Módulo para suporte à parametrização dinâmica de Métodos de Otimização Multi-objetivo utilizando técnicas de refatoração: um relato de caso sobre o framework *pymoo*

Davi Marchetti Giacomel, André Luiz Brun, Victor Francisco Araya Santander
Programa de Pós-Graduação em Ciência da Computação
Universidade Estadual do Oeste do Paraná
Cascavel, Paraná, Brasil
Email: {davi.giacomel1, andre.brun, victor.santander}@unioeste.br

***Abstract*—This paper describes a case study on the creation of an add-on module for the library *pymoo* for multi-objective optimization in the context of exploring proposals for dynamic parameterization of these methods. Starting from a proof-of-concept script that used the tools available in the library to effectively perform the dynamic parameterization of the mutation and crossover rates of *NSGA-II*, a sequence of refactorings was performed, resulting in a *wrapper* and other auxiliary classes that allow for simplified problem solving and dynamic parameter control by the library used. The main outcome of the work, besides documenting this refactoring process, is a proposed module that is useful for research and implementations that apply dynamic control to the parameterization of multi-objective methods.**

***Keywords*—NSGA; refactoring; dynamic parameterization.**

***Resumo*—Este artigo descreve um estudo de caso sobre a criação de um módulo adicional à biblioteca de otimização multi-objetivo *pymoo*, no contexto das pesquisas do ramo que estudam propostas de parametrização dinâmica destas técnicas. Partindo de um *script* de uma prova de conceito, que utilizou as ferramentas disponíveis na biblioteca para realizar efetivamente a parametrização dinâmica das taxas de mutação e cruzamento do *NSGA-II*, foram aplicadas sequências de refatorações que resultaram em um *wrapper*, bem como outras classes auxiliares, que permitem a resolução de problemas e controle dinâmico de parâmetros de forma simplificada por meio da biblioteca utilizada. O resultado principal do trabalho, além da documentação deste processo de refatoração, é uma proposta de módulo útil para pesquisas e implementações que aplicam controle dinâmico à parametrização de métodos multi-objetivo.**

***Palavras-chave*—NSGA, Refatoração, Parametrização Dinâmica**

I. Introdução

Algoritmos de Otimização Multi-Objetivo (*Multi-Objective Optimization Algorithm* - MOA) são uma variação de algoritmos de otimização mais clássicos que utilizam técnicas baseadas em computação evolutiva e, alternativamente ou conjuntamente, inteligência de enxame, como *Ant Colony Optimization*, *Particle Swarm Optimization*, e Algoritmos Genéticos. Estes são três exemplos de técnicas que buscam evoluir um conjunto de soluções candidatas de forma iterativa, buscando soluções que se aproximem do ótimo global ou mínimo global de uma única função objetivo [1].

Estas técnicas são importantes para diversos domínios teóricos e práticos. Sua aplicabilidade se deve ao fato de que, para grande parte das funções objetivo de interesse, derivar ótimos globais de forma analítica ou exata nem sempre é viável ou possível, além da característica estocástica e amplamente multi-dimensional de alguns domínios de aplicação. Neste sentido, estas técnicas, que são aproximativas, representam alternativas viáveis que frequentemente obtém soluções "boas o bastante" se parametrizados da forma correta, tarefa complexa que depende, na maioria das vezes, de especialistas em projetos de algoritmos de otimização e especialistas de domínio.

Tais métodos de otimização surgiram originalmente para otimizar uma única função objetivo. Entretanto, com o desenvolvimento desta área de pesquisa, observou-se a necessidade de otimizar mais de uma função ao mesmo tempo para a mesma entrada, para que diversos aspectos do domínio de aplicação pudessem ser considerados durante uma tarefa de otimização. Mais detalhes a respeito desta variação serão discutidos na Seção II, já que esta classe de métodos, chamados de Algoritmos de Otimização Multi-Objetivo constituem parte do objeto de estudos deste relato de caso.

Diante desse contexto, este artigo apresenta um relato de caso sobre a criação incremental de uma estrutura modular, voltada à experimentação com estratégias de parametrização dinâmica

em algoritmos de otimização multiobjetivo. O desenvolvimento foi realizado com base em sucessivas refatorações sobre um código inicial funcional, porém monolítico, utilizando como ponto de partida um exemplo disponível na biblioteca *pymoo* [2].

