Universidad Autónoma de Ciudad Juárez

Instituto de Ingeniería y Tecnología

Departamento de Ingeniería Eléctrica y Computación

Reconstrucción de superficies mediante procesos Gaussianos

Tesis presentada por Manuel Guillermo López Buenfil (98671)
para obtener el grado de Ingeniero en Sistemas Computacionales

Asesor: Boris de Jesús Mederos Madrazo

16 de mayo de 2014

Autorización de Impresión

Los abajo firmantes, miembros del comité evaluador, autorizamos la impresión del proyecto de titulación:

Reconstrucción de superficies mediante procesos Gaussianos

Elaborado por el alumno:

Manuel Guillermo López Buenfil (98671)

Jorge Rodas Osollo
Profesor de la Materia

Boris Mederos Madrazo
Asesor

Declaración de Originalidad

Yo, Manuel Guillermo López Buenfil, declaro que el material contenido en esta publicación fue generado con la revisión de los documentos que se mencionan en la sección de Referencias, y que el Programa de Cómputo (Software) desarrollado es original y no ha sido copiado de ninguna otra fuente, ni ha sido usado para obtener otro título o reconocimiento en otra Institución de Educación Superior.

Manuel Guillermo López Buenfil

Agradecimientos

Agradezco a las siguientes personas por su apoyo:

- A mi asesor, Boris de Jesús Mederos Madrazo, por introducirme a las ideas presentadas en esta tesis, por su conocimiento y por sus ideas
- A mi familia, por su ayuda en momentos de estrés
- A la comunidad de *Stack Exchange*, por su habilidad para resolver preguntas

Índice general

Autorización de Impresión 2

Declaración de Originalidad 3

Agradecimientos 4

1. Introducción 12

1.1. Definición del problema . . . 12
1.2. Antecedentes . . . 13
1.3. Objetivo . . . 15
1.4. Preguntas de investigación . . . 15
1.5. Justificación . . . 15
1.6. Solución propuesta . . . 16
1.7. Método propuesto . . . 16

2. Marco teórico 17

2.1. Superficie implícita . . . 18
2.2. Partición de la Unidad . . . 18
2.3. Partición Multinivel de la Unidad (MPU) . . . 19
2.4. Cálculo de la superficie implícita . . . 21
2.4.1. Cálculo de la superficie como función de $\mathbb{R}^2$ en $\mathbb{R}$ . . . 21
2.4.2. Cálculo de la superficie como función de $\mathbb{R}^3$ en $\mathbb{R}$ . . . 23
2.5. Aproximación local por el método MPU . . . 24
2.6. Modificación propuesta . . . 25
2.7. Procesos Gaussianos . . . 26

2.8. Procesos Gaussianos para regresión 26
2.9. Regresión Online 29
2.10. Regresión Dispersa 33
2.11. Kernel 36
2.12. Triangulación de la superficie implícita 36

3. Desarrollo **38**
3.1. Kernels 38
3.2. Optimización de parámetros 39
3.2.1. Gradiente del Kernel 40
3.2.2. Estructura de log-verosimilitud 41
3.3. Algoritmo MPU 44
3.4. Condiciones de división 44
3.5. Proceso de aprendizaje 45
3.5.1. Análisis de componentes principales 46
3.5.2. Desplazamiento de normales 48
3.6. Cálculo del error 48
3.7. Pesos 49
3.8. Triangulación 51
3.9. Evaluación de la superficie reconstruida 51
3.10. Análisis de complejidad 52
3.10.1. Proceso Gaussiano en Batch 52
3.10.2. Proceso Gaussiano Online Disperso 53

4. Resultados **54**
4.1. Modelos con suficientes puntos 54
4.2. Modelos con bordes 58
4.3. Modelos con regiones faltantes 60
4.4. Modelos con detalles finos 61
4.5. Modelos con ruido 64
4.6. Modelos dispersos 65
4.7. Proceso Gaussiano online disperso 67
4.8. Cantidad de divisiones 70

5. Conclusión y trabajo futuro **73**
5.1. Contribuciones 73
5.2. Trabajo futuro 74

Bibliografía **75**

A. Glosario **79**

B. Detalles de implementación **81**
B.1. Herramientas utilizadas 81
B.2. Procesos Gaussianos 81
B.3. Kernels 82
B.4. Optimización de parámetros 83
B.4.1. Gradiente del Kernel 85
B.5. MPU 85
B.6. Hilos 86
B.7. Parámetros 86

C. Código reutilizable **88**
C.1. Thread Pool 88

D. Manual Técnico **92**

Índice de figuras

2.1. Nodo con su soporte compacto 21
2.2. Superficie y su plano base 23
2.3. Media como aproximación de la superficie 28
2.4. Aproximación online del proceso 30
2.5. Actualización del conjunto BV 36

3.1. Función Goldstein-Price 42
3.2. Error según parámetros 42
3.3. Error de kernels 43
3.4. Análisis de componentes principales 47
3.5. Desplazamiento de normales 49
3.6. Spline cuadrático 50

4.1. Modelo Stanford Bunny 55
4.2. Modelo Armadillo 56
4.3. Modelo Knot 57
4.4. Modelo Double torus 58
4.5. Modelo Cubo 59
4.6. Modelo Squirrel 60
4.7. Modelo Dino 61
4.8. Modelo Dino 62
4.9. Modelo Dragon 63
4.10. Modelo Knot con ruido 64
4.11. Modelo Stanford Bunny con ruido 65
4.12. Modelo Knot disperso 66

4.13. Modelo Stanford Bunny disperso 66
4.14. Modelos Online 67
4.15. Modelos Online 68
4.16. Modelos Online 69
4.17. Modelos Online 70
4.18. Octree del modelo Stanford Bunny 71
4.19. Octree del modelo Knot 71

Índice de cuadros

3.1. Kernels 38

Lista de algoritmos

1. MPU 44
2. MPU paralelizable 45
3. Aprendizaje 46

Capítulo 1

Introducción

En muchas aplicaciones (tales como odontología, manufactura, arqueología o aplicaciones culturales) es necesario utilizar modelos tridimensionales (superficies poligonales) que provengan de objetos del mundo real, tales como esculturas, piezas de maquinaria, objetos arqueológicos, prototipos de objetos para la creación de juegos, entre otros. Para esto se utilizan dispositivos de adquisición tales como los escáneres 3D, que generan una nube de puntos que se usa para reconstruir un modelo del objeto original.

1.1. Definición del problema

Se tiene un conjunto finito de puntos que pertenecen o aproximan a un objeto tridimensional, y se requiere crear un modelo que represente el objeto original de una manera precisa.

La descripción formal del problema es la siguiente: Dado un conjunto finito de puntos P en $\mathbb{R}^3$ muestreados de una superficie S en el espacio, el objetivo es obtener una función implícita $F : \mathbb{R}^3 \rightarrow \mathbb{R}$ tal que la isosuperficie de nivel cero, $F^{-1}(0)$, aproxime adaptativamente a S usando control de error local. Este problema presenta los siguientes retos:

- Los puntos pueden presentar ruido y anomalías (valores atípicos)
- Se pueden tener zonas con insuficientes puntos

- Se busca conservar los detalles de la textura del objeto
- La cantidad de puntos puede ser muy grande
- Los objetos pueden presentar una topología compleja

1.2. Antecedentes

Existe un amplia gama de algoritmos para la reconstrucción de superficies que han sido propuestos en años recientes. Para tratar este problema diferentes enfoques han sido propuestos, basados en diferentes áreas de la matemática, como geometría y topología computacional, técnicas de la teoría de aproximación y métodos basados en ideas estadísticas y de *machine learning*. En la primera categoría, los métodos se basan en la triangulación de Delaunay-Voronoy. El trabajo pionero en esta área fue propuesto por Edelsbrunner y Mucke [13], conocido como el algoritmo *alpha shape*. Posteriormente, Amenta y Bern [1] dieron algoritmos eficientes con garantías teóricas basados en la triangulaciónn de Delaunay, introduciendo el concepto de *Polo*, que es clave en las técnicas modernas de reconstrucción basadas en la triangulación de Delaunay. Además, por primera vez introducieron condiciones de muestreo conocidas como *r-muestreo*, bajo las cuales probaron que para un determinando *r-muestreo* (r menor a cierto valor), la superficie reconstruida por su algoritmo es homeomorfa a la superficie original del objeto geométrico que se está reconstruyendo. Posteriores mejoras de estas ideas han sido presentadas en los trabajos de Amenta, Choi, Dey y Leekha [2], Dey et. al. [12] [11]. También en Mederos et. al. [19] se hicieron extensiones al trabajo de Amenta para el caso con ruido.

Los métodos basados en teoría de aproximación se basan en obtener una superficie implícita que pasa por lo puntos mediante la obtención de una función que tiene como conjunto de nivel 0 a esta superficie. Para obtener esta función, diferentes estrategias han sido propuestas, casi todas basadas en la idea de las *Radial Basis Functions* (RBF). Por ejemplo, Savchenko et. al. [26], Carr et. al. [5], Turk y O'Brien [14], Carr et. al. [6] presentaron métodos que eficientemente reconstruyen la superficie, pero que tienen un costo computacional elevado.

Otra idea relacionada con teoría de la aproximación es el método de *Moving Least*

Square (MLS), introducido por Levin [15] y sus posteriores extensiones (Amenta y Kil [3]).

Recientemente, han sido usados exitosamente métodos basados en descomposición del dominio, como el método MPU propuesto por Ohtake et. al. [21]. La idea de estos métodos es facilitar y controlar la aproximación dividiendo el dominio en partes más pequeñas donde se pueda realizar de manera más fácil una aproximación local. Estas aproximaciones locales pueden ser determinadas de diferentes maneras usando distintos enfoques, tales como los provistos por la teoría de aproximación, enfoques estadísticos, y del área de *machine learning*. Scholkopf et. al. [27] usaron la técnica de *machine learning* conocida como *support vector machine* para reconstruir superficies de manera implícita.

En este trabajo se pretende aplicar esta idea del método MPU usando técnicas estadísticas provenientes del área de *machine learning*, conocidas como *procesos Gaussianos* [24]. Vale la pena recalcar que hasta el momento esta técnica no se ha aplicado en este contexto (con excepción de [31], el cual requiere una cantidad de puntos muy pequeña, y por lo tanto, no obtiene resultados satisfactorios). Se conoce que la técnica de *procesos Gaussianos* puede aproximar eficientemente a un conjunto de datos con ruido usando poca información presente en este conjunto, es decir, usando sólo la información relevante dentro de la nube de puntos (*proceso Gaussiano disperso*). Basado en lo anterior, esperamos que esta combinación permita reconstruir superficies adecuadamente, lo cual produciría una compresión significativa de los datos usados en la reconstrucción, manteniendo la topología de la superficie, y permita manejar eficientemente nubes de datos gigantescas. En el caso de usar el proceso Gaussiano en batch, pretendemos obtener superficies con un nivel de detalle mayor que el método MPU, el cual es uno de los métodos del estado del arte en reconstrucción de superficies. También se pretende lograr una reconstrucción en regiones con datos faltantes o densidad baja de puntos mejor que la del método MPU, además de lograr reconstruir superficies con topología compleja.

El esquema de reconstrucción propuesto es una extensión de la reconstrucción de superficies tridimensionales propuesta en [29]. En esta extensión introducimos una aproximación local basada en procesos Gaussianos. El algoritmo se basa en las ideas principales de MPU que subdivide jerárquicamente el dominio en varias partes y

después calcula aproximaciones locales en cada parte.

1.3. Objetivo

Mejorar las técnicas existentes para reconstruir la superficie del objeto tridimensional a partir de la nube de puntos:

- Lograr una buena reconstrucción de zonas donde no haya suficientes puntos
    - Una reconstrucción es buena si se parece al objeto original del que provienen los puntos
- Preservar detalles del objeto
- Reconstruir superficies usando sólo los datos más representativos
- Reconstruir nubes de puntos con topología compleja
- Crear una implementación computacionalmente eficiente de este método

1.4. Preguntas de investigación

¿Cómo se puede lograr una mejor reconstrucción de superficies en base a una nube de puntos?

- ¿Cómo se puede reconstruir la superficie en zonas con insuficientes puntos?
- ¿Cómo preservar los detalles relevantes de la nube de puntos?
- ¿Cómo reconstruir superficies usando sólo los datos relevantes de la nube de puntos?

1.5. Justificación

Actualmente existen métodos para reconstrucción de superficie, pero suelen producir resultados no muy satisfactorios ante la presencia de ruido (valores atípicos) o

de datos faltantes (regiones incompletas). Se busca crear un algoritmo que sea robusto a estas condiciones, lo que daría lugar a una mejor reconstrucción de las superficies del objeto preservando detalles significativos presentes en la nube de puntos.

1.6. Solución propuesta

Se propone partir del método MPU (*Multi-level Partition of Unity*) y modificarlo mediante la implementación de una técnica de aproximación conocida como regresión mediante proceso Gaussiano (*GP regression*). Si el objetivo anterior se cumple y el tiempo lo permite, se tienen como posibles expansiones el implementar diversas variantes de este tipo de regresión, tales como el proceso Gaussiano disperso (sparse GP).

1.7. Método propuesto

Se propone un método nuevo que parte de una implementación existente de la técnica MPU (Multi-level Partition of Unity) y la combina con procesos Gaussianos.

- En primer lugar se realiza la formalización matemática de los procesos Gaussianos; en particular, se presenta ésta en el contexto de reconstrucción de superficies.
- Se modifica la técnica de aproximación de MPU usando la técnica basada en procesos Gaussianos.
- Validar los resultados con juegos de datos disponibles públicamente que presentan la complejidad mencionada en la definición del problema (ver si se completan las regiones donde faltan datos, cómo reacciona al ruido y valores atípicos, cómo preserva la textura y detalles)
- Comparar con técnicas anteriores, como el algoritmo MPU clásico

Capítulo 2

Marco teórico

La reconstrucción de superficies es un problema complejo, no sólo porque las relaciones de adyacencia y proximidad de los datos son desconocidos, sino también por una gran cantidad de adversidades:

Los datos con que trabajan estos algoritmos usualmente vienen de un scanner 3D. Actualmente, estos dispositivos son capaces de manejar objetos reales de gran complejidad, y las nubes de puntos resultantes de la adquisición de datos contienen detalles finos, grandes variaciones geométricas, topología compleja y texturas. Sin embargo, el proceso de capturar la nube de puntos introduce muestras dispersas, agujeros (debido a la oclusión de ciertas partes del objeto por otras partes) y ruido.

El método MPU es un esquema implícito para reconstrucción de superficies. Las superficies implícitas son una representación muy útil de objetos tridimensionales, principalmente porque la forma inferida es calculada por una fórmula que permite el cómputo de operaciones básicas de modelado en una forma relativamente sencilla. La mayoría de los bordes de objetos creados por el hombre están compuestos de varias partes que pueden ser aproximados por superficies algebraicas. Cuando la forma del objeto es compleja, un procedimiento común es elevar el grado algebraico para obtener una mayor precisión en la aproximación. Sin embargo, en este caso, debido a la pobre representación de estos métodos, algunos componentes inapropiadamente conectados aparecen en la superficie reconstruida. Una solución alternativa es descomponer el dominio jerárquicamente en partes compactas y obtener aproximaciones locales para el objeto en cada parte, y luego juntar todas las partes para obtener

una descripción global del objeto. Un esquema practico que usa dicha solución es el método MPU (Multilevel Partition of Unity implicit), el cual provee una aproximación adaptativa controlada por el error de la función de distancia con signo hacia la superficie.

Las técnicas existentes que utilizan MPU se basan en aproximaciones polinomiales, las cuales no dan una buena aproximación en superficies con bastantes irregularidades y ruido. Por tanto, en vez de usar aproximaciones polinomiales, se usarán aproximaciones locales dadas por un proceso Gaussiano, las cuales se van a determinar a través de un enfoque de regresión bayesiana conocido como *Gaussian Process Regression* [7, 24].

2.1. Superficie implícita

Un subconjunto $O \in \mathbb{R}^3$ es llamado una superficie implícita si existe una función suave (continuamente diferenciable) $F : U \to \mathbb{R}, O \subset U$ y un número real $c \in \mathbb{R}$ tal que $O = F^{-1}(c)$. La superficie implícita $F^{-1}(c)$ es regular si F es diferenciable y satisface la condición de que para cada punto $x \in F^{-1}(c)$ el gradiente de F en x no se desvanece, esto es, $\nabla F(x) \neq 0, \forall x \in F^{-1}(c)$.

2.2. Partición de la Unidad

Una partición de la unidad (PU) es una herramienta matemática muy útil para combinar aproximaciones locales con el objetivo de obtener una aproximación global de una superficie. Este algoritmo se basa en la técnica de divide y vencerás. Propiedades importantes tales como el error global máximo y el orden de convergencia son heredadas de las aproximaciones locales, las cuales se pueden realizar de una manera mucho más fácil.

El algoritmo básico para la construcción de la aproximación global usando partición de la unidad es el siguiente:

- Dividir el dominio en partes.

- Obtener una aproximación local para cada parte.

- Combinación de las aproximaciones locales para obtener una aproximación global.

Más precisamente, considere un dominio compacto $\omega \subset \mathbb{R}^3$ y sea $\{\phi_i\}_{i=1...n}$ el conjunto de funciones no negativas y suaves con soporte compacto en ω tal que $\sum_{i=1}^{n} \phi_i(x, y, z) = 1$ para todos los puntos $(x, y, z) \in \omega$. Sea F_i la clase de funciones definidas en $supp(\phi_i), \forall i = 1...n$. De cada clase F_i se escoge una función f_i que mejor aproxime los puntos de $P \cap supp(\phi_i)$. (P es el conjunto de puntos muestreados, $P \in \omega$). Una aproximación global para la función $F : \omega \rightarrow \mathbb{R}$ puede ser obtenida de la siguiente manera:

$$\sum_{i=1}^{n} \phi_i(x, y, z) f_i(x, y, z) \tag{2.1}$$

Las funciones f_i son determinadas de acuerdo a algún criterio de aproximación, generalmente mínimos cuadrados, *Ridge Regression*, u otros. En nuestro caso, estas funciones serán determinadas usando la técnica de procesos Gaussianos. La ecuación 2.1 es la base del algoritmo *Multilevel Partition of Unity* (MPU) propuesto por Ohtake et. al. [21].

2.3. Partición Multinivel de la Unidad (MPU)

El método de Partición multinivel de la Unidad (MPU) fue propuesto por Ohtake et. al. [21] originalmente para construir una superficie implícita que aproxime un conjunto de puntos con sus vectores normales asociados en $\mathbb{R}^3$. El método MPU utiliza una partición de la unidad para obtener una aproximación global de la superficie implícita del objeto combinando aproximaciones locales.

El método MPU se divide en las siguientes etapas:

- Dividir el dominio en partes. Usa un octree como esquema jerárquico para guiar la subdivisión del dominio.

- Regresión en cada uno de las hojas del octree para obtener una superficie local definida implícitamente por f_i, usando un subconjunto de los datos que le pertenezcan a dicha hoja.

- Unión de las distintas superficies locales para conformar una sola superficie global implícita, mediante la expresión 2.1.
- Triangulación de la superficie implícita. Se utiliza el método Marching Cubes [16, 17].

El método inicia aplicando una normalización a los puntos de P, de forma que todos los puntos queden contenidos en el cubo $H = [-1, 1]^3$. En adelante, usaremos a P para denotar el conjunto de puntos normalizados. El método construye un octree usando un procedimiento recursivo en el cual la subdivisión de cada nodo es controlada por el error de la aproximación local. En otras palabras, el criterio de refinamiento para un nodo i del octree consiste en calcular el error local de la aproximación, y si este error es mayor que una tolerancia dada, entonces el nodo se subdivide en 8 nuevos nodos, y en cada uno de éstos se aplica la misma prueba recursivamente. El error local puede calcularse de las siguientes maneras:

- Error máximo.
- Error cuadrático medio.
- Método de Taubin [30].

Cada nodo i del octree es asociado con la función ϕ_i usada en la ecuación (2.1). El soporte compacto de ϕ_i es definido como una esfera de radio r_i centrado en el medio del nodo i. El radio es elegido proporcionalmente al tamaño de la diagonal d_i del nodo i. La proporción exacta utilizada es un parámetro ajustable (ver figura 2.1). El método MPU que se propone utiliza regresión mediante procesos gaussianos para aproximar localmente el conjunto de puntos. Es decir, aproximar por una superficie los puntos que caen en la región de soporte de cada nodo. Algunas veces (especialmente cuando la densidad de P no es uniforme) la esfera de radio r_i del nodo i no contiene suficientes puntos para estimar adecuadamente la superficie implícita en dicho nodo. Si el número de puntos en la región de soporte de un nodo no es suficiente, entonces se aumenta el radio de la región de soporte hasta que se cuente con suficientes puntos.

Si se supone que la superficie S de la que los puntos P son muestreados es una isosuperficie de nivel 0 de una función $f : \mathbb{R}^3 \to \mathbb{R}$ entonces se puede obtener una

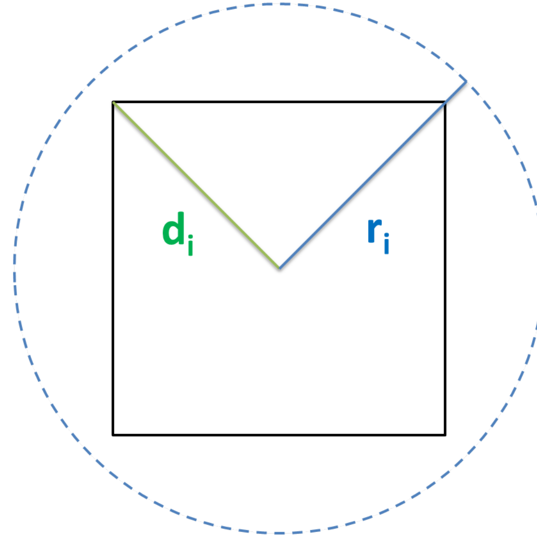


Figura 2.1: Representación en 2D de un nodo con su soporte compacto

función $F : \mathbb{R}^3 \to \mathbb{R}$ que aproxime globalmente a f:

$$f(x, y, z) \approx F(x, y, z) = \sum_{i=1}^{n_L} \phi_i(x, y, z) f_i(x, y, z) \tag{2.2}$$

Donde las funciones f_i son obtenidas mediante el método de procesos Gaussianos anteriormente mencionado.

2.4. Cálculo de la superficie implícita

Existen dos métodos para el cálculo de la superficie implícita en cada nodo: como una superficie de $\mathbb{R}^2$ en $\mathbb{R}$ ó como una superficie de $\mathbb{R}^3$ en $\mathbb{R}$.

2.4.1. Cálculo de la superficie como función de $\mathbb{R}^2$ en $\mathbb{R}$

El primer paso en cada nodo del octree es calcular un plano base en el cual sea más fácil representar la superficie como una función $s : \mathbb{R}^2 \to \mathbb{R}$, la cual nos dirá la "altura" de la superficie para cada punto de este plano. Para esto, se utiliza una de las técnicas siguientes:

- Promedio de normales: Usada en [18, 21], la desventaja de esta técnica es que requiere del conocimiento previo de las normales n_i de cada punto $p_i \in P$. El eje ortogonal al plano es determinado por el promedio de las normales:

$$v_3 = \frac{\sum_{i=1}^{n_L} n_i}{n_L},$$

uno de los ejes del plano es calculado como un vector v_1 arbitrario ortogonal a v_3, y el otro eje v_2 es calculado mediante la formula $v_2 = v_1 \times v_3$.

- Análisis de componentes principales: No requiere del conocimiento previo de las normales. Esta técnica es la que se ha adoptado para la solución propuesta. Se calcula la matriz de covarianza de los datos:

$$Cov = \sum_{i=1}^{n_L} (p_i - c)(p_i - c)^T,$$

donde c es el centroide de los puntos: $c = \frac{\sum_{i=1}^{n_L} p_i}{n_L}$. A esta matriz se le calculan los eigenvalores $\lambda_1 > \lambda_2 > \lambda_3$ y sus eigenvectores v_1, v_2, v_3 asociados. v_3 es la dirección de menos variación y corresponde al vector ortogonal al plano, v_1 y v_2 son los ejes del plano. (ver figura 2.2)

Los vectores v_1, v_2, v_3 que definen el sistema de coordenadas se considerarán con norma 1 para facilitar el cálculo de las nuevas coordenadas.

