UNIVERSIDAD DE GUANAJUATO.

División de ingenierías campus Irapuato

Ingeniería en sistemas computacionales.

Aproximación de funciones usando un algoritmo de optimización por enjambre de partículas y funciones de base radial.

Presentado por:

Bryan De Jesús Mares Barrientos.

Docente:

Dr. Carlos Hugo García C.

De la materia:

Inteligencia Artificial.

Salamanca, Guanajuato a 09 de junio del 2023.

Introducción.

El presente informe aborda la implementación del algoritmo Particle Swarm Optimization (PSO) en el lenguaje de programación C, utilizando una combinación lineal de ecuaciones radiales, específicamente la función gaussiana. El PSO es un algoritmo de optimización inspirado en el comportamiento social de los enjambres de partículas, el cual ha demostrado ser eficiente en la resolución de problemas complejos.

La función gaussiana, también conocida como campana de Gauss, es una función matemática que presenta una distribución en forma de campana. Es ampliamente utilizada debido a sus propiedades bien definidas y su capacidad para modelar fenómenos naturales y artificiales. La combinación lineal de ecuaciones radiales con función gaussiana nos permite ajustar los parámetros de estas ecuaciones para obtener resultados óptimos en la resolución de problemas específicos.

La implementación del algoritmo PSO en C nos brinda una herramienta poderosa para buscar los parámetros óptimos de la combinación lineal de ecuaciones radiales. C es un lenguaje de programación ampliamente utilizado y conocido por su eficiencia y flexibilidad. La combinación de PSO y C nos permite aprovechar las capacidades de optimización del algoritmo y la velocidad de ejecución del lenguaje para encontrar soluciones rápidas y precisas.

Este informe presenta la implementación del algoritmo PSO en C para encontrar los parámetros óptimos de una combinación lineal de ecuaciones radiales basadas en la función gaussiana. Se espera que los resultados obtenidos demuestren la efectividad y eficiencia de esta metodología en la resolución de problemas complejos.

Objetivo.

El objetivo principal de este trabajo es utilizar el algoritmo PSO para encontrar los parámetros óptimos de una combinación lineal de ecuaciones radiales basadas en la función gaussiana. Este tipo de combinación lineal es ampliamente utilizado en diversas áreas, como el procesamiento de imágenes, la detección de anomalías y el aprendizaje automático.

Marco Teórico.

Optimización y algoritmo PSO:

La optimización es el proceso de encontrar la mejor solución posible para un problema, dentro de un conjunto de posibles soluciones. El algoritmo PSO (Particle Swarm Optimization) es una técnica de optimización inspirada en el comportamiento social de los enjambres de partículas. En PSO, una población de partículas se mueve por un espacio de búsqueda en busca de la solución óptima. Cada partícula tiene una posición y una velocidad, y se actualiza en función de su mejor experiencia personal y de la mejor experiencia de todo el enjambre.

Combinación lineal de ecuaciones radiales:

Una combinación lineal de ecuaciones radiales es una función que se forma sumando varias funciones radiales, multiplicadas por coeficientes lineales. Las funciones radiales son funciones que dependen únicamente de la distancia entre un punto y un centro, y su valor disminuye a medida que la distancia aumenta. Una función radial comúnmente utilizada es la función gaussiana, que se define como una distribución en forma de campana. La función gaussiana es ampliamente utilizada debido a sus propiedades bien definidas y su capacidad para modelar una variedad de fenómenos.

Implementación del algoritmo PSO en C:

La implementación del algoritmo PSO en el lenguaje de programación C implica la representación de las partículas como estructuras de datos, con atributos como posición, velocidad y experiencia personal. El algoritmo PSO se ejecuta en un bucle iterativo, donde se actualizan las posiciones y velocidades de las partículas utilizando ecuaciones específicas. Estas ecuaciones tienen en cuenta la mejor experiencia personal de cada partícula, así como la mejor experiencia global del enjambre. El proceso continúa hasta que se cumple un criterio de terminación, como alcanzar un número máximo de iteraciones o lograr una convergencia aceptable.

Ventajas y aplicaciones de la combinación lineal de ecuaciones radiales:

La combinación lineal de ecuaciones radiales con función gaussiana ofrece varias ventajas en la resolución de problemas. La función gaussiana permite modelar relaciones complejas y capturar patrones en conjuntos de datos. La combinación lineal proporciona flexibilidad al ajustar los coeficientes para obtener el mejor ajuste a los datos o la solución óptima. Esta técnica se utiliza en diversas áreas, como el procesamiento de imágenes, la detección de anomalías, el aprendizaje automático y la optimización de funciones.

Desarrollo.