Este processo, oriundo de livre exploração de técnicas simples de refatoração e motivado pela necessidade de dois dos autores de realizar experimentos com parametrização dinâmica de MOA's, foi executado com a finalidade principal de suprir a falta de uma ferramenta que permitisse e disponibilizasse uma forma prática e reutilizável de alterar parâmetros de MOA's já codificados e testados previamente. Nesse sentido, não buscou-se um estudo de convergência ou de eficácia dos métodos, nem um estudo comparativo entre diferentes técnicas, mas sim a criação de um módulo que, em pesquisas futuras, possa ser reutilizado no contexto de pesquisas neste campo.

Esta pesquisa também tem como contexto geral, dessa forma, a discussão de reprodutibilidade, replicabilidade e reuso de código em aplicações científicas [3], questão pertinente que vem levantando preocupações sobre uma crise de reprodutibilidade em experimentos publicados que envolvem código disponibilizado [4].

O artigo está estruturado como segue: na Seção II estão citadas algumas definições acerca de métodos de Otimização Multi-Objetivo e está introduzido o problema da parametrização dinâmica destes métodos, bem como alguns aspectos sobre o *framework* alvo do processo de refatoração/criação de um novo módulo; a Seção III apresenta os materiais e métodos utilizados para a realização do estudo de caso, incluindo a concepção do processo de refatoração e as técnicas adotadas; na Seção IV está documentado todo o processo de refatoração, bem como estão explicitados os códigos intermediários produzidos. Por fim, na Seção V, estão listados os principais aprendizados da refatoração e o processo é avaliado como um todo e com relação ao produto de código final.

II. Parametrização Dinâmica de MOAs

Algoritmos de Otimização Multi-Objetivo (MOAs) são métodos utilizados para resolver problemas nos quais múltiplas funções objetivo devem ser otimizadas simultaneamente. Diferentemente da otimização tradicional e das abordagens envolvendo somas ponderadas, que buscam um único ótimo, os MOAs produzem um conjunto de soluções não-dominadas, conhecidas como fronteira de Pareto, representando diferentes *trade-offs* otimizados entre os objetivos [5]. Esses algoritmos são amplamente aplicados em domínios nos quais há múltiplos critérios conflitantes, como nas engenharias de forma geral.

Dentre os MOAs mais conhecidos estão o NSGA-II (*Non-dominated Sorting Genetic Algorithm II*), SPEA2 (*Strength Pareto Evolutionary Algorithm 2*) e MOEA/D (*Multiobjective Evolutionary Algorithm Based on Decomposition*), inspirados em métodos evolutivos. Em geral, esses algoritmos requerem a definição de diversos metaparâmetros, como taxas de mutação e cruzamento e tamanho de população, em algumas abordagens, que influenciam diretamente no tempo de convergência para boas soluções e na qualidade dos Fronts de Pareto obtidos [5].

Dessa forma, ao longo dos anos, surgiram diversas propostas de parametrização dinâmica em pesquisas da área, que visam ajustar tais valores (também aplicáveis a algoritmos que utilizam inteligência de enxame) ao longo da execução do algoritmo utilizando técnicas diversas [6]. O presente trabalho tem como contexto a codificação desta abordagem por meio da biblioteca *pymoo*, explorando aspectos de Engenharia de Software que possibilitam generalizar esse tipo de estratégia por meio da ferramenta em um processo simples de refatoração.

A. Pymoo framework

A biblioteca *pymoo* vem ganhando popularidade nos últimos anos, com amplo uso dentro da área acadêmica, mesmo se tratando de um *software* relativamente novo [2] . Trata-se de um *framework* amplo e generalista para Otimização Multi-Objetivo, abrangendo uma ampla gama de MOAs e suas variações. É escrita principalmente em *python*, utilizando bibliotecas bem consolidadas da linguagem e recursos que permitem a execução paralela em alguns casos, embora o foco principal da ferramenta não seja a alta performance.

Além de fornecer diversos algoritmos diferentes, a *pymoo* também fornece implementações de variadas métricas, funções de *benchmark*, opções de customização, operadores e visualização/logging relacionados à otimização multi-objetivo. Vale ressaltar também que, além de uma interface funcional que permite prototipação rápida, a biblioteca é construída sobre uma estrutura orientada a objetos robusta e bem-definida, extensível e de fácil compreensão para desenvolvedores de novos módulos.