En este nuevo sistema de coordenadas, existe una alta probabilidad de que los puntos en el nodo se comporten como el gráfico de una función. Una vez que tenemos determinado el sistema de coordenadas, el cálculo de las coordenadas de un punto en este nuevo sistema se realiza a través de la función $T : \mathbb{R}^3 \to \mathbb{R}^3$ que lleva los puntos del espacio original al espacio que tiene como base el plano calculado. Si el punto original es (x', y', z'), entonces la función T tiene la siguiente forma:

$$T(x', y', z') = (x, y, z) = (\langle (x', y', z'), v_1 \rangle, \langle (x', y', z'), v_2 \rangle, \langle (x', y', z'), v_3 \rangle) \qquad (2.3)$$

Donde (x, y, z) son los puntos en el espacio transformado (Ver figura 2.2). Usando estos nuevos puntos, se procede a calcular la función s_i que represente a la superficie. Una vez se tenga esta función s_i, podemos calcular la función $f_i : \mathbb{R}^3 \to \mathbb{R}$ que

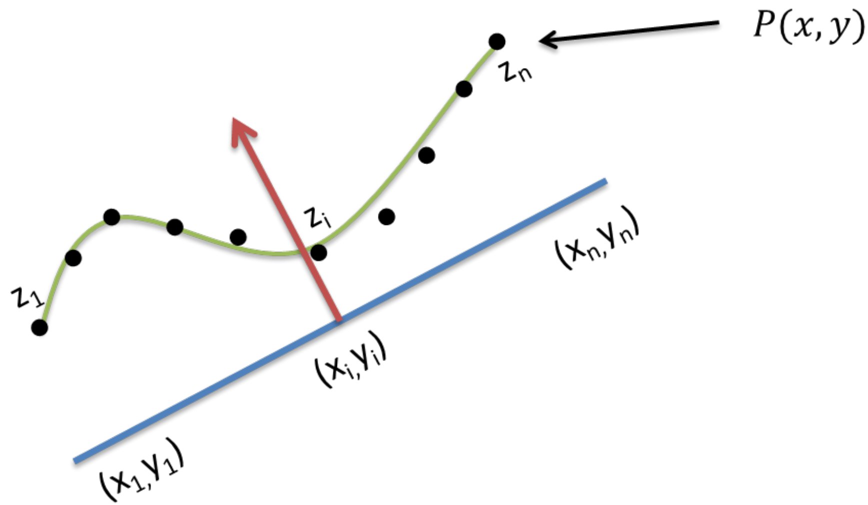


Figura 2.2: Representación en 2D de una superficie y su plano base calculado. El eje rojo representa al eje definido por el vector v_3, la recta azul representa al plano definido por los vectores v_1 y v_2, y las coordenadas (x_i, y_i, z_i) son las coordenadas del punto p_i en el nuevo sistema de coordenadas.

represente el conjunto de nivel 0 de la siguiente manera:

$$f_i(x, y, z) = z - s_i(x, y) \tag{2.4}$$

2.4.2. Cálculo de la superficie como función de $\mathbb{R}^3$ en $\mathbb{R}$

En este método se toman los puntos del nodo i con coordenadas (x, y, z) y se busca una función aproximante s_i de $\mathbb{R}^3$ en $\mathbb{R}$ que sea 0 o próximo a 0 en estos puntos. La aproximación local en este nodo es:

$$f_i(x, y, z) = s_i(x, y, z) \tag{2.5}$$

Generalmente, s_i se calcula usando el método de mínimos cuadrados, aunque otros enfoques también pudieran ser utilizados.

2.5. Aproximación local por el método MPU

El método usado en el algoritmo MPU para calcular s_i se basa en aproximaciones polinomiales usando la técnica de los mínimos cuadrados [21].

En cada nodo de las subdivisiones, el algoritmo propuesto en [21] determina un polinomio minimizando la siguiente función de energía:

$$e_{mpu}(a) = \sum_{i=1}^{n} P_d(x_i, y_i, z_i)^2 + \mu \sum_{i=1}^{8} (P_d(x_i^v, y_i^v, z_i^v) - d_i)^2 \tag{2.6}$$

Donde:

- P_d es el polinomio de orden 2 o 3 a determinar, a través de sus coeficientes $a = \{a_0, a_1, ..., a_k\}$ donde $k = 9$ o $k = 14$ según el orden del polinomio
- μ es una constante de balance (tradeoff) que determina el peso del ajuste a las normales con respecto al ajuste de los puntos
- $v_i = (x_i^v, y_i^v, z_i^v)$ son los 8 vértices del nodo del octree.
- $d_i = n_i^t(v_i - p_i)$ donde p_i es el punto más cercano a v_i dentro del nodo y n_i es su normal. Nótese que d_i es la distancia con signo de v_i a la nube de puntos

Para obtener una mejor aproximación basada en la información de las normales y puntos es el método conocido como *regularized gradient one fitting* (GOF) (T. Tasdizen et.al. [29]), en [18] se propone una modificación usando el método de Tasdizen et.al. y la técnica de subdivisión de MPU que aproxima la superficie con un polinomio que se calcula minimizando la siguiente función de aproximación:

$$e_{grad}(a) = \sum_{i=1}^{n} [P_d(x_i, y_i, z_i)^2 + \mu(n_i^t \nabla P_d(x_i, y_i, z_i) - 1)] + \lambda \|a\|_2. \tag{2.7}$$

Donde:

- P_d es el polinomio de orden 2 o 3 a determinar, a través de sus coeficientes $a = \{a_0, a_1, ..., a_k\}$ donde $k = 9$ o $k = 14$ según el orden del polinomio
- μ es una constante de balance (tradeoff) que determina el peso del ajuste a las normales con respecto al ajuste de los puntos

- λ es una constante de regularización que fuerza a que los coeficientes del polinomio sean pequeños para evitar sobreajuste [4]

Dentro de la ecuación 2.7, el término $P_d(x_i, y_i, z_i)^2$ representa el ajuste a los puntos, $\mu(n_i^t \nabla P_d(x_i, y_i, z_i) - 1)$ representa el ajuste a las normales, y $\lambda \|a\|_2$ es el término de regularización que previene el sobreajuste (*overfitting*).

También en [18], se propone una mejor aproximación polinomial en cada nodo, generalizando la función (2.7) de la siguiente forma:

$$e_{grad}(a) = \sum_{i=1}^{n} w_i [P_d(x_i, y_i, z_i)^2 + \mu(n_i^t \nabla P_d(x_i, y_i, z_i) - 1)] + \lambda \|a\|_2 \tag{2.8}$$

Donde w_i es un peso que penaliza los datos (x_i, y_i, z_i) más alejados del centro de sistema de coordenadas transformado, dándoles menos importancia.

2.6. Modificación propuesta

Aunque los métodos anteriores son computacionalmente simples (su solución conlleva resolver un sistema lineal de dimensiones pequeñas), la calidad de la aproximación no es muy buena, ya que la aproximación se hace a través de superficies polinomiales que no consiguen capturar las irregularidades u oscilaciones de la nube de puntos en el nodo. Debido a ésto, nosotros proponemos una técnica de ajuste que no sea computacionalmente muy cara y que produzca mejores aproximaciones. Además, el método propuesto presenta la opción de usar sólo los datos relevantes (los puntos más representativos de la nube de puntos) en su versión llamada dispersa (*sparse*).

La solución propuesta usa una función de error distinta que aproxima mejor los datos, la cual se basa en métodos bayesianos provenientes del área de *machine learning* conocidos como *Gaussian Process*, los cuales se explicarán en las secciones siguientes. Empezaremos revisando la teoría de aprendizaje bayesiano, específicamente, los procesos Gaussianos para regresión.

2.7. Procesos Gaussianos

Definición 1 *Una sucesión de variables aleatorias* $\{f(x_i)\}_{i\in I}$ *se dice que es un proceso Gaussiano si dado cualquier conjunto finito* $x_1, ..., x_m$*, el vector* $\boldsymbol{f} = \{f(x_1), ..., f(x_m)\}$ *tiene distribución normal* $N(m_f, K)$ *con media:*

$$m_f = (m_0(x_1), ..., m_0(x_m)), \tag{2.9}$$

donde $m_0(x_i) = E(f(x_i))$*, y matriz de covarianza* $K = (k_{i,j})$ *con componentes:*

$$k_{i,j} = cov(f(x_i), f(x_j)) = k(x_i, x_j) = E[(f(x_i) - m(f(x_i)))(f(x_j) - m(f(x_j)))] \tag{2.10}$$

Es decir,

$$p_0(\boldsymbol{f}) = \frac{1}{\sqrt{(2\pi)^m |K|}} exp\left\{ -\frac{1}{2}(\boldsymbol{f} - m_f)^T K^{-1} (\boldsymbol{f} - m_f) \right\} \tag{2.11}$$

Se usará p_0 como la prior en el proceso bayesiano. Dado un conjunto de entrenamiento D denotaremos por f_D el vector $\{f_{x_1}, f_{x_2}, ..., f_{x_n}\}$, donde $\{x_1, x_2, ..., x_n\} = \mathbf{X}$

2.8. Procesos Gaussianos para regresión

Los métodos bayesianos tienen ventajas sobre otros métodos por su tratamiento probabilístico del problema, porque permiten incorporar información a priori sobre el problema tratado. Una ventaja inmediata es que se puede estimar la incertidumbre sobre un resultado predicho.

Para aplicar el aprendizaje bayesiano, asumiremos un modelo probabilístico de los datos. Sean $x_i \in \mathbb{R}^m$ las entradas, $y_i \in \mathbb{R}^d$ las salidas, y asúmase que tenemos un conjunto de N pares de entradas-salidas: $D = \{(x_1, y_1), ..., (x_N, y_N)\} = \{\mathbf{X}, \mathbf{y}\}$, que se relacionan de la siguiente manera:

$$y = f(x) + \epsilon, \tag{2.12}$$

donde ϵ es un error que se distribuye de forma Gaussiana con media 0 y varianza σ^2. En adelante usaremos indistintamente $f(x)$ y f_x. El objetivo es determinar la función y a partir del conjunto de datos D usando un enfoque bayesiano. En nuestro

problema se va a asumir que la función $f(x)$ proviene de un *proceso Gaussiano*, lo cual le dará flexibilidad y tratabilidad al problema.

El uso de procesos Gaussianos en regresión consta de dos partes: entrenamiento, en la que se utilizan los datos conocidos para ajustar los parámetros internos, y predicción, en la que se calcula el valor de la función f en un dato nuevo que no pertenece al conjunto de entrenamiento. Sea D el conjunto de observaciones usadas en la etapa de entrenamiento:

$$D = \{(x_i, y_i) : x_i \in \mathbb{R}^n, y_i \in \mathbb{R}, i = 1, ..., n\} \tag{2.13}$$

Para determinar el valor de la función f en un nuevo dato x, se requiere calcular la distribución a posteriori de f_x:

$$p_{post}(f_x) = p(f_x \mid D) \tag{2.14}$$

La distribución a posteriori de f_x dado el conjunto de entrenamiento D se expresa como sigue:

$$p_{post}(f_x) = \frac{p(D \mid f_x)p_0(f_x)}{p(D)} \tag{2.15}$$

Esto se puede reescribir de la siguiente manera:

$$p_{post}(f_x) = \frac{\int p(\mathbf{y} \mid f_x)p_0(f_x, f_D)df_D}{p(D)} \tag{2.16}$$

Se puede obtener analíticamente una fórmula si $p(\mathbf{y} \mid f_x)$ es gaussiana. En este caso, obtenemos que $p_{post}(f_x)$ es también gaussiana (ver Rammusen y William [24] y Csato [7]):

$$f_x|D \sim \mathcal{N}(m_{post}(x), \sigma_{post}(x)^2), \tag{2.17}$$

$$m_{post}(x) = \boldsymbol{k}^T(x)(\boldsymbol{K} + \sigma^2 I)^{-1}\boldsymbol{y} \tag{2.18}$$

$$\sigma^2_{post}(x) = k(x, x) - \boldsymbol{k}^T(x)(\boldsymbol{K} + \sigma^2 I)^{-1}\boldsymbol{k}(x) \tag{2.19}$$

donde:

- $\boldsymbol{k}(x) = (k(x, x_1), ..., k(x, x_n))^T$

- $\boldsymbol{K}$ es la matriz de covarianzas del proceso Gaussiano evaluada en D
- $\boldsymbol{y}$ es el vector $(y_1, ..., y_n)$
- σ^2 es la varianza de ϵ

La media m_{post} es el valor de más alta probabilidad en la distribución a posteriori (es la moda), por lo que es un buen estimador de la superficie, por lo tanto, la superficie aproximante s_i se tomará como m_{post} en la ecuación 2.4. La función que determina la isosuperficie de nivel 0 queda de la siguiente manera para el caso $\mathbb{R}^2 \rightarrow \mathbb{R}$ (2.20) y $\mathbb{R}^3 \rightarrow \mathbb{R}$ (2.21) (ver figura 2.3):

$$f(x, y, z) = z - m_{post}(x, y) \tag{2.20}$$

$$f(x, y, z) = m_{post}(x, y, z) \tag{2.21}$$

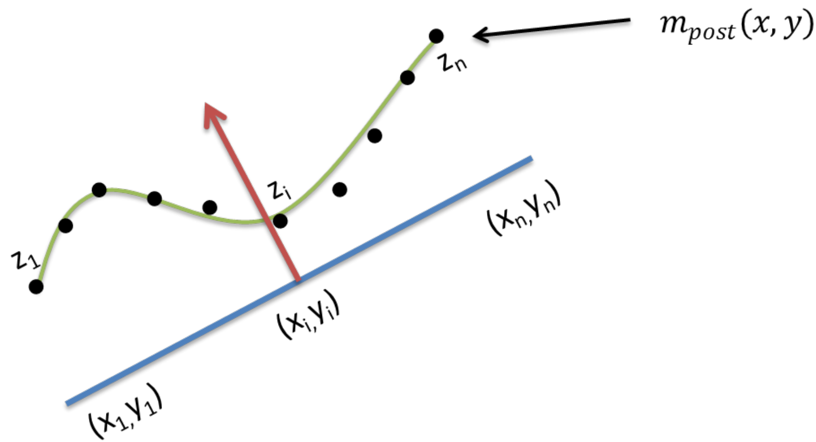


Figura 2.3: Ejemplo de función de $\mathbb{R}^2$ en $\mathbb{R}$, donde los pares (x_i, y_i) se toman como entrada y z_i como salida del proceso Gaussiano. Se utiliza la media m_{post} como una aproximación de la superficie.

La varianza $\sigma^2_{post}(\cdot)$ nos da una idea de la precisión de la aproximación realizada a través de la media.

Tanto la media como la varianza se pueden calcular de forma online y sin tener que usar todos los datos (dispersa), es decir, usando sólo los datos más significativos. En las siguientes secciones explicaremos cómo se realiza este proceso.

2.9. Regresión Online

La idea para calcular la expresión de la media y varianza (2.18), (2.19) se basa en el siguiente lema de parametrización, que calcula los momentos (media y varianza) de la posterior como combinación lineal y bilineal de los kernels en los datos. Este resultado fue propuesto por Csato en [7], [9] y a continuación es presentado:

Lema 1 (Csato) *: Dado un conjunto de entrenamiento* $D = \{\boldsymbol{x}, \boldsymbol{y}\} = \{(\boldsymbol{x}_n, y_n) \mid n = 1, ..., N\}$*, una verosimilitud arbitraria* $p(D \mid f_D)$ *y una prior con media* $m_0(\boldsymbol{x})$ *y función de covarianza* $k(\boldsymbol{x}, \boldsymbol{x}')$ *entonces el resultado de la actualización bayesiana (2.15) es un proceso con funciones de media y covarianza dadas por:*

$$m_{post}(\boldsymbol{x}) = m_0(\boldsymbol{x}) + \sum_{i=1}^{N} k(\boldsymbol{x}, \boldsymbol{x}_i) q(i) \tag{2.22}$$

$$k_{post}(\boldsymbol{x}, \boldsymbol{x}') = k(\boldsymbol{x}, \boldsymbol{x}') + \sum_{i,j=1}^{N} k(\boldsymbol{x}, \boldsymbol{x}_i) R(ij) k(\boldsymbol{x}_i, \boldsymbol{x}'). \tag{2.23}$$

Los parámetros q(i) y R(i,j) están dados por:

$$\begin{aligned} q(i) &= \frac{1}{Z} \int p_0(\boldsymbol{f}) \frac{\partial P(D \mid \boldsymbol{f}_D)}{\partial f(\boldsymbol{x}_i)} d\boldsymbol{f} \\ &= \frac{\partial}{\partial m_0(x_i)} ln \int p_0(\boldsymbol{f}_D) P(D \mid \boldsymbol{f}_D) d\boldsymbol{f}_D \end{aligned} \tag{2.24}$$

$$\begin{aligned} R(ij) &= \frac{1}{Z} \int p_0(\boldsymbol{f}) \frac{\partial^2 P(D \mid \boldsymbol{f}_D)}{\partial f(\boldsymbol{x}_i) \partial f(\boldsymbol{x}_j)} d\boldsymbol{f} - q(i) q(j) \\ &= \frac{\partial^2}{\partial m_0(x_i) \partial m_0(x_i)} ln \int p_0(\boldsymbol{f}_D) P(D \mid \boldsymbol{f}_D) d\boldsymbol{f}_D \end{aligned} \tag{2.25}$$

donde $\boldsymbol{f}_D = [f(\boldsymbol{x}_1), ..., f(\boldsymbol{x}_N)]^T$ *y* $Z = \int p_0(\boldsymbol{f}_D) P(D \mid \boldsymbol{f}_D) d\boldsymbol{f}_D$ *es una constante de normalización y las derivadas parciales son con respecto a la media a priori en* $\boldsymbol{x_i}$*.*

La demostración de este lema está fuera del alcance de este trabajo (ver [7]). Para obtener una aproximación online de las fórmulas (2.22) y (2.23), asumiremos que los datos son condicionalmente independientes, por lo tanto la verosimilitud se factoriza de la siguiente manera:

$$P(D \mid \boldsymbol{f}_D) = \prod_{n=1}^{N} P(y_n \mid f_n, \boldsymbol{x}_n) \tag{2.26}$$

El proceso Gaussiano online procesa el conjunto de entrenamiento una observación a la vez. En cada paso t, la aproximación Gaussiana obtenida después de la observación $(\mathbf{x}_t, y_t)$ es denotada por $\hat{p}_t$. La posterior p_{t+1} en el paso $t+1$ es calculada como (ver [8, 23]):

$$p_{t+1}(\boldsymbol{f}) = \frac{P(y_{t+1} \mid \boldsymbol{f})\hat{p}_t(\boldsymbol{f})}{E_t(P(y_{t+1} \mid \boldsymbol{f}_D))} \tag{2.27}$$

Esto se visualiza en la siguiente figura.

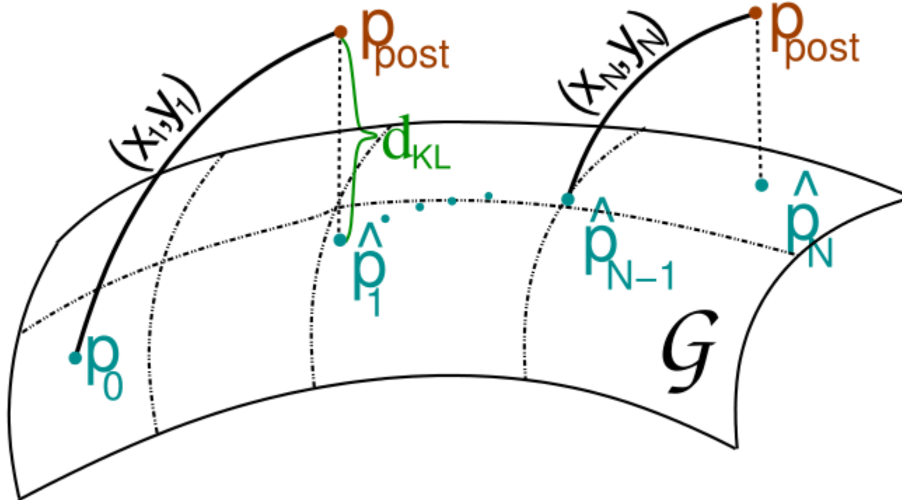


Figura 2.4: Visualización de la aproximación online del proceso posterior, donde $p_{post} = p_N$ en el paso N. El proceso aproximado resultante de la iteración anterior es usado como prior para el siguiente. Imagen extraída de [7].

Puesto que esta posterior p_{t+1} ya no es gaussiana, se repite el mecanismo de aproximarla por la gaussiana más cercana, $\hat{p}_{t+1}$ (ver figura 2.4). La forma en que se

determina esta aproximación $\hat{p}_{t+1}$ es minimizando la divergencia de Kullback-Leibler d_{KL} conocida como *KL-divergence*:

$$d_{KL}(P\|Q) = \int_{-\infty}^{\infty} p(x) \ln\left(\frac{p(x)}{q(x)}\right) dx \tag{2.28}$$

Para calcular las aproximaciones *online* de la media y la función de covarianza k_t aplicamos el lema 1 secuencialmente con un único término de verosimilitud $P(y_t \mid f_t, \boldsymbol{x}_t)$ en cada paso. Procediendo recursivamente, llegamos a:

$$m_{t+1}(\boldsymbol{x}) = m_t(\boldsymbol{x}) + q^{(t+1)} k_t(\boldsymbol{x}, \boldsymbol{x}_{t+1}) \tag{2.29}$$

$$k_{t+1}(\boldsymbol{x}, \boldsymbol{x}') = k_t(\boldsymbol{x}, \boldsymbol{x}') + r^{(t+1)} k_t(\boldsymbol{x}, \boldsymbol{x}_{t+1}) k_t(\boldsymbol{x}_{t+1}, \boldsymbol{x}') \tag{2.30}$$

Donde los escalares $q^{(t+1)}$ y $r^{(t+1)}$ siguen directamente de aplicar el lema 1 con una única verosimilitud $P(y_{t+1}|f_{\boldsymbol{x}_{t+1}})$ en el paso t+1:

$$q^{(t+1)} = \frac{\partial}{\partial m_t(x_{t+1})} ln E_t(P(y_{t+1} \mid f_{t+1})) \tag{2.31}$$

$$r^{(t+1)} = \frac{\partial^2}{\partial m_t(x_{t+1})} ln E_t(P(y_{t+1} \mid f_{t+1})) \tag{2.32}$$

Donde el tiempo t se refiere al orden en el que los términos de verosimilitud individuales son incluidos en la aproximación. Los promedios en (2.32) son determinados con respecto al proceso Gaussiano en el tiempo t y las derivadas tomadas con respecto a $m_t(\boldsymbol{x}_{t+1})$. Nótese que estos promedios sólo requieren una integración uni-dimensional sobre el proceso en la entrada $\boldsymbol{x}_{t+1}$.

Desarrollando la recursividad de las ecuaciones (2.29), (2.30) llegamos a una fórmula cerrada de la posterior aproximada de GP en el tiempo t como una función del kernel inicial y las verosimilitudes:

$$m_t(\boldsymbol{x}) = \sum_{i=1}^{t} k(\boldsymbol{x}, \boldsymbol{x}_i) \alpha_t(i) = \boldsymbol{\alpha}_t^T \boldsymbol{k}_{\boldsymbol{x}} \tag{2.33}$$

$$k(\boldsymbol{x}, \boldsymbol{x}') = k(\boldsymbol{x}, \boldsymbol{x}') + \sum_{i,j=1}^{t} k(\boldsymbol{x}, \boldsymbol{x}_i) C_t(ij) k(\boldsymbol{x}_j, \boldsymbol{x}') = k(\boldsymbol{x}, \boldsymbol{x}') + \boldsymbol{k}_{\boldsymbol{x}}^T \boldsymbol{C}_t \boldsymbol{k}_{\boldsymbol{x}}, \tag{2.34}$$

con coeficientes $\alpha_t(i)$ y $C_t(ij)$ definiendo las aproximaciones a $\boldsymbol{q}$ y $\boldsymbol{R}$ de las ecuaciones (2.24), (2.25). Los coeficientes dados por el lema de parametrización 1 y las ecuaciones (2.33), (2.34) son equivalentes sólo en el caso de regresión gaussiana. Usando estas últimas dos ecuaciones, $\alpha_t(i)$ y $C_t(ij)$ se calculan de la siguiente manera:

$$\begin{aligned} s_{t+1} &= \begin{bmatrix} C_t k_{t+1} \\ 0 \end{bmatrix} + \begin{bmatrix} \mathbf{0} \\ 1 \end{bmatrix} \\ \alpha_{t+1} &= \begin{bmatrix} \alpha_t \\ 0 \end{bmatrix} + q_{t+1} s_{t+1} \\ C_{t+1} &= \begin{bmatrix} C_t & \mathbf{0} \\ \mathbf{0} & 0 \end{bmatrix} + r_{t+1} s_{t+1} s_{t+1}^T \end{aligned} \tag{2.35}$$

Vamos a probar la ecuación (2.33) por inducción: usando la hipótesis de inducción $\boldsymbol{\alpha}_0 = \boldsymbol{C}_0 = 0$ para el tiempo $t = 1$, tenemos $\alpha_1(1) = q^{(1)}$ y $C_1(1,1) = r^{(1)}$. La función media en el tiempo $t = 1$ es $m_1(\boldsymbol{x}) = \alpha_1(1) K_0(\boldsymbol{x}_1, \boldsymbol{x})$ (del lema 1 para un único punto de datos, ecuación (2.29)). Similarmente, el kernel modificado es $K_1(\boldsymbol{x}, \boldsymbol{x}') = K_0(\boldsymbol{x}, \boldsymbol{x}') + K(\boldsymbol{x}, \boldsymbol{x}_1) C_1(1,1) K_0(\boldsymbol{x}_1, \boldsymbol{x}')$ con $\boldsymbol{\alpha}$ y $\boldsymbol{C}$ independientes de $\boldsymbol{x}$ y $\boldsymbol{x}'$, probando la hipótesis de inducción.

