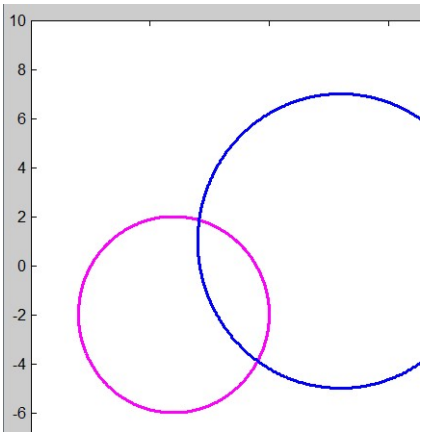
# SISTEMAS de ECUACIONES no LINEALES

## 1. Método de Newton y método secante multidimensional

En clase nos hemos limitado al caso de **ecuaciones no lineales 1D** con una sola incógnita. En la práctica veremos el caso de sistemas de ecuaciones no lineales (varias incógnitas en varias ecuaciones no lineales).




Consideremos el sistema de dos ecuaciones con dos incógnitas:

$$\begin{cases} f_1(x,y) &= \sqrt{(x+4)^2+(y+2)^2}-4 &= 0 \\ f_2(x,y) &= \sqrt{(x-3)^2+(y-1)^2}-6 &= 0 \end{cases}$$

cuya representación grafica son dos circunferencias de radios 4 y 6. Ejecutar **`>>demo2d`** para visualizar esta gráfica. Las soluciones son los puntos de corte (dos en este caso) de las dos circunferencias donde las dos ecuaciones se cumplen de forma simultánea.

Para resolver estos problemas podemos usar el método de Newton (o alguna de sus variantes), que es generalizable de forma inmediata a N dimensiones. En 1D el método de Newton era:

$$x_{n+1} = x_n + \Delta x \quad \text{siendo} \quad \Delta x = -\frac{f(x_n)}{f'(x_n)}$$

En el caso multidimensional, la incógnita x es ahora un vector $\underline{X}$ conteniendo las dos incógnitas (x,y). La función a resolver f(x): R→R se convierte en una función $\underline{F}(\underline{X})$: $R^2 \rightarrow R^2$, que recibe un vector $\underline{X}$ = (x,y) y devuelve otro vector $\underline{F}$ (con los 2 valores de $f_1(x,y)$ y $f_2(x,y)$).

Lo primero, como en otros ejercicios de este tema, es escribir la función MATLAB que codifique la función cuyas soluciones hay que encontrar. La única diferencia con los ejemplos de clase es que ahora la función recibe un vector X=(x,y) y devuelve también un vector F (con las evaluaciones de $f_1$ y $f_2$).

Completad la función fun.m suministrada para que codifique el sistema anterior. La función recibe X, un vector **columna** (2 x 1) cuyas componentes son (x,y) y debe devolver un vector **columna** F cuyas componentes serán las evaluaciones de las dos ecuaciones (f1 y f2) para los valores de (x, y) dados en X.

Una vez completada probadla con: **`>> X=[1 -1]'; F=fun(X)`**. El resultado F debe ser un **vector columna** con valores [1.0990 -3.1716]. Adjuntad el resultado de fun() para X=[-1 1]'.

Una vez que estamos trabajando con una función $\underline{F}(\underline{x})$ vectorial, el método de Newton se convierte en:

$$\bar{x}_{n+1} = \bar{x}_n + \Delta\bar{x}\,.$$

En el caso 1D el incremento a aplicar en cada paso era -f(x)/f'(x). Ahora $\Delta\underline{x}$ tiene un aspecto muy similar. La diferencia es que ahora el valor de la función $\underline{F}(\underline{x})$ es un vector y la "derivada" una matriz 2x2, ya que hay que considerar la derivada de las dos componentes de la función $\underline{F}$ (f1 y f2) con respecto a las dos incógnitas (x e y):

$$\Delta\bar{x} = -J^{-1}(\bar{x})\cdot\bar{F}(\bar{x}) \text{ con } J(\bar{x}) = \begin{pmatrix} \frac{\partial f_1}{\partial x} & \frac{\partial f_1}{\partial y} \\ \frac{\partial f_2}{\partial x} & \frac{\partial f_2}{\partial y} \end{pmatrix} = \begin{pmatrix} \frac{(x+4)}{\sqrt{(x+4)^2+(y+2)^2}} & \frac{(y+2)}{\sqrt{(x+4)^2+(y+2)^2}} \\ \frac{(x-3)}{\sqrt{(x-3)^2+(y-1)^2}} & \frac{(y-1)}{\sqrt{(x-3)^2+(y-1)^2}} \end{pmatrix}$$

En la expresión para $\Delta$x cambiamos la inversa de la derivada (1D) por la inversa de la matriz J que incluye todas las derivadas. La única diferencia es que el cálculo es ahora más complicado: en lugar de tener que calcular una derivada, hay que calcular cuatro (en este caso con 2 ecuaciones y dos incógnitas). Luego, en vez de simplemente dividir por la derivada, hay que invertir la matriz J.