Este aspecto de capacidade de interação simplificada com o *framework* em níveis mais baixos e mais altos em termos de interface é o que possibilitou a realização deste processo de refatoração e o que motivou, entre outros aspectos técnicos de estudos comparativos [7], a escolha da ferramenta para integrar este estudo de caso. As atualizações constantes no repositório principal do *pymoo*, na época de realização deste trabalho, também foram determinantes para sua seleção.

III. Metodologia

Como primeiro passo, para validar a possibilidade de parametrizar MOAs utilizando *pymoo*, foi realizada uma prova de conceito, modificando parâmetros dinamicamente através de um exemplo da documentação que manipula o *framework* em níveis mais baixos de abstração. Essa prova de conceito serviu

como entrada para o processo de refatoração, que utilizou de forma geral alguns dos itens dos catálogos bem conhecidos e praticados nos dias atuais, notadamente presentes em [8].

O MOA a ser utilizado foi o NSGA-II, e os parâmetros que foram controlados, para fins de simplificação e "abreviação" do código, foram a taxa de mutação e taxa de cruzamento. Estas escolhas *a priori* não devem incorrer em uma perda de generalidade, pois o produto final da sequência de refatorações deve ser diretamente, ou pelo menos de uma forma mais facilitada, aplicável a outros conjuntos de parâmetros do NSGA-II, outros métodos de otimização com seus respectivos parâmetros e outras estratégias de parametrização dinâmica, a partir das possibilidades da biblioteca.

Nas primeiras fases da refatoração, o foco das iterações foi sobretudo funcional, buscando compreender e separar aspectos comportamentais e de responsabilidade. Nas etapas finais, esperava-se que fosse possível derivar uma arquitetura orientada a objetos para o módulo proposto que, embora simples, fosse reutilizável e extensível em outras pesquisas da área. Cada código resultante foi testado com um *hash* simples do *front* de Pareto encontrado, para garantir que os códigos, apesar de refatorados, produziriam comportamentos idênticos à partir das mesmas *seeds* randômicas, garantindo a equivalência funcional.

Durante o processo, foram utilizados os conceitos de *code smells* [9] e alguns dos padrões de refatorações já existentes. Não foi o objetivo do processo aqui descrito, entretanto, abordar estas duas temáticas em profundidade ou de aplicar diferentes refatoração a título de comparação. A proposta aqui apresentada é a de aplicação de uma sequência de passos simples de refatoração, definidos de maneira exploratória e *ad-hoc* pelos autores, para a geração de uma estrutura reutilizável que, no contexto de aplicações científicas, pudessem ser úteis a mais de um trabalho de pesquisa.

IV. Processo de Refatoração

Nesta seção está documentado o processo de criação da nova funcionalidade utilizando a metodologia descrita na Seção III. A cada passo serão descritos em linhas gerais os motivos que levaram às refatorações, e o código resultante será mostrado e comentado brevemente. Por razões relacionadas ao leiaute do artigo e à legibilidade dos exemplos de código, alguns trechos serão abreviados ou omitidos com reticências e/ou comentários, sem perda de generalidade, buscando também evidenciar apenas as mudanças pertinentes às refatorações específicas a que a implementação se refere. O código completo pode ser solicitado via *e-mail* ao autor ou acessado através do DOI [10].

No Código 1, são utilizadas algumas das interfaces, classes e métodos padrão da biblioteca *pymoo*. Das linhas 1 a 7 é instanciado um problema sem restrições, onde o espaço de *design* contém 30 variáveis de decisão que variam de 0 a 1. Das linhas 8 a 9 são instanciados operadores de mutação e cruzamento, que são utilizados para instanciar um algoritmo NSGA-II, a partir da linha 10, com 100 indivíduos em sua população. No restante do código, o algoritmo é controlado manualmente, modificando randomicamente os operadores de mutação e cruzamento a cada iteração. O critério de parada adotado no exemplo, sem perda de generalidade, foi o número de iterações.