Asumimos que en un tiempo t tenemos los parámetros $\boldsymbol{\alpha}_t$ y $\boldsymbol{C}_t$ independientes de los puntos $\boldsymbol{x}$ y $\boldsymbol{x}'$, y que tenemos las funciones kernel y media expresadas en las ecuaciones (2.34) y (2.33). La actualización para la media puede ser escrita de la siguiente forma (usando (2.29) y (2.30)):

$$\begin{aligned} m_{t+1}(\boldsymbol{x}) &= \sum_{i=1}^{t} k(\boldsymbol{x}_i, \boldsymbol{x}) \alpha_t(i) + q^{(t+1)} \sum_{i,j=1}^{t} k(\boldsymbol{x}_i, \boldsymbol{x}) C_t(i,j) k(\boldsymbol{x}_j, \boldsymbol{x}_{t+1}) + \\ &\qquad q^{(t+1)} k(\boldsymbol{x}, \boldsymbol{x}_{t+1}) \\ &= \sum_{i=i}^{t} k(\boldsymbol{x}, \boldsymbol{x}_i) [\alpha_t(i) + q^{(t+1)} \sum_{j=1}^{t} C_t(i,j) k(\boldsymbol{x}_j, \boldsymbol{x}_{t+1})] + q^{(t+1)} k(\boldsymbol{x}, \boldsymbol{x}_{t+1}) \\ &= \sum_{i=1}^{t+1} k(\boldsymbol{x}, \boldsymbol{x}_i) \alpha_{t+1}(i), \end{aligned} \tag{2.36}$$

donde $\boldsymbol{\alpha_{t+1}}(i)$ no involucra a la entrada $\boldsymbol{x}$, y su substitución se obtiene usando (2.35). La demostración para los kernels $k_{t+1}(\boldsymbol{x}, \boldsymbol{x}')$ se realiza de manera similar. El siguiente paso es calcular el logaritmo de la verosimilitud promedio:

$$\ln E_t(P(y_{t+1} \mid f_{\boldsymbol{x}_{t+1}})) = -\frac{1}{2}\ln[2\pi(\sigma_0^2 + \sigma_{t+1}^2)] - \frac{(y_{t+1} - m_{t+1})^2}{2(\sigma_0^2 + \sigma_{t+1}^2)} \tag{2.37}$$

Derivando con respecto a m_{t+1} una vez y dos veces obtenemos los escalares:

$$q^{(t+1)} = \frac{y_{t+1} - m_{t+1}}{2(\sigma_0^2 + \sigma_{t+1}^2)} \tag{2.38}$$

$$r^{(t+1)} = -\frac{1}{2(\sigma_0^2 + \sigma_{t+1}^2)} \tag{2.39}$$

2.10. Regresión Dispersa

La limitación principal del GP en *batch* y *online* es que ambos requieren mantener en memoria todo el conjunto de entrenamiento. Esto es una limitación fuerte para casi todos los problemas prácticos. Para resolver este problema, un GP disperso fue propuesto por Csato [10], conocido como *Sparse Online Gaussian Process* (SOGP). En el algoritmo SOGP, existe un parámetro de capacidad que determina el máximo número de observaciones relevantes a mantener en memoria durante el proceso de entrenamiento. El conjunto de observaciones relevantes es llamado el conjunto de vectores base (BV). Dado un nuevo dato de entrenamiento $(\boldsymbol{x}, y)$ en el paso de aprendizaje $t + 1$, el vector $\boldsymbol{x}$ se descompone como la suma de sus proyecciones en el espacio generado por el conjunto BV más el vector residual correspondiente v_{res}, el cual es ortogonal al espacio generado por el conjunto BV. El valor $\gamma = ||v_{res}||$ mide qué tan cerca se encuentra el vector de entrada $\boldsymbol{x}$ al espacio generado por el conjunto BV. Si γ es menor que una tolerancia dada, entonces el punto nuevo se considera redundante y $\boldsymbol{x}$ es proyectado en el espacio generado por el conjunto BV. En este caso, la proyección de $\boldsymbol{x}$ es usada para calcular los valores de α_{t+1} y C_{t+1}, sin incrementar sus tamaños y sin cambiar el conjunto BV. En caso contrario, el vector $\boldsymbol{x}$ es incluido en el conjunto BV, y como resultado α_{t+1} y C_{t+1} aumentan su tamaño. Después de agregar un nuevo vector al conjunto BV, SOGP verifica si el tamaño

del conjunto BV ha sobrepasado su capacidad. De ser así, el vector que contribuya menos a la representación del GP es removido del conjunto BV, lo que implica un re-cálculo de α_{t+1} y C_{t+1}.

Lo anterior se resume en el siguiente algoritmo: Empezamos con un conjunto vacío y valores 0 para los parámetros $\boldsymbol{\alpha}$ y $\boldsymbol{C}$, fijamos el tamaño máximo del conjunto BV a d y el kernel a priori a k. Un parámetro de tolerancia ϵ_{tol} se usa para prevenir que la matriz de Gram $k(x_i, x_j)$ se vuelva singular.

- Calculamos $q^{(t+1)}$ y $r^{(t+1)}$ (los coeficientes de actualización para el dato nuevo) usando las ecuaciones (2.31), (2.32); los kernels k^*_{t+1} y $\boldsymbol{k}_{t+1}$ (2.40); las coordenadas de proyección $\hat{e}_{t+1}$ (2.41) y la longitud del residual γ_{t+1} (2.42).

- Si $\gamma_{t+1} < \epsilon_{tol}$, entonces realizamos la actualización dispersa usando la ecuación (2.43) sin extender el tamaño del conjunto de parámetros ($\boldsymbol{\alpha}$ y $\boldsymbol{C}$).

- En caso contrario, se realiza la actualización mostrada en la ecuación (2.35). Se agrega el punto nuevo al conjunto BV y se calcula la inversa de la matriz extendida de Gram (2.45). Si el tamaño del conjunto BV es mayor a d, se calculan los *scores* ϵ_i para cada vector $\mathbf{x}_i \in BV$ (2.46), se encuentra el vector con el puntaje mínimo, y se elimina del conjunto BV usando las ecuaciones (2.48).

En las siguientes ecuaciones, $\boldsymbol{K}_t$ es la matriz del kernel, y $k(\boldsymbol{x}_i, \boldsymbol{x}_j) = \langle \Phi(\mathbf{x}_i), \Phi(\mathbf{x}_j) \rangle$, donde $\Phi(x_i)$ es la proyección del vector $\mathbf{x}_i$ en un espacio de características asociado con el kernel o función de covarianza k [28].

$$\begin{aligned} \boldsymbol{k}_{t+1} = [k(\boldsymbol{x}_1, \boldsymbol{x}_{t+1}), ..., k(\boldsymbol{x}_t, \boldsymbol{x}_{t+1})]^T \\ k^* = k(\boldsymbol{x}_{t+1}, \boldsymbol{x}_{t+1}) \end{aligned} \tag{2.40}$$

$$\hat{\boldsymbol{e}}_{t+1} = \boldsymbol{K}_t^{-1} \boldsymbol{k}_{t+1} \tag{2.41}$$

El residual puede calcularse de la siguiente forma:

$$\gamma_{t+1} = k^* - \boldsymbol{k}_{t+1}^T \hat{\boldsymbol{e}}_{t+1} \tag{2.42}$$

La actualización dispersa se realiza de la siguiente manera:

$$\begin{aligned}\hat{\boldsymbol{\alpha}}_{t+1} &= \boldsymbol{\alpha}_t + q^{(t+1)}\eta_{t+1}\hat{\boldsymbol{s}}_{t+1} \\ \hat{\boldsymbol{C}}_{t+1} &= \boldsymbol{C}_t + r^{(t+1)}\eta_{t+1}\hat{\boldsymbol{s}}_{t+1}\hat{\boldsymbol{s}}_{t+1}^T \\ \hat{\boldsymbol{s}}_{t+1} &= \boldsymbol{C}_t\boldsymbol{k}_{t+1} + \hat{\boldsymbol{e}}_{t+1}\end{aligned} \tag{2.43}$$

Donde η_{t+1} se calcula como sigue:

$$\eta_{t+1} = \left(1 + \gamma_{t+1} r^{(t+1)}\right)^{-1} \tag{2.44}$$

La matriz $\boldsymbol{Q}$ es la inversa de la matriz de Gram:

$$\boldsymbol{Q}_{t+1} = \boldsymbol{K}_{t+1}^{-1} \tag{2.45}$$

En la siguiente ecuación, $q(i)$, $c(i)$ y $s(i)$ son los elementos de la diagonal de $\boldsymbol{Q}$, $\boldsymbol{C}$ y $\boldsymbol{S}$ respectivamente. La matriz $\boldsymbol{S}$ viene de (2.47).

$$\epsilon_{t+1}(i) = \frac{\alpha^2(i)}{q(i)+c(i)} - \frac{s(i)}{q(i)} + \ln\left(1 + \frac{c(i)}{q(i)}\right) \tag{2.46}$$

$$\boldsymbol{S}_{t+1} = (\boldsymbol{C}_{t+1}^{-1} + \boldsymbol{K}_{t+1})^{-1} \tag{2.47}$$

La reducción de BV se realiza de la siguiente forma: Asúmase que el conjunto BV tiene $t+1$ elementos, y se desea eliminar uno de ellos. Asúmase, por simplicidad, que el elemento a eliminar se encuentra en la última posición $(t+1)$, como se muestra en la figura 2.5.

$$\begin{aligned}\hat{\boldsymbol{\alpha}}_{t+1} &= \boldsymbol{\alpha}^{(r)} - \frac{\alpha^*}{c^*+q^*}(\boldsymbol{Q}^* + \boldsymbol{C}^*) \\ \hat{\boldsymbol{C}}_{t+1} &= \boldsymbol{C}^{(r)} + \frac{\boldsymbol{Q}^*\boldsymbol{Q}^{*T}}{q^*} - \frac{(\boldsymbol{Q}^* + \boldsymbol{C}^*)(\boldsymbol{Q}^* + \boldsymbol{C}^*)^T}{q^*+c^*} \\ \hat{\boldsymbol{Q}}_{t+1} &= \boldsymbol{Q}^{(r)} - \frac{\boldsymbol{Q}^*\boldsymbol{Q}^{*T}}{q^*}\end{aligned} \tag{2.48}$$

Para más detalles sobre la deducción de estas fórmulas, ver [7].

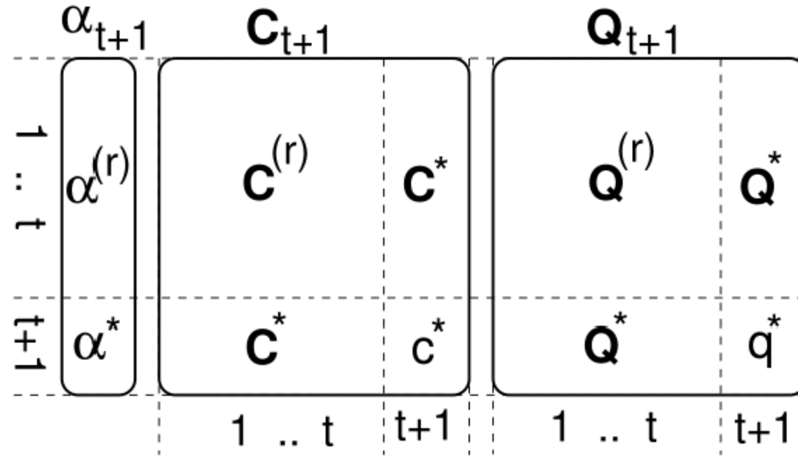


Figura 2.5: Agrupamiento de los parámetros de SOGP para la reducción del conjunto BV. Imagen extraída de [7].

2.11. Kernel

La función de covarianza k mencionada en (2.10) puede reemplazarse por cualquier función kernel. A continuación daremos la definición de una función kernel.

Definición 2 *Una función $k : X \times X \to \mathbb{R}^+$ se denomina kernel si para todo conjunto finito $\{u_1, u_2, ..., u_n\} \subset X$ la matriz de kernels K con componentes $K_{i,j} = k(u_i, u_j)$ es simétrica y semidefinida positiva:*

2.12. Triangulación de la superficie implícita

Una vez que se determinaron las funciones f_i (y por lo tanto la función F (2.1)), se pretende representar el conjunto de nivel 0 de esta función a través de una superficie triangular por partes. Para triangular la superficie de nivel, un algoritmo que ha resultado altamente efectivo es *Marching Cubes* [17]. Este algoritmo tiene varias características importantes, como:

- Respeta la geometría y topología de la función implícita F
    - Número de componentes conexas

- Número de agujeros (*holes*)
- Número de nudos

Una versión que ha resultado altamente efectiva es el algoritmo propuesto por Lewiner et. al. [16]. Se utiliza dicho algoritmo como una caja negra: se parte de una implementación existente, la cual no se modifica.

Capítulo 3

Desarrollo

3.1. Kernels

Los procesos Gaussianos requieren del uso de un kernel, para lo cual se han considerado las opciones que se muestran en el cuadro 3.1. La calidad de la reconstrucción depende directamente del tipo de kernel y de sus parámetros.

Se eligió usar el kernel Thin Plate, debido a que sus resultados son excelentes y a que su parámetro R se determina fácilmente como la mayor distancia entre los puntos de entrada, como se menciona en [31]. El motivo para esta elección se describe con más detalle en la sección 3.2.2.

Cuadro 3.1: Kernels

Nombre	Parámetros	Kernel
Radial Basis Function	A, b, $\mathbf{w}$	$Ae^{\frac{-1}{2d}\sum_{i=1}^{d}\mathbf{w}_i(\mathbf{x}_i-\mathbf{y}_i)^2}+b$
RBF simple	w	$e^{\frac{-1}{2d}w\Vert\mathbf{x}-\mathbf{y}\Vert^2}$
Polinomial	α, c, d	$(\alpha\mathbf{x}^T\mathbf{y}+c)^d$
Thin Plate	R	$2\Vert\mathbf{x}-\mathbf{y}\Vert^3-3R\Vert\mathbf{x}-\mathbf{y}\Vert^2+R^3$

3.2. Optimización de parámetros

El proceso Gaussiano depende de un conjunto de parámetros θ. Un primer parámetro es s_0^2 que representa la varianza del ruido de los datos. Adicionalmente, el kernel utilizado depende otros parámetros de cuyos valores depende la calidad de la regresión. Una forma determinar θ es buscando el valor de θ^* que haga que el conjunto de entrenamiento tenga la mayor probabilidad de haber ocurrido. Esto puede hacerse maximizando la verosimilitud $L(\theta)$ ($\theta^* = argmax(L(\theta))$), donde:

$$L(\theta) = P(\mathcal{D} \mid \theta) = \frac{1}{|K_y|(2\pi)^n} exp\left(-\frac{\mathbf{Y}^t\mathbf{K_y}^{-1}\mathbf{Y}}{2}\right); \tag{3.1}$$

y $\mathbf{K_y} = \mathbf{K} + \sigma\mathbf{I}$. La log-verosimilitud está dada por:

$$\mathcal{L}(\theta) = \log(L(\theta)) = -\frac{1}{2}\log|\mathbf{K_y}| - \frac{1}{2}\mathbf{Y^T}\mathbf{K_y^{-1}}\mathbf{Y} - \frac{N}{2}\log 2\pi \tag{3.2}$$

Las derivadas parciales de $\mathcal{L}$ con respecto a los parámetros θ_i se expresan mediante la ecuación:

$$\frac{\partial\mathcal{L}}{\partial\theta_i} = -\frac{1}{2}tr\left(\mathbf{K_y}^{-1}\frac{\partial\mathbf{K_y}}{\partial\theta_i}\right) + \frac{1}{2}\mathbf{Y^T}\mathbf{K_y^{-1}}\frac{\partial\mathbf{K_y}}{\partial\theta_i}\mathbf{K_y^{-1}}\mathbf{Y}; \tag{3.3}$$

Esa ecuación puede optimizarse usando cualquier algoritmo de optimización.

Para encontrar un mínimo de la función $-\mathcal{L}(\theta)$, se utiliza el algoritmo del gradiente conjugado escalado [20], el cual requiere de lo siguiente:

- Un vector p_0 que contiene una aproximación inicial.
- Una función de error $f : \mathbb{R}^d \to \mathbb{R}$
- El gradiente del error $\nabla f : \mathbb{R}^d \to \mathbb{R}^d$

La salida es un vector p que contiene el mínimo encontrado de la función f. Adicionalmente, se tienen las siguientes variables que controlan la terminación del algoritmo y que pueden ajustarse:

- El número máximo de iteraciones i_{max}
- Un límite inferior x_{min} para el cambio en el vector p

- Un límite inferior y_{min} para el cambio en el error $f(p)$

Los criterios de terminación son los siguientes:

- El número de iteraciones supera i_{max}
- El gradiente es menor a un valor ϵ (determinado como la distancia entre 1 y el menor valor mayor a 1 que es representable)
- El cambio en p es menor a x_{min} y el cambio en $f(p)$ es menor a y_{min}

Generalmente, los parámetros del kernel pertenecen a $\mathbb{R}^{d+}$, lo cual conlleva a un problema de optimización con restricciones. Sin embargo, este caso puede llevarse a un problema de optimización sin restricciones calculando el logaritmo de los parámetros, haciendo las siguientes transformaciones:

- Parámetros: Se obtiene el vector $\mathbf{L} = (\mathbf{L}_i)$ que cumple $\mathbf{L}_i = \log(\mathbf{p}_i)$
- Error: Primero se recupera el vector $\mathbf{p} = (\mathbf{p}_i)$ de la forma $\mathbf{p}_i = e^{\mathbf{L}_i}$, y luego se evalúa la función de error $f(\mathbf{p})$
- Gradiente: La transformación del gradiente se obtiene al derivar $f(\mathbf{p})$ con respecto a $\mathbf{L}$, usando la siguiente expresión general:

  $$\frac{\partial f(e^{\mathbf{u}_1}, ..., e^{\mathbf{u}_d})}{\partial \mathbf{u}} = \nabla f(e^{\mathbf{u}_1}, ..., e^{\mathbf{u}_d}) \circ (e^{\mathbf{u}_1}, ..., e^{\mathbf{u}_d}) \tag{3.4}$$

  Primero se recupera el vector $\mathbf{p}$ de la forma $\mathbf{p}_i = e^{\mathbf{L}_i}$, y luego se evalúa el gradiente $\nabla f(\mathbf{p}) \circ \mathbf{p}$
- Finalmente, se recuperan el vector $\mathbf{p}$ de parámetros en el espacio original: $\mathbf{p}_i = e^{\mathbf{L}_i}$

Nota: El símbolo $\circ$ denota al producto de Schur (ver A.1).

3.2.1. Gradiente del Kernel

Los gradientes de cada kernel con respecto cada uno de sus parámetros se muestran a continuación:

- Kernel RBF:

$$\begin{aligned} \frac{\partial f}{\partial A} &= 1 \\ \frac{\partial f}{\partial b} &= 0 \\ \frac{\partial f}{\partial \mathbf{w}_i} &= \frac{-1}{2d}(\mathbf{x}_i - \mathbf{y}_i)^2 A e^{\frac{-1}{2d}\sum\limits_{i=1}^{d}(\mathbf{x}-\mathbf{y})^2\mathbf{w}_i} \end{aligned} \tag{3.5}$$

- Kernel RBF simple:

$$\frac{\partial f}{\partial w} = \frac{-1}{2d}\left\|\mathbf{x}-\mathbf{y}\right\|^2 e^{\frac{-1}{2d}w\|\mathbf{x}-\mathbf{y}\|^2} \tag{3.6}$$

- Kernel Polinomial:

$$\begin{aligned} \frac{\partial f}{\partial a} &= d(\alpha\mathbf{x}^T\mathbf{y}+c)^{d-1}(\mathbf{x}^T\mathbf{y}) \\ \frac{\partial f}{\partial c} &= d(\alpha\mathbf{x}^T\mathbf{y}+c)^{d-1} \\ \frac{\partial f}{\partial d} &= (\alpha\mathbf{x}^T\mathbf{y}+c)^{d}\ln(\alpha\mathbf{x}^T\mathbf{y}+c) \end{aligned} \tag{3.7}$$

- Kernel Thin Plate:

$$\frac{\partial f}{\partial R} = -3\left\|\mathbf{x}-\mathbf{y}\right\|^2 + 3R^2 \tag{3.8}$$

3.2.2. Estructura de log-verosimilitud

La función de log-verosimilitud 3.2, en base a pruebas realizadas con diversas funciones, tiene generalmente un único mínimo, por lo que al optimizarla se puede obtener el mínimo global. Sin embargo, en ocasiones puede tener varios mínimos locales. Se observó que es muy difícil generar una función con varios mínimos locales, por lo que no se llevaron a cabo medidas para manejar este caso.

En la figura 3.2 veremos la estructura que tienen la funciones $-\mathcal{L}(\theta)$ (la cual será considerada como el error) y $\frac{\partial \mathcal{L}}{\partial \theta_i}$ cuando los datos $\mathcal{D}$ son obtenidos muestreando la función Goldstein-Price (figura 3.1). En la matriz de kernel $\mathbf{K}$ se ha usado el kernel RBF simple mencionado en el cuadro 3.1.

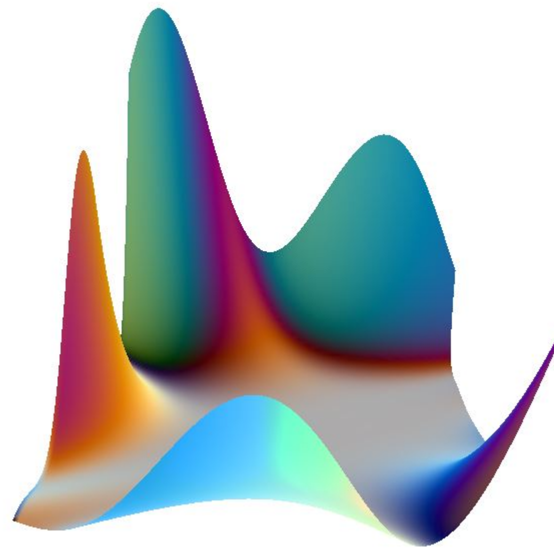


Figura 3.1: Gráfico de la función Goldstein-Price

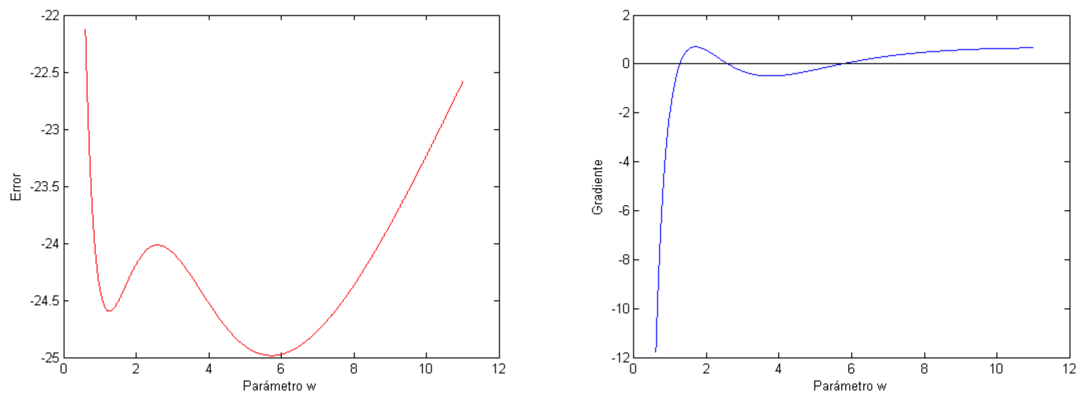


Figura 3.2: Izquierda: Error de regresión en función del parámetro w. Derecha: Gradiente del error.

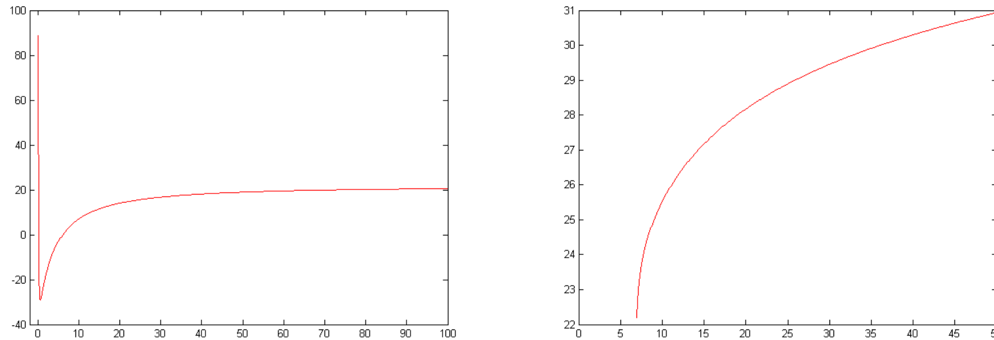


Figura 3.3: Izquierda: Error usando el kernel RBF. Derecha: Error usando el kernel ThinPlate

En la figura 3.2 se puede ver cómo varía el error con el parámetro w. Puede verse que la función de error no es convexa, y puede tener varios mínimos locales.

Usualmente, el error usando un kernel específico tiene un mismo tipo de gráfico. En la figura 3.3 se ve que el error obtenido con el kernel RBF inicia con un valor muy alto, luego baja abruptamente y vuelve a subir, volviéndose casi horizontal. Esto tiene el efecto de que al aplicarle un algoritmo de optimización, si se inicia con un punto a la izquierda del óptimo, la pendiente lo mueve fuertemente hacia la derecha, donde la pendiente es casi 0, causando que el algoritmo se detenga. El kernel polinomial tuvo un comportamiento equivalente al RBF. El error obtenido usando el kernel Thin Plate empieza generando NaN hasta un cierto valor, desde el cual empieza a subir monótonamente. Esto nos indica que conviene tomar el menor valor posible que no genere NaN. Este límite es provisto en [31], donde se fija como la mayor distancia posible entre los puntos de entrada.

Adicionalmente, se observó que los kernels RBF y polinomial son altamente sensibles, ya que la calidad de las reconstrucciones empeora rápidamente cuando los parámetros se alejan del óptimo. En cambio, el kernel Thin Plate demostró tener una alta robustez, mostrando muy pequeñas variaciones ante grandes cambios en su parámetro.

3.3. Algoritmo MPU

Se utiliza un octree para dividir los puntos. El algoritmo MPU funciona de manera recursiva, como se indica en el algoritmo 1.