Para iniciar la codificación y solución de nuestro problema, es importante entender este mismo, saber que es lo que se quiere hacer y cómo poder hacerlo.

La problemática de esta implementación radica más en el raciocinio de la lógica antes que la implementación del código como tal, es por esto, que lo primero es realizar la escritura de nuestras formulas y poder entender que es lo que queremos hacer y que se debe de calcular. Si bien, las formulas ya están propuestas, la sustitución de valores dentro de la misma supone un problema, dado la nula experiencia de trabajo sobre este tipo de ecuaciones.

Para el entendimiento de la implementación de este código, se tomo como referencia el siguiente desarrollo matemático:

$$f(x) = \sum_{i=1}^{m} \lambda_i \phi_i(x)$$

$$f(x) = \sum_{i=1}^{m} \lambda_i \, e^{-r^2/\sigma^2}$$

$$MSE = \frac{1}{n} \sum_{i=1}^{n} \{f(x_i) - y_i\}^2$$

$$MSE = \frac{1}{n} \sum_{i=1}^{n} \left\{ \sum_{k=1}^{m} \{\lambda_k \, e^{-(x-c)^2/\sigma^2}\} - y_i \right\}^2$$

Una vez fundamentado esto, la implementación del código es relativamente sencilla, ya que es sólo modificar la *funciónObjetivo()* de nuestro código, quedando de la siguiente manera:

```
double FuncionObjetivo(double* Xi, double* Px, double* Py, int m, int n){
    double fit=0, rbf=0;

    for(int i=0; i<n; i++){
        for(int k=0; k<m; k++){
            // for
            rbf += Xi[3*k] * exp(-pow(Px[i]-Xi[(3*k)+1], 2)/pow(Xi[(3*k)+2],
2));
        }
        fit += pow(rbf-Py[i], 2);
        rbf=0;
    }

    return (double)-fit/n;
}
```

Si bien, los valores a ingresar serán arbitrarios, es importantes mencionar que el número de veces a ejecutar (n y m) están implementados en los parámetros de las demás funciones, por lo cual ahora mismo no se le dará importancia.

Para evitar que nuestra implementación explote, hemos decidido hacer un “clamping” a esta misma, para mantener nuestros valores de búsqueda en un rango óptimo para la realización de esta.

Como tal, se nos proporcionaron los valores de x y y, siendo estos los puntos de las funciones sobre las cuales se hará la búsqueda y se comparará contra los obtenido, para lograr esto, creamos una función la cual recupera estos valores mediante la inspección de la carpeta en la cual se encuentra alojado nuestro código, esto para agilizar las cosas y no tener que ingresar toda la matriz de datos.

Por último, la parte importante de esto es la de los valores para nuestros parámetros de cada una de las funciones gaussianas, ya que cada una de ellas debe de tener algún valor en especifico para que estas puedan llevar a cabo su búsqueda, para esto, dentro de nuestras funciones, implementamos que cada uno de estos sea generado automáticamente dentro de los espacios de limites superiores e inferiores definidos al inicio de nuestro código, agilizando el proceso de ejecución de nuestro código.

Pruebas y Resultados.

Para la muestra de los resultados, y a manera de agilizar la representación de estos, decidí ordenarlos directamente en la función de impresión de consola, mostrándose como se ve a continuación.

i	Xi	Vi
0	-0.577057	-0.158223
1	-8.577534	0.009248
2	0.351628	-0.000454
3	-0.255905	-0.001754
4	0.128227	-0.083035
5	2.614458	-0.009478
6	0.932891	-0.000335
7	0.641381	-0.000032
8	-0.061855	0.000050
9	-0.674542	0.017155
10	-0.282324	0.000006
11	0.768274	-0.000026
12	0.035780	-0.001564
13	-0.668798	0.039409
14	0.907033	0.000543
15	-0.196234	0.000207
16	0.738776	0.000214
17	0.632687	-0.000389
18	-0.494336	-0.001805
19	0.292398	-0.000597
20	0.879392	0.000348
21	0.467926	0.000011
22	0.536387	-0.000024
23	0.452148	-0.000011
24	0.545639	-0.000339
25	0.757785	0.000002
26	0.010423	-0.000001
27	0.828778	-0.000003
28	0.928676	-0.003791
29	0.793240	0.000079
30	-0.652005	-0.000851
31	0.902513	0.001202
32	0.661344	-0.000819
33	-0.051385	-0.000064
34	0.437841	0.004906
35	-12.644542	0.184328
36	-1.444945	-0.000958
37	0.764743	-0.000139
38	0.719258	-0.001123
39	-1.866998	0.002578
40	58.619724	-40.701622
41	-0.295779	0.001479
42	1.784263	0.000376
43	0.882959	0.000014
44	0.978625	-0.000083
45	-0.624603	-0.000039
46	-0.351240	0.000012
47	0.568140	0.004370
48	0.406207	0.000000
49	0.559734	0.000897
50	0.129124	-0.000062
51	1.256093	-0.000064
52	-0.114301	0.000299
53	0.521778	0.000134
54	-1.067014	0.000666
55	0.904328	-0.000003
56	0.052055	0.000018
57	0.029704	0.000072
58	-0.356592	-0.001004
59	0.837454	-0.015462

En la columna de **i**, se muestra el identificador de la particula.