En el ejemplo anterior para el punto X=[1 -1]' substituyendo x=1 e y=-1 en la expresión de la matriz J obtendríamos los siguientes valores: **0.9806 0.1961**
**-0.7071 -0.7071**

Vamos a completar la función fun() para que tras evaluar F(X) calcule también la matriz J(X) y las devuelva en dos argumentos de salida (F y J). Podríamos usar las expresiones anteriores para J, pero cada vez que cambiáramos la función F tendríamos que recalcular las nuevas derivadas, lo que puede ser complicado (o imposible si desconocemos la expresión para F(x)). Además, la complejidad crece rápidamente en sistemas con más ecuaciones e incógnitas, al aumentar el número de derivadas a calcular. Por esta razón, en sistemas de ecuaciones es especialmente interesante usar métodos del tipo "secante", donde la matriz J de derivadas se calcula de forma aproximada (en vez de hallar todas las derivadas de forma analítica). Vamos a completar la función fun() para que calcule una aproximación de J sin tener que hallar derivadas.

Podemos ver que cada columna de la matriz J es la derivada de $\underline{F}$ con respecto a las sucesivas variables x, y:

$$J = \begin{pmatrix} \frac{\partial f_1}{\partial x} & \frac{\partial f_1}{\partial y} \\ \frac{\partial f_2}{\partial x} & \frac{\partial f_2}{\partial y} \end{pmatrix} = \begin{pmatrix} \frac{\partial}{\partial x}\begin{pmatrix} f_1 \\ f_2 \end{pmatrix} & \frac{\partial}{\partial y}\begin{pmatrix} f_1 \\ f_2 \end{pmatrix} \end{pmatrix} = \begin{pmatrix} \frac{\partial \bar{F}}{\partial x} & \frac{\partial \bar{F}}{\partial y} \end{pmatrix} = \left(\bar{F}_x, \bar{F}_y\right)$$

Para aproximar $F_x(\underline{X})$ podemos evaluar F en $\underline{X}$ y en un punto cercano X+[h;0]:

$$\bar{F}_X \approx \frac{\bar{F}\left(\bar{X}+\bar{h}_X\right)-\bar{F}\left(\bar{X}\right)}{h} \quad con \quad \bar{h}_X = \begin{pmatrix} h \\ 0 \end{pmatrix}$$ (usando un valor pequeño de h=0.1)

La fórmula aproxima $\underline{F_x}$ (la derivada de $\underline{F}$ con respecto a x) viendo cómo cambia el valor de $\underline{F}$ al movernos un poco (h) en la dirección del eje X ($h_x$).

Si repetimos moviéndonos ahora un poco (h) en la dirección y, aproximaremos el valor de la derivada Fy (2ª columna de J):

$$\overline{F_Y} \approx \frac{\overline{F}\left(\overline{X}+\overline{h}_Y\right)-\overline{F}\left(\overline{X}\right)}{h} \quad con \quad \overline{h}_Y = \begin{pmatrix} 0 \\ h \end{pmatrix}$$

Vemos que basta evaluar la función en (X+hx) y (X+hy) (usando h=0.1), para lo que basta hacer fun(X+$h_x$) y fun(X+$h_y$). Luego restamos a dichos resultados el valor F correspondiente a fun(X) y dividimos por h para estimar las derivadas Fx y Fy. Juntando Fx y Fy (vectores 2x1) se construye la matriz J como:

J = [Fx Fy],

obteniendo así nuestra aproximación a la matriz J (2 x 2) del sistema. Añadid vuestro código a fun.m después de la línea: `if nargout==1, return; end`

De esta forma si se llama a la función con solo un argumento de salida F=fun(X) solo calculará la función. Si la llamamos con 2 argumentos [F, J]=fun(X), el valor de **nargout** será 2 y se ejecutará la 2ª parte del código, calculándose también J.