Algoritmo 1 MPU

Entrada: Un conjunto de puntos **P**
Salida: El valor pesado w de cada punto en **P**
1: **Para** cada punto **p** en **P** :
2: Inicia el recorrido del octree en la raíz
3: **Si p** cae dentro del soporte compacto del nodo **entonces:**
4: **Si** no se ha realizado el aprendizaje en el nodo **entonces:**
5: Se realiza el aprendizaje en el nodo actual
6: **Si** se cumplen las condiciones de división **entonces:**
7: El nodo se divide
8: **Si** el nodo actual es una hoja **entonces:**
9: Se realiza la predicción para **p**
10: Se calcula el peso de **p**
11: Se agrega el valor predecido y el peso a w, siguiendo la fórmula (2.1)
12: **Si no**
13: Continúa el recorrido en todos los hijos (paso 3)

Es fácil notar que la fórmula (2.1) sólo se evalúa en los nodos que contienen al punto a evaluar en su soporte compacto.

En este algoritmo se realiza el aprendizaje "en demanda", según se van recorriendo los nodos. También es posible realizar primero el aprendizaje de todo el octree antes de evaluar los puntos, lo que resulta en un algoritmo dividido en dos partes: construcción del octree (aprendizaje) y predicción (ver algoritmo 2). La ventaja es que haciendo este cambio el algoritmo se vuelve paralelizable: el aprendizaje en cada nodo y la predicción en cada punto puede calcularse independientemente (bajo la condición de que el aprendizaje se realice antes que la predicción).

3.4. Condiciones de división

Un nodo se divide si se cumple alguna de las siguientes condiciones:

Algoritmo 2 MPU paralelizable

Entrada: Un conjunto de puntos $\mathbf{P}$
Salida: El valor pesado w de cada punto en $\mathbf{P}$
1: Inicia el recorrido del octree en la raíz
2: Se realiza el aprendizaje en el nodo actual
3: **Si** se cumplen las condiciones de división **entonces:**
4: El nodo se divide
5: Continúa el recorrido en todos los hijos (paso 2)
6: **Para** cada punto $\mathbf{p}$ en $\mathbf{P}$ **:**
7: Inicia el recorrido del octree en la raíz
8: **Si** $\mathbf{p}$ cae dentro del soporte compacto del nodo **entonces:**
9: **Si** el nodo actual es una hoja **entonces:**
10: Se realiza la predicción para $\mathbf{p}$
11: Se calcula el peso de $\mathbf{p}$
12: Se agrega el valor predecido y el peso a w, siguiendo la fórmula 2.1
13: **Si no**
14: Continúa el recorrido en todos los hijos (paso 8)

- El error de predicción es mayor al máximo permitido
- La cantidad de puntos en el nodo es mayor al máximo permitido

Sin embargo, aunque se cumplan las condiciones anteriores, un nodo no puede dividirse si se cumple alguna de las siguientes condiciones:

- La profundidad de los hijos excedería el máximo permitido
- La cantidad de puntos en el nodo actual es menor al mínimo permitido

3.5. Proceso de aprendizaje

El aprendizaje en un nodo se realiza con el algoritmo 3:

El primer paso en el proceso de aprendizaje es determinar el conjunto de puntos que caen dentro del soporte compacto del nodo (ver figura 2.1). Se toma una esfera que contiene al nodo con un radio inicial, y si no contiene suficientes puntos para realizar el aprendizaje, se incrementa el radio hasta que la cantidad de puntos sea suficiente. El radio inicial, la cantidad mínima de puntos y el aumento al radio en cada paso son parámetros ajustables.

Algoritmo 3 Aprendizaje

Entrada: Un conjunto de puntos D, un radio inicial r_0, la cantidad mínima de puntos p_{min}, el factor de aumento λ

Salida: Una función de predicción

1: Consigue el conjunto P de puntos en D que caen dentro del soporte compacto del nodo usando el radio $r = r_0$
2: **Mientras** $|P| < p_{min}$ **:**
3: Aumenta el radio: $r = r + \lambda r_0$
4: Consigue el conjunto P de puntos en D que caen dentro del soporte compacto del nodo con radio r
5: Calcula la función de aproximación con los puntos P
6: Calcula la el error de la función de aproximación

Se tienen varios métodos para realizar el aprendizaje en cada nodo, dependiendo de si se tienen las normales, y de la dimensión usada para el proceso Gaussiano. En las siguientes secciones, consideraremos a $\mathbf{P}$ como el conjunto de puntos que caen dentro del soporte compacto del nodo, y $\mathbf{N}$ como las normales de dichos puntos.

3.5.1. Análisis de componentes principales

El objetivo de este método es buscar un plano base para poder obtener una aproximación de $\mathbb{R}^2$ en $\mathbb{R}$ como se explicó en la sección 2.4.1. En esta sección daremos más detalles. Este método puede realizarse sin conocimiento de las normales. Primero se calcula un vector $\mathbf{n}$ que se considerará como normal a un plano aproximado por los puntos en P de la siguiente manera:

Se calcula la media $\mathbf{m}$ de $\mathbf{P}$ y la matriz de covarianza $C = cov(P)$, y se obtienen los eigenvalores λ y la matriz con los eigenvectores $\mathbf{V}$ de C. Se considera al eigenvector asociado al eigenvalor más pequeño como la normal $\mathbf{n}$ buscada (ver figura 3.4).

Posteriormente, se obtiene un nuevo conjunto de puntos $\mathbf{P}'$ que estén en el plano determinado por $\mathbf{n}$ usando la transformación afín $\mathbf{p}'_i = \mathbf{V}(\mathbf{p}_i - \mathbf{m})$ para todos los puntos $\mathbf{p}_i \in \mathbf{P}$, y se realiza el aprendizaje Gaussiano con los puntos en $\mathbf{P}'$, tomando como entrada las coordenadas (x,y) y como salida la coordenada (z). Al realizar la predicción, se realiza la misma transformación con los puntos cuyo valor se quiere obtener.

Cuando se obtienen los planos sin conocer las normales, es posible que en dos

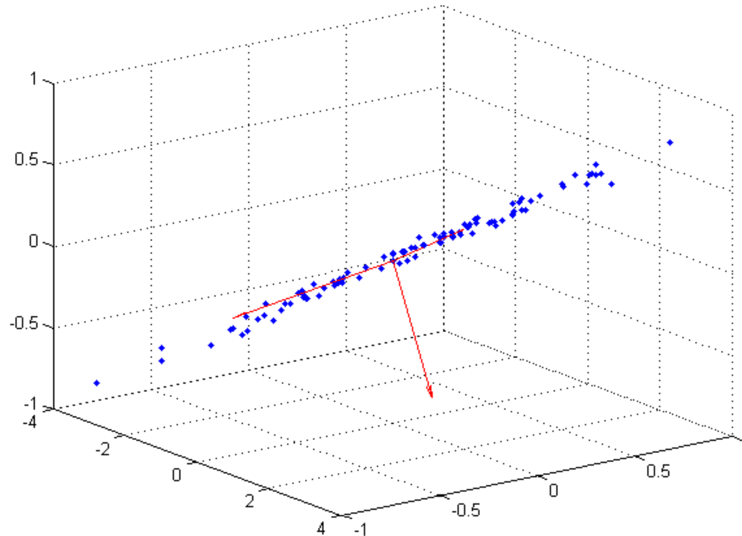


Figura 3.4: Nube de puntos $\mathbf{P}$ con sus eigenvectores. El eigenvector que apunta hacia abajo se considera como $\mathbf{n}$.

nodos adyacentes se obtengan orientaciones opuestas. Más formalmente, si los nodos i y j son adyacentes y sus vectores correspondientes $\mathbf{n}_i$, $\mathbf{n}_j$ cumplen que $\mathbf{n}_i \cdot \mathbf{n}_j < 0$ entonces consideraremos que sus orientaciones son opuestas. Es posible invertir la orientación del plano de un nodo multiplicando la matriz $\mathbf{V}$ por -1, lo que puede hacerse en las siguientes condiciones:

1. Si se conocen las normales, puede invertirse la orientación del plano si $\mathbf{n} \cdot \overline{\mathbf{N}} < 0$, donde $\overline{\mathbf{N}}$ es la media de las normales

2. Si no se conocen las normales, entonces puede invertirse la orientación del plano si la cantidad de nodos adyacentes con orientaciones opuestas es mayoría.

En nuestra implementación, se ha usado la primera condición. Para la segunda, hace falta determinar cuáles nodos consideraremos como adyacentes. Podemos considerar a dos nodos como adyacentes usando una variedad de condiciones:

1. Tienen el mismo padre

2. Se tocan en una cara

3. Se tocan en una arista

4. Se tocan en un vértice

3.5.2. Desplazamiento de normales

Este proceso de aprendizaje encuentra mediante regresión una función $\mathbb{R}^3 \to \mathbb{R}$ (2.1) que aproxima en cada uno de los nodos el valor correspondiente de la salida, por lo tanto, si se determina el conjunto de nivel 0 de esta función, se obtendrá una superficie que pasa cerca de los puntos, la cual es nuestra superficie objetivo.

Este método requiere del conocimiento de las normales. Se utilizan los puntos extendidos (x,y,z,w), donde w indica la *altura* del punto con respecto a la superficie:

- Un valor positivo indica que el punto se encuentra en el exterior de la superficie
- Un valor negativo indica que el punto se encuentra en el interior de la superficie
- Un valor 0 indica que el punto se encuentra en la superficie

Las coordenadas (x,y,z) se pasan como las entradas en el aprendizaje del proceso Gaussiano, y los valores de w se pasan como las salidas. Se toman los siguientes tres conjuntos de puntos, como se muestra en la figura 3.5:

- Los puntos muestreados con altura 0
- Los puntos muestreados sumados con sus normales escaladas por un factor h, con altura h
- Los puntos muestreados menos sus normales escaladas por un factor h, con altura $-h$

El factor h es un parámetro ajustable.

3.6. Cálculo del error

Cuando se obtiene una aproximación en un nodo, es necesario obtener una medida de qué tan cercana es comparada con la superficie "real". Esto implica calcular una

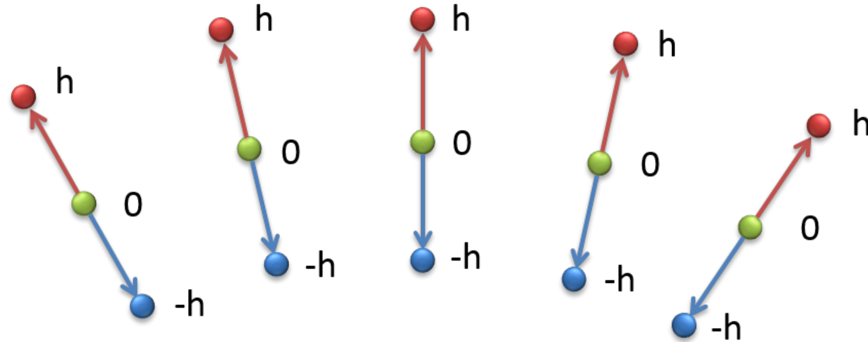


Figura 3.5: Desplazamiento de normales

medida del error, la cual puede obtenerse obtenerse evaluando la aproximación obtenida en los puntos de entrada. Esto nos da un conjunto de diferencias $(d_1, d_2, ..., d_n)$ entre el valor calculado y el "real", de las cuales puede obtenerse una sola medida de error usando técnicas como las siguientes:

- Método de Taubin [30]: $\sum_{i=1}^{n} \frac{|f(x_i)|}{\|\nabla f(x_i)\|}$ donde $f(x_i)$ es la evaluación de la superficie en el punto de entrada x_i
- Máximo: $\max_{i=1...n} |d_i|$
- Media aritmética: $\sum_{i=1}^{n} \frac{|d_i|}{n}$
- Media cuadrática: $\sqrt{\sum_{i=1}^{n} \frac{d_i^2}{n}}$
- Alguna otra medida

Se eligió la media cuadrática por los resultados obtenidos empíricamente.

3.7. Pesos

Cuando se tienen las superficies implícitas en todos los nodos, es necesario calcular un conjunto de "pesos" $\phi_i(x, y, z)$ para cada nodo i y punto (x, y, z) (ver ecuación 2.1). Estos pesos pueden calcularse con los siguientes factores:

- La varianza en el punto (x, y, z)
- La distancia del centro c_i del nodo al punto (x, y, z)
- El error de la aproximación en el punto (x, y, z)

El método elegido utiliza la distancia al centro y el radio r_i de la esfera usada como soporte compacto del nodo, usando un *spline* cuadrático [21] de la siguiente manera:

$$
\begin{aligned}
d =& \frac{3|(x,y,z) - c_i|_2}{2r} \\
b(d) =& \begin{cases} d^2 + 0.75 & \text{si } 0 \leq d \leq 0.5 \\ (1.5 - d)^2 & \text{si } 0.5 < d \leq 1.5 \\ 0 & \text{si } d > 1.5 \end{cases}
\end{aligned} \tag{3.9}
$$

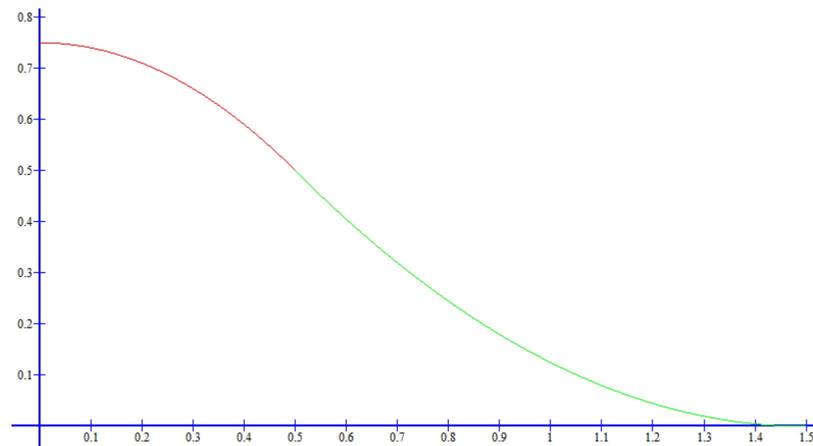


Figura 3.6: Spline cuadrático usado como función de peso (3.9)

Este *spline* se puede ver como una función radial con soporte en la esfera de centro c_i y radio r_i. Esta función decrece con la distancia al centro y vale 0 en la frontera de esta esfera. El comportamiento de este *spline* se puede ver en la figura 3.6 como función de d (3.9). El radio r_i es un parámetro ajustable.

3.8. Triangulación

Para realizar la triangulación de la superficie, se utiliza el algoritmo *Marching Cubes*, el cual requiere como entrada una discretización del espacio con los valores de la superficie implícita. Para obtener esos datos, se realiza una iteración para cada uno de los puntos de la discretización, y se calcula el valor en cada punto recorriendo el octree como se indica en el algoritmo 2. La cantidad de divisiones del espacio a usar en la discretización es un parámetro ajustable.

3.9. Evaluación de la superficie reconstruida

Las superficies obtenidas se evalúan de manera visual. Es posible obtener medidas cuantitativas de la calidad, pero una evaluación visual ha demostrado ser un método más sencillo y preciso: Al usar medidas cuantitativas se tiene que evaluar la proximidad de la superficie reconstruida a los puntos de entrada y viceversa. El primer paso es buscar una correspondencia entre estos dos conjuntos, lo cual es una tarea costosa. Además, generalmente se obtienen muchos más detalles en la superficie resultante que en la nube de puntos, por lo que no se puede realizar una medida de aproximación adecuada. Los casos de regiones faltantes y ruido también dificultan la determinación del error de aproximación.

En el área de reconstrucción de superficies es común evaluar las superficies de manera visual, puesto que en muchas aplicaciones de computación gráfica lo que importa es la apariencia visual de la superficie: dos superficies con diferente error de aproximación pudieran tener la misma calidad visual al ojo humano, por lo que desde el punto de vista de estas aplicaciones son equivalentes. Al comparar dos resultados visualmente, suele ser evidente cuál de ellos es mejor.

3.10. Análisis de complejidad

3.10.1. Proceso Gaussiano en Batch

Consiste en las ecuaciones 2.18 y 2.19, las cuales se volverán a presentar por conveniencia:

$$m_{post}(x) = \boldsymbol{k}^T(x)(\boldsymbol{K} + \sigma^2 I)^{-1}\boldsymbol{y}$$

$$\sigma_{post}(x)^2 = k(x, x) - \boldsymbol{k}^T(x)(\boldsymbol{K} + \sigma^2 I)^{-1}\boldsymbol{k}(x)$$

La complejidad de estas ecuaciones está dominada por el término $(\boldsymbol{K}+\sigma^2 I)^{-1}$, lo que lleva la complejidad a $O(n^3)$, siendo n la cantidad de datos usados como entrada para el aprendizaje. Sin embargo, este término puede calcularse una vez (durante el aprendizaje) y almacenarse para su uso en posteriores predicciones. Al calcular la media, puede guardarse el resultado de $C_y = (\boldsymbol{K} + \sigma^2 I)^{-1}\boldsymbol{y}$, lo que es un vector con n elementos, por lo que disminuye la cantidad de memoria requerida, y reduce la predicción a la siguiente ecuación:

$$m_{post}(x) = \boldsymbol{k}^T(x)C_y \tag{3.10}$$

La complejidad para el cálculo de 3.10 es simplemente $O(n)$. Si se requiere la varianza, entones sólo podemos almacenar el resultado de $C = (\boldsymbol{K} + \sigma^2 I)^{-1}$, el cual es una matriz de $n \times n$, lo que además aumenta el tiempo de predicción:

$$\sigma_{post}(x)^2 = k(x, x) - \boldsymbol{k}^T(x)C\boldsymbol{k}(x) \tag{3.11}$$

La complejidad para el cálculo de 3.11 es $O(n^2)$.

Si tomamos a d como la cantidad de de divisiones del espacio a usar en la discretización para el algoritmo *Marching Cubes*, entonces se realizan predicciones para d^3 puntos. En las pruebas realizadas, la cantidad de predicciones por punto era baja (alrededor de 5). Tomando a l como la cantidad de hojas del octree y a $\bar{n}$ como la cantidad promedio de puntos por nodo, entonces la complejidad del algoritmo es aproximadamente:

- $O(l\bar{n}^3 + d^3\bar{n})$ si sólo se usa la media

- $O(l\bar{n}^3 + d^3\bar{n}^2)$ si se usa la varianza

En un conjunto de puntos promedio, se generan cerca de $5K$ nodos y $10M$ evaluaciones. El tiempo de aprendizaje suele ser menor al tiempo de predicción.

3.10.2. Proceso Gaussiano Online Disperso

Sea k la capacidad elegida. De las ecuaciones (2.35) y (2.48) se puede ver que agregar un elemento durante el aprendizaje tiene complejidad $O(k^2)$, por lo que agregar n elementos resulta en $O(k^2 n)$ por nodo para el aprendizaje. La media se obtiene con la ecuación (2.33) en $O(k)$, y la varianza se obtiene de (2.34) en $O(k^2)$, por lo que la complejidad del algoritmo es aproximadamente:

- $O(lk^2\bar{n} + d^3k)$ si sólo se usa la media
- $O(lk^2\bar{n} + d^3k^2)$ si se usa la varianza

Se puede ver que depende fuertemente de la capacidad k elegida.

Capítulo 4

Resultados

A lo largo de las pruebas realizadas, el método propuesto demostró constantemente mejoras en comparación con el método MPU tomado como base.

4.1. Modelos con suficientes puntos

En el caso de nubes de puntos con alta densidad y sin regiones faltantes, los modelos eran reconstruidos adecuadamente por el método original, y también con el propuesto. A continuación se muestran ejemplos de conjuntos de datos tridimensionales de este tipo que son reconstruidos de manera equivalente por ambos métodos.

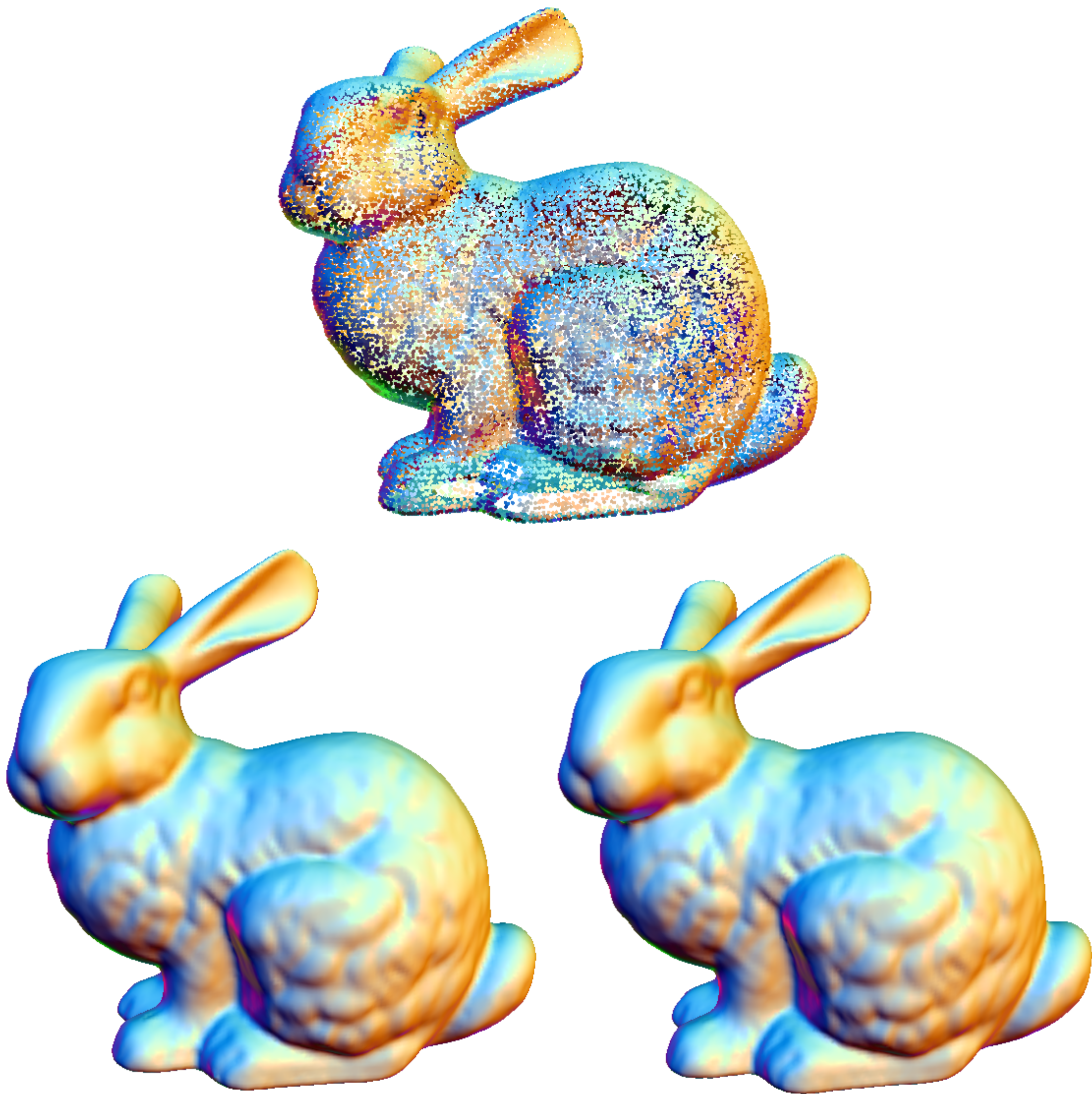


Figura 4.1: Modelo Stanford Bunny (35933 puntos). Arriba: conjunto original de puntos. Izquierda: reconstrucción usando método original. Derecha: reconstrucción usando método propuesto.

Figura 4.2: Modelo Armadillo (172974 puntos). Arriba: conjunto original de puntos. Izquierda: reconstrucción usando método original. Derecha: reconstrucción usando método propuesto.

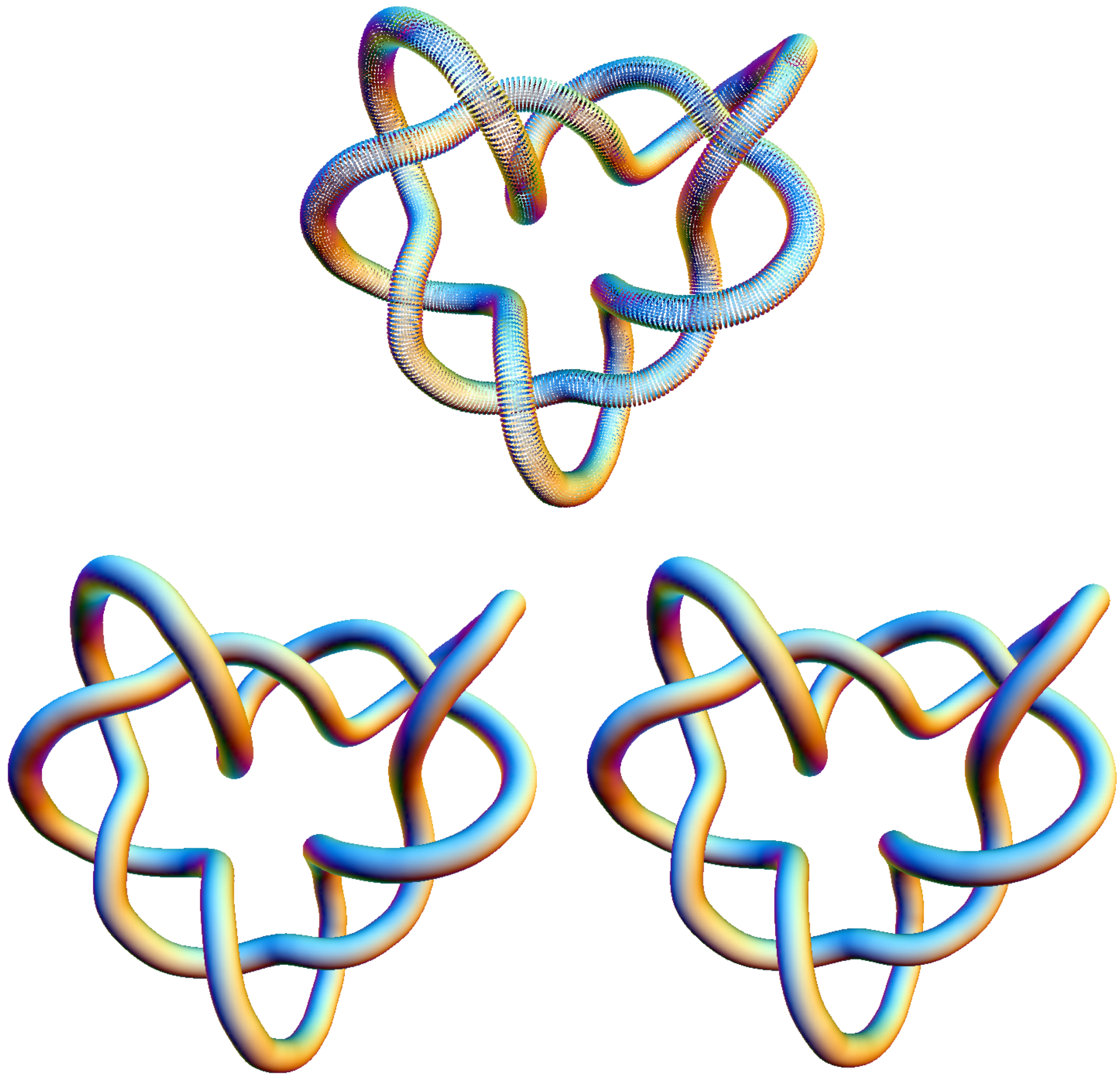


Figura 4.3: Modelo Knot (28659 puntos). Arriba: conjunto original de puntos. Izquierda: reconstrucción usando método original. Derecha: reconstrucción usando método propuesto.