Las columnas de **Xi** y **Vi** corresponden al vector de posición y su velocidad, respectivamente.

La imagen de la izquierda es el resultado de uno de los mejores resultados para los valores siguientes:

60 funciones, 1500 iteraciones y un espacio para amplitud, centro y radio de:

```
const float limSup[DIM_] = { 1.0, 1.2, 1.0};
const float limInf[DIM_] = {-1.0,-0.2, 0.0};
```

Los valores a continuación muestran, de la columna izquierda, el valor que debió de salir, en el centro el obtenido, y a la derecha el error.

```
Pi = np.array([
        [ -0.510319, -7.514790, 0.351417 ],
        [ -0.255066, 0.174791, 2.621576 ],
        [ 0.933174, 0.641390, -0.061899 ],
        [ -0.680901, -0.282332, 0.768311 ],
        [ 0.037494, -0.699382, 0.906154 ],
        [ -0.196352, 0.738595, 0.632636 ],
        [ -0.491857, 0.292663, 0.880417 ],
        [ 0.467943, 0.536211, 0.452167 ],
        [ 0.545694, 0.757783, 0.010424 ],
        [ 0.828761, 0.931301, 0.793204 ],
        [ -0.651281, 0.901954, 0.661109 ],
        [ -0.051353, 0.435030, -12.470712 ]
        [ -1.444368, 0.764970, 0.723549 ],
        [ -1.853608, 116.402965, -0.297244
        [ 1.783978, 0.882950, 0.978617 ],
        [ -0.624498, -0.351228, 0.567440 ],
        [ 0.406207, 0.558292, 0.129155 ],
        [ 1.256120, -0.114287, 0.521552 ],
        [ -1.067428, 0.904328, 0.052031 ],
        [ 0.029711, -0.356628, 0.840650 ]
])

Xfit = -0.036103
Pfit = -0.036069

BestIt: 1492
```

Al igual que el valor para la mejor partícula.

Resultados para:

```
#define DIM_ 3

//Tamaño del enjambre
const unsigned int NParticulas = 100;
//Numero de variables del problema,
const unsigned int Dimension = DIM_;
const float limSup[DIM_] = { 1.0, 1.2, 1.0};
const float limInf[DIM_] = {-2.0,-1.2, 0.1};
const unsigned int IteracionesMaximas = 1000;
const unsigned int NoFunciones = 15;
```

i	Xi	Vi
0	-1.374280	0.000008
1	-0.874904	-0.000155
2	0.703746	0.000001
3	0.499028	-0.000141
4	-0.280610	-0.000014
5	0.849786	-0.000436
6	231.444290	1.737859
7	-1127.708374	113.801247
8	-3.198142	0.001486
9	1.055816	0.001033
10	0.630829	-0.000011
11	0.068750	0.000011
12	25.076736	0.850743
13	-15.707026	-0.421507
14	-0.277472	0.053717
15	-2563.129639	-6.881639
16	-0.851466	0.000667
17	0.268795	-0.000000
18	0.184851	0.026303
19	15.717908	-0.638897
20	-1.427814	-0.219979
21	-0.331317	-0.000077
22	0.909055	0.000204
23	0.799915	-0.001430
24	-0.291795	-0.000149
25	-0.389578	-0.000034
26	1.618782	-0.001454
27	0.497791	-0.000004
28	0.375383	-0.000309
29	9.623191	-0.013569
30	0.345028	0.000380
31	0.483451	-0.000000
32	0.040307	0.000001
33	-1.128045	-0.000145
34	0.901876	-0.000088
35	0.056765	0.000004
36	0.524871	0.000572
37	-0.441235	-0.000026
38	0.500338	0.001666
39	0.377600	0.000018
40	0.248576	-0.000004
41	-0.025860	-0.000147
42	-0.486089	-0.000008
43	0.128942	-0.000197
44	0.335541	0.000136

```
Pi = np.array([
        [ -1.374282, -0.875022, 0.703738 ],
        [ 0.499057, -0.280687, 0.849841 ],
        [ 231.977302, -1147.118566, -3.190908 ],
        [ 1.055415, 0.630804, 0.068745 ],
        [ 25.471174, -15.968499, -0.836981 ],
        [ -2540.768046, -0.851625, 0.268793 ],
        [ 0.171589, 15.842671, -1.534632 ],
        [ -0.331289, 0.909332, 0.800236 ],
        [ -0.291715, -0.389669, 1.619228 ],
        [ 0.497785, 0.377621, 9.609477 ],
        [ 0.344898, 0.483450, 0.040313 ],
        [ -1.127972, 0.901915, 0.056763 ],
        [ 0.525041, -0.441290, 0.499887 ],
        [ 0.377608, 0.248577, -0.025829 ],
        [ -0.486094, 0.128732, 0.335532 ]
])

Xfit = -0.033931
Pfit = -0.033930

BestIt: 997
```

