 2015.
title = text_type(
    "METOP-B              ")
line1 = text_type(
    "1 38771U 12049A 15081.20924951 .00000136 00000-0 82093-4 0 9995")
line2 = text_type(
    "2 38771 98.7074 142.3656 0002161 94.6318 332.5406 14.21481556130061")

# Importa efemerides a la variable ephem.
ephem = Ephemeris.from_tle(title, line1, line2)

# Exporta efemerides como tres lineas de texto a la variable output.
output = ephem.to_tle()
```

Generalmente esta clase no se usa de forma independiente, sino que es una propiedad de estructuras más complejas, como las clases `Orbit` y `Scene`, que se ven a continuación.

3.2.2. Predicción de la posición del satélite

Para georreferenciar una imagen AVHRR es necesario saber la posición del satélite en el instante de captura de cada una de las líneas de la imagen. Con este fin, se define la clase `Orbit` (ver A.3) como un propagador lineal de efemérides expresadas en forma de instancias de `Ephemeris`.

Las propiedades de `Orbit` son todas aquellas presentes en su instancia `Ephemeris` asociada, y además la posición y velocidad del satélite en diversos sistemas de referencia.

Supóngase que se dispone de la variable `ephem` creada en el código de la subsección anterior y que se desea propagar los parámetros keplerianos a las 12:43:00.000 del día 22 de marzo de 2015. El código que debe escribirse es:

```
# Ejemplo de propagacion de parametros orbitales.
################################################################################

from __future__ import print_function
from datetime import datetime
from sat.Orbit import Orbit

# Crea un objeto Orbit con las efemerides de ephem y la fecha de prediccion.
orb = Orbit(ephem)
dat = datetime(2015, 03, 22, 12, 43, 0, 0)

# Llama al metodo de computacion e imprime las coordenadas geograficas.
orb.compute(dat)
print(orb.satellite_position_geo)
```

3.2.3. Estimación de las coordenadas de la imagen

La estructura encargada del cálculo de las coordenadas de una imagen AVHRR es `Scene` (ver A.4). Esta recibe como entrada dos parámetros, a saber, la propia ruta a la imagen AVHRR en estudio y un objeto `Ephemeris` temporalmente próximo al momento de captura. La propia instancia crea en su interior un objeto `Orbit` usando las efemérides introducidas y lee la fecha de captura de la primera línea para propagar los parámetros keplerianos al instante de tiempo asociado con cada una de las líneas que componen la imagen.

El método encargado de todas estas operaciones es `compute`, que calcula las coordenadas geográficas para una malla de píxeles equiespaciados en la imagen, con una cierta separación en la dirección de la trayectoria y otra en la dirección normal. Además, el método `build_geographic_window` permite proyectar la imagen AVHRR original sobre una ventana geográfica en coordenadas latitud-longitud definida de forma automática.

Por ejemplo, supóngase una imagen AVHRR L1B en la ruta `./imagen.l1b` cuyas efemérides asociadas son las de los dos ejemplos anteriores. El código básico para estimar las coordenadas de la imagen es el siguiente:

```
# Ejemplo de estimacion de las coordenadas de una imagen AVHRR L1B.
################################################################################

from sat.Scene import Scene

# Crea un objeto Scene con las efemerides de ephem.
path_scen = text_type("./imagen1.l1b")
scen = Scen(path_scen, ephem)

# Estima las coordenadas de la imagen en una malla con separacion de 25 pixeles
# en la direccion de la trayectoria y 40 pixeles en la direccion normal. El
# resultado es accesible en scen.gcp_position_geo.
scen.compute(40, 25)

# Proyecta la imagen a una ventana geografica en coordenadas latitud-longitud.
# El resultado es accesible en scen.hrpt_img.
scen.build_geographic_window()
```

En este punto, la georreferenciación de la imagen no será correcta, y tendrá errores modelables como la combinación de un desfase en el reloj a bordo y de un cambio de postura, pero la mayor parte de la distorsión de la imagen habrá sido eliminada.

3.2.4. Actualización de las coordenadas de la imagen

La actualización de las coordenadas de un objeto `Scene` se realiza a través de la clase `Matcher`, que es un contenedor del detector FAST, del descriptor BRISK y de la correspondencia Brute-Force utilizando la norma binaria de Hamming (ver A.5). Esta clase recibe como argumentos de entrada la ruta a una imagen GeoTIFF georreferenciada con exactitud y que se utiliza como fuente de puntos característicos, el umbral asociado al detector FAST y el umbral del descriptor BRISK.

Para conectar un objeto `Matcher` con un determinado objeto `Scene`, debe utilizarse el método `set_target`. Una vez realizado este paso, se encuentran disponibles los métodos `fix_attitude` y `fix_clock_drift`, que corrigen respectivamente la postura y el desfase del reloj por correspondencia entre la imagen fuente y la imagen en el objeto `Scene`. Ambos métodos permiten filtrar los puntos equivalentes mediante una cota superior en la distancia geográfica y una cota superior en la distancia binaria entre descriptores.

Supóngase que se desea corregir el objeto `scen` del ejemplo anterior con una imagen de referencia localizada en la ruta `./referencia.tif`, y que se establece un umbral de valor 20 para el detector de esquinas FAST y un umbral de valor 50 para el descriptor BRISK. Supóngase además que el error de posicionamiento de la imagen es inferior a 15 km y que se establece 70 como la distancia de Hamming máxima para que la correspondencia sea fiable. El código básico para este proceso se muestra a continuación:

```
# Ejemplo de estimacion de las coordenadas de una imagen AVHRR L1B.
################################################################################

from sat.Matcher import Matcher

# Crea un objeto Matcher con una imagen de referencia y unos umbrales para el
# detector FAST y el descriptor BRISK. Asocia scen como imagen objetivo.
path_ref = text_type("./referencia.tif")
bfm = Matcher(path, fast_lim=20, brisk_lim=50)
bfm.set_target(scen)

# Corrige el error en desplazamiento asociado al desfase del reloj y actualiza
# las coordenadas de la imagen.
bfm.fix_clock_drift(max_dist=15000, max_hamm=70)
scen.compute(40, 25)

# Corrige el error en postura y actualiza las coordenadas de la imagen.
bfm.fix_attitude(max_dist=15000, max_hamm=70)
scen.compute(40, 25)

# Proyecta la imagen corregida a una ventana geografica definida automaticamente
# en coordenadas latitud-longitud.
scen.build_geographic_window()
```

Después de este último paso, la imagen AVHRR original debería tener asociada una matriz de coordenadas latitud-longitud que georreferencie todos los píxeles con elevada exactitud.

Capítulo 4

Aplicaciones y resultados

Una vez estudiado el marco teórico y propuesto la metodología de trabajo, el siguiente paso es aplicarlo a casos reales para comprobar la bondad de la implementación y aquellos aspectos que pueden ser mejorados en trabajos futuros.

Este capítulo se divide, pues, en dos bloques:

1. La aplicación de la librería `python-sat` y sus clases `Ephemeris` y `Orbit` para estimar las órbitas del satélite MetOp-B a partir de unas efémerides de referencia en una determinada época (procedentes de un TLE) en las que se aplica la propagación temporal lineal de los parámetros keplerianos.

2. La aplicación de las clases `Scene` y `Matcher` para georreferenciar imágenes AVHRR de MetOp-B mediante teoría orbital y posteriormente corregirlas mediante procedimientos de correspondencia de imágenes con una imagen de referencia.

Las aplicaciones de `python-sat` son, además, comparadas con información que puede considerarse estándar o fiable. En el estudio de la propagación orbital de los parámetros keplerianos, los resultados son comparados con los arrojados por la librería `pyephem`, que, en última instancia, utiliza el modelo SGP4 para calcular la posición de los satélites. En el caso de la georreferenciación de imágenes AVHRR, las mallas de coordenadas estimadas por `python-sat` son comparadas con las mallas de GCP contenidas dentro de las propias imágenes, disponibles por proceder estas del repositorio CLASS NOAA.

4.1. Predicción de las coordenadas de MetOp-B

Para verificar el funcionamiento de la librería `python-sat` en la predicción de órbitas satelitales, se emplearon como entrada los TLE del satélite MetOp-B correspondientes a los días 12 y 16 de marzo de 2015, y estos se propagaron en el tiempo desde la época de referencia durante 50 000 s. Los TLE tenían la siguiente estructura:

```
METOP-B
1 38771U 12049A   15071.21400035  .00000107  00000-0  68994-4 0  9999
2 38771  98.7094 132.5045 0002133 108.3585 318.7212 14.21477919128644