A. Versão 0 - Prova de Conceito

Nesta primeira versão, buscou-se utilizar a biblioteca *pymoo*, a partir de um exemplo da documentação oficial, para realizar a alteração dinâmica dos parâmetros de mutação e cruzamento do algoritmo NSGA-II. Pode-se observar, no Código 1, que das linhas 22 a 25 foram atribuídos valores aleatórios a estas variáveis durante as dez iterações definidas pelo laço iniciado na linha 21. Essa estratégia permitiu simular, de forma controlada, a mudança de parâmetros ao longo do tempo, o que é central no conceito de parametrização dinâmica.

Através de operadores de comparação de instância e técnicas de *debugging*, foi possível verificar que, de fato, os valores de mutação e cruzamento internos do objeto referente ao NSGA-II foram modificados de forma randômica a cada geração, confirmando que a manipulação direta dos atributos associados aos operadores teve efeito prático durante a execução. Essa verificação foi importante para assegurar que não havia problemas relacionados a cópias superficiais, referências inválidas ou reatribuições incorretas, o que poderia comprometer a validade da prova de conceito.

Embora esta versão seja funcional e tenha servido de base para as etapas seguintes de refatoração, sua estrutura monolítica e a ausência de uma separação clara entre as diferentes responsabilidades do código limitam sua manutenção, extensão e legibilidade. Ainda assim, trata-se de uma etapa fundamental do processo, pois mostra empiricamente que o *framework* permite, mesmo sem modificações significativas, a implementação de comportamentos de parametrização dinâmica por meio das classes já existentes.

B. Versão 1 - Separação Funcional de Responsabilidades

Nesta segunda etapa, cujo produto é o Código 2, buscou-se realizar uma primeira tentativa de organização mais clara das responsabilidades envolvidas na execução do algoritmo. Para isso, o código foi reorganizado por meio da extração de blocos de lógica em funções nomeadas, o que permite tornar explícita a separação entre a configuração do problema, a criação do algoritmo, a geração de objetivos e a modificação dinâmica dos parâmetros. Buscou-se, neste passo, uma estrutura de código mais compreensível, onde é possível localizar e alterar partes da implementação de maneira mais simplificada.

Código 1: Versão 0 - Prova de conceito de parametrização dinâmica no *pymoo*

```
    problem = Problem(
        n_var=30,
        n_obj=2,
        n_constr=0,
        xl=np.zeros(30),
        xu=np.ones(30)
    )
    mutation = PolynomialMutation(prob=1.0, eta=20)
    crossover = SBX(prob=1.0, eta=15)
    algorithm = NSGA2(
        pop_size=100,
        sampling=FloatRandomSampling(),
        mutation=mutation,
        crossover=crossover,
        eliminate_duplicates=True
    )
    termination = NoTermination()
    algorithm.setup(problem, termination=termination)
    np.random.seed(1)
    evaluator = Evaluator()
    for n_gen in range(10):
        new_mutation_prob = np.random.uniform(0.05, 1.0)
        new_crossover_prob = np.random.uniform(0.05, 1.0)
        mutation.prob = new_mutation_prob
        crossover.prob = new_crossover_prob
        pop = algorithm.ask()
        X = pop.get("X")
        f1 = X[:, 0]
        g = 1 + 9.0 / (problem.n_var - 1) * np.sum(X[:, 1:], axis=1)
        f2 = g * (1 - np.sqrt(f1 / g))
        F = np.column_stack([f1, f2])
        static = StaticProblem(problem, F=F)
        evaluator.eval(static, pop)
        algorithm.tell(pop)
```

A principal modificação implementada foi a aplicação da refatoração *Extract Function*, resultando em funções como *setup_problem*, *setup_algorithm*, *update_mutation_prob* e *update_crossover_prob*. Essa divisão permitiu encapsular comportamentos recorrentes ou semanticamente distintos, facilitando a leitura do corpo principal do programa. Um dos *code smells* inicialmente presentes, do tipo *Long Function*, especialmente perceptível no loop principal da versão anterior, foi parcialmente mitigado com essa reorganização.

Ainda que esta organização inicial não resolva todos os problemas estruturais presentes, ela já representa uma melhoria em relação à versão anterior. A modularização torna mais evidente onde ocorrem as alterações dinâmicas nos parâmetros, e viabiliza, nas próximas iterações, a substituição destas funções por estratégias mais genéricas ou unificadas.