Probad la función haciendo **`[F,J]=fun([1;-1])`**. F debe salir lo mismo que antes, y en J obtendréis una aproximación de la matriz J anterior:

$$J = \begin{pmatrix} 0.9810 & 0.2055 \\ -0.6980 & -0.6980 \end{pmatrix} \approx \begin{pmatrix} 0.9806 & 0.1961 \\ -0.7071 & -0.7071 \end{pmatrix}$$

Ahora, si cambian las ecuaciones solo hay que modificar la primera parte de la función (el cálculo de F), ya que J se aproximará usando las nuevas funciones.
Adjuntad código de fun() y dar sus resultados (F y J) para el punto X=[-1; 2].

---

En todos los métodos iterativos necesitamos un punto inicial X0=(x0,y0) para arrancar la iteración. Al ejecutar `>>demo2d 1` veréis que podéis pinchar con el ratón, lo que marca el punto inicial X0 del método iterativo. Sin embargo, ahora la demo no hace nada porque por debajo llama al **script** iteracion.m que no está completo.

Editar iteracion.m para completarlo. El único código que hay ahora es X=X0, que inicializa X con el punto inicial X0. A partir de esa X debéis implementar un bucle que actualice su valor siguiendo el método que acabamos de describir:

1) Inicializad dX=1; y un contador de iteraciones n=0;

2) Mientras que norm(dX)>0.001 AND n<200 repetir los pasos siguientes:

- Usad fun(X) para obtener la función F y el jacobiano J en X.
- Calcular dX, resolviendo en MATLAB el sistema lineal J·dX=-F
- Actualizar X sumándole el incremento dX e incrementar el contador n.
- Llamar a show_results(X); Esta función (ya escrita) recibe el resultado X al final de cada paso y muestra gráficamente su evolución en la aplicación.

Observad que iteramos hasta que la norma del incremento dX (diferencia entre pasos sucesivos) sea lo suficientemente pequeña, ya que, como sabemos, dicha diferencia está asociada al error. También hemos puesto un límite de 200 iteraciones para no quedar atrapados si no hubiera convergencia. Una vez completado el script la demo debería funcionar.

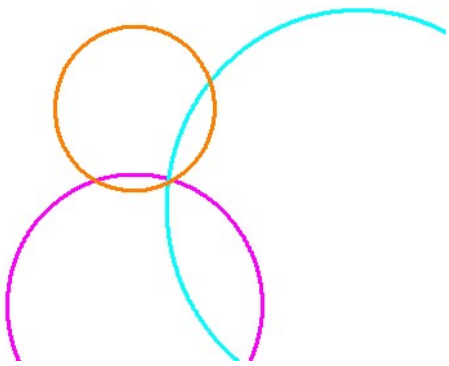
Volved a ejecutar `>>demo2d`. Tras pinchar en un punto para elegir el X0 inicial el código que habéis escrito calculará las sucesivas iteraciones de X y la llamada a **`show_results(X)`** mostrará de forma gráfica su evolución. Deberíais ver como la iteración termina en alguna de las dos soluciones. Si no es así o la iteración no converge, revisad vuestro código. Adjuntad código del script iteracion.m

Corred la demo y marcad ~10 puntos alrededor de cada solución para observar la convergencia. En este caso la iteración converge salvo que X0 esté cerca de la línea que une los centros de ambas circunferencias. Probad a encontrar un punto sobre esa línea y veréis que la iteración no termina o Matlab da un error. El problema es que en esos casos podemos pasar por un punto X tal que la matriz J(X) sea singular (el equivalente a tener f'(x)=0 en 1D). En ese caso obtenemos Inf o NaN. Adjuntad captura de la figura con vuestras pruebas.

## 1.2 Resolución de Sistemas No Lineales Sobredeterminados

En este apartado vamos a generalizar el método para trabajar con sistemas con más ecuaciones que incógnitas. Ahora la función fun() recibirá un vector X de N componentes (incógnitas) y devolverá un vector F de M (M>N) componentes (las evaluaciones de las M funciones) y una matriz J de tamaño MxN.

En ajuste de datos vimos que el problema de ajustar una curva a una tabla de puntos se reducía a plantear un sistema de ecuaciones sobredeterminado, donde tenemos demasiadas condiciones (ecuaciones) para las incógnitas disponibles. En este caso no es posible verificar las ecuaciones simultáneamente y no habrá una solución. Lo que hacíamos era resolverlas "aproximadamente" usando el criterio de mínimos cuadrados. De forma análoga podemos tener sistemas no lineales sin solución, donde no existe ningún punto que verifique todas las ecuaciones y se trata de encontrar una aproximación. Consideremos un sistema con las mismas 2 incógnitas (x,y) pero ahora con 3 ecuaciones:



$$\begin{cases} f_1(x,y) = \sqrt{(x+4)^2+(y+2)^2} - 4 = 0 \\ f_2(x,y) = \sqrt{(x-3)^2+(y-1)^2} - 6 = 0 \\ f_3(x,y) = \sqrt{(x+4)^2+(y-4)^2} - 2.5 = 0 \end{cases}$$

El sistema es el mismo que el del primer apartado, y se le ha añadido una 3ª circunferencia adicional. Con `>>demo2d 2` podéis ver la gráfica adjunta de las tres circunferencias involucradas.