METOP-B
1 38771U 12049A   15075.15304370  .00000091  00000-0  61681-4 0  9999
2 38771  98.7085 136.3908 0002145 103.0082 310.0514 14.21478931129209
```

Los resultados para estos dos días pueden observarse respectivamente en las Figuras 4.1 y 4.2, donde se representan las trayectorias de MetOp-B en el plano latitud-longitud. Estas presentan su forma característica, oscilando del polo norte al polo sur con un desplazamiento negativo en longitud a medida que avanza el tiempo como consecuencia de la rotación antihoraria del esferoide terrestre.

Con el fin de comprobar las discrepancias entre este modelo y otras librerías populares actualmente, se compararon las coordenadas geográficas obtenidas por `python-sat` con las calculadas en las mismas condiciones por la librería `pyephem`, que utiliza el modelo SGP4 en la determinación de órbitas satelitales.

En las Figuras 4.3 y 4.4 se muestran las discrepancias en altura entre los dos modelos. Estas oscilan entre unos valores mínimo de 0 km y máximo de aproximadamente 27 km. En el caso de `python-sat`, teniendo en cuenta que el período orbital es de aproximadamente 100 minutos (6000 s), en cada período pueden observarse dos máximos y dos mínimos: los primeros corresponden al paso del satélite por latitudes próximas a los polos, donde el radio terrestre es menor; los segundos ocurren en el paso del satélite por el ecuador, donde el radio terrestre alcanza su valor máximo.

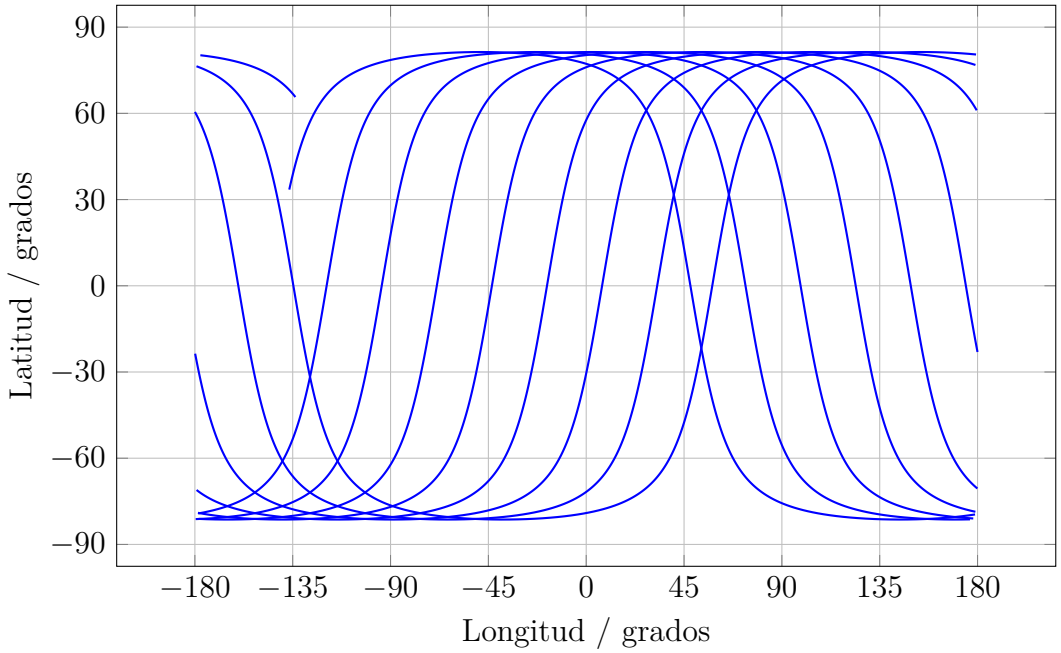

Figura 4.1: Representación de la órbita del satélite MetOp-B propagada temporalmente desde la época 2015/03/12 05:08:09.630240 hasta la fecha 2015/03/12 19:01:29.630240.

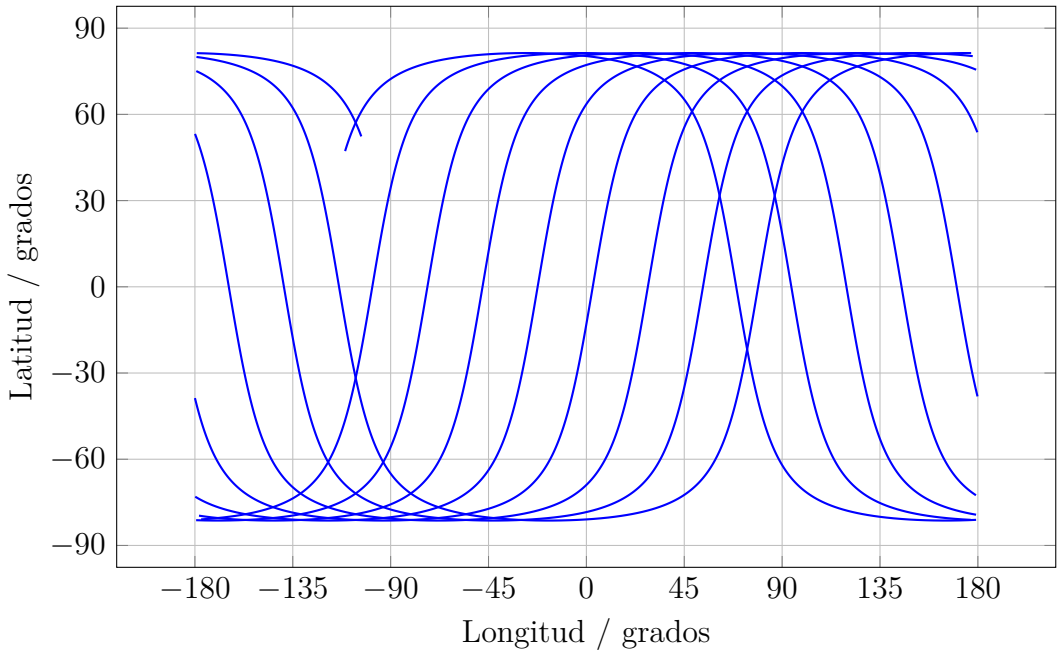

Figura 4.2: Representación de la órbita del satélite MetOp-B propagada temporalmente desde la época 2015/03/16 03:40:22.975680 hasta la fecha 2015/03/16 17:33:42.975680.

En este mismo período, la altura hallada por `pyephem` solo cuenta con un máximo y un mínimo: el primero al pasar cerca del polo sur y el segundo al pasar cerca del polo norte. Así, las mayores discrepancias entre las dos librerías se producen en el paso por los polos, mientras que en el paso por el ecuador los dos modelos coinciden.

Un análisis exhaustivo de estas diferencias en altura implicaría abordar con detalle el funcionamiento del modelo SGP4, que excede el propósito del presente trabajo. No obstante, queda patente el hecho de que `python-sat` sobreestima la altura del satélite, y esto tiene repercursión de cara a estimar las coordenadas de las imágenes AVHRR mediante propagación de los parámetros de Kepler con la librería `python-sat`. Teniendo en cuenta el paso angular del motor del sensor, estimado para el modelo SGP4 (3.3):

$$\delta_0 \approx 0.054102°.$$

un incremento en la altura del satélite se traduce en que para el mismo ángulo de visión se captura una extensión mayor en la dirección perpendicular a la trayectoria del satélite. Este efecto, mínimo en nadir y pequeño en sus proximidades, provoca, no obstante, deformaciones considerables en los extremos de la imagen. Por ello, en la geocorrección de imágenes es necesario disminuir ligeramente el valor de δ_0. Mediante fotointerpretación sobre una colección de imágenes AVHRR de MetOp-B, el nuevo valor δ_0' del paso angular se estimó en:

$$\delta_0' \approx 0.053910°. \tag{4.1}$$

El análisis de discrepancia en la posición subsatelital entre las librerías `python-sat` y `pyephem` se muestra en las Figuras 4.5 y 4.6, donde se ve claramente que las variaciones tienen carácter periódico en el tiempo, con dos subperíodos bien diferenciados: el primero, que tiene dos máximos de 27 km y un mínimo de 6 km, coincide con la trayectoria del satélite sobre el hemisferio norte; el segundo subperíodo, acotado entre 2.5 km y 10 km de discrepancia, coincide con la órbita de MetOp-B sobre el hemisferio sur.

Desde un punto de vista pragmático, la discrepancia es corregible incluyéndola como parte del desfase en el reloj a bordo del satélite, ya que una variación Δt en el tiempo del reloj se traduce en una variación Δr en distancia a lo largo de la trayectoria del satélite.

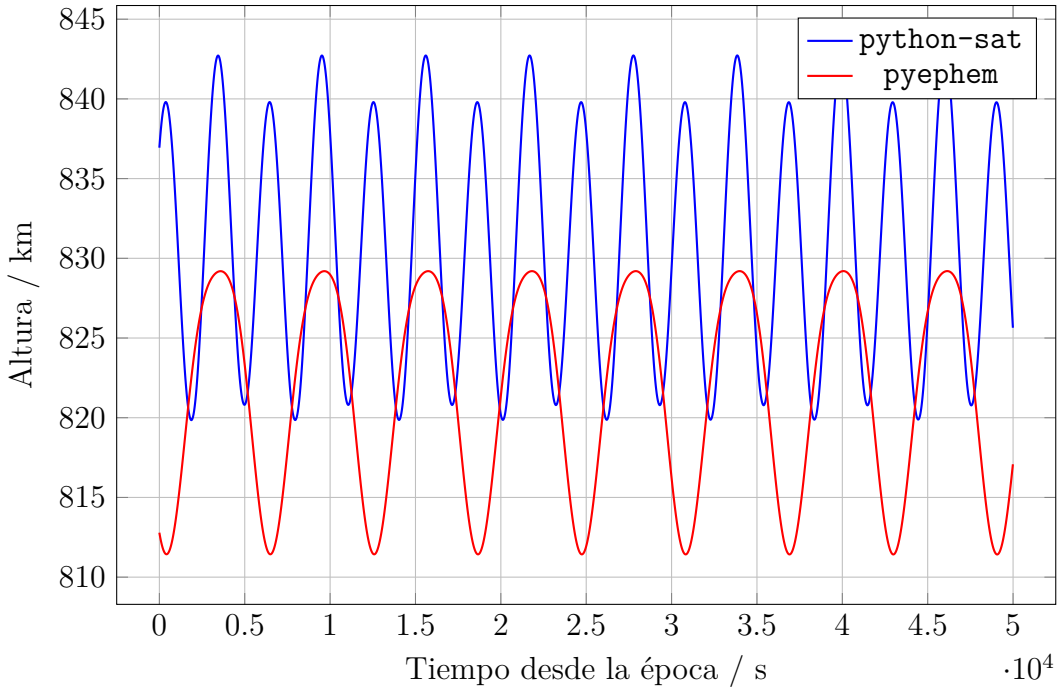

Figura 4.3: Evolución de la altura de MetOp-B desde 2015/03/12 05:08:09.630240.

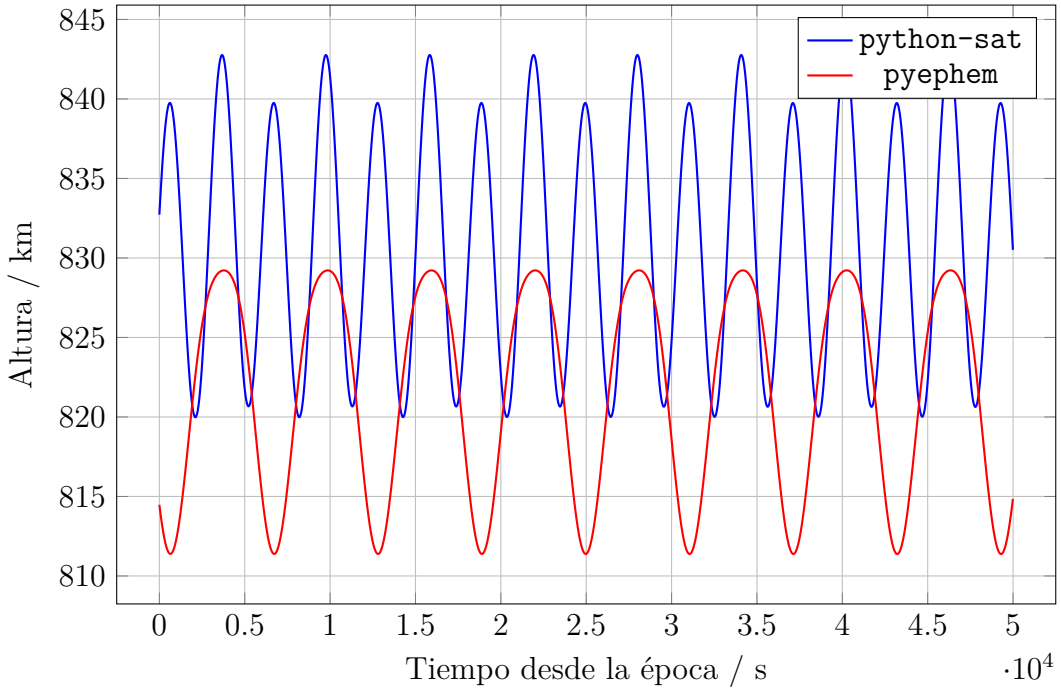

Figura 4.4: Evolución de la altura de MetOp-B desde 2015/03/16 03:40:22.975680.

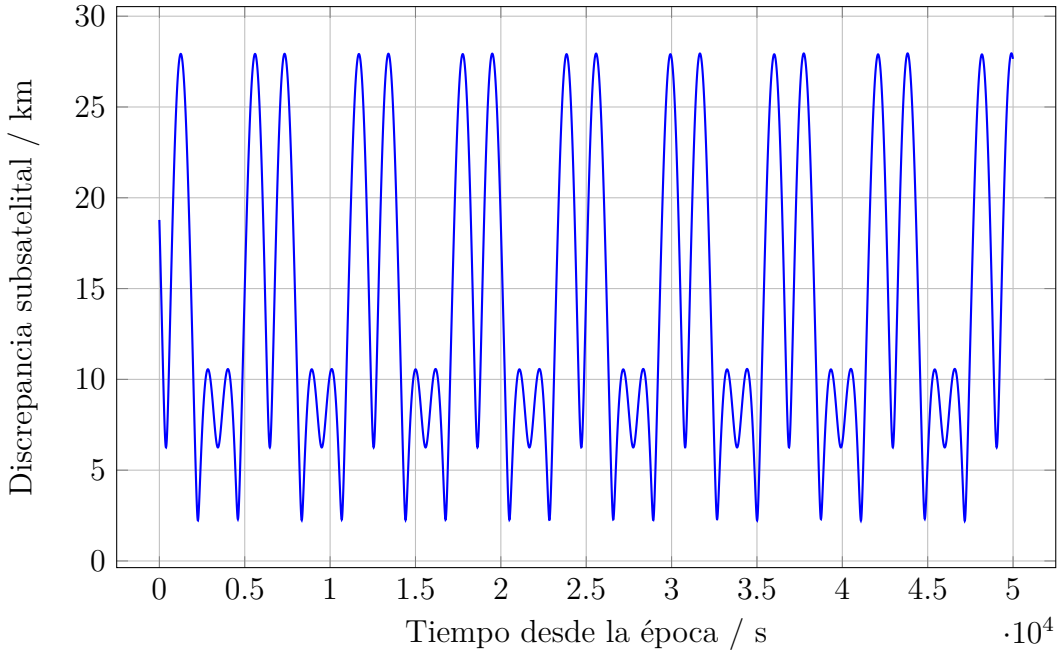

Figura 4.5: Diferencia de posición subsatelital en la propagación orbital de `python-sat` frente a la librería `pyephem` desde la época 2015/03/12 05:08:09.630240.

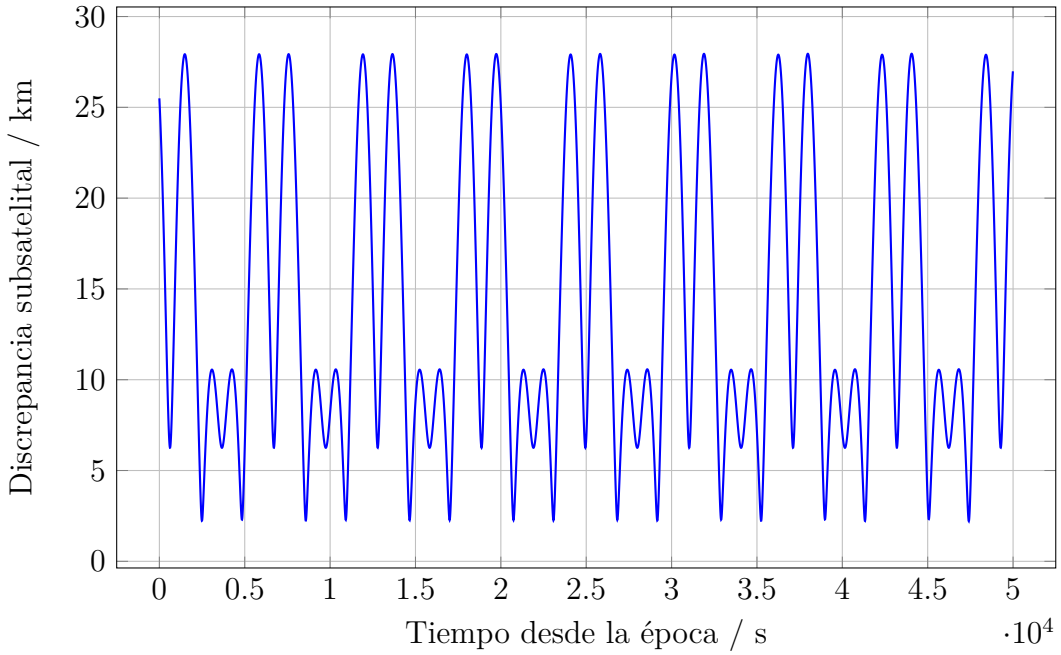

Figura 4.6: Diferencia de posición subsatelital en la propagación orbital de `python-sat` frente a la librería `pyephem` desde la época 2015/03/16 17:33:42.975680.

4.2. Georreferenciación de imágenes AVHRR

La aplicación principal buscada a lo largo de este trabajo es la georreferenciación de imágenes AVHRR procedentes de MetOp-B de modo que se puedan reproducir con cierto margen de error los puntos de control (*Ground Control Points*, GCP) disponibles en las imágenes descargadas del repositorio CLASS NOAA pero no presentes en las imágenes capturadas por una antena en tiempo real.

Para este propósito, se tomaron cinco imágenes AVHRR L1B del repositorio CLASS NOAA, abarcando diferentes horas de pase sobre la Península Ibérica (cuadro 4.1).

Imagen	1	2	3	4	5
Fecha	2015/03/20	2015/03/21	2015/03/22	2015/03/23	2015/03/24
Hora inicial	11:05:15.293	10:44:27.956	10:23:59.450	10:03:23.112	09:42:42.941
Hora final	11:08:53.293	10:48:16.289	10:27:35.284	10:06:52.779	09:46:05.941

Cuadro 4.1: Metadatos temporales de las imágenes de prueba utilizadas.

Como imagen de referencia se eligió una imagen AVHRR del mismo satélite pero de diferente día (2015/03/13 10:10:09.326 a 2015/03/13 10:13:40.826), y se proyectó en una ventana geográfica con coordenadas latitud-longitud y tamaño de píxel de $0.01° \times 0.01°$ (Figura 4.7).

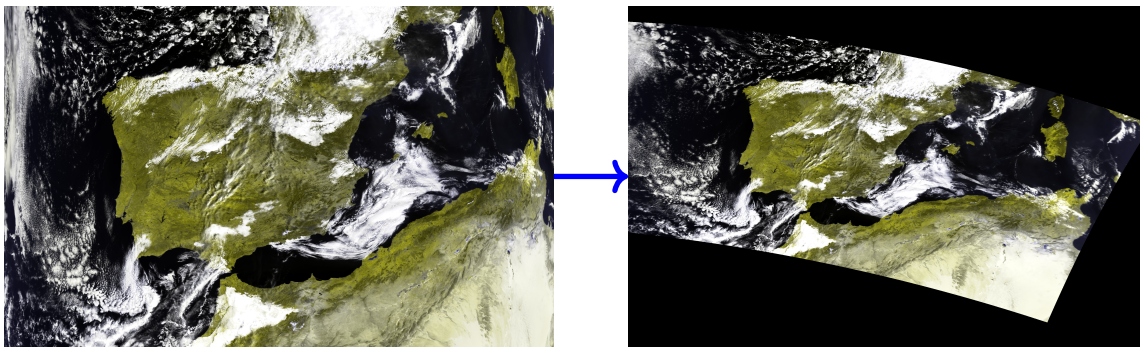

Figura 4.7: Imagen AVHRR de referencia usada en la georreferenciación de las imágenes de prueba, capturada entre 2015/03/13 10:10:09.326 y 2015/03/13 10:13:40.826.

El ángulo utilizado para el paso del motor del sensor fue el indicado en la expresión (4.1). En el momento de redacción de este trabajo, la herramienta de corrección de la postura `fix_attitude` continuaba en fase de pruebas, así que la metodología descrita en el capítulo anterior tuvo que ser ligeramente modificada.

Se escogió una de las cinco imágenes de prueba y de forma manual se realizó una estimación de la postura del sensor, con objeto de usarla en el resto de las imágenes, asumiendo que esta postura permanecía constante. Los valores tomados fueron:

$$\xi_{\mathrm{r}} = 0.065^{\circ}, \qquad \xi_{\mathrm{p}} = 0.000^{\circ}, \qquad \xi_{\mathrm{y}} = -0.070^{\circ}. \tag{4.2}$$

Con esta consideración, todos los pasos de la metodología se mantienen a excepción de 3.2.4, en el que se elimina el proceso de corrección de postura. Además, se impone un proceso recursivo, de modo que todos los pasos se vuelvan a repetir siempre que la variación del *drift* del reloj sea superior a 0.010 s (salvo que no se encuentren correspondencias).

Respecto a la correspondencia con la imagen de referencia, se eligieron los umbrales 20 para el detector FAST y 70 para el descriptor BRISK. Se impuso, además, que los puntos equivalentes no podían encontrarse a una distancia geodésica superior a 15 km, y la distancia de Hamming de corte se estableció en 60.

El resultado de ejecutar las rutinas de `python-sat` queda reflejado en las Figuras 4.8, 4.9, 4.10, 4.11 y 4.12. Puede observarse que el resultado es a primera vista aceptable, especialmente teniendo en cuenta la gran nubosidad existente en todas las imágenes y, en consecuencia, la escasez de líneas de costa en las que encontrar puntos de correspondencia. Asimismo, los resultados son generalmente mejores en aquellas regiones que se encuentran próximas al centro de la imagen.

Para hacer un análisis más específico de la georreferenciación de estas cinco imágenes, se construyó una malla de GCP equivalente a la contenida por las imágenes del repositorio CLASS NOAA, y a continuación se estudió la distancia geodésica entre parejas de GCP (uno procedente de la malla original, el otro procedente de la malla estimada con idéntica fila y columna asociadas).

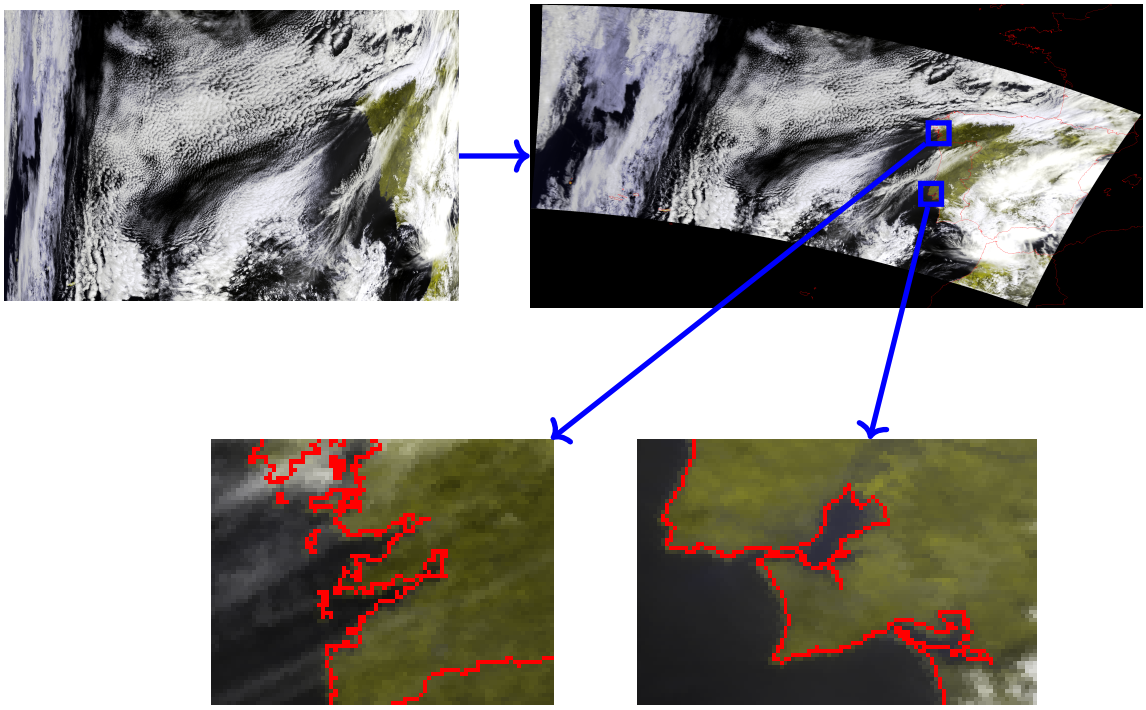

Figura 4.8: Georreferenciación de la imagen de prueba 1.

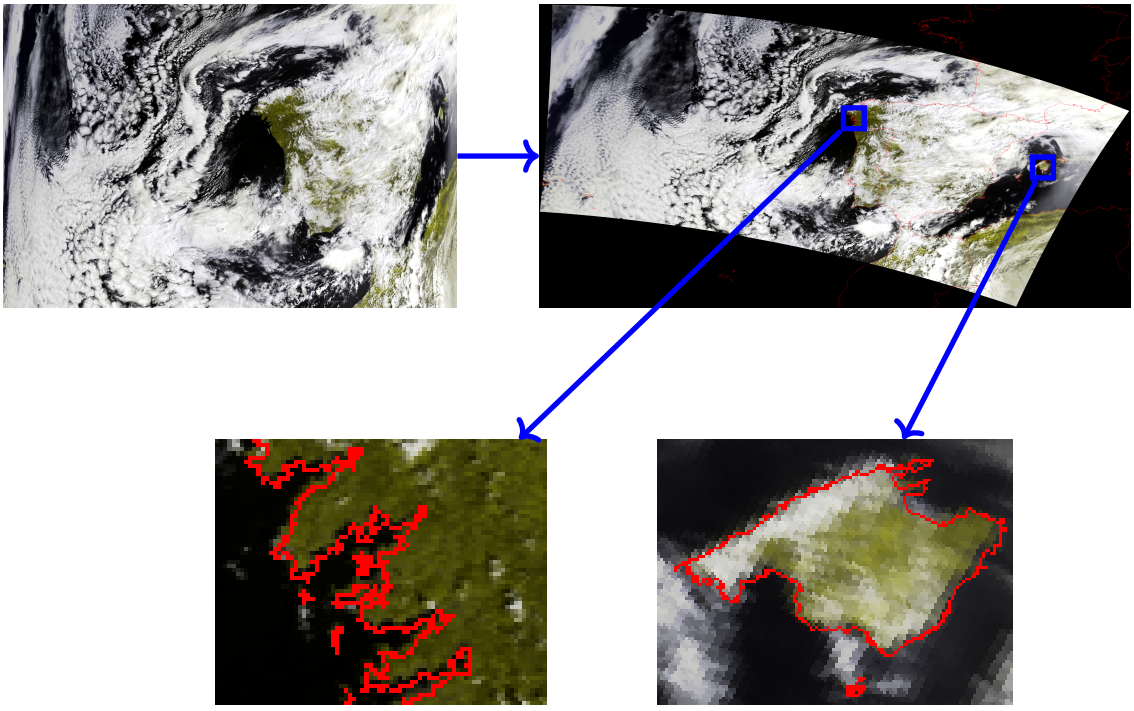

Figura 4.9: Georreferenciación de la imagen de prueba 2.

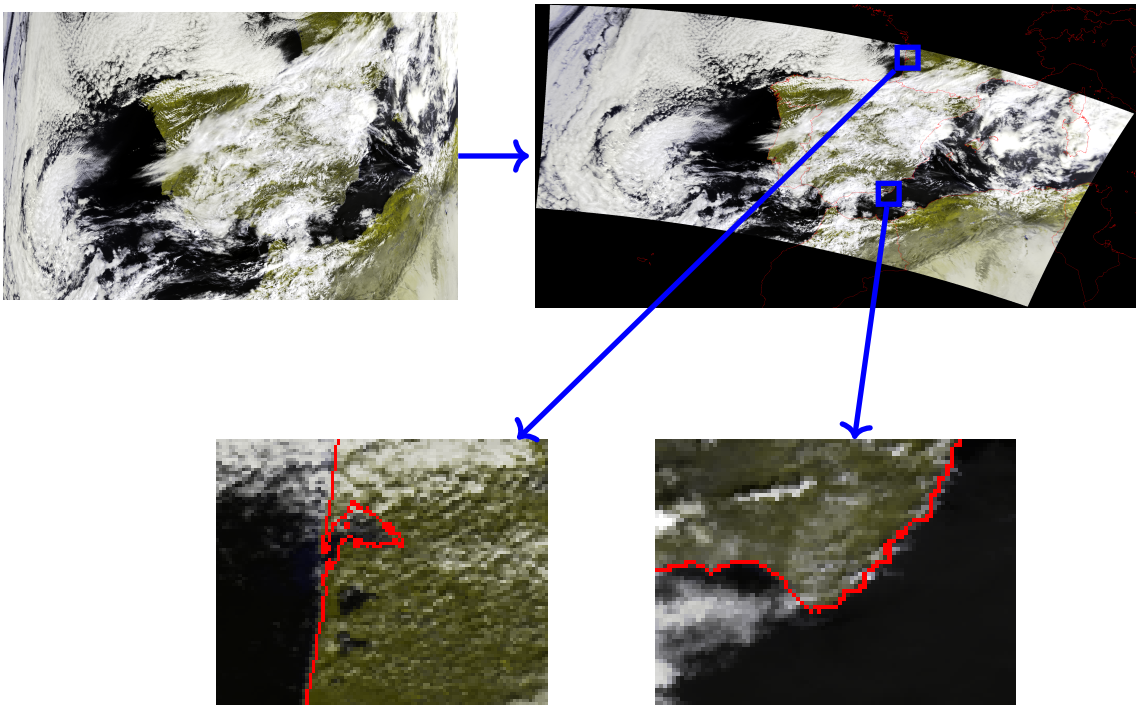

Figura 4.10: Georreferenciación de la imagen de prueba 3.

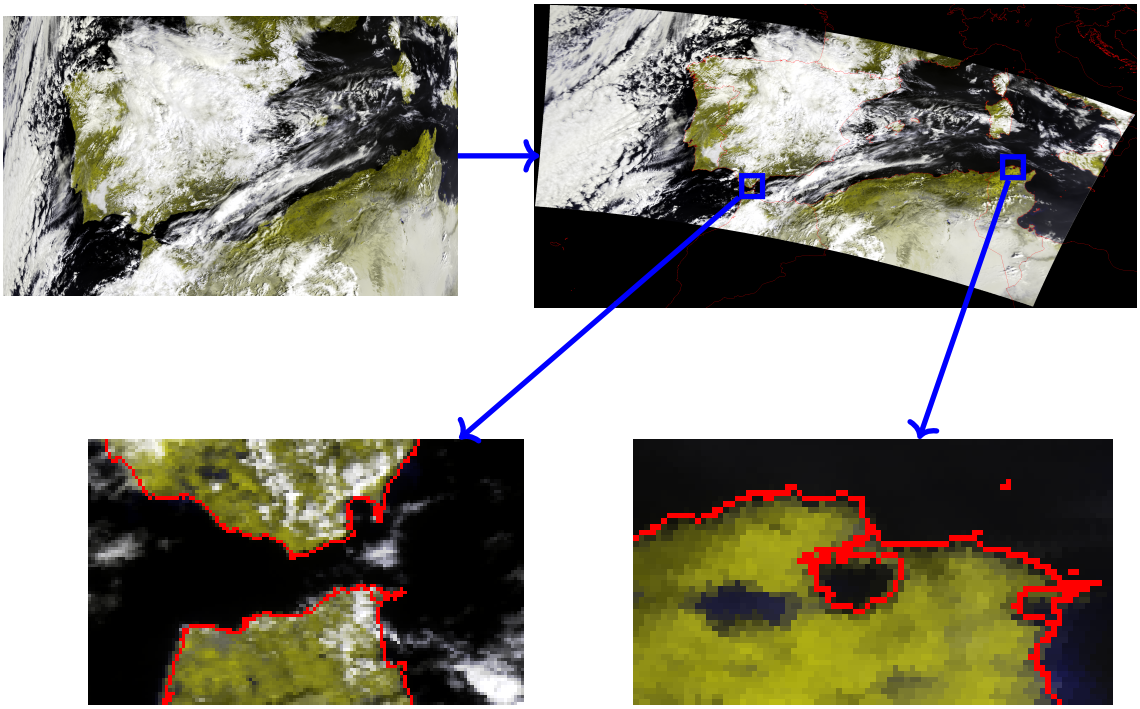

Figura 4.11: Georreferenciación de la imagen de prueba 4.

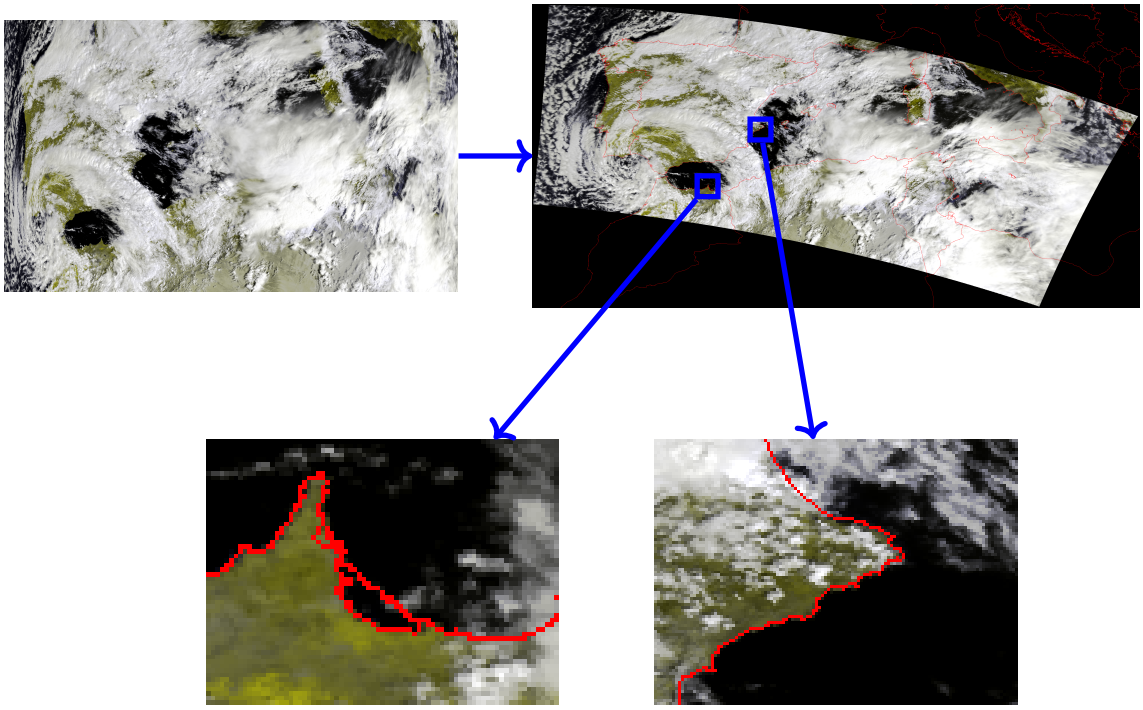

Figura 4.12: Georreferenciación de la imagen de prueba 5.

Para el cálculo de la distancia geodésica d entre dos puntos de coordenadas latitud-longitud (φ_1, λ_1) y (φ_2, λ_2) se empleó la fórmula del semiverseno [28]:

$$d = 2\,R\,\arcsin\sqrt{\sin^2\left(\frac{\varphi_1-\varphi_2}{2}\right)+\cos\varphi_1\cdot\cos\varphi_2\cdot\sin^2\left(\frac{\lambda_1-\lambda_2}{2}\right)}, \tag{4.3}$$

donde R es el radio promedio terrestre, que se tomó igual a 6371 km. Para cada una de las imágenes, se analizó la discrepancia en los GCP en función del índice del fila y del índice de columna en que se encontraban. Las gráficas se presentan en las Figuras 4.13, 4.14, 4.15, 4.16, 4.17, 4.18, 4.19, 4.20, 4.21 y 4.22. La imagen con una menor discrepancia en la georreferenciación respecto a los GCP de CLASS NOAA es la número 3.