4.2. Modelos con bordes

En el caso de nubes de puntos con bordes y esquinas, el método propuesto obtiene resultados mejores que el original.

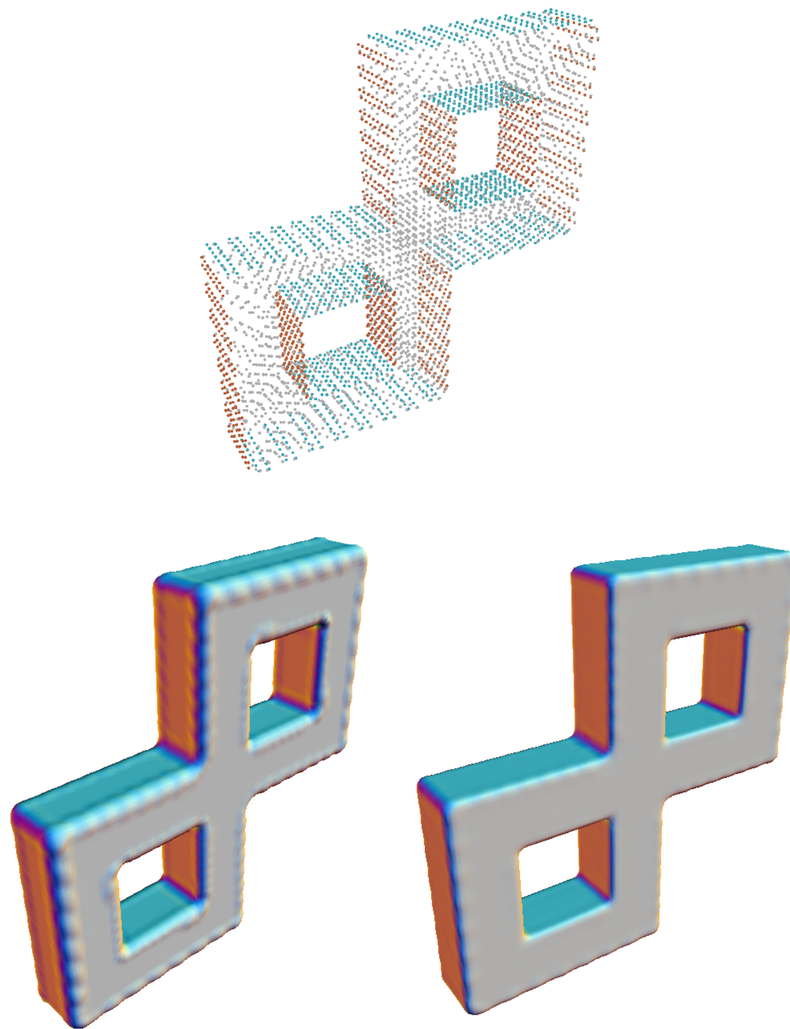


Figura 4.4: Modelo Double torus (4352 puntos). Arriba: conjunto original de puntos. Izquierda: reconstrucción usando método original. Derecha: reconstrucción usando método propuesto.

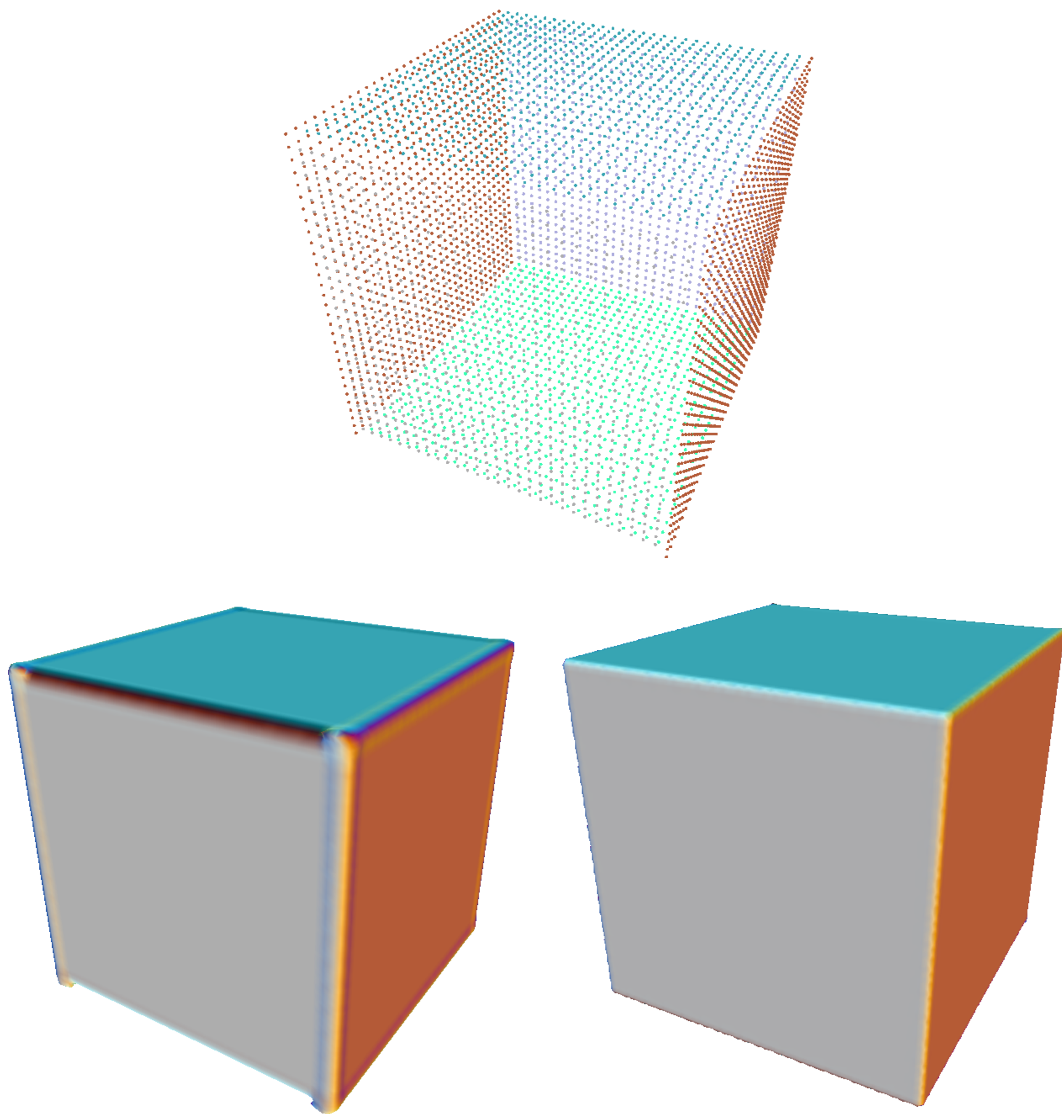


Figura 4.5: Modelo Cubo (5344 puntos). Arriba: conjunto original de puntos. Izquierda: reconstrucción usando método original. Derecha: reconstrucción usando método propuesto.

4.3. Modelos con regiones faltantes

En el caso de nubes de puntos con poca densidad de puntos o con zonas faltantes, el método propuesto obtiene resultados mejores que el original.

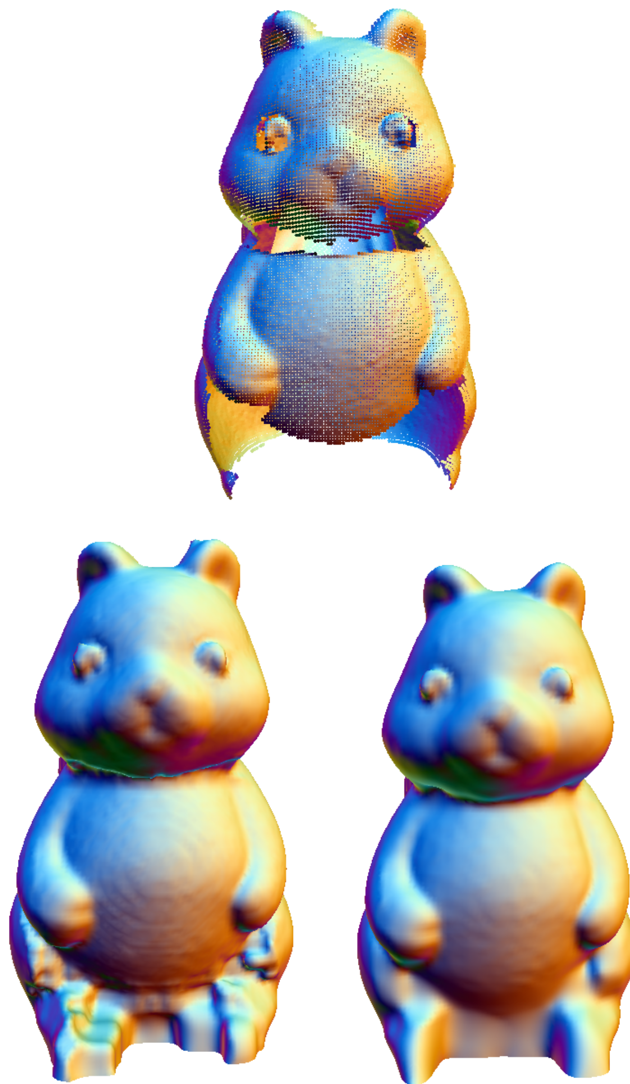


Figura 4.6: Modelo Squirrel (40627 puntos). Arriba: conjunto original de puntos. Izquierda: reconstrucción usando método original. Derecha: reconstrucción usando método propuesto.

4.4. Modelos con detalles finos

En los conjuntos de datos tridimensionales que se presentan a continuación, nuestro algoritmo fue capaz de reconstruir la superficie recuperando detalles finos y oscilaciones presentes en estos datos con mejor calidad que el algoritmo MPU clásico. Esto se puede observar claramente en la cara del dino (figura 4.8) y en las escamas del dragón (figura 4.9)

Figura 4.7: Modelo Dino (11642 puntos). Arriba: conjunto original de puntos. Medio: reconstrucción usando método original. Abajo: reconstrucción usando método propuesto.

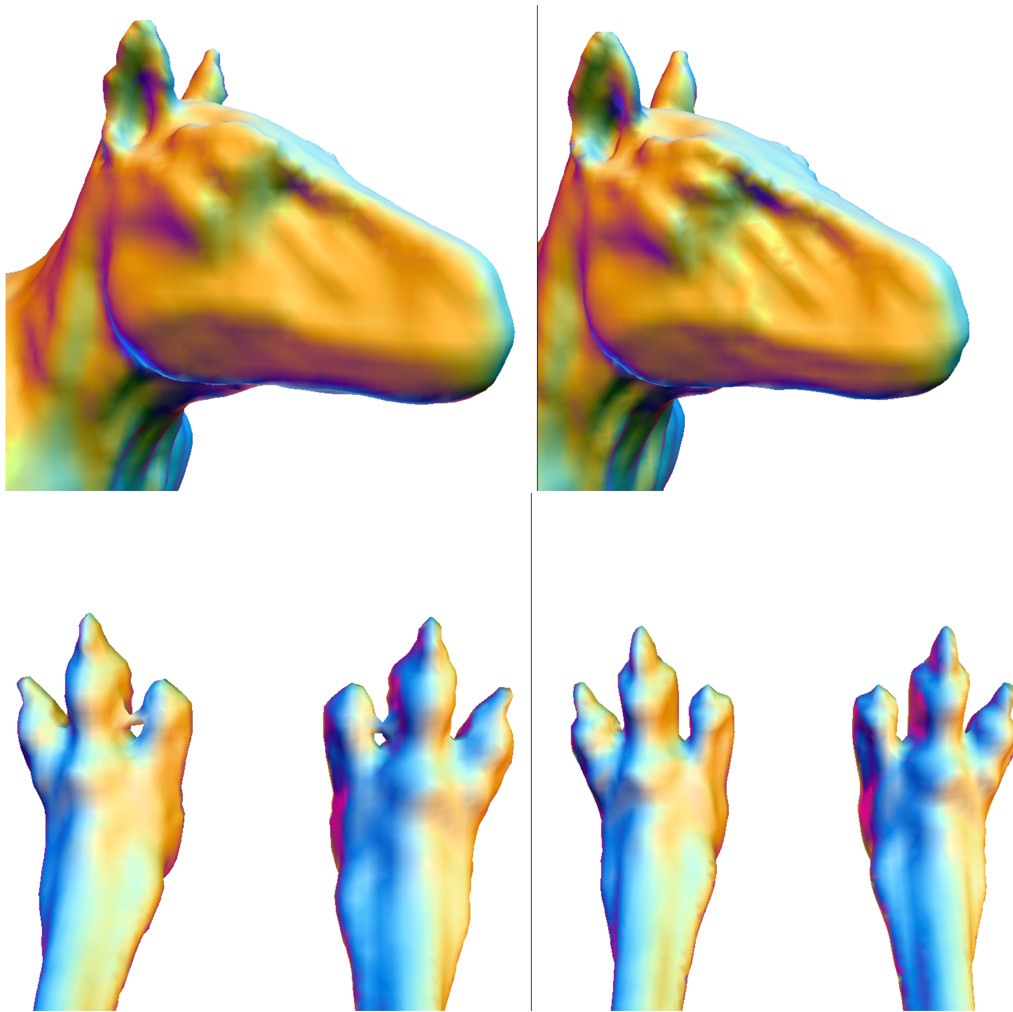


Figura 4.8: Acercamiento del modelo Dino. Izquierda: reconstrucción usando método original. Derecha: reconstrucción usando método propuesto.

Figura 4.9: Modelo Dragon (22998 puntos). Arriba: conjunto original de puntos. Medio: reconstrucción usando método original. Abajo: reconstrucción usando método propuesto.

4.5. Modelos con ruido

Se realizaron pruebas agregando ruido proveniente de una distribución normal a los puntos. Sin embargo, las normales no fueron alteradas. Bajo estas condiciones, se muestra en las figuras siguientes que el método propuesto es capaz de eliminar considerablemente el ruido. Nótese que éste no es un enfoque realista, pero es un primer paso para resolver el problema de reconstrucción bajo ruido donde no se tiene la información de las normales, el cual será abordado como un trabajo futuro.

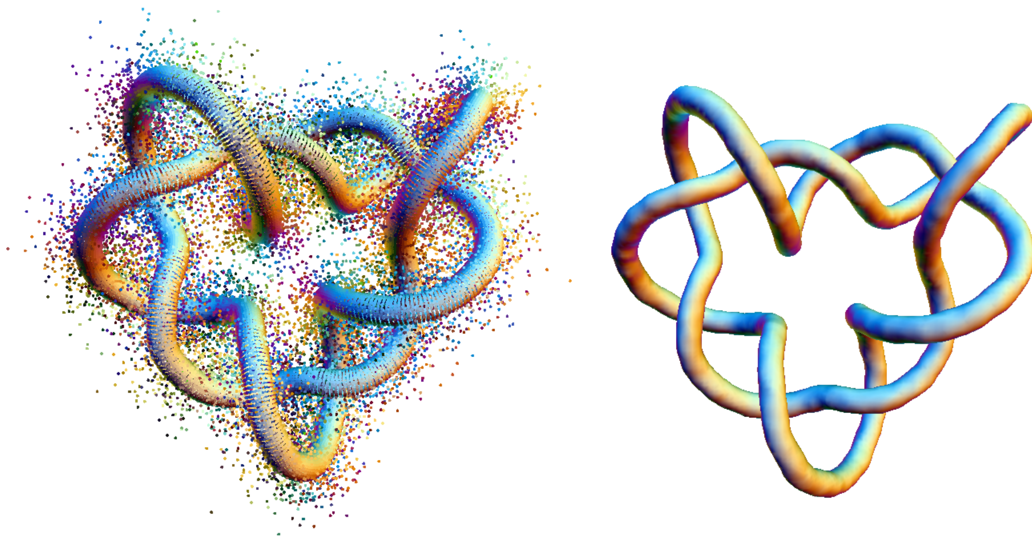


Figura 4.10: Izquierda: modelo Knot con ruido agregado. Derecha: reconstrucción usando método propuesto

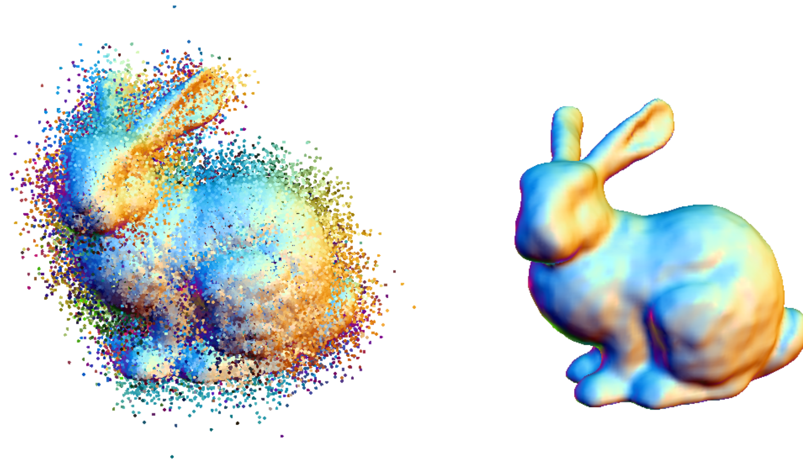


Figura 4.11: Izquierda: modelo Stanford Bunny con ruido agregado. Derecha: reconstrucción usando método propuesto

4.6. Modelos dispersos

Como modelos dispersos se entiende nubes de puntos con extremada escasez de datos. En este contexto nuestro algoritmo fue capaz de generar una superficie topológicamente correcta y con una geometría bastante próxima a la del modelo original.

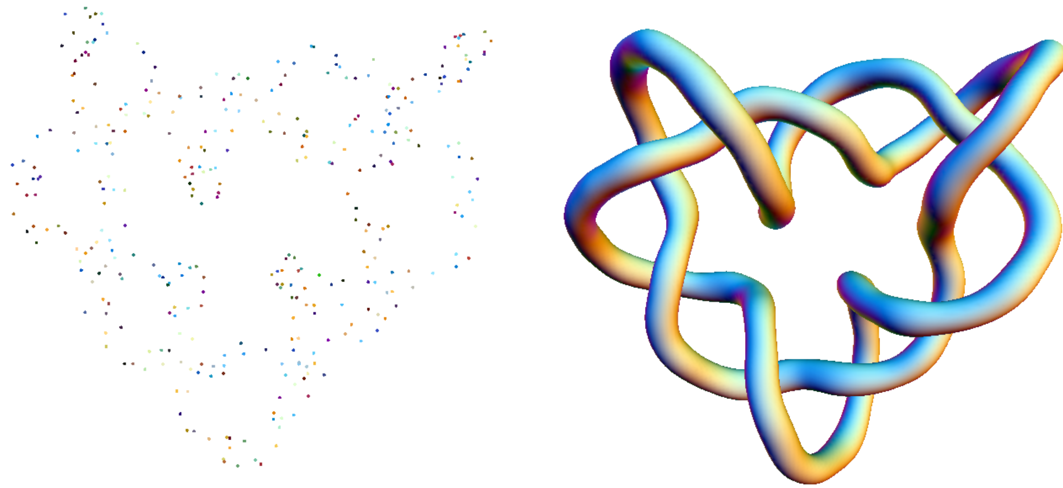


Figura 4.12: Izquierda: modelo Knot con 350 puntos. Derecha: reconstrucción usando método propuesto

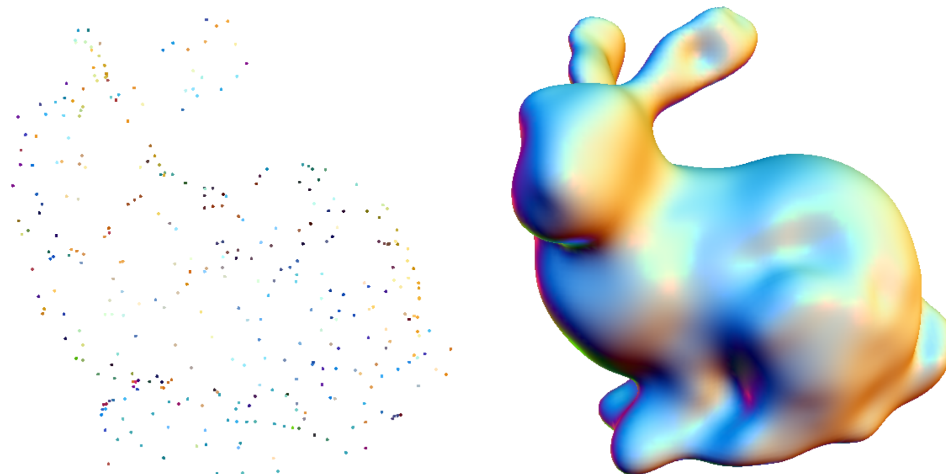


Figura 4.13: Izquierda: modelo Stanford Bunny con 350 puntos. Derecha: reconstrucción usando método propuesto

4.7. Proceso Gaussiano online disperso

En las secciones 2.9 y 2.10 se propone un algoritmo que es capaz de reconstruir superficies usando solamente los puntos más relevantes para la reconstrucción. Resultados de este algoritmo (denominado online disperso) se muestran en las figuras siguientes, donde el método es capaz de determinar con bastante precisión la topología y geometría de los modelos. Nótese que en modelos con datos faltantes como el “Squirrel”, los resultados son prácticamente idénticos a la técnica “batch” que usa todos los puntos.