C. Versão 2 - Isolamento da Parametrização Dinâmica

Na terceira versão do código, deu-se continuidade à organização modular proposta anteriormente, com foco mais específico no isolamento da lógica de parametrização dinâmica. O objetivo principal dessa iteração foi tornar mais clara e centralizada a responsabilidade pela atualização dos parâmetros do algoritmo, reunindo todas as alterações em uma única função que recebe o estado atual da execução. Essa modificação permite que, independentemente da estratégia adotada, todo o controle dinâmico seja descrito e realizado de maneira concentrada, facilitando tanto a leitura quanto a futura substituição da lógica por abordagens mais complexas a partir de uma função de atualização. O resultado desta reorganização pode ser visualizado no Código 3.

Além da continuação da aplicação da refatoração *Extract Function*, a parametrização dos MOA's passa a ser um compo-

Código 2: Versão 1 - Separação Funcional de Responsabilidades

```
    def setup_problem():
        return Problem(...)
    def setup_algorithm(problem, mutation, crossover):
        algorithm = NSGA2(...)
        algorithm.setup(problem, termination=NoTermination())
        return algorithm
    def update_mutation_prob(n_gen, algorithm, problem, evaluator, mutation, crossover):
        mutation.prob = np.random.uniform(0.05, 1.0)
    def update_crossover_prob(n_gen, algorithm, problem, evaluator, mutation, crossover):
        crossover.prob = np.random.uniform(0.05, 1.0)
    def generate_objectives(X):
        #returns array of function evaluations
    def main():
        np.random.seed(1)
        problem = setup_problem()
        mutation = PolynomialMutation(prob=1.0, eta=20)
        crossover = SBX(prob=1.0, eta=15)
        algorithm = setup_algorithm(problem, mutation, crossover)
        evaluator = Evaluator()
        for n_gen in range(10):
            update_mutation_prob(n_gen, algorithm, problem, evaluator, mutation, crossover)
            update_crossover_prob(n_gen, algorithm, problem, evaluator, mutation, crossover)
            pop = algorithm.ask()
            X = pop.get("X")
            F = generate_objectives(X)
            static = StaticProblem(problem, F=F)
            evaluator.eval(static, pop)
            algorithm.tell(pop)
    if __name__ == "__main__":
        main()
```

nente isolado, o que abre espaço para a integração de controladores externos, como modelos de aprendizado ou heurísticas programáveis.

A eliminação das funções separadas para cada tipo de operador (como na versão anterior) também contribui para resolver parcialmente o *code smell* do tipo *Duplicated Code*, já que a nova abordagem unifica a modificação dos parâmetros em um único ponto de entrada. Essa versão representa, portanto, o início da abstração da parametrização como entidade lógica própria, ainda que sem encapsulamento em objetos. A partir daqui, já seria possível modificar a implementação para que quaisquer parâmetros do métodos pudessem ser atualizados.

D. Versão 3 - Identificação e Criação de classe Controlled Algorithm

Nesta versão, foi identificada uma recorrência estrutural que justificava a criação de uma nova abstração de mais alto nível: uma classe que encapsulasse o comportamento de um algoritmo de otimização acoplado a um mecanismo de parametrização dinâmica. A partir dessa observação, foi definida a classe *ControlledAlgorithm*, cujo objetivo é centralizar a execução do algoritmo, o controle de parâmetros e a avaliação dos indivíduos. Essa abstração permite tornar explícito, a nível de arquitetura, que o processo de parametrização dinâmica é parte integrante do ciclo do MOA. O resultado desta etapa pode ser visualizado no Código 4.

Com a introdução dessa classe, aplica-se uma combinação de refatorações, em especial *Introduce Parameter Object*, ao empacotar o estado da execução em um dicionário passado às funções auxiliares, e *Extract Class*, ao deslocar a responsabilidade da execução do laço principal para dentro de um objeto com métodos bem definidos (*step* e *run*). Essa modificação resolve diretamente o *code smell* do tipo *Feature Envy*, que anteriormente aparecia na função de atualização de parâmetros, fortemente acoplada ao objeto *algorithm* sem estar encapsulada em um contexto que justificasse seu acesso direto a atributos internos.