Si hacéis zoom en el punto donde parece estar la solución, veréis que en realidad no hay ningún punto del plano donde coincidan las tres curvas, por lo que el sistema no tiene solución. Sin embargo, al igual que vimos en el Tema de ajuste, podemos encontrar una solución de compromiso: un punto donde las ecuaciones "casi" valgan cero. Esto es, un punto que caiga lo más "cerca" posible de las tres curvas. Si aceptamos este criterio "aproximado" se puede resolver el problema usando el mismo método iterativo que antes.

Lo único que hay que hacer es cambiar el código de la función fun.m para que ahora devuelva un vector F (ahora de tamaño 3x1) con las tres evaluaciones (f1, f2, f3) de las ecuaciones. Podéis partir de la función fun del primer caso (las dos primeras circunferencias son comunes), añadiendo una 3ª componente a F que evalúe la 3ª ecuación del sistema. Notad que el vector de entrada X sigue teniendo 2 componentes (x,y) pero la salida F es un vector 3x1 (f1, f2, f3). La matriz J debe ser ahora de tamaño (3x2): 3 ecuaciones x 2 incógnitas. Probando con **`[F,J]=fun([1;-1])`** debéis obtener:

```
                1.0990          0.9810    0.2055
F=             -3.1716    J = -0.6980   -0.6980
                4.5711          0.7106   -0.7035
```

Volcad los valores de F y J obtenidos para X=[-1; 2]

Una vez lista la función fun.m ejecutad `>>demo2d 2` y probad con unos 20 puntos iniciales, repartidos por toda la imagen. Adjuntad captura del resultado. Haced un zoom en la zona de la figura donde convergen las iteraciones y adjuntad una captura. ¿Es razonable ese punto como solución aproximada del sistema?

¿A qué solución (x,y) converge el método? ¿Se cumplen las ecuaciones? Evaluar fun en dicha solución y volcad el resultado. Desde la perspectiva del ajuste de datos, ¿qué interpretación le daríamos a los valores obtenidos al evaluar fun() en la "solución"?

## 2. Determinación posición/orientación de una cámara

Una vez que entendemos cómo funcionan estos métodos iterativos los usaremos para una aplicación real, donde el uso de derivadas aproximadas es fundamental ya que no dispondremos de la expresión analítica de la función.

El problema es determinar la posición y orientación que tenía una cámara cuando se tomó una fotografía. La información de la que se dispone son las coordenadas 3D (Este, Norte, Altura) de ciertos puntos que salen en la foto y las coordenadas 2D (píxeles) donde esos puntos aparecen en la fotografía.

También tenemos una función **`function [px,py]=mod_cam(coord3D,X)`** que modela la proyección de un punto 3D -> 2D que tiene lugar en mi cámara al tomar una fotografía (teniendo en cuenta aspectos como la focal usada, número de píxeles y tamaño del sensor, distorsiones de la lente, etc.).

Los argumentos de entrada de esta función son:

- **coord3D**: vector 3x1 o matriz 3xN con las coordenadas 3D (Este, Norte, Altura) de 1 o más puntos del mundo exterior a los que tomo la foto.

- **X:** vector columna (6x1) con la posición y orientación de la cámara:

  - X(1:3) = posición de la cámara (coordenadas E, N, h en metros).
  - X(4:6) = tres ángulos (en grados º) con la orientación de la cámara.

Los tres ángulos usados para describir la orientación son el **azimuth** o dirección en la que apunta la cámara respecto al Norte (Norte 0º, Este 90º, Oeste -90º, Sur 180º), la **elevación** (si la cámara apunta hacia arriba o hacia abajo) y la **inclinación** (si el horizonte esta nivelado o inclinado a derecha o izquierda).