Estas gráficas muestran que, en general, la discrepancia decrece conforme aumenta el índice de fila, o sea, conforme la latitud disminuye. No obstante, esta tendencia no se observa en 4.15 y 4.17. Especial mención merece la gráfica 4.15, debido a que corresponde a una imagen realmente desfavorable para la correcta localización de puntos equivalentes como consecuencia de la elevada nubosidad.

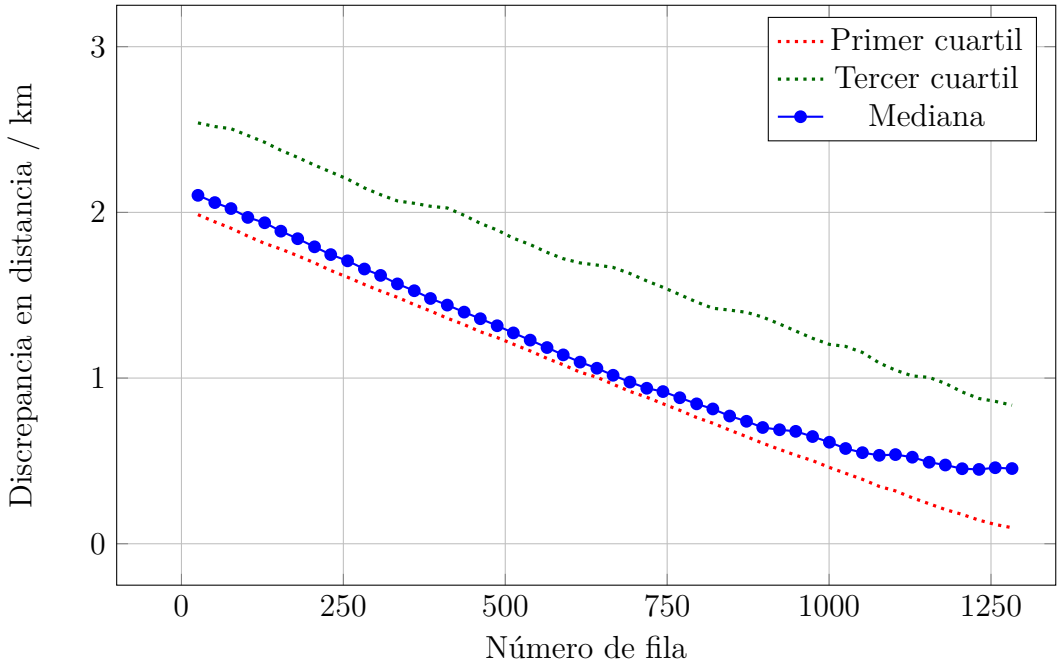

Figura 4.13: Discrepancia entre los GCP oficiales y los hallados con `python-sat` para el pase de 2015/03/20 11:05:15.293 a 2015/03/20 11:08:53.293 en función de la fila.

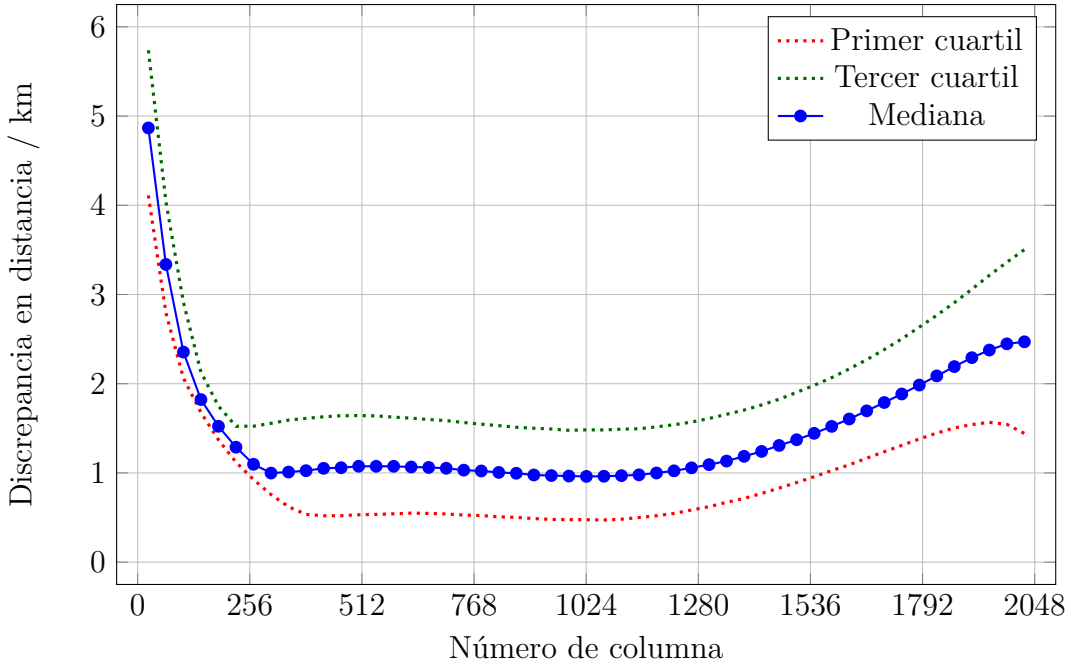

Figura 4.14: Discrepancia entre los GCP oficiales y los hallados con `python-sat` para el pase de 2015/03/20 11:05:15.293 a 2015/03/20 11:08:53.293 en función de la columna.

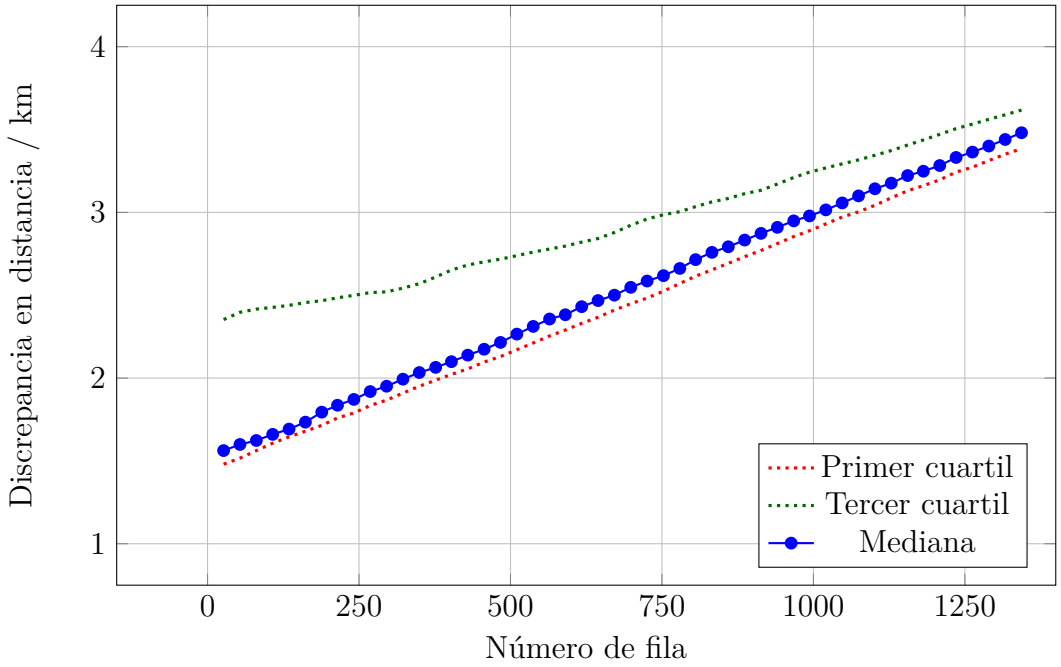

Figura 4.15: Discrepancia entre los GCP oficiales y los hallados con `python-sat` para el pase de 2015/03/21 10:44:27.956 a 2015/03/21 10:48:16.289 en función de la fila.

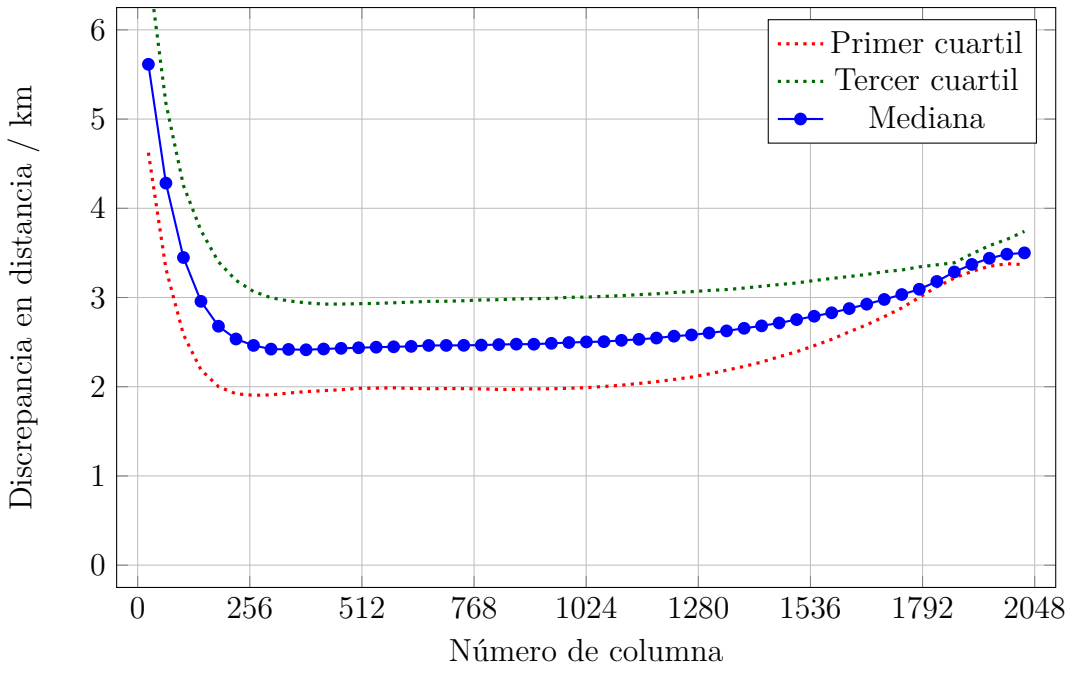

Figura 4.16: Discrepancia entre los GCP oficiales y los hallados con `python-sat` para el pase de 2015/03/21 10:44:27.956 a 2015/03/21 10:48:16.289 en función de la columna.

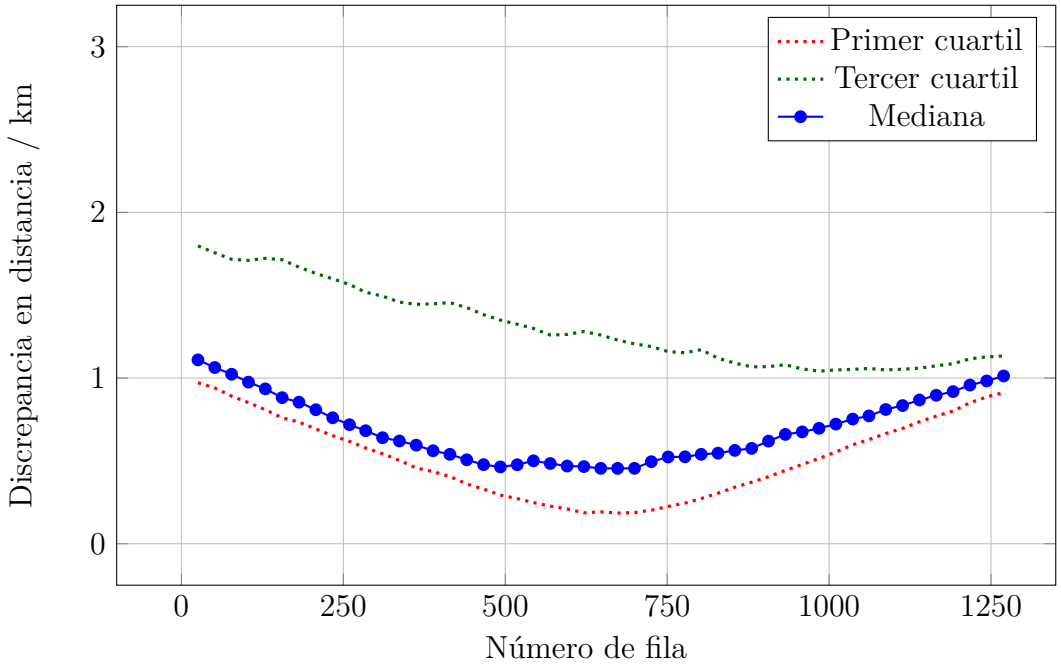

Figura 4.17: Discrepancia entre los GCP oficiales y los hallados con `python-sat` para el pase de 2015/03/22 10:23:59.450 a 2015/03/22 10:27:35.284 en función de la fila.

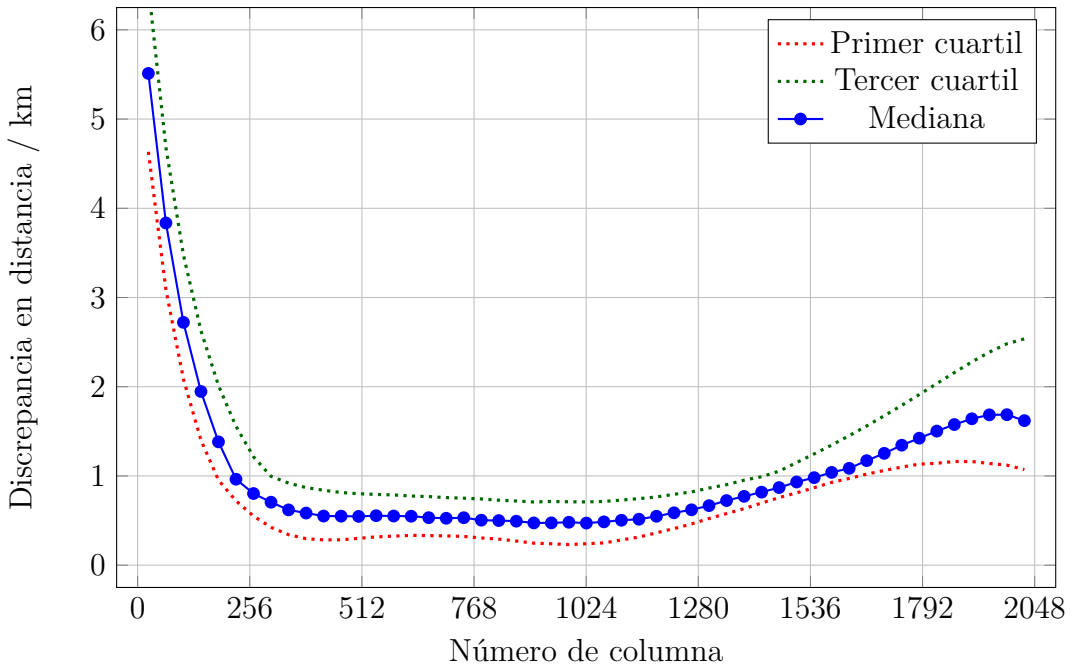

Figura 4.18: Discrepancia entre los GCP oficiales y los hallados con `python-sat` para el pase de 2015/03/22 10:23:59.450 a 2015/03/22 10:27:35.284 en función de la columna.

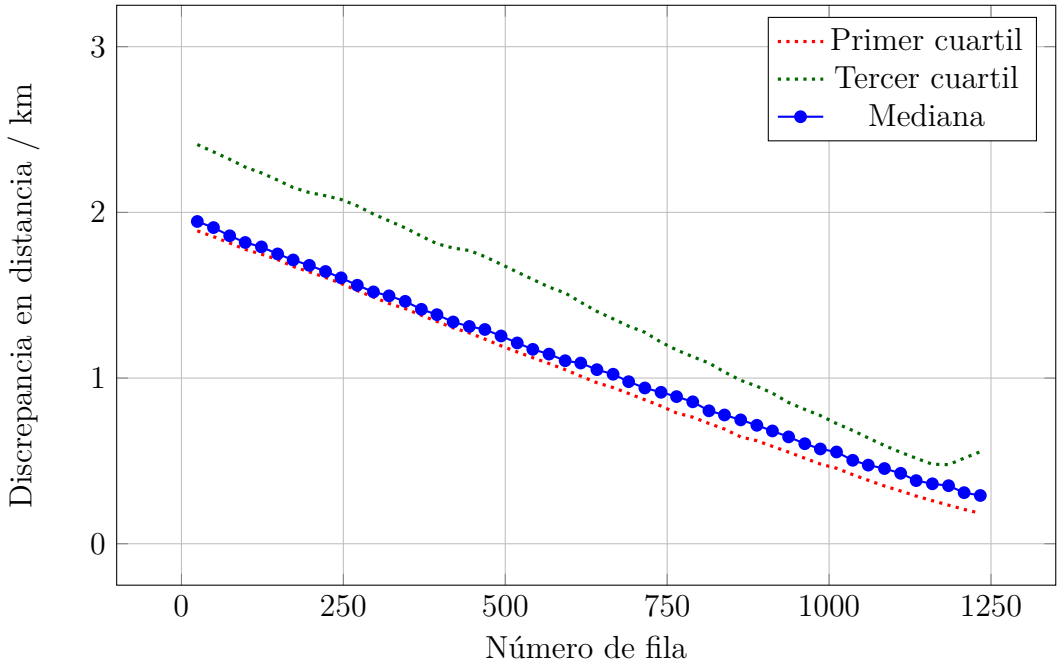

Figura 4.19: Discrepancia entre los GCP oficiales y los hallados con `python-sat` para el pase de 2015/03/23 10:03:23.112 a 2015/03/23 10:06:52.779 en función de la fila.

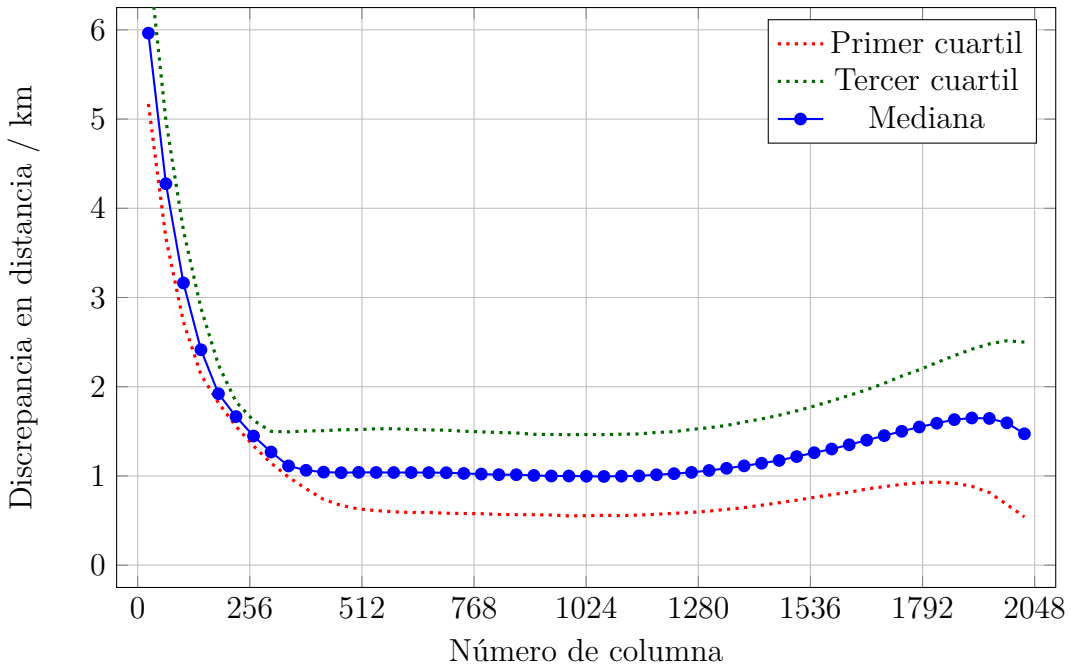

Figura 4.20: Discrepancia entre los GCP oficiales y los hallados con `python-sat` para el pase de 2015/03/23 10:03:23.112 a 2015/03/23 10:06:52.779 en función de la columna.

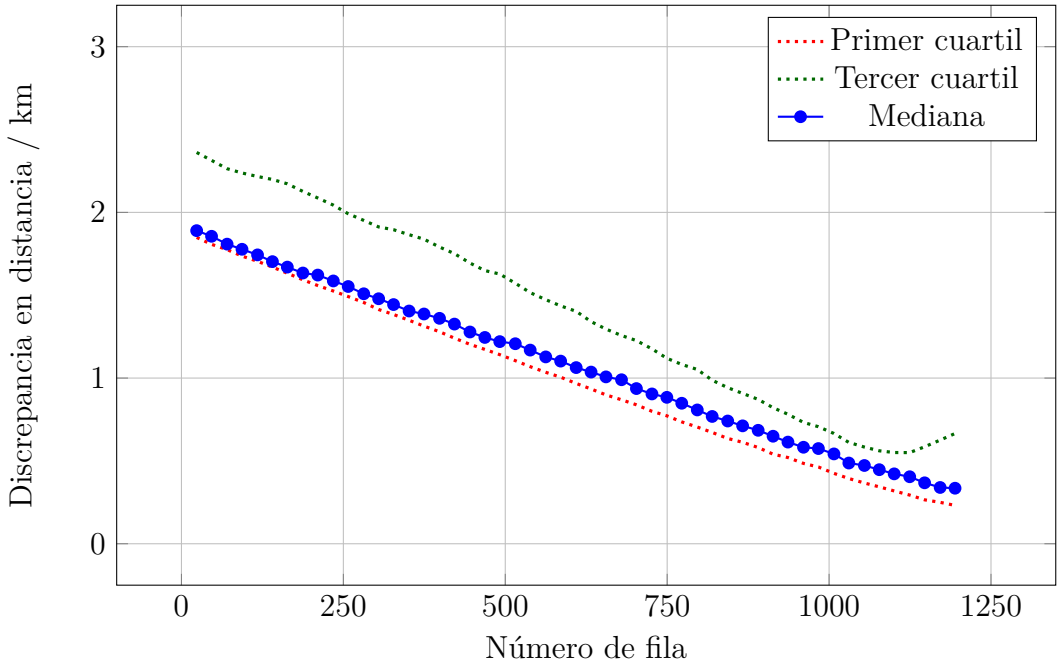

Figura 4.21: Discrepancia entre los GCP oficiales y los hallados con `python-sat` para el pase de 2015/03/24 09:42:42.941 a 2015/03/24 09:46:05.941 en función de la fila.

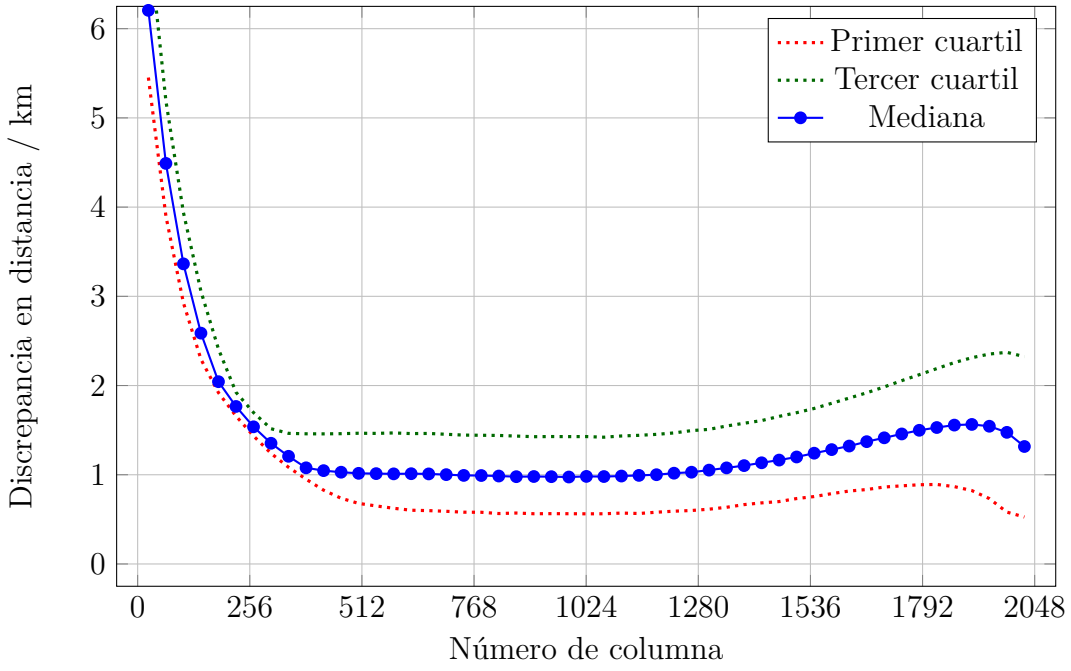

Figura 4.22: Discrepancia entre los GCP oficiales y los hallados con `python-sat` para el pase de 2015/03/24 09:42:42.941 a 2015/03/24 09:46:05.941 en función de la columna.

El comportamiento de la discrepancia en función del índice de columna es más claro. La diferencia se comporta prácticamente plana en los dos tercios centrales de la imagen: el suelo de este valle depende de la bondad de los puntos equivalentes determinados por correspondencia. El valor de este valle se encuentra en el entorno de 1 km salvo en el caso de la imagen desfavorable 2. Una vez que los píxeles se sitúan fuera de esta parte central, la discrepancia se eleva considerablemente. Esto es debido a que los píxeles en los extremos cubren una superficie terrestre mucho mayor que aquellos situados sobre el nadir. No obstante, cabría esperar que las curvas de discrepancia en función del índice de columna fueran simétricas respecto al centro de la imagen AVHRR, y esto no sucede. Ello puede indicar que la estimación realizada para los ángulos de postura es mejorable.

Imagen	1	2	3	4	5
Correspondencias	2	1	100	204	1
Número de iteraciones	3	1	4	4	4
Desfase del reloj / s	-1.668611	-1.14599	-1.575088	-1.585284	-1.680022

Cuadro 4.2: Información sobre la georreferenciación de las imágenes de prueba.

Capítulo 5

Conclusiones y líneas futuras

A lo largo del presente trabajo se han abordado los diferentes pasos necesarios para estimar la georreferenciación de imágenes AVHRR y corregirla posteriormente mediante correspondencia con una imagen de referencia. Pese a que el modelo orbital desarrollado presenta discrepancias con el modelo SGP4, estas son salvables: la diferencia en altura se corrige reduciendo el paso angular del motor del sensor, mientras que la diferencia en la posición subsatélite se enmascara en el desfase temporal del reloj de a bordo.

Los resultados obtenidos con las cinco imágenes ejemplo son aceptables, más aún teniendo en cuenta la limitación que presentan debido a su elevada nubosidad, la cual dificulta la correspondencia de esquinas de costa entre la imagen de estudio y la imagen de referencia. En el caso más desfavorable solo se disponía de un punto de control y el error medio se encontraba en el entorno de los 2.5 km.

El trabajo tiene, no obstante, margen de mejora, especialmente en el ámbito de la corrección en postura. El planteamiento está desarrollado, y la mayor parte del código está escrito, pero se ha tenido que hacer una estimación manual de los ángulos de postura. Este procedimiento ha mostrado deficiencia en aquellas imágenes más limitadas en la tasa de correspondencias.

Por otro lado, el procedimiento solamente ha sido testado en imágenes diurnas. En el caso de las imágenes nocturnas, los canales 1 y 2 no están disponibles, por lo que la

correspondencia tiene que hacerse con los canales 3B, 4 o 5. Mientras que durante el día la línea de costa es perfectamente identificable en el canal 2 en condiciones despejadas, esto no sucede en ninguno de los canales nocturnos, ya que la temperatura en las tierras costeras puede ser similar a la del agua próxima. Un breve análisis con dos imágenes AVHRR nocturnas mostró que el detector FAST es ineficiente en la detección de puntos característicos con cualquiera de los canales nocturnos. Sería necesario, pues, buscar una estrategia de correspondencia con otros algoritmos más sofisticados. Una posible alternativa a los vértices de costa sería buscar lagos y embalses, pues los cuerpos de agua dentro de la tierra son mucho más oscuros que su entorno en el canal 3B.

Con vistas a incluir este trabajo en un proyecto de captura y procesado de imágenes MetOp-B AVHRR, quedaría aún mucho trabajo pendiente, ya que solamente se ha abordado un apartado de la conversión de Level 1A a Level 1B. Es necesario conectar este paso con la generación del fichero Level 1A a partir del binario Level 0 y la extracción de los metadatos de interés. El software iDAP/MacroPro no es una alternativa, puesto que, como ya se comentó, no se tiene acceso al código ni a los ficheros intermedios generados entre el Level 0 y el Level 1B proyectado. El software AAPP sí podría ser una alternativa viable, e incluso sería de interés analizar la calidad de su georreferenciación después de haber realizado el comunicado en el foro de usuarios acerca de los cambios en los ángulos de postura. La tercera alternativa es crear un traductor propio de Level 0 a Level 1A a partir de las especificaciones en [1]. Es una tarea viable pero tediosa; no obstante, la estructura del binario se repite para cada línea de la imagen capturada. Asimismo, a pesar de que la resolución de la imagen es de 10 bits por palabra, realmente cada una es almacenada en 16 bits cuyos primeros 6 bits son siempre nulos. Esto facilita la lectura del archivo binario, ya que las herramientas de desempaquetado de los lenguajes de programación de alto nivel (por ejemplo, `struct` en Python), solo pueden leer números enteros de bytes. El almacenamiento en 10 bits reales prácticamente obligaría a crear el traductor en un lenguaje de bajo nivel, como C, y realizar la manipulación bit a bit. Una vez leído este binario, las bandas sin calibrar estarán disponibles, así como los metadatos (instante de captura, parámetros de calidad y de calibración, etc.).

El trabajo, por todo lo expuesto anteriormente, no termina aquí, y la evolución será observable en el repositorio de GitHub https://github.com/molinav/python-sat, donde se encuentra todo el código disponible para compartirlo, editarlo o escribir código derivado.

Apéndice A

Código fuente de python-sat

En el presente apéndice se muestra la librería `python-sat` desarrollada para la resolución de los problemas planteados en este trabajo. Pueden distinguirse cinco bloques:

1. Las definiciones preliminares, o sea, aquellas piezas de código que generalizan funciones utilizadas con frecuencia en el resto de la librería. En ellas se incluyen los decoradores (funciones que realizan modificaciones simples sobre otras funciones ya existentes); la clase `Angle`, que gestiona variables angulares y sus conversiones entre diferentes unidades; la clase `Coordinates`, que contiene funciones de conversión entre sistemas de coordenadas; y la clase `Error`, donde se definen las excepciones alcanzadas cuando la librería llega a un error.

2. La clase `Ephemeris`, que gestiona la lectura y escritura de TLE, donde se encuentran las efemérides de un satélite para una determinada época.

3. La clase `Orbit`, que propaga los parámetros keplerianos a las fechas solicitadas a partir de las efemérides de una época.

4. La clase `Scene`, que contiene todos los métodos necesarios para georreferenciar una imagen una vez conocida la posición del satélite en los instantes de captura.

5. La clase `Matcher`, donde se tienen todas las herramientas de correspondencia de imágenes para corregir las coordenadas de una imagen a partir de otra de referencia.

A.1. Definiciones preliminares

Código A.1: ../python-sat/sat/decorators/_str.py