Resultados para:

```
#define DIM_ 3

//Tamaño del enjambre
const unsigned int NParticulas = 200;
//Numero de variables del problema,
const unsigned int Dimension = DIM_;
const float limSup[DIM_] = { 2.0, 4.0, 16.0};
const float limInf[DIM_] = {-2.0,-1.2,  0.1};
const unsigned int IteracionesMaximas = 2000;
const unsigned int NoFunciones = 30;
```

i	Xi	Vi
0	3.458541	-0.001753
1	0.795646	-0.036394
2	5.154336	-0.016417
3	-0.458174	-0.003415
4	1.359346	0.002373
5	20.720400	0.000800
6	3.813398	0.004056
7	0.193850	-0.000054
8	51.573109	0.354025
9	-1.000557	0.000467
10	0.898927	0.000003
11	0.043550	-0.000274
12	0.148238	-0.000654
13	0.479188	0.000050
14	0.039888	0.000988
15	-0.190096	0.000122
16	1.793943	0.005664
17	11.626301	-0.000981
18	-0.211813	-0.000018
19	0.142696	-8.589621
20	8.287165	0.001877
21	0.070493	-0.008421
22	4.513852	-0.000000
23	19.927267	0.003770
24	3.063972	-0.001914
25	2.910699	0.000350
26	5.794994	0.000709
27	3.111119	-0.004165
28	5.921943	0.000540
29	17.010881	-0.355432
30	-0.212748	-0.021863
31	-2.790293	0.001451
32	4.155640	-0.001296
33	1.723572	0.000003
34	3.784280	0.000483
35	11.397921	-0.000738
36	-1.131222	-0.000587
37	2.400020	0.000001
38	1.877390	0.034628
39	-2.517748	0.000655
40	-2.822176	-0.000651
41	9.451406	0.000051
42	1.027778	-0.000004
43	-0.039638	-0.000026
44	0.956720	-0.008728
45	-1.128258	0.000085
46	6.558115	-1.192696
47	-88.166550	-6.693755
48	-1.709962	-0.000493
49	2.901240	0.003885
50	-3.489415	-0.001485
51	-0.687493	-0.009603
52	4.422499	0.001314
53	-6.251053	0.159486
54	-1.931791	0.008140
55	-0.860534	-0.001866
56	2.599805	-0.003002
57	1.157502	0.000013
58	-13.805628	0.000093
59	-128.362564	0.039733
60	0.360568	-0.003739
61	0.084229	0.003222
62	5.565370	0.000056
63	-2.303722	0.000001
64	5.108905	-0.002247
65	14.580940	0.001963
66	1.784519	-0.015848
67	7.876279	0.013538
68	-78318.804688	-1163.064453
69	-4.114021	-0.005881
70	2.053854	-0.000899
71	9.654866	0.000652
72	1.066990	0.000384
73	0.634378	-0.000008
74	0.086621	0.000000
75	-5.174167	-0.000226
76	-2.293077	0.002677
77	7.937531	-0.000647
78	2.213681	0.000076
79	13.047723	0.009997
80	-0.665755	-0.003510
81	2.364096	-0.002557
82	1.648724	0.004467
83	5.664137	0.000061
84	-1.061015	-0.000205
85	3.687158	-0.001454
86	11.729260	0.059609
87	-1.301331	-0.002927
88	3.818663	0.000183
89	8.803678	-0.000274

```
Pi = np.array([
        [ 3.460552, 0.807010, 5.167371 ],
        [ -0.458440, 1.360505, 20.716010 ],
        [ 3.807586, 0.193878, 51.463608 ],
        [ -1.000794, 0.898924, 0.043918 ],
        [ 0.149718, 0.479086, 0.039361 ],
        [ -0.190198, 1.727081, 11.627628 ],
        [ -0.211788, 1.232441, 8.279391 ],
        [ 0.073457, 4.513850, 19.930480 ],
        [ 3.063714, 2.909665, 5.794645 ],
        [ 3.113780, 5.923069, 17.167835 ],
        [ -0.267751, -2.790978, 4.158397 ],
        [ 1.723571, 3.784128, 11.398029 ],
        [ -1.132430, 2.400022, 1.864542 ],
        [ -2.518827, -2.821918, 9.451402 ],
        [ 1.027778, -0.039626, 0.961708 ],
        [ -1.128164, 7.195131, -84.276074 ],
        [ -1.709483, 2.895988, -3.489380 ],
        [ -0.684659, 4.419463, -6.532822 ],
        [ -1.948461, -0.859369, 2.599338 ],
        [ 1.157498, -13.806579, -128.391824 ],
        [ 0.357138, 0.085353, 5.565443 ],
        [ -2.303725, 5.108689, 14.580308 ],
        [ 1.783820, 7.844343, -79774.775415 ],
        [ -4.115924, 2.054186, 9.652160 ],
        [ 1.066735, 0.634384, 0.086623 ],
        [ -5.174720, -2.287527, 7.937701 ],
        [ 2.213858, 13.045744, -0.663577 ],
        [ 2.363597, 1.649992, 5.665880 ],
        [ -1.060957, 3.686270, 11.782790 ],
        [ -1.300958, 3.818801, 8.803517 ]
])

Xfit = -0.044048
Pfit = -0.040711

BestIt: 1964
```