Além da melhora na organização, essa estrutura oferece um caminho para extensões futuras, como a integração de

Código 3: Versão 2 - Isolamento da Parametrização Dinâmica

```
    def setup_problem():
        return Problem(...)
    def setup_algorithm(problem, mutation, crossover):
        algorithm = NSGA2(...)
        algorithm.setup(problem, termination=NoTermination())
        return algorithm
    def update_dynamic_parameters(state):
        alg = state["algorithm"]
        alg.mating.mutation.prob = np.random.uniform(0.05, 1.0)
        alg.mating.crossover.prob = np.random.uniform(0.05, 1.0)
    def generate_objectives(X):
        ...
    def main():
        np.random.seed(1)
        problem = setup_problem()
        mutation = PolynomialMutation(prob=1.0, eta=20)
        crossover = SBX(prob=1.0, eta=15)
        algorithm = setup_algorithm(problem, mutation, crossover)
        evaluator = Evaluator()
        for n_gen in range(10):
            state = {
                "algorithm": algorithm,
                "problem": problem,
                "iteration": n_gen
            }
            update_dynamic_parameters(state)
            pop = algorithm.ask()
            X = pop.get("X")
            F = generate_objectives(X)
            static = StaticProblem(problem, F=F)
            evaluator.eval(static, pop)
            algorithm.tell(pop)
    if __name__ == "__main__":
        main()
```

estratégias de controle baseadas em aprendizado, mecanismos de *logging* (muito relevantes em aplicações científicas), análise *online* ou mesmo adaptação de múltiplos algoritmos com comportamentos distintos. A partir deste ponto, a parametrização dinâmica passa a ser um elemento de projeto, controlado e extensível, mais alinhado aos princípios de coesão e encapsulamento.

E. Versão 4 - Classes Parameter Strategy e Objective Function

Na quarta versão do código, o objetivo foi explicitar e isolar dois papéis conceituais que vinham sendo tratados apenas como funções auxiliares: a lógica de atualização dos parâmetros e o cálculo da função objetivo. Para isso, foram introduzidas as classes abstratas *ParameterStrategy* e *ObjectiveFunction*, com suas respectivas implementações concretas, *RandomProbabilityStrategy* e *ZDT1Objective*. Essa mudança organiza o código em torno de contratos bem definidos, facilitando a substituição ou extensão de comportamentos sem a necessidade de alterar a estrutura principal do sistema. O código correspondente pode ser observado no Código 5. Essa reorganização envolveu a aplicação da refatoração *Extract Class*, ao mover para classes separadas as responsabilidades de atualização de parâmetros e avaliação da função objetivo, anteriormente concentradas na classe *ControlledAlgorithm*. Além disso, essa versão adota de forma explícita o padrão de projeto *Strategy*, permitindo que diferentes comportamentos sejam encapsulados em objetos substituíveis, sem impactar a lógica de execução do controlador. Entre os *code smells* mitigados, destaca-se o *Shotgun Surgery*, pois a modificação de um aspecto específico da lógica — como a troca da função objetivo — agora pode ser feita localmente, sem impacto em outras partes do sistema. A estrutura de classes resultante pode ser observada no diagrama da Figura 1.

Código 4: Versão 3 - Identificação e Criação de classe *Controlled Algorithm*

```
    class ControlledAlgorithm:
        def __init__(self, algorithm, problem, update_fn, evaluate_fn):
            # assign constructor parameters to class attributes
            self.algorithm.setup(problem, termination=NoTermination())
        def step(self):
            state = {
                "algorithm": self.algorithm,
                "problem": self.problem,
                "iteration": self.n_gen
            }
            self.update_fn(state)
            pop = self.algorithm.ask()
            X = pop.get("X")
            F = self.evaluate_fn(X, state)
            static = StaticProblem(self.problem, F=F)
            self.evaluator.eval(static, pop)
            self.algorithm.tell(pop)
            self.n_gen += 1
        def run(self, n_generations):
            for _ in range(n_generations):
                self.step()
    def update_parameters(state):
        ...
    def evaluate_zdt1(X, state):
        ...
    def main():
        np.random.seed(1)
        problem = Problem(...)
        mutation = PolynomialMutation(prob=1.0, eta=20)
        crossover = SBX(prob=1.0, eta=15)
        algorithm = NSGA2(...)
        controlled = ControlledAlgorithm(
            algorithm=algorithm,
            problem=problem,
            update_fn=update_parameters,
            evaluate_fn=evaluate_zdt1
        )
        controlled.run(n_generations=10)
    if __name__ == "__main__":
        main()
```

V. Resultados e Conclusões

O processo descrito ao longo deste relato teve como resultado principal a construção incremental de uma estrutura modular para suporte à parametrização dinâmica de algoritmos de otimização multiobjetivo utilizando a biblioteca *pymoo*. A partir de uma prova de conceito funcional, mas não sistematizada, foram aplicadas sucessivas refatorações visando tornar o código mais legível, reutilizável e extensível.