La salida (px,py) son dos vectores 1xN con la posición (en píxeles) donde caerían esos N puntos sobre la foto: en px su posición horizontal y en py la vertical. La función `mod_cam()` me permite saber (si conozco la posición de la cámara y hacia donde apunta) en qué píxel de la foto aparece un punto/s de coordenadas 3D dadas:

3D (E,N,h) + X ---> 2D en foto (px,py)

Por ejemplo si ejecutáis:

```
punto3D = [303000; 4505000; 1200];
X   = [300000; 4500000; 1000; 10; 2; 0];
[px,py]=mod_cam(punto3D,X),
```

obtendréis **px=2.6681e+03~2668 y py=1.0056e+03~1006**.

Esto quiere decir que si coloco mi cámara en la posición E=300000, N=4500000, h=1000, apuntando en una dirección 10º respecto del Norte (desviada hacia el Este), mirando un poco (elevación=2º) hacia arriba y tomo una foto, un punto P de coordenadas [303000 4505000 1200] aparecería aproximadamente en el píxel (2668, 1006) de mi foto.

Mi problema es ligeramente distinto: hallar la posición y la orientación (vector X) de la cámara que hace que ciertos puntos de control con coordenadas 3D (E,N,h) conocidas salgan en la foto en ciertos píxeles 2D (px,ppy) también conocidas:

3D (E,N,h) + X ?? ---> 2D en foto (px, py)

Si os fijáis esto es muy similar al problema de la anterior práctica de calibrar un mapa digital a partir de las coordenadas conocidas de unos puntos de control. La diferencia es que ahora la relación entre coordenadas 3D y píxeles 2D involucra operaciones no lineales y no conozco los detalles de las operaciones dentro de la función mod_cam(), por lo que no podremos despejar los parámetros (X) de la transformación. Vamos a ver cómo resolver este problema con los métodos que hemos aprendimos en la primera parte de esta práctica.

## 2.1 Datos iniciales: coordenadas 3D/píxeles 2D de los puntos de control

En la figura adjunta se muestran los N=4 puntos de control cuyas coordenadas 3D conozco. De izquierda a derecha en la foto (señalados con círculos rojos):

| | 3ºHermanito | Casquerazo | Pico Huertos | Risco Negro |
|---|---|---|---|---|
| Este: | 306188 | 305888 | 305463 | 305690 |
| Norte: | 4457504 | 4457242 | 4458875 | 4459485 |
| Altura: | 2339 | 2436 | 2460 | 2289 |

Cread un nuevo script e inicializarlo con el siguiente código, que lee la info de los puntos de control (en datos.mat) y carga/visualiza la imagen que usaremos.

```
clear; global ctrl_3D ctrl_PX ctrl_PY
load datos;
im=imread('gredos.jpg'); figure(1); image(im);
```

Junto con **ctrl_3D** (matriz 3x4 con las coordenadas 3D de los puntos de control) hay declaradas otras dos variables (**ctrl_PX, ctrl_PY**) con las posiciones de esos puntos sobre la foto (en píxeles). Haced un plot (modificador 'ro') superponiendo (hold on) en la foto dichos puntos para comprobar que caen dónde deben. Veréis que solo aparecen 3 puntos, ya que falta el primero de ellos. Vuestra primera tarea será medir sobre la foto la posición de ese primer punto. Haced zoom dos o tres veces para ampliar la zona alrededor de 1$^{er}$ punto de control (el más a la izquierda) y usad el cursor de datos para determinar su posición en píxeles sobre la imagen. Escoged el punto que parezca la posición más alta (cima) del risco en cuestión. ¿Posición 1$^{er}$ punto sobre imagen?

Asignar la posición encontrada para el 1$^{er}$ punto a la primera casilla de ctrl_PX y ctrl_PY y volved a hacer un plot de los puntos para comprobar que habéis asignado correctamente la posición del punto que faltaba. Haced un zoom sobre la zona del primer punto de control y adjuntar la imagen resultante.

## 2.2) Completar la función fun_pix para obtener F y J

**Cálculo del vector de residuos F:** Una vez que tenemos las coordenadas 3D (`ctrl_3D`) de los 4 puntos de control, y su posición (en píxeles) sobre la imagen (`ctrl_PX,ctrl_PY`), el siguiente paso es crear una función (análoga a la función fun.m de antes) que se anule cuando encontremos la posición/orientación correcta de la cámara. Partid del template suministrado en **`fun_pix.m`**

El argumento de entrada de esta función es el vector X (6x1) con las incógnitas, los 6 parámetros de la cámara: 3 de la posición (Este, Norte y altura en metros) junto con los 3 ángulos de orientación (azimuth, elevación e inclinación en º).

Las variables **`ctrl_3D,ctrl_PX,ctrl_PY`** con la información 3D y 2D de los puntos de control están declaradas como variables globales para que sean visibles desde la función sin tener que pasarlas como argumentos.