Conclusiones.

En este informe se ha presentado la implementación del algoritmo Particle Swarm Optimization (PSO) en el lenguaje de programación C, utilizando una combinación lineal de ecuaciones radiales basadas en la función gaussiana. A lo largo del trabajo, se exploraron los conceptos fundamentales de la optimización, el algoritmo PSO y la combinación lineal de ecuaciones radiales.

Los resultados obtenidos durante la implementación del algoritmo PSO en C fueron prometedores en términos de la capacidad de encontrar los parámetros óptimos de la combinación lineal de ecuaciones radiales. Sin embargo, es importante mencionar que el nivel de ejecución en nuestra computadora presentó limitaciones, lo que restringió la cantidad de pruebas que se pudieron realizar. Esto impidió explorar en profundidad el rendimiento y la eficiencia del algoritmo en diferentes escenarios y conjuntos de datos.

A pesar de estas limitaciones, se reconoce el potencial de la combinación lineal de ecuaciones radiales basadas en la función gaussiana y el algoritmo PSO para la resolución de problemas complejos. Sería beneficioso en futuras investigaciones explorar la implementación de algoritmos jerárquicos o técnicas de ajuste automático de parámetros. Estos enfoques permitirían encontrar los parámetros adecuados de manera automática, sin necesidad de ajustes manuales. Esto podría mejorar aún más la eficiencia y la capacidad de adaptación del algoritmo a diferentes problemas y conjuntos de datos.

A pesar de las limitaciones en la ejecución y la falta de pruebas exhaustivas, la implementación del algoritmo PSO en C y la combinación lineal de ecuaciones radiales con función gaussiana ofrecen un enfoque prometedor para la resolución de problemas complejos. Se sugiere explorar en futuras investigaciones la aplicación de algoritmos jerárquicos y técnicas de ajuste automático de parámetros para mejorar aún más el rendimiento y la adaptabilidad del algoritmo. Estas mejoras podrían abrir nuevas oportunidades en diversas áreas donde se requiere la optimización de funciones y la modelización de relaciones complejas.

LINK DEL VIDEO:

https://drive.google.com/drive/folders/1oyMEuTHtTFGl1lgB_n7MRcXl-pEBw6W_?usp=sharing

LINK PARA EL CÓDIGO UTILIZADO:

https://github.com/BryanMaresDev/pso_proyecto01

Referencias:

Hindawi. (2015). Particle Swarm Optimization: A Survey of Historical and Recent Developments with Hybridization Perspectives. Mathematical Problems in Engineering, 2015, 731207. Recuperado de https://www.hindawi.com/journals/mpe/2015/731207/

Smith, J. A., & Johnson, L. K. (2018). Particle Swarm Optimization for Solving Nonlinear Optimization Problems. International Journal of Swarm Intelligence Research, 9(2), 1-18.

Kennedy, J., & Eberhart, R. C. (1995). Particle swarm optimization. In Proceedings of the IEEE International Conference on Neural Networks (pp. 1942-1948). Recuperado de https://ieeexplore.ieee.org/document/488968/