```
"""Private functions for nicely printing of TypeError messages."""


def str_format_types(types):
    return "({})".format(", ".join(t.__name__ for t in types))


def str_type_error_message(f, args, types, case):
    msg = "'{}' {} {}, but {} {}".format(
        f.__name__,
        ("accepts", "returns")[case],
        str_format_types(types),
        ("was given", "result is")[case],
        str_format_types(map(type, args)),
        )
    return msg
```

Código A.2: ../python-sat/sat/decorators/accepts.py

```
from . _str import str_type_error_message


def accepts(*types, **kwargs):
    """Decorator that checks the types of function's arguments.

    Parameters:

    types
        the expected types of the decorated function's input values
    kwargs
        keyword arguments only accepts boolean flag "static" that
        sets the decorator for static functions (default False)
    """

    # Verify the structure of keyword arguments.
    kwargs_flag = list(kwargs.keys())
    assert kwargs_flag == [] or kwargs_flag == ["static"]
    try:
        static = kwargs["static"]
    except KeyError:
        static = False

    def decorator(f):

        # Define inner function.
        def new_f(*args):
            # Assert characteristics of input arguments.
            flag = bool(not static)
```

```
            assert len(args) == len(types) + flag
            assert min(map(isinstance, types, (type for t in types)))
            # Check that the input values have the expected type.
            if not min(map(isinstance, args[flag:], types)):
                msg = str_type_error_message(f, args[flag:], types, 0)
                raise TypeError(msg)
            return f(*args)

        # Return inner function.
        new_f.__name__ = f.__name__
        return new_f

    # Return decorator.
    return decorator
```

Código A.3: ../python-sat/sat/decorators/limits.py

```
from numbers import Real


def limits(left, minv, maxv, right, getter=False, static=False):
    """Getter decorator that checks the value is within the valid range.

    Parameters:

    left
        type of lower limit ('(' for open, '[' for closed)
    minv
        the lower limit value
    maxv
        the upper limit value
    right
        type of upper limit (')' for open, ']' for closed)
    getter
        flag that sets getter mode if True, otherwise setter mode is used
        (default False)
    static
        boolean flag that sets the decorator for static functions
        (default False)
    """

    def decorator(f):

        assert left in "([" and right in ")]"
        assert min(map(isinstance, (minv, maxv), (Real, Real)))
        assert minv < maxv

        # Define inner function.
        def new_f(*args):
            # Assert characteristics of input arguments.
            flag = bool(not static)
            if getter:
                assert len(args) is 0 + flag
                val = f(*args)
```

```
38              else:
39                  assert len(args) is 1 + flag
40                  val = args[flag]
41              assert isinstance(val, Real)
42              # Check that input value is within the expected range.
43              min_flag = val <= minv if left is "(" else val < minv
44              max_flag = val >= maxv if left is ")" else val > maxv
45              if min_flag or max_flag:
46                  msg = "value '{}' is not within range [{:.4g}, {:.4g}]"\
47                      .format(val, minv, maxv)
48                  raise TypeError(msg)
49              return val if getter else f(*args)
50
51          # Return inner function.
52          new_f.__name__ = f.__name__
53          return new_f
54
55      # Return decorator.
56      return decorator
```

Código A.4: ../python-sat/sat/decorators/maximum_text_length.py

```
1  from six import text_type
2
3
4  def maximum_text_length(max_length, getter=False, static=False):
5      """Decorator that checks the maximum length of a string value.
6
7      Parameters:
8
9      max_length
10         maximum value for string length
11     getter
12         flag that sets getter mode if True, otherwise setter mode is used
13         (default False)
14     static
15         boolean flag that sets the decorator for static functions
16         (default False)
17     """
18
19     def decorator(f):
20
21         assert isinstance(max_length, int)
22         assert max_length > 0
23
24         # Define inner function.
25         def new_f(*args):
26             # Assert characteristics of input arguments.
27             flag = bool(not static)
28             if getter:
29                 assert len(args) is 0 + flag
30                 val = f(*args)
31             else:
32                 assert len(args) is 1 + flag
```

```
33                 val = args[flag]
34             assert isinstance(val, text_type)
35             # Check that the string variable has the expected length.
36             if len(val) > max_length:
37                 msg = "text string exceeds the maximum length ({})"\
38                     .format(max_length)
39                 raise ValueError(msg)
40             return val if getter else f(*args)
41
42         # Return inner function.
43         new_f.__name__ = f.__name__
44         return new_f
45
46     # Return decorator.
47     return decorator
```

Código A.5: ../python-sat/sat/decorators/pattern.py

```
1  import re
2  from six import text_type
3
4
5  def pattern(regex, getter=False, static=False):
6      """Decorator that checks valid patterns of a string attribute.
7
8      Parameters:
9
10     regex
11         regular expression as string
12     getter
13         flag that sets getter mode if True, otherwise setter mode is used
14         (default False)
15     static
16         boolean flag that sets the decorator for static functions
17         (default False)
18     """
19
20     def decorator(f):
21
22         assert isinstance(regex, text_type)
23
24         # Define inner function.
25         def new_f(*args):
26             # Assert characteristics of input arguments.
27             flag = bool(not static)
28             if getter:
29                 assert len(args) is 0 + flag
30                 val = f(*args)
31             else:
32                 assert len(args) is 1 + flag
33                 val = args[flag]
34             assert isinstance(val, text_type)
35             # Check that the string variable has the expected length.
36             if not re.match(regex, val):
```

```
37                 att = f.__name__.replace("_", " ")
38                 msg = "invalid {} '{}'".format(att, val)
39                 raise ValueError(msg)
40             return val if getter else f(*args)
41
42         # Return inner function.
43         new_f.__name__ = f.__name__
44         return new_f
45
46     # Return decorator.
47     return decorator
```

Código A.6: ../python-sat/sat/decorators/returns.py

```
1  from . _str import str_type_error_message
2
3
4  def returns(types):
5      """Decorator that checks the type of function's return value.
6
7      Parameters:
8
9      types
10         the expected type of the decorated function's return value
11     """
12
13     def decorator(f):
14
15         # Define inner function.
16         def new_f(*args):
17             # Assert characteristics of input arguments.
18             assert isinstance(types, type)
19             # Check that the output value has the expected type.
20             val = f(*args)
21             if not isinstance(val, types):
22                 msg = str_type_error_message(f, (val,), (types,), 1)
23                 raise TypeError(msg)
24             return val
25
26         # Return inner function.
27         new_f.__name__ = f.__name__
28         return new_f
29
30     # Return decorator.
31     return decorator
```

Código A.7: ../python-sat/sat/Angle.py

```
1  from __future__ import division
2  from . constants.Angle import DEG
3  from . constants.Angle import GRD
4  from . constants.Angle import RAD
5  from . constants.Angle import REV
```

```
6  from . constants.Angle import DEG2RAD
7  from . constants.Angle import GRD2RAD
8  from . constants.Angle import RAD2RAD
9  from . constants.Angle import REV2RAD
10 from . constants.Angle import DEFAULT_ANGLE_VALUE
11 from . constants.Angle import INFINITY
12 from . constants.Angle import PI
13 from . decorators import accepts
14 from . decorators import limits
15 from . decorators import returns
16 from numbers import Real
17
18
19 class Angle(object):
20     """Base class which handles angles in different notations."""
21
22     __slots__ = [
23         "_angle",
24         ]
25
26     _angle_min = -INFINITY
27     _angle_max = +INFINITY
28     _limit_min = "["
29     _limit_max = "]"
30
31     def __init__(self, **kwargs):
32         self._angle = DEFAULT_ANGLE_VALUE
33         # Set angle value.
34         num_args = len(kwargs)
35         if num_args is 0:
36             pass
37         elif num_args is 1:
38             key, val = list(kwargs.items())[0]
39             try:
40                 factor = {
41                     DEG: DEG2RAD, GRD: GRD2RAD, RAD: RAD2RAD, REV: REV2RAD}
42                 self.rad = val * factor[key]
43             except KeyError:
44                 msg = "invalid key '{}' (expected '{}', '{}', '{}', '{}')"\
45                     .format(key, DEG, GRD, RAD, REV)
46                 raise AttributeError(msg)
47         else:
48             msg = "too many arguments (expected 1)"
49             raise AttributeError(msg)
50
51     def norm(self):
52         """return an Angle instance within the interval [0, 2*pi) rad"""
53         val = self.rad % (2*PI)
54         return Angle(rad=val)
55
56     def wrap(self):
57         """return an Angle instance within the interval (-pi, pi] rad"""
58         val = self.rad % (2*PI)
59         val = val - (2*PI) if val > PI else val
60         return Angle(rad=val)
```

```
61
62      def normed(self):
63          """normalize an angle within the interval [0, 2*pi) rad"""
64          val = self.rad % (2*PI)
65          self.rad = val
66
67      def wrapped(self):
68          """wrap the instance value within the interval (-pi, pi] rad"""
69          val = self.rad % (2*PI)
70          self.rad = val - (2*PI) if val > PI else val
71
72      def _return_angle(self, factor):
73          """generic getter that returns the angle value"""
74          return self._angle / factor
75
76      def _update_angle(self, factor, val):
77          """generic setter that updates the angle value"""
78
79          value_min = self._angle_min / factor
80          value_max = self._angle_max / factor
81
82          @limits(self._limit_min, value_min, value_max, self._limit_max)
83          def _calc_safe(s, v):
84              s._angle = v * factor
85
86          _calc_safe(self, val)
87
88      @property
89      @returns(Real)
90      def deg(self):
91          """angle value in degrees"""
92          return self._return_angle(DEG2RAD)
93
94      @deg.setter
95      @accepts(Real)
96      def deg(self, val):
97          self._update_angle(DEG2RAD, val)
98
99      @property
100     @returns(Real)
101     def grd(self):
102         """angle value in gradians"""
103         return self._return_angle(GRD2RAD)
104
105     @grd.setter
106     @accepts(Real)
107     def grd(self, val):
108         self._update_angle(GRD2RAD, val)
109
110     @property
111     @returns(Real)
112     def rad(self):
113         """angle value in radians"""
114         return self._return_angle(RAD2RAD)
115
```

```
116     @rad.setter
117     @accepts(Real)
118     def rad(self, val):
119         self._update_angle(RAD2RAD, val)
120 
121     @property
122     @returns(Real)
123     def rev(self):
124         """angle value in revolutions"""
125         return self._return_angle(REV2RAD)
126 
127     @rev.setter
128     @accepts(Real)
129     def rev(self, val):
130         self._update_angle(REV2RAD, val)
131 
132 
133 class AziAngle(Angle):
134     """Angle subclass whose angle is restricted to [0, 2*pi] rad."""
135 
136     _angle_min = 0
137     _angle_max = 2*PI
138     _limit_min = "["
139     _limit_max = ")"
140 
141 
142 class ZenAngle(Angle):
143     """Angle subclass whose angle is restricted to [0, pi] rad."""
144 
145     _angle_min = 0
146     _angle_max = PI
147     _limit_min = "["
148     _limit_max = "]"
149 
150 
151 class LatAngle(Angle):
152     """Angle subclass whose angle is restricted to [-pi/2, +pi/2] rad."""
153 
154     _angle_min = -PI/2
155     _angle_max = +PI/2
156     _limit_min = "["
157     _limit_max = "]"
158 
159 
160 class LonAngle(Angle):
161     """Angle subclass whose angle is restricted to [-pi, +pi] rad."""
162 
163     _angle_min = -PI
164     _angle_max = +PI
165     _limit_min = "("
166     _limit_max = "]"
```

Código A.8: ../python-sat/sat/Coordinates.py

```
from __future__ import division
from . constants.Angle import REV2RAD
from . constants.Coordinates import EARTH_ANGULAR_SPEED
from . constants.Coordinates import EARTH_SEMIMAJOR_AXIS
from . constants.Coordinates import EARTH_FLATTENING_FACTOR
from . constants.Coordinates import GEODETIC_COORDINATES_TOLERANCE
from . constants.Orbit import DAY2SEC
import numpy as np


class Coordinates(object):

    def __init__(self):
        pass

    @classmethod
    def greenwich_mean_sidereal_time(cls, datetime):
        """Compute Greenwich mean sidereal time.

        Parameters:

        datetime
            datetime64 array from numpy library
        """

        # Call necessary constants.
        we = EARTH_ANGULAR_SPEED
        j2000 = np.datetime64("2000-01-01T00:00:00Z")

        # Compute decimal Julian day.
        dtm0 = ((datetime - j2000) / np.timedelta64(1, "D")).astype(float)
        # Compute Julian centuries since epoch 2000/01/01 12:00:00.
        dtm1 = ((dtm0 - 1) // 1 + 0.5) / 36525
        # Compute solar time.
        dtm2 = (dtm0 % 1) * DAY2SEC

        # Compute Greenwich mean solar time in radians.
        c0, c1, c2, c3 = [24110.54841, 8640184.812866, 0.093104, -6.2e-6]
        gst0 = (c0 + c1 * dtm1 + c2 * dtm1**2 + c3 * dtm1**3) / DAY2SEC
        gst1 = REV2RAD * gst0 + we * dtm2

        return gst1

    @classmethod
    def from_eci_to_ecf(cls, obj, datetime):
        """Compute ECF coordinates from ECI coordinates.

        Parameters:

        obj
            (n, 3) array containing (x_ecef, y_ecef, z_ecef) in meters
        datetime
            (n,) datetime64 array from numpy library
```

```
54          """
55
56          # Call necessary properties.
57          rx, ry, rz = [x[:, None] for x in obj.T]
58          # Compute the Greenwich Sidereal Time (GST), that is, the angle
59          # between the vernal point and the Greenwich meridian (which is
60          # also the angle between the ECI and ECF reference systems).
61          gst = Coordinates.greenwich_mean_sidereal_time(datetime)
62          # Compute ECF coordinates as a rotation of ECI coordinates.
63          rx_s = + np.cos(gst)*rx + np.sin(gst)*ry
64          ry_s = - np.sin(gst)*rx + np.cos(gst)*ry
65          rz_s = rz
66
67          return np.hstack([rx_s, ry_s, rz_s])
68
69      @classmethod
70      def from_ecf_to_geo(cls, obj):
71          """Compute geodetic coordinates from ECF coordinates.
72
73          Parameters:
74
75          obj
76              (n, 3) array containing (x_ecef, y_ecef, z_ecef) in meters
77          """
78
79          # Call necessary constants.
80          ae = EARTH_SEMIMAJOR_AXIS
81          e2 = EARTH_FLATTENING_FACTOR
82          # Call necessary properties.
83          rx_s, ry_s, rz_s = [x[:, None] for x in obj.T]
84          p_factor = np.sqrt(rx_s**2 + ry_s**2)
85
86          def _estimate_n_factor(latitude):
87              return ae / np.sqrt(1 - e2 * np.sin(latitude)**2)
88
89          def _estimate_height(n_factor, latitude):
90              return p_factor / np.cos(latitude) - n_factor
91
92          def _estimate_latitude(n_factor, altitude):
93              sin_lat = rz_s / (n_factor*(1-e2) + altitude)
94              return np.arctan((rz_s + e2*n_factor*sin_lat)/p_factor)
95
96          def _estimate_longitude():
97              return np.arctan2(ry_s, rx_s)
98
99          # Estimate recursively the values of altitude and latitude.
100         n_factor_old = ae
101         alt_old = 0
102         lat_old = _estimate_latitude(n_factor_old, alt_old)
103         dif = np.asarray([1])
104         # Repeat recursive method until a tolerance is reached.
105         while (dif > 0).all():
106             n_factor_new = np.where(
107                 dif > 0, _estimate_n_factor(lat_old), n_factor_old)
108             alt_new = np.where(
```

```
                dif > 0, _estimate_height(n_factor_new, lat_old), alt_old)
            lat_new = np.where(
                dif > 0, _estimate_latitude(n_factor_new, alt_new), lat_old)
            dif1 = np.abs(alt_new - alt_old)
            dif2 = np.abs(lat_new - lat_old)
            dif = dif1 + dif2 - GEODETIC_COORDINATES_TOLERANCE
            # Overwrite old temporary variables with new values.
            n_factor_old = n_factor_new
            alt_old = alt_new
            lat_old = lat_new
        # Compute longitude for a specific time.
        alt = alt_old
        lat = lat_old
        lon = _estimate_longitude()

        return np.hstack([lat, lon, alt])

    @classmethod
    def from_geo_to_ecf(cls, obj):
        """Compute ECF coordinates from geodetic coordinates.