Como antes, la función tendrá 2 posibles salidas. En este apartado solo estamos preocupados por la primera de ellas. La salida F es un vector con las diferencias entre posiciones reales en la imagen (**`ctrl_PX,ctrl_PY`**) y las calculadas (px,py) cuando proyectamos los datos de **`ctrl_3D`** usando el modelo de la cámara con los parámetros dados por X. Esta primera parte de la rutina es muy sencilla:

1) Usar mod_cam para proyectar las coordenadas 3D de los puntos de control (ctrl_3D) usando el vector de parámetros X, para obtener las posiciones (px, py) sobre la imagen que predice el modelo de mi cámara.
2) Hallar las diferencias en horizontal y en vertical entre las posiciones reales sobre la foto (ctrl_PX, ctrl_PY) y las obtenidas por el modelo (px, py).
3) Finalmente devolver esas diferencias en el argumento de salida F, como un **vector columna.** Para ello juntad ambos vectores diferencia uno al lado del otro y trasponer el resultado para obtener F. Comprobad que la salida sea un vector columna de tamaño 8x1.

La función fun_pix(X) es el equivalente a la función fun(X) de la primera parte. Como entonces, el objetivo es hallar una X tal que las diferencias F calculadas con F=fun_pix(X) sean nulas (o muy pequeñas). En ese caso los puntos caerán justo (o casi) donde los vemos en la foto, indicando que hemos averiguado la posición y la orientación de la cámara con las que se tomó la foto. Resolver el problema consiste en hallar el valor de X tal que fun_pix(X)=0. Para el valor de `X=[308000;4458000;2000;-100;0;0]`, volcad valores de F devueltos por fun_pix(X). ¿Cuál de los 4 puntos de control muestra los mayores residuos para esta X?

**Cálculo de la matriz J del sistema:** Vamos a completar la función fun_pix para que, además de F, también calcule la matriz J correspondiente a este problema. Como antes, tras calcular el primer argumento F añadid en la función la línea:

```
if nargout==1, return; end
```

que hace que el resto del código no se ejecute si no se pide el 2º argumento J.

Para conocer el tamaño de J preguntaremos por la longitud N del vector X de incógnitas y la longitud M del vector F ya evaluado. J será una matriz de tamaño M x N: en este caso serán 8 filas (longitud de F) y 6 columnas (longitud de X).

Recordad que siempre necesitamos un número de ecuaciones mayor o igual que el número de incógnitas (6). En este caso tenemos 4 puntos, cada uno de los cuales da lugar a 2 ecuaciones (una en x y otra en y), por lo que tendremos un sistema sobredeterminado de tamaño 8x6, cuya solución será aproximada, como nos pasaba en el problema de las tres circunferencias del apartado 2).

Para calcular cada columna haremos como antes, aproximando la derivada de la función fun_pix de la misma forma. Por ejemplo para la 1ª columna:

$$J(:,1) = (fun\_pix(X+h_1) - F) / h \quad \text{siendo } h_1 = [h\ 0\ 0\ 0\ 0\ 0]' \quad (\text{con } h = 0.1).$$

Repitiendo en las diferentes direcciones (moviendo de posición la componente h) aproximaremos las diferentes derivadas de la función fun_pix con respecto a las 6 variables, creando así las 6 columnas de la matriz J.

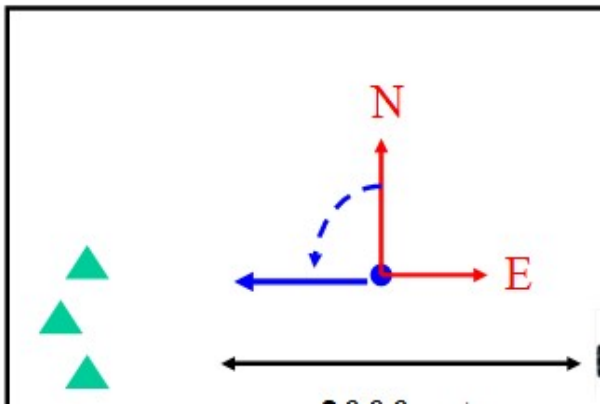
Reservar una matriz J del tamaño adecuado y con un bucle rellenar sus columnas siguiendo las indicaciones anteriores. Usad h = 0.1 para aproximar las derivadas.
Adjuntad el código completo de fun_pix (se valorará que no haya que modificar el cálculo de J aunque se cambie el número de ecuaciones o incógnitas).
Para el valor de `X=[308000;4458000;2000;-100;0;0]` volcad la matriz J obtenida.