O produto final de todas estas etapas, embora simples, permite que diferentes estratégias de parametrização e funções objetivo sejam facilmente integradas ao fluxo de execução dos algoritmos disponíveis na biblioteca, com um esforço mínimo de adaptação. Um diagrama de classes simplificado, representando a estrutura criada, pode ser observado na Figura 1. Espera-se que, com um pouco mais de modificações e testes que podem ser necessários, bem como possíveis adaptações de acordo com as diretrizes dos mantenedores do repositório oficial, o código gerado neste trabalho possa ser integrado ao *framework pymoo*.

Durante esse processo, observou-se que, mesmo em contextos de pesquisa científica, nos quais *scripts* são frequentemente desenvolvidos com foco exclusivo na obtenção de resultados,

Código 5: Versão 4 - Classes *Parameter Strategy* e *Objective Function*

```
    class ParameterStrategy:
        def update(self, state: dict):
            raise NotImplementedError
    class ObjectiveFunction:
        def evaluate(self, X: np.ndarray, state: dict) -> np.ndarray:
            raise NotImplementedError
    class RandomProbabilityStrategy(ParameterStrategy):
        def __init__(self, mutation_range=(0.05, 1.0), crossover_range=(0.05, 1.0)):
            ...
        def update(self, state):
            ...
    class ZDT1Objective(ObjectiveFunction):
        def evaluate(self, X, state):
            ...
    class ControlledAlgorithm:
        def __init__(self, algorithm, problem, parameter_strategy, objective_function):
            # assign constructor parameters to class attributes
            self.algorithm.setup(problem, termination=NoTermination())
        def step(self):
            state = {
                "algorithm": self.algorithm,
                "problem": self.problem,
                "iteration": self.n_gen
            }
            self.parameter_strategy.update(state)
            pop = self.algorithm.ask()
            X = pop.get("X")
            F = self.objective_function.evaluate(X, state)
            static = StaticProblem(self.problem, F=F)
            self.evaluator.eval(static, pop)
            self.algorithm.tell(pop)
            self.n_gen += 1
        def run(self, n_generations):
            for _ in range(n_generations):
                self.step()
    def main():
        np.random.seed(1)
        problem = Problem(...)
        mutation = PolynomialMutation(prob=1.0, eta=20)
        crossover = SBX(prob=1.0, eta=15)
        algorithm = NSGA2(...)
        parameter_strategy = RandomProbabilityStrategy()
        objective_function = ZDT1Objective()
        controller = ControlledAlgorithm(...)
        controller.run(n_generations=10)
        final_F = controller.algorithm.pop.get("F")
    if __name__ == "__main__":
        main()
```