        Parameters:

        obj
            (n, 3) array containing (x_ecef, y_ecef, z_ecef) in meters
        """

        # Call necessary constants.
        ae = EARTH_SEMIMAJOR_AXIS
        e2 = EARTH_FLATTENING_FACTOR
        # Call necessary properties.
        lat, lon, alt = [x[:, None] for x in obj.T]

        # Compute ECF coordinates fro GEO coordinates.
        n = ae / np.sqrt(1 - e2 * np.sin(lat)**2)
        rx_s = (n + alt) * np.cos(lat) * np.cos(lon)
        ry_s = (n + alt) * np.cos(lat) * np.sin(lon)
        rz_s = (n * (1 - e2) + alt) * np.sin(lat)

        return np.hstack([rx_s, ry_s, rz_s])
```

Código A.9: ../python-sat/sat/Error.py

```
"""Custom errors for classes from python-sat."""


class SatError(Exception):

    def __init__(self, message):
        """Constructor of a SatError instance."""
        self.message = message

    def __str__(self):
        """Generic text structure for printing a SatError."""
```

```
12
13          title = self.__class__.__name__
14          delstr = 26*"\b"
15          msgstr = "{}: {}".format(title, self.message).ljust(26)
16          return "{}{}".format(delstr, msgstr)
17
18
19  class EphemerisError(SatError):
20      pass
21
22
23  class FilepathError(SatError):
24      pass
25
26
27  class MatcherError(SatError):
28      pass
29
30
31  class OrbitError(SatError):
32      pass
33
34
35  class SceneError(SatError):
36      pass
```

A.2. Clase Ephemeris

Código A.10: ../python-sat/sat/constants/Ephemeris.py

```
"""Necessary constants for sat.Ephemeris class."""

from six import text_type


# Limit values for Ephemeris properties.
LIMITS_ECCENTRICITY =\
    ["[", 0.0, 1.0, "]"]
LIMITS_ELEMENT_SET_NUMBER =\
    ["[", 0, 9999, "]"]
LIMITS_EPOCH_REVOLUTION_NUMBER =\
    ["[", 0, 99999, "]"]
LIMITS_MEAN_MOTION_FIRST_DIF =\
    ["[", -1., +1., "]"]
LIMITS_SATELLITE_NUMBER =\
    ["[", 0, 99999, "]"]

# Patterns used within the class Ephemeris.
PATTERN_LAUNCH_PIECE =\
    text_type("^[A-Z]{1,3}$")
PATTERN_SATELLITE_CLASSIFICATION =\
    text_type("^[A-Z]$")
PATTERN_SATELLITE_NAME =\
    text_type("^[^\t\n\r\f\v]{1,24}$")

# Patterns for a two-line element set.
PATTERN_TITLE = text_type(
    "^[^a-z\t\n\r\f\v]{24}$")
PATTERN_LINE1 = text_type(
    "^1 {in5}{cap} {day}{cod} {day}{tim} {dec} {exp} {exp} 0 {end}{chk}$".
    format(cap="([A-Z])",
           chk="(\d)",
           cod="([A-Z] |[A-Z]{2} |[A-Z]{3})",
           day="(\d{2}(?:[0-2]\d{2}|3[0-6]\d|36[0-5]))",
           dec="([ |-]\.\d{8})",
           end="(\d{4}| \d{3}| \d{2}| \d)",
           exp="([ |-]\d{5}[+|-]\d)",
           in5="(\d{5})",
           tim="(\.\d{8})",))
PATTERN_LINE2 = text_type(
    "^2 {in5} {an1} {an2} {in7} {an2} {an2} {end}{chk}$".
    format(an1="((?: \d| \d{2}|1[0-7]\d).\d{4})",
           an2="((?: \d|[ 1-2]\d{2}|3[0-5]\d).\d{4})",
           chk="(\d)",
           end="((?: \d|\d{2})\.\d{8})( \d|  \d{2}| \d{3}| \d{4}|\d{5})",
           in5="(\d{5})",
           in7="(\d{7})",))
```

Código A.11: ../python-sat/sat/Ephemeris.py

```
from __future__ import division
from . constants.Ephemeris import LIMITS_ECCENTRICITY
from . constants.Ephemeris import LIMITS_ELEMENT_SET_NUMBER
from . constants.Ephemeris import LIMITS_EPOCH_REVOLUTION_NUMBER
from . constants.Ephemeris import LIMITS_MEAN_MOTION_FIRST_DIF
from . constants.Ephemeris import LIMITS_SATELLITE_NUMBER
from . constants.Ephemeris import PATTERN_LAUNCH_PIECE
from . constants.Ephemeris import PATTERN_SATELLITE_CLASSIFICATION
from . constants.Ephemeris import PATTERN_SATELLITE_NAME
from . constants.Ephemeris import PATTERN_TITLE
from . constants.Ephemeris import PATTERN_LINE1
from . constants.Ephemeris import PATTERN_LINE2
from . decorators import accepts
from . decorators import limits
from . decorators import pattern
from . decorators import returns
from . Angle import AziAngle
from . Angle import ZenAngle
from . Error import EphemerisError
from datetime import datetime
from datetime import date
from datetime import timedelta
from six import text_type
import re


class Ephemeris(object):

    __slots__ = [
        "_argument_of_perigee",
        "_drag_term",
        "_eccentricity",
        "_element_set_number",
        "_epoch_datetime",
        "_epoch_revolution_number",
        "_inclination",
        "_launch_date",
        "_launch_piece",
        "_mean_anomaly",
        "_mean_motion",
        "_mean_motion_first_dif",
        "_mean_motion_second_dif",
        "_longitude_of_the_ascending_node",
        "_satellite_classification",
        "_satellite_name",
        "_satellite_number",
        ]

    def __init__(self):
        """Constructor of a generic Ephemeris instance."""

        for item in self.__slots__:
            self.__setattr__(item, None)
```

```
54
55      def __bool__(self):
56          """Return True if all the instance attributes are well-defined."""
57
58          try:
59              flag = min(self.__getattribute__(item) is not None
60                         for item in self.__slots__)
61          except (AssertionError, TypeError, ValueError):
62              flag = False
63          return flag
64
65      def __repr__(self):
66          """Fancy representation of an Ephemeris instance."""
67
68          suplist = [
69              [item[1:].replace("_", " ") + ":", self.__getattribute__(item)]
70              for item in self.__slots__
71              ]
72          suplist = [
73              [x, y if not isinstance(y, (AziAngle, ZenAngle)) else y.deg]
74              for (x, y) in suplist
75              ]
76
77          text = (
78              "\nEphemeris information\n"
79              "\n  {:35s} {:.4f} deg"
80              "\n  {:35s} {:.4e} per Earth radii"
81              "\n  {:35s} {:.7f}"
82              "\n  {:35s} {}"
83              "\n  {:35s} {}"
84              "\n  {:35s} {}"
85              "\n  {:35s} {:.4f} deg"
86              "\n  {:35s} {}"
87              "\n  {:35s} {}"
88              "\n  {:35s} {:.4f} deg"
89              "\n  {:35s} {:.8f} rev per day"
90              "\n  {:35s} {:.4e} rev per day^2"
91              "\n  {:35s} {:.4e} rev per day^3"
92              "\n  {:35s} {:.4f} deg"
93              "\n  {:35s} {}"
94              "\n  {:35s} {}"
95              "\n  {:35s} {}").format(
96                 *[item for sublist in suplist for item in sublist])
97          return text
98
99      @staticmethod
100     @accepts(text_type, text_type, text_type, static=True)
101     def from_tle(title, line1, line2):
102         """Return an Ephemeris instance from a two-line element set.
103
104         Parameters:
105
106         title
107             two-line element set title
108         line1
```

```
109             first line of the two-line element set
110         line2
111             second line of the two-line element set
112         """
113
114         # Remove \n characters from string lines.
115         obj = Ephemeris()
116         lst = [x.replace("\n", "") for x in (title, line1, line2)]
117
118         # Verify title, line1 and line2 with regular expressions.
119         match0 = re.match(PATTERN_TITLE, lst[0])
120         match1 = re.match(PATTERN_LINE1, lst[1])
121         match2 = re.match(PATTERN_LINE2, lst[2])
122
123         if not match0:
124             msg = "invalid structure for TLE title"
125             raise EphemerisError(msg)
126         if not match1:
127             msg = "invalid structure for TLE line 1"
128             raise EphemerisError(msg)
129         if not match2:
130             msg = "invalid structure for TLE line 2"
131             raise EphemerisError(msg)
132
133         # Start transcription of title.
134         tmp0 = match0.group(0).strip()
135         obj.satellite_name = tmp0
136
137         # Verify checksums.
138         chk1 = [int(match1.group(11)), int(match2.group(9))]
139         chk2 = [obj._calc_checksum(row[:-1]) for row in lst[1:]]
140         if chk1 != chk2:
141             msg = "checksum error, found {}, expected {}".format(chk1, chk2)
142             raise EphemerisError(msg)
143
144         # Verify and transcript satellite number.
145         tmp1 = int(match1.group(1))
146         tmp2 = int(match2.group(1))
147         if tmp1 != tmp2:
148             msg = "satellite number within the lines does not match"
149             raise EphemerisError(msg)
150         obj.satellite_number = tmp1
151
152         # Start transcription of line1.
153         # Transcript satellite classification.
154         tmp1 = match1.group(2)
155         obj.satellite_classification = tmp1
156
157         # Transcript launch date.
158         tmp1 = datetime.strptime(match1.group(3), "%y%j").date()
159         obj.launch_date = tmp1
160
161         # Transcript launch piece.
162         tmp1 = match1.group(4).strip()
163         obj.launch_piece = tmp1
```

```
164
165         # Transcript epoch datetime.
166         day1 = match1.group(5)
167         day2 = match1.group(6)
168         tmp1 = datetime.strptime(day1, "%y%j") + timedelta(days=float(day2))
169         obj.epoch_datetime = tmp1
170
171         # Transcript first derivative of mean motion.
172         tmp1 = 2 * float(match1.group(7))
173         obj.mean_motion_first_dif = tmp1
174
175         # Transcript second derivative of mean motion.
176         tmp1 = match1.group(8)
177         tmp1 = 6 * float("{}.{}e{}".format(tmp1[0], tmp1[1:6], tmp1[6:]))
178         obj.mean_motion_second_dif = tmp1
179
180         # Transcript drag term.
181         tmp1 = match1.group(9)
182         tmp1 = float("{}.{}e{}".format(tmp1[0], tmp1[1:6], tmp1[6:]))
183         obj.drag_term = tmp1
184
185         # Transcript element set number.
186         tmp1 = int(match1.group(10))
187         obj.element_set_number = tmp1
188
189         # Start transcription of line2.
190         # Transcript inclination.
191         tmp2 = ZenAngle(deg=float(match2.group(2)))
192         obj.inclination = tmp2
193
194         # Transcript longitude of the ascending node.
195         tmp2 = AziAngle(deg=float(match2.group(3)))
196         obj.longitude_of_the_ascending_node = tmp2
197
198         # Transcript eccentricity.
199         tmp2 = float(".{}".format(match2.group(4)))
200         obj.eccentricity = tmp2
201
202         # Transcript argument of perigee.
203         tmp2 = AziAngle(deg=float(match2.group(5)))
204         obj.argument_of_perigee = tmp2
205
206         # Transcript mean anomaly.
207         tmp2 = AziAngle(deg=float(match2.group(6)))
208         obj.mean_anomaly = tmp2
209
210         # Transcript mean motion.
211         tmp2 = float(match2.group(7))
212         obj.mean_motion = tmp2
213
214         # Transcript revolution number at epoch.
215         tmp2 = int(match2.group(8))
216         obj._epoch_revolution_number = tmp2
217
218         # Return the Ephemeris instance.
```

```
219         return obj
220
221     def to_copy(self):
222         """Return a deep copy of the Ephemeris instance."""
223
224         obj = Ephemeris()
225         for item in self.__slots__:
226             obj.__setattr__(item, self.__getattribute__(item))
227         return obj
228
229     def to_tle(self):
230         """Return a two-line element set as a list."""
231
232         if not self:
233             msg = "Ephemeris instance is not complete"
234             raise AttributeError(msg)
235
236         title = self.satellite_name.ljust(24)
237         line1 = text_type(
238             "1 {:5d}{:1s} {:5s}{:3s} {:5s}{:9s} {:1s}{:9s} {:8s} {:8s} 0 {:4d}".
239             format(self.satellite_number,
240                    self.satellite_classification,
241                    self.launch_date.strftime("%y%j"),
242                    self.launch_piece.ljust(3),
243                    self.epoch_datetime.strftime("%y%j"),
244                    "{:10.8f}".format(
245                        timedelta(0,
246                                  self.epoch_datetime.second,
247                                  self.epoch_datetime.microsecond,
248                                  0,
249                                  self.epoch_datetime.minute,
250                                  self.epoch_datetime.hour,
251                                  0,
252                                  ).total_seconds()/86400)[1:],
253                    "-" if self.mean_motion_first_dif < 0 else " ",
254                    "{:10.8f}".format(abs(self.mean_motion_first_dif)/2)[1:],
255                    " 00000-0" if self.mean_motion_second_dif == 0 else
256                    "{: =06d}{:+d}".
257                    format(*[[int(round(float(x)*10000)), int(y)+1] for x, y in
258                             ["{:.5e}".format(
259                              self.mean_motion_second_dif/6).split("e")]][0]),
260                    " 00000-0" if self.drag_term == 0 else
261                    "{: =06d}{:+d}".
262                    format(*[[int(round(float(x)*10000)), int(y)+1] for x, y in
263                             ["{:.5e}".format(
264                              self.drag_term).split("e")]][0]),
265                    self.element_set_number))
266         line2 = text_type(
267             "2 {:5d} {:8.4f} {:8.4f} {:7s} {:8.4f} {:8.4f} {:11.8f}{:5d}".
268             format(self.satellite_number,
269                    self.inclination.deg,
270                    self.longitude_of_the_ascending_node.deg,
271                    "{:9.7f}".format(self.eccentricity)[2:],
272                    self.argument_of_perigee.deg,
273                    self.mean_anomaly.deg,
```

```
                    self.mean_motion,
                    self.epoch_revolution_number))
        line1 = "".join([line1, text_type(self._calc_checksum(line1))])
        line2 = "".join([line2, text_type(self._calc_checksum(line2))])
        return [title, line1, line2]

    @property
    @returns(AziAngle)
    def argument_of_perigee(self):
        """argument of perigee stored as an AziAngle instance"""
        return self._argument_of_perigee

    @argument_of_perigee.setter
    @accepts(AziAngle)
    def argument_of_perigee(self, val):
        self._argument_of_perigee = val

    @argument_of_perigee.deleter
    def argument_of_perigee(self):
        self._argument_of_perigee = None

    @property
    @returns(float)
    def drag_term(self):
        """BSTAR radiation pressure coefficient in per Earth radii"""
        return self._drag_term

    @drag_term.setter
    @accepts(float)
    def drag_term(self, val):
        self._drag_term = val

    @drag_term.deleter
    def drag_term(self):
        self._drag_term = None

    @property
    @returns(float)
    @limits(*LIMITS_ECCENTRICITY, getter=True)
    def eccentricity(self):
        """orbital eccentricity"""
        return self._eccentricity

    @eccentricity.setter
    @accepts(float)
    @limits(*LIMITS_ECCENTRICITY)
    def eccentricity(self, val):
        self._eccentricity = val

    @eccentricity.deleter
    def eccentricity(self):
        self._eccentricity = None

    @property
    @returns(int)
```

```
329     @limits(*LIMITS_ELEMENT_SET_NUMBER, getter=True)
330     def element_set_number(self):
331         """element set number, incremented when a new two-line element
332         set is generated for the satellite"""
333         return self._element_set_number
334 
335     @element_set_number.setter
336     @accepts(int)
337     @limits(*LIMITS_ELEMENT_SET_NUMBER)
338     def element_set_number(self, val):
339         self._element_set_number = val
340 
341     @element_set_number.deleter
342     def element_set_number(self):
343         self._element_set_number = None
344 
345     @property
346     @returns(datetime)
347     def epoch_datetime(self):
348         """datetime for the current epoch"""
349         return self._epoch_datetime
350 
351     @epoch_datetime.setter
352     @accepts(datetime)
353     def epoch_datetime(self, val):
354         self._epoch_datetime = val
355 
356     @epoch_datetime.deleter
357     def epoch_datetime(self):
358         self._epoch_datetime = None
359 
360     @property
361     @returns(int)
362     @limits(*LIMITS_EPOCH_REVOLUTION_NUMBER, getter=True)
363     def epoch_revolution_number(self):
364         """revolution number corresponding to the epoch"""
365         return self._epoch_revolution_number
366 
367     @epoch_revolution_number.setter
368     @accepts(int)
369     @limits(*LIMITS_EPOCH_REVOLUTION_NUMBER)
370     def epoch_revolution_number(self, val):
371         self._epoch_revolution_number = val
372 
373     @epoch_revolution_number.deleter
374     def epoch_revolution_number(self):
375         self._epoch_revolution_number = None
376 
377     @property
378     @returns(ZenAngle)
379     def inclination(self):
380         """orbital inclination as an ZenAngle instance"""
381         return self._inclination
382 
383     @inclination.setter
```

```
384     @accepts(ZenAngle)
385     def inclination(self, val):
386         self._inclination = val
387 
388     @inclination.deleter
389     def inclination(self):
390         self._inclination = None
391 
392     @property
393     @returns(date)
394     def launch_date(self):
395         """date of launch of the satellite"""
396         return self._launch_date
397 
398     @launch_date.setter
399     @accepts(date)
400     def launch_date(self, val):
401         self._launch_date = val
402 
403     @launch_date.deleter
404     def launch_date(self):
405         self._launch_date = None
406 
407     @property
408     @returns(text_type)
409     @pattern(PATTERN_LAUNCH_PIECE, getter=True)
410     def launch_piece(self):
411         """piece of launch which corresponds to the satellite"""
412         return self._launch_piece
413 
414     @launch_piece.setter
415     @accepts(text_type)
416     @pattern(PATTERN_LAUNCH_PIECE)
417     def launch_piece(self, val):
418         self._launch_piece = val
419 
420     @launch_piece.deleter
421     def launch_piece(self):
422         self._launch_piece = None
423 
424     @property
425     @returns(AziAngle)
426     def longitude_of_the_ascending_node(self):
427         """longitude of the ascending node stored as an AziAngle instance"""
428         return self._longitude_of_the_ascending_node
429 
430     @longitude_of_the_ascending_node.setter
431     @accepts(AziAngle)
432     def longitude_of_the_ascending_node(self, val):
433         self._longitude_of_the_ascending_node = val
434 
435     @longitude_of_the_ascending_node.deleter
436     def longitude_of_the_ascending_node(self):
437         self._longitude_of_the_ascending_node = None
438 
```

```
439     @property
440     @returns(AziAngle)
441     def mean_anomaly(self):
442         """mean anomaly of the orbit stored as an AziAngle instance"""
443         return self._mean_anomaly
444 
445     @mean_anomaly.setter
446     @accepts(AziAngle)
447     def mean_anomaly(self, val):
448         self._mean_anomaly = val
449 
450     @mean_anomaly.deleter
451     def mean_anomaly(self):
452         self._mean_anomaly = None
453 
454     @property
455     @returns(float)
456     def mean_motion(self):
457         """mean motion (time-average angular velocity) of the satellite
458         over an orbit in revs per day"""
459         return self._mean_motion
460 
461     @mean_motion.setter
462     @accepts(float)
463     def mean_motion(self, val):
464         self._mean_motion = val
465 
466     @mean_motion.deleter
467     def mean_motion(self):
468         self._mean_motion = None
469 
470     @property
471     @returns(float)
472     @limits(*LIMITS_MEAN_MOTION_FIRST_DIF, getter=True)
473     def mean_motion_first_dif(self):
474         """first time derivative of the mean motion in revs per day^2"""
475         return self._mean_motion_first_dif
476 
477     @mean_motion_first_dif.setter
478     @accepts(float)
479     @limits(*LIMITS_MEAN_MOTION_FIRST_DIF)
480     def mean_motion_first_dif(self, val):
481         self._mean_motion_first_dif = val
482 
483     @mean_motion_first_dif.deleter
484     def mean_motion_first_dif(self):
485         self._mean_motion_first_dif = None
486 
487     @property
488     @returns(float)
489     def mean_motion_second_dif(self):
490         """second time derivative of the mean motion in revs per day^3"""
491         return self._mean_motion_second_dif
492 
493     @mean_motion_second_dif.setter
```

```
494     @accepts(float)
495     def mean_motion_second_dif(self, val):
496         self._mean_motion_second_dif = val
497 
498     @mean_motion_second_dif.deleter
499     def mean_motion_second_dif(self):
500         self._mean_motion_second_dif = None
501 
502     @property
503     @returns(text_type)
504     @pattern(PATTERN_SATELLITE_CLASSIFICATION, getter=True)
505     def satellite_classification(self):
506         """classification of the satellite (U = unclassified)"""
507         return self._satellite_classification
508 
509     @satellite_classification.setter
510     @accepts(text_type)
511     @pattern(PATTERN_SATELLITE_CLASSIFICATION)
512     def satellite_classification(self, val):
513         self._satellite_classification = val
514 
515     @satellite_classification.deleter
516     def satellite_classification(self):
517         self._satellite_classification = None
518 
519     @property
520     @returns(text_type)
521     @pattern(PATTERN_SATELLITE_NAME, getter=True)
522     def satellite_name(self):
523         """name of the satellite"""
524         return self._satellite_name
525 
526     @satellite_name.setter
527     @accepts(text_type)
528     @pattern(PATTERN_SATELLITE_NAME)
529     def satellite_name(self, val):
530         self._satellite_name = val
531 
532     @satellite_name.deleter
533     def satellite_name(self):
534         self._satellite_name = None
535 
536     @property
537     @returns(int)
538     @limits(*LIMITS_SATELLITE_NUMBER, getter=True)
539     def satellite_number(self):
540         """number code which identifies the satellite"""
541         return self._satellite_number
542 
543     @satellite_number.setter
544     @accepts(int)
545     @limits(*LIMITS_SATELLITE_NUMBER)
546     def satellite_number(self, val):
547         self._satellite_number = val
548 
```

```
549     @satellite_number.deleter
550     def satellite_number(self):
551         self._satellite_number = None
552 
553     @accepts(text_type)
554     def _calc_checksum(self, line):
555         """Calculate the checksum mod 10 of a TLE line."""
556 
557         nums = re.findall("\d", line.replace("-", "1"))
558         return sum(int(x) for x in nums) % 10 if nums else 0
```

A.3. Clase Orbit

Código A.12: ../python-sat/sat/constants/Orbit.py

```
"""Necessary constants for sat.Orbit class."""