Observando la matriz J, ¿cuál de los 6 parámetros causa un mayor cambio en las posiciones horizontales px de los píxeles? Justificar.

### 2.3) Elección de la hipótesis inicial y código de la iteración

En un problema real como este es fundamental tener una idea de por dónde está la solución para poder elegir una hipótesis de partida X0 razonable. Para estimar la posición y orientación inicial de la cámara (vector X0) usaremos las siguientes indicaciones, en base a mis recuerdos sobre cómo se tomó la foto:

- **Posición Este**: creo recordar que la foto la tomé mirando más o menos hacia el Oeste, a unos 2000 m de las montañas que aparecen en la foto. Por lo tanto para la coordenada E0 inicial de la cámara sumaré 2000 m a la media de las coordenadas Este de los puntos de control.
- **Posición Norte:** dado el esquema anterior la coordenada norte N0 será aproximadamente la misma que la de media de las coordenadas Norte de los cuatro puntos de control.
- **Altura**: como los puntos de control son montañas y yo estoy más abajo, restaré 500 metros a la media de los puntos de control como una primera hipótesis para la altura inicial H0.



- **Azimuth:** (ángulo entre la dirección de la cámara y el Norte). Si la foto se tomó apuntando hacia el Oeste el ángulo de azimuth será ~ -90º.
- **Elevación e Inclinación:** una hipótesis razonable es inicializarlos a 0º, asumiendo que la cámara estaba nivelada al tomar la foto.

Vector X0 (6x1) con los valores iniciales usados (usad grados para los ángulos).

Una vez que tenemos de la función **`fun_pix`**, junto con una hipótesis razonable de partida (X0) completaremos nuestro script para resolver el problema. El código de la iteración será muy similar al que escribimos en el 1er apartado:

1) Inicializar X=X0 (hipótesis inicial) y hacer dX=1.
2) Repetir los pasos siguientes (mientras que ||dX||>=0.1) :
   a) Evaluar F y J para la hipótesis X actual, usando [F J]=fun_pix(X).
   b) Calcular incremento dX resolviendo con MATLAB el sistema J·dX=-F.
   c) Actualizar la solución X sumándole el incremento dX.

El criterio de parada usado es que la norma de dX sea menor que 0.1. Eso quiere decir que hemos llegado a un punto en el que la variación de los parámetros requerida es tan pequeña (del orden de 0.1 metros en la posición o 0.1º en los ángulos) que no merece la pena seguir iterando.

Para apreciar la progresión de la iteración, en cada paso del bucle volcad usando fprintf el valor de la norma de F con 2 decimales (%.2f). Si todo va bien esos valores deben ir disminuyendo, lo que indica que los píxeles proyectados con el modelo son cada vez más parecidos a los medidos sobre la foto: el proceso converge y la solución X es cada vez mejor. Adjuntad volcado con los resultados de las iteraciones. ¿Cuántas iteraciones hacéis?

Al terminar el bucle de la iteración presentar los resultados finales:

1) Volcar con fprintf el vector X con los parámetros finales usando un decimal (%.1f) tanto para las posiciones (0.1 m) como para los ángulos (0.1º).
2) Volcar las discrepancias finales entre los píxeles medidos sobre la foto y los proyectados con la solución final X. Salvo que hayáis sido muy descuidados al marcar el 1$^{er}$ punto, los residuos deben ser del orden unos pocos píxeles o incluso menores.

Adjuntad volcado de parámetros finales y de los residuos no corregidos.
¿Eran correctos mis recuerdos sobre la dirección en la que se tomó la fotografía?

Visualizar gráficamente los residuos superponiendo sobre la foto los puntos de control: en color rojo los valores "correctos" (ctrl_PX,ctrl_PY) y en azul los proyectados (px,py) con el modelo usando los parámetros X definitivos. Hacer zoom sobre alguno de los puntos para ver las diferencias (que deberían ser del orden de 1-2 píxeles como mucho). Adjuntad zoom de la imagen en alguno de los puntos de control para visualizar los residuos anteriores.

Adjuntad vuestro script con todo el código del proceso de iteración.

## 2.4) Uso de la calibración encontrada

Finalmente veremos para que podemos usar el hecho de conocer la posición y la orientación de la cámara en el momento de tomar la fotografía. Si hacéis **`>>load ruta3D`** veréis una matriz ruta (tamaño 3x200) con las coordenadas 3D (E, N, h) de un recorrido que va desde un refugio de la zona hasta el Pico de los Huertos (que coincide con uno de los puntos usados en la calibración).