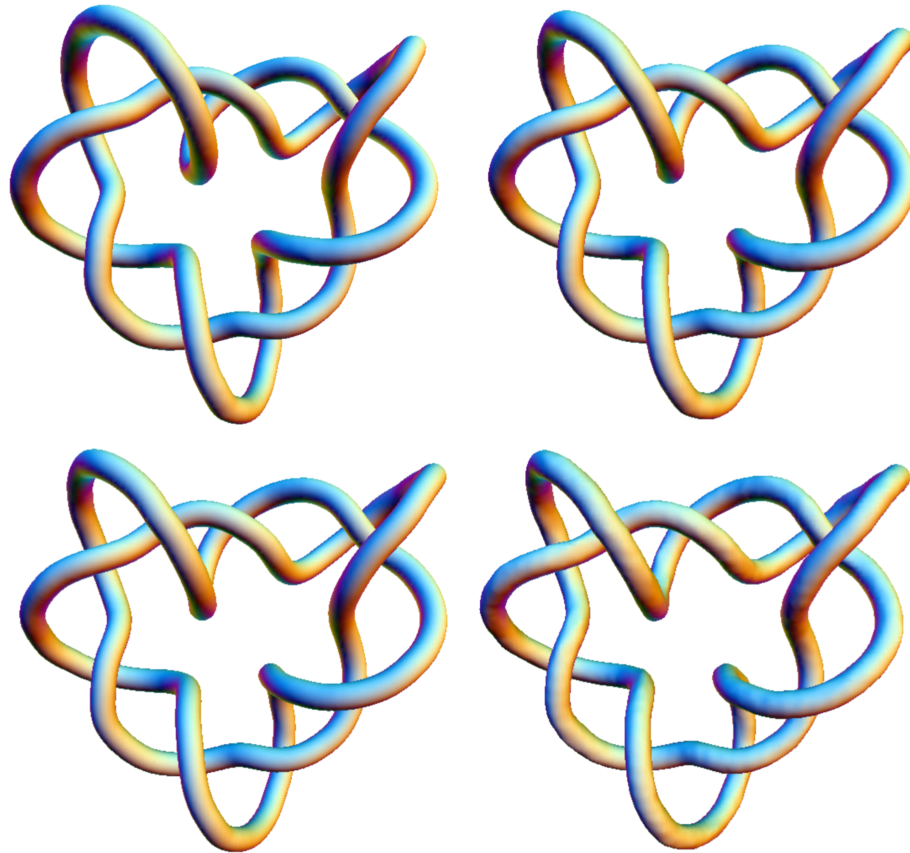


Figura 4.14: Reconstrucciones del modelo Knot (28659 puntos) obtenidas usando la versión online dispersa de GP, con 100 % (superior izquierda), 50 % (superior derecha), 25 % (inferior izquierda) y 12.5 % (inferior derecha) de los datos

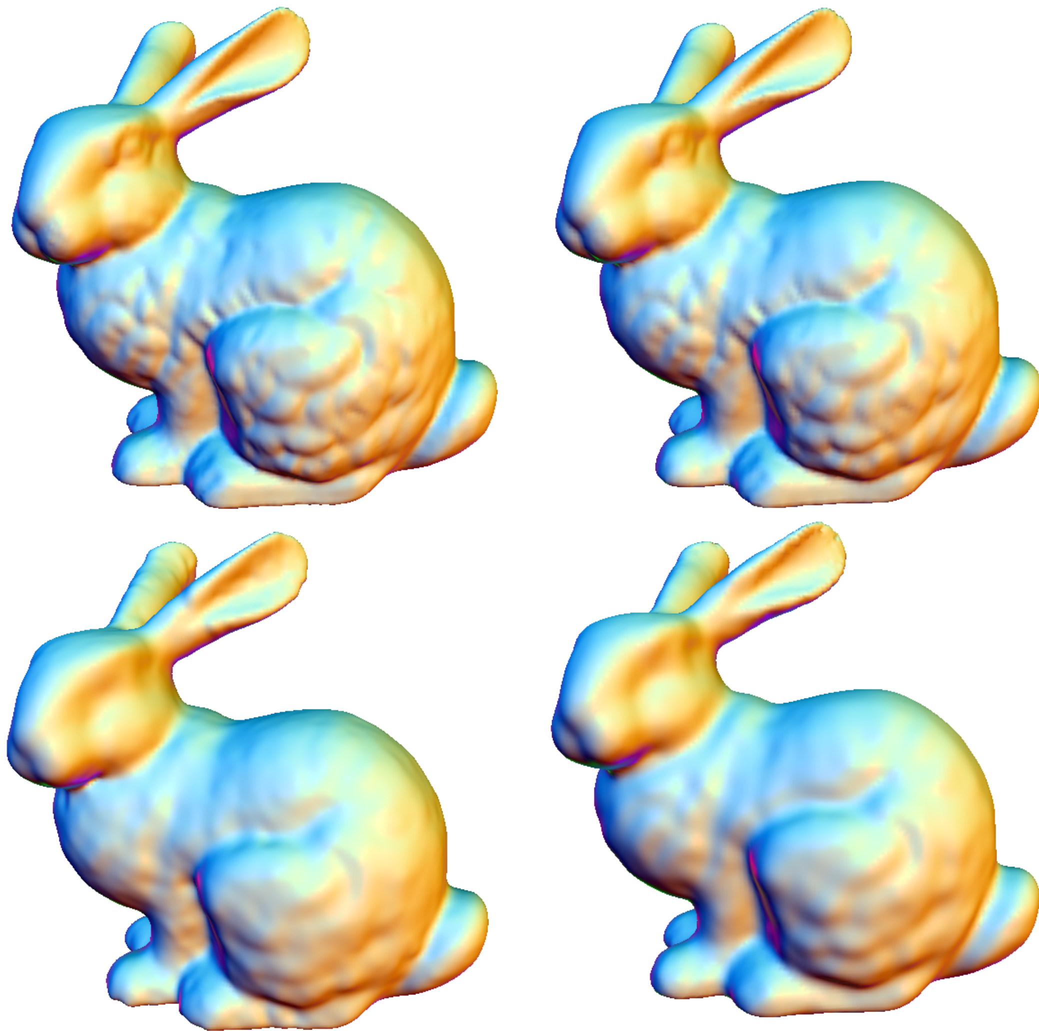


Figura 4.15: Reconstrucciones del modelo Stanford Bunny (35933 puntos) obtenidas usando la versión online dispersa de GP, con 100 % (superior izquierda), 50 % (superior derecha), 25 % (inferior izquierda) y 12.5 % (inferior derecha) de los datos

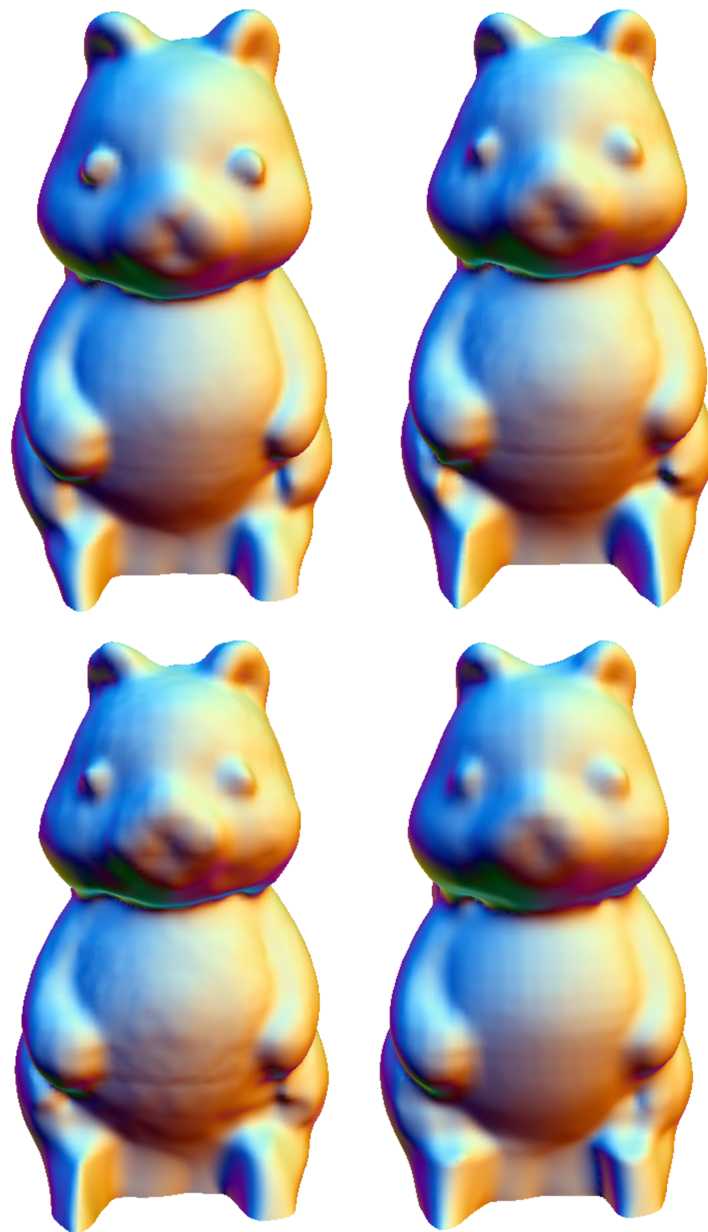


Figura 4.16: Reconstrucciones del modelo Squirrel (40627 puntos) obtenidas usando la versión online dispersa de GP, con 100 % (superior izquierda), 50 % (superior derecha), 25 % (inferior izquierda) y 12.5 % (inferior derecha) de los datos

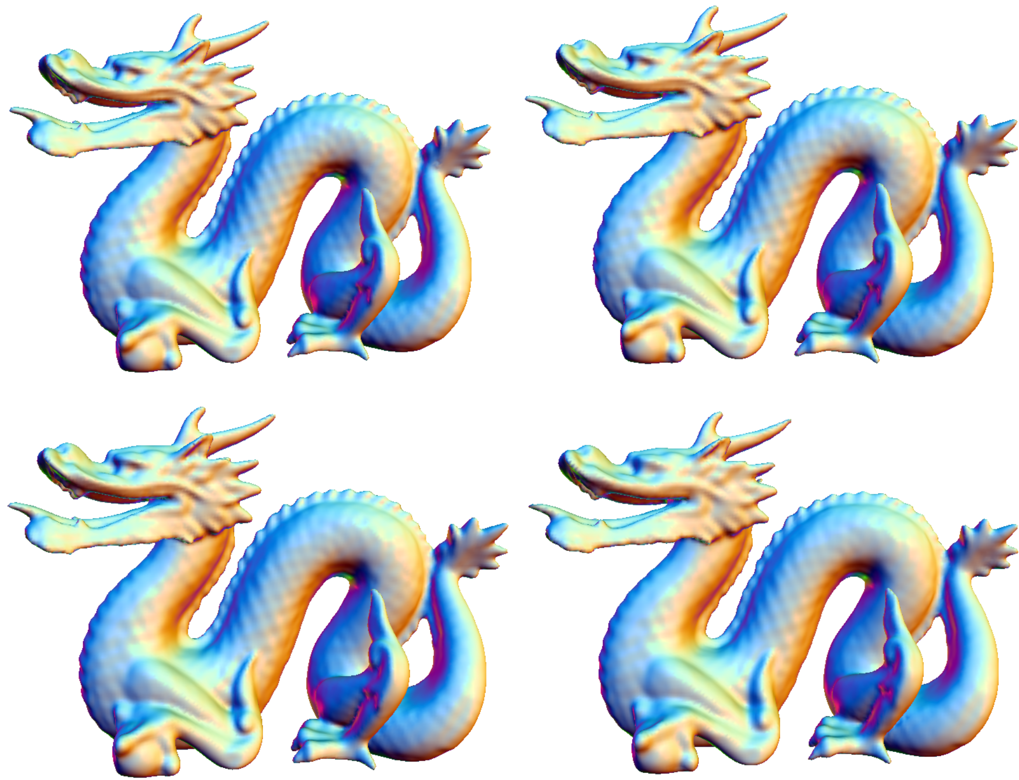


Figura 4.17: Reconstrucciones del modelo Dragon (100250 puntos) obtenidas usando la versión online dispersa de GP, con 100 % (superior izquierda), 50 % (superior derecha), 25 % (inferior izquierda) y 12.5 % (inferior derecha) de los datos

4.8. Cantidad de divisiones

El método propuesto requiere una cantidad mucho menor de divisiones, puesto que es capaz de crear reconstrucciones más complejas usando mayor cantidad de puntos por nodo.

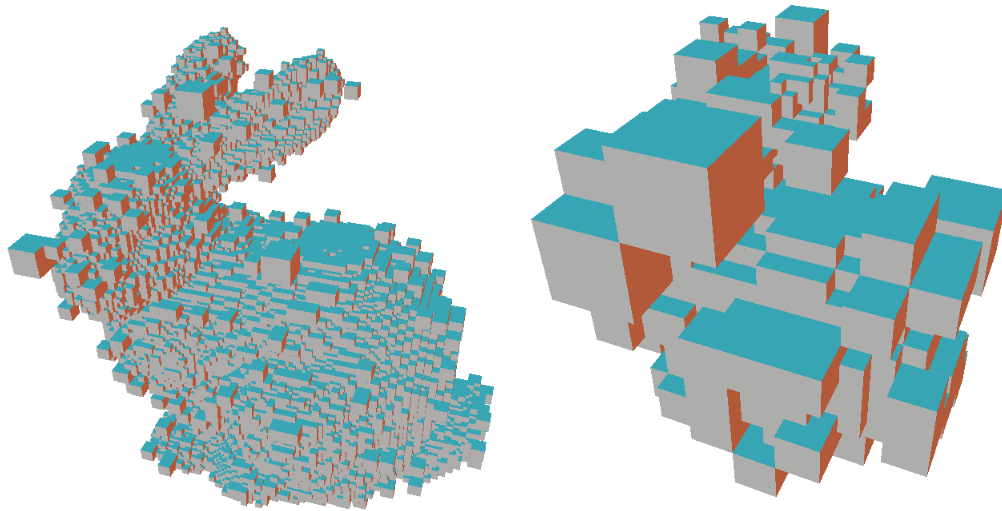


Figura 4.18: Octree del modelo Stanford Bunny. Izquierda: método original. Derecha: Método propuesto.

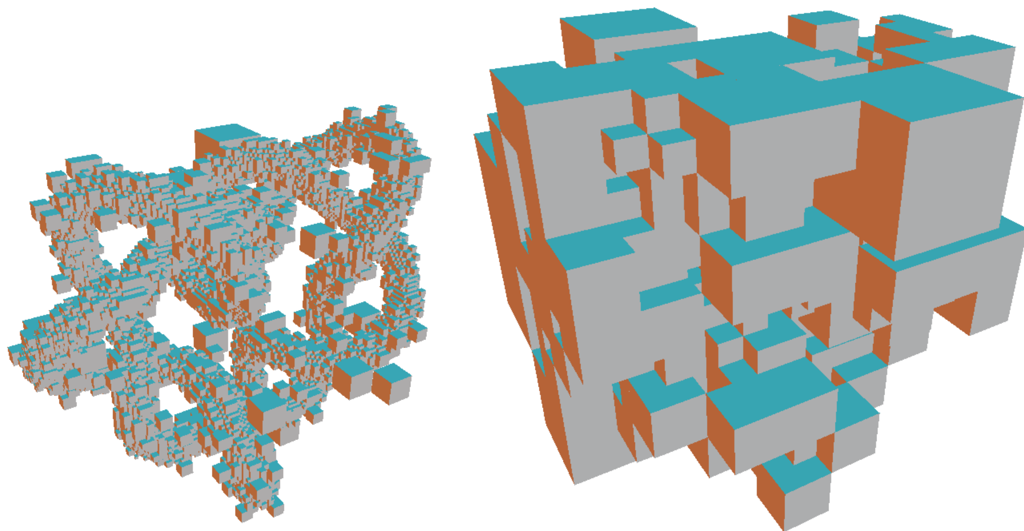


Figura 4.19: Octree del modelo Knot. Izquierda: método original. Derecha: Método propuesto.

Cuando se realiza una mayor cantidad de divisiones, se toma una porción menor

del modelo en cada nodo, lo que aumenta el riesgo de tener insuficientes datos para poder reconstruir los detalles o llenar los huecos. Adicionalmente, se aumenta el tiempo de ejecución y la memoria requerida. Por estos motivos es mejor tener una menor cantidad de divisiones.

Capítulo 5

Conclusión y trabajo futuro

Propusimos un método que combina dos técnicas poderosas: procesos Gaussianos y MPU. Por un lado, el método de procesos Gaussianos ha sido considerado por la comunidad de reconocimiento de patrones como uno de los que produce mejores aproximaciones, puesto que intenta tener una topología correcta en la reconstrucción de superficies. Por otro lado, el método MPU es un método implícito que es ahora una de las técnicas de reconstrucción más frecuentemente usada. Nuestro esquema de reconstrucción no sólo toma ventaja de estos dos métodos bien reconocidos, sino que también los unifica en un arreglo sencillo.

5.1. Contribuciones

- Por primera vez se combina el método MPU con un método de reconstrucción poderoso (proceso Gaussiano).
    - Hasta donde conocemos, en la literatura existe una única publicación que utiliza procesos Gaussianos (ver [31]), sin embargo, presenta la limitación de que debido a la complejidad del algoritmo, requiere de una pequeña cantidad de puntos, lo que no permite validar correctamente cómo sería una reconstrucción realista.
- Se logró una reconstrucción mejor de las zonas con una baja densidad de puntos
- Se logró una reconstrucción mejor de las zonas donde no había puntos

- Los detalles fueron recuperados más notoriamente
- Se lograron usar sólo los datos más representativos para obtener superficies simplificadas en la versión online dispersa
- Se preservó en todas las reconstrucciones la topología de los datos.

5.2. Trabajo futuro

- Investigar más opciones para kernels, de manera que se preserven mejor los detalles
- Buscar otras formas de eliminar la necesidad de las normales
- Conseguir robustez al ruido de una forma que no requiera normales
- Mejorar la calidad y estabilidad del proceso Gaussiano online disperso

Bibliografía

[1] Nina Amenta, Marshall W. Bern, y Manolis Kamvysselis. A new voronoi-based surface reconstruction algorithm. En *SIGGRAPH*, págs. 415–421. 1998. URL http://dblp.uni-trier.de/db/conf/siggraph/siggraph1998.html#AmentaBK98.

[2] Nina Amenta, Sunghee Choi, Tamal K. Dey, y N. Leekha. A simple algorithm for homeomorphic surface reconstruction. En *Symposium on Computational Geometry*, págs. 213–222. 2000.

[3] Nina Amenta y Yong Joo Kil. Defining point-set surfaces. *ACM Trans. Graph.*, 23(3):264–270, 2004.

[4] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006. ISBN 0387310738.

[5] J. C. Carr, R. K. Beatson, J. B. Cherrie, T. J. Mitchell, W. R. Fright, B. C. McCallum, y T. R. Evans. Reconstruction and representation of 3d objects with radial basis functions. En *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '01, págs. 67–76. ACM, New York, NY, USA, 2001. ISBN 1-58113-374-X. doi:10.1145/383259.383266. URL http://doi.acm.org/10.1145/383259.383266.

[6] J. C. Carr, R. K. Beatson, B. C. McCallum, W. R. Fright, T. J. McLennan, y T. J. Mitchell. Smooth surface reconstruction from noisy range data. En *Proceedings of the 1st International Conference on Computer Graphics and Interactive Techniques in Australasia and South East Asia*, GRAPHITE '03,

págs. 119–ff. ACM, New York, NY, USA, 2003. ISBN 1-58113-578-5. doi: 10.1145/604471.604495. URL http://doi.acm.org/10.1145/604471.604495.

[7] Lehel Csato. *Gaussian Processes - Iterative Sparse Approximations*. Tesis Doctoral, Aston University, 2002.

[8] Lehel Csató, Ernest Fokoué, Manfred Opper, Bernhard Schottky, y Ole Winther. Efficient approaches to gaussian process classification. En Sara A. Solla, Todd K. Leen, y Klaus-Robert Müller, eds., *NIPS*, págs. 251–257. The MIT Press, 1999. ISBN 0-262-19450-3. URL http://dblp.uni-trier.de/db/conf/nips/nips1999.html#CsatoFOSW99.

[9] Lehel Csató y Manfred Opper. Sparse on-line gaussian processes. *Neural Computation*, 14(3):641–668, 2002.

[10] Lehel Csató y Manfred Opper. Sparse on-line gaussian processes. *Neural Comput.*, 14(3):641–668, 2002. ISSN 0899-7667. doi:10.1162/089976602317250933. URL http://dx.doi.org/10.1162/089976602317250933.

[11] Tamal K. Dey y Samrat Goswami. Tight cocone: A water-tight surface reconstructor. *J. Comput. Inf. Sci. Eng.*, 3(4):302–307, 2003.

[12] Tamal K. Dey y Samrat Goswami. Provable surface reconstruction from noisy samples. En *Symposium on Computational Geometry*, págs. 330–339. 2004.

[13] Herbert Edelsbrunner y Ernst P. Mücke. Three-dimensional alpha shapes. *ACM Trans. Graph.*, 13(1):43–72, 1994.

[14] Arnaud Gelas, Sébastien Valette, Rémy Prost, y Wieslaw L. Nowinski. Variational implicit surface meshing. *Computers & Graphics*, 33(3):312–320, 2009. URL http://dblp.uni-trier.de/db/journals/cg/cg33.html#GelasVPN09.

[15] D. Levin. *Mesh-independent surface interpolation*, págs. 37–49. Spinger-Verlag, 2003.

[16] Thomas Lewiner, Hélio Lopes, Antônio Wilson Vieira, y Geovan Tavares. Efficient implementation of marching cubes cases with topological guarantees. *Jour-*

nal of Graphics Tools, 8(2):1–15, 2003. doi:10.1080/10867651.2003.10487582. URL http://thomas.lewiner.org/pdfs/marching_cubes_jgt.pdf.

[17] William E. Lorensen y Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *COMPUTER GRAPHICS*, 21(4):163–169, 1987.

[18] B. Mederos, M. Lage, S. Arouca, F. Petronetto, L. Velho, T. Lewiner, y H. Lopes. Regularized implicit surface reconstruction from points and normals. *Journal of the Brazilian Computer Society*, 13(4):7–15, 2007. ISSN 0104-6500. doi: 10.1007/BF03194253. URL http://dx.doi.org/10.1007/BF03194253.

[19] Boris Mederos, Nina Amenta, Luiz Velho, y Luiz Henrique de Figueiredo. Surface reconstruction for noisy point clouds. En *Symposium on Geometry Processing*, págs. 53–62. 2005.

[20] Ian T. Nabney. *NETLAB. Algorithms for Pattern Recognition*. Advances in Pattern Recognition. Springer, 2002. ISBN 1852334401.

[21] Yutaka Ohtake, Alexander Belyaev, Marc Alexa, Greg Turk, y Hans-Peter Seidel. Multi-level partition of unity implicits. *ACM Trans. Graph.*, 22(3):463–470, 2003. ISSN 0730-0301. doi:10.1145/882262.882293. URL http://doi.acm.org/10.1145/882262.882293.

[22] Manfred Opper. A bayesian approach to online learning. *Journal of Graphics Tools*, 1998. URL http://www.ki.tu-berlin.de/fileadmin/fg135/publikationen/opper/Op98b.pdf.

[23] Manfred Opper y Ole Winther. Gaussian processes for classification: Mean-field algorithms. *Neural Computation*, 12(11):2655–2684, 2000. URL http://dblp.uni-trier.de/db/journals/neco/neco12.html#OpperW00.

[24] Carl Edward Rasmussen y Christopher K. I. Williams. *Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning)*. The MIT Press, 2005. ISBN 026218253X.

[25] Conrad Sanderson. Armadillo: An open source c++ linear algebra library for fast prototyping and computationally intensive experiments. Inf. téc., NICTA, Australia, 2010.

[26] Vladimir V. Savchenko, Alexander A. Pasko, Oleg G. Okunev, y Tosiyasu L. Kunii. Function representation of solids reconstructed from scattered surface points and contours. *Comput. Graph. Forum*, 14(4):181–188, 1995. URL http://dblp.uni-trier.de/db/journals/cgf/cgf14.html#SavchenkoPOK95.

[27] Bernhard Scholkopf, Joachim Giesen, y Simon Spalinger. Kernel methods for implicit surface modeling. En *NIPS*. 2004.

[28] John Shawe-Taylor y Nello Cristianini. *Kernel Methods for Pattern Analysis*. Cambridge University Press, 2004. ISBN 978-0-521-81397-6.

[29] Tolga Tasdizen, Jean-Philippe Tarel, y David B. Cooper. Algebraic curves that work better. En *CVPR*, págs. 2035–2041. IEEE Computer Society, 1999. ISBN 0-7695-0149-4. URL http://dblp.uni-trier.de/db/conf/cvpr/cvpr1999.html#TarelC99.

[30] Gabriel Taubin. Distance approximations for rasterizing implicit curves. *ACM Trans. Graph.*, 13(1):3–42, 1994. URL http://dblp.uni-trier.de/db/journals/tog/tog13.html#Taubin94.

[31] Oliver Williams y Andrew Fitzgibbon. Gaussian process implicit surfaces. 2007.

Apéndice A

Glosario

Soporte compacto

Una función tiene soporte compacto si es cero fuera de un conjunto compacto. Alternativamente, uno puede decir que una función tiene soporte compacto si su soporte es un conjunto compacto.

Soporte

El conjunto cerradura del conjunto de argumentos de una función f para el cual f no es 0.

Cerradura

La cerradura topológica de un subconjunto A de un espacio topológico $\boldsymbol{X}$ es el subconjunto cerrado más pequeño de $\boldsymbol{X}$ que contiene a A.

Conjunto compacto

Un subconjunto S de un espacio topológico $\boldsymbol{X}$ es compacto si para cada cubierta abierta de S existe una subcubierta finita de S.

Cubierta abierta

Colección de conjuntos abiertos de un espacio topológico cuya unión contiene a un subconjunto dado.

Función suave

Una función suave es una función que tiene derivadas continuas hasta cierto orden deseado sobre algún dominio.

Producto de Schur

También conocido como producto de Hadamard o producto componente a componente. Se denota con el símbolo $\circ$. Si A y B son dos matrices (o vectores) con las mismas dimensiones, entonces $(A \circ B)$ es una matriz de las mismas dimensiones que cumple:

$$(A \circ B)_{i,j} = A_{i,j} \cdot B_{i,j} \tag{A.1}$$

Apéndice B

Detalles de implementación

B.1. Herramientas utilizadas

El lenguaje seleccionado fue C++, junto con la librería de álgebra lineal Armadillo [25], la cual a su vez utiliza las librerías LAPACK y BLAS. Los principales motivantes fueron la velocidad de ejecución, seguida de la facilidad de uso provista por Armadillo. Se utilizó la librería wxWidgets para generar la interfaz gráfica, y Code::Blocks como IDE de desarrollo.

B.2. Procesos Gaussianos

Se partió de una librería para procesos Gaussianos Online basada en el trabajo de Csato [7], la cual fue completamente reescrita para incorporar los procesos Gaussianos en batch, y para utilizar la librería Armadillo. Además, se agregó la funcionalidad para obtener la varianza en las predicciones, lo que permite obtener una idea sobre la calidad de las mismas.

Los procesos Gaussianos se implementan usando una clase base `GP`, la cual es heredada por las siguientes clases:

- `BatchGp`: Aprendizaje en batch
- `BatchMeanGP`: Aprendizaje en batch que sólo calcula la media (no la varianza). Es más eficiente.

- `SOGP` Aprendizaje online disperso. Es la versión más eficiente, pero tiene menos precisión.

Todas estas clases comparten la misma interfaz, que básicamente consiste de las siguientes funciones miembro:

- Constructor:

```
GP(int kernelType=RBF,double w=0.1, double s20=0.1);
```

  Recibe el tipo de kernel a usar, un parámetro para el kernel y la varianza. La clase SOGP tiene un argumento extra: la capacidad.

- Aprendizaje:

```
void learn(const arma::mat &input, const arma::mat &output);
```

  Recibe una matriz con los datos de entrada (cada columna es un dato), y una matriz con las salidas. Realiza el aprendizaje con dichas matrices.