# Conversion factor from days to seconds.
DAY2SEC =\
    86400

# Anomaly tolerance in rad.
ECCENTRIC_ANOMALY_TOLERANCE =\
    1e-9

# Coefficient for the second zonal term, related to the oblateness of the
# Earth, expressed in m^5 per s^2.
SECOND_ZONAL_TERM_J2 =\
    1.7555e25

# Conversion factor from mean solar time to mean sidereal time.
SOL2SID =\
    1.00273790935

# Standard gravitational parameter, that is, the product of the gravitational
# constant G and the Earth's mass M, expressed in m^3 per s^2.
STANDARD_GRAVITATIONAL_PARAMETER =\
    3.986004418e14
```

Código A.13: ../python-sat/sat/Orbit.py

```
from __future__ import division
from . constants.Angle import REV2RAD
from . constants.Coordinates import EARTH_ANGULAR_SPEED
from . constants.Orbit import DAY2SEC
from . constants.Orbit import ECCENTRIC_ANOMALY_TOLERANCE
from . constants.Orbit import SECOND_ZONAL_TERM_J2
from . constants.Orbit import STANDARD_GRAVITATIONAL_PARAMETER
from . decorators import accepts
from . decorators import returns
from . Coordinates import Coordinates
from . Ephemeris import Ephemeris
from . Error import OrbitError
import numpy as np


class Orbit(object):

    __slots__ = [
        "_argument_of_perigee",
        "_datetime",
        "_eccentric_anomaly",
```

```
22          "_ephemeris",
23          "_longitude_of_the_ascending_node",
24          "_mean_anomaly",
25          "_mean_motion",
26          "_position_ecf",
27          "_position_eci",
28          "_position_geo",
29          "_semimajor_axis",
30          "_timedelta",
31          "_true_anomaly",
32          "_velocity_ecf",
33          "_velocity_eci",
34          ]
35
36      @accepts(Ephemeris)
37      def __init__(self, ephemeris):
38          """Constructor of a generic Orbit instance.
39
40          Parameters:
41
42          ephemeris
43              Ephemeris instance containing the Kepler parameters
44          """
45
46          for item in self.__slots__:
47              self.__setattr__(item, None)
48
49          if ephemeris:
50              self._ephemeris = ephemeris.to_copy()
51          else:
52              msg = "Ephemeris instance is not complete"
53              raise OrbitError(msg)
54
55      @accepts(np.ndarray)
56      def compute(self, datetime):
57          """Compute elliptical orbit for specified datetime.
58
59          Parameters:
60
61          datetime
62              datetime for predicted elliptical orbit parameters
63          """
64
65          sup = self.__class__
66          sup._add_datetime(self, datetime)
67          sup._calc_timedelta(self)
68          sup._calc_mean_motion(self)
69          sup._calc_mean_anomaly(self)
70          sup._calc_eccentric_anomaly(self)
71          sup._calc_true_anomaly(self)
72          sup._calc_semimajor_axis(self)
73          sup._calc_argument_of_perigee_and_longitude_of_the_ascending_node(self)
74          sup._calc_coordinates_from_orb_to_eci(self)
75          sup._calc_coordinates_from_eci_to_ecf(self)
76          sup._calc_coordinates_from_ecf_to_geo(self)
```

```
77
78      @classmethod
79      def _add_datetime(cls, obj, date):
80          """Safe setter for datetime attribute."""
81
82          try:
83              obj._datetime = np.asarray([
84                  x.strftime("%Y-%m-%dT%H:%M:%S.%fZ")
85                  for x in date.ravel()], dtype="datetime64")[:, None]
86          except (TypeError, ValueError):
87              msg = "Invalid array for input time"
88              err = OrbitError(msg)
89              err.__cause__ = None
90              raise err
91
92      @classmethod
93      def _calc_timedelta(cls, obj):
94          """Compute time difference with respect to epoch."""
95
96          try:
97              fmt = "%Y-%m-%dT%H:%M:%S.%fZ"
98              epoch = np.asarray([
99                  obj.ephemeris.epoch_datetime.strftime(fmt)
100                 ]).astype("datetime64")
101             delta = (obj.datetime - epoch)
102             obj._timedelta = (delta / np.timedelta64(1, "D")).astype(float)
103         except TypeError:
104             msg = "Ephemeris attribute from Orbit instance is not complete"
105             err = OrbitError(msg)
106             err.__cause__ = None
107             raise err
108
109     @classmethod
110     def _calc_mean_motion(cls, obj):
111         """Compute mean motion for a specific datetime."""
112
113         # Call necessary properties.
114         dt = obj.timedelta
115         n0 = obj.ephemeris.mean_motion
116         n1 = obj.ephemeris.mean_motion_first_dif
117         # Set mean motion property.
118         obj._mean_motion = n0 + n1 * dt
119
120     @classmethod
121     def _calc_mean_anomaly(cls, obj):
122         """Compute mean anomaly for a specific datetime."""
123
124         # Call necessary properties.
125         dt = obj.timedelta
126         m0 = obj.ephemeris.mean_anomaly.rad
127         n = obj.mean_motion * REV2RAD
128         # Set mean anomaly property.
129         obj._mean_anomaly = m0 + n * dt
130
131     @classmethod
```

```
132     def _calc_eccentric_anomaly(cls, obj):
133         """Compute eccentric anomaly for a specific datetime."""
134 
135         # Call necessary properties.
136         eccent = obj.ephemeris.eccentricity
137         m = obj.mean_anomaly
138         ean_old = m
139         dif = np.asarray([1])
140         # Solve Kepler's equation by recursive methods until a tolerance
141         # is reached.
142         while (dif > 0).all():
143             ean_new = np.where(dif > 0, m + eccent*np.sin(ean_old), ean_old)
144             dif = np.abs(ean_new - ean_old) - ECCENTRIC_ANOMALY_TOLERANCE
145             ean_old = ean_new
146         # Set eccentric anomaly property.
147         obj._eccentric_anomaly = ean_old
148 
149     @classmethod
150     def _calc_true_anomaly(cls, obj):
151         """Compute true anomaly for a specific datetime."""
152 
153         # Call necessary properties.
154         eccent = obj.ephemeris.eccentricity
155         factor = np.sqrt((1+eccent)/(1-eccent))
156         ean = obj.eccentric_anomaly
157         # Set true anomaly property.
158         obj._true_anomaly = 2 * np.arctan(factor * np.tan(ean/2))
159 
160     @classmethod
161     def _calc_semimajor_axis(cls, obj):
162         """Compute semimajor axis for a specific datetime."""
163 
164         # Call necessary constants and properties.
165         mu = STANDARD_GRAVITATIONAL_PARAMETER
166         n = obj.mean_motion * REV2RAD / DAY2SEC
167         # Set semimajor axis property using Kepler's third law.
168         obj._semimajor_axis = (mu / n**2)**(1/3)
169 
170     @classmethod
171     def _calc_argument_of_perigee_and_longitude_of_the_ascending_node(cls, obj):
172         """Compute argument of perigee and longitude of the ascending
173         node for a specific datetime."""
174 
175         # Call necessary constants.
176         j2 = SECOND_ZONAL_TERM_J2
177         mu = STANDARD_GRAVITATIONAL_PARAMETER
178         # Call necessary properties.
179         dt = obj.timedelta
180         i = obj.ephemeris.inclination.rad
181         n = obj.mean_motion
182         eccent = obj.ephemeris.eccentricity
183         a = obj.semimajor_axis
184         p = a * (1-eccent**2)
185         # Compute first derivatives of argument of perigee and longitude
186         # of the ascending node.
```

```
187         factor = -REV2RAD * j2 / (mu*p**2) * 3 * n
188         w0 = obj.ephemeris.argument_of_perigee.rad
189         w1 = factor * ((5/4)*np.sin(i)**2 - 1)
190         o0 = obj.ephemeris.longitude_of_the_ascending_node.rad
191         o1 = (factor/2) * np.cos(i)
192         # Set argument of perigee property.
193         obj._argument_of_perigee = w0 + w1 * dt
194         # Set longitude of the ascending node property.
195         obj._longitude_of_the_ascending_node = o0 + o1 * dt
196 
197     @classmethod
198     def _calc_coordinates_from_orb_to_eci(cls, obj):
199         """Compute ECI coordinates for a specific datetime."""
200 
201         # Call necessary constants.
202         mu = STANDARD_GRAVITATIONAL_PARAMETER
203         # Call necessary properties.
204         i = obj.ephemeris.inclination.rad
205         o = obj.longitude_of_the_ascending_node
206         w = obj.argument_of_perigee
207         v = obj.true_anomaly
208         eccent = obj.ephemeris.eccentricity
209         a = obj.semimajor_axis
210         # Compute intermediate factors.
211         p = a * (1 - eccent**2)
212         r = p / (1 + eccent * np.cos(v))
213         l = np.sqrt(mu * p)
214         # Compute (X,Y,Z) coordinates in ECI reference system.
215         rx = r * (np.cos(o)*np.cos(w+v) - np.sin(o)*np.sin(w+v)*np.cos(i))
216         ry = r * (np.sin(o)*np.cos(w+v) + np.cos(o)*np.sin(w+v)*np.cos(i))
217         rz = r * (np.sin(w+v)*np.sin(i))
218         # Compute (Vx,Vy,Vz) velocities in ECI reference system.
219         vx = (rx * l * eccent)/(r * p) * np.sin(v) - l/r * (
220             np.cos(o)*np.sin(w+v) + np.sin(o)*np.cos(w+v)*np.cos(i))
221         vy = (ry * l * eccent)/(r * p) * np.sin(v) - l/r * (
222             np.sin(o)*np.sin(w+v) - np.cos(o)*np.cos(w+v)*np.cos(i))
223         vz = (rz * l * eccent)/(r * p) * np.sin(v) + l/r * (
224             np.cos(w+v)*np.sin(i))
225         # Set ECI satellite position and velocity properties.
226         obj._position_eci = np.hstack([rx, ry, rz])
227         obj._velocity_eci = np.hstack([vx, vy, vz])
228 
229     @classmethod
230     def _calc_coordinates_from_eci_to_ecf(cls, obj):
231         """Compute ECF coordinates for a specific datetime."""
232 
233         rx, ry, rz = [x[:, None] for x in obj.position_eci.T]
234         vx, vy, vz = [x[:, None] for x in obj.velocity_eci.T]
235         vx += EARTH_ANGULAR_SPEED * ry
236         vy -= EARTH_ANGULAR_SPEED * rx
237 
238         obj._position_ecf = Coordinates.from_eci_to_ecf(
239             np.hstack([rx, ry, rz]), obj.datetime)
240         obj._velocity_ecf = Coordinates.from_eci_to_ecf(
241             np.hstack([vx, vy, vz]), obj.datetime)
```

```
242
243     @classmethod
244     def _calc_coordinates_from_ecf_to_geo(cls, obj):
245         """Compute geodetic coordinates for a specific datetime."""
246         obj._position_geo = Coordinates.from_ecf_to_geo(obj.position_ecf)
247
248     @classmethod
249     def _calc_coordinates_from_geo_to_ecf(cls, obj):
250         """Compute ECF coordinates using geodetic coordinates."""
251         obj._position_ecf = Coordinates.from_geo_to_ecf(obj.position_geo)
252
253     @property
254     @returns(np.ndarray)
255     def argument_of_perigee(self):
256         """argument of perigee in radians"""
257         return self._argument_of_perigee
258
259     @property
260     @returns(np.ndarray)
261     def datetime(self):
262         """datetime for predicted elliptical orbit parameters"""
263         return self._datetime
264
265     @property
266     @returns(np.ndarray)
267     def eccentric_anomaly(self):
268         """eccentric anomaly in radians"""
269         return self._eccentric_anomaly
270
271     @property
272     @returns(Ephemeris)
273     def ephemeris(self):
274         """Ephemeris object containing the reference orbital parameters"""
275         return self._ephemeris
276
277     @property
278     @returns(np.ndarray)
279     def longitude_of_the_ascending_node(self):
280         """longitude of the ascending node in radians"""
281         return self._longitude_of_the_ascending_node
282
283     @property
284     @returns(np.ndarray)
285     def mean_anomaly(self):
286         """mean anomaly in radians"""
287         return self._mean_anomaly
288
289     @property
290     @returns(np.ndarray)
291     def mean_motion(self):
292         """mean motion in revs per day"""
293         return self._mean_motion
294
295     @property
296     @returns(np.ndarray)
```

```
297     def position_ecf(self):
298         """satellite position in ECF reference system"""
299         return self._position_ecf
300 
301     @property
302     @returns(np.ndarray)
303     def position_eci(self):
304         """satellite position in ECI reference system"""
305         return self._position_eci
306 
307     @property
308     @returns(np.ndarray)
309     def position_geo(self):
310         """satellite position in geodetic reference system"""
311         return self._position_geo
312 
313     @property
314     @returns(np.ndarray)
315     def velocity_ecf(self):
316         """satellite velocity in ECF reference system"""
317         return self._velocity_ecf
318 
319     @property
320     @returns(np.ndarray)
321     def velocity_eci(self):
322         """satellite velocity in ECI reference system"""
323         return self._velocity_eci
324 
325     @property
326     @returns(np.ndarray)
327     def semimajor_axis(self):
328         """semimajor axis of the elliptic orbit in meters"""
329         return self._semimajor_axis
330 
331     @property
332     @returns(np.ndarray)
333     def timedelta(self):
334         """increment of time since the epoch, in days"""
335         return self._timedelta
336 
337     @property
338     @returns(np.ndarray)
339     def true_anomaly(self):
340         """true anomaly in radians"""
341         return self._true_anomaly
```

A.4. Clase Scene

Código A.14: ../python-sat/sat/constants/Scene.py

```
"""Necessary constants for sat.Scene class."""

from six import text_type


# Set tolerance for recursive processes within sat.Scene class.
DATASET_COORDINATE_TOLERANCE =\
    1e-9

# GDAL dataset driver for AVHRR images.
DATASET_DRIVER = text_type(
    "NOAA Polar Orbiter Level 1b Data Set")

# Date pattern for AVHRR images opened with GDAL.
DATE_PATTERN = text_type(
    "^year: (\d{4}), day: (\d{1,3}), millisecond: (\d{1,8})$")

# Default scan step angle for AVHRR sensor in degrees.
DEFAULT_SCAN_STEP_LOOK_ANGLE =\
    0.053910

# Margin value when obtaining the border mask of a gridded indices array.
HPRT_BORDER_MASK_MARGIN =\
    200

# Threshold values for cloud masking of HRPT images. They are chosen to
# overestimate the presence of clouds, as cloud mask is used to remove non-valid
# key points from key-feature extraction.
HRPT_CLOUD_THRESHOLD_B5 =\
    500

# Default steps when georeferencing and HRPT image into geographic coordinates.
HRPT_LATITUDE_STEP =\
    -0.01
HRPT_LONGITUDE_STEP =\
    +0.01

# Constants value which determine the spacecraft's direction.
SPACECRAFT_ASCENDING =\
    +1
SPACECRAFT_DESCENDING =\
    -1
```

Código A.15: ../python-sat/sat/Scene.py

```
from __future__ import division
from __future__ import print_function
from . constants.Angle import DEG2RAD
```

```
4  from . constants.Coordinates import EARTH_FLATTENING_FACTOR
5  from . constants.Coordinates import EARTH_SEMIMAJOR_AXIS
6  from . constants.Scene import DATASET_COORDINATE_TOLERANCE
7  from . constants.Scene import DATASET_DRIVER
8  from . constants.Scene import DATE_PATTERN
9  from . constants.Scene import DEFAULT_SCAN_STEP_LOOK_ANGLE
10 from . constants.Scene import HPRT_BORDER_MASK_MARGIN
11 from . constants.Scene import HRPT_LATITUDE_STEP
12 from . constants.Scene import HRPT_LONGITUDE_STEP
13 from . constants.Scene import HRPT_CLOUD_THRESHOLD_B5
14 from . constants.Scene import SPACECRAFT_ASCENDING
15 from . constants.Scene import SPACECRAFT_DESCENDING
16 from . decorators import accepts
17 from . decorators import returns
18 from . Angle import Angle
19 from . Coordinates import Coordinates
20 from . Ephemeris import Ephemeris
21 from . Error import SceneError
22 from . Orbit import Orbit
23 from datetime import datetime
24 from datetime import timedelta
25 from scipy.interpolate import griddata
26 from scipy.interpolate import NearestNDInterpolator
27 from scipy.interpolate import RectBivariateSpline
28 from scipy.ndimage import binary_dilation
29 from scipy.ndimage import binary_erosion
30 from scipy.ndimage import distance_transform_edt
31 from scipy.sparse import coo_matrix
32 from six import text_type
33 import gdal
34 import numpy as np
35 import re
36
37
38 class Scene(object):
39
40     __slots__ = [
41         "_dataset",
42         "_clock_drift",
43         "_end_datetime",
44         "_gcp_position_ecf",
45         "_gcp_position_geo",
46         "_gcp_position_pix",
47         "_gcp_xrange",
48         "_gcp_yrange",
49         "_hrpt_geo",
50         "_hrpt_img",
51         "_hrpt_msk",
52         "_hrpt_gtr",
53         "_hrpt_pix",
54         "_orbit",
55         "_scan_step_look_angle",
56         "_scan_timedelta",
57         "_scan_xsize",
58         "_scan_ysize",
```

```
59          "_spacecraft_attitude",
60          "_spacecraft_direction",
61          "_spacecraft_fixed_look_angle",
62          "_spacecraft_fixed_unit_look_vector",
63          "_spacecraft_geodetic_unit_vector",
64          "_start_datetime",
65          ]
66
67      @accepts(text_type, Ephemeris)
68      def __init__(self, path, ephemeris):
69          """Constructor of a generic Scene instance.
70
71          Parameters:
72
73          path
74              string path where the image is stored
75          ephemeris
76              Ephemeris instance containing the Kepler parameters
77          """
78
79          # Check that the dataset is valid.
80          gdal.PushErrorHandler("CPLQuietErrorHandler")
81          dataset = gdal.Open(path)
82          if not dataset:
83              msg = "Invalid path for image file"
84              raise SceneError(msg)
85          if dataset.GetDriver().LongName != DATASET_DRIVER:
86              msg = "Image file must be a {}".format(DATASET_DRIVER)
87              raise SceneError(msg)
88          for item in self.__slots__:
89              self.__setattr__(item, None)
90          # Build Orbit instance from list of TLE lines.
91          orbit = Orbit(ephemeris.to_copy())
92          # Set main attributes.
93          self._dataset = dataset
94          self._orbit = orbit
95
96      @accepts(int, int)
97      def compute(self, x_density, y_density):
98          """Compute scene coordinates for corresponding datetime.
99
100         Parameters:
101
102         x_density
103             x spacing between ground control points
104         y_density
105             y spacing between ground control points
106         """
107
108         # Proceed only if the Scene instance's attributes are well-defined.
109         try:
110             # Check that point density is valid both along and across track.
111             if x_density > (self.scan_xsize // 10):
112                 msg = "{} (minimum {}, found {})".format(
113                     "Insufficient across-track point density",
```

```
                    self.scan_xsize // 4,
                    x_density)
                raise SceneError(msg)
            if y_density > (self.scan_ysize // 10):
                msg = "{} (minimum {}, found {})".format(
                    "Insufficient along-track point density",
                    self.scan_ysize // 4,
                    y_density)
                raise SceneError(msg)
            # Set control point indices.
            self._gcp_xrange = np.arange(
                0.5 + (self.scan_xsize // 2) % x_density,
                self.scan_xsize,
                x_density)
            self._gcp_yrange = np.arange(
                0.5 + (self.scan_ysize // 2) % y_density,
                self.scan_ysize,
                y_density)
            # Compute expected datetimes for every scan line and propagate
            # attributes from Orbit property.
            times = np.asarray([
                self.start_datetime - self.clock_drift + n * self.scan_timedelta
                for n in range(self.scan_ysize)])[:, None]
            self.orbit.compute(times)
        except AttributeError:
            parent = self.__class__.__name__
            msg = "{} attributes are not complete".format(parent)
            err = SceneError(msg)
            err.__cause__ = None
            raise err
        # Call necessary constants.
        ae = EARTH_SEMIMAJOR_AXIS
        be = ae * np.sqrt(1 - EARTH_FLATTENING_FACTOR)
        # Call necessary properties.
        r_vec = self.orbit.position_ecf
        u_orb = self._compute_spacecraft_fixed_unit_look_vectors()
        w_xyz = self._compute_spacecraft_geodetic_unit_vectors()
        # Read attitude angles and build rotation array.
        roll, pitch, yaw = self.spacecraft_attitude
        rot_a = np.array([[1, pitch, -roll], [-pitch, 1, yaw], [roll, -yaw, 1]])
        # Create temporary arrays where ECF and geodetic coordinates will be
        # stored during the computation.
        xnum = len(self.gcp_xrange)
        ynum = len(self.gcp_yrange)
        xran = self.gcp_xrange.astype(int)
        tmp_ecf = np.empty((xnum*ynum, 3))
        tmp_geo = np.empty((xnum*ynum, 3))
        tmp_pix = np.empty((xnum*ynum, 2))
        for i, j in enumerate(self.gcp_yrange):
            rot_t = w_xyz[:, :, j]
            u_xyz = np.dot(np.dot(rot_t.T, rot_a.T), u_orb[:, xran])
            # Estimate recursively the ECF coordinates for every image's pixel.
            r_vec_ii = r_vec[j][:, None]
            pix_norm = ae
            sat_norm = np.linalg.norm(r_vec_ii, axis=0)[None, :]
```

```
169             dotru = np.sum(r_vec_ii * u_xyz, axis=0)[None, :]
170             dif = np.array([1])
171             # Repeat recursive method until a tolerance is reached.
172             while (dif > DATASET_COORDINATE_TOLERANCE).all():
173                 # Calculate new ECF vectors for every pixel.
174                 d_norm = - dotru - np.sqrt(pix_norm**2 - sat_norm**2 + dotru**2)
175                 pix_vec_ii = d_norm * u_xyz + r_vec_ii
176                 # Update image position coordinates.
177                 self._gcp_position_ecf = pix_vec_ii.T
178                 self._gcp_position_geo = Coordinates.from_ecf_to_geo(
179                     self.gcp_position_ecf)
180                 # Compute new norm for pixel vectors.
181                 lat = self.gcp_position_geo[:, 0]
182                 pix_norm2 = np.sqrt(
183                     ((ae**2 * np.cos(lat))**2 + (be**2 * np.sin(lat))**2) /
184                     ((ae * np.cos(lat))**2 + (be * np.sin(lat))**2))
185                 # Overwrite old temporary variables with new values.
186                 dif = (pix_norm2 - pix_norm)**2
187                 pix_norm = pix_norm2
188             # Grow temporary variables which store ECF and geodetic coordinates.
189             tmp_ecf[i*xnum:(i+1)*xnum, :] = self.gcp_position_ecf
190             tmp_geo[i*xnum:(i+1)*xnum, :] = self.gcp_position_geo
191             tmp_pix[i*xnum:(i+1)*xnum, :] = np.asarray([
192                 j * np.ones((xnum,)), self.gcp_xrange]).T
193         # Update ECF and geodetic coordinates.
194         self._gcp_position_ecf = tmp_ecf
195         self._gcp_position_geo = tmp_geo
196         self._gcp_position_pix = tmp_pix
197 
198     def build_geographic_window(self, slices=400, buffer=100, fill_value=-999):
199         """Calc geodetic coordinates for all the hrpt image and build
200         geodetic grid for pixel indices and its geotransform.
201 
202         Optional parameters:
203 
204         slices
205             number of rows in which the method is split (default 400)
206         buffer
207             number of overlapping rows between two slices (default 100)
208         fill_value
209             no data value (default -999)
210         """
211 
212         # Verify the types and ranges of all the input arguments.
213         if not isinstance(slices, int):
214             msg = "slices must be a positive int"
215             raise SceneError(msg)
216         if slices < 0 or slices >= self.scan_ysize:
217             msg = "slices must be positive and not greater than image length"
218             raise SceneError(msg)
219         if not isinstance(buffer, int):
220             msg = "buffer must be a positive int"
221             raise SceneError(msg)
222         if buffer < 0 or buffer >= self.scan_ysize:
223             msg = "buffer must be positive and not greater than image length"
```