La ruta se muestra en la figura adjunta, donde hemos usado las coordenadas E y N para superponerla sobre un mapa como hicisteis en la práctica anterior. En esa práctica vimos que para superponer la ruta en un mapa hay que tener un **mapa calibrado**, donde se conozca la transformación que nos permite pasar de las coordenadas (E,N) a los píxeles (x,y).


Lo que hemos hecho en el apartado anterior al encontrar la posición y orientación de la cámara es equivalente a "calibrar" nuestra foto. Conocidos los parámetros X podemos ahora usar mod_cam() para superponer sobre la foto cualquier punto de coordenadas 3D dadas. Usando **`mod_cam()`** y la solución X encontrada en el apartado anterior, proyectad las coordenadas 3D de la ruta a píxeles 2D sobre la imagen. Marcar la posición del refugio (1er punto de la ruta) sobre la foto con un cuadrado verde, con las opciones:

```
plot(...,'gs','MarkerFaceCol','g','MarkerSize',6)
```

Luego pintar la ruta usando una línea discontinua de color rojo con el comando:

```
plot(...,'r:','LineWidth',2);
```

Adjuntad código usado para mostrar las gráficas y la foto con el resultado final. ¿Es visible el refugio desde el punto en el que se tomo la fotografía?

En la resolución del problema hemos usado 4 puntos de control, llegando a un sistema con 8 ecuaciones y 6 incógnitas. Al tener solo 6 incógnitas nos bastaría con 3 puntos de control (para un total de 6 ecuaciones y 6 incógnitas). Volver a ejecutar vuestro script pero ahora usando únicamente los 3 primeros puntos de control. Si habéis hecho correctamente el cálculo de J no necesitaréis cambiar nada en el código. Volcad los residuos finales obtenidos en este caso. ¿Qué valor tienen? ¿Por qué? ¿Indica esto que la nueva solución es más fiable?

Para ver los problemas de resolver sin disponer de puntos redundantes restad 100 píxeles al valor de ctrl_PY para el primer punto (simulando una equivocación al marcar los puntos en la foto y volver a resolver usando esos 3 primeros puntos. Volcar los residuos obtenidos. Usando la nueva solución X obtenida en este caso volver a proyectar la ruta sobre la foto. Adjuntad la imagen resultante.

## 2.5) Mejora de la robustez del algoritmo (backtracking)

Tener menos puntos de control también puede afectar a la convergencia, como pasaba en la primera parte de la práctica, donde la solución aproximada (con tres circunferencias) tenía mejores propiedades de convergencia que la solución exacta (intersección de dos circunferencias).

Si observáis la norma de los residuos durante el proceso de convergencia en los casos anteriores es posible que veáis que en alguno de los pasos dicha norma aumente. Eso indica que en ciertas ocasiones el resultado final de un paso es peor que el inicial. Aunque en los casos anteriores este comportamiento no parece afectar a la convergencia final es conveniente evitar esta situación (que podría dar problemas en casos más críticos).

Por ejemplo, si intentáis resolver el problema usando únicamente los tres últimos puntos de control veréis que la iteración no converge (al menos con la hipótesis inicial X0 usada).

Vamos a tratar de evitar este problema complementando el algoritmo con una técnica denominada de "backtracking".

La idea es que antes de aceptar el paso indicado por dX se evalúan los residuos en la nueva solución fun_pix(X+dX) para comprobar si su norma disminuye. Si es así se acepta el paso. En caso de que la norma de los residuos aumente se reduce el paso (p.e a dX/2) y se vuelve a probar. El proceso se repite hasta que la norma de los residuos disminuya.

Este algoritmo se denomina de "backtracking" porque si el desplazamiento dX no reduce los residuos, retrocedemos y probamos con un paso dX/2, dX/4, etc. La idea es que el vector dX nos indica la dirección en la que tenemos que mover la solución. Sin embargo es posible pasarnos del punto ideal y obtener un resultado peor que el original: en ese caso seguimos probando en la misma dirección dX, pero avanzando menos (dX/2, dX/4, etc.) hasta encontrar el paso que reduzca el error.

Adjuntad el código añadido para implementar el algoritmo de backtracking.

Una vez que tengáis implementado el backtracking, aplicadlo al caso de resolver el problema usando únicamente los tres últimos puntos (que como hemos visto antes no convergía con el algoritmo original).

En este caso debéis ver que en todos los pasos el error ||F|| disminuye. Aunque el número de iteraciones aumente se recupera la convergencia.

Adjuntad volcado del ||F|| en las sucesivas iteraciones y el resultado final X obtenido (volcando sus valores con 1 decimal).