- Predicción:

```
arma::mat predict(const arma::mat &input,arma::vec *sigma=nullptr)
```

  Recibe una matriz con los datos de entrada (cada columna es un dato), y opcionalmente un vector para guardar la varianza resultante. Realiza la predicción con los datos de entrada, y opcionalmente escribe la varianza en el vector *sigma*.

B.3. Kernels

Los procesos Gaussianos requieren el uso de un Kernel, el cual ha sido implementado usando una clase base llamada *Kernel* que contiene los siguientes métodos básicos:

- Constructor:

```
Kernel(size_t dimension);
```

Recibe la dimensión del kernel.

- Cálculo del kernel:

```
arma::mat calculateKernel(const arma::mat \&X,const arma::mat \&Y);
```

Recibe dos matrices A, B y devuelve una matriz R donde R_ij es igual al kernel calculado entre la columna i de la matriz A y la columna j de la matriz B.

- Derivada:

```
arma::mat getDerivativeMatrix(arma::mat X, size_t indexParameter);
```

Recibe una matriz X y devuelve una matriz R donde R_ij es igual a la derivada del kernel calculada entre la columna i y la columna j de la matriz X.

B.4. Optimización de parámetros

Se creó una clase abstracta llamada `Optimizer`, la cual es heredada por la clase `ScaleConjugateGradient`, como se muestra a continuación:

```
class Optimizer {
public:
   double xPrecision=1e-4,yPrecision=1e-4;
   size_t maxIter=100;
   virtual arma::vec solve(arma::vec params,
                  std::function<double(arma::vec)> error,
                  std::function<arma::vec(arma::vec)> gradient)=0;
};

class ScaleConjugateGradient : public Optimizer {
public:
   arma::vec solve(arma::vec params,
                  std::function<double(arma::vec)> error,
                  std::function<arma::vec(arma::vec)> gradient);
};
```

El método `solve` requiere de los siguientes argumentos:

- Un vector `params` que contiene una aproximación inicial.
- Una función `error` : $\mathbb{R}^d \to \mathbb{R}$
- Una función `gradient` : $\mathbb{R}^d \to \mathbb{R}^d$

Es posible ajustar los parámetros `maxIter` , `xPrecision` y `yPrecision`, los cuales controlan la terminación del algoritmo. Los criterios de terminación son los siguientes:

- El número de iteraciones supera `maxIter`
- El gradiente es menor a un valor ϵ (determinado como la distancia entre 1 y el menor valor mayor a 1 que es representable)
- El cambio en `params` es menor a `xPrecision` y el cambio en la función de error es menor a `yPrecision`

Las funciones `error` y `gradient` fueron implementadas en la clase `GP`:

- Error (ecuación 3.2)

```
double GP::calculateError(arma::vec params);
```

- Gradiente (ecuación 3.3)

```
vec GP::calculateGradient(arma::vec params);
```

Generalmente, los parámetros del kernel pertenecen a $\mathbb{R}^{d+}$, lo cual conlleva a un problema de optimización con restricciones. Sin embargo, este caso puede llevarse a un problema de optimización sin restricciones calculando el logaritmo de los parámetros, haciendo las siguientes transformaciones:

- Parámetros: `log(params)`
- Error: `calculateError(exp(logparams))`
- Gradiente: `calculateGradient(exp(logparams))%exp(logparams)`

- Parámetros en el espacio original: `exp(logparams)`

La transformación del gradiente se obtiene al derivar `calculateError(exp(logparams))` con respecto a `logparams`, usando la siguiente expresión general:

$$\frac{\partial f(e^{\mathbf{u}_1}, ..., e^{\mathbf{u}_d})}{\partial \mathbf{u}} = \nabla f(e^{\mathbf{u}_1}, ..., e^{\mathbf{u}_d}) \circ (e^{\mathbf{u}_1}, ..., e^{\mathbf{u}_d}) \tag{B.1}$$

Nota: En código de Armadillo, el producto de Schur se representa con el operador `%`.

B.4.1. Gradiente del Kernel

Para calcular gradiente, se creó la siguiente función:

```
virtual double Kernel::getDerivative(const arma::vec& d1, const arma::vec&
    d2, size_t indexParameter);
```

Esta función cambia en cada clase derivada de `Kernel` y según el parámetro con respecto al cual se está derivando.

B.5. MPU

El algoritmo MPU construye un octree y, para cada nodo, requiere obtener el conjunto de puntos que caen dentro de su soporte compacto (ver figura 2.1). Para hacer esta búsqueda, se creó un KD-tree que permite encontrar rápidamente todos los puntos contenidos en una esfera arbitraria.

Adicionalmente, para mejorar el rendimiento, se agregó una cantidad máxima de puntos en un nodo p_{max}, de modo que si un nodo tiene más de p_{max} puntos en su soporte compacto, entonces se divide inmediatamente sin realizar el aprendizaje. Esto no es necesario para lograr una buena reconstrucción, pero reduce considerablemente el tiempo de ejecución.

El aprendizaje en cada uno de los nodos puede realizarse de manera independiente a los demás, lo que permite procesar cada nodo en un hilo diferente. De igual manera, la evaluación de los puntos puede hacerse en múltiples hilos.

B.6. Hilos

El estándar C++11 tiene soporte para hilos, pero no para grupos de hilos (“Thread Pool”). Se creó una clase llamada `ThreadPool` que asigna tareas a un conjunto fijo de hilos. Esta clase es independiente del resto del código y puede ser reutilizada en otros proyectos, Consiste de los siguientes métodos básicos:

- Constructor:

  ```
  ThreadPool(size_t nThreads_);
  ```

  Recibe la cantidad de hilos. Si se llama son argumentos, la cantidad de hilos se toma como el máximo soportado por el procesador.

- Agregar tareas:

  ```
  void addTask(std::function<void(void)> f);
  ```

  Recibe una función y la agrega a la lista en espera. Es ejecutada tan pronto como sea posible por uno de los hilos.

- Esperar a la terminación de las tareas:

  ```
  void waitUntilFinished();
  ```

  Espera a que los hilos acaben con sus tareas y no haya más tareas pendientes.

B.7. Parámetros

Los parámetros escogidos son los siguientes:

- Factor de aumento del radio de los nodos del Octree
  0.05

- Error máximo permitido en la aproximación de cada nodo
  0.001 en figuras sin ruido, 0.1 en figuras con ruido

- Radio inicial del soporte compacto de cada nodo
  0.75 multiplicado por la longitud de la diagonal del cubo

- Profundidad máxima del octree
  15

- Partes a usar en la discretización para Marching Cubes
  Entre 80 y 300, dependiendo del nivel de detalle de la figura

- Mínima cantidad de puntos por nodo
  5

- Máxima cantidad de puntos por nodo
  300

- Varianza
  0.001 en figuras sin ruido, 0.1 en figuras con ruido

- Kernel
  Thin Plate, con R igual al diámetro del nodo (más dos veces el desplazamiento de normales en caso de usar ese método).

- Desplazamiento de normales
  0.005

Apéndice C

Código reutilizable

Este apéndice contiene las clases que fueron creadas de manera modular con el propósito de mantenerlas independientes del resto y hacerlas reutilizables para otros proyectos. Todo el código que se ve aquí puede copiarse directamente hacia otros proyectos. Se utiliza el lenguaje C++, con el estándar C++11.

C.1. Thread Pool

Asigna tareas a un conjunto fijo de hilos.

Listing C.1: ThreadPool.h

```
#ifndef THREADPOOL_H
#define THREADPOOL_H

#include <deque>
#include <vector>
#include <functional>
#include <thread>
#include <mutex>
#include <condition_variable>

class ThreadPool {
public:
```

```
    //Main thread only
    ThreadPool(size_t nThreads_=0);
    ~ThreadPool();
    void waitUntilFinished();
    void abort();
    //Thread safe
    void addTask(std::function<void(void)> f);
private:
    size_t nThreads,availableThreads;
    bool paused=false,aborted=false,terminating=false;
    std::vector<std::thread> workers;
    std::deque<std::function<void(void)> > tasks;
    std::mutex myMutex;
    std::condition_variable finishedWaiter, newTaskWaiter;
    ThreadPool (const ThreadPool&) = delete; //This object should not be
        copied
    ThreadPool& operator = (const ThreadPool&) = delete; //This object
        should not be copied
    void runThread(size_t id);
};

#endif // THREADPOOL_H
```

Listing C.2: ThreadPool.cpp

```
#include "ThreadPool.h"
typedef std::unique_lock<std::mutex> Lock;

ThreadPool::ThreadPool(size_t nThreads_) {
    nThreads = nThreads_;
    if (nThreads==0) {
        nThreads = std::thread::hardware_concurrency();
        if (nThreads==0) nThreads=1;
    }
    availableThreads=0;
```

```
    for (size_t i=0;i<nThreads;i++)
        workers.emplace_back(&ThreadPool::runThread,this,i);
}
ThreadPool::~ThreadPool() {
    waitUntilFinished();
    terminating = true;
    newTaskWaiter.notify_all(); //this tells every worker thread to
        terminate
    for (auto &worker:workers)
        worker.join();
}
void ThreadPool::addTask(std::function<void(void)> f) {
    Lock lock(myMutex);
    if (aborted) return;
    tasks.push_back(f);
    newTaskWaiter.notify_one();
}
void ThreadPool::waitUntilFinished() {
    Lock lock(myMutex);
    finishedWaiter.wait(lock,[this] {return
        (this->availableThreads==this->nThreads && this->tasks.empty());});
}
void ThreadPool::runThread(size_t id) {
    while (true) {
        Lock lock(myMutex);
        availableThreads++;
        if (tasks.empty() && availableThreads==nThreads)
            finishedWaiter.notify_all();
        newTaskWaiter.wait(lock,[this,id] {return (!this->tasks.empty() ||
            this->terminating || id>=this->nThreads);});
        availableThreads--;
        if (terminating || id>=nThreads)
            return;
        std::function<void(void)> f;
        f.swap(tasks.front());
```

```
        tasks.pop_front();
        lock.unlock();
        f();
    }
}
void ThreadPool::abort(){
    myMutex.lock();
    aborted=true;
    tasks.clear();
    myMutex.unlock();

    waitUntilFinished();

    myMutex.lock();
    aborted=false;
    myMutex.unlock();
}
```

Apéndice D

Manual Técnico

El siguiente documento es el manual técnico generado con *Doxygen* del código desarrollado.

Gaussian Process MPU

Generated by Doxygen 1.8.5

Wed May 14 2014 15:34:47

Contents

1 Hierarchical Index **1**

1.1 Class Hierarchy . . . 1

2 Class Index **2**

2.1 Class List . . . 2

3 Class Documentation **3**

3.1 BatchGP Class Reference . . . 3

3.1.1 Detailed Description . . . 3

3.2 BatchMeanGP Class Reference . . . 3

3.2.1 Detailed Description . . . 4

3.3 GP Class Reference . . . 4

3.3.1 Detailed Description . . . 6

3.3.2 Constructor & Destructor Documentation . . . 6

3.3.3 Member Function Documentation . . . 6

3.3.4 Member Data Documentation . . . 7

3.4 GpOctree Class Reference . . . 8

3.4.1 Detailed Description . . . 9

3.4.2 Member Function Documentation . . . 9

3.4.3 Member Data Documentation . . . 11

3.5 KDCell Class Reference . . . 11

3.5.1 Detailed Description . . . 12

3.5.2 Member Function Documentation . . . 12

3.5.3 Member Data Documentation . . . 14

3.6 Kernel Class Reference . . . 14

3.6.1 Detailed Description . . . 15

3.6.2 Constructor & Destructor Documentation . . . 15

3.6.3 Member Function Documentation . . . 16

3.6.4 Member Data Documentation . . . 20

3.7 Optimizer Class Reference . . . 21

3.7.1 Detailed Description . . . 21

3.7.2 Member Function Documentation . . . 21

3.8 PolynomialKernel Class Reference . . . 22

3.8.1 Detailed Description . . . 22

3.9 RBFKernel Class Reference . . . 22

3.9.1 Detailed Description . . . 23

3.10 RBFKernelSimple Class Reference . . . 23

3.10.1 Detailed Description . . . 24

3.11 ScaleConjugateGradient Class Reference . . . 24

3.11.1 Detailed Description 25
3.12 SimpleSurface Class Reference 25
3.12.1 Detailed Description 26
3.13 SOGP Class Reference 26
3.13.1 Detailed Description 26
3.14 ThinPlateKernel Class Reference 27
3.14.1 Detailed Description 27
3.15 ThreadMediator Class Reference 27
3.15.1 Detailed Description 28
3.15.2 Constructor & Destructor Documentation 28
3.15.3 Member Function Documentation 28
3.16 ThreadPool Class Reference 29
3.16.1 Detailed Description 30
3.16.2 Constructor & Destructor Documentation 30
3.16.3 Member Function Documentation 31

Index **33**

1 Hierarchical Index

1.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

GP 4
- **BatchGP** 3
- **BatchMeanGP** 3
- **SOGP** 26

GpOctree 8

KDCell 11

Kernel 14
- **PolynomialKernel** 22
- **RBFKernel** 22
- **RBFKernelSimple** 23
- **ThinPlateKernel** 27

Optimizer 21
- **ScaleConjugateGradient** 24

SimpleSurface 25

ThreadMediator 27

ThreadPool 29

2 Class Index

2.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

BatchGP
Batch Gaussian Process class 3

BatchMeanGP
Mean-only Batch Gaussian Process class 3

GP
Gaussian Process base class 4

GpOctree
Octree cell for the MPU algorithm 8

KDCell
Cell for the KD tree that stores the samples 11

Kernel
Gaussian Process Kernel 14

Optimizer
Class used as base for optimizer methods 21

PolynomialKernel
Polynomial kernel 22

RBFKernel
RBF kernel 22

RBFKernelSimple
RBF simple kernel 23

ScaleConjugateGradient
Scale Conjugate Gradient optimizer 24

SimpleSurface
Class which represents a simple 3D surface 25

SOGP
Sparse Online Gaussian Process class 26

ThinPlateKernel
Thin plate kernel 27

ThreadMediator
Class used to communicate threads with the GUI 27

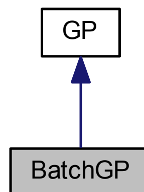
ThreadPool
Thread Pool class 29

3 Class Documentation

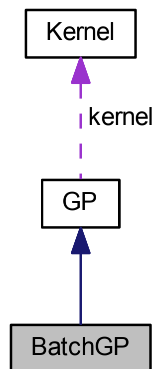
3.1 BatchGP Class Reference

Batch Gaussian Process class.

Inheritance diagram for BatchGP:



Collaboration diagram for BatchGP:



Additional Inherited Members

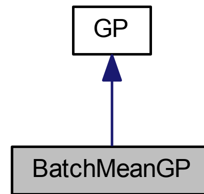
3.1.1 Detailed Description

Batch Gaussian Process class.

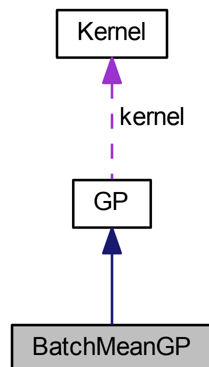
3.2 BatchMeanGP Class Reference

Mean-only Batch Gaussian Process class.

Inheritance diagram for BatchMeanGP:



Collaboration diagram for BatchMeanGP:



Additional Inherited Members

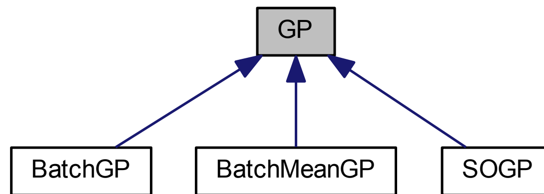
3.2.1 Detailed Description

Mean-only Batch Gaussian Process class.

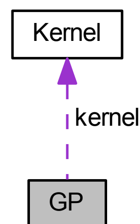
3.3 GP Class Reference

Gaussian Process base class.

Inheritance diagram for GP:



Collaboration diagram for GP:



Public Member Functions

- GP (size_t dimension_, int kernelType_=KERNEL_RBF, double w_=0.1, double s20_=0.1)

  Initializes the GP object.
- virtual void learn (const arma::mat &input, const arma::mat &output)=0

  Perform the learning process.
- virtual arma::mat predict (const arma::mat &input, arma::vec ∗sigma=nullptr)=0

  Perform prediction on a set of points, optionally returning variance.
- double calculateError (arma::vec params)

  Calculate the error (log-likelihood) for a set of parameters with the data used for learning.
- arma::vec calculateGradient (arma::vec params)

  Calculate the gradient of the error (log-likelihood) for a set of parameters with the data used for learning.
- void setPriors (const arma::vec means, const arma::vec variances)

  Set the prior mean and variance.
- void optimizeParameters (int maxIter=100, double xPrecision=1e-4, double yPrecision=1e-4)

  Use the Scale Conjugate Gradient optimizer to adjust parameters.

Protected Attributes

- double s20

  Variance.
- Kernel ∗ kernel

  Kernel object.
- int kernelType

  Kernel type.
- bool hasPrior

  Indicates whether setPriors() has been called.
- arma::vec priorMeans

  Vector with the prior means.
- arma::vec priorVariances

  Vector with the prior variances.
- arma::mat X

  Input data used for learning.
- arma::vec Y

  Output data used for learning.

3.3.1 Detailed Description

Gaussian Process base class.

3.3.2 Constructor & Destructor Documentation

3.3.2.1 GP::GP (size_t *dimension_*, int *kernelType_* = KERNEL_RBF, double *w_* = 0.1, double *s20_* = 0.1)

Initializes the GP object.

Parameters

dimension	Dimension to use, passed to the kernel.
*kernelType_*	Enumeration value from KERNEL_TYPES for the desired kernel.
*w_*	Parameter to pass to the kernel.
*s20_*	Variance.

3.3.3 Member Function Documentation

3.3.3.1 double GP::calculateError (arma::vec *params*)

Calculate the error (log-likelihood) for a set of parameters with the data used for learning.

Parameters

params	Vector with the parameters. The first one is the variance s20, the others are passed to the kernel.

Returns

The error (log-likelihood).

3.3.3.2 vec GP::calculateGradient (arma::vec *params*)

Calculate the gradient of the error (log-likelihood) for a set of parameters with the data used for learning.

Parameters

params	Vector with the parameters. The first one is the variance s20, the others are passed to the kernel.

Returns

The gradient of the error (log-likelihood).

3.3.3.3 virtual void GP::learn (const arma::mat & *input*, const arma::mat & *output*) [pure virtual]

Perform the learning process.

Parameters

input	Data to use as input, with one column per point.
output	Data to use as output, with one column per point.

3.3.3.4 void GP::optimizeParameters (int *maxIter* = 100, double *xPrecision* = 1e-4, double *yPrecision* = 1e-4)

Use the Scale Conjugate Gradient optimizer to adjust parameters.

Parameters

maxIter	Maximum number of iterations
xPrecision	Precision limit on the argument
yPrecision	Precision limit on the function image

3.3.3.5 virtual arma::mat GP::predict (const arma::mat & *input*, arma::vec * *sigma* = nullptr) [pure virtual]

Perform prediction on a set of points, optionally returning variance.

Parameters

input	Data on which to perform prediction, with one column per point.
sigma	Optional pointer to a vector on which to return the variance.

Returns

The predicted values, with one column per value.

3.3.3.6 void GP::setPriors (const arma::vec *means*, const arma::vec *variances*)

Set the prior mean and variance.

These can be used to optimize parameters.

3.3.4 Member Data Documentation

3.3.4.1 bool GP::hasPrior [protected]

Indicates whether setPriors() has been called.

3.3.4.2 int GP::kernelType [protected]

Kernel type.

This is used to save (serialize) and load (deserialize) the object.

3.3.4.3 arma::vec GP::priorMeans [protected]

Vector with the prior means.

This can be used to optimize parameters

3.3.4.4 arma::vec GP::priorVariances [protected]

Vector with the prior variances.

This can be used to optimize parameters

3.3.4.5 arma::mat GP::X [protected]

Input data used for learning.

This can be used to optimize parameters

3.3.4.6 arma::vec GP::Y [protected]

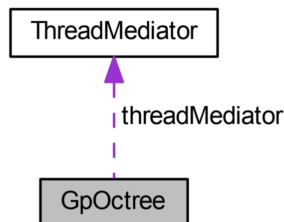
Output data used for learning.

This can be used to optimize parameters

3.4 GpOctree Class Reference

Octree cell for the MPU algorithm.

Collaboration diagram for GpOctree:



Public Member Functions

- void clearRoot ()

  Clears the MPU tree for future re-initialization.
- void clearReadedFile ()

  Deletes any readed point samples, and clears any calculated MPU tree.
- void runMarchingCubes ()

  Runs the MPU algorithm.
- void buildTree ()

  Builds the octree recursively.
- bool readed ()

  Indicates if a file with samples has been read.
- void drawSingleCell (int drawType)

  Draws the octree.

Static Public Attributes

- static size_t minPointsRequired = 50

  Minimum number of points needed to do the computation.
- static size_t maxPointsRequired = 250

  Maximum number of points allowed to do the computation.
- static double _lambda1 = 0.05

  Neighborhood radius.
- static double maxAllowedError = 0.005

  Maximum error allowed for a cell.
- static bool Gp3D =true

  Indicates whether to use the 3D version with normal-displacement (true), or the 2D version with principal component analysis (false).
- static bool shouldUseBatch =false

  Indicates whether to use the batch or online version of GP.
- static bool shouldPostProcess =false

  Indicates whether to apply post-processing.
- static double gpW =1.0

  Parameter to pass to the GP kernel.
- static double gps20 =0.01

  GP variance.
- static size_t kernelType =KERNEL_THINPLATE

  Kernel type to pass to the GP object.
- static double supportSizeParam = 1.5

  Support size parameter.
- static int maxDepth = 7

  Octree maximum depth.
- static int mcSamples = 80

  Number of partitions for Marching cubes.
- static ThreadMediator ∗ threadMediator =nullptr

  Object used to communicate with the GUI.
- static bool solved =false

  Indicates if the function buildTree() has been called.
- static MarchingCubes mc

  Marching Cubes object.

3.4.1 Detailed Description

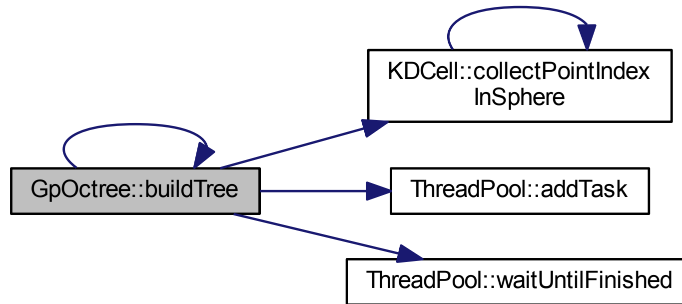
Octree cell for the MPU algorithm.

3.4.2 Member Function Documentation

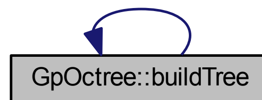
3.4.2.1 void GpOctree::buildTree ( )

Builds the octree recursively.

Here is the call graph for this function:



Here is the caller graph for this function:

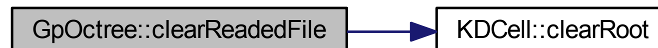


3.4.2.2 void GpOctree::clearReadedFile ( )

Deletes any readed point samples, and clears any calculated MPU tree.

This function can be called repeatedly without problem.

Here is the call graph for this function:



3.4.2.3 void GpOctree::clearRoot ( )

Clears the MPU tree for future re-initialization.

This function preserves the readed point samples.

This function can be called repeatedly without problem.

3.4.2.4 void GpOctree::drawSingleCell (int *drawType*)

Draws the octree.

Parameters

drawType	Can be OCT_FRAME, OCT_FACE, OCT_SUPPORT or several of them joined with the operator |

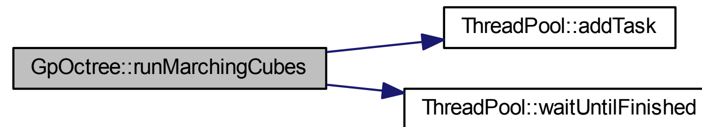
3.4.2.5 void GpOctree::runMarchingCubes ( )

Runs the MPU algorithm.

This function will delete any previously calculated MPU tree.

This function assumes that a file with samples has been readed.

Here is the call graph for this function:



3.4.3 Member Data Documentation

3.4.3.1 bool GpOctree::solved =false [static]

Indicates if the function buildTree() has been called.

3.5 KDCell Class Reference

Cell for the KD tree that stores the samples.

Public Member Functions

- void initializeRoot (const char ∗filename, size_t maxPoints=0, double s20=0)

  Initiates the root cell.
- void clearRoot ()

  Clears the tree for future re-initialization.
- std::vector< int > collectPointIndexInSphere (arma::vec3 &point, double radius)

  Returns a vector with all the points within a distance of a point.
- bool hasAnyPointInBox (arma::vec3 vmin, arma::vec3 vmax)

  Tests whether a box with given opposite corners contains any point.
- int getClosestPoint (const arma::vec3 &point)

  Gets the index of the point closest to a given point.
- void getClosestPoint (const arma::vec3 &point, int &id, double &closest)

  Gets the index of the point closest to a given point.

Static Public Attributes

- static int totalInstances =0

    Total number of instances.

3.5.1 Detailed Description

Cell for the KD tree that stores the samples.

3.5.2 Member Function Documentation

3.5.2.1 void KDCell::clearRoot ( )

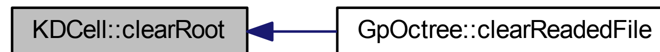
Clears the tree for future re-initialization.

This is called automatically by initializeRoot, so calling it manually isn't needed.

It can also be used to clear the memory used by the tree when it will no longer be used.

This function can be called repeatedly without problem.