```
224             raise SceneError(msg)
225         if not isinstance(fill_value, int):
226             msg = "fill value must be an int"
227             raise SceneError(msg)
228 
229         # Show start message.
230         msg = "\nComputing HRPT attributes for {} instance with id {}"
231         print(msg.format(self.__class__.__name__, id(self)))
232 
233         # Call necessary properties.
234         xstep = 0.5 + np.arange(self.scan_xsize)
235         ystep = 0.5 + np.arange(self.scan_ysize)
236 
237         # Show info message.
238         msg = "\n Calculating geodetic coordinates for all the HRPT pixels"
239         print(msg)
240 
241         def calc_hrpt_geo():
242             """Inner function where self._hrpt_geo is computed."""
243 
244             # Call necessary properties.
245             k = 3
246             yran = self.gcp_yrange
247             xran = self.gcp_xrange
248             gcps = self.gcp_position_geo[:, :2] / DEG2RAD
249             gcps = np.rollaxis(gcps.reshape((len(yran), len(xran), 2)), 2)
250             # Compute latitude and longitude interpolators.
251             lat_spl = RectBivariateSpline(x=yran, y=xran, z=gcps[0], kx=k, ky=k)
252             lon_spl = RectBivariateSpline(x=yran, y=xran, z=gcps[1], kx=k, ky=k)
253             # Compute latitude and longitude for all the pixels.
254             latlon = np.asarray([f(ystep, xstep) for f in [lat_spl, lon_spl]])
255             height = np.zeros((1,) + latlon.shape[1:])
256             self._hrpt_geo = np.vstack([latlon * DEG2RAD, height])
257 
258         # Compute latitudes and longitudes for all the HRPT pixels.
259         calc_hrpt_geo()
260         raw_ymin, raw_xmin = self.hrpt_geo.min(axis=1).min(axis=1)[:2] / DEG2RAD
261         raw_ymax, raw_xmax = self.hrpt_geo.max(axis=1).max(axis=1)[:2] / DEG2RAD
262 
263         # Show info message.
264         msg = "\n Calculating geotransform for gridded HRPT indices image"
265         print(msg)
266 
267         def calc_hrpt_gtr():
268             """Inner function where self._hrpt_gtr is computed."""
269 
270             # Compute geotransform parameters.
271             prj_st = np.asarray([HRPT_LATITUDE_STEP, HRPT_LONGITUDE_STEP])
272             prj_nw = np.asarray([raw_ymax, raw_xmin]) - 0.5 * prj_st
273             prj_se = np.asarray([raw_ymin, raw_xmax]) + 0.5 * prj_st
274             prj_sz = np.round((prj_se - prj_nw) / prj_st).astype(int)
275 
276             # Set geotransform property.
277             self._hrpt_gtr = (prj_nw[1], prj_st[1], 0, prj_nw[0], 0, prj_st[0])
278 
```

```
279             # Compute pixel indices in the new grid.
280             prj_st3 = prj_st[:, None, None]
281             prj_nw3 = prj_nw[:, None, None]
282             prj_rc = (self.hrpt_geo[:2]/DEG2RAD - prj_nw3) // prj_st3
283 
284             return prj_sz, prj_rc
285 
286         # Compute geotransform for the new grid.
287         prj_size, prj_rowcol = calc_hrpt_gtr()
288 
289         # Show info message.
290         msg = "\n Calculating gridded HRPT border mask"
291         print(msg)
292 
293         def calc_border_mask():
294             """Inner function where gridded border mask is obtained."""
295 
296             margin = HPRT_BORDER_MASK_MARGIN
297             iter_n = int(margin//2)
298             msk_sz = prj_size + 2 * margin
299             msk_rowcol = prj_rowcol + margin
300             row, col = msk_rowcol.reshape((2, np.prod(msk_rowcol.shape[1:])))
301             # Arrange test array.
302             ones = np.ones(row.shape)
303             test = coo_matrix((ones, (row, col)), shape=msk_sz).toarray()
304             test = test > 0
305             kern = np.ones((3, 3), dtype=bool)
306             # Dilate and erode test array to fill holes.
307             test = binary_dilation(test, structure=kern, iterations=iter_n)
308             test = binary_erosion(test, structure=kern, iterations=iter_n+5)
309             return np.logical_not(test[None, margin:-margin, margin:-margin])
310 
311         # Set pixels within border mask to fill value.
312         border_mask = calc_border_mask()
313 
314         # Show info message.
315         msg = "\n Calculating gridded HRPT indices"
316         print(msg)
317 
318         def calc_hrpt_pts():
319             """Inner function that returns xx, yy and vv values."""
320 
321             # Compute intermediate arrays.
322             sz = (-1, 2)
323             tp = np.float32
324             raw_rc = np.mgrid[:self.scan_ysize, :self.scan_xsize] + 0.5
325             qry_rc = np.mgrid[:prj_size[0], :prj_size[1]] + 0.5
326 
327             # Compute outputs.
328             rc = [prj_rowcol, raw_rc, qry_rc]
329             return [np.rollaxis(x, 0, 3).reshape(sz).astype(tp) for x in rc]
330 
331         # Set pixels from gridded image as xx and pixels from hrpt as yy. Set
332         # also pixels to be evaluated as vv.
333         xx, yy, vv = calc_hrpt_pts()
```

```
334
335         def calc_hrpt_pix_legacy1():
336             """Legacy inner function where self._hrpt_pix is computed."""
337
338             func = NearestNDInterpolator(xx, yy)
339             temp = func(vv).reshape((prj_size[0], prj_size[1], 2))
340             self._hrpt_pix = np.rollaxis(temp, 2)
341             self._hrpt_pix[border_mask] = fill_value
342
343         def calc_hrpt_pix_legacy2():
344             """Legacy inner function where self._hrpt_pix is computed."""
345
346             # Prepare info message for loop.
347             nfmt = len(str(prj_size[0]))
348             msg2 = "\n   rows {1:{0}d} - {2:{0}d} out of {3}"
349
350             # For every slice project the pixel indices.
351             temp = np.empty((2, prj_size[0], prj_size[1]), dtype=np.int16)
352
353             for lim1 in range(0, prj_size[0], slices):
354
355                 lim2 = min(prj_size[0], lim1 + slices)
356
357                 # Show info message.
358                 print(msg2.format(nfmt, lim1, lim2, prj_size[0]))
359
360                 # Get array slices for the current loop.
361                 flag = (xx[:, 0] > lim1-buffer) & (xx[:, 0] < lim2+buffer)
362                 xx_n = xx[flag]
363                 yy_n = yy[flag]
364                 vv_n = vv[lim1*prj_size[1]:lim2*prj_size[1]]
365
366                 # Convert to gridded data.
367                 size = (lim2-lim1, prj_size[1], 2)
368                 temp[:, lim1:lim2] = np.rollaxis(
369                     griddata(xx_n, yy_n, vv_n, method="nearest",
370                              fill_value=fill_value,).reshape(size), 2)
371
372             # Set array into self._hrpt_pix and mask border values.
373             self._hrpt_pix = temp
374             self._hrpt_pix[border_mask] = fill_value
375
376         def calc_hrpt_pix():
377             """Inner function where self._hrpt_pix is computed."""
378
379             # Arrange test array.
380             row, col = xx.T
381             ones_array = np.ones((yy.shape[0],), dtype=int)
382             test_array = np.asarray([
383                 coo_matrix((val, (row, col)), shape=prj_size).toarray()
384                 for val in [yy[:, 0], yy[:, 1], ones_array]])
385
386             # Divide array of indices by the array of weights.
387             nums = test_array[None, 2]
388             mass = np.where(nums, nums, 1)
```

```
389             data = (test_array[:2] // mass).astype(np.int16)
390
391             # Build necessary masks.
392             nodata_mask = np.equal(nums, 0)
393             values_mask = np.logical_not(border_mask)
394             distance_mask = np.repeat(nodata_mask, 2, axis=0) & values_mask
395
396             # Prepare info message for loop.
397             nfmt = len(str(prj_size[0]))
398             msg2 = "\n   rows {1:{0}d} - {2:{0}d} out of {3}"
399
400             n = data.shape[1]
401             temp = np.empty(data.shape, dtype=np.int16)
402
403             for lim1 in range(0, n, slices):
404
405                 lim2 = min(n, lim1 + slices)
406                 lim1_b = max(0, lim1 - buffer)
407                 lim2_b = min(n, lim2 + buffer)
408
409                 # Show info message.
410                 print(msg2.format(nfmt, lim1, lim2, data.shape[1]))
411
412                 # Store array of indices in a temporary array.
413                 indices = distance_transform_edt(
414                     distance_mask[:, lim1_b:lim2_b, :],
415                     return_distances=False,
416                     return_indices=True)
417                 temp_b = data[:, lim1_b:lim2_b, :][tuple(indices)]
418                 temp[:, lim1:lim2, :] = temp_b[:, lim1-lim1_b:lim2-lim1_b, :]
419
420             # Store temporary array of indices in self._hrpt_pix.
421             self._hrpt_pix = temp
422             self._hrpt_pix[:, border_mask[0]] = fill_value
423
424         # Calculate HRPT indices image.
425         calc_hrpt_pix()
426
427         # Show info message.
428         msg = "\n Calculating gridded HRPT bands"
429         print(msg)
430
431         def calc_hrpt_img():
432             """Inner function where self._hrpt_img is computed"""
433
434             # Extract HRPT bands from GDAL Dataset instance.
435             bnd_raw = np.vstack([
436                 self.dataset.GetRasterBand(i+1).ReadAsArray(
437                     0, 0, self.scan_xsize, self.scan_ysize)[None, :]
438                 for i in range(self.dataset.RasterCount)])
439
440             # Generate the array of absolute indices that connect the final
441             # geodetic dataset with the original raw dataset.
442             index = self.hrpt_pix.astype(int)
443             index = index[0] * bnd_raw.shape[2] + index[1]
```

```
444             bnd_raw = bnd_raw.reshape((bnd_raw.shape[0], -1))
445
446             # Assign digital-numbers using the indices array.
447             bnd_geo_shape = (bnd_raw.shape[0],) + index.shape
448             bnd_geo = bnd_raw.take(index.ravel(), axis=1, mode="warp")
449             bnd_geo = bnd_geo.reshape(bnd_geo_shape).astype(np.int16)
450             bnd_geo[:, border_mask[0]] = 0
451
452             self._hrpt_img = bnd_geo
453
454         # Calculate HRPT 5-band digital-number image.
455         calc_hrpt_img()
456
457         # Show info message.
458         msg = "\n Calculating gridded HRPT cloud masks"
459         print(msg)
460
461         def calc_hrpt_msk():
462             """Inner function where self._hrpt_msk is computed"""
463
464             # Set thermal evaluator (channel 5).
465             cld = self.hrpt_img[4] > HRPT_CLOUD_THRESHOLD_B5
466
467             self._hrpt_msk = 255 * cld[None, :].astype(np.uint8)
468
469         # Calculate HRPT cloud mask.
470         calc_hrpt_msk()
471
472         # Show info message.
473         msg = "\n HRPT channels were successfully gridded with shape {}"
474         print(msg.format(self.hrpt_img.shape))
475
476     def _compute_spacecraft_fixed_unit_look_vectors(self):
477         """Return unit look vectors in spacecraft-fixed coordinates."""
478
479         self._spacecraft_fixed_unit_look_vector = np.vstack([
480             np.cos(self.spacecraft_fixed_look_angle),
481             np.zeros((1, self.scan_xsize)),
482             np.sin(self.spacecraft_fixed_look_angle)])
483         return self._spacecraft_fixed_unit_look_vector
484
485     def _compute_spacecraft_geodetic_unit_vectors(self):
486         """Return unit vectors in spacecraft-geodetic coordinates."""
487
488         # Call necessary properties.
489         lat, lon, alt = [x[:, None] for x in self.orbit.position_geo.T]
490         vs = self.orbit.velocity_ecf
491         # Compute unit vectors (wx, wy, wz).
492         wx = np.hstack([
493             -np.cos(lat)*np.cos(lon), -np.cos(lat)*np.sin(lon), -np.sin(lat)])
494         wz = np.cross(vs, wx)
495         wz /= np.linalg.norm(wz, axis=1)[:, None]
496         wy = np.cross(wz, wx)
497         # Store all the unit vectors as 3x3 arrays with [wx, wy, wz].T
498         # structure concatenated along the third axis.
```

```
499         w_xyz = np.vstack([wx.T[None, :], wy.T[None, :], wz.T[None, :]])
500         self._spacecraft_geodetic_unit_vector = w_xyz
501         return w_xyz
502 
503     @classmethod
504     def _parse_datetime(cls, text_date):
505         """Return a datetime object from a valid datetime string."""
506 
507         match = re.match(DATE_PATTERN, text_date)
508         if match:
509             yr = int(match.group(1))
510             dy = int(match.group(2)) - 1
511             ms = int("".join([match.group(3), "000"]))
512             return datetime(yr, 1, 1) + timedelta(days=dy, microseconds=ms)
513         else:
514             msg = "Input text does not match datetime pattern"
515             raise ValueError(msg)
516 
517     @property
518     @returns(timedelta)
519     def clock_drift(self):
520         """on board clock variation"""
521         if self._clock_drift is None:
522             self._clock_drift = timedelta(seconds=0)
523         return self._clock_drift
524 
525     @property
526     @returns(gdal.Dataset)
527     def dataset(self):
528         """GDAL dataset containing the image"""
529         return self._dataset
530 
531     @property
532     @returns(datetime)
533     def end_datetime(self):
534         """datetime for last scanline"""
535 
536         if self._end_datetime is None:
537             try:
538                 text_date = self.dataset.GetMetadata()["STOP"]
539                 self._end_datetime = self._parse_datetime(text_date)
540             except KeyError:
541                 msg = "Scene dataset does not contain end datetime"
542                 err = SceneError(msg)
543                 err.__cause__ = None
544                 raise err
545         return self._end_datetime
546 
547     @property
548     @returns(np.ndarray)
549     def gcp_xrange(self):
550         """column indices where control points are defined"""
551         return self._gcp_xrange
552 
553     @property
```

```
554     @returns(np.ndarray)
555     def gcp_yrange(self):
556         """row indices where control points are defined"""
557         return self._gcp_yrange
558 
559     @property
560     @returns(np.ndarray)
561     def hrpt_geo(self):
562         """interpolated latitude and longitude values for all the pixels"""
563         return self._hrpt_geo
564 
565     @property
566     @returns(np.ndarray)
567     def hrpt_img(self):
568         """HRPT image in gridded window"""
569         return self._hrpt_img
570 
571     @property
572     @returns(np.ndarray)
573     def hrpt_msk(self):
574         """HRPT cloud mask in gridded window"""
575         return self._hrpt_msk
576 
577     @property
578     @returns(tuple)
579     def hrpt_gtr(self):
580         """geotransform for HRPT gridded window"""
581         return self._hrpt_gtr
582 
583     @property
584     @returns(np.ndarray)
585     def hrpt_pix(self):
586         """gridded image of HRPT indices"""
587         return self._hrpt_pix
588 
589     @property
590     @returns(Orbit)
591     def orbit(self):
592         """Orbit instance for the associated Earth satellite"""
593         return self._orbit
594 
595     @property
596     @returns(np.ndarray)
597     def gcp_position_ecf(self):
598         """dataset gcp position in ECF reference system"""
599         return self._gcp_position_ecf
600 
601     @property
602     @returns(np.ndarray)
603     def gcp_position_geo(self):
604         """dataset gcp position in geodetic reference system"""
605         return self._gcp_position_geo
606 
607     @property
608     @returns(np.ndarray)
```

```
609     def gcp_position_pix(self):
610         """dataset gcp position in pixel (row, col) reference system"""
611         return self._gcp_position_pix
612 
613     @property
614     @returns(timedelta)
615     def scan_timedelta(self):
616         """period of time which is necessary to measure a scanline"""
617 
618         if self._scan_timedelta is None:
619             sec = (self.end_datetime - self.start_datetime).total_seconds()
620             sec /= (self.scan_ysize - 1)
621             self._scan_timedelta = timedelta(seconds=sec)
622         return self._scan_timedelta
623 
624     @property
625     @returns(Angle)
626     def scan_step_look_angle(self):
627         """angle between across-track pixels within the image"""
628 
629         if self._scan_step_look_angle is None:
630             self._scan_step_look_angle = Angle(deg=DEFAULT_SCAN_STEP_LOOK_ANGLE)
631         return self._scan_step_look_angle
632 
633     @property
634     @returns(int)
635     def scan_xsize(self):
636         """number of columns in the Scene instance"""
637         if self._scan_xsize is None:
638             self._scan_xsize = self.dataset.RasterXSize
639         return self._scan_xsize
640 
641     @property
642     @returns(int)
643     def scan_ysize(self):
644         """number of rows in the Scene instance"""
645         if self._scan_ysize is None:
646             self._scan_ysize = self.dataset.RasterYSize
647         return self._scan_ysize
648 
649     @property
650     @returns(np.ndarray)
651     def spacecraft_attitude(self):
652         """spacecraft (roll, pitch, yaw) angles"""
653         if self._spacecraft_attitude is None:
654             self._spacecraft_attitude = np.zeros((3,))
655         return self._spacecraft_attitude
656 
657     @property
658     @returns(int)
659     def spacecraft_direction(self):
660         """spacecraft direction (+1 ascending, -1 descending)"""
661         if self._spacecraft_direction is None:
662             direction = self.dataset.GetMetadata()["LOCATION"].lower()
663             if direction == "ascending":
```

```
664                 self._spacecraft_direction = SPACECRAFT_ASCENDING
665             else:
666                 self._spacecraft_direction = SPACECRAFT_DESCENDING
667         return self._spacecraft_direction
668 
669     @property
670     @returns(np.ndarray)
671     def spacecraft_fixed_look_angle(self):
672         """look angles in spacecraft-fixed coordinate system"""
673 
674         if self._spacecraft_fixed_look_angle is None:
675             # Call necessary properties.
676             dstep = self.scan_step_look_angle.rad
677             xsize = self.scan_xsize
678             # Compute look angles.
679             dlook = -((xsize//2 - 0.5 - np.arange(xsize)[None, :]) * dstep)
680             self._spacecraft_fixed_look_angle = dlook
681         return self._spacecraft_fixed_look_angle
682 
683     @property
684     @returns(np.ndarray)
685     def spacecraft_fixed_unit_look_vector(self):
686         """unit look vectors in spacecraft-fixed coordinate system"""
687         return self._spacecraft_fixed_unit_look_vector
688 
689     @property
690     @returns(np.ndarray)
691     def spacecraft_geodetic_unit_vector(self):
692         """unit vectors in spacecraft-geodetic coordinate system"""
693         return self._spacecraft_geodetic_unit_vector
694 
695     @property
696     @returns(datetime)
697     def start_datetime(self):
698         """datetime for first scanline"""
699 
700         if self._start_datetime is None:
701             try:
702                 text_date = self.dataset.GetMetadata()["START"]
703                 self._start_datetime = self._parse_datetime(text_date)
704             except KeyError:
705                 msg = "Scene dataset does not contain start datetime"
706                 err = SceneError(msg)
707                 err.__cause__ = None
708                 raise err
709         return self._start_datetime
```

A.5. Clase Matcher

Código A.16: ../python-sat/sat/constants/Matcher.py

```
"""Necessary constants for sat.Matcher class."""

# Maximum absolute value in radians for any of the attitude angles.
MAX_ATTITUDE_ANGLE =\
    0.02

# Maximum geographic distance in meters to validate a matching.
MAX_DISTANCE =\
    15000

# Maximum descriptor distance to validate a matching.
MAX_HAMMING =\
    100
```

Código A.17: ../python-sat/sat/Matcher.py

```
from __future__ import division
from . constants.Angle import DEG2RAD
from . constants.Matcher import MAX_ATTITUDE_ANGLE
from . constants.Matcher import MAX_DISTANCE
from . constants.Matcher import MAX_HAMMING
from . constants.Scene import HRPT_CLOUD_THRESHOLD_B5
from . constants.Scene import SPACECRAFT_ASCENDING
from . decorators import returns
from . Coordinates import Coordinates
from . Error import MatcherError
from . Scene import Scene
from datetime import timedelta
from scipy.ndimage import binary_dilation
import cv2
import gdal
import os.path
import numpy as np


class Matcher(object):

    __slots__ = [
        "_bfmatcher",
        "_descriptor",
        "_detector",
        "_match_table",
        "_source",
        "_source_cloudmask",
        "_source_descriptors",
        "_source_gtr",
        "_source_keypoints",
        "_source_matchband",
```

```
33          "_target",
34          "_target_cloudmask",
35          "_target_descriptors",
36          "_target_gtr",
37          "_target_keypoints",
38          "_target_matchband",
39          ]
40
41      def __init__(self, source, fast_lim=40, brisk_lim=50):
42          """Constructor of a generic Matcher instance.
43
44          Parameters:
45
46          source
47              code for reference dataset
48
49          Optional parameters:
50
51          fast_lim
52              limit value introduced in the FAST constructor
53          brisk_lim
54              limit value introduced in the BRISK constructor
55          """
56
57          for item in self.__slots__:
58              self.__setattr__(item, None)
59
60          # Create detector, descriptor and Brute-Force matcher.
61          try:
62              self._detector = cv2.FastFeatureDetector(fast_lim)
63          except AttributeError:
64              self._detector = cv2.FastFeatureDetector_create(fast_lim)
65          try:
66              self._descriptor = cv2.BRISK(thresh=brisk_lim)
67          except AttributeError:
68              self._descriptor = cv2.BRISK_create(thresh=brisk_lim)
69          self._bfmatcher = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)
70
71          # Build source path.
72          dirname = os.path.dirname(__file__)
73          srcpath = os.path.join(dirname, "constants", "_matcher", source)
74
75          # Open the source and build its attributes.
76          gdal.PushErrorHandler("CPLQuietErrorHandler")
77          self._source = gdal.Open(srcpath)
78          self._build_source_geotransform()
79
80      def draw_matches(self, path, color=(0, 255, 255), radius=5, thickness=1):
81          """Export matches and the matches bands to an image file.
82
83          Parameters:
84
85          path
86              output path where the image will be stored
87
```

```
88          Optional parameters:
89
90          color
91              BGR tuple defining the color of circles and lines
92              (default (0, 255, 255) = yellow)
93          radius
94              circle radius (default 5)
95          thickness
96              circle and line thickness (default 1)
97          """
98
99          # Call necessary properties.
100         srcrows, srccols = self.source_matchband.shape
101         tgtrows, tgtcols = self.target_matchband.shape
102
103         # Create the output image with all its values set to 0.
104         shp = (max(srcrows, tgtrows), srccols+tgtcols, 3)
105         img = np.zeros(shp, dtype=np.uint8)
106
107         # Place the source and target match bands in the output image.
108         img[:srcrows, :srccols] = np.dstack(3*[self.source_matchband])
109         img[:tgtrows, srccols:] = np.dstack(3*[self.target_matchband])
110
111         # For each match, draw yellow circles over the matching pixels and a
112         # line between them.
113         for elem in self.match_table:
114             # Read points.
115             y1, x1 = elem[0:2].astype(int)
116             y2, x2 = elem[2:4].astype(int)
117             # Draw a circle representing every point.
118             cv2.circle(img, (x1, y1), radius, color, thickness)
119             cv2.circle(img, (x2+srccols, y2), radius, color, thickness)
120             # Draw a line between the two circles.
121             cv2.line(img, (x1, y1), (x2+srccols, y2), color, thickness)
122
123         # Save the image to a file.
124         cv2.imwrite(path, img)
125
126     def draw_source_keypoints(self, path):
127         """Export source keypoints and the match band to an image file.
128
129         Parameters:
130
131         path
132             output path where image will be stored
133         """
134         self.__class__._draw_keypoints(
135             path, self.source_matchband, self.source_keypoints)
136
137     def draw_target_keypoints(self, path):
138         """Export target keypoints and the match band to an image file.
139
140         Parameters:
141
142         path
```