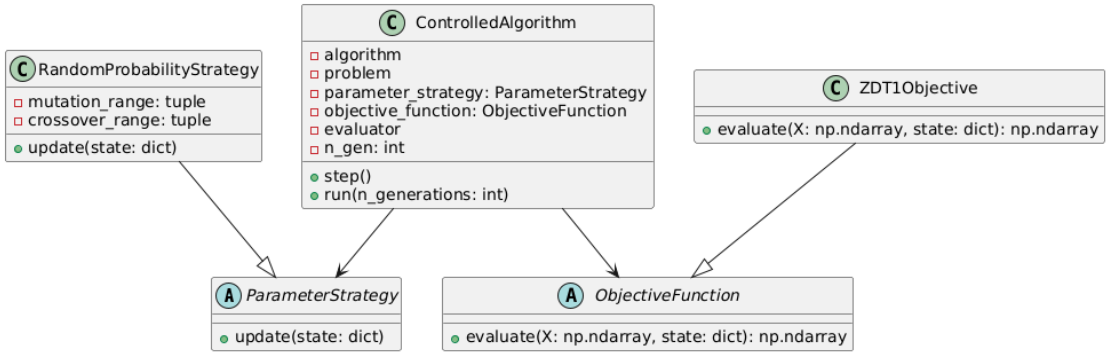

Fig. 1: Diagrama de Classes Simplificado

há benefícios claros na adoção de práticas de engenharia de software. Refatorações como *Extract Function*, *Extract Class* e o uso do padrão de projeto *Strategy* contribuíram para tornar o código mais robusto e menos suscetível a erros ao longo de iterações futuras. Além disso, a modularização permite que pesquisadores com diferentes perfis — inclusive não especialistas em computação — possam explorar, modificar e testar estratégias de controle dinâmico com maior facilidade em outros domínios do conhecimento.

Embora a estrutura do módulo proposto ainda possa ser aprimorada, especialmente no que diz respeito à generalização para múltiplos algoritmos e operadores, o modelo atual já se mostra suficiente para dar suporte a diversos experimentos envolvendo a modificação dinâmica de parâmetros, atendendo aos objetivos iniciais dos autores. Assim, o módulo desenvolvido pode servir como base para extensões posteriores ou como ponto de partida para outras propostas no contexto de pesquisa em otimização multiobjetivo.

Agradecimentos

Os autores agradecem ao Programa de Pós-Graduação em Ciência da Computação da Universidade Estadual do Oeste do Paraná (UNIOESTE) pelo apoio institucional, bem como à Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) por meio da bolsa de Mestrado concedida ao primeiro autor.

Referências

[1] K. Rajwar, K. Deep, and S. Das, "An exhaustive review of the metaheuristic algorithms for search and optimization: taxonomy, applications, and open challenges," *Artificial Intelligence Review*, vol. 56, no. 11, p. 13187–13257, Apr. 2023. [Online]. Available: http://dx.doi.org/10.1007/s10462-023-10470-y

[2] J. Blank and K. Deb, "pymoo: Multi-objective optimization in python," *IEEE Access*, vol. 8, pp. 89 497–89 509, 2020. [Online]. Available: https://doi.org/10.1109/ACCESS.2020.2990567

[3] F. C. Y. Benureau and N. P. Rougier, "Re-run, repeat, reproduce, reuse, replicate: Transforming code into scientific contributions," *Frontiers in Neuroinformatics*, vol. 11, Jan. 2018. [Online]. Available: http://dx.doi.org/10.3389/fninf.2017.00069

[4] A. Trisovic, M. K. Lau, T. Pasquier, and M. Crosas, "A large-scale study on research code quality and execution," *Scientific Data*, vol. 9, no. 1, Feb. 2022. [Online]. Available: http://dx.doi.org/10.1038/s41597-022-01143-6

[5] J. Li and E. Sopov, "A comparative study of state-of-the-art multi-objective optimization algorithms," *ITM Web of Conferences*, vol. 59, p. 02024, 2024. [Online]. Available: http://dx.doi.org/10.1051/itmconf/20245902024

[6] M. Gomes Pereira de Lacerda, L. F. de Araujo Pessoa, F. Buarque de Lima Neto, T. B. Ludermir, and H. Kuchen, "A systematic literature review on general parameter control for evolutionary and swarm-based algorithms," *Swarm and Evolutionary Computation*, vol. 60, p. 100777, Feb. 2021. [Online]. Available: http://dx.doi.org/10.1016/j.swevo.2020.100777

[7] A. Pătrăușanu, A. Florea, M. Neghină, A. Dicoiu, and R. Chiș, "A systematic review of multi-objective evolutionary algorithms optimization frameworks," *Processes*, vol. 12, no. 5, p. 869, Apr. 2024. [Online]. Available: http://dx.doi.org/10.3390/pr12050869

[8] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[9] M. Jerzyk and L. Madeyski, "Code smells: A comprehensive online catalog and taxonomy," in *Developments in Information and Knowledge Management Systems for Business Applications: Volume 7*. Springer, 2023, pp. 543–576. [Online]. Available: https://doi.org/10.1007/978-3-031-25695-0_24

[10] D. Marchetti, "dmgiacomel/dynamic-parameterization-on-pymoo: v1.0." Oct. 2025. [Online]. Available: https://doi.org/10.5281/zenodo.17316684