Here is the caller graph for this function:



3.5.2.2 std::vector< int > KDCell::collectPointIndexInSphere (arma::vec3 & *point*, double *radius*)

Returns a vector with all the points within a distance of a point.

This function internally calls its overloaded alternative.

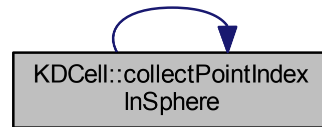
Parameters

point	The point
radius	The maximum distance to the point

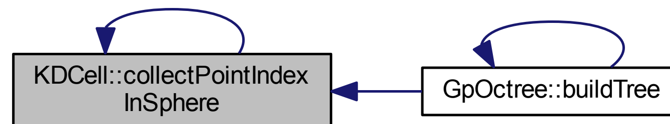
Returns

A vector with indexes of the points found.

Here is the call graph for this function:



Here is the caller graph for this function:



3.5.2.3 int KDCell::getClosestPoint (const arma::vec3 & *point*)

Gets the index of the point closest to a given point.

Parameters

point	Point for which to find the closest point.

Returns

Index of the closest point found.

3.5.2.4 void KDCell::getClosestPoint (const arma::vec3 & *point*, int & *id*, double & *closest*)

Gets the index of the point closest to a given point.

Parameters

point	Point for which to find the closest point.
id	Index of the closest point found (return value).
closest	Distance to the closest point found (return value).

3.5.2.5 bool KDCell::hasAnyPointInBox (arma::vec3 *vmin*, arma::vec3 *vmax*)

Tests whether a box with given opposite corners contains any point.

Parameters

vmin	Vector with coordinates (xmin, ymin, zmin)
vmax	Vector with coordinates (xmax, ymax, zmax)

Returns

True if any point is contained in that box, false otherwise.

3.5.2.6 void KDCell::initializeRoot (const char * *filename,* size_t *maxPoints* = 0, double *s20* = 0)

Initiates the root cell.

Parameters

filename	The file with the model (a pwm file)

3.5.3 Member Data Documentation

3.5.3.1 int KDCell::totalInstances =0 [static]

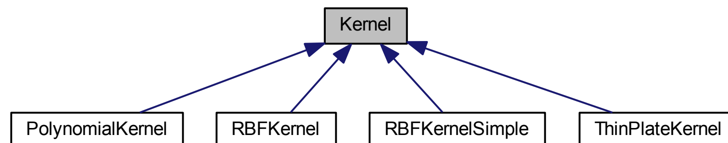
Total number of instances.

For debugging purposes only.

3.6 Kernel Class Reference

Gaussian Process Kernel.

Inheritance diagram for Kernel:



Public Member Functions

- Kernel (size_t dimension_)

  Creates the kernel with a fixed dimension.
- Kernel (size_t dimension_, const arma::vec params)

  Creates the kernel with a fixed dimension and a vector of parameters.
- double calculateSingleKernel (const arma::vec &X)

  Calculate the kernel function between a vector and itself.
- double calculateSingleKernel (const arma::vec &X, const arma::vec &Y)

  Calculate the kernel function between two vectors.
- arma::vec calculateKernelDiagonal (const arma::mat &X)

  Calculate the diagonal of the kernel matrix from a set of vectors with itself.
- arma::mat calculateKernel (const arma::mat &X)

Calculate the kernel matrix from a set of vectors with itself.

- arma::mat calculateKernel (const arma::mat &X, const arma::mat &Y)

  Calculate the kernel matrix between two sets of vectors.
- arma::mat getDerivativeMatrix (arma::mat X, size_t indexParameter)

  Calculate the derivative of the kernel matrix between two sets of vectors.
- bool save (std::ostream &file)

  Save the kernel to a file or any other stream (serialize).
- bool load (std::istream &file)

  Load the kernel from a file or any other stream (de-serialize).
- virtual double kernel (const arma::vec &d1, const arma::vec &d2)=0

  Calculate the kernel function between two vectors.
- virtual arma::vec getParameters ()

  Get the current parameters in a vector.
- virtual void setParameters (const arma::vec params)

  Sets the parameters to the values of a vector.
- virtual double getDerivative (const arma::vec &d1, const arma::vec &d2, size_t indexParameter)

  Calculate the derivative of the kernel function between two vectors.

Public Attributes

- const size_t dimension

  The kernel dimension.

3.6.1 Detailed Description

Gaussian Process Kernel.

3.6.2 Constructor & Destructor Documentation

3.6.2.1 Kernel::Kernel (size_t *dimension_*)

Creates the kernel with a fixed dimension.

Parameters

*dimension_*	The kernel dimension.

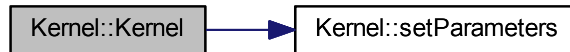
3.6.2.2 Kernel::Kernel (size_t *dimension_*, const arma::vec *params*)

Creates the kernel with a fixed dimension and a vector of parameters.

Parameters

*dimension_*	The kernel dimension.
params	The vector of parameters. Its size must match with the number of parameters used by the kernel.

Here is the call graph for this function:



3.6.3 Member Function Documentation

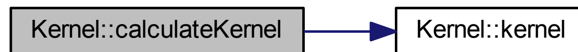
3.6.3.1 arma::mat Kernel::calculateKernel (const arma::mat & *X*)

Calculate the kernel matrix from a set of vectors with itself.

Parameters

X	The matrix with vectors on which to calculate the kernel. Each column is taken as a vector.

Here is the call graph for this function:



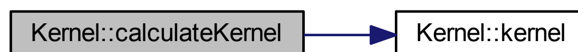
3.6.3.2 arma::mat Kernel::calculateKernel (const arma::mat & *X*, const arma::mat & *Y*)

Calculate the kernel matrix between two sets of vectors.

Parameters

X	The first matrix with vectors on which to calculate the kernel. Each column is taken as a vector.
Y	The second matrix with vectors on which to calculate the kernel. Each column is taken as a vector.

Here is the call graph for this function:



3.6.3.3 arma::vec Kernel::calculateKernelDiagonal (const arma::mat & *X*)

Calculate the diagonal of the kernel matrix from a set of vectors with itself.

This is equivalent to calculating the kernel of each vector with itself.

Parameters

X	The matrix with vectors on which to calculate the kernel. Each column is taken as a vector.

Here is the call graph for this function:

3.6.3.4 double Kernel::calculateSingleKernel (const arma::vec & *X*)

Calculate the kernel function between a vector and itself.

Parameters

X	The vector on which to calculate the kernel.

Here is the call graph for this function:

3.6.3.5 double Kernel::calculateSingleKernel (const arma::vec & *X*, const arma::vec & *Y*)

Calculate the kernel function between two vectors.

Parameters

X	The first vector on which to calculate the kernel.
Y	The second vector on which to calculate the kernel.

Here is the call graph for this function:

3.6.3.6 double Kernel::getDerivative (const arma::vec & *d1*, const arma::vec & *d2*, size_t *indexParameter*) [virtual]

Calculate the derivative of the kernel function between two vectors.

This function should be defined in a derived class if GP::optimizeParameters() will be called.

Parameters

d1	The first vector.
d2	The second vector.
indexParameter	Number of the parameter from which to calculate the derivative (starting at 0).

Here is the caller graph for this function:

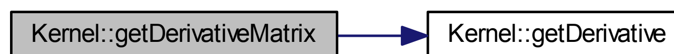
3.6.3.7 mat Kernel::getDerivativeMatrix (arma::mat *X*, size_t *indexParameter*)

Calculate the derivative of the kernel matrix between two sets of vectors.

Parameters

X	The first matrix with vectors on which to calculate the kernel derivative. Each column is taken as a vector.
Y	The second matrix with vectors on which to calculate the kernel derivative. Each column is taken as a vector.
indexParameter	Number of the parameter from which to calculate the derivative (starting at 0).

Here is the call graph for this function:



3.6.3.8 arma::vec Kernel::getParameters ( ) [virtual]

Get the current parameters in a vector.

This function should be defined in a derived class if GP::optimizeParameters() will be called.

Here is the caller graph for this function:

3.6.3.9 virtual double Kernel::kernel (const arma::vec & *d1*, const arma::vec & *d2*) [pure virtual]

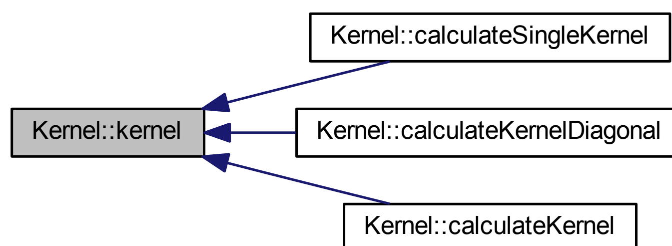
Calculate the kernel function between two vectors.

This function must be defined in every derived class.

Parameters

d1	The first vector.
d2	The second vector.

Here is the caller graph for this function:



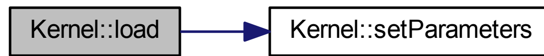
3.6.3.10 bool Kernel::load (std::istream & *file*)

Load the kernel from a file or any other stream (de-serialize).

Parameters

file	A stream from which to read the kernel.

Here is the call graph for this function:



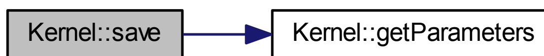
3.6.3.11 bool Kernel::save (std::ostream & *file*)

Save the kernel to a file or any other stream (serialize).

Parameters

file	A stream on which to write the kernel.

Here is the call graph for this function:



3.6.3.12 void Kernel::setParameters (const arma::vec *params*) [virtual]

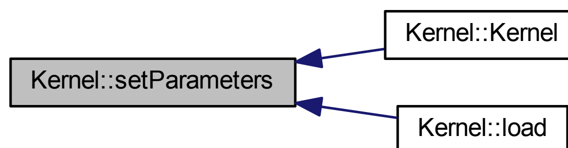
Sets the parameters to the values of a vector.

This function should be defined in a derived class if GP::optimizeParameters() will be called.

Parameters

params	The parameters vector.

Here is the caller graph for this function:



3.6.4 Member Data Documentation

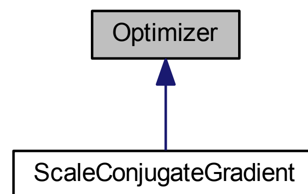
3.6.4.1 const size_t Kernel::dimension

The kernel dimension.

3.7 Optimizer Class Reference

Class used as base for optimizer methods.

Inheritance diagram for Optimizer:



Public Member Functions

- virtual arma::vec solve (arma::vec params, std::function< double(arma::vec)> error, std::function< arma::vec(arma::vec)> gradient)=0

  Solver function.

Public Attributes

- double xPrecision =1e-4

  Precision limit on the argument.
- double yPrecision =1e-4

  Precision limit on the function image.
- size_t maxIter =100

  Maximum number of iterations.

3.7.1 Detailed Description

Class used as base for optimizer methods.

3.7.2 Member Function Documentation

3.7.2.1 virtual arma::vec Optimizer::solve (arma::vec *params,* std::function< double(arma::vec)> *error,* std::function< arma::vec(arma::vec)> *gradient*) [pure virtual]

Solver function.

This function must be defined in every derived class.

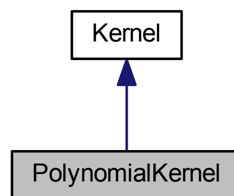
Parameters

params	Initial vector of parameters.
error	Error function, which receives a vector of parameters and returns a scalar.
gradient	Gradient function, which receives a vector of parameters and returns a vector.

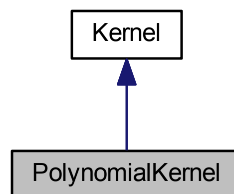
3.8 PolynomialKernel Class Reference

Polynomial kernel.

Inheritance diagram for PolynomialKernel:



Collaboration diagram for PolynomialKernel:



Additional Inherited Members

3.8.1 Detailed Description

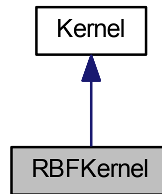
Polynomial kernel.

For information about this class please refer to Kernel.

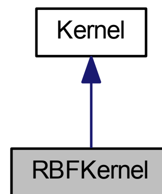
3.9 RBFKernel Class Reference

RBF kernel.

Inheritance diagram for RBFKernel:



Collaboration diagram for RBFKernel:



Additional Inherited Members

3.9.1 Detailed Description

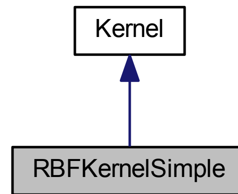
RBF kernel.

For information about this class please refer to Kernel.

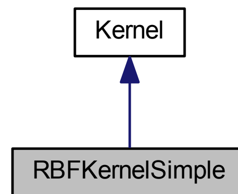
3.10 RBFKernelSimple Class Reference

RBF simple kernel.

Inheritance diagram for RBFKernelSimple:



Collaboration diagram for RBFKernelSimple:



Additional Inherited Members

3.10.1 Detailed Description

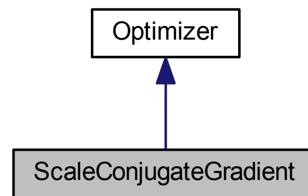
RBF simple kernel.

For information about this class please refer to Kernel.

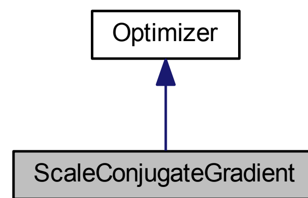
3.11 ScaleConjugateGradient Class Reference

Scale Conjugate Gradient optimizer.

Inheritance diagram for ScaleConjugateGradient:



Collaboration diagram for ScaleConjugateGradient:



Additional Inherited Members

3.11.1 Detailed Description

Scale Conjugate Gradient optimizer.

3.12 SimpleSurface Class Reference

Class which represents a simple 3D surface.

Public Member Functions

- bool save (const char ∗filename)

  Save the surface to a file.
- bool load (const char ∗filename)

  Load the surface from a file.
- void copyFromMC (MarchingCubes &mc)

  Copy the surface from a MarchingCubes object.
- void draw ()

  Draw the surface.

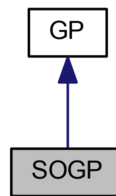
3.12.1 Detailed Description

Class which represents a simple 3D surface.

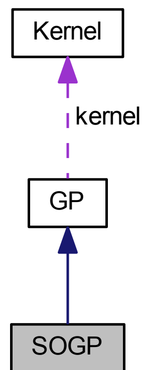
3.13 SOGP Class Reference

Sparse Online Gaussian Process class.

Inheritance diagram for SOGP:



Collaboration diagram for SOGP:



Additional Inherited Members

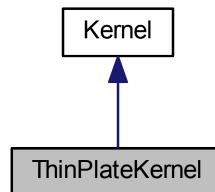
3.13.1 Detailed Description

Sparse Online Gaussian Process class.

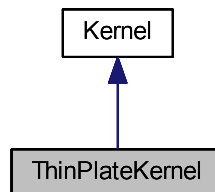
3.14 ThinPlateKernel Class Reference

Thin plate kernel.

Inheritance diagram for ThinPlateKernel:



Collaboration diagram for ThinPlateKernel:



Additional Inherited Members

3.14.1 Detailed Description

Thin plate kernel.

For information about this class please refer to Kernel.

3.15 ThreadMediator Class Reference

Class used to communicate threads with the GUI.

Public Member Functions

- ThreadMediator (wxFrame ∗parent)

    Create the object with a frame with which to communicate.
- void clearFlags ()

    Clears the "stop" flag.

- void stop ()

  Tells the threads to stop from the GUI.
- bool isStopped ()

  Called by the threads to test whether the GUI has called stop().
- void reportProgress (int x, bool additive=false)

  Report integer progress to the GUI.
- void reportFractionalProgress (double x, bool additive=false)

  Report fractional progress to the GUI.
- void reportMaxProgress (int x)

  Set the maximum integer progress to the GUI.
- void reportMaxFractionalProgress (double x=1)

  Set the maximum fractional progress to the GUI.
- void reportFinished ()

  Report to the GUI that we have finished.
- void reportFailed (const wxString &errorMessage)

  Report to the GUI that an error occurred, so we failed.
- void reportMessage (const wxString &message)

  Report a message to the GUI to show to the user.

Static Public Member Functions

- static void log (std::string message)

  Save a message to a log file.

3.15.1 Detailed Description

Class used to communicate threads with the GUI.

3.15.2 Constructor & Destructor Documentation

3.15.2.1 ThreadMediator::ThreadMediator (wxFrame * *parent*)

Create the object with a frame with which to communicate.

Parameters

parent	Frame with which to communicate

3.15.3 Member Function Documentation

3.15.3.1 void ThreadMediator::clearFlags ( )

Clears the "stop" flag.

3.15.3.2 bool ThreadMediator::isStopped ( )

Called by the threads to test whether the GUI has called stop().

3.15.3.3 void ThreadMediator::log (std::string *message*) [static]

Save a message to a log file.

Parameters

message	A message to record in a log file.

3.15.3.4 void ThreadMediator::reportFailed (const wxString & *errorMessage*)

Report to the GUI that an error occurred, so we failed.

Parameters

errorMessage	A message to show to the user.

3.15.3.5 void ThreadMediator::reportFractionalProgress (double *x*, bool *additive* = `false`)

Report fractional progress to the GUI.

Parameters

x	The amount of progress
additive	Whether this progress should be added to the previous one, or replace it

3.15.3.6 void ThreadMediator::reportMaxFractionalProgress (double *x* = `1`)

Set the maximum fractional progress to the GUI.

Parameters

x	The maximum progress

3.15.3.7 void ThreadMediator::reportMaxProgress (int *x*)

Set the maximum integer progress to the GUI.

Parameters

x	The maximum progress

3.15.3.8 void ThreadMediator::reportMessage (const wxString & *message*)

Report a message to the GUI to show to the user.

Parameters

message	A message to show to the user.

3.15.3.9 void ThreadMediator::reportProgress (int *x*, bool *additive* = `false`)

Report integer progress to the GUI.

Parameters

x	The amount of progress
additive	Whether this progress should be added to the previous one, or replace it

3.16 ThreadPool Class Reference

Thread Pool class.

Public Member Functions

- ThreadPool (size_t nThreads_=0)

  Creates the thread pool with a certain number of threads.
- ∼ThreadPool ()

  Waits for all threads to terminate before destroying them.
- void waitUntilFinished ()

  Waits for all threads to finish all tasks.
- void setNumberOfThreads (size_t newSize)

  Changes the number of available threads.
- void abort ()

  Clears the waiting tasks and waits for all threads to terminate their current tasks.
- void addTask (std::function< void(void)> f)

  Adds a task to the (LIFO or FIFO) queue of waiting tasks.
- void useFIFO ()

  Causes the tasks to be fetched in a FIFO way.
- void useLIFO ()

  Causes the tasks to be fetched in a LIFO way.
- void pause ()

  Pauses the execution of threads.
- void resume ()

  Resumes the execution of threads.

3.16.1 Detailed Description

Thread Pool class.

3.16.2 Constructor & Destructor Documentation

3.16.2.1 ThreadPool::ThreadPool (size_t *nThreads_* = 0)

Creates the thread pool with a certain number of threads.

Parameters

*nThreads_*	The desired number of threads. A value of 0 (or no value) indicates that it will be determined automatically.

3.16.2.2 ThreadPool::∼ThreadPool ( )

Waits for all threads to terminate before destroying them.

It is guaranteed that all tasks will be executed before destroying this object.

Here is the call graph for this function:

3.16.3 Member Function Documentation

3.16.3.1 void ThreadPool::abort ( )

Clears the waiting tasks and waits for all threads to terminate their current tasks.

In the time interval while this function is executing, adding new threads is disabled. It is guaranteed that after calling this thread no thread will be working.

Here is the call graph for this function:

Here is the caller graph for this function:

3.16.3.2 void ThreadPool::addTask (std::function< void(void)> *f*)

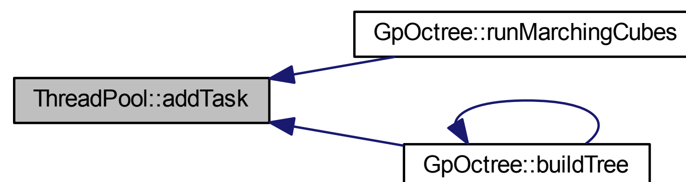
Adds a task to the (LIFO or FIFO) queue of waiting tasks.

If a thread is available when this function is called, it will be assigned this task immediately. This function can be called from a worker thread. Note that it will have no effect if the main thread is currently calling the function abort().

Parameters

f	The task.

Here is the caller graph for this function:



3.16.3.3 void ThreadPool::pause ( )

Pauses the execution of threads.

Tasks already in execution will not be affected, but no new task will start executing until the function resume() is called. Calling the destructor will resume execution of the remaining tasks.

3.16.3.4 void ThreadPool::resume ( )

Resumes the execution of threads.

This should be called at some point after the function pause(). Calling the destructor will also resume execution of the remaining tasks.

3.16.3.5 void ThreadPool::setNumberOfThreads (size_t *newSize*)

Changes the number of available threads.

Increasing the number of threads will immediately try to assign tasks to the new threads. Decreasing the number will pause the current thread until the exceeding threads terminate their current tasks. It is guaranteed that after calling this function the number of threads will be the value specified.

Parameters

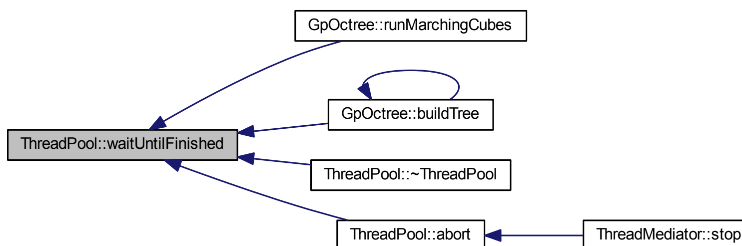
*nThreads_*	The desired new number of threads.

3.16.3.6 void ThreadPool::waitUntilFinished ( )

Waits for all threads to finish all tasks.

Calling this function will pause the current thread to wait. It is guaranteed that after calling this function all threads will have finished. This function can only be called from the main thread. Calling this function from a working thread will cause a deadlock (because the thread will try to wait for itself to terminate).

Here is the caller graph for this function:



Index

~ThreadPool
    ThreadPool, 30

abort
    ThreadPool, 31
addTask
    ThreadPool, 31

BatchGP, 3
BatchMeanGP, 3
buildTree
    GpOctree, 9

calculateError
    GP, 6
calculateGradient
    GP, 6
calculateKernel
    Kernel, 16
calculateKernelDiagonal
    Kernel, 16
calculateSingleKernel
    Kernel, 17
clearFlags
    ThreadMediator, 28
clearReadedFile
    GpOctree, 10
clearRoot
    GpOctree, 10
    KDCell, 12
collectPointIndexInSphere
    KDCell, 12

dimension
    Kernel, 20
drawSingleCell
    GpOctree, 10

GP, 4
    calculateError, 6
    calculateGradient, 6
    GP, 6
    GP, 6
    hasPrior, 7
    kernelType, 7
    learn, 7
    optimizeParameters, 7
    predict, 7
    priorMeans, 7
    priorVariances, 8
    setPriors, 7
    X, 8
    Y, 8
getClosestPoint
    KDCell, 13
getDerivative
    Kernel, 18
getDerivativeMatrix
    Kernel, 18
getParameters
    Kernel, 18
GpOctree, 8
    buildTree, 9
    clearReadedFile, 10
    clearRoot, 10
    drawSingleCell, 10
    runMarchingCubes, 11
    solved, 11

hasAnyPointInBox
    KDCell, 13
hasPrior
    GP, 7

initializeRoot
    KDCell, 14
isStopped
    ThreadMediator, 28

KDCell, 11
    clearRoot, 12
    collectPointIndexInSphere, 12
    getClosestPoint, 13
    hasAnyPointInBox, 13
    initializeRoot, 14
    totalInstances, 14
Kernel, 14
    calculateKernel, 16
    calculateKernelDiagonal, 16
    calculateSingleKernel, 17
    dimension, 20
    getDerivative, 18
    getDerivativeMatrix, 18
    getParameters, 18
    Kernel, 15
    kernel, 19
    load, 19
    save, 20
    setParameters, 20
kernel
    Kernel, 19
kernelType
    GP, 7

learn
    GP, 7
load
    Kernel, 19
log
    ThreadMediator, 28

optimizeParameters

GP, 7
Optimizer, 21
solve, 21

pause
ThreadPool, 31
PolynomialKernel, 22
predict
GP, 7
priorMeans
GP, 7
priorVariances
GP, 8

RBFKernel, 22
RBFKernelSimple, 23
reportFailed
ThreadMediator, 29
reportFractionalProgress
ThreadMediator, 29
reportMaxFractionalProgress
ThreadMediator, 29
reportMaxProgress
ThreadMediator, 29
reportMessage
ThreadMediator, 29
reportProgress
ThreadMediator, 29
resume
ThreadPool, 32
runMarchingCubes
GpOctree, 11

SOGP, 26
save
Kernel, 20
ScaleConjugateGradient, 24
setNumberOfThreads
ThreadPool, 32
setParameters
Kernel, 20
setPriors
GP, 7
SimpleSurface, 25
solve
Optimizer, 21
solved
GpOctree, 11

ThinPlateKernel, 27
ThreadMediator, 27
clearFlags, 28
isStopped, 28
log, 28
reportFailed, 29
reportFractionalProgress, 29
reportMaxFractionalProgress, 29
reportMaxProgress, 29
reportMessage, 29
reportProgress, 29
ThreadMediator, 28
ThreadMediator, 28
ThreadPool, 29
~ThreadPool, 30
abort, 31
addTask, 31
pause, 31
resume, 32
setNumberOfThreads, 32
ThreadPool, 30
ThreadPool, 30
waitUntilFinished, 32
totalInstances
KDCell, 14

waitUntilFinished
ThreadPool, 32

X
GP, 8

Y
GP, 8