```
143             output path where the image will be stored
144         """
145         self.__class__._draw_keypoints(
146             path, self.target_matchband, self.target_keypoints)
147 
148     def fix_attitude(self, max_dist=MAX_DISTANCE, max_hamm=MAX_HAMMING):
149         """Evaluate optimal values for target's attitude angles.
150 
151         Parameters:
152 
153         max_dist
154             maximum allowed geographic distance to validate a matching
155         max_hamm
156             maximum allowed descriptor distance to validate a matching
157         """
158 
159         # Verify that the target's attitude angles have not been evaluated yet.
160         if not np.allclose(self.target.spacecraft_attitude, 0):
161             msg = "Target spacecraft already has attitude angles defined"
162             raise MatcherError(msg)
163 
164         # Call finder.
165         new_r_pix = self._find_matches(max_dist, max_hamm)[0]
166 
167         # Calculate new look unit vector.
168         num = len(new_r_pix)
169         new_r_sat = np.empty(new_r_pix.shape)
170         new_d = np.empty(new_r_pix.shape)
171         new_ub = np.empty(new_r_pix.shape)
172         new_ug = np.empty(new_r_pix.shape)
173         for i in range(num):
174             # Get the row and column on the spacecraft original imagery.
175             row1, col1 = self.match_table[i, [2, 3]].astype(int)
176             row2, col2 = self.target.hrpt_pix[:, row1, col1]
177             # Get the new look vector in ECF coordinate system.
178             new_r_sat[i] = self.target.orbit.position_ecf[row2, :]
179             new_d[i] = new_r_pix[i] - new_r_sat[i]
180             # Transform look vector into spacecraft geodetic coordinate system.
181             rot_T = self.target.spacecraft_geodetic_unit_vector[:, :, row2].T
182             new_d[i] = np.dot(new_d[i], rot_T)
183             # Calculate new spacecraft-geodetic and spacecraft-fixed unit look
184             # vectors
185             new_ug[i] = new_d[i] / np.linalg.norm(new_d[i])
186             new_ub[i] = self.target.spacecraft_fixed_unit_look_vector[:, col2]
187 
188         # Calculate optimal attitude values which relates spacecraft-geodetic
189         # and spacecraft-fixed unit look vectors by LSM.
190         if num is not 1:
191             cnt = -1
192             flag = True
193             while flag:
194                 values = list(range(-(cnt < 0), cnt)) + list(range(cnt+1, num))
195                 ux, uy, uz = new_ug.T[:, values]
196                 cd, nd, sd = new_ub.T[:, values]
197                 # Create temporary values for LSM solution.
```

```
198                 g1 = + np.sum(sd * sd)
199                 g2 = - np.sum(sd * cd)
200                 g3 = + np.sum(cd * cd)
201                 h1 = - np.sum(uy * sd)
202                 h2 = + np.sum(ux * sd - uz * cd)
203                 h3 = + np.sum(uy * cd)
204                 # Calculate optimal roll, pitch and yaw.
205                 r_angle = h2 / len(cd)
206                 p_angle = (g1*h3 - g2*h1) / (g1*g3 - g2*g2)
207                 y_angle = (g3*h1 - g2*h3) / (g1*g3 - g2*g2)
208                 attitude = np.asarray([r_angle, p_angle, y_angle])
209                 # Check the values are within the valid range.
210                 flag = np.any(np.abs(attitude) > MAX_ATTITUDE_ANGLE)
211                 cnt += 1
212                 # Alternative if the common LSM method does not give a solution.
213                 if cnt == num + 1:
214                     break
215         if num is 1:
216             ux, uy, uz = new_ug.T
217             cd, nd, sd = new_ub.T
218             # Estimate roll, pitch and yaw minimizing pitch and yaw.
219             r_angle = + ux * sd - uz * cd
220             p_angle = + uy * cd
221             y_angle = - uy * sd
222             # Choose the median values.
223             attitude = np.asarray([r_angle, p_angle, y_angle]).reshape((-1, 3))
224             attitude = np.median(attitude, axis=0).flatten()
225             flag = np.any(np.abs(attitude) > MAX_ATTITUDE_ANGLE)
226             if flag:
227                 msg = "attitude angles outside range"
228                 raise MatcherError(msg)
229
230         self.target._spacecraft_attitude = attitude
231         return attitude
232
233     def fix_clock_drift(self, max_dist=MAX_DISTANCE, max_hamm=MAX_HAMMING):
234         """Evaluate optimal value for the target's clock drift's optimal.
235
236         Parameters:
237
238         max_dist
239             maximum allowed geographic distance to validate a matching
240         max_hamm
241             maximum allowed descriptor distance to validate a matching
242         """
243
244         # Call finder.
245         dr_xyz_alongtrack = self._find_matches(max_dist, max_hamm)[1]
246
247         # Calculate velocity vectors for every ECF distance vector.
248         num = len(dr_xyz_alongtrack)
249         v_sat_alongtrack = np.empty(dr_xyz_alongtrack.shape)
250         for i in range(num):
251             # Get the row and column on the spacecraft original imagery.
252             row1, col1 = self.match_table[i, [2, 3]].astype(int)
```

```
253             row2, col2 = self.target.hrpt_pix[:, row1, col1]
254             # Get the velocity vector.
255             v_sat_i = self.target.orbit.velocity_ecf[row2, :]
256             old_wy_i = self.target.spacecraft_geodetic_unit_vector[1, :, row2]
257             v_sat_alongtrack[i] = np.sum(v_sat_i * old_wy_i)
258
259         # The list of vectors dr_xyz is related to the list of velocities v_sat
260         # so that v_sat * dt = dr_xyz, where dt is the elapsed time. Solve it
261         # by using LSM method: A*X = B.
262         aa = v_sat_alongtrack[:, None]
263         bb = dr_xyz_alongtrack[:, None]
264         xx = np.linalg.lstsq(aa, bb)[0]
265         rs = np.abs(np.dot(aa, xx) - bb)
266
267         # Calculate clock drift as a timedelta instance.
268         dtime = xx[0, 0]
269         drift = timedelta(seconds=dtime)
270
271         # Update target clock drift and return drift value.
272         self.target._clock_drift += drift
273         return drift
274
275     def set_target(self, target):
276         """Add a Scene instance to be matched and compute its attributes.
277
278         Parameters:
279
280         target
281             a Scene instance that needs matching correction
282         """
283
284         # Open the target and build its attributes.
285         self._target = target
286         self._build_target_cloudmask()
287         self._build_target_geotransform()
288         self._build_source_cloudmask()
289
290         # Build the match bands for both the source and the target.
291         self._build_matchbands()
292         self._build_keypoints()
293         self._build_descriptors()
294
295     def _build_source_cloudmask(self):
296         """Create cloud mask based on channel 5 brightness temperature."""
297
298         # Call necessary properties.
299         k = np.ones((3, 3))
300         xsize = self.source.RasterXSize
301         ysize = self.source.RasterYSize
302         b5min = HRPT_CLOUD_THRESHOLD_B5
303
304         # Create cloud mask for source dataset and dilate.
305         src_cld = self.source.GetRasterBand(5).ReadAsArray(
306             0, 0, xsize, ysize).astype(np.int16)
307         src_cld = binary_dilation((src_cld > b5min), k, iterations=20)
```

```
308
309         self._source_cloudmask = 255 * src_cld.astype(np.uint8)[None, :]
310
311     def _build_target_cloudmask(self):
312         """Create cloud mask based on channel 5 brightness temperature."""
313
314         # Call necessary properties.
315         k = np.ones((3, 3))
316
317         # Dilate the already existing cloud mask from target dataset.
318         tgt_cld = self.target.hrpt_msk[0]
319         tgt_cld = binary_dilation(tgt_cld, k, iterations=20)
320
321         self._target_cloudmask = 255 * tgt_cld.astype(np.uint8)[None, :]
322
323     def _build_source_geotransform(self):
324         """Create geotransform array for source dataset."""
325
326         srcgtr = np.asarray(self.source.GetGeoTransform()).reshape((2, 3))
327         self._source_gtr = srcgtr
328
329     def _build_target_geotransform(self):
330         """Create geotransform array for target dataset."""
331         self._target_gtr = np.asarray(self.target.hrpt_gtr).reshape((2, 3))
332
333     def _build_matchbands(self):
334         """Create match bands based on channels 2/3 for day/night."""
335
336         # Get index of the dataset band used in matching (it starts in 1).
337         idx = 2 + int(self.target.spacecraft_direction is SPACECRAFT_ASCENDING)
338         xsize = self.source.RasterXSize
339         ysize = self.source.RasterYSize
340
341         def enclose(band, qmin, qmax):
342             cmin = np.percentile(band[band > 0], qmin)
343             cmax = np.percentile(band[band > 0], qmax)
344             return np.maximum(0, np.minimum(255, 255/(cmax-cmin) * (band-cmin)))
345
346         # Create match band for source dataset.
347         srcband = self.source.GetRasterBand(idx).ReadAsArray(
348             0, 0, xsize, ysize).astype(np.int16)
349         srcband[srcband < 0] = 0
350         srcband = enclose(srcband, 1, 99)
351         srcband = np.dstack(3 * [srcband]).astype(np.uint8)
352         srcband = cv2.cvtColor(srcband, cv2.COLOR_BGR2GRAY)
353         self._source_matchband = srcband
354
355         # Create match band for target dataset.
356         tgtband = self.target.hrpt_img[idx-1]
357         tgtband[tgtband < 0] = 0
358         tgtband = enclose(tgtband, 1, 99)
359         tgtband = np.dstack(3 * [tgtband]).astype(np.uint8)
360         tgtband = cv2.cvtColor(tgtband, cv2.COLOR_BGR2GRAY)
361         self._target_matchband = tgtband
362
```

```
363     def _build_keypoints(self):
364         """Create lists of keypoints using the FAST detector."""
365 
366         k = np.ones((3, 3))
367         srcclds = self.source_cloudmask[0]
368         tgtclds = self.target_cloudmask[0]
369         srcmask = binary_dilation((self.source_matchband == 0), k, iterations=5)
370         tgtmask = binary_dilation((self.target_matchband == 0), k, iterations=5)
371 
372         # Create keypoints for source dataset.
373         srckps = self.detector.detect(self.source_matchband, None)
374         self._source_keypoints = np.asarray([
375             x for x in srckps if
376             not srcclds[x.pt[1], x.pt[0]] and not srcmask[x.pt[1], x.pt[0]]])
377 
378         # Create keypoints for target dataset.
379         tgtkps = self.detector.detect(self.target_matchband, None)
380         self._target_keypoints = np.asarray([
381             x for x in tgtkps if
382             not tgtclds[x.pt[1], x.pt[0]] and not tgtmask[x.pt[1], x.pt[0]]])
383 
384     def _build_descriptors(self):
385         """Create lists of descriptors using the BRISK algorithm."""
386 
387         # Create source descriptors.
388         srckps, srcdes = self.descriptor.compute(
389             self.source_matchband, self.source_keypoints)
390         self._source_keypoints = np.asarray(srckps)
391         self._source_descriptors = srcdes
392 
393         # Create target descriptors.
394         tgtkps, tgtdes = self.descriptor.compute(
395             self.target_matchband, self.target_keypoints)
396         self._target_keypoints = np.asarray(tgtkps)
397         self._target_descriptors = tgtdes
398 
399     @staticmethod
400     def _draw_keypoints(path, img, kps):
401         """Export keypoints over the dataset to an image file."""
402 
403         fmt = cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS
404         try:
405             img2 = cv2.drawKeypoints(img, kps, flags=fmt)
406         except TypeError:
407             img2 = np.empty(img.shape, dtype=np.uint8)
408             img2 = cv2.drawKeypoints(img, kps, img2, flags=fmt)
409         cv2.imwrite(path, img2)
410 
411     def _find_matches(self, max_dist, max_hamm):
412         """Find feature matches between the source and the target."""
413 
414         # Call necessary properties.
415         srckp1 = self.source_keypoints
416         tgtkp1 = self.target_keypoints
417         srcdes = self.source_descriptors
```

```
418         tgtdes = self.target_descriptors
419         srcgtr = self.source_gtr
420         tgtgtr = self.target_gtr
421 
422         # Call Brute-Force matcher.
423         m = self.bfmatcher.match(srcdes, tgtdes)
424         num = len(m)
425         if num is 0:
426             msg = "No matches source-target were found"
427             raise MatcherError(msg)
428         m = sorted(m, key=lambda x: x.distance)
429         match_difference = np.asarray([x.distance for x in m])
430         srckp2 = np.asarray([[y+0.5 for y in srckp1[x.queryIdx].pt] for x in m])
431         tgtkp2 = np.asarray([[y+0.5 for y in tgtkp1[x.trainIdx].pt] for x in m])
432 
433         # Convert pixel coordinates to geodetic and ECF coordinates.
434         alt = np.zeros((len(m), 1))
435         new_lonlat = srcgtr[:, 0][None, :] + np.dot(srckp2, srcgtr[:, 1:].T)
436         old_lonlat = tgtgtr[:, 0][None, :] + np.dot(tgtkp2, tgtgtr[:, 1:].T)
437         new_xyz = Coordinates.from_geo_to_ecf(
438             np.hstack([new_lonlat[:, [1, 0]], alt]) * DEG2RAD)
439         old_xyz = Coordinates.from_geo_to_ecf(
440             np.hstack([old_lonlat[:, [1, 0]], alt]) * DEG2RAD)
441         distance_xyz = old_xyz - new_xyz
442 
443         # Compute geographic and temporal distances between matches.
444         distance_xyz_alongtrack = np.empty((num,))
445         time_difference = np.empty((num,))
446         for i in range(num):
447             # Get the row and column on the spacecraft original imagery.
448             row1, col1 = tgtkp2[i, [1, 0]].astype(int)
449             row2, col2 = self.target.hrpt_pix[:, row1, col1]
450             # Compute the along-track geographic distance.
451             old_wy_i = self.target.spacecraft_geodetic_unit_vector[1, :, row2]
452             distance_xyz_alongtrack[i] = np.sum(distance_xyz[i] * old_wy_i)
453             # Get the along-track velocity.
454             v_sat_i = self.target.orbit.velocity_ecf[row2, :]
455             old_wy_i = self.target.spacecraft_geodetic_unit_vector[1, :, row2]
456             v_sat_alongtrack_i = np.sum(v_sat_i * old_wy_i)
457             # Compute the time difference between matches positions.
458             # time_difference_i = distance_xyz_alongtrack_i / v_sat_alongtrack_i
459             time_difference[i] = distance_xyz_alongtrack[i] / v_sat_alongtrack_i
460 
461         # Create match table with information about feature matches.
462         match_table = np.hstack([
463             srckp2[:, [1, 0]],
464             tgtkp2[:, [1, 0]],
465             new_lonlat[:, [1, 0]],
466             old_lonlat[:, [1, 0]],
467             match_difference[:, None],
468             np.linalg.norm(distance_xyz, axis=1)[:, None],
469             distance_xyz_alongtrack[:, None],
470             time_difference[:, None],
471             ])
472         # Validate matches by using their geographic distances.
```

```
473         mask1 = np.abs(match_table[:, 9]) < max_dist
474         match_table = match_table[mask1]
475         if len(match_table) is 0:
476             msg = "No matches source-target after distance restraint"
477             raise MatcherError(msg)
478         # Validate matches by using their descriptor distances.
479         mask2 = match_table[:, 8] < max_hamm
480         match_table = match_table[mask2]
481         if len(match_table) is 0:
482             msg = "No matches source-target after distance+descriptor restraint"
483             raise MatcherError(msg)
484 
485         self._match_table = match_table
486 
487         return new_xyz[mask1][mask2], distance_xyz_alongtrack[mask1][mask2]
488 
489     @property
490     def bfmatcher(self):
491         """Brute-Force matcher"""
492         return self._bfmatcher
493 
494     @property
495     def descriptor(self):
496         """BRISK descriptor"""
497         return self._descriptor
498 
499     @property
500     def detector(self):
501         """FAST detector"""
502         return self._detector
503 
504     @property
505     @returns(np.ndarray)
506     def match_table(self):
507         """table containing all the relevant matches attributes"""
508         return self._match_table
509 
510     @property
511     @returns(gdal.Dataset)
512     def source(self):
513         """source dataset used as reference"""
514         return self._source
515 
516     @property
517     @returns(np.ndarray)
518     def source_cloudmask(self):
519         """cloud mask for source dataset"""
520         return self._source_cloudmask
521 
522     @property
523     @returns(np.ndarray)
524     def source_descriptors(self):
525         """descriptors from source dataset keypoints"""
526         return self._source_descriptors
527 
```

```
528     @property
529     @returns(np.ndarray)
530     def source_gtr(self):
531         """tuple geotransform for source dataset"""
532         return self._source_gtr
533 
534     @property
535     @returns(np.ndarray)
536     def source_keypoints(self):
537         """keypoints from source dataset"""
538         return self._source_keypoints
539 
540     @property
541     @returns(np.ndarray)
542     def source_matchband(self):
543         """source dataset's band used in matching process"""
544         return self._source_matchband
545 
546     @property
547     @returns(Scene)
548     def target(self):
549         """target dataset to be geometrically corrected"""
550         return self._target
551 
552     @property
553     @returns(np.ndarray)
554     def target_cloudmask(self):
555         """cloud mask for target dataset"""
556         return self._target_cloudmask
557 
558     @property
559     @returns(np.ndarray)
560     def target_descriptors(self):
561         """descriptors from target dataset keypoints"""
562         return self._target_descriptors
563 
564     @property
565     @returns(np.ndarray)
566     def target_gtr(self):
567         """tuple geotransform for target dataset"""
568         return self._target_gtr
569 
570     @property
571     @returns(np.ndarray)
572     def target_keypoints(self):
573         """keypoints from target dataset"""
574         return self._target_keypoints
575 
576     @property
577     @returns(np.ndarray)
578     def target_matchband(self):
579         """target dataset's band used in matching process"""
580         return self._target_matchband
```

Bibliografía

[1] Natural Environment Research Council (NERC) Earth Observation Data Acquisition and Analysis Service (NEODAAS) (2015) *HRPT format*, disponible en: http://www.sat.dundee.ac.uk/hrptformat.html.

[2] EUMETSAT (2011) *AVHRR level 1b product guide.*

[3] D. Brouwer (1959) *Solution of the problem of artificial satellite theory without drag*, The astronomical journal, 1959, 64, no. 1274, pp. 378−396.

[4] P. Brunel, A. Marsouin (1987) *An operational method using Argos orbital elements for navigation of AVHRR imagery*, International journal of remote sensing, 1987, vol. 8, pp. 569−578.

[5] P. Illera, J. A. Delgado, A. Calle (1996) *A navigation algorithm for satellite images*, International journal of remote sensing, 1996, vol. 17, no. 3, pp. 577−588.

[6] G. W. Rosborough, D. G. Baldwin, W. J. Emery (1994) *Precise AVHRR image navigation*, IEEE Transactions on geoscience and remote sensing, mayo 1994, vol. 32, no. 3, pp. 644−657.

[7] A. Calle, J. L. Casanova, J. Sanz (2000) *Algoritmo automático de corrección geográfica para imágenes NOAA-AVHRR: desarrollo y análisis de errores*, Revista de teledetección, diciembre 2010, no. 14.

[8] D. A. Vallado, P. Crawford, R. Hujsak, T. S. Kelso (2006) *Revisiting spacetrack report #3*, American Institute of Aeronautics and Astronautics, 2006−6753,

disponible en:
https://celestrak.com/publications/AIAA/2006-6753/AIAA-2006-6753.pdf.

[9] F. R. Hoots, R. L. Roehrich (1980) *Spacetrack report no. 3: Models for propagation of NORAD element sets*, disponible en:
https://celestrak.com/NORAD/documentation/spacetrk.pdf.

[10] B. C. Rhodes (2015) *PyEphem documentation*, disponible en:
http://rhodesmill.org/pyephem/#documentation.

[11] E. C. Downey (2015) *XEphem reference manual*, disponible en:
http://www.clearskyinstitute.com/xephem/help/xephem.html#mozTocId74433.

[12] G. Kruger (2012) *Utilisation de EPS MetOp Manager*, disponible en:
http://www.satsignal.eu/software/metop_manager_French.pdf.

[13] Dartcom (2014) *Dartcom iDAP/MacroPro software user guide*, disponible en:
ftp://ftp.dartcom.co.uk/pub/manuals/iDAPMacroProSoftwareUserGuide.zip

[14] T. Labrot, N. Atkinson, P. Roquet (2015) *AAPP documentation*, disponible en:
https://nwpsaf.eu/deliverables/aapp/NWPSAF-MF-UD-002_Software.pdf.

[15] National Imagery and Mapping Agency (2000) *Department of Defense World Geodetic System 1984*, tercera edición, disponible en:
http://earth-info.nga.mil/GandG/publications/tr8350.2/wgs84fin.pdf.

[16] A. B. Gonzalo (2015) *Posicionamiento y navegación*, Tema 1, Máster en Geotecnologías Cartográficas en Ingeniería y Arquitectura, Universidad de Salamanca.

[17] P. A. Tipler, G. Mosca (2004) *Física para la ciencia y la tecnología*, vol. 1, cap. 11.

[18] T. S. Kelso (1998) *Frequently asked questions: two-line element set format*, Satellite Times, 4, no. 3, enero 1998, disponible en:
https://celestrak.com/columns/v04n03/.

[19] P. E. El'Yasberg (1967) *Introduction to the theory of flight of artificial satellites*, Israel Program for Scientific Translations.

[20] R. F. Alitappeh, F. Mahmoudi (2010) *MGS-SIFT: A new illumination feature based on SIFT*, 3rd International conference on machine vision (ICMV 2010), 28 a 30 de diciembre de 2010, Hong-Kong.

[21] S. Leutenegger, M. Chli, Y. Siegwart (2011) *BRISK: Binary robust invariant scalable keypoints*, In Computer Vision (ICCV), 2011 IEEE International Conference on, pp. 2548−2555.

[22] J. M. Morel, G. Yu (2009) *ASIFT: A new framework for fully affine invariant image comparison*, SIAM Journal on imaging sciences, vol. 2 no. 2, pp. 438–469.

[23] E. Rosten, T. Drummond (2006) *Machine learning for high speed corner detection*, 9th European conference on computer vision, 2006, vol. 1, pp. 430−443.

[24] M. Calonder, V. Lepetit, C. Strecha, P. Fua (2010) *Binary robust independent elementary features*, Proceedings of the European conference on computer vision.

[25] NumPy Community (2014) *NumPy reference*, disponible en: http://docs.scipy.org/doc/numpy-dev/numpy-ref.pdf.

[26] SciPy Community (2015) *SciPy reference guide*, disponible en: http://docs.scipy.org/doc/scipy-dev/scipy-ref.pdf.

[27] Free Software Foundation (2007) *GNU General Public License, Version 3*, documento completo de la licencia disponible en: http://www.gnu.org/copyleft/gpl.html.

[28] R. W. Sinnott (1984) *Virtues of the haversine*, Sky and telescope, 1984, vol. 68, no. 2, p. 159.